**COLLECTİON TYPES**

**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).

**Tanıtım**

Python'da çeşitli koleksiyon türleri vardır. Int ve str gibi türler tek bir değer tutarken, koleksiyon türleri birden çok değer tutar.  
  
Programlarınızda, genellikle tek bir nesne olarak işlemek için birkaç öğeyi gruplandırmanız gerekir. Bu işi yapmak için veri toplama türlerini kullanıyoruz.  
  
Python'daki en kullanışlı koleksiyonlardan biri listedir. Python'da bir liste, yalnızca geçerli Python değerlerinin sıralı bir koleksiyonudur.  
  
Liste türü muhtemelen Python'da en sık kullanılan koleksiyon türüdür. Adına rağmen, bir liste daha çok diğer bazı dillerde (örneğin JavaScript) bir dizi gibidir.

**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 Zealand']

**Scratch Time ! :**Create this list with **[scratch](https://scratch.mit.edu/projects/341599722/editor/" \t "_blank)**.

**💡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.

**Liste Oluşturma**

Değerleri virgülle ayırarak köşeli parantez 👉🏻[] içine alarak bir liste oluşturulabilir.  
  
Bazı ülke adlarını içeren basit bir liste oluşturalım.  
  
Ülke = ['ABD', 'Brezilya', 'İngiltere', 'Almanya', 'Türkiye', 'Yeni Zelanda']  
Bu, bu kurstaki ilk listemiz. Şimdi listeyi yazdıralım.  
  
Giriş:  
  
Ülke = ['ABD', 'Brezilya', 'İngiltere', 'Almanya', 'Türkiye', 'Yeni Zelanda']  
  
Baskı(ülke)  
Çıktı :  
  
['ABD', 'Brezilya', 'İngiltere', 'Almanya', 'Türkiye', 'Yeni Zelanda']  
Kazıma Zamanı! : Bu listeyi sıfırdan oluşturun.  
  
  
  
💡İpuçları:  
Listeler sıralı olduğu için tüm ülke adları listede saklandıkları sırayla yazdırılır.  
Liste oluşturmanın başka bir yolu da 'list()' işlevini çağırmaktır.  
  
Bunu, yinelenebilir bir nesneden bir liste oluşturmak istediğinizde yaparsınız: yani öğelerini tek tek içe aktarabileceğiniz nesne türü. Listeler, diğer koleksiyonlar ve dize türleri gibi yinelenebilir. List() işlevini kullanarak başka bir liste oluşturalım ve 👉🏻'[]' ile karşılaştıralım.  
  
Giriş:  
  
String\_1 = 'Sigarayı bıraktım'  
  
New\_list\_1 = list(string\_1) # çoklu eleman listesi oluşturduk  
Yazdır(new\_list\_1)  
  
New\_list\_2 = [string\_1] # bu tek elemanlı bir listedir  
Yazdır(new\_list\_2)  
Çıktı :  
  
['Ben', ' ', 'q', 'u', 'ben', 't', ' ', 's', 'm', 'o', 'k', 'i', 'n' , 'G']  
['Sigara içmeyi bıraktım']  
💡İpuçları:  
List() işlevi kullanılarak, boşluklar dahil olmak üzere string\_1'in tüm karakterlerinin bir new\_list\_1'e taşındığını unutmayın.  
Fark ettiyseniz, listeler aynı değerden birden fazlasını içerebilir.  
Göründüğü gibi, list() işlevi, dize gibi belirli bir yinelenebilir nesnenin her bileşenini içeren bir liste oluşturur. Ne yapacağınıza bağlı olarak köşeli parantez veya list() işlevlerini kullanabilirsiniz.  
  
Python'un dinamik bir dil olduğu göz önüne alındığında, bir listenin bileşenleri tek bir veri türüyle sınırlı değildir: ör.  
  
Karışık\_liste = [11, 'Joseph', Yanlış, 3.14, Yok, [1, 2, 3]]  
  
💡İpuçları:  
Yukarıda gördüğünüz gibi, bir veya daha fazla liste öğesi bir liste bile olabilir.

**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', 'Sydney']

city.append('Addis Ababa')

print(city)

output :

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

| **append** |
| --- |
| *Diagram of '.append( )' Method* |

* **.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', 'Sydney', 'Addis Ababa']

city.insert(2, 'Stockholm')

print(city)

output :

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

| **Tiobe_Index** |
| --- |
| *Diagram of '.insert( )' Method* |

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', 'Sydney', 'Addis Ababa']

city.sort() # lists the items in alphabetical order

print(city)

output :

['Addis Ababa', 'Istanbul', 'New York', 'Seoul', 'Stockholm', '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

**Listelerle Temel İşlemler**

Python'da liste yapılarıyla başa çıkmak için birçok yöntem ve işlev vardır. Temel ve en yaygın olanlardan bazılarını öğreneceksiniz. Hadi başlayalım:  
  
Çoğu durumda, daha sonra istediğiniz verilerle doldurmak için boş bir liste oluşturmamız gerekecek.  
  
Boş\_list\_1= []  
  
Boş\_list\_2 = liste()  
.append() veya .insert() yöntemlerini kullanarak bir listeye eleman ekleyebiliriz.  
  
.append() : Listenin sonuna bir nesne ekler. Yalnızca list.append(element) sözdizimi kullanıldığında hiçbiri döndürülür. Eklenen yeni listeyi görmek istiyorsanız, aramanız veya yazdırmanız gerekir. Örneğe bakın:  
Giriş:  
  
Boş\_list\_1 = []  
Boş\_list\_1.append('114')  
Empty\_list\_1.append('plastiksiz deniz')  
  
Yazdır(empty\_list\_1)  
Çıktı :  
  
['114', 'plastiksiz deniz']  
Giriş:  
  
Şehir = ['New York', 'Londra', 'İstanbul', 'Seul', 'Sidney']  
City.append('Addis Ababa')  
  
Yazdır(şehir)  
Çıktı :  
  
['New York', 'Londra', 'İstanbul', 'Seul', 'Sidney', 'Addis Ababa']  
  
'.append()' Yönteminin Şeması  
.insert() : Belirli bir dizinde listelenecek yeni bir nesne ekleyin. Sözdizimi list.insert(index, object) gibi görünür. Örneğe bakın:  
Giriş:  
  
Şehir = ['New York', 'Londra', 'İstanbul', 'Seul', 'Sidney', 'Addis Ababa']  
City.insert(2, 'Stockholm')  
  
Yazdır(şehir)  
Çıktı :  
  
['New York', 'Londra', 'Stockholm', 'İstanbul', 'Seul', 'Sidney', 'Addis Ababa']  
  
'.insert()' Yönteminin Şeması  
List.remove() yöntemini kullanarak listelerdeki öğeleri kaldırabilir veya list.sort() yöntemini kullanarak öğeleri sıralayabiliriz. Örneği inceleyin:  
  
Giriş:  
  
Şehir = ['New York', 'Londra', 'Stockholm', 'İstanbul', 'Seul', 'Sidney', 'Addis Ababa']  
City.remove('Londra')  
Print(city) # 'Londra'yı sildik  
Çıktı :  
  
['New York', 'Stockholm', 'İstanbul', 'Seul', 'Sidney', 'Addis Ababa']  
Giriş:  
  
Şehir = ['New York', 'Stockholm', 'İstanbul', 'Seul', 'Sidney', 'Addis Ababa']  
City.sort() # öğeleri alfabetik sırayla listeler  
Yazdır(şehir)  
Çıktı :  
  
['Addis Ababa', 'İstanbul', 'New York', 'Seul', 'Stockholm', 'Sidney']  
💡İpuçları:  
Unutma! Bir listenin elemanları soldan sağa doğru sayılır ve string türlerinde olduğu gibi sıfır ile başlar.  
Aynı şekilde, liste elemanlarının uzunluğu da len() fonksiyonu ile hesaplanabilir. Elimizdeki 'city' değişkeninin uzunluğunu hesaplayalım.  
  
Giriş:  
  
Şehir = ['Addis Ababa', 'İstanbul', 'New York', 'Seul', 'Stockholm', 'Sidney']  
Print(len(şehir))  
Çıktı :  
  
6

**ACCESSİNG LİSTS**

Introduction

You know that there are several types of collections for storing data in Python, like **list, tuple, dictionary**.

Each item or element in a list, as well as every character in a string, has an index corresponding to their location. Using indexes, we can access elements within a sequence. Now, let's see how can we do that?

**Tanıtım**

Python'da veri depolamak için list, tuple, sözlük gibi çeşitli koleksiyon türleri olduğunu biliyorsunuz.  
  
Bir listedeki her öğe veya öğenin yanı sıra bir dizedeki her karakter, konumlarına karşılık gelen bir dizine sahiptir. Dizinleri kullanarak bir dizi içindeki öğelere erişebiliriz. Şimdi, bakalım bunu nasıl yapabiliriz?

### Indexing a List

If we want to access or use the elements of a list, we can do that using index numbers of the list enclosed by **square brackets**.

**⚠️Avoid ! :**

* Do not start indexing with **1**. The first index of the element of a list is **0**. We will never stop remembering that!

First, let's begin with a simple example :

input :

colors = ['red', 'purple', 'blue', 'yellow', 'green']

print(colors[2]) # If we start at zero,

# the second element will be 'blue'.

output :

blue

Now, let's learn the subject in detail through the examples :

input :

city = ['New York', 'London', 'Istanbul', 'Seoul', 'Sydney']

city\_list = []

city\_list.append(city) # we have created a nested list

print(city\_list)

output :

[['New York', 'London', 'Istanbul', 'Seoul', 'Sydney']]

**city\_list** includes only **one** element which is the city list.

**💡Tips :**

* If you notice that city\_list has double square brackets.

input :

city\_list = [['New York', 'London', 'Istanbul', 'Seoul', 'Sydney']]

print(city\_list[0]) # access to first and only element

output :

['New York', 'London', 'Istanbul', 'Seoul', 'Sydney']

'**city\_list[0]**' is a list type data. So that, we can still access its elements via indexing. Let's access its second element :

input :

city\_list = [['New York', 'London', 'Istanbul', 'Seoul', 'Sydney']]

print(city\_list[0][2])

output :

Istanbul

'**city\_list[0][2]**' is a string type data. So, we can also access its elements via indexing. Let's access its third element :

input :

city\_list = [['New York', 'London', 'Istanbul', 'Seoul', 'Sydney']]

print(city\_list[0][2][3])

output :

a

**Liste İndeksleme**

Bir listenin öğelerine erişmek veya bunları kullanmak istiyorsak, bunu listenin köşeli parantez içindeki dizin numaralarını kullanarak yapabiliriz.  
  
⚠️ Kaçının ! :  
İndekslemeye 1 ile başlamayın. Bir liste elemanının ilk indeksi 0'dır. Bunu hatırlamaktan asla vazgeçmeyeceğiz!  
Önce basit bir örnekle başlayalım:  
  
Giriş:  
  
Renkler = ['kırmızı', 'mor', 'mavi', 'sarı', 'yeşil']  
Print(colors[2]) # Sıfırdan başlarsak,  
# ikinci eleman 'mavi' olacaktır.  
Çıktı :  
  
Mavi  
Şimdi örnekler üzerinden konuyu detaylı bir şekilde öğrenelim:  
  
Giriş:  
  
Şehir = ['New York', 'Londra', 'İstanbul', 'Seul', 'Sidney']  
  
City\_list = []  
City\_list.append(city) # yuvalanmış bir liste oluşturduk  
  
Yazdır(şehir\_listesi)  
Çıktı :  
  
[['New York', 'Londra', 'İstanbul', 'Seul', 'Sidney']]  
City\_list, şehir listesi olan yalnızca bir öğe içerir.  
  
💡İpuçları:  
City\_list'in çift köşeli parantez olduğunu fark ederseniz.  
Giriş:  
  
City\_list = [['New York', 'Londra', 'İstanbul', 'Seul', 'Sidney']]  
Print(city\_list[0]) # ilk ve tek öğeye erişim  
Çıktı :  
  
['New York', 'Londra', 'İstanbul', 'Seul', 'Sidney']  
'city\_list[0]' bir liste tipi veridir. Böylece öğelerine indeksleme yoluyla erişmeye devam edebiliriz. İkinci elemanına erişelim:  
  
Giriş:  
  
City\_list = [['New York', 'Londra', 'İstanbul', 'Seul', 'Sidney']]  
Print(city\_list[0][2])  
Çıktı :  
  
İstanbul  
'city\_list[0][2]' bir dize tipi veridir. Böylece öğelerine indeksleme yoluyla da erişebiliriz. Üçüncü elemanına erişelim:  
  
Giriş:  
  
City\_list = [['New York', 'Londra', 'İstanbul', 'Seul', 'Sidney']]  
Print(city\_list[0][2][3])  
Çıktı :  
  
A

**Slicing a List**

We can access individual elements of a list, as well as part of those items. We use index numbers again for slicing but we do it by typing it a little differently. Look at the example :

input :

numbers = [1, 3, 5, 7, 9, 11, 13, 15, 17]

print(numbers[2:5]) # we get the elements from index=2 to index=5(5 is not included)

output :

[5, 7, 9]

**💡Tips :**

* Slicing is just similar to indexing. The difference is adding **colon** or **colons** in square brackets.

In slicing, pay attention to the stop index in square brackets. We got the elements from **index=2** to **index=5** (5 is not included): It means that we got 'second', 'third' and 'forth' element of the list.

You can keep in mind the formula syntax below for slicing a sequence. From '**start**' to '**stop-1**', by '**step**'.

**The formula syntax is : sequence[start:stop:step]**

This formula produces a slice of the sequence where **start** is an index of the first element required (the element is included in the slice) and **stop** is an index of the end element (the element is not included in the slice), **step** is an interval between elements to be chosen.

Now let's apply this formula on a few examples. In this example, we will create a list of numbers from 1 to 10 using 'range()' function and select even ones:

input :

count = list(range(11))

print(count)

print(count[0:11:2])

output :

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

[0, 2, 4, 6, 8, 10]

By the way, range() function returns an object that produces a sequence of integers from start (including) to stop (excluding) by step.

**The formula syntax is : range(start, stop[, step])**

Each part of the slice has a default value, so they are **optional**. If we don't assign a value to the **start** index, it is considered to be **0**; if we don't assign a value to the **stop** index, it will be the **same as** the **length** of the sequence.

* **my\_list[:]**: returns the full copy of the sequence
* **my\_list[start:]** : returns elements from start to the end element
* **my\_list[:stop]** : returns element from the 1st element to stop-1
* **my\_list[::step]** : returns each element with a given step

Let's do some more examples to grasp it.

The following example outputs the same as the input list (animals).

input :

animals = ['elephant', 'bear', 'fox', 'wolf', 'rabbit', 'deer', 'giraffe']

print(animals[:]) # all elements of the list

output :

['elephant', 'bear', 'fox', 'wolf', 'rabbit', 'deer', 'giraffe']

The following example slices the animals starts at **index=3** to the end.

input :

animals = ['elephant', 'bear', 'fox', 'wolf', 'rabbit', 'deer', 'giraffe']

print(animals[3:])

output :

['wolf', 'rabbit', 'deer', 'giraffe']

The following example slices the animals starts at **index=0** to the **index=4**.

input :

animals = ['elephant', 'bear', 'fox', 'wolf', 'rabbit', 'deer', 'giraffe']

print(animals[:5])

output :

['elephant', 'bear', 'fox', 'wolf', 'rabbit']

And the last example slices animals starts at **index=0** to the **end** with **2 step**.

input :

animals = ['elephant', 'bear', 'fox', 'wolf', 'rabbit', 'deer', 'giraffe']

print(animals[::2])

output :

['elephant', 'fox', 'rabbit', 'giraffe']

Q: In Python what is slicing?  
A: A mechanism to select a range of items from sequence types like list, tuple, strings etc. is known as slicing.

**Liste Dilimleme**

Bir listenin tek tek öğelerine ve bu öğelerin bir kısmına erişebiliriz. Dilimleme için yine indeks numaralarını kullanıyoruz ama bunu biraz farklı yazarak yapıyoruz. Örneğe bakınız :  
  
Giriş:  
  
Sayılar = [1, 3, 5, 7, 9, 11, 13, 15, 17]  
Print(numbers[2:5]) # index=2'den index=5'e kadar olan elemanları alıyoruz(5 dahil değil)  
Çıktı :  
  
[5, 7, 9]  
💡İpuçları:  
Dilimleme, indekslemeye benzer. Aradaki fark, köşeli parantez içinde iki nokta üst üste veya iki nokta üst üste eklemektir.  
Dilimlemede köşeli parantez içindeki stop indeksine dikkat ediniz. İndex=2'den index=5'e (5 dahil değildir) elemanları aldık: Bu, listenin 'ikinci', 'üçüncü' ve 'ileri' unsurlarını aldığımız anlamına gelir.  
  
Bir diziyi dilimlemek için aşağıdaki formül söz dizimini aklınızda tutabilirsiniz. 'Başlangıç'tan 'stop-1'e, 'adım'.  
  
Formül sözdizimi şöyledir: sıra[başlangıç:durdur:adım]  
  
Bu formül, dizinin bir dilimini oluşturur; burada başlangıç, gereken ilk öğenin (eleman dilime dahil edilmiştir) bir indeksidir ve stop, bitiş elemanının bir indeksidir (eleman dilime dahil değildir), adım bir Seçilecek elemanlar arasındaki aralık.  
  
Şimdi bu formülü birkaç örnek üzerinde uygulayalım. Bu örnekte, 'range()' işlevini kullanarak 1'den 10'a kadar bir sayı listesi oluşturacağız ve çift olanları seçeceğiz:  
  
Giriş:  
  
Say = liste(aralık(11))  
Yazdır(sayım)  
  
Yazdır(sayım[0:11:2])  
Çıktı :  
  
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]  
[0, 2, 4, 6, 8, 10]  
Bu arada, range() işlevi, başlangıçtan (dahil) bitişe (hariç) kadar bir tamsayı dizisi üreten bir nesneyi adım adım döndürür.  
  
Formül sözdizimi şöyledir: aralık(başlat, durdur[, adım])  
  
Dilimin her parçasının bir varsayılan değeri vardır, bu nedenle bunlar isteğe bağlıdır. Başlangıç ​​dizinine bir değer atamazsak 0 olarak kabul edilir; Stop indeksine bir değer atamazsak dizinin uzunluğu ile aynı olacaktır.  
  
My\_list[:]: dizinin tam kopyasını döndürür  
  
My\_list[start:] : başlangıçtan bitiş öğesine kadar öğeleri döndürür  
  
My\_list[:stop] : 1. elemandan stop-1'e eleman döndürür  
  
My\_list[::step] : her öğeyi belirli bir adımla döndürür  
  
Anlamak için birkaç örnek daha yapalım.  
  
Aşağıdaki örnek, girdi listesiyle (hayvanlar) aynı çıktıyı verir.  
  
Giriş:  
  
Hayvanlar = ['fil', 'ayı', 'tilki', 'kurt', 'tavşan', 'geyik', 'zürafa']  
  
Print(animals[:]) # listenin tüm elemanları  
Çıktı :  
  
['fil', 'ayı', 'tilki', 'kurt', 'tavşan', 'geyik', 'zürafa']  
Aşağıdaki örnek, hayvanları indeks=3 ile başlayıp sonuna kadar dilimler.  
  
Giriş:  
  
Hayvanlar = ['fil', 'ayı', 'tilki', 'kurt', 'tavşan', 'geyik', 'zürafa']  
Yazdır(hayvanlar[3:])  
Çıktı :  
  
['kurt', 'tavşan', 'geyik', 'zürafa']  
Aşağıdaki örnek, hayvanları indeks=0 ile indeks=4 arasında dilimler.  
  
Giriş:  
  
Hayvanlar = ['fil', 'ayı', 'tilki', 'kurt', 'tavşan', 'geyik', 'zürafa']  
Yazdır(hayvanlar[:5])  
Çıktı :  
  
['fil', 'ayı', 'tilki', 'kurt', 'tavşan']  
Ve son örnekte hayvanlar index=0'dan başlayıp 2 adımla sona eriyor.  
  
Giriş:  
  
Hayvanlar = ['fil', 'ayı', 'tilki', 'kurt', 'tavşan', 'geyik', 'zürafa']  
Yazdır(hayvanlar[::2])  
Çıktı :  
  
['fil', 'tilki', 'tavşan', 'zürafa']  
S: Python'da dilimleme nedir?  
C: Liste, tanımlama grubu, dizeler vb. gibi dizi türlerinden bir dizi öğeyi seçme mekanizması dilimleme olarak bilinir.

**Negative Indexing & Slicing**

**Negative indexing** is the best and shortest way to reach the elements at the end of the list. The negative indexing works in reverse. We can reach the last element of a list as list\_name[-1]. See the example below :

input :

city = ['New York', 'London', 'Istanbul', 'Seoul', 'Sydney']

print(city[-4])

output :

London

| **negative_index** |
| --- |
| *Diagram of Negative Indexing* |

**Negative slicing** also works similarly, as we see in single element access. In this case, **step index** can also be negative. If the step index is negative the elements of sequence will return in **reverse order**. Let's see in examples :

input :

reef = ['swordfish', 'shark', 'whale', 'jellyfish', 'lobster', 'squid', 'octopus']

print(reef[-3:])

output :

['lobster', 'squid', 'octopus']

| **negative_index_1** |
| --- |
| *Diagram-1 of Negative Slicing* |

input :

reef = ['swordfish', 'shark', 'whale', 'jellyfish', 'lobster', 'squid', 'octopus']

print(reef[:-3])

output :

['swordfish', 'shark', 'whale', 'jellyfish']

| **negative_index_2** |
| --- |
| *Diagram-2 of Negative Slicing* |

input :

reef = ['swordfish', 'shark', 'whale', 'jellyfish', 'lobster', 'squid', 'octopus']

print(reef[::-1]) # we have produced the reverse of the list

output :

['octopus', 'squid', 'lobster', 'jellyfish', 'whale', 'shark', 'swordfish']

| **negative_index_3** |
| --- |
| *Diagram-3 of Negative Slicing* |

input :

reef = ['swordfish', 'shark', 'whale', 'jellyfish', 'lobster', 'squid', 'octopus']

print(reef[::-2])

output :

['octopus', 'lobster', 'whale', 'swordfish']

**💡Tips :**

* If you choose negative step with the start and end indexes together, those should be used accordingly, that is, the end index should be less than the start index.

input :

odd\_no = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

print(odd\_no[7:3:-1])

print(odd\_no[2:6:-1])

output :

[8, 7, 6, 5]

[]

If you eager to find **more on lists** see [**here**](https://docs.python.org/3.8/tutorial/datastructures.html#more-on-lists).

**Q**: What does list[::-1] do?  
**A**: list[::-1] is used to reverse the order of a sequence of the elements in the list.

**Negatif İndeksleme ve Dilimleme**

Negatif indeksleme, listenin sonundaki öğelere ulaşmanın en iyi ve en kısa yoludur. Negatif indeksleme tersine çalışır. Bir listenin son elemanına list\_name[-1] olarak ulaşabiliriz. Aşağıdaki örneğe bakın:  
  
Giriş:  
  
Şehir = ['New York', 'Londra', 'İstanbul', 'Seul', 'Sidney']  
Yazdır(şehir[-4])  
Çıktı :  
  
Londra  
  
Negatif İndeksleme Şeması  
Tek elemanlı erişimde gördüğümüz gibi, negatif dilimleme de benzer şekilde çalışır. Bu durumda adım indeksi de negatif olabilir. Adım indeksi negatifse, dizinin elemanları ters sırada dönecektir. Örneklerde görelim:  
  
Giriş:  
  
Resif = ['kılıç balığı', 'köpekbalığı', 'balina', 'denizanası', 'ıstakoz', 'kalamar', 'ahtapot']  
Print(resif[-3:])  
Çıktı :  
  
['ıstakoz', 'kalamar', 'ahtapot']  
  
Negatif Dilimlemenin Şeması-1  
Giriş:  
  
Resif = ['kılıç balığı', 'köpekbalığı', 'balina', 'denizanası', 'ıstakoz', 'kalamar', 'ahtapot']  
Yazdır(resif[:-3])  
Çıktı :  
  
['kılıç balığı', 'köpekbalığı', 'balina', 'denizanası']  
  
Negatif Dilimlemenin Şeması-2  
Giriş:  
  
Resif = ['kılıç balığı', 'köpekbalığı', 'balina', 'denizanası', 'ıstakoz', 'kalamar', 'ahtapot']  
Print(reef[::-1]) # listenin tersini ürettik  
Çıktı :  
  
['ahtapot', 'kalamar', 'ıstakoz', 'denizanası', 'balina', 'köpekbalığı', 'kılıç balığı']  
  
Negatif Dilimlemenin Şeması-3  
Giriş:  
  
Resif = ['kılıç balığı', 'köpekbalığı', 'balina', 'denizanası', 'ıstakoz', 'kalamar', 'ahtapot']  
Print(resif[::-2])  
Çıktı :  
  
['ahtapot', 'ıstakoz', 'balina', 'kılıç balığı']  
💡İpuçları:  
Başlangıç ​​ve bitiş indeksleri birlikte negatif adımı seçerseniz, bunlar buna göre kullanılmalıdır, yani bitiş indeksi başlangıç ​​indeksinden küçük olmalıdır.  
Giriş:  
  
Tek\_no = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]  
  
Print(odd\_no[7:3:-1])  
Print(odd\_no[2:6:-1])  
Çıktı :  
  
[8, 7, 6, 5]  
[]  
Listelerde daha fazlasını bulmak istiyorsanız buraya bakın.  
S: list[::-1] ne yapar?  
A: list[::-1], listedeki öğelerin sırasını tersine çevirmek için kullanılır.

**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)

**Tanımlar**

Dersimizin bu bölümüne kadar Python'un en çok kullanılan koleksiyon türlerini gördük: liste. Tuple, listeye çok benzeyen birden çok veriyi tutabilen başka bir koleksiyon türüdür.  
  
Listeden en önemli farkı, tuple'ın değişmez olmasıdır. Bu nedenle, bu tür işlemlerde append() veya remove() gibi yöntemler yoktur.  
  
Tuple'lar, IP adresi ve bağlantı noktası gibi değişmesi gerekmeyecek küçük değer koleksiyonları için yaygın olarak kullanılır. Değişmemiş verimiz varsa, listelerden çok daha hızlı olduğu için tuple'ları seçmeliyiz.  
  
Listeleri tanımlamak için 👉🏻'[]' köşeli parantezleri kullandık. Tuple'da normal parantezler 👉🏻'()' kullanılır.  
  
Listeler için aynı indeksleme kuralları, tanımlama grupları için de geçerlidir. Tuple'lar da iç içe olabilir ve değerler geçerli herhangi bir Python geçerli olabilir.  
  
S: Liste ve demet arasındaki fark nedir?  
A:  
LİSTELER:  
  
Listeler değiştirilebilir yani düzenlenebilir.  
Listeler, demetlerden daha yavaştır.  
Sözdizimi: list\_1 = [True, 'Space', 20]  
TUPLE'ler:  
Tuple'lar değişmezdir (tuple'lar düzenlenemeyen listelerdir).  
Tuple'lar listeden daha hızlıdır.  
Sözdizimi: tup\_1 = (Doğru, 'Boşluk' , 20)

**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 tuples 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 string 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 tuples :

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')

**Bir Tuple Oluşturma**

Bir tanımlama grubu, virgülle ayrılmış değerleri parantez içine alarak da oluşturulabilir.  
  
Tuple'ı bir vakayla karşılaştırabilirsiniz. Değiştirmemesini istediğiniz verileri koyup kapağı kapattığınızda artık bu verileri değiştiremez, boyutunu değiştiremez ve düzenleyemezsiniz.  
  
Basit bir boş demet oluşturalım:  
  
Boş\_tuple = ()  
Bu, bu kurstaki ilk tuple'ımız. Şimdi türünü yazdıralım.  
  
Giriş:  
  
Boş\_tuple = ()  
Print(type(boş\_tuple))  
Çıktı :  
  
Sınıf 'tuple'  
Tek elemanlı bir demet oluşturmak istiyorsanız, virgül kullanmalısınız.  
  
Giriş:  
  
Try\_tuple = ('aşk')  
  
Yazdır(try\_tuple)  
  
Print(type(try\_tuple)) # demet türü değil.  
Çıktı :  
  
Aşk  
Sınıf 'str'  
Yalnızca tek elemanlı demetlerde oluşur ve elemanın sonunda virgül kullanarak sorunu çözebiliriz.  
  
💡İpuçları:  
Singleton Tuple tanımlarken her zaman virgül kullanmayı unutmayın.  
Giriş:  
  
Try\_tuple = ('aşk',)  
  
Yazdır(try\_tuple)  
  
Print(type(try\_tuple)) # bu bir demet türüdür.  
Çıktı :  
  
('Aşk',)  
Sınıf 'tuple'  
Aslında, demetiniz birden fazla eleman içeriyorsa, elemanları virgülle ayırmanız yeterli olacaktır.  
  
Bir tanımlama grubu oluşturmanın başka bir yolu da tuple() işlevini çağırmaktır. Bunu, yinelenebilir bir nesneden bir tanımlama grubu oluşturmak istediğinizde yaparsınız: yani öğelerini tek tek içe aktarabileceğiniz bir nesne türü.  
  
Tuple, diğer koleksiyonlar ve dize türleri gibi yinelenebilir. Tuple() fonksiyonunu kullanarak başka bir Tuple oluşturalım. Bu fonksiyon ile boş bir demet de oluşturabilirsiniz.  
  
Tuple oluşturmanın bazı örneklerini inceleyelim:  
  
Giriş:  
  
Gezegenler = 'cıva', 'jüpiter', 'satürn'  
  
Yazdır(gezegenler)  
Print(type(gezegenler))  
Çıktı :  
  
('cıva', 'jüpiter', 'satürn')  
Sınıf 'tuple'  
Giriş:  
  
Boş\_tuple\_1 = demet()  
  
Yazdır(empty\_tuple\_1)  
Print(type(empty\_tuple\_1))  
Çıktı :  
  
()  
Sınıf 'tuple'  
Aşağıdaki örneklerde olduğu gibi liste ve demet arasında dönüştürme yapmak kolaydır:  
  
Giriş:  
  
My\_tuple=(1, 4, 3, 4, 5, 6, 7, 4)  
  
My\_list = liste(my\_tuple)  
  
Print(type(my\_list), my\_list)  
Çıktı :  
  
Sınıf 'listesi' [1, 4, 3, 4, 5, 6, 7, 4]  
Giriş:  
  
My\_list = [1, 4, 3, 4, 5, 6, 7, 4]  
  
My\_tuple = demet(my\_list)  
  
Print(type(my\_tuple), my\_tuple)  
Çıktı :  
  
'tuple' sınıfı (1, 4, 3, 4, 5, 6, 7, 4)  
Yinelenebilir bir dize, bir Tuple'a dönüştürülebilir:  
  
Giriş:  
  
Dağ = demet('Alpler')  
Yazdır(dağ)  
Çıktı :  
  
('Alpler')

### 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 tuples.

Both lists and tuples 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 tuples.

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

**Tuple'ı Nasıl Kullanabiliriz?**Dilerseniz liste ve tuple'ın ortak özelliklerine bir göz atalım. Böylece tuple'larla ne yapacağınız hakkında bir fikriniz olabilir.  
  
Hem listeler hem de demetler sıralanır. Bu, öğeleri bu kaplara depolarken sıralarının aynı kalacağından emin olabileceğiniz anlamına gelir. Ayrıca değerleri çoğaltabilir veya farklı veri türlerini demetlerde karıştırabilirsiniz.  
  
Giriş:  
  
Mix\_value\_tuple = (0, 'kuş', 3.14, Doğru)  
  
Print(len(mix\_value\_tuple))  
Çıktı :  
  
4  
Başta belirttiğimiz gibi, listeler gibi, tupleler de indekslemeyi destekler:  
  
Giriş:  
  
Eşit\_no = (0, 2, 4)  
Yazdır(even\_no[0])  
Print(even\_no[1])  
Yazdır(even\_no[2])  
Yazdır(even\_no[3])  
Çıktı :  
  
0  
2  
4  
-------------------------------------------------- -------------------------  
Print(even\_no[3]): IndexError: demet dizini aralık dışında  
Tuple'ların listelerden en önemli farklarından biri de 'tuple' nesnesinin öğe atamasını desteklememesidir. Evet, çünkü tuple değişmez. Örneğe bakın:  
  
Giriş:  
  
City\_list = ['Tokyo', 'İstanbul', 'Moskova', 'Dublin']  
  
City\_list[0] = 'Atina'  
City\_list[1] = 'Kahire'  
Yazdır(şehir\_listesi)  
Çıktı :  
  
['Atina', 'Kahire', 'Moskova', 'Dublin']  
Giriş:  
  
City\_list = ['Tokyo', 'İstanbul', 'Moskova', 'Dublin']  
  
City\_tuple = demet(şehir\_listesi)  
  
City\_tuple[0] = 'New York' # bir değer atayamazsınız  
Çıktı :  
  
-------------------------------------------------- -------------------------  
TypeError: 'tuple' nesnesi öğe atamasını desteklemiyor

### Benefits of Immutability

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

* Tuples are faster and more powerful in-memory than lists. 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 tuples.
* 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 lists as keys. And this is the usefulness of tuples in the data processing.

**Değişmezliğin Faydaları**

Tuple'ların temel avantajlarına bir göz atalım:  
  
Tuple'lar, listelerden daha hızlı ve daha güçlü bellek içidir. Büyük miktarda veriyle uğraşmanız gerektiğinde bunu bir düşünmelisiniz. Verilerinizi değiştirmek istemiyorsanız, demetleri seçmeniz gerekebilir.  
  
Değişmezliği nedeniyle, bir demet içinde saklanan veriler yanlışlıkla değiştirilemez.  
  
Tuple bir sözlük (bir sonraki derste göreceğiz) anahtarı olarak kullanılabilirken, 'TypeError' anahtar olarak listelere neden olabilir. Ve bu, veri işlemede demetlerin kullanışlılığıdır.

**DİCTİONARİES**

### Definitions

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.

**Tanımlar**

Bu konumuzda eşya çiftlerini saklayan koleksiyon türlerini inceleyeceğiz. Bunun anlamı ne?  
  
Gerçek bir sözlük düşünün. Kelimeleri ve anlamlarını içerir. Python'da kelimeleri anahtar, kelimelerin anlamlarını değer olarak kabul edebilirsiniz.  
  
Python'daki bir sözlük, bir sözlüğün öğeleri olarak adlandırılan bir anahtar-değer çiftleri topluluğudur. Sözlük kaşlı ayraçlar içinde 👉🏻{}. Her bir çift (öğe) virgülle, anahtar ve değer ise iki nokta üst üste ile ayrılır.

**Creating a Dictionary**

A dictionary 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 dictionary : '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 item

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'], # list type

'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', number=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'}

**Sözlük Oluşturma**

Ayrıca, virgülle ayrılmış çiftleri kaşlı ayraçlar içine alarak da bir sözlük oluşturulabilir. Liste veya demet gibi görünüyor, değil mi?  
  
Ve elbette, bir sözlük oluşturmak için bir fonksiyon kullanabiliriz: 'dict()' fonksiyonu. Basit bir boş dict oluşturalım:  
  
Boş\_dict\_1 = {}  
  
Boş\_dict\_2 = dict()  
Bu, bu dersteki ilk diktimiz. Şimdi türünü yazdıralım.  
  
Giriş:  
  
Boş\_dict\_1 = {}  
  
Print(type(empty\_dict\_1))  
Çıktı :  
  
Sınıf 'dikt'  
Temel dict biçimi şöyle görünür:  
  
My\_dict = {'anahtar1': 'değer1',  
'anahtar2': 'değer2',  
'anahtar3': 'değer3'  
}  
Bir öğeye erişmek için sözdizimi çok basittir. Köşeli parantez içinde erişmek istediğimiz anahtarı yazıyoruz. Bu yöntem hem bir dict'e öğe eklemek için hem de onları oradan okumak için çalışır.  
  
Aşağıdaki örneklerde, bir dikte oluşturmamıza ve ona bir anahtar/değer çifti eklememize izin veren birkaç yöntem göreceksiniz.  
  
Giriş:  
  
State\_capitals = {'Arkansas': 'Küçük Kaya',  
'Colorado': 'Denver',  
'Kaliforniya': 'Sacramento',  
'Gürcistan': 'Atlanta'  
}  
  
Print(state\_capitals['Colorado']) # erişim yöntemi  
Çıktı :  
  
Denver  
Giriş:  
  
State\_capitals = {'Arkansas': 'Küçük Kaya',  
'Colorado': 'Denver',  
'Kaliforniya': 'Sacramento',  
'Gürcistan': 'Atlanta'  
}  
  
State\_capitals['Virginia'] = 'Richmond' # yeni bir öğe ekleme  
  
Print(durum\_büyük harf)  
Çıktı :  
  
{'Arkansas': 'Küçük Kaya',  
'Colorado': 'Denver',  
'Kaliforniya': 'Sacramento',  
'Gürcistan': 'Atlanta',  
'Virginia': 'Richmond'}  
💡İpuçları:  
Anahtarların ve değerlerin farklı türlerde olabileceğini unutmayın.  
Mix\_values ​​= {'hayvan': ('köpek', 'kedi'), # demet türü  
'gezegen': ['Neptün', 'Satürn', 'Jüpiter'], # liste türü  
'sayı': 40, # int türü  
'pi': 3.14, # kayan nokta tipi  
'is\_good': Doğru} # bool türü  
  
Mix\_keys = {22 : 'tamsayı',  
1.2 : 'yüzer',  
Doğru: 'boole',  
'anahtar': 'dize'}  
Şimdi bir sözlük oluşturmak için dict() işlevini kullanalım:  
  
Giriş:  
  
Dict\_by\_dict = dict(hayvan='köpek', gezegen='neptun', sayı=40, pi=3.14, is\_good=Doğru)  
  
Yazdır(dict\_by\_dict)  
Çıktı :  
  
{'hayvan': 'köpek',  
'gezegen': 'neptun',  
'sayı': 40,  
'pi': 3.14,  
'is\_good': Doğru}  
⚠️ Kaçının ! :  
Sözlük oluşturmak için dict() işlevini kullanırken anahtarlar için tırnak işaretleri kullanmayın.  
Yinelenebilirleri bir sözlük oluşturmak için anahtar olarak kullanamazsınız.  
S: Python'da sözlük nedir?  
A: Python sözlüğü, Python'da desteklenen veri türlerinden biridir. Sırasız bir elementler topluluğudur. Sözlüklerdeki öğeler, anahtar/değer çiftleri olarak depolanır. Sözlükler anahtarlarla indekslenir. Örneğin, aşağıda my\_dict adında bir dict var. İki anahtar, meyve ve sebze, bunlara karşılık gelen değerler, muz ve soğan içerir.  
  
My\_dict = {'meyve':'muz', 'sebze':'soğan'}

**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 dictionary. 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 dictionary.

* 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

**Sözlüklerle Ana İşlemler**

Öğelere, anahtarlara ve değerlere erişmemize izin veren birkaç yöntem vardır. Tüm öğelere .items() yöntemini kullanarak, tüm anahtarlara .keys() yöntemini kullanarak ve tüm değerlere .values() yöntemini kullanarak erişebilirsiniz:  
  
Giriş:  
  
Dict\_by\_dict = {'hayvan': 'köpek',  
'gezegen': 'neptun',  
'sayı': 40,  
'pi': 3.14,  
'is\_good': Doğru}  
  
Print(dict\_by\_dict.items(), '  
')  
Print(dict\_by\_dict.keys(), '  
')  
Print(dict\_by\_dict.values())  
Çıktı :  
  
Dict\_items([('hayvan', 'köpek'), ('gezegen', 'neptun'),  
('sayı', 40), ('pi', 3.14), ('iyi\_iyi', Doğru)])  
  
Dict\_keys(['hayvan', 'gezegen', 'sayı', 'pi', 'iyi\_iyi'])  
  
Dict\_values(['köpek', 'neptun', 40, 3.14, Doğru])  
Sözlükte olmayan bir anahtara değer atayarak yeni bir öğe ekleyebileceğinizi öğrendiniz. Aynı şekilde, .update() yöntemini kullanarak yeni öğeler ekleyebilirsiniz. Görelim :  
  
Giriş:  
  
Dict\_by\_dict = {'hayvan': 'köpek',  
'gezegen': 'neptun',  
'sayı': 40,  
'pi': 3.14,  
'is\_good': Doğru}  
  
Dict\_by\_dict.update({'is\_bad': Yanlış})  
  
Yazdır(dict\_by\_dict)  
Çıktı :  
  
{'hayvan': 'köpek',  
'gezegen': 'neptun',  
'sayı': 40,  
'pi': 3.14,  
'is\_good': Doğru,  
'is\_bad': Yanlış}  
Ayrıca del işlevini kullanarak bir öğeyi kaldırabilirsiniz:  
  
Formül sözdizimi şöyledir: del sözlük\_adı['anahtar'].  
  
Örneğe bakın.  
  
Giriş:  
  
Dict\_by\_dict = {'hayvan': 'köpek',  
'gezegen': 'neptun',  
'sayı': 40,  
'pi': 3.14,  
'is\_good': Doğru,  
'is\_bad': Yanlış}  
  
Del dict\_by\_dict['hayvan']  
  
Yazdır(dict\_by\_dict)  
Çıktı :  
  
{'gezegen': 'neptün',  
'sayı': 40,  
'pi': 3.14,  
'is\_good': Doğru,  
'is\_bad': Yanlış}  
In ve not in operatörünü kullanarak, anahtarın sözlükte olup olmadığını kontrol edebilirsiniz.  
  
In operatörünü kullandığımızda; Anahtar sözlükteyse, sonuç True, aksi halde False olacaktır.  
  
Not in'i kullandığımızda; Anahtar sözlükte değilse, sonuç True, aksi halde False olacaktır.  
  
Örneğe bakınız :  
  
Giriş:  
  
Dict\_by\_dict = {'gezegen': 'neptun',  
'sayı': 40,  
'pi': 3.14,  
'is\_good': Doğru,  
'is\_bad': Yanlış}  
  
Print(dict\_by\_dict içinde 'pi')  
Print('animal' dict\_by\_dict'te değil) # unutmayın, 'animal'ı sildik  
Çıktı :  
  
NS  
NS

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": 16}

},

},

"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 dicts :

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

| **nested_dictionary** |
| --- |
| *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.

**İç İçe Sözlükler**

Bazı durumlarda, iç içe dict ile çalışmanız gerekir. Veri biliminde uzmanlaşmaya karar verdiğinizde gelecekte sözlüklerle çok sık çalışacağız.  
  
Okul\_kayıtları={  
'kişisel bilgi':  
{'çocuk':{'tom': {'sınıf': 'orta', 'yaş': 10},  
'dava': {'sınıf': 'ilkokul', 'yaş': 8}  
},  
'genç':{'joseph':{'sınıf': 'kolej', 'yaş': 19},  
'evlenmek':{'sınıf': 'lise', 'yaş': 16}  
},  
},  
  
'grades\_info':  
{'çocuk':{'tom': {'matematik': 88, 'konuşma': 69},  
'dava': {'matematik': 90, 'konuşma': 81}  
},  
'genç':{'joseph':{'kodlama': 80, 'matematik': 89},  
'evlenmek':{'kodlama': 70, 'matematik': 96}  
},  
},  
}  
Dahili diktelere erişmek için köşeli parantez kullanabiliriz:  
  
Giriş:  
  
Okul\_kayıtları={  
'kişisel bilgi':  
{'çocuk':{'tom': {'sınıf':'orta', 'yaş':10},  
'dava': {'sınıf':'ilköğretim', 'yaş':8}  
},  
'genç':{'joseph':{'sınıf':'kolej', 'yaş':19},  
'evlenmek':{'sınıf':'lise', 'yaş':16}  
},  
},  
}  
  
Print(school\_records['personal\_info']['genç']['evlenmek']['yaş'])  
Çıktı :  
  
16  
  
İç İçe Sözlük Diyagramı  
💡İpuçları:  
Sözlükler büyük ölçüde JSON sözdizimine benzer. Python standart kitaplığındaki yerel json modülü, JSON ve sözlükler arasında dönüştürme yapmak için kullanılabilir.

**SETS**

### Definitions

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.

**Tanımlar**

Küme, tekrarı olmayan ve ekleme sırası olmayan ancak sıralanmış bir öğeler topluluğudur.  
  
Temel kullanımlar, üyelik testini ve yinelenen girişleri ortadan kaldırmayı içerir. Küme nesneleri aynı zamanda birleşim, kesişim, fark ve simetrik fark gibi matematiksel işlemleri de destekler.  
  
İçlerinde birden fazla veri tutabilirler, ancak yalnızca bir değeri vardır. Hangi sırayla dahil edildiklerinin değil, yalnızca bazı şeylerin birlikte gruplandırılmasının önemli olduğu durumlarda kullanılırlar.

**Creating a Set**

Curly braces **'{}'** or the **set()** function can be used to create sets. 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 dictionary.

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 sets have unordered and unique objects.

input :

colorset = {'purple', 'orange', 'red', 'darkblue', 'yellow', '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 string type data are only written once in the set. Within this scope, using sets 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', 'orchid', '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.

**Set Oluşturma**

Kümeler oluşturmak için küme parantezleri '{}' veya set() işlevi kullanılabilir. Ancak boş bir küme oluşturmanın tek yolu şudur: set() işlevini kullanmak.  
  
⚠️ Kaçının ! :  
Boş bir küme oluşturmak için set() işlevini kullanmanız gerektiğini unutmayın. Boş bir küme oluşturmak için {} kullanmayın. Aksi takdirde boş bir sözlük oluşturacaksınız.  
Basit bir boş küme oluşturalım:  
  
Boş\_küme = küme()  
Bu dersteki ilk setimiz. Şimdi türünü yazdıralım.  
  
Giriş:  
  
Boş\_küme = küme()  
  
Yazdır(tür(boş\_küme))  
Çıktı :  
  
Sınıf 'set'  
Şimdi kümelerin nasıl sırasız ve benzersiz nesnelere sahip olduğunu göreceğiz.  
  
Giriş:  
  
Colorset = {'mor', 'turuncu', 'kırmızı', 'lacivert', 'sarı', 'kırmızı'}  
  
Yazdır(renk seti)  
Yazdır(renk seti)  
Çıktı :  
  
{'lacivert', 'turuncu', 'mor', 'kırmızı', 'sarı'}  
{'lacivert', 'mor', 'turuncu', 'sarı', 'kırmızı'}  
Çıktıda da görebileceğiniz gibi sette tanımladığımız iki 'kırmızı' değer bire düşmüş durumda. Ve seti her yazdırdığınızda, setteki nesnelerin sırası değişir.  
  
Başka bir örneğe bakalım:  
  
Giriş:  
  
S = set('bencillik')  
  
Baskı(lar)  
Çıktı :  
  
{'f', 'l', 'ben', 'u', 'e', ​​'n', 'h', 's'}  
Görüldüğü gibi string türündeki verilerin harfleri sette sadece bir kez yazılır. Bu kapsamda set kullanmak tekrarlardan kaçınmanıza yardımcı olabilir. Bir listeyi bir kümeye dönüştürelim ve elemanlarının tekrarlarına bakalım:  
  
Giriş:  
  
Flower\_list = ['gül', 'menekşe', 'karanfil', 'gül', 'orkide', 'gül', 'orkide']  
Çiçek takımı = set(flower\_list)  
Çiçek listesi = liste(çiçek seti)  
  
Baskı(çiçek seti)  
Yazdır(çiçek listesi)  
Çıktı :  
  
{'orkide', 'karanfil', 'menekşe', 'gül'}  
['orkide', 'karanfil', 'menekşe', 'gül']  
✏️Ödev:  
{'karanfil', 'orkide', 'gül', 'mor'} 👈👉 {'gül', 'orkide', 'gül', 'menekşe', 'karanfil'} Bu iki set aynı çıktıyı mı veriyor ve neden ? (Not: Oyun Alanında koşmadan önce cevabı bulmaya çalışın)  
S: Python'da küme oluşturmak için aşağıdakilerden hangisi doğru sözdizimi değildir?  
A:  
A. Set([[1,2],[3,4],[4,5]])  
B. Küme([1,2,2,3,4,5])  
C. {1,2,3,4}  
NS. Küme((1,2,3,4))  
  
Açıklama: Küme için verilen yinelenebilir argüman doğru şekilde kullanılmalıdır.

**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.

**Setlerle Ana İşlemler**  
Kümelere öğe eklememize ve kümelerden öğe çıkarmamıza izin veren birkaç yöntem vardır. Ayrıca kümelerin kesişim, birleştirme ve farklılaşma yöntemlerine sahibiz:  
  
Bu yöntemler şunlardır:  
  
.add() : Kümeye yeni bir öğe ekler.  
  
.remove() : Bir öğeyi silmemizi sağlar.  
  
.intersection() : İki kümenin kesişimini döndürür.  
  
.union() : İki kümenin birleşimini döndürür.  
  
.difference() : İki kümenin farkını alır.  
  
Şimdi bu yöntemlerden bazı örnekler yapalım:  
  
Giriş:  
  
A = küme('abrakadabra')  
  
Yazdır(a)  
Çıktı :  
  
{'a', 'b', 'c', 'd', 'r'}  
Giriş:  
  
A = küme('abrakadabra')  
B = set('alacazam')  
  
Print(a - b) # '.difference()' yöntemiyle aynı  
Print(a.difference(b)) # b'den bir fark  
Çıktı :  
  
{'b', 'd', 'r'}  
{'b', 'd', 'r'}  
Giriş:  
  
A = küme('abrakadabra')  
B = set('alacazam')  
  
Print(a | b) # '.union()' yöntemiyle aynı  
Print(a.union(b)) # a'nın b ile birleştirilmesi  
Çıktı :  
  
{'a', 'b', 'c', 'd', 'l', 'm', 'r', 'z'}  
{'a', 'b', 'c', 'd', 'l', 'm', 'r', 'z'}  
Giriş:  
  
A = küme('abrakadabra')  
B = set('alacazam')  
  
Print(a & b) # '.intersection()' yöntemiyle aynı  
Print(a.intersection(b)) # a ve b'nin kesişimi  
Çıktı :  
  
{'AC'}  
{'AC'}  
Giriş:  
  
A = küme('abrakadabra')  
  
A.remove('c') # setten 'c'yi siliyoruz  
Yazdır(a)  
Çıktı :  
  
{'a', 'b', 'd', 'r'}  
Giriş:  
  
A = küme('abrakadabra')  
  
A.add('c') # kümeye tekrar 'c' ekliyoruz  
Yazdır(a)  
Çıktı :  
  
{'a', 'b', 'c', 'd', 'r'}  
Ek olarak şunları yapabilirsiniz:  
  
Len() işlevini kullanarak kümenin eleman sayısını alın,  
Bir öğenin belirli bir kümeye ait olup olmadığını kontrol edin (operatörlerde / değil), boole değerini alırsınız.  
Böylece Python'da en önemli konu olan bu konuyu tamamlamış olduk.