Tuple (Demet) Veri Yapısı

Python'da tuple (demet) veri yapısı, verileri sıralı ve değiştirilemez bir biçimde saklamak için kullanılan bir veri tipidir. tuple veri yapısı, listelere oldukça benzese de, önemli bir farkı vardır: Tuple'lar değiştirilemezdir (immutable). Bu, bir tuple oluşturulduktan sonra içindeki elemanların değiştirilemeyeceği anlamına gelir. Tuple'lar genellikle sabit veri kümeleri veya değiştirilmeyecek bilgileri saklamak için tercih edilir.

Bir tuple'ın genel özellikleri:

1. **Sıralıdır (Ordered):** Tuple elemanları belirli bir sıraya sahiptir ve bu sıra korunur.
2. **Değiştirilemezdir (Immutable):** Tuple oluşturulduktan sonra içeriği değiştirilemez. Eleman ekleme, çıkarma veya değiştirme gibi işlemler yapılamaz.
3. **Heterogeneous (Çok Türlü):** Farklı veri türlerinden (sayılar, dizeler, booleanlar vb.) verileri bir arada tutabilir.
4. **Tekrar Edilebilir (Iterable):** Tuple üzerinde döngülerle (for döngüsü gibi) işlem yapılabilir.

Bir tuple'ın kullanım alanları:

- **Sabit Veriler:** Değiştirilmeyecek sabit bir veri kümesini saklamak için kullanılır (örneğin, aylar, günler, ülkeler vb.).
- **Fonksiyonlardan Çoklu Değer Döndürme:** Fonksiyonlar, birden fazla değeri bir tuple olarak döndürebilir.
- **Veri Güvenliği:** Tuple'lar sabit oldukları için birden fazla kişi veya işlem tarafından kullanıldıklarında verinin bozulmaması sağlanır.
- **Dictionary Anahtarı (Key) Olarak Kullanma:** Tuple'lar değiştirilemez olduklarından, bir dict anahtarı olarak kullanılabilirler.

Tuple'lar, Python'da hızlı, sabit ve güvenilir bir veri yapısı olarak yaygın bir şekilde kullanılır. Bir verinin sabit kalması isteniyorsa, tuple'lar en ideal bir seçimdir.

Tuple veri yapısının kullanımı

Bir **tuple** veri yapısı listelere çok benzer. Demetlerde yer alan veriler yuvarlak normal parantez `( )` içinde ve virgülle ayrılarak gösterilir. Listeler gibi verileri aldıkları sırada saklar ve saklanan veriye indeks numarası ile erişmek mümkündür. Ancak listelerden farklı olarak bir defa tanımlandıktan sonra demet veri yapısındaki değerleri değiştirmek mümkün değildir. Bu nedenle, oluşturulan bir verinin kesinlikle değiştirilmemesi gereken zamanlarda demet veri yapısı daha güvenilirdir. Bu özellik, veri güvenliği ve korunmasının gerekli olduğu işlerde çok faydalıdır.

```
demet1 = (3.14, 2.78, 4.27, 7.45)
demet2 = ("İstanbul", "Ankara", "İzmir", "Bursa", "Adana")
demet3 = ("Fizik", "Matematik", "Kimya", 100, 99, 97)
```

Görüldüğü gibi tuple'da listeler gibi farklı türde verileri içerebilir ve indeks işlemleri de listelerde kullanıldığı gibidir. Ayrıca demet içindeki verileri direkt diğer değişkenlere aktarılabilir. Ancak değişken sayısı tupel'daki eleman sayısı ile aynı olmalıdır, ne eksik ne de fazla. Aksi durumda hata mesajı verilir.

```
print(demet3[1])
'Matematik'
print(demet3[2:4])
('Kimya', 100)
```

```
sabit = (3.14, 2.78)
pi, e = sabit
print(pi, e)
3.14 2.78
```

```
demet3[1] = "Biyoloji"
TypeError: 'tuple' object does not support item assignment
```

```
sayi = (3, 6, 7)
a, b = sayi
ValueError: too many values to unpack (expected 2)

a, b, c, d = sayi
ValueError: not enough values to unpack (expected 4, got 3)
```

Tuple veri yapısı kullanılan metotlar

Tuple veri yapısı üzerinde kullanılabilecek kısıtlı sayıda yerleşik metot vardır, çünkü tuple'lar değiştirilemezdir. Sadece iki metot bulunmaktadır ve bunlar şu şekilde özetlenebilir.

index(x): Belirtilen değerin tuple içinde ilk geçtiği konumun indeksini döndürür. Eğer eleman bulunamazsa hata verir.

Kullanım Şekli: tuple.index(eleman, başlama_indeksi, bitiş_indeksi)

başlama_indeksi ve bitiş_indeksi isteğe bağlı olup verilmediği durumda tuple elemanlarını baştan sona doğru tarama yapar ve kısaca aramanın yapılacağı aralığı belirler.

Örnek 1: İlk geçişin indeksini bulma

```
hayvanlar = ("kedi", "köpek", "kuş", "köpek", "balık")
ilk_kopek = hayvanlar.index("köpek")

print(f"'köpek' ilk olarak {ilk_kopek}. indeksde bulundu.")
'köpek' ilk olarak 1. indeksde bulundu.
```

Örnek 2: Belirli bir aralıkta arama

```
sayilar = (10, 20, 30, 20, 40, 50, 20)
ilk_20 = sayilar.index(20, 2)  # 2. indeksten itibaren arar

print(f"20 rakamı 2. indeksten sonra ilk olarak {ilk_20}. indekste bulundu.")
20 rakamı 2. indeksten sonra ilk olarak 3. indekste bulundu.
```

Tuple veri yapısı kullanılan metotlar

Örnek 3: count ve index metotlarının birlikte kullanımı

```
veriler = (1, 2, 3, 2, 4, 5, 2, 6)
eleman = 2

tekrar_sayisi = veriler.count(eleman)
ilk_konum = veriler.index(eleman)

print(f"{eleman} rakamı {tekrar_sayisi} kez geçiyor ve ilk olarak {ilk_konum}. indekste bulunuyor.")
2 rakamı 3 kez geçiyor ve ilk olarak 1. indekste bulunuyor.
```

count(x): Belirtilen değerin tuple içinde kaç kez geçtiğini döndürür.

Kullanım şekli: tuple.count(eleman)

Örnek 1: Tekrar eden eleman sayısını bulma

```
meyveler = ("elma", "armut", "elma", "kiraz", "elma")
elma_sayisi = meyveler.count("elma")

print(f"'elma' kelimesi {elma_sayisi} kez geçiyor.")
'elma' kelimesi 3 kez geçiyor.
```

Örnek 2: Sayıların tekrar sayısını bulma

```
sayilar = (1, 2, 3, 4, 2, 2, 5, 6)
iki_sayisi = sayilar.count(2)

print(f"2 rakamı {iki_sayisi} kez geçiyor.")
2 rakamı 3 kez geçiyor.
```

List (Liste) Veri Yapısı

Python'da liste tipi veri yapısı (list), veri topluluklarını sıralı ve değiştirilebilir bir şekilde saklamak için kullanılır. Listeler, Python'un en yaygın kullanılan veri yapılarından biridir ve çeşitli veri türlerini (sayılar, karakter dizileri, nesneler vb.) bir arada saklayabilir. Listelerin özellikleri, metotları ve kullanım şekilleri şu şekilde özetlenebilir.

Bir listenin genel özellikleri:

1. **Sıralıdır (Ordered):** Listelerdeki elemanlar eklenme sırasına göre sıralanır. İlk eklenen eleman, listenin başında yer alır ve bu sıra korunur.
2. **Değiştirilebilir (Mutable):** Listelerdeki elemanlar değiştirilebilir. Listenin herhangi bir elemanı değiştirilebilir, silinebilir veya yeni elemanlar eklenebilir.
3. **Kapsayıcıdır (Heterogeneous):** Bir liste içerisinde farklı veri türleri bir arada bulunabilir (örneğin, int, str, float vb.).
4. **Tekrar Edilebilir (Iterable):** Listeler üzerinde döngüler (örneğin, for döngüsü) kullanılabilir ve elemanlar teker teker erişilebilir.

Bir listenin kullanım alanları:

- **Veri depolama:** Farklı veri türlerinden verileri bir arada tutmak.
- **Döngülerle işlem yapmak:** Listeler üzerinde yinelemeli işlemler yapmak.
- **Veri filtreleme:** Koşullara göre listeden veri seçmek veya yeni listeler oluşturmak.
- **Dizilerle çalışma:** Matematiksel veya istatistiksel işlemler yapmak için sayı listeleri oluşturmak.

Listeler, Python'da güçlü ve esnek bir veri yapısıdır. Hem basit veri kümeleri hem de daha karmaşık veri yapıları için uygun bir çözümdür.

Liste veri yapısı kullanımı

Liste, isminden de anlaşılacağı üzere sıralı nesnelerden oluşan liste ya da dizilerdir. Listelerde yer alan veriler köşeli parantez `[ ]` içinde ve virgülle ayrılarak gösterilir. Listeler veriyi eklenme sırasına göre tutan ve içeriği sonradan değiştirilebilen veri yapılarıdır. Bir listeyi tanımladıktan sonra listenin bir veya daha fazla elemanını güncellemek mümkündür.

Listenin elemanları önceki derslerde anlatıldığı üzere veri tiplerinden herhangi birisi olabilir. Bir listenin elemanları farklı veri tipinde de olabilir. Hatta, bir listenin elemanları farklı listeler ve değişken de olabilir.

```
liste1 = [41, 42, 43, 44, 45]
print(liste1)
[41, 42, 43, 44, 45]
```

```
liste2 = [123, 456, "kim", "xyz"]
print(liste2)
[123, 456, 'kim', 'xyz']
```

```
notlar1 = ['Eren',100, 'İlhan',99, 'Ali',98, 'Erman',97, 'Mehmet',96]
notlar2 = [['Eren', 100],
           ['Ilhan',99],
           ['Ali',98],
           ['Erman',97],
           ['Mehmet',96]]
notlar3 = [['Eren', 'İlhan', 'Ali', 'Erman', 'Mehmet'],
           [100, 99, 98, 97, 96]]

kisi_1 = "Eren"
kisi_2 = "İlhan"
not_1 = 100
not_2 = 96
notlar4 = [[kisi_1, not_1], [kisi_2, not_2]]
print(notlar4)
[['Eren', 100], ['İlhan', 96]]
```

Liste veri yapısı kullanılan metotlar

Python'da listeler üzerinde çeşitli işlemler yapmamıza olanak sağlayan birçok yerleşik metot vardır. Bunlar şu şekilde özetlenebilir.

1. **append(x):** Listenin sonuna bir eleman ekler. Örnek: liste.append(5) # [1, 2, 3, 5]
2. **extend(iterable):** Listeyi verilen sayılabilir eleman (örneğin, liste veya dizi) ile genişletir. Örnek: liste.extend([6, 7]) # [1, 2, 3, 5, 6, 7]
3. **insert(i, x):** Belirtilen indekse (i) eleman ekler. Örnek: liste.insert(2, 10) # [1, 2, 10, 3, 5, 6, 7]
4. **remove(x):** Belirtilen elemanı listeden siler (ilk bulduğu yerde). Örnek: liste.remove(10) # [1, 2, 3, 5, 6, 7]
5. **pop([i]):** Belirtilen indeksteki elemanı siler ve döndürür. İndeks verilmezse son elemanı siler. Örnek: eleman = liste.pop(2) # 3, liste -> [1, 2, 5, 6, 7]
6. **clear():** Listedeki tüm elemanları siler. Örnek: liste.clear() # []
7. **index(x):** Belirtilen elemanın ilk bulunduğu konumun indeksini döndürür. Örnek: indeks = liste.index(5) # 3
8. **count(x):** Belirtilen elemanın listede kaç kez bulunduğunu döndürür. Örnek: sayi = liste.count(5) # 1
9. **sort():** Liste elemanlarını sıralar. Örnek: liste.sort() # küçükten büyüğe sıralar, liste.sort(reverse=True) # büyükten küçüğe sıralar
10. **reverse():** Listedeki elemanların sırasını ters çevirir. Örnek: liste.reverse() # Listeyi tersine çevirir
11. **copy():** Listenin bir kopyasını döndürür. Örnek: yeni_liste = liste.copy()

Liste indeksleme ve dilimleme

Python'da bütün değişkenler dizi mantığı işlem görür. Bu ise bir değişkenin kendi içindeki elemanlara tek tek erişebileceği anlamına gelir. Liste veri yapılarında bütün elemanlar virgülle ayrıldığından, elemanların liste içinde belli bir sıraları bulunmaktadır. Bu sıra dizisine indeks numaraları ile erişmek mümkündür. Örneğin;

sınav = [41, 33, 47, 12, 25, 17, 81, 21, 19]

şeklindeki bir listeyi göz önüne alalım. Burada elemanlar en soldan itibaren sağa doğru dizilmektedir. Doğal olarak en soldaki ilk elaman 1., arkasından gelen 2., sonraki 3. ve hakeza diğerleri de benzer şekilde bir sıra numarası alacaklardır. Python'da diğer kodlama dillerinde olduğu gibi bu sıra numarası 1'den değil 0 (sıfır) ile başlatılmakta ve sağa doğru ardışık artmaktadır.

Listenin elemanları bir dizi olması sebebiyle sondan başa doğru da bir indeksleme yapılması mümkün olmaktadır. Ancak bu durumda indeks numarası sıfır ile değil -1 ile başlamaktadır. Liste elemanlarını belirli grup halinde erişme işlemi **":"** üst üste iki nokta sembolü ile yapılır ve **"dilimleme"** adı verilmektedir.

0	1	2	3	4	5	6	7	8
41	**33**	**47**	**12**	**25**	**17**	**81**	**21**	**19**
-9	-8	-7	-6	-5	-4	-3	-2	-1

başlangıç indisi : bitiş indisi : atlama sayısı

```
sınav = [41, 33, 47, 12, 25, 17, 81, 21, 19]
print(sınav[0])
41

print(sınav[3])
12

print(sınav[6])
81
```

```
print(sınav[-1])
19

print(sınav[-2])
21

print(sınav[-4])
25
```

```
print(sınav[3:])
[12, 25, 17, 81, 21, 19]

print(sınav[:4])
[41, 33, 47]

print(sınav[2:7:2])
[47, 25, 81]
```

Liste elemanlarına erişim

Eğer bir liste içinde başka listeler olursa, bu alt listelere ikinci bir indeks kullanarak erişilmektedir. Liste isminin yanındaki 1. indeks alt listenin indeksini 2. indeks ise alt listedeki elemanın indeksini gösterecektir. Alt listede her hangi indeks aralığındaki elemanları almak veya görmek için önceki indeks kuralları kullanılır.

```
notlar = [['Eren', 'İlhan', 'Ali', 'Erman', 'Mehmet'],
                [100, 99, 98, 97, 96]]

print(notlar[0])
['Eren', 'İlhan', 'Ali', 'Erman', 'Mehmet']

print(notlar[0][2])
'Ali'

print(notlar[0][1:3])
['İlhan', 'Ali', 'Erman']
```

Listedeki herhangi bir elemanın liste içindeki sıra numarasını yani indeks numarasını öğrenmek için **.index()** metodunu kullanabiliriz.

```
#           0      1      2     3     4     5     6     7      8      9
notlar = ["Eren", 100, "İlhan", 99, "Ali", 98, "Erman", 97, "Mehmet", 96]

print(notlar.index("Ali"))
4

print(notlar.index(99))
3
```

Liste metotları - örnekler

Listeden belirli bir sıradaki elemanı silmek için **.pop()** metodu kullanmaktır. Bu metot istenen bir elemanı listeden silerken aynı zamanda silinen elemanı başka bir değişkene atayabilme imkanını vermektedir. Bir listeden eleman silmenin diğer bir yol **del()** fonksiyonunu kullanmaktır.

```
#            0        1       2       3        4
isimler = ["Eren", "İlhan", "Ali", "Erman", "Mehmet"]

sil = isimler.pop(3)

print(isimler)
['Eren', 'İlhan', 'Ali', 'Mehmet']

print(sil)
'Erman'
```

```
del(isimler[1])

print(isimler)

['Eren', 'Ali', 'Erman', 'Mehmet']
```

Ayrıca bir liste içindeki bir elemanı direkt değeriyle silmek için **.remove()** metodu ve liste içinde bir elemandan kaç tane olduğunu bulmak için ise **.count()** metodu kullanılır.

```
isimler.remove("Eren")

print(isimler)

['İlhan', 'Ali', 'Erman', 'Mehmet']
```

```
olcum_degerleri = [1, 3, 6, 9, 3, 4, 4, 7, 8, 2, 4, 2, 1]

print(olcum_değerleri.count(4))

3
```

Eğer bir listede olmayan bir indeks numarasıyla atama yapılmak istenir bir hata mesajı alınacaktır. Çünkü indeks numarası eleman sayısında büyük olacaktır.

```
sira = [1, 2, 3, 10]
sira[4] = 15

Traceback (most recent call last):
File "<stdin>", line 1, in <module>
IndexError: list assignment index out of range
```

Liste metotları - örnekler

Mevcut bir listeye yeni eleman ya da elemanlar eklemenin 4 farklı yöntemi bulunmaktadır. İlki ve en kolay normal **"+" toplama** operatörü iki listeyi birbirine eklemedir. Diğer 3 yöntem liste fonksiyonlarıdır ki bunlar;

.**extend()** fonksiyonu ile bir liste diğer bir listeye,
.append() fonksiyonu ile bir listenin sonuna eleman,
.insert() fonksiyonu ile bir listenin belirli bir sırasına eleman,

eklenebilir.

```
isim_1 = ['Eren', 'İlhan', 'Ali']
isim = isim_1 + ['Erman', 'Mehmet']

print(isim)
['Eren', 'İlhan', 'Ali', 'Erman', 'Mehmet']

isim[1] = 'Hakan'

print(isim)
['Eren', 'Hakan', 'Ali', 'Erman', 'Mehmet']

isim.append('Semih')

print(isim)
['Eren', 'Hakan', 'Erman', 'Mehmet', 'Semih']
```

```
liste = [1, 2, 3, 4]
harf = ['bir', 'iki', 'üç', 'dört']

liste.extend(harf)

print(liste)
[1, 2, 3, 4, 'bir', 'iki', 'üç', 'dört']

liste.insert(4, 7)

print(liste)
[1, 2, 3, 4, 7, 'bir', 'iki', 'üç', 'dört']
```

Sayısal değerler olan bir listede elemanları küçükten büyüğe sıralamak için **sorted()** fonksiyonu kullanılır. Eğer bu işlemin tersini istersek fonksiyonun **reverse=True** parametresini kullanmak gerekir.

```
liste = [3, 7, 2, 11, 3, 7, 9, 5, 18, 4]
yeni_liste = sorted(liste)

print(yeni_liste)
[2, 3, 3, 4, 5, 7, 7, 9, 11, 18]
```

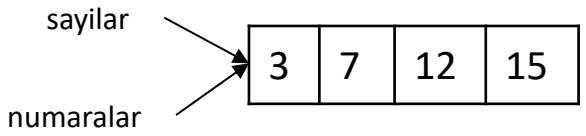
```
yeni_liste = sorted(liste, reverse=True)

print(yeni_liste)
[18, 11, 9, 7, 7, 5, 4, 3, 3, 2]
```

Liste kopyalama

Python'da bir liste oluşturduğumuzda bu liste bilgisayar hafızasında saklanır. Bu listeye isim verdiğimizde, bu isim aslında listedeki verileri içermez. Listedeki verilerin geçici hafızada saklandığı adresi gösteren bir işaret görevi görür. Örneğin sayılar isminde sayilar = [3, 7, 12, 15] bir liste olsun. Diyelim ki yeni bir liste oluşturup **sayilar** listesini **numaralar** isimli bir listeye atamak istiyoruz.

```
sayilar = [3, 7, 12, 15]
numaralar = sayilar
print(numaralar)
[3, 7, 12, 15]
```



Bu durumda **sayilar** ve **numaralar** aynı adresi göstermiş olacaktır. Birinde yapılan değişiklik diğerinde de olacaktır. Listelerin bir birinden ayrı olması için iki farklı kopyalama metodu vardır.

```
sayilar[1] = 99
print(sayilar)
[3, 99, 12, 15]
print(numaralar)
[3, 99, 12, 15]
```

```
numaralar = sayilar[:]
numaralar = list(sayilar)
sayilar[1] = 99
print(sayilar)
[3, 99, 12, 15]
print(numaralar)
[3, 7, 12, 15]
```

Bir listeyi ters sırada düzenlemek **.reverse()** fonksiyonu kullanılır. Ayrıca dilimle operatörleriyle de tersine çevirme işlemi yapılabilir.

```
sayilar = [3, 7, 12, 15]
sayilar.reverse()
print(sayilar)
[15, 12, 7, 3]
```

```
sayilar = [3, 7, 12, 15]
yeni_sayilar = sayilar[::-1]
print(yeni_sayilar)
[15, 12, 7, 3]
```

Liste Anlama (List Comprehension)

Python'da liste oluşturmanın kısa ve etkili bir yoludur. Liste Anlama kullanarak, bir listeyi kısa bir ifadeyle hızlı bir şekilde oluşturabilirsiniz. Bu yapı, genellikle bir döngü ve koşullu ifadeleri bir arada kullanarak bir liste oluşturmak için tercih edilir. Liste Anlama, daha okunabilir ve genellikle daha performanslı bir kod yazmanızı sağlar.

Liste Anlama yapısı, bir liste tanımlamak için kullanılan köşeli parantezler [] içerisine yazılan bir ifadeden oluşur. Genel yapı şu şekildedir:

yeni_liste = [ifade for eleman in dizi if koşul]

- **ifade:** Oluşturulacak listenin her elemanı için kullanılan ifade.
- **for eleman in dizi:** Döngü ile bir iterasyon gerçekleştirilir (bir liste, dize, range veya başka bir yineleyici olabilir).
- **if koşul (isteğe bağlı):** Elemanların belirli bir koşulu sağlaması gerektiğinde kullanılan şart

Temel Kullanım: Bir listedeki elemanların karesini almak

```
sayilar = [1, 2, 3, 4, 5]
kareler = [x**2 for x in sayilar]
print(kareler)
[1, 4, 9, 16, 25]
```

Karmaşık İfade Kullanımı: Sayıların karesini alıp, sadece 10'dan büyük olanları seçmek

```
sayilar = [1, 2, 3, 4, 5]
buyuk10 = [x**2 for x in sayilar if x**2 > 10]
print(buyuk10)
[16, 25]
```

Koşullu Kullanım: Sadece çift sayılardan oluşan bir liste oluşturmak:

```
sayilar = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
cift_sayilar = [x for x in sayilar if x % 2 == 0]
print(cift_sayilar)
[2, 4, 6, 8, 10]
```

Liste anlama - matrisler

List anlama, iki boyutlu listeler (matrisler) üzerinde işlem yapmak için de kullanılabilir. Örneğin 3x3 bir matris şu şekilde oluşturulabilir.

```
matris = [[j for j in range(3)] for i in range(3)]

print(matris)
[[0, 1, 2], [0, 1, 2], [0, 1, 2]]
```

Diğer bir örnek olarak iki boyutlu listeyi tek boyutlu bir listeye dönüştürme işlemi şöyle yapılabilir.

```
matris = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
düz_liste = [eleman for satir in matris for eleman in satir]

print(düz_liste)
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

List Anlama ve if-else kullanımı

Koşullu ifadeleri, hem eleman seçiminde (if) hem de eleman üzerinde işlem yaparken (if-else) kullanabiliriz. Örneğin belirli sayıların çift ve tek olup olmalarına göre farklı işlemler yapılabilir.

```
sayilar = [1, 2, 3, 4, 5]
islenmis = [x*2 if x % 2 == 0 else x*3 for x in sayilar]

print(islenmis)
[3, 4, 9, 8, 15]
```

Sözlük (Dictionary) Veri Yapısı

Python'daki sözlük veri yapısı, anahtar-değer (key-value) çiftlerini saklayan bir veri yapısıdır. Dictionary, veri aramayı hızlı hale getiren ve esnek bir şekilde veri depolamanıza olanak tanıyan güçlü bir yapıdır. Python'da sözlükler süslü parantezler **"{ }"** içinde tanımlanır ve anahtar-değer çiftleri virgülle ayrılır.

Bir sözlük veri yapısının genel özellikleri:

1. **Anahtarlar (Keys) benzersizdir:** Aynı anahtar birden fazla kullanılamaz. Eğer aynı anahtar yeniden kullanırsa son değeri kabul edilir.
2. **Değerler (Values) değiştirilebilir:** Dictionary içinde bulunan değerler değiştirilebilir (mutable), yani bir değeri tekrar değiştirebilirsiniz.
3. **Sırasız veri yapısıydı (Python 3.6'dan Önce):** Python 3.6 ve sonrasında, sözlükler ekleme sırasını korur, ancak bu özellik daha eski sürümlerde garanti edilmezdi.
4. **Anahtarlar değiştirilemez olmalıdır:** Anahtarlar, değiştirilemez (immutable) türde değişken olmalıdır. Örneğin, metinler (str), sayılar (int, float) ve tuple'lar anahtar olarak kullanılabilir, ancak listeler (list) veya diğer sözlükler bunun için kullanılamaz.

Bir sözlük veri yapısının kullanım alanları:

- **Veriyi yapılandırılmış olarak saklama:** Verileri daha anlamlı ve yapılandırılmış bir şekilde saklamanıza olanak tanır.
- **Hızlı veri erişimi ve arama işlemleri:** Anahtarları kullanarak verilere hızlı bir şekilde erişim sağlanır.
- **Yapılandırma ayarları depolama:** Bir programın yapılandırma (config) ayarlarını depolamak için sözlükler kullanılabilir.
- **Veritabanı benzeri yapılar:** Küçük ölçekli veri depolama ve erişim ihtiyaçları için sözlükler, bir veritabanı olarak ayarlanabilir.

Sözlük veri yapısı kullanılan metotlar

Python'da sözlükler üzerinde çeşitli işlemler yapmamıza olanak sağlayan birçok yerleşik metot vardır. Bunlar şu şekilde özetlenebilir.

1. **clear():** Sözlüğün içindeki tüm öğeleri siler ve boş bir sözlük bırakır. Örnek: mdict = {'a': 1, 'b': 2}, mdict.clear() # {}
2. **copy():** Sözlüğün yüzeysel bir kopyasını (shallow copy) döner. Örnek: new_dict = mdict.copy()
3. **fromkeys(keys, value=None):** Belirtilen anahtarlarla ve isteğe bağlı bir varsayılan değerle yeni bir sözlük oluşturur. Örnek: keys = ['a', 'b', 'c']; new_dict = dict.fromkeys(keys, 0) # {'a': 0, 'b': 0, 'c': 0}
4. **get(key, default=None):** Belirtilen anahtarın değerini döner. Anahtar yoksa, belirtilen varsayılan değeri döner. Örnek: value = my_dict.get('b') # 2
5. **items():** Sözlükteki tüm anahtar-değer çiftlerini (key-value pairs) bir liste olarak döner. Örnek: mdict.items() # [('a', 1), ('b', 2)]
6. **keys():** Sözlükteki tüm anahtarları bir liste olarak döner. Örnek: mdict.keys())) # ['a', 'b']
7. **values():** Sözlükteki tüm değerleri bir liste olarak döner. Örnek: mdict.values())) # [1, 2]
8. **pop(key, default=None):** Belirtilen anahtarı sözlükten kaldırır ve o anahtarın değerini döner. Anahtar yoksa, varsayılan değeri döner. Örnek: sil = mdict.pop('a') # {'b': 2}
9. **popitem():** Sözlükten rastgele bir anahtar-değer çiftini kaldırır ve bu çifti bir tuple olarak döner. (Python 3.7 ve sonrası için, son eklenen öğeyi kaldırır.) Örnek: sil = mdict.popitem() # {'a': 1}
10. **setdefault(key, default=None):** Belirtilen anahtarın değerini döner. Eğer anahtar sözlükte yoksa, anahtarı belirtilen varsayılan değerle ekler. Örnek: mdict = {'a': 1}, mdict.setdefault('b', 2) # {'a': 1, 'b': 2}
11. **update([other]):** Başka bir sözlük veya anahtar-değer çiftlerini sözlüğe ekler veya günceller. Örnek: mdict = {'a': 1}, mdict. .update({'b': 2, 'c': 3}) # {'a': 1, 'b': 2, 'c': 3}

Sözlük veri yapısı bilgisi

Dictionary ya da sözlük veri yapısı, adından da anlaşılacağı üzere anahtar kelimelere değerlerin atandığı sözlük benzeri yapılardır. Diğer dillerde çok fazla rastlanmasa da sözlükler Python'da en fazla kullanılan veri yapılarındandır. Nasıl ki bir sözlükte her sözcüğün karşısında bir açıklama yazıyorsa **dict** veri yapısında da her bir anahtar kelimenin karşısında yazan bir değer vardır. Diğer bir deyişle, **dictionary**, her bir değere bağlı bir anahtar kelimenin olduğu veri türüdür. Anahtarlar harf ve rakamlardan oluşan alfa nümerik türde veriler iken değerler herhangi bir veri tipinde olabilir. Hatta **anahtar:değer** ikilisindeki değer bir başka sözlük de olabilir. Sözlüklerin **anahtar:değer** ikilileri ayrıca birer tuple veri nesnesi oluşturur. Sözlük veri yapısının avantajı bu tür veri yapıları ile çok hızlı işlem yapılabilmesidir.

Sözlükler veri yapısında küme parantezleri içinde, anahtar kelime eğer metin ise iki tırnak işareti içinde, sayısal ise normal yazılıp iki noktadan sonra anahtar kelimenin alacağı değer yazılır. Farklı anahtar-değer çitleri ise birbirlerinden virgülle ayrılır. Sözlüklerde kullanılabilecek veri türleri, hem anahtarlar hem de değerler için farklı kurallar ve sınırlamalara tabidir.

Anahtarlar, sözlükte benzersiz olmalıdır ve değiştirilemez bir veri türü olmalıdır. Bu özellik, anahtarların sözlükte hızlı bir şekilde arama yapılabilmesi için gereklidir.

Tamsayılar (int): {1: "Bir", 2: "İki"} gibi.
Ondalık Sayılar (float): {1.5: "Bir buçuk", 2.0: "İki"} gibi.
Metinler (str): {"ad": "Ali", "yaş": 25} gibi.
Tuple: {("x", "y"): 10} gibi.
Boole (bool): {True: "Doğru", False: "Yanlış"} gibi.

Not: Listeler ve sözlükler değiştirilebilir veri türleri olduklarından anahtar olarak kullanılamaz.

Değerler için herhangi bir sınırlama yoktur. Yani, Python'da desteklenen herhangi bir veri türü değer olarak kullanılabilir.

Tamsayılar (int): {"bir": 1, "iki": 2} gibi.
Ondalık Sayılar (float): {"a": 3.14, "b": 1.618} gibi.
Metinler (str): {"isim": "Ayşe", "şehir": "İstanbul"} gibi.
Listeler (list): {"rakamlar": [1, 2, 3], "harfler": ["a", "b", "c"]}.
Tuple: {"koordinatlar": (10, 20), "zaman": (12, 30)}.
Sözlükler (dict): {"öğrenci": {"isim": "Mehmet", "yaş": 20}}.
Set (Küme): {"eşsiz_rakamlar": {1, 2, 3}}.
Boole (bool): {"doğru_mu": True, "yanlış_mı": False}.
Nesneler (Object): Her hangi bir class nesnesi.

Sözlük veri yapısı bilgisi

Bir sözlük veri tipi örneği olarak belirli firmaların hisse senedi fiyatlarının olduğu bir yapıyı göz önüne alalım. Burada firma isimleri anahtar hisse senedi fiyatı değer durumunda olacaktır. Buna göre oluşturulan sözlük veri yapısı şöyle olacaktır.

```
hisseler = {"firma1":17.70, "firma3":28.70, "firma5":7.05, "firma7":13.56}
print(hisseler)
{'firma1':17.70, 'firma3':28.70, 'firma5':7.05, 'firma7':13.56}
```

Bir sözlükteki anahtarlara, değerlere ya da her ikisine ulaşmak için Python'da farklı metotlar bulunmaktadır.

```
print(hisseler.keys())
['firma1', 'firma3', 'firma5', 'firma7']

print(hisseler.values())
[17.7, 28.7, 7.05, 13.56]

print(hisseler.items())
[('firma1', 17.7), ('firma3', 28.7), ('firma5', 7.05), ('firma7', 13.56)}]
```

Çıktılarda görüleceği üzere **keys()**, **values()** ve **items()** metotları birer liste oluşturmaktadır. Bu ise verilerin **for** döngüsü kullanılarak kolaylık işlenmesine olanak vermektedir. Şöyle ki;

```
for firma in hisseler.keys():
    print(f"Firma adı: {firma}")
```

```
for firma, hisse in hisseler.items():
    print(f"Firma adı: {firma}, hisse fiyatı: {hisse}")
```

Sözlük oluşturmak

Bir sözlük oluşturmanın birkaç farklı yolu bulunmaktadır. Bunlara birer örnek olarak;

```
# Boş bir sözlük oluşturma
sözlük = {}
```

```
# dict() fonksiyonu ile oluşturma
ürün = dict(ad="Elma", "fiyat"=3.5, stok=50)
```

```
# Anahtar-değer çiftleriyle dolu bir sözlük oluşturma
öğrenci = {"isim": "Ali", "yas": 20, "bolum": "Bilgisayar Mühendisliği"}
```

Oluşturulan bu veri tiplerinin gerçekten bir sözlük olup olmadığı kontrol edelim. Bunun için **type()** fonksiyonunu kullanmamız gerekmektedir.

```
print(sözlük)
print(type(sözlük))

{}
<class 'dict'>
```

```
print(ürün)
print(type(ürün))

{'ad': 'Elma', 'fiyat': 3.5, 'stok': 50}
<class 'dict'>
```

```
print(öğrenci)
print(type(öğrenci))

{'isim': 'Ali', 'yas': 20, 'bolum': 'Bilgisayar Mühendisliği'}
<class 'dict'>
```

Buradan görüleceği üzere Python dilinde sözlük tipi verileri "dict" olarak isimlendirilmektedir. Bir sözlükteki eleman sayısını len() fonksiyonu ile elde edebiliyoruz. Burada bunu denediğimizde len(sözlük) 0, len(ürün) 3 ve len(öğrenci) 3 sayılarını elde ederiz.

Sözlük elemanlarına erişmek

Bir sözlükteki elemanlarına erişmenin genel yazım düzeni şu şekildedir:

sözlük_nesnesi [anahtar_adı]

Sözlükler anahtar:değer çiftlerinde oluştukların için önemli parametre "anahtar" öğesidir. Çünkü sözlükte esas amaç **"anahtar"** öğesiyle onun işaret ettiği **"değer"** öğesine erişmektir. Örneğin Türkçe'den İngilizce'ye kelime çevirisi yapmayı sağlayan bir sözlüğü nesnemiz olduğunu düşünelim.

```
sözlük = {
   "kitap"      : "book",
   "bilgisayar" : "computer",
   "programlama": "programming",
   "dil"        : "language",
   "defter"     : "nootebook"
}
```

Şimdi mesela bu sözlükteki "kitap" adlı elemanın değerini öğrenmek istediğimiz düşünelim. Yukarıdaki yazım düzenine göre ayarlama yaptığımızda aşağıdaki çıktıları alırız.

```
print(sözlük["kitap"])

book
```

```
print(sözlük["bilgisayar"])

computer
```

Sözlük içindeki alt elemanlara erişim

Alt eleman kavramı, bir sözlüğün değer verileri bir liste ya da başka bir sözlük gibi karmaşık veri yapıları içerdiğinde geçerlidir. Bu tür durumlarda, sözlüğün değeri basit bir veri türü (örneğin bir sayı veya metin) değil, birden fazla alt veriye sahip bir yapı olur. Listeler ve sözlükler gibi veri yapılarının kendi içlerinde de elemanları bulunduğu için, bu elemanlara erişmek için farklı yollar kullanmak gerekir.

Eğer sözlüğün değeri başka bir sözlükse, ana sözlüğe erişim için birinci düzeyde anahtar kullanılırken, içteki alt sözlüğün elemanlarına erişmek için ikinci bir anahtar kullanılması gerekmektedir. Bu durumda, iç içe geçmiş sözlüklere kolayca ulaşabilmek için her iki seviyedeki anahtarlar sırasıyla belirtilir. Eğer ana sözlüğün değeri bir liste içeriyorsa, liste elemanlarına erişmek için indeks numaraları kullanılır.

Bir firmada çalışan kişilere ait doğum yeri, görevleri ve yaşları gibi belirli bilgilerin olduğu bir veri yapısını dikkate alalım. Bu veri yapısında kişi isimleri anahtar diğer bilgiler değer verisi olacaktır. Bir'den fazla değer verisi olması sebebiyle liste ya da sözlük tipi yapı kullanılması gereklidir. Buna göre alt elemanlara erişim farklı olacaktır.

```
kişi_liste = {
   "Ahmet Civelek": ["İstanbul", "Öğretmen", 34],
   "Mehmet Yağcı" : ["Adana",    "Mühendis", 40],
   "Seda Altaylı" : ["Malatya",    "Doktor", 30]
}
```

```
print(kişi_liste["Ahmet Civelek"][1])

Öğretmen
```

```
kişi_sözlük = {
   "Ahmet Civelek": {"Memleket": "İstanbul", "Meslek": "Öğretmen", "Yaş": 34},
   "Mehmet Yağcı" : {"Memleket": "Adana",    "Meslek": "Mühendis", "Yaş": 40},
   "Seda Altaylı" : {"Memleket": "Malatya",  "Meslek": "Doktor",   "Yaş": 30}
}
```

```
print(kişi_sözlük["Seda Altaylı"]["Yaş"])

40
```

Sözlük veri yapısının kullanımı – örnek

Küçük ölçekli veri depolama ve erişim ihtiyaçları için sözlükler, bir veritabanı gibi kullanılabilir. Özellikle sabit ve basit veri yapılarını depolarken oldukça etkili olabilir. Bunun için bir örnek olarak küçük bir şirkette her bir çalışanın adını, görevini, maaşını ve görev süresini gösteren bir mini veritabanı oluşturalım. Biz burada çalışanın adıyla diğer bilgilere erişmek istediğimiz varsayacağız. Buna göre iki ayrı sözlük yapısı kullanılması gereklidir.

```
calisan = {
       "Ali":101,
      "Ayşe":102,
    "Mehmet":103,
     "Fatma":104,
     "Samet":105,
     "Hülya":106
}
```

```
bilgiler = {
    101: {"pozisyon": "Müdür"     , "maas": 7000, "sure": 20},
    102: {"pozisyon": "Mühendis"  , "maas": 6500, "sure": 14},
    103: {"pozisyon": "Teknisyen" , "maas": 5500, "sure": 18},
    104: {"pozisyon": "Ustabaşı"  , "maas": 3000, "sure":  7},
    105: {"pozisyon": "Muhasebesi", "maas": 6000, "sure": 25},
    106: {"pozisyon": "Temizlikçi", "maas": 3500, "sure":  4}
}
```

Şimdi verilen isme göre bu kişiye ait bilgileri yazan kodu yazalım.

```
kişi = input("Kişinin adını giriniz: ")

if kisi in calisan:
    kisi_kod = calisan[kisi]
    pozisyon, maas, sure = bilgiler[kisi_kod].values()

    print(f"Adı: {kisi}\nPozisyonu: {pozisyon}\nÇalışma süresi: {sure} yıl\nMaaşı: {maas} TL")

else:
    print(f"{kisi} isimli kişi listede yoktur.")
```

```
Adı: Samet
Pozisyonu: Muhasebesi
Çalışma süresi: 25 yıl
Maaşı: 4000 TL
```

Sözlük veri yapısının kullanımı – nesne örneği

Sözlüklerde değer olarak nesnelerin kullanılması karmaşık veri yapılarının oluşturulmasına imkan vermektedir. Bu sözlüklerin oldukça efektif ve esnek bir araç olmasını sağlar. Aşağıda, bir sınıf tanımlanıp bu sınıftan oluşturulmuş nesnelerin bir sözlükte değer olarak kullanıldığı bir örnek verilmiştir.

Bir sınıf tanımlayalım

```
class Ogrenci:
    def __init__(self, isim, yas, bolum):
        self.isim = isim
        self.yas = yas
        self.bolum = bolum

    def bilgi_goster(self):
        return f"İsim: {self.isim}, Yaş: {self.yas}, Bölüm: {self.bolum}"
```

Şimdi **Ogrenci** sınıfından nesneler oluşturalım ve öğrencileri bir sözlükte anahtar-değer çifti olarak saklayalım. Daha sonra sözlükteki nesnelere erişelim ve bilgiler terminale yazdıralım.

```
ogrenci1 = Ogrenci("Ali", 21, "Bilgisayar Mühendisliği")
ogrenci2 = Ogrenci("Ayşe", 22, "Makine Mühendisliği")
ogrenci3 = Ogrenci("Mehmet", 20, "Elektrik-Elektronik Mühendisliği")

# sözlükte nesneleri ayarlayalım
ogrenci_sozluk = {
    "ogrenci1": ogrenci1,
    "ogrenci2": ogrenci2,
    "ogrenci3": ogrenci3
}

# Sözlükteki bilgiler gösterelim
for anahtar, ogrenci in ogrenci_sozluk.items():
    print(f"{anahtar} -> {ogrenci.bilgi_goster()}")
```

```
ogrenci1 -> İsim: Ali, Yaş: 21, Bölüm: Bilgisayar Mühendisliği
ogrenci2 -> İsim: Ayşe, Yaş: 22, Bölüm: Makine Mühendisliği
ogrenci3 -> İsim: Mehmet, Yaş: 20, Bölüm: Elektrik-Elektronik Mühendisliği
```

Set (Küme) Veri Yapısı

Python'da set (küme), benzersiz ve sırasız elemanlar koleksiyonunu tutmak için kullanılan bir veri yapısıdır. set veri yapısı, matematikteki kümelere benzer şekilde çalışır ve tekrarlanan elemanları otomatik olarak filtreler. Setler, verilerin hızlı bir şekilde saklanması, karşılaştırılması ve kesişim gibi kümeler arası işlemler için kullanılır.

Bir set'in genel özellikleri:

1. **Sırasızdır (Unordered):** Setlerde elemanlar belirli bir sıraya göre saklanmaz. Bu nedenle, elemanlara indeksle erişilemez.
2. **Benzersiz Elemanlar (Unique Elements):** Set içerisinde aynı elemandan birden fazla bulunamaz. Her eleman sadece bir kez yer alır.
3. **Değiştirilebilir (Mutable):** Sete yeni elemanlar eklenebilir veya elemanlar çıkarılabilir. Ancak setin elemanları sırasız olduğundan, belirli bir pozisyona eleman ekleme veya çıkarma işlemi yapılamaz.
4. **Tekrar Edilebilir (Iterable):** Set üzerinde döngü (for döngüsü gibi) kullanarak işlem yapılabilir.

Bir set'in kullanım alanları:

- **Tekrarlardan Kurtulma:** Bir veri kümesindeki tekrarlanan elemanları hızlıca filtrelemek için kullanılır.
- **Hızlı Eleman Kontrolü:** Setler, eleman içerip içermediğini kontrol etmek için listelerden daha hızlıdır.
- **Kümeler Arası İşlemler:** Matematiksel kümeler üzerinde işlem yapmak için kullanılır (birleşim, kesişim vb.).

Set avantajları:

- **Hızlı Arama:** Elemanların varlığı setlerde çok hızlı bir şekilde kontrol edilebilir.
- **Benzersizlik:** Setler, aynı elemanın birden fazla yer almasını engeller.
- **Kapsamlı Kümeler Arası İşlemler:** Matematiksel kümeler arası işlemler için çok uygun bir yapıdır.

Set dezavantajları:

- **Sırasızlık:** Elemanların sırası korunmaz, bu yüzden indeksleme yapılamaz.
- **Değiştirilemez Elemanlar:** Setin içine yerleştirilecek elemanlar değiştirilebilir olmamalıdır (örneğin, bir set içinde list olamaz).

Set veri yapısı kullanılan metotlar

Python'da set veri yapısı üzerinde kullanılabilecek birçok yerleşik metot vardır. Bunlar şu şekilde özetlenebilir.

1. **add(x):** Sete bir eleman ekler. Eğer eleman zaten varsa, ekleme yapılmaz. Örnek: sayilar = {1, 2, 3}; sayilar.add(4) # {1, 2, 3, 4}
2. **remove(x):** Setten belirtilen elemanı kaldırır. Eğer eleman yoksa hata verir. Örnek: sayilar.remove(3) # {1, 2, 4}
3. **discard(x):** Setten belirtilen elemanı kaldırır. Eğer eleman yoksa hata vermez. Örnek: sayilar.discard(5) # {1, 2, 4} - Hata vermez
4. **pop():** Setten rastgele bir elemanı kaldırır ve döndürür. Eğer set boşsa hata verir. Örnek: eleman = sayilar.pop() # Rastgele bir eleman çıkarılır
5. **clear():** Setin tüm elemanlarını siler. Örnek: sayilar.clear() # Boş set: set()
6. **union(set):** İki setin birleşimini döndürür. Örnek: A = {1, 2, 3}; B = {3, 4, 5}; birlesim = A.union(B) # {1, 2, 3, 4, 5}
7. **intersection(set):** İki setin kesişimini döndürür. Örnek: kesisim = A.intersection(B) # {3}
8. **difference(set):** Bir setin diğer setten farkını döndürür. Örnek: fark = A.difference(B) # {1, 2}
9. **issubset(set):** Bir setin, diğer bir setin alt kümesi olup olmadığını kontrol eder. Örnek: C = {1, 2}; sonuc = C.issubset(A) # True
10. **issuperset(set):** Bir setin, diğer bir setin üst kümesi olup olmadığını kontrol eder. Örnek: sonuc = A.issuperset(C) # True

Set (Küme) Veri Yapısı

Veri nesnelerini herhangi bir sıralama olmaksızın ve her nesneden sadece bir tane olacak şekilde kaydetmek istersek **set (küme)** veri yapısını kullanabiliriz. Set veri yapısı adını matematikteki küme tanımından almıştır. Tıpkı matematikteki küme tanımı gibi bu veri yapılarında her veriden sadece bir tane bulunur ve verilerin sırası önemli değildir. Setler de listeler gibi sonradan değiştirilebilir (mutable) verilerdir. Set (küme) veri yapısı oluşturmak için **set()** fonksiyonu kullanılır. Örneğin, bir alışveriş listesini bir set veri yapısı olarak kaydedildiğini varsayalım.

```
alisveris_listesi = ['ekmek', 'süt', 'yoğurt', 'pirinç', 'bal', 'meyve', 'ekmek', 'süt']
alisveris = set(alisveris_listesi)

print(alisveris)
{'süt', 'pirinç', 'bal', 'meyve', 'yoğurt', 'ekmek'}
```

Gördüğünüz gibi orijinal listede ekmek ve `süt` ve `ekmek` iki kez yazdığınız halde **set()** fonksiyonu ile oluşturduğunuz küme verisinde sadece birer kez yazılıyor.

Bir kümeye yeni bir eleman eklemek için **.add()** metodu kullanılır. Eğer kümede olan bir veri eklenmek istenirse küme içeriğinde herhangi bir değişiklik meydana gelmez. Kümeye, birden fazla eleman eklemek için **.update()** metodu kullanılır. Bu metotla kümeye bir liste iletilir. Listeden bir eleman çıkarmak için ise **.discard()** metodu kullandır.

```
yeni_malzemeler = ['sebze', 'gazete', 'dergi']
alisveris.update(yeni_malzemeler)

print(alisveris)
{'süt', 'pirinç', 'bal', 'meyve', 'yoğurt',
'ekmek', 'sebze', 'gazete', 'dergi'}
```

```
alisveris.add('deterjan'); print(alisveris)
{'süt', 'pirinç', 'bal', 'meyve', 'yoğurt',
'ekmek', 'sebze', 'gazete', 'dergi', 'deterjan'}

alisveris.discard('ekmek') ; print(alisveris)
{'süt', 'pirinç', 'bal', 'meyve', 'yoğurt',
'sebze', 'gazete', 'dergi', 'deterjan'}
```

Set metotlar - örnekler

Kümeden rastgele bir eleman seçip onu listeden silmek için almak **.pop()** fonksiyonu kullanılır. Her kullanımda farklı bir eleman seçilmektedir. Ayrıca iki kümenin birleşimi, kesişimi ve farkı için **.union()**, **.intersection()** ve **.difîerence()** metotları kullanılır.

```
alisveris_listesi = ['süt', 'pirinç', 'bal', 'meyve', 'yoğurt', 'ekmek']
alisveris = set(alisveris_listesi)
```

```
urun = alisveris.pop()  # ilk deneme
print(urun)
yoğurt

print(alisveris)
{'süt', 'pirinç', 'bal', 'meyve', 'ekmek'}
```

```
urun = alisveris.pop()   # ikinci deneme
print(urun)
süt

print(alisveris)
{'pirinç', 'bal', 'meyve', 'ekmek'}
```

```
alisveris1 = set(['ekmek', 'süt', 'yoğurt', 'peynir'])
alisveris2 = set(['bal', 'şeker', 'tuz', 'ekmek', 'süt'])

print(alisveris1.union(alisveris2))
{'şeker', 'süt', 'peynir', 'yoğurt', 'tuz', 'ekmek', 'bal'}

print(alisveris1.intersection(alisveris2))
{'ekmek', 'süt'}

print(alisveris2.difference(alisveris1))
{'tuz', 'bal', 'şeker'}
```