

17 Ocak 2018

Bilgisayar Bilimi

Kur1 Ders Notları

Veri Yapıları

Ahmet Tuna POTUR

Veri Yapıları.....	1
1. None Veri Tipi.....	1
2. Bool Veri Tipi.....	1
2.1. <i>bool()</i> fonksiyonu	1
3. Liste (list) Veri Tipi.....	2
3.1. <i>Listeleri Oluşturmak ve list() Fonksiyonu.....</i>	2
3.2. <i>Liste Elemanlarına Erişmek (İndeks)</i>	2
3.3. <i>len() Fonksiyonu</i>	4
3.4. <i>Liste Elemanlarının Değerlerini Değiştirme(Mutable).....</i>	4
3.5. <i>İndeks Sınırlarına Çıkmak, Hatalı İndeks Değeri.....</i>	5
3.6. <i>Liste Metotları.....</i>	6
3.6.1. <i>append() Metodu</i>	6
3.6.2. <i>pop() Metodu</i>	6
3.6.3. <i>sort() Metodu ve sorted() Fonksiyonu.....</i>	7
3.6.4. <i>extend() Metodu</i>	9
3.6.5. <i>insert() Metodu</i>	10
3.6.6. <i>remove() Metodu</i>	11
3.6.7. <i>index() Metodu.....</i>	12
3.6.8. <i>count() Metodu</i>	12
3.6.9. <i>clear() Metodu.....</i>	13
3.6.10. <i>copy() Metodu</i>	13
3.6.11. <i>reverse() Metodu</i>	14
3.7. <i>Listelere Eleman Ekleme (Liste Birleşimi)</i>	15
3.8. <i>Listeleri Çarparak Tekrar Ettirme</i>	15
3.9. <i>Dilimleme</i>	16
3.10. <i>Liste Verileri Üzerinde Doğrudan Çalışmak</i>	18
3.11. <i>in Operatörü.....</i>	19
3.12. <i>range() Fonksiyonu</i>	19
3.13. <i>İç İççe Listeler.....</i>	20
4. Demetler (tuple) Veri Tipi.....	21
4.1. <i>Demetleri Oluşturmak ve tuple() Fonksiyonu.....</i>	21
4.2. <i>Demet Elemanlarına Erişmek(İndeks)</i>	22
4.3. <i>len() Fonksiyonu</i>	22

4.4. Demetler Değişirilemez(Immutable).....	22
4.5. İndeks Sınırlarına Çıkmak, Hatalı İndeks Değeri.....	23
4.6. Demet Metotları	23
4.6.1. count() Metodu	23
4.6.2. index() Metodu.....	23
4.7. Demetlere Eleman Ekleme(Demet Birleşimi)	23
4.8. Demetleri Çarparak Tekrar Ettirme.....	24
4.9. Dilimleme	24
4.10. Demet Verileri Üzerinde Doğrudan Çalışmak.....	24
4.11. in Operatörü.....	25
4.12. range() Fonksiyonu	25
4.13. İç İç Demetler	25
4.14. Sekans Açma (sequence unpacking)	25
4.15. Demetlerin Kullanım Alanları	26
4.15.1. Fonksiyonlara Belirsiz Sayıda Argüman Girmek.....	26
4.15.2. Format Metodu İle Demetlerin Kullanımı.....	27
5. Karakter Dizileri (string) Veri Tipi.....	28
5.1. Karakter Dizilerini Oluşturmak ve str() Fonksiyonu	28
5.2. Karakter Dizisi Elemanlarına Erişmek (İndeks).....	29
5.3. len() Fonksiyonu	30
5.4. Karakter Dizileri Değişirilemez(Immutable).....	30
5.5. İndeks Sınırlarına Çıkmak, Hatalı İndeks Değeri.....	30
5.6. Karakter Metotları	30
5.6.1. format() Metodu	30
5.6.2. zfill() Metodu	36
5.6.3. replace() Metodu.....	37
5.6.4. index(), find(), rindex() ve rfind() Metotları.....	37
5.6.5. join() Metodu	39
5.6.6. split() Metodu.....	39
5.6.7. strip(),rstrip() ve lstrip() Metodu	39
5.6.8. upper() ve isupper() Metotları.....	40
5.6.9. lower() ve islower() Metotları	40
5.6.10. title() ve istitle() Metotları.....	41
5.6.11. capitalize() Metodu	41
5.6.12. swapcase() Metodu	41

5.6.13. isspace() Metodu	41
5.6.14. center(), rjust() ve ljust() Metotları	41
5.6.15. count() Metodu	42
5.6.16. isnumeric() Metodu	42
5.6.17. endswith() ve startswith() Metotları	42
5.7. Karakter Dizilerine Eleman Ekleme (Karakter Birleşimi)	43
5.8. Karakter Dizilerini Çarparak Tekrar Ettirme	43
5.9. Dilimleme	44
5.10. Karakter Dizileri Üzerinde Doğrudan Çalışmak	44
5.11. in Operatörü	44
5.12. Kaçış Dizileri (Özel Karakterler)	45
5.12.1. Satır Başı (\n)	45
5.12.2. Satır Başı Tab (\t)	45
5.12.3. Ters Bölü (\)	45
5.13. print() Fonksiyonu	46
5.13.1. print() Fonksiyonunun Parametreleri	47

Veri Yapıları

1. None Veri Tipi

```
>>> boş = None
>>> print(baş)
None
>>> type(baş)
<class 'NoneType'>
```

```
>>> boş = 5
>>> print(baş)
5
>>> type(baş)
<class 'int'>
```

Eğer bir değişkenin değerini sonradan belirlemek isterseniz bu değişken **None** (atanmamış anlamında) değerine eşitleyebilirsiniz. **None** tipiyle oluşturulan değişkeni sonradan değer atayıp kullanabiliriz.

2. Bool Veri Tipi

```
>>> doğru = True
>>> type(doğru)
<class 'bool'>
```

```
>>> yanlış = False
>>> type(yanlış)
<class 'bool'>
```

İngilizcede Boolean olarak geçen **bool** veri tipi sadece **True** ve **False** olarak iki değerden oluşur.

2.1. bool() fonksiyonu

```
>>> bool(1)
True
>>> bool(375)
True
>>> bool(-57)
True
>>> a = 0.000001
>>> bool(a)
True
>>> bool(0)
False
>>> z = 0
>>> bool(z)
False
```

```
>>> bool(3.14)
True
>>> bool(-12.25)
True
>>> bool(-0.000001)
True
>>> b = 5.41
>>> bool(b)
True
>>> bool(0.0)
False
>>> y = 0.0
>>> bool(y)
False
```

bool() fonksiyonu değer olarak aldığı sayı verisini **bool** değere dönüştürür. Python'da **0** dışındaki tüm sayı değerleri **True** değerine karşılık gelir, **0** sayı değeri **False** değerine karşılık gelir. Üstteki **bool()** fonksiyonu örnekleri incelendiğinde **bool(0)** ve **bool(0.0)** tip dönüşümleri **False** değerini döner, bunun dışında tüm sayısal dönüşümler **True** değere dönüşür.

```
>>> 17 < 2043
True
>>> "Tuna" == "Tuna"
True
```

```
>>> 17 > 2043
False
>>> "Ahmet" == "Tuna"
False
```

bool veri tipi karşılaştırma operatöründen sonra ortaya çıkan sonuç değeridir. Karşılaştırma operatörleri sonucun yanlış çıktığı durumda **False**, doğru çıktığı durumda **True** değerini döner. **bool** veri tipi ileride göreceğimiz koşullu durumlar ve döngüler konularında kullanılır.

3. Liste (list) Veri Tipi

Liste, sıralı bir dizidir. Listeler bir küme gibi, elemanlardan (bazen öge veya terim de denir) oluşur. Elemanların sayısına **listenin uzunluğu** denir. Kümenin aksine sıralı ve aynı öğeler dizide farklı konumlarda birkaç kez bulunabilir. Örneğin, **Fibonacci dizisi**, her sayının kendine önceki sayı ile toplanması sonucu oluşan bir sayı dizisidir. İlk iki öge 1 ile 1'dir. Böylece 9 elemanlı [1,1,2,3,5,8,13,21,34] dizisi elde edilir. [P,Y,T,H,O,N] ilk harfi 'K' ve son harfi 'P' olan bir listedir.

Listeler çok yararlı bir veri tipidir, indekslenirler, parçalanırlar ve üzerinde değişik işlemler yapabildiğimiz fonksiyonları barındırırlar. Bir listede her veri tipinden elemanı saklayabiliriz. Listeler her zaman birden çok eleman taşıdığından listelerin isimlerini 'veriler', 'insanlar', 'çalışanlar' gibi çoğul isim olarak vermek gerekir. Python'da kareli parantez '[']' içerisinde listeleri barındırır. Listelerin içindeki elemanlar virgülle birbirinden ayrılır.

3.1. Listeleri Oluşturmak ve list() Fonksiyonu

```
>>> # Boş Liste
>>> liste_1 = list()
>>> liste_2 = []
>>> # Sadece int sayılar listesi
>>> liste_3 = [1,1,2,3,5,8,13,21,34]
>>> # değişik tipte veriler listesi
>>> liste_4 = [5,3.14,"Tuna"]

>>> # Boş Liste
>>> print(liste_1, liste_2)
[] []
>>> liste_3
[1, 1, 2, 3, 5, 8, 13, 21, 34]
>>> liste_4
[5, 3.14, "Tuna"]
```

list() fonksiyonu liste tipinde değişkenleri oluşturmak ve karşılığı olan tipleri **list** tipine çevirmek için kullanılır.

Listeleri değer vermeden boş olarak veya değer vererek dolu olarak oluşturabiliriz.

liste_1 listesi: **list()** fonksiyonuyla **liste_1 = list()** atamasıyla **boş liste**.

liste_2 listesi: **[]** boş liste operatörüyle **liste_2 = []** atamasıyla **boş liste**

liste_3 listesi: **liste_3 = [1,1,2,3,4,8,13,21,34]** atamasıyla **int** değerlerden oluşan Fibonacci listesi

liste_4 listesi: **liste_4 = [5,3.14,"Tuna"]** atamasıyla **değişik veri tiplerinden** oluşan bir liste

oluşturulmuştur.

```
>>> # type() fonksiyonu listenin tipini list olarak gösterir
>>> veriler = [17,3.14,"Ahmet"]
>>> type(veriler)
<class 'list'>
```

Liste tipinde değişkenler **type()** fonksiyonuyla incelenir. **type()** fonksiyonu listelerin tipleri için 'list' tipini döner.

```
>>> # string değer listeye çevriliyor
>>> selam = "Merhaba Dünya"
>>> liste_selam = list(selam)
>>> liste_selam
['M', 'e', 'r', 'h', 'a', 'b', 'a', ' ', 'D', 'ü', 'n', 'y', 'a']
```

Karakter dizisi(string) **list()** fonksiyonuyla liste verisine dönüştürülebilir. Karakter dizisi listeye çevrildiğinde karakter dizisi içinde bulunan tüm harfler tek tek liste elemanı haline dönüşür.

3.2. Liste Elemanlarına Erişmek (İndeks)

veriler	=	[3,	3,	3.14,	-1.44,	"Tuna",	"Emel",	[10 , 20 , 30],	27.57,	39]
index	→		0	1	2	3	4	5	6	7	9	
			-9	-8	-7	-6	-5	-4	-3	-2	-1	

```

n elemanlı bir liste

a[0]          1. terim
a[1]          2. terim
a[2]          3. terim
⋮
⋮
a[n-1]        (n). terim

```

Liste elemanlarına erişmek için indeks operatörü([]) kullanılır. İndeks operatörü listeler için kullanılan değişken adından hemen sonra yazılan köşeli parantez([]) karakterleridir(veriler[index]). İndeks operatörü sadece listeler için kullanılmaz, listelere benzeyen **demet**, **karakter dizisi**, **sözlük** gibi veri tipleri için kullanılır. İndeks operatörünün kullanımı benzer veri tipleri için aynı listelerde olduğu gibidir. **a[indeks]** a listesinde, indeks operatörü([]) içerisindeki sayıya **indeks** denmektedir. [] içerisine girilen **indeks** değeri liste içerisinde ulaşılacak istenen değer için kullanılır. **Listelerin indeksleri 0 ile başlar**. 0, başlangıç indeksi olduğu için **n** elemanlı bir listenin eleman sayısı **n-1** olur.

```

>>> veriler = [1,2,3,4,5,6,7]
# indeks numarasıyla elemanı çağırmak. veriler[indeks] işlemi
>>> veriler[0]          >>> veriler[5]          >>> veriler[-1]
1                        6                        7
>>> veriler[1]          >>> veriler[6]          >>> veriler[-2]
2                        7                        6

```

veriler = [1,2,3,4,5,6,7] atanmasıyla 1'den 7'ye sayılardan oluşan bir liste oluşturduk. **veriler[indeks]** ifadesiyle liste elemanları içinde istenen elemana ulaşılır.

veriler[0] değeri listenin **ilk değeri** 1'i
veriler[1] değeri serideki baştan **ikinci değeri** 2'yi
veriler[5] değeri serideki baştan **altıncı değeri** 6'yı
veriler[6] değeri serideki baştan **yedinci değeri** 7'yi
veriler[-1] değeri, serideki **son elemanı** 7'yi.
veriler[-2] değeri **sondan bir önceki değeri** 6'yı ifade eder.

```

>>> veriler = ["Tuna", "Can", "Emel", "Oya", "Ahmet", "Mehmet"]
# değişken ile indeks numaralı elemanı çağırmak
>>> indeks = 0          >>> indeks = -1          >>> indeks = 0
>>> veriler[indeks]     >>> veriler[indeks]       >>> veriler[indeks+1]
'Tuna'                  'Mehmet'                  'Can'
>>> indeks = 3          >>> indeks = -4          indeks = -1
>>> veriler[indeks]     >>> veriler[indeks]       >>> veriler[indeks-1]
'Oya'                   'Emel'                   'Ahmet'

```

Liste elemanlarına erişmek için tam sayı değişkenleri kullanılabilir. Değişkenin değeri o anda ne ise o indeks numarasındaki eleman çağrılır.

```

# farklı tipte veriler içeren liste          # listenin tipi
>>> veriler = [5, 3.5, [10,20,30], "Tuna"]  >>> type(veriler)
<class 'list'>

# liste içinde bulunan elemanlar kendi veri tiplerindedir
>>> type(veriler[0])          >>> type(veriler[2])
<class 'int'>                <class 'list'>
>>> type(veriler[1])          >>> type(veriler[3])
<class 'float'>              <class 'str'>

```

veriler listesi **list** tipindedir. Liste içinde bulunan değerler çağrıldığında kod içerisinde çağrılan eleman kendi tipinde işlem görecektir. Üstteki örnekte görüldüğü gibi **type(veriler[0])** işlemi veriler listesinin ilk elemanın tipini **int** olarak döner.

3.3. len() Fonksiyonu

```
>>> liste_say = [1,2,3,4,5]      >>> liste_karisik = [7,1.22,"Can",[1,2,3]]
>>> len(liste_say)              >>> len(liste_karisik)
5                                4
```

Listenin eleman sayısına listenin uzunluğu denir. **len(liste)** fonksiyonu değer olarak aldığı listenin eleman sayısını verir. **len** İngilizcedeki uzunluk **length** kelimesinin kısaltmasıdır. **liste_karisik** listesinin içerisinde "**Can**" karakter dizisi ve **[1,2,3]** listesi birer eleman olarak sayıldığı için **liste_karisik** listesindeki eleman sayısı **4** olarak hesaplar.

```
>>> veriler = [0,1,2,3,4,5,6,7]
# eleman sayısı
>>> len(veriler)
8
# son elemanın indeks numarası
>>> veriler[len(veriler)-1]
7
# son eleman
>>> veriler[-1]
7
>>> veriler[7]
7

# Listenin Son Elemanı
>>> elemanSayisi = len(veriler)
>>> elemanSayisi
8
# eleman sayısından 1 çıkarsa
# son elemanın indeksi bulunur
>>> sonEleman=len(veriler)-1
>>> veriler[sonEleman]
7
```

len() fonksiyonu listelerin sayısını bulmak için kullanılıyordu. **len(veriler)-1** işlemi ile listenin eleman sayısının bir eksiği listedeki son elemanın indeks numarasını verir. **liste[len(veriler)-1]** işlemi listesindeki son elemanı getirir. **liste[-1]** işlemiyle de listenin son elemanını bulabilirsiniz.

3.4. Liste Elemanlarının Değerlerini Değiştirme(Mutable)

```
veriler = [1,2,3,4,5]
# indeks numarasıyla elemanı değiştirmek
>>> veriler[0]= 10      >>> veriler[2]= 30      >>> veriler[-1]= 50
>>> veriler             >>> veriler             >>> veriler
[10, 2, 3, 4, 5]      [10, 2, 30, 4, 5]      [10, 2, 30, 4, 50]
```

Listeler değiştirilebilen (**mutable**) bir veri tipidir. Dolayısıyla listeler üzerinde doğrudan değişiklik yapabiliriz. Listenin içerisindeki hangi elemanın değerini değiştirmek istiyorsanız indeks kullanarak istediğiniz elemanın değerini değiştirebilirsiniz. Yukarıdaki örnekte ilk elemanın, orta elemanın ve son elemanın değerleri on katlarıyla değiştirilmiş.

```
>>> veriler = ['Tuna', 'Can', 'Emel', 'Oya']

# dizinin ilk elemanı değiştiriliyor      # dizinin 3. elemanı değiştiriliyor
>>> veriler[0] = 'Fırat'                  >>> veriler[2] = 'Mert'
>>> veriler                               >>> veriler
['Fırat', 'Can', 'Emel', 'Oya']          ['Fırat', 'Can', 'Mert', 'Oya']

# dizinin eleman sayısının bir eksiği son elemanı değiştiriliyor
>>> indeks = len(veriler)
>>> veriler[indeks-1] = 'Emel'
>>> veriler
['Fırat', 'Can', 'Mert', 'Emel']
```

İndeks kullanarak içinde karakter dizileri barındıran listenin elemanları da değiştirilebilir.

```

>>> veriler = [0, 1, 2, 3, 4, 5]
# veriler listesinin ilk elemanı diğer elemanlar ile hesaplanarak değişiyor
>>> veriler[0] = (veriler[2] + veriler[4])*3
>>> veriler
[18, 1, 2, 3, 4, 5]
# veriler listesinin 4. Elemanı değişken ile yapılan hespla değişiyor
>>> x = 3
>>> veriler[4] = (x + 7)*5
>>> veriler
[18, 1, 2, 3, 50, 5]

```

Listelerin elemanları hesaplama yapılarak değiştirilebilir. Hatta listenin bir elemanı listenin diğer elemanları kullanılarak yapılan bir işlemin sonucuyla da değiştirilebilir.

```

# değişkenlere sırayla değerleri atanıyor
>>> a,b,c,d,e = 5,3.14,"Can","39",1981
# değişkenler kullanılarak veriler oluşturuluyor
>>> veriler = [a,b,c,d,e]
# veriler değişkenlerin değeriyle oluşturuluyor
>>> veriler
[5, 3.14, 'Can', '39', 1981]
>>> a
5
# değişkenin değerini değiştirmek listenin elemanlarını değiştirmez
>>> a = 19
>>> veriler
[5, 3.14, 'Can', '39', 1981]

```

Değişkenlerin taşıdıkları değerlerden bir liste oluşturmak mümkündür. Değişkenler ile liste oluşturulduğunda değişkenler sadece listeyi oluşturmak için kullanılır. Sonrasında değişkenlerin değerleri değişse bile listenin verileri değişmeyecektir. Yani değişken değerleriyle oluşan liste ve değişkenler ayrı verilerdir.

3.5. İndeks Sınırlarına Çıkmak, Hatalı İndeks Değeri

```

>>> veriler = [3,5,7,9,11]
# İndeks sınırlarına çıkılırsa hata oluşur
>>> veriler[17]
Traceback (most recent call last):
  File "<pyshell#11>", line 1, in <module>
    veriler[17]
IndexError: list index out of range
# indeks değeri tam sayı olmak zorunda
>>> veriler[3.14] # indeks float
Traceback (most recent call last):
  File "<pyshell#11>", line 1, in <module>
    veriler[3.14]
TypeError: list indices must be integers or slices, not float
>>> 8//2-2 # // ile sonuç int
2
>>> veriler[8//2-2]
7

>>> veriler[-7] = 27
Traceback (most recent call last):
  File "<pyshell#11>", line 1, in <module>
    veriler[-7] = 27
IndexError: list assignment index out of range

>>> 8/2-2
2.0 # sonuç float
>>> veriler[8/2-2] # indeks float
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    veriler[8/2-2]
TypeError: list indices must be integers or slices, not float

```

Eğer listeden bir eleman çağırmak için olmayan bir indeksi verilirse veya olmayan bir indeksteki elemana değer atanırsa hata çıkar. Listedten eleman çağırırken indeks sınırları dışına çıkılmaz.

İndeks değerleri **tam sayı** olmak zorundadır. **float** değer indeks değeri olamaz. Önceden anlatıldığı gibi bölme işleminin sonucu her zaman **float** çıkar ve **float** verilerle işlem yapılırsa sonuç her zaman **float** çıkar. İndeks değeri matematiksel bir işlemin sonucuysa işlem sonucunun mutlaka **tam sayı** çıkması gerekir. Üstteki örnekte bölme işlemi sonucu **2.0** çıkmış. Bölme işleminin sonucunun tam sayı çıkmasını istiyorsanız **tam sayı bölme operatörünü(//)** kullanmanız gerekir.

3.6. Liste Metotları

Sınıflar için özelleşmiş fonksiyonlara metot denir. Nesne tabanlı programlama konusunda metot kavramı detaylı bir şekilde anlatılacak.

3.6.1. append() Metodu

```
# append() ile listelere değer ekleme
>>> veriler = [3,5,7]
>>> veriler.append(9)
>>> veriler.append(3.14)
>>> veriler.append("Can")
>>> veriler
[3, 5, 7, 9, 3.14, 'Can']
```

append(eleman) metodu listeye eleman eklemek için kullanılır. **append()** ile eklenen veri listenin sonuna eklenir.

```
# append() fonksiyonu ile aynı işi yapan listeye eleman ekleme
>>> veriler = [3,5,7]
>>> veriler[len(veriler):] = [9]
>>> veriler[len(veriler):] = [3.14]
>>> veriler[len(veriler):] = ["Can"]
>>> veriler
[3, 5, 7, 9, 3.14, 'Can']
```

veriler[len(veriler):] = [9] işlemi **veriler.append(9)** işlemiyle aynı işi yaparak listenin sonuna eleman ekler.

```
# += ile listeye değer ekleme
>>> veriler = [3,5,7]
>>> veriler += [9]
>>> veriler += [3.14]
>>> veriler += ["Can"]
>>> veriler
[3, 5, 7, 9, 3.14, 'Can']
>>> veriler += [0,1,2]
>>> veriler
[3, 5, 7, 9, 3.14, 'Can', 0, 1, 2]
>>> a = 4
>>> veriler += [a]
>>> veriler += [a+4]
>>> veriler
[3, 5, 7, 9, 3.14, 'Can', 0, 1, 2, 4, 8]
```

append() metodunun dışında **+="** operatörüyle diziye eleman eklenebilir. Fakat **+="** operatörü diziyi kendisiyle toplayarak yeniden diziye atama yaptığı için diziye eleman eklemek **append()** metoduna göre daha yavaş olacaktır. Bu yüzden bir diziye eleman eklemenin en iyi yolu **append()** metodunu kullanmaktır.

3.6.2. pop() Metodu

<pre># son elemanı listeden çıkarmak >>> veriler [3, 5, 7, 9, 3.14, 'Can'] >>> veriler.pop() 'Can' >>> veriler.pop() 3.14 >>> veriler.pop() 9 >>> veriler [3, 5, 7]</pre>	<pre># belirli elemanı listeden çıkarmak >>> veriler [3, 5, 7, 9, 3.14] >>> veriler.pop(2) 7 >>> veriler [3, 5, 9, 3.14] >>> veriler.pop(-2) 9 >>> veriler [3, 5, 3.14]</pre>
--	--

pop() metodu değer vermeden kullanılırsa listenin son indeksindeki eleman listeden atılır ve atılan eleman ekrana basılır. **pop(indeks)** metoduna indeks değeri verirse **pop()** metodu verdiğimiz değere karşılık gelen indeksteki elemanı listeden atar ve attığı elemanı ekrana basar.

```
# del ile elemanı listeden çıkarmak
>>> veriler = [3, 5, 7, 9, 3.14, 'Can']
>>> del veriler[2] # 7 siliniyor
>>> veriler
[3, 5, 9, 3.14, 'Can']
>>> veriler[1:4]
[5, 9, 3.14]
>>> del veriler[1:4] # 5, 9, 3.14 siliniyor
>>> veriler
[3, 'Can']
```

del fonksiyonuyla **del veriler[indeks]** şeklinde belirtilen liste elemanı listeden silinebilir. **del veriler[başlangıç:bitiş:basamak]** ifadesiyle dilimlenerek listeden eleman silmek **del** fonksiyonun **pop()** metoduna göre farklıdır. **del veriler[1:4]** ile belirtilen elemanlar dilimlenerek listeden silinir.

3.6.3. sort() Metodu ve sorted() Fonksiyonu

3.6.3.1. sort() Metodu

```
>>> veriler = [5,7,0,1,9,2,6,4,3,8]
# kalıcı şekilde küçükten büyüğe sıralama
>>> veriler.sort()
>>> veriler
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
# kalıcı şekilde büyükten küçüğe sıralama
>>> veriler.sort(reverse = True)
>>> veriler
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

sort() metoduyla liste içerisindeki elemanlar küçükten büyüğe doğru sıralanır. Eğer liste elemanları büyükten küçüğe doğru sıralanmak isteniyorsa **sort()** metodunun **reverse = True** değeri girilir. **sort(reverse = True)** ifadesi listeyi büyükten küçüğe doğru sıralar. **sort()** metoduyla liste içindeki verilerin sırası kalıcı olarak değişir.

```
>>> veriler = ["Can","Tuna","Ahmet","Mehmet"]
# kalıcı şekilde alfabetik olarak küçükten büyüğe sıralama
>>> veriler.sort()
>>> veriler
['Ahmet', 'Can', 'Mehmet', 'Tuna']
# kalıcı şekilde alfabetik olarak büyükten küçüğe sıralama
>>> veriler.sort(reverse = True)
>>> veriler
['Tuna', 'Mehmet', 'Can', 'Ahmet']
```

sort() metodu liste içindeki karakter değerlerini kalıcı bir şekilde alfabetik olarak sıralar.

3.6.3.2. sorted() Fonksiyonu

```
>>> veriler = [5,7,0,1,9,2,6,4,3,8]
# geçici küçükten büyüğe sıralama
>>> sorted(veriler)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

# geçici büyükten küçüğe sıralama
>>> sorted(veriler,reverse=True)
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]

# liste verilerinin yeri değişmiyor
>>> veriler
[5, 7, 0, 1, 9, 2, 6, 4, 3, 8]
```

Liste elemanlarını kalıcı olarak sıralamak istemiyorsanız `sorted()` fonksiyonunu kullanabilirsiniz. `sorted()` fonksiyonu aldığı liste tipli veriyi sıralı bir şekilde döner.

```
# geçici şekilde alfabetik olarak küçükten büyüğe sıralama
>>> veriler = ["Can", "Tuna", "Ahmet", "Mehmet"]
>>> sorted(veriler)
['Ahmet', 'Can', 'Mehmet', 'Tuna']
# geçici şekilde alfabetik olarak büyükten küçüğe sıralama
>>> sorted(veriler, reverse=True)
['Tuna', 'Mehmet', 'Can', 'Ahmet']
# liste verilerinin yeri değişmiyor
>>> veriler
['Can', 'Tuna', 'Ahmet', 'Mehmet']
```

`sorted()` metodu liste içindeki karakter değerlerini geçici bir şekilde alfabetik olarak sıralar.

3.6.3.3. Karakter Dizisi Elemanlarının Alfabetik Sıralanması

Python harfe duyarlı (**case-sensitive**) bir dil olduğu için büyük ve küçük harflerin sıralanması farklı olacaktır. Büyük harflerin **ASCII** tablosundaki değerleri daha küçük olduğu için büyük harfli karakter elemanları sıralamada önde olur. Bu yüzden bir listenin karakter elemanlarını alfabetik olarak sıralamak, tüm karakterler küçük harf değilse karmaşık bir işlemdir.

```
# Liste içindeki karakter elemanları
>>> veriler = ["Abd", "ABE", "abF", "abc"]
# geçici şekilde alfabetik olarak sıralanmıyor
>>> sorted(veriler)
['ABE', 'Abd', 'abF', 'abc']
# kalıcı şekilde alfabetik olarak sıralanmıyor
>>> veriler.sort()
>>> veriler
['ABE', 'Abd', 'abF', 'abc']
```

Örnekte görüldüğü gibi liste içerisinde ilk iki harfi büyük veya küçük **'ab'** ile başlayan karakter dizisi elemanları bulunmakta. **veriler** listesi sıralandığında karakter elemanlarının büyük harfle başlayanlarının önce, küçük harfle başlayanların sonra sıralandığı görülmekte. Bu tip bir sıralama bizim istediğimiz **'abc', 'Abd', 'ABE', 'abF'** alfabetik sıralaması değil.

```
# Liste içindeki karakter elemanları
>>> veriler = ["Abd", "ABE", "abF", "abc"]
# key ile alfabetik olarak geçici sıralanıyor
>>> sorted(veriler, key=str.lower)
['abc', 'Abd', 'ABE', 'abF']
# key ile alfabetik olarak kalıcı sıralanıyor
>>> veriler.sort(key=str.lower)
>>> veriler
['abc', 'Abd', 'ABE', 'abF']
```

`sorted()` fonksiyonunda ve `sort()` metodunda kullanılan **key=str.lower** argümanı liste içinde sıralanacak karakter elemanlarının harflerini küçük harfe çevirir. Tüm karakter elemanları küçük harfli olduğunda alfabetik sıralama düzgün yapılır ve elemanlar orijinal halleriyle sıralı bir şekilde listelenirler.

```
# Liste içindeki karakter elemanları
>>> veriler = ["Abd", "ABE", "abF", "abc"]
# key ile tersten alfabetik olarak geçici sıralanıyor
>>> sorted(veriler, key=str.lower, reverse=True)
['abF', 'ABE', 'Abd', 'abc']
```

```
# key ile tersten alfabetik olarak kalıcı sıralanıyor
>>> veriler.sort(key=str.lower, reverse=True)
>>> veriler
['abF', 'ABE', 'Abd', 'abc']
```

Liste içinde karakter elemanlarını tersten sıralamak istiyorsanız `sort()` metodunda ve `sorted()` fonksiyonunda `reverse=True` argümanını girmeniz gerekir.

```
# Liste içindeki karakter elemanları
>>> veriler = ["abc","Abd","ABE","abF"]
# karakter elemanlarının tümü küçük harfli oluşturuldu
>>> list(map(str.lower,veriler))
['abd', 'abe', 'abf', 'abc']
# geçici şekilde alfabetik olarak sıralanmıyor
>>> sorted(list(map(str.lower,veriler)))
['abc', 'abd', 'abe', 'abf']
```

İleriki konularda göreceğimiz liste içindeki karakter elemanlarını alfabetik sıralama kod örneği. Bu kodun diğer kodlardan farkı listedeki elemanların tümünü hem küçük harfli yapar hem de alfabetik olarak sıralar.

3.6.4. extend() Metodu

```
>>> veriler = [1,3,5,7,9]
>>> liste_ek = [0,2,4,6,8]
# liste'ye liste_ek ekleniyor.
>>> veriler.extend(liste_ek)
>>> veriler
[1, 3, 5, 7, 9, 0, 2, 4, 6, 8]
# bir liste eklendiğinde liste yeniden sıralanarak düzenlenebilir.
>>> veriler.sort()
>>> veriler
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
# liste'ye liste verisi ekleniyor.
>>> veriler.extend([10,11,12])
>>> veriler
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
```

`extend(liste)` metodu listeye başka bir liste eklemek için kullanılır.

```
>>> veriler = [1,3,5,7,9]
>>> liste_ek = [0,2,4,6,8]
# liste'ye liste_ek ekleniyor.
>>> veriler[len(veriler):] = liste_ek
>>> veriler
[1, 3, 5, 7, 9, 0, 2, 4, 6, 8]
# liste yeniden sıralanıyor
>>> veriler.sort()
>>> veriler
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
# liste'ye liste verisi ekleniyor.
>>> veriler[len(veriler):] = [10,11,12]
>>> veriler
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
```

`veriler[len(veriler):] = [10,11,12]` işlemi `veriler.extend([10,11,12])` işlemiyle aynı işi yapar.

```
>>> veriler = ["Tuna","Can","Emel"]
>>> liste_ek = ["Oya","Ahmet"]
# liste + operatörü ile ek listeye birleştirildi.
>>> veriler = veriler + liste_ek
>>> veriler
['Tuna', 'Can', 'Emel', 'Oya', 'Ahmet']
```

```
# liste + operatörü ile ek liste verisiyle birleştirildi.
>>> veriler = veriler + ["Mehmet","Tolga"]
>>> veriler
['Tuna', 'Can', 'Emel', 'Oya', 'Ahmet', 'Mehmet', 'Tolga']
```

Bir listeye + operatörüyle kendi ile toplanarak başka bir liste eklenebilir. Fakat + operatörüyle bir listeye liste eklemek `extend()` metoduna göre daha fazla kaynak gerektirir ve yavaştır. Bunun sebebi, `extend()` metodu listeye sadece diğer listeyi ekler ama + operatörü listenin kendisini ve ek listeyi birleştirerek kendisine atar.

```
>>> veriler = [1,3,5]
# + ile listeye liste dışı veri eklenemez
>>> veriler = veriler + 7
Traceback (most recent call last):
  File "<pyshell#17>", line 1, in <module>
    veriler = veriler + 7
TypeError: can only concatenate list (not "int") to list
# extend() ile listeye liste dışı veri eklenemez
>>> veriler.extend(7)
Traceback (most recent call last):
  File "<pyshell#18>", line 1, in <module>
    veriler.extend(7)
TypeError: 'int' object is not iterable
```

`extend()` ve + operatörüyle listeye ekleme yapmak için eklenecek verinin de liste türünde olması gerekir.

3.6.5. insert() Metodu

```
>>> veriler = [1,3,5]
# 1. indekse 14 girildi
>>> veriler.insert(1,14)
>>> veriler
[1, 14, 3, 5]
# 3. indekse 21 girildi
>>> veriler.insert(3,21)
>>> veriler
[1, 14, 3, 21, 5]

# append() ile aynı işi yapıyor
>>> veriler.insert(len(veriler),47)
>>> veriler
[1, 14, 3, 21, 5, 47]
# ilk eleman 63 girildi
>>> veriler.insert(0,63)
>>> veriler
[63, 1, 14, 3, 21, 5, 47]
```

```
>>> veriler = ["Ahmet","Tuna"]
# 0. indekse eleman ekleniyor
>>> veriler.insert(0,"Can")
>>> veriler
['Can', 'Ahmet', 'Tuna']
# append metodu gibi liste sonuna eleman ekleniyor
>>> veriler.insert(len(veriler),"Potur")
>>> veriler
['Can', 'Ahmet', 'Tuna', 'Potur']
# 2. indekse eleman ekleniyor
>>> veriler.insert(2,"Mehmet")
>>> veriler
['Can', 'Ahmet', 'Mehmet', 'Potur', 'Tuna']
```

`insert()` metodu bir elemanı listenin belli bir indeksine eklememizi sağlar. `veriler.insert(len(veriler),"Potur")` ve `veriler.append("Potur")` aynı işi yapar.

3.6.6. remove() Metodu

```
>>> veriler = ["Can","Ahmet","Tuna","Potur","Mehmet"]
# liste elemanı remove ile listeden çıkarılıyor
>>> veriler.remove("Ahmet")
>>> veriler
['Can', 'Tuna', 'Potur', 'Mehmet']
# remove fonksiyonu değişken ile kullanılarak listeden eleman çıkartılıyor
>>> isim = "Can"
>>> veriler.remove(isim)
>>> veriler
['Tuna', 'Potur', 'Mehmet']
```

remove(eleman) metodu listeden belirtilen elemanın atılması için kullanılır.

```
# listede 2 üç defa kullanılmış
>>> veriler = [1,2,3,4,2,5,2]
# listede bulunan ilk '2' değeri kaldırılır
>>> veriler.remove(2)
>>> veriler
[1, 3, 4, 2, 5, 2]
# listede bulunan ilk '2' değeri kaldırılır
>>> veriler.remove(2)
>>> veriler
[1, 3, 4, 5, 2]
>>> veriler.sort()
>>> veriler
[1, 2, 3, 4, 5]
```

Eğer listede aynı elemandan birden fazla varsa ilk eleman listeden çıkarılır. Üstteki örnekte 2 sayısı listenin 1., 4. ve 6. indeks numaralarında bulunmakta. **veriler.remove(2)** metodu çalıştırılınca önce 1. indekste bulunan 2 değeri kaldırılır. Eğer **veriler.remove(2)** bir daha çalıştırılırsa 3. indekste bulunan 2 değeri kaldırılır.

```
>>> veriler = ['Tuna', 'Can', 'Emel', 'Oya', 'Ahmet', 'Mehmet', 'Tolga']
# liste bulunmayan eleman remove ile çıkartılmaya çalışınca hata çıkar
>>> veriler.remove("Hasan")
Traceback (most recent call last):
  File "<pyshell#20>", line 1, in <module>
    veriler.remove("Eddard")
ValueError: list.remove(x): x not in list
```

Listede bulunmayan elemanı **remove()** metodu ile listeden çıkartılmaya çalışırsanız hata oluşacaktır.

```
>>> veriler = ['Can', 'Emel', 'Oya', 'Ahmet', 'Mehmet']
# eleman listede var mı kontrol ediliyor
>>> if 'Tuna' in veriler :
    veriler.remove('Tuna') # eleman varsa listeden çıkar

>>> veriler
['Can', 'Emel', 'Oya', 'Ahmet', 'Mehmet']
```

Koşullu durumlarda göreceğimiz **if** komutuyla ve **in** operatörüyle çıkartmak istediğiniz elemanın listede olup olmadığı kontrol edilir. Eğer eleman listede varsa listeden çıkartılır yoksa listeden çıkartma işlemi yapılmaz. Çıkartılacak elemanın listede varlığını kontrol etmek hata oluşmasını engelleyecektir. Bu şekilde işlem yapmak kodunuzun daha güvenli çalışmasını sağlayacaktır.

```
>>> veriler = ['Tuna', 'Can', 'Emel', 'Oya', 'Ahmet', 'Mehmet']
# eleman listede var mı kontrol ediliyor
>>> if 'Tuna' in veriler :
    veriler.remove('Tuna') # eleman varsa listeden çıkar

>>> veriler
['Can', 'Emel', 'Oya', 'Ahmet', 'Mehmet']
```

if komutu ve in operatörüyle elemanın listede olduğu kontrol edilerek eleman listeden çıkartılabilir. Üstteki örnekte eleman güvenli bir şekilde listeden çıkartıldı.

3.6.7. index() Metodu

```
>>> veriler = [0,1,2,3,4,5,6,7,8,9]
# 6. eleman baştan başlayarak 6.indekste
>>> veriler.index(6)
6
# 7. eleman 3. indeksten itibaren arandığından 7. indekste
>>> veriler.index(7,3)
7
```

index(eleman) metodu verilen bir değerın baştan başlayarak hangi indekste olduğunu döner. Elemanın hangi indeksten başlanarak aranacağı **index(eleman , arama başlama indeksi)** ifadesiyle yapılır. **index(7,3)** ifadesiyle arama 3. indeksten başlar ve aranan eleman bulunduğunda indeks değeri döner.

```
# indeksi bulunacak eleman listenin başından başlayarak aranır
# Listede aynı elemandan birden çok varsa
# ilk bulunan elemanın indeksi döner
>>> veriler = ['Tuna', 'Can', 'Emel', 'Tuna', 'Mehmet', 'Tuna']
# arama listenin başından başladığı için ilk bulunan elemanın indeksi döner
>>> veriler.index('Tuna')
0
# arama 2. elemandan itibaren başladığı için 2. elemanın indeksi döner
>>> veriler.index('Tuna',2)
3
# arama 4. elemandan itibaren başladığı için 4. elemanın indeksi döner
>>> veriler.index('Tuna',4)
5
```

index() fonksiyonuyla liste içerisinde indeksi bulunmak istenen elemanın aranması listenin başından başlanarak yapılır. Eğer listede aynı elemandan birden çok varsa ilk bulunan listenin indeks numarası sonuç olarak döner. Üstteki örnekte yapıldığı gibi elemanın nerede aranacağı belirtilirse indeks değeri değişecektir. 'Tuna' 0., 3. ve 5. indekslerde bulunmakta **liste.index('Tuna',2)** şeklinde yapılan aramada ilk 'Tuna' elemanı atlanacak arama 2. indeksle başlayacağı için bulunan indeks değeri 3 çıkar. Liste içinde aramanın nereden başlanarak yapılacağını belirtmek büyük listelerde kodunuzun daha hızlı çalışmasını sağlar.

3.6.8. count() Metodu

```
>>> veriler = ["Emel", "Emel", "Oya", "Emel", "Ahmet", "Mehmet", "Ahmet"]
# count() metodu listede elemanın kaç tane olduğunu döner
>>> veriler.count("Emel")
3
>>> veriler.count("Ahmet")
2
>>> veriler.count("Oya")
1
>>> veriler.count("Mehmet")
1
```

count() metodu listede aranan elemanın kaç kere kullanıldığını döner.

3.6.9. clear() Metodu

```
>>> veriler = [0,1,2,3,4,5,6,7,8,9]
# clear metodu liste içindeki elemanları silerek listeyi boşaltır
>>> veriler.clear()
>>> veriler
[]
# clear metodu ile aynı işi yapan diğer işlemler
>>> veriler = list()
>>> veriler = []
>>> veriler[:] = []
>>> del veriler[:]
```

clear() metodu listenin içindeki tüm elemanları silerek listeyi boş liste haline çevirir. **clear()** metoduyla liste boş olarak kod içerisinde varlığını sürdürür. Listelerin içleri boş liste atamasıyla da silinebilir. **liste = list()**, **liste= []**, **veriler[:]=[]** veya **del veriler[:]** işlemleriyle de listelerin içleri silinebilir.

```
# liste değişkeni tamamen siliniyor
>>> del veriler
>>> veriler
Traceback (most recent call last):
  File "<pyshell#12>", line 1, in <module>
    veriler
NameError: name 'veriler' is not defined
```

del fonksiyonu **del liste** şeklinde kullanıldığı zaman liste değişkenini tamamen siler.

3.6.10. copy() Metodu

```
>>> veriler = ["Ahmet", "Tuna", "Potur"]
# liste veriler_yeni' ye kopyalanarak iki liste oluşturuluyor
>>> veriler_yeni = veriler.copy()
>>> veriler_yeni
['Ahmet', 'Tuna', 'Potur']
>>> veriler_yeni.append("Can")
>>> veriler_yeni.append("Mehmet")
>>> veriler
['Ahmet', 'Tuna', 'Potur']
# liste_yeni listesine eklemeler yapılmış
>>> veriler_yeni
['Ahmet', 'Tuna', 'Potur', 'Can', 'Mehmet']
```

copy() metodu listeyi başka bir listeye kopyalar. Bilgilerin kopyalandığı listede herhangi bir değişiklik yaparsanız dahi kopyalanan liste bu değişikliklerden etkilenmez. Listeleri kopyalamanın neden **copy()** metoduyla yapıldığı Nesne tabanlı programlama konusunda anlatılacak.

```
>>> veriler = [0,2,4,6,8]
# liste içindeki tüm bilgiler [:] ile liste_yeni'ye kopyalanıyor
>>> veriler_yeni = veriler[:]
>>> veriler_yeni
[0, 2, 4, 6, 8]
>>> veriler_yeni.extend([1,3,5,7,9])
>>> veriler
[0, 2, 4, 6, 8]
>>> veriler_yeni
[0, 2, 4, 6, 8, 1, 3, 5, 7, 9]
```

[:] operatörü kullanarak da listede bulunan elamanlar başka bir listeye kopyalanabilir.

```
>>> veriler = [0,2,4,6,8]
# = ile listeler bir birlerine kopyalanamaz
>>> veriler_yeni = veriler
>>> veriler_yeni.extend([1,3,5,7,9])
>>> veriler
[0, 2, 4, 6, 8, 1, 3, 5, 7, 9]
>>> veriler_yeni
[0, 2, 4, 6, 8, 1, 3, 5, 7, 9]
```

Eğer '=' operatörü ile değişkenlerde yapıldığı gibi bir liste başka bir değişkene atanırsa iki farklı liste oluşmaz aynı listeyi gösteren farklı isimde liste adı oluşur. Örnekte `veriler_yeni = veriler` işlemiyle `veriler_yeni` listesi `veriler` listesinin elemanlarını kullanır hale geldi. Yani `veriler` ve `veriler_yeni` aynı değerlere sahip oldu. `veriler_yeni.extend([1,3,5,7,9])` işlemiyle hem `veriler` listesine hem `veriler_yeni` listesine elemanlar eklendi. Listelerde '=' operatörüyle yapılan atamanın neden bu şekilde çalıştığı Nesne tabanlı programlama konusunda anlatılacak.

3.6.11. reverse() Metodu

3.6.11.1. reverse() Metodu

```
# karakter verileri
>>> veriler = ["abc","Abd","ABE","abF"]
# liste tersten yerleştirildi
>>> veriler.reverse()
>>> veriler
['abF', 'ABE', 'Abd', 'abc']
# sayısal veriler
>>> veriler = [10,5,9,3,7]
# liste tersten yerleştirildi
>>> veriler.reverse()
>>> veriler
[7, 3, 9, 5, 10]
# sayısal olarak büyükten küçüğe sıralama
>>> sorted(veriler)
[3, 5, 7, 9, 10]
# sayısal olarak küçükten büyüğe sıralama
>>> sorted(veriler,reverse=True)
[10, 9, 7, 5, 3]
```

`reverse()` metodu liste içinde elemanları tersi şekilde sıralayarak listeye yeniden yerleştirir. Örnekte görüldüğü gibi `reverse()` metodu liste elemanlarını büyükten küçüğe doğru sıralamadı listenin içindeki yerleşim şekline göre tersten sıraladı.

3.6.11.2. reversed() Fonksiyonu

```
>>> veriler = [10,5,9,3,7]
>>> # liste geçici olarak tersten yerleştirildi
>>> list(reversed(veriler))
[7, 3, 9, 5, 10]
```

Liste elemanlarını geçici olarak tersten yerleştirmek istemiyorsanız `reversed()` fonksiyonunu kullanabilirsiniz. `reversed()` fonksiyonu aldığı liste tipli veriyi elemanların yerleşimi tersten olacak şekilde döner. `reversed()` fonksiyonun kullanılabilmesi için `list()` fonksiyonun içinde `list(reversed(liste))` şeklinde kullanılması gerekir.

3.7. Listelere Eleman Ekleme (Liste Birleşimi)

```
>>> veriler = [1,2,3]
>>> veriler[len(veriler):] = [4]
>>> veriler = veriler + [5]
>>> veriler
[1, 2, 3, 4, 5, 6, 7, 8]

>>> veriler += [6]
>>> veriler.append(7)
>>> veriler.insert(len(veriler)+1,8)
```

Farklı birçok yol ile listelere eleman eklenebilir. Üstte önceden anlatılan listeye eleman ekleme yöntemlerine örnek verildi. Listeye eleman eklemenin en kullanışlı yolu `append()` metodunu kullanmaktır.

```
>>> liste_1 = ['a','b','c','d']
>>> liste_2 = ['e','f','g','h']
# listeler toplanabilir
>>> liste_1 + liste_2
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
```

Listeler birbiriyle toplanabilir. İki liste toplandığı zaman ikinci liste birinci listeye eklenir ve sonuç liste olarak çıkar.

```
# listeler sadece listelerle toplanabilir
>>> ['a','b','c','d'] + 'e'
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in <module>
    ['a','b','c','d'] + 'e'
TypeError: can only concatenate list (not "str") to list
```

Toplama işlemini listelerde kullanabilmek için her iki değerinde liste olması gerekir.

```
>>> veriler = ['a','b','c','d']
# iki liste toplanarak bir listeye atanabilir
>>> liste_t = veriler + [1,2,3,4]
>>> liste_t
['a', 'b', 'c', 'd', 1, 2, 3, 4]
```

İki liste toplandığında listeler birbirine eklenir. Oluşan liste başka bir listeye eklenerek yeni bir liste oluşturur.

```
# listeleri birleştirmek için extend metodu kullanılabilir.
>>> veriler = [1,2,3]
>>> veriler.extend([4,5,6])
>>> veriler
[1, 2, 3, 4, 5, 6]
```

Listeleri birleştirmek için önceden kullanılan `extend()` metodu kullanılabilir.

3.8. Listeleri Çarparak Tekrar Ettirme

```
>>> liste = ['a','b','c','d']
# liste elenaları çarpılarak tekrar ettirilebilir
>>> liste * 3
['a', 'b', 'c', 'd', 'a', 'b', 'c', 'd', 'a', 'b', 'c', 'd']
```

Liste çarpılarak tekrar ettirilebilir. Sonuç tekrar ettirilerek oluşan yeni bir listedir.

```
>>> veriler = ['a','b','c','d']
# listenin içeriğini değiştirmek istiyorsanız işlemi listeye atamalısınız.
>>> veriler
['a', 'b', 'c', 'd']
>>> veriler = veriler * 3
>>> veriler
['a', 'b', 'c', 'd', 'a', 'b', 'c', 'd', 'a', 'b', 'c', 'd']
```

Listeler ile yaptığınız işlemlerin sonuçlarını bir listeye atamazsanız çalıştığınız listenin değeri değişmeyecektir. Eğer çarpma işlemi gibi bir işlem yapıp listenin değerini çıkan sonuca göre değiştirmek istiyorsanız, yaptığınız işlemi kullandığınız listeye atamalısınız. Önce işlem sonra atama yapılarak listenin değeri işlemin sonucuyla değişecektir.

3.9. Dilimleme

liste[Başlangıç : Bitiş : Basamak]

```
>>> veriler = [0,1,2,3,4,5,6,7,8,9,10,11,12]
# Veriler İçerisinden Belirli Sayıda ve Koşulda Elemanları Seçme
# 3 - 9 arasında 2 şer arayla          # En baştan son elemana 3 er
>>> veriler[3:10:2]                    >>> veriler[:3]
[3, 5, 7, 9]                          [0, 3, 6, 9, 12]
# en baştan 4 Eleman                  # 2.'den son elemana 3 er
>>> veriler[:4]                        >>> veriler[2:3]
[0, 1, 2, 3]                          [2, 5, 8, 11]
# en baştan 4 Eleman                  # baştan 5'e 2 şer
>>> veriler[0:4]                      >>> veriler[:5:2]
[0, 1, 2, 3]                          [0, 2, 4]
# 3 - 10 arası elemanlar              # verilerin tersi
>>> veriler[3:10]                    >>> veriler[::-1]
[3, 4, 5, 6, 7, 8, 9]                [12,11,10,9,8,7,6,5,4,3,2,1,0]
# 8. Elemandan son elemana           # 2'şer arayla verilerin tersi
>>> veriler[8:]                      >>> veriler[::-2]
[8, 9, 10, 11, 12]                  [12, 10, 8, 6, 4, 2, 0]
# Verilerdeki elemanların tümünü farklı bir yolla listelenmiş
>>> veriler[:]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
```

Liste içerisinde farklı aralıklarda istenen sayıda elemana ulaşılabilir. **liste[Başlangıç : Bitiş : Basamak]** işlemiyle liste içerisinde hangi elemanların seçileceği belirtilir. Üsteki örnekte liste içerisinde belirli sayıda ve koşulda elemanların nasıl seçileceği örneklerle gösterilmiştir. Listelerde yapılan bu işleme **dilimleme** denir.

```
>>> veriler = [0,1,2,3,4,5,6,7,8,9,10,11,12]
# Dilimlemede bitiş değerindeki eleman listeye alınmaz.
# 7. Eleman listelenmiyor          # 12. Eleman listelenmiyor
>>> veriler[:7]                      >>> veriler[2:12:2]
[0, 1, 2, 3, 4, 5, 6]                [2, 4, 6, 8, 10]
>>> veriler[7]                      >>> veriler[12]
7                                    12
# 11. Eleman listelenmiyor          # 11. Eleman listelenmiyor
>>> veriler[3:11]                    >>> veriler[1:11:2]
[3, 4, 5, 6, 7, 8, 9, 10]            [1, 3, 5, 7, 9]
>>> veriler[11]                      >>> veriler[11]
11                                    11
```

Listelenen elemanlar **Bitiş** değerinden önceki elemanlardır. **Bitiş** değerindeki eleman listeye alınmaz.

```
>>> veriler = [1,2,3]
>>> veriler
[1,2,3]
# listenin tüm elemanlarını dilimlenerek değiştirmek
>>> veriler[:] = 'Oya', 'Ahmet', 'Mehmet'
>>> veriler
['Oya', 'Ahmet', 'Mehmet']
```

Bir listeye listenin eleman sayısında değer atanarak listenin tüm değerleri değiştirilebilir. **veriler[:]='Oya','Ahmet','Mehmet'** işlemiyle liste içindeki tüm değerler değişecektir.

```

>>> veriler = ['a', 'b', 'c', 'd', 1, 2, 3, 4]
# listede belirtilen elemanlar dilimlenerek değiştirilebilir
# baştan iki eleman değiştiriliyor
>>> veriler[:2] = [50,60]
>>> veriler
[50, 60, 'c', 'd', 1, 2, 3, 4]
>>> veriler[2:4]
['c', 'd']
# 2. 3. ve 4. Eleman değiştiriliyor
>>> veriler[2:5] = [70,80,90]
>>> veriler
[50, 60, 70, 80, 90, 2, 3, 4]

```

Liste içerisinde belirtilen elemanlar değiştirilebilir. Üsteki listede baştan iki eleman **50** ve **60** değeriyle, sonra 2. eleman dahil 5. elemana kadar olan elemanlar **70**, **80** ve **90** değeriyle değiştirilmiş.

```

>>> isimler = ['Tuna', 'Can', 'Emel', 'Oya', 'Ahmet', 'Mehmet', 'Tolga']
>>> isimler[1:5]
['Can', 'Emel', 'Oya', 'Ahmet']
>>> isimler[1:5] = ['Kerem', 'Alp', 'Ada', 'Mert']
>>> isimler
['Tuna', 'Kerem', 'Alp', 'Ada', 'Mert', 'Mehmet', 'Tolga']

```

isimler listesindeki 1., 2., 3. ve 4. elemanlar farklı 4 elemanla değiştirildi.

```

>>> isimler = ['Tuna', 'Can', 'Emel', 'Oya', 'Ahmet', 'Mehmet', 'Tolga']
# 2 ve 5 arasındaki elemanlar
>>> isimler[2:5]
['Emel', 'Oya', 'Ahmet']
# 2 ve 5 arasındaki elemanlar listeden siliniyor
>>> isimler[2:5] = []
>>> isimler
['Tuna', 'Can', 'Mehmet', 'Tolga']

```

isimler[2:5] = [] işlemiyle dilimle yöntemiyle liste içinde belirtilen elemanlar listeden silinebilir.

```

>>> veriler = [0,1,2,3,4,5,6,7,8,9,10,11,12]
# Değişkenler kullanılarak liste elemanları dilimlenebilir
# 9. Elemandan sonrası          # 3. Eleman ile 11. Eleman arası 2 şer
>>> Başlangıç = 9                >>> Başlangıç, Bitiş, Basamak = 3, 11, 2
>>> veriler[Başlangıç:]          >>> veriler[Başlangıç : Bitiş : Basamak]
[9, 10, 11, 12]                 [3, 5, 7, 9]
# 5 ve 11. Elemana kadar 11 yok  # Baştan 11. Elemana 3 er
>>> Başlangıç, Bitiş = 5, 11      >>> Başlangıç, Bitiş, Basamak = 0, 11, 3
>>> veriler[Başlangıç:Bitiş]      >>> veriler[Başlangıç : Bitiş : Basamak]
[5, 6, 7, 8, 9, 10]             [0, 3, 6, 9]

```

Değişkenler kullanılarak dilimleme yapılabilir.

```

# Değişkenler kullanılarak liste elemanları dilimlenebilir
>>> Başlangıç, Bitiş, Basamak = None, None, None
>>> veriler = list(range(10))
>>> veriler[Başlangıç:Bitiş:Basamak]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> Başlangıç, Bitiş, Basamak = 3, None, None
>>> veriler[Başlangıç:Bitiş:Basamak]
[3, 4, 5, 6, 7, 8, 9]
>>> Başlangıç, Bitiş, Basamak = 3, 8, None
>>> veriler[Başlangıç:Bitiş:Basamak]
[3, 4, 5, 6, 7]

```

```
>>> Başlangıç, Bitiş, Basamak = 3, 8, 2
>>> veriler[Başlangıç:Bitiş:Basamak]
[3, 5, 7]
>>> Başlangıç, Bitiş, Basamak = None, None, -1
>>> veriler[Başlangıç:Bitiş:Basamak]
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

Geniş kapsamlı bir program yaptığınızda listelerin elemanlarına değişkenler kullanarak erişirsiniz. Üstteki örnekte listenin **Başlangıç**, **Bitiş** ve **Basamak** değişkenlerine değer verilerek dilimlenmesi gösterilmiştir.

3.10. Liste Verileri Üzerinde Doğrudan Çalışmak

Python nesne tabanlı bir programlama dili olduğu için Python'da her şey bir nesnedir. Nesne tabanlı programlama konusunda Python'un bu özelliğinden detaylı bir şekilde bahsedilecek. Listelerde diğer tüm veri tipleri gibi nesnel bir veri tipi olduğundan liste verilerini aynı değişkenleri kullandığımız gibi kullanabiliriz. Özellikle bu tarz kullanım şekli Karakter dizilerinde formatlama metodunda karşımıza çıkacak.

```
# liste verileri üzerinde doğrudan çalışmak
>>> [0,1,2,3,4,5,6,7,8,9][3]          >>> [1,2,2,3,3,3,4,4,4,4].count(3)
3                                     3
>>> [0,1,2,3,4,5,6,7,8,9][3:8]        >>> [0,1,2,3,4,5,6,7,8,9].pop()
[3, 4, 5, 6, 7]                      9
>>> [0,1,2,3,4,5,6,7,8,9][3:8:2]      >>> [0,1,2,3,4,5,6,7,8,9].index(5)
[3, 5, 7]                             5
>>> ['Tuna', 'Can', ['Emel', 'Oya', 'Ahmet'], 39, 3.14][3:]
[39, 3.14]
>>> ['Tuna', 'Can', ['Emel', 'Oya', 'Ahmet'], 39, 3.14][2][1]
'Oya'
>>> ['Tuna', 'Can', ['Emel', 'Oya', 'Ahmet'], 39, 3.14].index('Can')
1
>>> veriler = ['Tuna', 'Can', ['Emel', 'Oya', 'Ahmet'], 39, 3.14][2]
>>> veriler
['Emel', 'Oya', 'Ahmet']
```

Örnekte görüldüğü gibi liste verisi aynı liste tipi değişkenler gibi kullanılabilir. Fakat bu tarz kullanım çok dezavantajlı olduğu için tercih edilmemelidir.

```
# liste içindeki elemanların kendi metotları kullanılabilir.
# liste içindeki karakter dizilerinin metotları kullanılabilir
>>> veriler = ['tuna', 'can', 'EMEL']
>>> veriler[0].title()                >>> veriler[2].lower()
'Tuna'                               'emel'
>>> veriler[1].upper()                >>> veriler[2].title()
'CAN'                                'Emel'
```

Liste içerisinde karakter dizisi olan elemanlar varsa karakter dizisi elemanları üzerinde karakter dizisi elemanlarına ait metotlar kullanılabilir. Liste içerisindeki karakter dizilerinin metotlarının kullanılabilmesi için karakter dizisi elemanlarının birer birer çağrılması gerekir. **title()** metodu karakterlerin ilk harfini büyük yapar, **lower()** karakterlerin tümünü küçük harf yapar ve **upper()** karakterlerin tüm harflerini büyük harf yapar. Karakter dizilerinin metotları ilerde daha detaylı incelenecek.

```
# dizi elemanlarını karakterler ile birleştirmek
>>> veriler
['tuna', 'can', 'EMEL']
>>> mesaj = "Ailenin en küçük çocuğu " + veriler[0].title() + "."
>>> mesaj
'Ailenin en küçük çocuğu Tuna.'
```


Liste içerisinde bulunan karakter elemanı listeden metot kullanarak çekilerek başka karakter dizileriyle birleştirilebilir. Örnekte `veriler[0].title()` ifadesiyle 'tuna' elemanı baş harfi büyük harf yapılarak ('Tuna') önündeki cümleyle birleştirilmiş.

3.11. in Operatörü

```
>>> veriler = ["Tuna", "Can", "Emel", "Oya", "Ahmet", "Mehmet"]
# Eleman verilerde varsa True          # Eleman verilerde yoksa False
>>> "Tuna" in veriler                  >>> "Tolga" in veriler
True                                  False
```

`in` operatörü bir elemanın listede olup olmadığını kontrol eder. Eğer eleman listede varsa **True** yoksa **False** döner. `in` operatörü koşullu durumlar ve döngüler konusunda yine karşımıza çıkacak.

3.12. range() Fonksiyonu

```
>>> veriler = list(range(10))
>>> veriler
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

`range()` fonksiyonu listeler için otomatik tam sayı değeri üretmekte kullanılır. `liste = list(range(10))` şeklinde kullanılan `range()` fonksiyonu 0'dan 10'a (10 hariç) kadar sayılardan oluşan liste oluşturur.

```
>>> veriler = list(range(1,20,2))
>>> veriler
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
>>> type(range(1,20,2))
<class 'range'>
>>> type(veriler)
<class 'list'>
```

`range(Başlangıç , Bitiş , Basamak)` `range()` fonksiyonun içine girilen

Başlangıç üretilecek ilk sayıdır

Bitiş kaçınıcı sayıya kadar sayı üretileceğini belirtir. Üretilen son sayı **Bitiş** değerinden bir küçüktür

Basamak üretilen sayıların kaçar kaçar üretileceğini belirtir.

`veriler = list(range(1,20,2))` işleminde `range()` fonksiyonu 1 den 20'ye (20 hariç) kadar ikişer ikişer artan sayılardan bir dizi oluşturur ve `veriler` isimli `list` tipindeki değişkene atar. `range()` fonksiyonunu en çok döngüler konusunda kullanacağız.

```
>>> veriler = list(range(11))
>>> veriler
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> veriler = list(range(3,10))
>>> veriler
[3, 4, 5, 6, 7, 8, 9]
>>> veriler = list(range(-9,10))
>>> veriler
[-9, -8, -7, -6, -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> veriler = list(range(0,10,3))
>>> veriler
[0, 3, 6, 9]
```

`range(Başlangıç)` şeklinde kullanılan `range()` fonksiyonu 0 dan **Başlangıç** değerine kadar sayı üretir.

`veriler = list(range(10))` 0'dan 10'a (10 hariç) kadar sayıları üretir. [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

`range(Başlangıç , Bitiş)` şeklinde kullanılan `range()` fonksiyonu **Başlangıç - Bitiş** arasında sayı üretir.

`veriler = list(range(3,10))` 3'ten 10'a (10 hariç) kadar sayıları üretir. [3, 4, 5, 6, 7, 8, 9]

`range(Başlangıç , Bitiş , Basamak)` şeklinde kullanılan `range()` fonksiyonu **Başlangıç - Bitiş** arasında sayıları **Basamak** değerine göre üretir.

`veriler = list(range(0,10,3))` 0'dan 10'a (10 hariç) kadar sayıları 3'er 3'er üretir. [0, 3, 6, 9]

3.13. İç İçe Listeler

```
>>> veriler=[[0,1,2],[3,4,5],[6,7,8]]
>>> veriler
[[0, 1, 2], [3, 4, 5], [6, 7, 8]]
# 1. Veriler 0. eleman
>>> veriler[1][0]
3
# 0. Veriler 2. eleman
>>> veriler[0][2]
2
# 2. Veriler 2. eleman
>>> veriler[2][2]
8
# 1. Veriler 1. eleman
>>> veriler[1][1]
4
# 2. Veriler 2. eleman
>>> veriler[-1][-1]
8

>>> list_1 = [9,8,7]
>>> list_2 = [6,5,4]
>>> list_3 = [3,2,1]
>>> veriler=[list_1,list_2,list_3]
>>> veriler
[[9, 8, 7], [6, 5, 4], [3, 2, 1]]
>>> a,b = 1,2
>>> veriler[a][b]
4
>>> a,b = 0,0
>>> veriler[a][b]
9
>>> a,b = 2,2
>>> veriler[a][b]
1
>>> a,b = -1,-1
>>> veriler[a][b]
1
```

İç içe listeler bir listenin başka bir liste içinde bulunduğu listelerdir. Bu tip listeler matris veya ağaç yapılarında kullanılmaktadır. Listenin elemanlarına çağırarak için iki tane `[]` operatörü kullanılır. `veriler[2][2]` 3. listenin 3. elemanı çağırarak için kullanılır.

```
>>> veriler = ['Tuna', 'Can', ['Emel', 'Oya', 'Ahmet'], 39, 41, 3.14, 5.67]
# ['Emel', 'Oya', 'Ahmet'] liste içinde 1 eleman olarak sayılır
>>> len(veriler)
7
# ['Emel', 'Oya', 'Ahmet'] elemanına veriler[2] ile erişilir
>>> veriler[2]
['Emel', 'Oya', 'Ahmet']
# liste[2] elemanlarına erişim için veriler[2][indeks] işlemi yapılır
>>> veriler[2][0]
'Emel'
>>> veriler[2][1]
'Oya'
>>> veriler[2][1:]
['Oya', 'Ahmet']
# liste[2] elemanlarıyla liste_yeni oluşturuluyor
>>> veriler_yeni = veriler[2]
# liste_yeni içinde iç liste yok
>>> veriler_yeni
['Emel', 'Oya', 'Ahmet']
# liste_yeni elemanları liste_yeni[indeks] ile çağrılır
>>> veriler_yeni[0]
'Emel'
>>> veriler_yeni[1]
'Oya'
```

`veriler` içinde `['Emel', 'Oya', 'Ahmet']` gibi bir liste daha var. Bu liste ana listenin elemanlarından biridir ve bu da öteki elemanlar gibi tek elemanlık bir yer kaplar. liste içinde bulunan `['Emel', 'Oya', 'Ahmet']` listesi 2. indekste olduğu için bu iç listeye erişim `veriler[2]` işlemiyle yapılır. `veriler[2]` listesinin elemanlarına erişim için `veriler[2][indeks]` işlemiyle yapılır. `veriler[2]` listesi daha rahat kullanım için `veriler_yeni = veriler[2]` işlemiyle yeni bir listeye kopyalanırsa `veriler_yeni` tekli listeler gibi kullanılabilir.

4. Demetler (tuple) Veri Tipi

Demetler ilk defa Python ile birlikte programlama dünyasına tanıtılmıştır. Türkçe karşılığı tam olarak bulunmayan tuple, demet veya tüp olarak Türkçe 'ye çevrilebilir. Demetler, listelere çok benzer ve tıpkı listeler gibi farklı veri tiplerini içinde barındırır, elemanlarına erişilir, dilimlenir. Fakat demetler ile listeleri ayıran en önemli fark tanımlandıktan sonra demetlerin üzerinde değişiklik yapılamamasıdır. Demetler **değiştirilemeyen (immutable)** özelliktedir. Nasıl ki listeleri ayırt eden özellik kareli parantez ise demetleri ayırt eden özellik de parantezlerdir. Demetleri tanımlamak için parantez '(') işaretlerinden yararlanılır.

4.1. Demetleri Oluşturmak ve tuple() Fonksiyonu

```
>>> # Boş Demet
>>> demet_1 = tuple()
>>> demet_2 = ()
>>> # Sadece int sayılar demeti
>>> demet_3 = (1,1,2,3,5,8,13)
>>> # değişik tipte veriler demeti
>>> demet_4 = (5,3.14,"Tuna")

>>> # Boş demet
>>> print(demet_1, demet_2)
() ()
>>> demet_3
(1, 1, 2, 3, 5, 8, 13)
>>> demet_4
(5, 3.14, 'Tuna')
```

tuple() fonksiyonu **tuple** tipinde **demetleri** oluşturmak ve karşılığı olan tipleri **tuple** tipine çevirmek için kullanılır. Demetleri değer vermeden boş olarak veya değer vererek dolu olarak oluşturabiliriz.

demet_1 demeti: **tuple()** fonksiyonuyla **demet_1 = tuple()** atamasıyla **boş demet**.

demet_2 demeti: **()** boş demet operatörüyle **demet_2 = ()** atamasıyla **boş demet**

demet_3 demeti: **demet_3 = [1,1,2,3,4,8,13]** atamasıyla **int** değerlerden oluşan Fibonacci demeti

demet_4 demeti: **demet_4 = [5,3.14,"Tuna"]** atamasıyla **değişik veri tiplerinden** oluşan bir demet oluşturulmuştur.

```
>>> # type() fonksiyonu demetin tipini tuple olarak gösterir
>>> veriler = (17,3.14,"Ahmet")
>>> type(veriler)
<class 'tuple'>
```

Demet tipinde değişkenler **type()** fonksiyonuyla incelenir. **type()** fonksiyonu demet tipleri için **'tuple'** tipini döner.

```
>>> # liste demete çevriliyor
>>> demet_tamSayilar = tuple([1,3,5,6,7,9])
>>> demet_tamSayilar
(1, 3, 5, 6, 7, 9)
>>> # string değer demete çevriliyor
>>> selam = "Merhaba Dünya"
>>> demet_selam = tuple(selam)
>>> demet_selam
('M', 'e', 'r', 'h', 'a', 'b', 'a', ' ', 'D', 'ü', 'n', 'y', 'a')
>>> type(demet_tamSayilar)
<class 'tuple'>
>>> type(demet_selam)
<class 'tuple'>
```

tuple() fonksiyonu listeleri veya karakter dizilerini demet veri tipine çevirir.

```
>>> # parantez olmadan demet oluşturulabilir
>>> veriler = 1.22, 23, "Can"
>>> type(veriler)
<class 'tuple'>
>>> veriler
(1.22, 23, 'Can')
```

Demet tanımlamanın birden fazla yolu vardır. Demetler parantez içinde gösterilir fakat demet tanımlamak için parantezlere gerek yoktur. Virgülle birbirlerinden ayrılan değerler değişkene atandığında demet tanımlanmış olur.

```
# bir değerli demet oluşturmak
>>> b = (5,)
# veya
>>> b = 5,
>>> b
(5,)
>>> type(b)
<class 'tuple'>

>>> d = 'Can',
# veya
>>> d = ('Can',)
>>> d
('Can',)
>>> type(d)
<class 'tuple'>

"""() parantez olsa bile ,virgül
olmadan oluşturulan değişken
demet tipinde değildir!"""
>>> a = (3)
# veya
>>> a = 3
>>> type(a)
<class 'int'>

>>> c = ('Can')
>>> c
'Can'
>>> type(c)
<class 'str'>
```

Virgül (,) olmadan `a = (3)` veya `a = 3` ataması ile tam sayı (int) tipinde `c = ('Can')` veya `c = 'Can'` atamasıyla(str) tipinde değişken tanımlanır. Tek öğeli bir demet oluşturma işlemi biraz tuhaftır. Bir değerli demet oluşturulurken değer sağına Virgül (,) konulur. `b = (5,)` veya `b = 5,` ataması, `d = 'Can',` veya `d = ('Can',)` ataması ile bir değerli demet oluşturulur.

4.2. Demet Elemanlarına Erişmek(İndeks)



```
>>> veriler = (1,2,3,4,5,6,7)
# indeks numarasıyla elemanı çağırmak. veriler[indeks] işlemi
>>> veriler[0]
1
>>> veriler[1]
2
>>> veriler[2:6:2]
(3, 5)

>>> veriler[5]
6
>>> veriler[6]
7
>>> veriler[:4]
(1, 2, 3, 4)

>>> veriler[-1]
7
>>> veriler[-2]
6
>>> veriler[1:6]
(2, 3, 4, 5, 6)
```

Önceden öğrendiğimiz indeksleme ve dilimleme kuralları demetler için de geçerli. Liste elemanları nasıl çağırıyorsa aynı şekilde demet elemanları da çağırılabilir.

4.3. len() Fonksiyonu

```
>>> #len() Fonksiyonu
>>> liste_say = (1,2,3,4,5)
>>> len(liste_say)
5

>>> liste_karistik = (7,1.22,"Can",[1,3],[2,4])
>>> len(liste_karistik)
5
```

`len(demet)` fonksiyonu değer olarak aldığı demetin eleman sayısını verir.

4.4. Demetler Değiştirilemez(Immutable)

```
>>> veriler = (1,2,3,4,5)
>>> # demet elemanları değişmez
>>> veriler[0] = 10
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    veriler[0] = 10
TypeError: 'tuple' object does not support item assignment

>>> # demet elemanları silinemez
>>> del veriler[2]
Traceback (most recent call last):
  File "<pyshell#21>", line 1, in <module>
    del veriler[2]
TypeError: 'tuple' object doesn't support item deletion
```

Demetler **değiştirilemeyen** (immutable) özelliktedir. Demetlerin değişmez olması demetlerin yanlışlıkla değişmesini engeller. Program akışı içerisinde değeri değişmeyecek tanımları demetler ile yapabilirsiniz. Ayrıca

demetler bellek kullanımından tasarruf sağlar ve üzerinde işlem yapmak listelere kıyasla daha hızlıdır. Dolayısıyla, performans avantajı nedeniyle de listeler yerine demetleri kullanmak isteyebilirsiniz. Örnekte görüldüğü gibi demet elemanlarından birinin değeri değiştirilmeye veya elemanlardan biri silinmeye çalışıldığında Python hata verecektir.

4.5. İndeks Sınırlarına Çıkmak, Hatalı İndeks Değeri

```
>>> veriler = 1,2,3,4,5,6,7 # demet sınırları dışına çıkılamaz
>>> veriler[7]
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    veriler[7]
IndexError: tuple index out of range

>>> veriler[-8]
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    veriler[-8]
IndexError: tuple index out of range
```

Demetlerde indeks sınırlarına çıkmak, hatalı indeks değeri vermek listelerde olduğu gibi hata verecektir.

4.6. Demet Metotları

4.6.1. count() Metodu

```
>>> veriler = ("Emel", "Emel", "Oya", "Emel", "Ahmet", "Mehmet", "Ahmet")
# count() metodu demette elemanın kaç tane olduğunu döner
>>> veriler.count("Emel")
3
>>> veriler.count("Ahmet")
2

>>> veriler.count("Oya")
1
>>> veriler.count("Mehmet")
1
```

count() metodu demette aranan elemanın kaç kere kullanıldığını döner.

4.6.2. index() Metodu

```
>>> veriler = (0,1,2,3,4,5,6,7,8,9)
# 6. eleman baştan başlayarak 6. indekste
>>> veriler.index(6)
6
```

index(eleman) metodu verilen bir değerın baştan başlayarak hangi indekste olduğunu döner. Elemanın hangi indeksten başlanarak aranacağı index(eleman , arama başlama indeksi) ifadesiyle yapılır. index(7,3) ifadesiyle arama 3. indeksten başlar ve aranan eleman bulunduğu anda indeks değeri döner.

4.7. Demetlere Eleman Ekleme(Demet Birleşimi)

```
>>> veriler = 1.22,23,"Merhaba"
# demete elemanlar ekleniyor
>>> veriler += 'Dünya',9,12
>>> veriler
(1.22, 23, 'Merhaba', 'Dünya', 9, 12)
# demete bir eleman eklemek için elemanın sağ yanına virgül ',' eklenmeli
>>> veriler += (17,)
>>> veriler
(1.22, 23, 'Merhaba', 'Dünya', 9, 12, 17)
# demete birden çok eleman eklendiğinde
# elemanların en sonuna virgül ',' eklemek hata vermez
>>> veriler += (3.14,'Tuna',)
>>> veriler
(1.22, 23, 'Merhaba', 'Dünya', 9, 12, 17, 3.14, 'Tuna')
```

Demetler üzerinde değişiklik yapılamaz. Fakat veriler += 'Dünya',9,12 işlemiyle demetler toplayarak demetlere eleman eklenebilir.

```
>>> veriler += 'Dünya',9,12
>>> veriler = veriler + 'Dünya',9,12
```

Demetlere eleman eklenirken demetler değiştirilemez kuralı çiğnenmez. Eleman eklenirken yapılan üstteki örnekte görüldüğü gibi demetin kendisiyle birlikte yeni elemanlarla toplanarak yeniden atanmasıdır.

```
>>> veriler += (17,) >>> veriler += 17,
```

Demete bir eleman eklemek için elemanın sağına virgöl konması gerekir. Sadece aynı tür veriler birbiriyle birleştirilebilir. Elemanın sağına virgöl eklendiğinde eleman demet haline gelir. Sağına virgöl konan eleman başka bir demetle toplanarak demete eklenebilir.

```
>>> veriler += (17) >>> veriler += 17
Traceback (most recent call last):
  File "<pyshell#22>", line 1, in <module>
    veriler += (17)
TypeError: can only concatenate tuple (not
"int") to tuple
Traceback (most recent call last):
  File "<pyshell#23>", line 1, in <module>
    veriler += 17
TypeError: can only concatenate tuple (not
"int") to tuple
```

Bir elemanı demete eklemeye çalıştığınızda farklı iki veri tipini birleştirmeye çalışırsınız bu durumda hata verir.

```
veriler += (3.14, 'Tuna',) veriler += 3.14, 'Tuna',
```

Birden çok eleman demete eklendiğinde elemanların sağına virgöl koymak hataya yol açmaz.

4.8. Demetleri Çarparak Tekrar Ettirme

```
>>> veriler = 'a','b','c','d'
# Demetler çarpılarak tekrar ettirilebilir
>>> veriler*3
('a', 'b', 'c', 'd', 'a', 'b', 'c', 'd', 'a', 'b', 'c', 'd')
```

Demetler çarpılarak tekrar ettirilebilir. Sonuç tekrar ettirilerek oluşan yeni bir demettir.

```
>>> veriler = 1,2,3
# Demet içeriğini değiştirmek istiyorsanız işlemi demete atamalısınız.
>>> veriler = veriler*3
>>> veriler
(1, 2, 3, 1, 2, 3, 1, 2, 3)
```

Eğer çarpma işlemi yapıp demetin değerini çıkan sonuca göre değiştirmek istiyorsanız, yaptığınız işlemi kullandığınız demete atamalısınız.

4.9. Dilimleme

```
>>> veriler = (0,1,2,3,4,5,6,7,8,9,10,11,12)
>>> veriler[3:10:2] >>> veriler[:3]
(3, 5, 7, 9) (0, 3, 6, 9, 12)
>>> veriler[8:] >>> veriler[2::-1]
(8, 9, 10, 11, 12) (2, 1, 0)
>>> veriler[::-1]
(12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0)
```

Demetler aynı listeler gibi dilimlenebilir.

4.10. Demet Verileri Üzerinde Doğrudan Çalışmak

```
>>> (0,1,2,3,4,5,6,7,8,9)[3] >>> (1,2,2,3,3,3,4,4,4,4).count(3)
3 3
>>> (0,1,2,3,4,5,6,7,8,9)[3:8:2] >>> (0,1,2,3,4,5,6,7,8,9).index(5)
(3, 5, 7) 5
```

Demetler nesnel bir veri tipi olduğundan demetleri aynı değişkenleri kullandığımız gibi kullanabiliriz.

4.11. in Operatörü

#in Operatörü

```
>>> veriler = ("Tuna","Can","Emel","Oya","Ahmet","Mehmet")
>>> "Tuna" in veriler
True
>>> "Tolga" in veriler
False
```

in operatörü bir elemanın demetin elemanı olup olmadığını kontrol eder.

4.12. range() Fonksiyonu

```
>>> veriler = tuple(range(10))
>>> veriler
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
>>> veriler = tuple(range(1,20,2))
>>> veriler
(1, 3, 5, 7, 9, 11, 13, 15, 17, 19)
```

range() fonksiyonuyla demetlere otomatik tam sayı elemanlar üretilir.

4.13. İç İçe Demetler

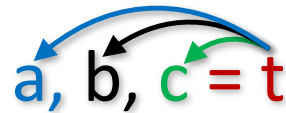
```
# İç İçe Demetler
>>> demet_1 = (9,8,7)
>>> demet_2 = (6,5,4)
>>> demet_3 = (3,2,1)
>>> veriler=(demet_1,demet_2,demet_3)
>>> veriler
((9, 8, 7), (6, 5, 4), (3, 2, 1))
>>> a,b = 1,2
>>> veriler[a]
(6, 5, 4)
>>> a,b = 2,2
>>> veriler[a][b]
1
>>> a,b = -1,-1
>>> veriler[a][b]
1
>>> veriler=((0,1,2),(3,4,5),(6,7,8))
>>> veriler[1]
(3, 4, 5)
>>> veriler[1][0]
3
>>> veriler[2][2]
8
# demet içinde liste tanımlanabilir
>>> veriler=([0,1,2],[3,4,5],[6,7,8])
>>> veriler[0]
[0, 1, 2]
>>> veriler[1][2]
5
>>> veriler[0][1]=10 # liste değişir
>>> veriler
([0, 10, 2], [3, 4, 5], [6, 7, 8])
```

İç içe demetler oluşturulabilir. Demet içinde liste tanımlanabilir. Listelerin demetlerin içinde tanımlanabilmesi liste elemanlarının değişmesine engel değildir. Demet içinde tanımlı liste elemanları değiştirilebilir.

4.14. Sekans Açma (sequence unpacking)

list, tuple ve range gibi veriler **sekans veri** tipidir. Sekans verisi içinde bulunan eleman sayısı kadar değişken sekans veri tipinin soluna virgül ile ayrılarak yazılır ve sekans verisine eşitlenirse sekans içinde bulunan elemanlar sırasıyla değişkenlere atanır. Sekans elemanlarının değişkenlere atanmasına **sekans açma** denir.

```
>>> t = 1981, 91754, 'Tuna'
>>> a, b, c = t
>>> print("Değişkenlerin Değerleri:",a,b,c)
Değişkenlerin Değerleri: 1981 91754 Tuna
```



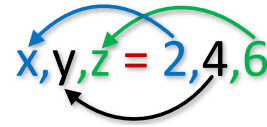
Demetler sekans veri tiplerindendir. Üsteki örnekte **t = 1981, 91754, 'Tuna'** atamasıyla **t** demeti oluşturmuş. Demet içinde bulunan 3 eleman **a, b** ve **c** değişkenlerine **a, b, c = t** sekans açma işlemiyle atanmıştır.

```
>>> l = [12, 17, 52, 23]
>>> x, y, z, t, = l
>>> print("Değişkenlerin Değerleri:",x,y,z,t)
Değişkenlerin Değerleri: 12 17 52 23
```



Sekans açma işlemi listelerle de yapılabilir.

```
# Farklı Değere Sahip Değişkenler Tanımlama
>>> x,y,z = 2,4,6
>>> print("x =",x," y =",y," z =",z)
x = 2 , y = 4 , z = 6
```



"Farklı Değere Sahip Değişkenler Tanımlama" konusunda demetleri önceden farkında olmadan kullandık. `x,y,z = 2,4,6` atamasında eşitliğin sağ tarafta demet veri tipi bulunmakta. Demet içinde bulunan elemanlar eşitliğin sol tarafında bulunan değişkenlere sırasıyla atanarak sekans açma işlemi yapılır.

```
>>> sefer = "13:00","18:00",30
>>> print("Kalkış:",sefer[0],"Varış:",sefer[1],"Mola:",sefer[2],"Dakika")
Kalkış: 13:00 Varış: 18:00 Mola: 30 Dakika
```

Üsteki örnekte görüldüğü gibi demet içindeki elemanların doğrudan kullanılarak kod içinde kullanılabildiğini hatırlatmakta yarar var.

```
>>> sefer = "13:00","18:00",30
>>> kalkış,varış,molaDakika = sefer
>>> print("Kalkış:",kalkış,"Varış:",varış,"Mola:",molaDakika,"Dakika")
Kalkış: 13:00 Varış: 18:00 Mola: 30 Dakika
```

Demet elemanları kod içinde doğrudan kullanılabildiği gibi `kalkış,varış,molaDakika = sefer` sekans açma işlemiyle demet elemanları değişkenlere atanarak değişkenler kod içinde kullanılabilir.

```
>>> eposta = "tunapotur@mail.com"
>>> eposta.split("@")
['tunapotur', 'mail.com']
>>> print("Kullanıcı Adı:",kullanıcıAdi, " Alan Adı:",alanAdi)
Kullanıcı Adı: tunapotur Alan Adı: mail.com
>>> kullanıcıAdi, alanAdi = eposta.split("@")
```

Sekans açma işlemi mail adresi girilen kullanıcıların kullanıcı adlarını ve alan adlarını ayırmak için kullanılabilir. "tunapotur@mail.com" karakter dizisi `eposta.split("@")` işlemi ile @ işaretinin olduğu kısımdan ayrılır ve iki elemanlı ['tunapotur', 'mail.com'] listesi oluşur. `kullanıcıAdi, alanAdi = eposta.split("@")` sekans açma işlemiyle `kullanıcıAdi='tunapotur'` ve `alanAdi='mail.com'` değişken atamaları yapılır.

4.15. Demetlerin Kullanım Alanları

Gereksiz gözüktü de demetler oldukça yaygın kullanılan bir veri tipidir. Programların ayar (conf) dosyalarında bu veri tipi sıklıkla kullanılır. Python tabanlı bir web çatısı (framework) olan Django'nun `settings.py` adlı ayar dosyasında pek çok değer bir demet olarak saklanır.

```
TEMPLATE_DIRS = ('/home/projects/djprojects/blog/templates',)
```

Django web sayfalarının şablonlarını (template) hangi dizin altında saklayacağınızı belirlediğiniz ayar üstte görüldüğü şekilde demetler ile yapılır.

```
>>> TEMPLATE_DIRS = ('/home/projects/djprojects/blog/templates')
>>> type(TEMPLATE_DIRS)
<class 'str'>
```

Yeri gelmişken hatırlatmakta fayda var üstteki koddaki görüldüğü gibi eğer elemanın sağına virgül konmazsa bir elemanlı demet oluşturulmaz karakter dizisi oluşturulur.

4.15.1. Fonksiyonlara Belirsiz Sayıda Argüman Girmek

```
>>> def topla(a,b,c): # sadece 3 değer alan fonksiyon
    return a+b+c
>>> topla(3,5,7) # 3 değer için fonksiyon sorunsuz çalışır
15
```



```
# 3 değerden fazla değer hatalı
>>> toplama(3,5,7,9)
Traceback (most recent call last):
  File "<pyshell#18>", line 1, in <module>
    toplama(3,5,7,9)
TypeError: toplama() takes 3 positional
arguments but 4 were given
```

```
# 3 değerden az değer hatalı
>>> toplama(3,5)
Traceback (most recent call last):
  File "<pyshell#19>", line 1, in <module>
    toplama(3,5)
TypeError: toplama() missing 1 required
positional argument: 'c'
```

Eğer sabit sayıda argüman alan fonksiyon tanımlarsanız fonksiyona tanımladığınız sayıda eleman girmeniz gerekir. Üç argümanlı fonksiyona üçten fazla veya az sayıda argüman girişi yaparsanız fonksiyon hata verecektir.

```
>>> def toplama(*degerler): # * operatörü ile fonksiyona demet girilir
    return sum(degerler)
>>> toplama(3)
3
>>> toplama(3,5)
8
>>> toplama(3,5,7,9)
24
>>> toplama(1,3,5,7,9)
25
```

Değişken sayıda argümanlı fonksiyon tanımlamak için demetler(tuple) kullanılır. `def toplama(*degerler):` tanımında argümandan önce kullanılan `*` operatörü fonksiyonun aldığı argümanın demet tipinde olacağını belirtir. Üsteki örnekte tanımlanan fonksiyona istenen sayıda argüman gönderilebiliyor. Demetlerin `3,5,7,9` şeklinde parantezler olmadan kullanılabildiği önceden belirtilmişti. `toplama(3,5,7,9)` şeklinde fonksiyona virgül ile değer girildiğinde aslında yaptığınız fonksiyon demet girişi yapmaktır.

```
>>> def enKucukBuyuk(*degerler): # * operatörü ile fonksiyona demet girilir
    return min(degerler),max(degerler)
# fonksiyonlar demetler ile birden çok değer döndürebilir.
>>> enKucukBuyuk(3,9,1,-13,-57,3.14,-1.44)
(-57, 9)
>>> enKucukBuyuk(9,12,1012,-97)
(-97, 1012)
```

Fonksiyonlar bir den çok değer döndüklerinde demet(tuple) tipinde değer dönerler. Üste tanımlanan fonksiyon demet tipinde girilen argümanların en küçük ve en büyüğünü demet tipinde döner. Fonksiyonların döndürdüğü değerlerin değişmemesi önemlidir yoksa fonksiyonun yanlış çalıştığı düşünülebilir. Demetlerin değiştirilemez olması birden çok değer dönen fonksiyonlar için kullanışlı bir özelliktir.

```
# print(*elemanlar) print fonksiyonu argüman olarak demet alır
>>> print('Tuna')
Tuna
>>> print(3,5,7)
3 5 7
>>> print(3.14,1.22,-57.41,73.54)
3.14 1.22 -57.41 73.54
>>> print(9,1.37,'Tuna')
9 1.37 Tuna
```

`print(*elemanlar)` fonksiyonu argüman olarak demet aldığından istenen sayıda değer girilebilir.

4.15.2. Format Metodu İle Demetlerin Kullanımı

```
>>> print("{0} {1} boyunda ve {2} yaşındadır".format("Can",1.85,16))
Can 1.85 boyunda ve 16 yaşındadır
```

Demetlerin bir diğer kullanım alanlarından biri ileride göreceğimiz `format()` metodudur. `format()` metodu argüman olarak demet alır. Argüman olarak girilen demet verisiyle `format()` metoduna istenen sayıda veri girilebilir. Üsteki örnekte görüldüğü gibi format metodu argüman olarak aldığı demet verisinin elemanlarını belirtilen sırayla karakter verisine yerleştirir.

```
>>> veriler = ["Can",1.85,16]
>>> print("{0} {1} boyunda ve {2} yaşındadır".format(*veriler))
Can 1.85 boyunda ve 16 yaşındadır
```

`*` operatörü `format()` metoduna girilen liste tipinde argümanı demet verisine çevirir ve liste içinde bulunan elemanlar sırasıyla belirtilen alanlara yerleştirilir.

5. Karakter Dizileri (string) Veri Tipi

Karakter dizilerini(string) daha önce basit bir şekilde görmüştük bu bölümde karakter dizilerini detaylı bir şekilde işleyeceğiz. Karakter dizileri yapıları gereği listelere oldukça benzerler. Karakter dizileri tıpkı listeler gibi, indekslenirler, parçalanırlar ve üzerinde değişik işlemler yapabildiğimiz fonksiyonları barındırırlar. Ancak karakter dizilerinin listelerden önemli farkları bulunmaktadır. Karakter dizileri değiştirilemez(**immutable**) bir veri tipidir.

5.1. Karakter Dizilerini Oluşturmak ve str() Fonksiyonu

```
>>> adı = 'Ahmet Tuna POTUR'
>>> print("Tipi:", type(adı), "Değeri:", adı)
Tipi: <class 'str'> Değeri: Ahmet Tuna POTUR
```

adı değişkenine tırnak(' ') içinde 'Ahmet Tuna POTUR' metni atayarak **karakter dizi(string)** değişkeni oluşturduk. **type(adı)** işlemiyle adı değişkeninin **str** tipinde olduğu ve **adı** değişkeni içinde bulunan karakter dizisi **print()** fonksiyonuyla ekrana yazdırıldı.

```
>>> #Tek tırnak ile      >>> #Çift Tırnak ile      >>> #Üç Tırnak ile
>>> print('Tuna POTUR') >>> print("Tuna POTUR") >>> print("""Tuna POTUR""")
Tuna POTUR             Tuna POTUR             Tuna POTUR
```

Tırnak içerisinde yazılan her ifade karakter verisi olarak algılanır. Python'da **string** oluşturmak için **çift tırnak("")**, **tek tırnak('')** veya **üç tırnak (""" """)** sembolleri kullanılır. Üstteki kodlarda gördüğümüz gibi ekrana yazdırılan metinlerin hiçbir farkı yok.

```
>>> a = str()    >>> a = str(315)    >>> a = str(-3.14)    >>> a = str([1,3,5])
>>> a            >>> a            >>> a            >>> a
''              '315'           '-3.14'         '[1, 3, 5]'
```

str() fonksiyonu içinde değer olmadan değişkene atanırsa boş karakter dizisi oluşturulur. **str()** fonksiyonuna girilen argüman tip dönüşümü yapılarak karakter dizisine dönüştürülür. Üsteki örnek kodda sırasıyla boş karakter verisi, **int**, **float** ve **list** veri tipleri karakter tipine dönüştürülerek değişkene atanıyor.

```
>>> a = ''      >>> a = ""      >>> a = """"      >>> a = str()
>>> a          >>> a          >>> a          >>> a
''            ''            ''            ''
```

Kodunuzun içinde sonradan doldurulması için önceden boş karakter dizisi oluşturmaya ihtiyaç duyabilirsiniz. Üsteki örnekte verilen dört kod örneğinde boş karakter dizisinin nasıl oluşturulduğu gösteriliyor.

```
>>> print('Lüleburgaz')    >>> print("Lüleburgaz")
```

SyntaxError: EOL while scanning string literal

Karakter dizini oluştururken hangi tırnak işaretini kullandıysanız karakter dizisini o tırnakla bitirmelisiniz. Eğer karakter dizinin başındaki ve sonundaki tırnak farklıysa hata oluşur.

```
>>> print('Can'ın 15. yaş günü')
```

SyntaxError: invalid syntax

tek tırnak('') ile oluşturulan bir karakter verisi kullanıyorsanız içinde **Can'ın** metnindeki gibi bir tırnak kullanamazsınız. Python karakter dizisini oluşturmaya başladığında önce **tek tırnak('')** ile başlanan karakteri okur ve karakter dizisi oluşturulacağını anlar. **İkinci tek tırnak** geldiğinde karakter dizisinin bittiğini düşünür. Eğer **ikinci tek tırnaktan** sonra yine bir tırnak daha okunursa Python burada ne yapılmak istendiğini anlamayacak ve hata verecektir.

```
>>> print("Can'ın 15. yaş günü")
Can'ın 15. yaş günü
```

Eğer karakter dizisi içerisinde **Can'ın** metnindeki gibi **tek tırnak(')** kullanmak istiyorsanız oluşturmak istediğiniz karakter dizisini **çift tırnak("")** içerisinde oluşturmalısınız.

```
>>> print('Bugün öğrenciler "Dostluk" adlı şiiri okudular.')
Bugün öğrenciler "Dostluk" adlı şiiri okudular.
```

Karakter dizisi içinde **çift tırnak("")** kullanılacaksa karakter dizisi **tek tırnak('')** içinde tanımlanabilir.

```
>>> print("""üç tırnak
birden çok satırlı
metinler için kullanılır""")
üç tırnak
birden çok satırlı
metinler için kullanılır
```

Üç tırnak (""") birden fazla satıra yayılmış karakter dizilerini tanımlamak için kullanılır.

```
>>> print("""
[A]=====PYTHON====[-][o][x]
|               |
|      Merhaba Dünya      |
|      Penceresine      |
|      Hoşgeldiniz      |
|               |
|=====|
""")
# Ekran Çıktısı
[A]=====PYTHON====[-][o][x]
|               |
|      Merhaba Dünya      |
|      Penceresine      |
|      Hoşgeldiniz      |
|               |
|=====|
```

Üsteki gibi bir çıktı vermek istiyorsanız **üç tırnak (""")** kullanmanız gerekir.

```
>>> ada_lovelace="""Ada Lovelace sahip olduğu maddi kaynak sayesinde
Babbage'in en büyük destekçilerinden biriydi.
Kendisi de matematikçi olan Ada Analitik Makine üzerine yazdığı notlarda
Bernouli sayılarını hesaplamak için Analitik Motorun Algoritmasını
tanımlıyordu. İlk algoritmayı tasarladığı için
Ada ilk bilgisayar programcısı olarak kabul edilir.
"""
```

```
>>> print(ada_lovelace)
Ada Lovelace sahip olduğu maddi kaynak sayesinde
Babbage'in en büyük destekçilerinden biriydi.
Kendisi de matematikçi olan Ada Analitik Makine üzerine yazdığı notlarda
Bernouli sayılarını hesaplamak için Analitik Motorun Algoritmasını
tanımlıyordu. İlk algoritmayı tasarladığı için
Ada ilk bilgisayar programcısı olarak kabul edilir.
```

İsterseniz **üç tırnak (""")** ile yazdığınız karakter dizisini bir değişkene atayabilirsiniz.

5.2. Karakter Dizisi Elemanlarına Erişmek (İndeks)

demet	=	"	M	E	R	H	A	B	A		D	Ü	N	Y	A	"
index	→		0	1	2	3	4	5	6	7	8	9	10	11	12	
			-13	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1	

```
>>> metin = "Merhaba Dünya" # Karakter dizileri indekslenir
>>> metin[0]          >>> metin[1]          >>> metin[7]          >>> metin[-1]
'M'                  'e'                  ' '                  'a'
```

Karakter dizisindeki harflere aynı liste ve demetler gibi indeks değerleriyle erişilir. Karakter dizisi içindeki her harf sırasıyla 0'dan başlayan indeks numaralarıyla indekslenir. Karakter dizisi içindeki hangi harf çağrılacaksa karakter dizisi indeks operatörüne harf indeksi girilir. Her harf bir karakter olduğu gibi kelimeler arasında kullanılan boşluklarda karakterdir. Üsteki örnekte **metin[7]** ile 7. indekste bulunan boşluk karakteri çağrılmış.

5.3. len() Fonksiyonu

```
>>> metin = "Merhaba Dünya"
>>> len(metin)
13
```

`len()` fonksiyonu karakter dizisinin uzunluğunu ölçer. Üstteki örnekte `len()` fonksiyonu metin karakter dizisinin içinde 13 karakter dolduğunu dönmüş.

5.4. Karakter Dizileri Değiştirilemez(Immutable)

```
>>> metin = "Merhaba Dünya"
# u karakteri ü ile değiştirilemez
>>> metin[10] = 'ü'
Traceback (most recent call last):
  File "<pyshell#8>", line 1, in <module>
    metin[10] = 'ü'
TypeError: 'str' object does not support item assignment

# u karakteri silinemez
>>> del metin[10]
Traceback (most recent call last):
  File "<pyshell#9>", line 1, in <module>
    del metin[10]
TypeError: 'str' object doesn't support item deletion
```

Karakter dizileri aynı demetler gibi değiştirilemez(`immutable`) özelliktedir. Üstteki örnekte Python hatalı 'u' harfi 'ü' ile değiştirilmeye çalışıldığı için hata vermiş. Aynı şekilde 'u' harfi silinmeye çalışıldığında da Python hata verecektir.

5.5. İndeks Sınırlarına Çıkmak, Hatalı İndeks Değeri

```
>>> metin = "Merhaba Dünya" # karakter dizisi sınırları dışına çıkılamaz
>>> metin[13]
Traceback (most recent call last):
  File "<pyshell#11>", line 1, in <module>
    metin[13]
IndexError: string index out of range

>>> metin[-14]
Traceback (most recent call last):
  File "<pyshell#13>", line 1, in <module>
    metin[-14]
IndexError: string index out of range
```

Karakter dizilerinde hatalı indeks değeri vermek hataya yol açar.

5.6. Karakter Metotları

Bu bölümde karakter dizisine(`str`) ait metotlar işlenecek. Kitabı hazırlarken kullanılan Python 3.5.4 versiyonunda karakter dizisi içinde 44 adet metot tanımlanmıştı. Bu kadar çok metodu burada tanıtmak gerekli değil. Bu yüzden 44 metodun içinden en önemli metotlar üzerinde durulacak. Daha iyi anlaşılacağı için benzer özellikte olan metotları gruplayarak işlenecek. Tüm karakter metotlarının kullanımını görmek için alta ki linki kullanabilirsiniz.

https://www.tutorialspoint.com/python/python_strings.htm

5.6.1. format() Metodu

Karakter dizilerinin içinde değerlerin yazdırılması büyük bir ihtiyaçtır. Karakter dizisinin metotlarından bir olan `format()` metodu değerlerin karakter dizisi içinde yazdırılması için kullanılabilecek en kullanışlı yollardan biridir.

`format()` metodu için kapsamlı kaynakları bulabileceğiniz adresler;

<https://docs.python.org/3.5/library/string.html#formatspec>

<https://pyformat.info/>

```
>>> "{} , {} ve {} pozitif tek sayılardır".format(3,5,7)
'3,5 ve 7 pozitif tek sayılardır'
```

`format()` metodu karakter dizisinin içinde bulunan süslü parantezlerin {} yerine sırasıyla aldığı argümanların değerlerini yerleştirir.

```
"{}, {} ve {} pozitif tek sayılardır".format(3,5,7)
```

Karakter dizileri üzerinde diğer tüm verilerde olduğu gibi doğrudan çalışılabilir. Karakter dizisinden sonra nokta ile **format()** metodu çağrılır. Çağrılan **format()** metodu doğrudan kendi karakter dizisi üzerinde etkili olacaktır.



```
" {}, {} ve {} pozitif tek sayılardır".format(3,5,7)
```

.format(3,5,7) format metodunun içinde bulunan argümanlar karakter dizisi içinde bulunan süslü parantezlerin **{}** yerlerine sırasıyla yerleştirilir.

```
'3,5 ve 7 pozitif tek sayılardır'
```

format() metodu aldığı argümanların değerlerini karakter katarına yerleştirir ve karakter dizisi son haliyle ekrana yazdırılır.

```
>>> adı,yaş,boy = 'Can',15,1.85
>>> print("{} {} yaşında {} boyundadır".format(adı,yaş,boy))
Can 15 yaşında 1.85 boyundadır
```

Üsteki örnek incelendiğinde karakter katarı içinde bulunan **{}** süslü parantezler yerine, format metodunda bulunan **adı, yaş ve boy** değişkenlerinin değerleri sırasıyla yerleştirilmiş. Son durumda karakter dizisi değişkenlerin değerleriyle birlikte kullanıcı için anlamlı bir şekilde ekrana yazdırılır.

```
>>> "{} {} {}".format(1,2)
Traceback (most recent call last):
  File "<pyshell#13>", line 1, in <module>
    "{} {} {}".format(1,2)
IndexError: tuple index out of range
```

format() metodunun argüman sayısı **{}** süslü parantez sayısından az olursa hata oluşur.

```
>>> "{} {} {}".format(1,2,3,4,5,6)
'1 2 3'
```

Eğer **{}** süslü parantezden fazla değer **format()** metoduna argüman olarak girilirse, sırasıyla **{}** süslü parantez sayısı kadar argüman kullanılır. Üsteki örnekte **format()** metodunun ilk üç argümanı karakter dizisine yerleştirilmiş.

```
>>> '{0} {1} {0}'.format('Müdür','Öğretmen')
'Müdür Öğretmen Müdür'
```

format() metodu içindeki argümanlar karakter dizisi içinde tekrar edebilir. **format()** argümanları metin içinde tekrar edilecekse süslü parantez içinde argüman indeks numarası kullanılmalıdır.

```
>>> a,b = 5,7
>>> metin = "{} + {} = {}"
>>> sonuç = metin.format(a,b,a+b)
>>> print(sonuç)
5 + 7 = 12
```

Şimdiye kadar kullanılan örneklerin tümünde karakter dizisi üzerinde doğrudan çalışıldı. Bu örnekte karakter dizisi **metin** değişkenine atanıyor. **metin** değişkeninde bulunan karakter dizisi **format()** metodunun argümanlarıyla şekillendirilerek **sonuç** değişkenine atanıyor.

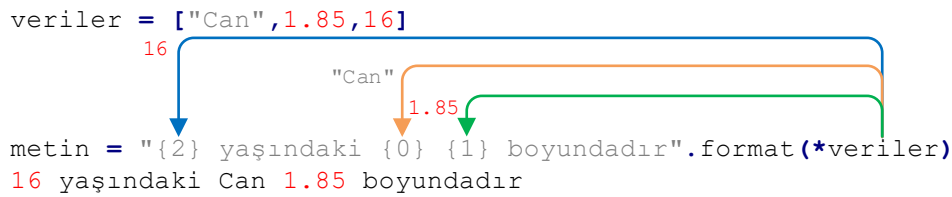
```
>>> adı,yaş,boy = 'Can',15,1.85
>>> print("{1} yaşındaki {0} {2} boyundadır".format(adı,yaş,boy))
15 yaşındaki Can 1.85 boyundadır
```

format() argümanları süslü parantezlerin içine sıra numarası belirtilerek istenen sırada karakter dizisine yerleştirilebilir.

```
>>> veriler = ["Can",1.85,16]
>>> metin = "{2} yaşındaki {0} {1} boyundadır".format(*veriler)
>>> print(metin)
16 yaşındaki Can 1.85 boyundadır
```

Fonksiyonlara ve metotlara girilen **list**, **tuple** ve **karakter** verisi tipinde argümanların başına ***** operatörü getirilirse argümanlar üzerinde **sekans açma işlemi** yapılır.

```
veriler = ["Can",1.85,16]
16
"Can"
1.85
metin = "{2} yaşındaki {0} {1} boyundadır".format(*veriler)
16 yaşındaki Can 1.85 boyundadır
```



Sekans açma işlemiyle dizi elemanlarına ayrılır. Dizi elemanları süslü parantezlerin **{}** içine sıra numarası belirtilerek karakter dizisine yerleştirilir.

```
# değeri girildiği gibi tırnaklarıyla yazdırma
>>> "Değeri Tırnak ile Birlikte Yazdırma : {!r} ".format('Değer')
'Değeri Tırnak ile Birlikte Yazdırma : 'Değer' "
```

{!r} şeklinde kullanımda karakter değeri tanımlandığı tırnak karakterleri ile birlikte yazdırılır.

```
# değeri tırnaksız yazdırma yazdırma. {} ile aynı işlem
>>> "Değeri Tırnak Olmadan Yazdırma : {!s} ".format('Değer')
'Değeri Tırnak Olmadan Yazdırma : Değer '
```

{!s} içerisinde kullanılan **!s** değer string olduğunu ve tırnaksız yazdırılacağını belirtir. **{}** kullanımı ile aynı etkiyi yaratır.

```
>>> '{x} * {y} = {sonuç}'.format(x=3,y=5,sonuç=3*5)
'3 * 5 = 15'
```

Küçük karakter dizileri içinde verileri **{}** ile görüntülemek kafanızı karıştırmaz. Fakat büyük bir metin ve çok sayıda değişken değeri görüntülediğiniz bir karakter dizisi kullanıyorsanız metin içinde kaybolmanız çok doğaldır. **format()** metodu içinde değerleri değişkenlere atayarak süslü parantez **{}** ile kullanabilirsiniz.

```
>>> adı = 'Tuna'
>>> soyadı= 'Potur'
>>> '{adı} {soyadı}'.format(adı=adı,soyadı=soyadı)
'Tuna Potur'
```

Değişkenleri **{}** içinde doğrudan kullanamazsınız. **{}** içinde değişkenleri kullanmak için **format()** metodu içinde değişkeni kendisine atamalısınız.

```
>>> veriler = [2,4,8,16,32]
>>> '1. indeks = {v[1]} ve 3. indeks = {v[3]}'.format(v=veriler)
'1. indeks = 4 ve 3. indeks = 16'
```

format() metodunda dizileri kolay kullanmak için dizileri tek harflik bir ifadeye atayabilirsiniz. Üsteki örnekte **veriler** dizisi **v** harfine atanmış. Karakter dizisinde süslü parantezler içinde **{v[1]}** ve **{v[3]}** kullanımı ile veriler dizisi 1 ve 3 indeksindeki elemanlar kullanılıyor.

```
>>> kisi = {'adı':'Ahmet','soyadı':'Çelik','babaAdı':'Cenk'}
>>> print('Adı: {adı} {soyadı}\nBaba Adı: {babaAdı}'.format(**kisi))
Adı: Ahmet Çelik
Baba Adı: Cenk
```

İleride göreceğimiz sözlük verisi format metodunda ****** operatörü yardımıyla kolay bir şekilde kullanılabilir. Sözlük veri tipinde indeksler yerine anahtarlar kullanılır. Veriler sözlük içinden anahtarlar yardımıyla çağrılır. Sözlük içindeki **'adı'**, **'soyadı'** ve **'babaAdı'** sözlük verisinin anahtarlarıdır. Anahtarların sağında **:** ile ayrılan alandakilerde verilerdir. Üsteki örnekte görüldüğü gibi format içinde sözlük değişkeni üzerinde ****** kullanıldığında sekans açmaya benze bir şekilde veriler metin içine kullanılabilir. Metin içinde sözlük verilerine **{ }** içine anahtar kelimeleri **{adı}** gibi yazılarak erişir.

5.6.1.1. Karakter Dizisi İçine Girilen Verilen Şekillendirilmesi

{:} Süslü parantez içinde iki nokta üst üste karakteri ile verilerin karakter dizisi içinde nasıl görüleceğe belirtilir. İki nokta üst üste **:** karakteri format metoduna değerlerin nasıl şekillendirileceğiyle ilgili büyük kontrol imkanı verir.

```
>>> "{:.2f} {:.3f} {:.4f}".format(3.14159,2.7182818,1.145793)
'3.14 2.718 1.1458'
```

: karakterinin en sık kullanılacağı yerler **float** sayıların kaç ondalıkla kullanılacağını belirtildiği uygulamalardır. Üsteki örnekte **:** karakterinden sonra **.2f**, **.3f** ve **.4f** ile float sayıların 2, 3 ve 4 ondalık hanesine sahip olacağı belirtilmiş. **f** karakteri float sayılar için düzenleme yapılacağını belirtir.

```
>>> 0.2 + 0.1
0.30000000000000004
>>> 3 * 0.1
0.30000000000000004

>>> "{:.2f}".format(0.2 + 0.1)
'0.30'
>>> "{:.2f}".format(3 * 0.1)
'0.30'
```

float sayılar arasında yapılan işlemlerde bazen kullanışlı olmayan değerler çıkabilir. Üsteki örnekte **{:.2f}** ifadesiyle format() metodu işlem sonuçları 2 ondalık olarak düzgün şekilde görüntüler.

```
# değerler olduğu gibi görüntülenir
>>> '{:f}; {:f}'.format(3.14, -3.14)
'3.140000; -3.140000'
>>> '{:-f}; {:-f}'.format(3.14, -3.14)
'3.140000; -3.140000'
# değerler işaretleriyle görüntülenir
>>> '{:+f}; {:+f}'.format(3.14, -3.14)
'+3.140000; -3.140000'
# değerler eğer pozitifse önü boş negatifse önü - ile görüntülenir.
>>> '{: f}; {: f}'.format(3.14, -3.14)
' 3.140000; -3.140000'
```

Üsteki örnekte görüldüğü gibi değerler işaretleriyle birlikte görüntülenebilir.

```
>>> # format binary sayıları da destekler
>>> "int:{0:d}; float:{0:f}; hex:{0:x}; oct:{0:o}; bin:{0:b}".format(41)
'int:41; float:41.000000; hex:29; oct:51; bin:101001'
```

: ile birlikte değer **tam**, **float**, **hexadecimal**, **octal** ve **binary** sayılara çevrilerek düzenlenebilir.

```
>>> # binary sayıların ön takıları 0x, 0o, ve 0b # ile kullanılabilir
>>> "hex:{0:#x}; oct:{0:#o}; bin:{0:#b}".format(41)
'hex:0x29; oct:0o51; bin:0b101001'
```

kullanarak, **hexadecimal**, **octal** ve **binary** sayıların ön takıları yazdırılır.

```
>>> # virgöl binlik haneler için kullanılır
>>> '{:,}'.format(1500250125)      >>> '{:,.2f}'.format(1500250125.6492)
'1,500,250,125'                  '1,500,250,125.65'
```

Türkçe 'de virgöl ondalık sayılarda, nokta binlik haneleri ayırmakta kullanılır. Fakat İngilizce 'de ve birçok diğer dilde bu durum tam tersidir. Üstteki örnekte büyük sayıların anlaşılması için sayının virgöl ile binlik haneleri ayırmakta nasıl kullanıldığı gösterilmiş.

```
>>> '{:.2%}'.format(19/22)      >>> '{:,.2%}'.format(0.8636)
'86.36%'                       '86.36%'
```

% kullanarak ondalıklı sayılar yüzde olarak ifade edilir.

```
>>> '{:6d}'.format(57)          >>> '{:06d}'.format(57)
'      57'                     '000057'
```

Ekrana yazdırılacak sayılar başlarına boşluk veya 0 eklenerek yazdırılabilir.

```
>>> '{:10}'.format('deneme')
'deneme      '
```

; karakterinin sağına girilen rakam format() metodunun gönderdiği değerin kaç karakter içine yazdırılacağını belirtir. Üsteki örneklerde **{:10}** kullanımında 'deneme' karakteri sola dayalı olarak 10 karakterlik alana yazdırılır.

```
>>> '{:<10}'.format('deneme')    >>> '{:*<10}'.format('deneme')
'deneme      '                  'deneme*****'
```

Normal kullanımda karakter yazılımı sola dayalı olarak yapılır. ; karakterinden sonra < kullanmakta karakterin sola dayalı olacağını belirtir. < karakterin önüne yazılan karakter format() metodunun gönderdiği karakterden boş kalan yerlere yazdırılır. Örnekte *< ile 'deneme' metninden sonra boşluklar '*****' ile doldurulmuş.

```
>>> '{:>10}'.format('deneme')    >>> '{:*>10}'.format('deneme')
'      deneme'                  '*****deneme'
```

format() metodunun gönderdiği metni sağa dayalı yazdırmak için > karakteri kullanılır. > karakterinin önüne yazılan karakter ile boşluklar doldurulur. Örnekte *> ile boşluklar 'deneme' metninden önce '*****' ile doldurulmuş.

```
>>> '{:^10}'.format('deneme')    >>> '{:*^10}'.format('deneme')
'  deneme  '                    '**deneme**'
```

format() metodunun gönderdiği metni ortalarak yazdırmak için ^ karakteri kullanılır. ^ karakterinin önüne yazılan karakter ile boşluklar doldurulur. Örnekte *^ ile boşluklar '**' ile doldurulmuş.

```
>>> '{:*^6}'.format('iki')
'*iki**'
```

Eğer ortalamak istediğiniz metin belirttiğiniz alanın tam ortasına yerleşmiyorsa fazla karakter sağ tarafa eklenir.

```
>>> '{:.4}'.format('Lüleburgaz')  >>> '{:10.4}'.format('Lüleburgaz')
'Lüle'                            'Lüle      '
>>> '{:*>10.4}'.format('Lüleburgaz') >>> '{:*^10.4}'.format('Lüleburgaz')
'*****Lüle'                     '***Lüle***'
```

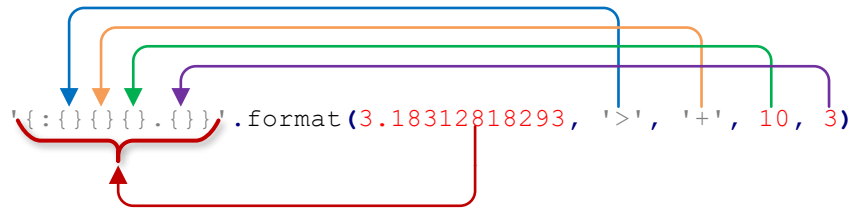

: karakterinden sonra .4 şeklinde yazılan sayı format() metodu içinde ekrana yazdırılacak karakterin baştan kaç harfinin yazdırılacağını belirtir. {:.4} 'Lüleburgaz' kelimesinin ilk dört harfi olan 'Lüle' kelimesini ekrana yazdırır. Eğer : karakterinden sonra float sayılara benzer şekilde nokta ile bir sayı yazılırsa belirtilen boşluk içinde kaç harf yazdırılacağı belirtilir. Üsteki örnekte {:.10.4} ifadesinde noktanın solundaki 10 sayısı yazdırılacak değerin kaç boşluk içine yazdırılacağını belirtir. 4 değeri ise değerin kaç karakterinin yazdırılacağını belirtir. {:*>10.4} ve {:*^10.4} şeklindeki kullanımda belirtilen boşlukların yerine * karakteri yerleştirilir.

```
>>> from datetime import datetime
>>> '{:%Y-%m-%d %H:%M:%S}'.format(datetime(2017,12,3,7,5,9))
'2017-12-03 07:05:09'
```

'{:%Y-%m-%d %H:%M}' kullanımıyla tarih verili şekillendirilir. %Y yıl, %m ay, %d gün, %H saat, %M dakika ve %S saniye verileri için kullanılır. from datetime import datetime ifadesi datetime modülünün kodunuzun içine yüklenmesi için kullanılır. Modüller konusu ileride görülecek.

```
>>> '{:>+10.3}'.format(3.18312818293)
'      +3.18'
>>> # format metodundan düzenleme bilgisinin gönderilmesi
>>> '{:({}{}){}}'.format(3.18312818293, '>', '+', 10, 3)
'      +3.18'
```

Metin içerisinde şekillendirilecek değerin nasıl şekillendirileceği iç içe süslü parantez {:} kullanılarak yapılabilir.



Süslü parantez içine iki noktadan sonra kullanılan süslü parantezlere değerin nasıl şekillendirileceği bilgisi format metodu içinden sırasıyla gönderilir. Üsteki şekilde görüldüğü {:000.0} süslü parantezi format metodundan değeri alır. Süslü parantez içinde iki noktadan sonra girilen {000.0} parantez grubuna format metodunda değerdan sonra gelen değerler sırasıyla gönderilir.

```
>>> '{:({}{isaret}){}}'.format(3.18312818293, '>', 10, 3, isaret='+')
'      +3.18'
```

format() metodu içinden değerleri metne değişkenle gönderebildiğimiz gibi şekillendirme bilgilerini de gönderebiliriz. Süslü parantez içinde kullanılan {:({isaret}){}} kullanılan süslü paranteze değişken adı yazılarak format metodu içindeki değişken tanımından isaret='+' düzenleme bilgisi gönderilebilir.

```
>>> '{:*^15}'.format('deneme')
'****deneme*****'
>>> # Değişkenler ile formatlama
>>> '{:({krktr}{yer}{boy})}'.format('deneme',krktr='*',yer='^',boy='15')
'****deneme*****'
```

Değişkenler yardımıyla karakterler şekillendirilebilir. {:({krktr}{yer}{boy})} ifadesiyle : karakterinden sonra süslü parantez içine format metodunda kullanılan metni şekillendirecek değişkenler yazılır.

```
>>> '{:08.4f}'.format(3.1482193)
'003.1482'
>>> # Değişkenler ile formatlama
>>> '{:({krktr}{yer}{boy})f}'.format(3.1482193,krktr=0,yer=8,boy=4)
'003.1482'
```

Değişkenler yardımıyla float sayısal ifadeleri üste örnekte görüldüğü gibi şekillendirilir.

```
>>> '{0:.{boy}}={1:.{boy}f}'.format('Lüleburgaz',3.1482953,boy=4)
'Lüle=3.1483'
```

Üste değişkenler yardımıyla karakter ve sayısal ifadelerin birlikte şekillendirilmesi örnek verilmiş. .{boy} değişken adının öndü nokta olduğuna dikkat edin. Nokta kullanımı kaç karakter kullanılacağını belirtir.

```
>>> '{0:{boy}}={1:{boy}}'.format('Lüleburgaz',3.1482193,boy='.4')
'Lüle=3.148'
```

Üsteki örneğin bir önceki örneğe göre farkı '.4' ile nokta bilgisi değişken içinde gönderilmiş.

```
>>> from datetime import datetime
>>> dt = datetime(2017,12,3,4,5)
>>> '{:{dfmt} {tfmt}}'.format(dt, dfmt='%Y-%m-%d', tfmt='%H:%M')
'2017-12-03 04:05'
```

Zaman bilgisinin formatlanmasında da değişkenler kullanılabilir.

```
>>> değerler = '{0:2d} {1:3d} {2:4d}'
>>> for x in range(1, 11):
    print(değerler.format(x, x*x, x*x*x))

>>> değerler = '{0:02d} {1:03d} {2:04d}'
>>> for x in range(1, 11):
    print(değerler.format(x, x*x, x*x*x))
```

```
1  1  1
2  4  8
3  9 27
4 16 64
5 25 125
6 36 216
7 49 343
8 64 512
9 81 729
10 100 1000

01 001 0001
02 004 0008
03 009 0027
04 016 0064
05 025 0125
06 036 0216
07 049 0343
08 064 0512
09 081 0729
10 100 1000
```

Üste karesi ve küpü alınan sayıların format() metodu ile düzgün bir şekilde ekrana yazdırılması örnek verilmiştir. Her iki örnekte de her sütun için en büyük sayının hanesi kadar boşluk ve '0' karakteri önceden hazırlanmıştır, boşluk ve '0' karakterleri işlem sonuçlarından arta kalan alanlara yazdırılmıştır. Birinci sütun için 2, ikinci sütun için 3 ve üçüncü sütun için 4 alanlık boşluk ve '0' işlem sonuçlarından arta kalan alanlara yazdırılmıştır.

5.6.2. zfill() Metodu

```
>>> '-3.14'.zfill(7)
'-003.14'
>>> '12'.zfill(5)
'00012'
>>> 'Tuna'.zfill(10)
'000000Tuna'
>>> 'Ahmet Tuna'.zfill(10)
'Ahmet Tuna'
>>> len('Ahmet Tuna')
10

>>> '3.14'.zfill(7)
'0003.14'
>>> '3.14159'.zfill(5)
'3.14159'
>>> 'Ahmet'.zfill(10)
'00000Ahmet'
>>> 'Merhaba Dünya'.zfill(10)
'Merhaba Dünya'
>>> len('Merhaba Dünya')
13
```

zfill() metodu karakter dizinin kaç karakterlik alan içinde oluşturulacağını belirtir. Eğer karakter dizisi zfill() metoduna girilen değerden az sayıda karaktere sahipse, karakter dizisinden arta kalan alanlara sıfır(0) karakteri yerleştirilir. Yerleştirilen 0 karakterleri karakter dizisinin sol başına yerleştirilir. Karakter dizisi zfill() metodunda belirtilenden çok karaktere sahipse 0 değerleri yerleştirilmez. Örnekte verilen '12'.zfill(5) koduna bakalım; zfill(5) metodunda karakter dizisinin 5 karakterlik alana yazılacağı belirtilmiş. '12' Karakter dizisi 2 elemanlı 5 - 2 = 3 sıfır(0) karakteri karakter dizisinin sol başına yerleştirilerek '00012' karakter dizisi elde edilir.

```
>>> deęerler = [1, 17, 27, 47, 139, 1098]
>>> for i in deęerler:
    str(i).zfill(4)
```

```
'0001'
'0017'
'0027'
'0047'
'0139'
'1098'
```

Üsteki örnekte deęerler isimli sayı listesi var. İleride göreceğimiz **for** döngüsü yardımıyla sayı listesinin elemanları tek tek başlarına **0** eklenerek **4** haneli şekilde yazdırılmış.

5.6.3. replace() Metodu

```
>>> metin = 'Ahmet Tolga POTUR'
>>> metin.replace('Tolga', 'Tuna')
'Ahmet Tuna POTUR'
```

replace() metodu karakter dizisi içinde bulunan kelimeleri metot içinde belirtilen karakterler ile değiştirir. **replace()** metodunun ilk parametresi deęişecek dizisi, ikinci parametresi yeni karakter dizisidir. Üsteki örnekte 'Tolga' karakter dizisi 'Tuna' karakter dizisi ile yer deęiştirmiş.

```
>>> 'Müdür Müdür'.replace('Müdür', 'Öğretmen')
'Öğretmen Öğretmen'
>>> 'Müdür Müdür'.replace('Müdür', 'Öğretmen', 1)
'Öğretmen Müdür'
```

replace() metodunun üçüncü parametresi deęişim işleminin kaç kere yapılacağını belirtir. Üstteki örnekte 'Müdür' karakter dizisinden iki tane var. 'Müdür' karakter dizisinin kaçının 'Öğretmen' karakter dizisi ile deęiştirileceęi belirtilmedięi için tüm 'Müdür' karakter dizileri deęiştirilmiş. İkinci örnekte **replace()** metodunun üçüncü parametresinde **1** girildięi için 'Müdür' karakter dizilerinden sadece bir tanesi deęiştirilmiştir.

5.6.4. index(), find(), rindex() ve rfind() Metotları

```
>>> metin = 'Ada ilk bilgisayar programcısı olarak kabul edilir.'
>>> metin.index('program')
19
>>> metin.index('Ada')
0
# find ve index arasındaki fark
>>> metin.find('Python')
-1
>>> metin.find('program')
19
>>> metin.find('Ada')
0
# index aranan yoksa hata veri
>>> metin.index('Python')
Traceback (most recent call last):
  File "<pyshell#14>", line 1, in <module>
    metin.index('Python')
ValueError: substring not found
```

Karakter dizisi içinde karakter veya metin aramak için **index()** veya **find()** metotları kullanılır. **index()** ve **find()** metotları karakterleri aramaya soldan başlar. Bu metotlar eęer aranan karakterler bulunursa solda bulunan ilk karakterin index deęeri döner. **find()** ve **index()** metotları arasındaki tek fark eęer aranan karakter bulunmazsa **find()** metodu **-1** döner, **index()** metodu hata döner. Bundan yapacaęım konu anlatımlarında ve örneklerde **find()** metodunu kullanacaęım.

```
>>> metin = 'alacaklar: 1 paket makarna, 1 paket tuz, 2 paket şeker'
>>> metin.find('1')          >>> metin.find('1',12)
11                          28
>>> metin.find('paket',12,28) >>> metin.find('2',30,50)
13                          41
>>> metin.rindex('paket')    >>> metin.rfind('paket',15,40)
43                          30
```

Karakter dizisi içinde aynı değerden birden çok varsa **find()** metodu solda bulunan ilk değerın adresini döner. **find()** ve **index()** metotları ikinci ve üçüncü parametrelere sahiptir. İlk parametre aranacak değer, ikinci parametre aramanın hangi elemandan başlatılacağı ve üçüncü parametre aramanın hangi elemanda bitirileceğidir.

```
>>> metin = 'alacaklar: 1 paket makarna, 1 paket tuz, 2 paket şeker'
#cümledeki 1. paket kelimesinin indexsi 13
>>> metin.find('paket')
13
#cümledeki 2. paket kelimesinin indexsi 30
>>> metin.find('paket',14)
30
#cümledeki 3. paket kelimesinin indexsi 43
>>> metin.find('paket',31)
43
```

find() ve **index()** metotlarının 2. ve 3. parametrelerini anlamak için önce örnek metnimizin içinde geçen tüm 'paket' kelimelerinin indekslerini bulalım.

```
>>> metin.find('paket')
13
```

İlk **find()** metodu kullanımında parametresiz arama yapıldı ve ilk 'paket' kelimesinin indeksinin 13 olduğu bulundu.

```
>>> metin.find('paket',14)
30
```

İkinci **find()** kullanımda **find()** metodunun 2. parametresi kullanıldı. İlk 'paket' kelimesinin indeksinin 13 olduğu bulunduğu için aramayı 14. karakterden başlatırsak 2. 'paket' kelimesinin indeks değeri bulunacaktır. Bu yüzden hangi karakterden aramanın başlatılacağı **find()** metodunun 2. parametresinde 14 olarak belirtildi. 14. karakterden başlanarak yapılan arama sonucunda 2. 'paket' kelimesinin 30. indekste olduğu bulundu.

```
>>> metin.find('paket',31)
43
```

İkinci 'paket' kelimesinin indeks değerini bulduğumuz gibi 3. 'paket' kelimesinin indeks değeri 43 olarak bulunur.

```
>>> metin = 'alacaklar: 1 paket makarna, 1 paket tuz, 2 paket şeker'
>>> metin.find('paket',19,41)
30
```

Her zaman kısa karakter dizileri arasında çalışmak mümkün olmaya bilir. Eğer arama yapacağınız karakter dizisi çok büyükse yapacağınız aramayı dizinin en başında ve dizinin tümünü kapsayacak şekilde yapmak uzun zaman alacaktır. Üsteki örnekte görüldüğü gibi aranacak 'paket' kelimesi metin karakter dizisinin 19. ve 41. karakterleri arasında yapılacaktır.

```
>>> metin = 'alacaklar: 1 paket makarna, 1 paket tuz, 2 paket şeker'
>>> metin.rfind('paket')
43
>>> metin.rfind('paket', 11, 40)
30
```

rfind() ve **rindex()** metodlarının **find()** ve **index()** metodlarına göre farkı aramanın sağdan başlayarak yapılmasıdır.

```
>>> metin.rfind('paket')
43
```

rfind() ile yapılan aramada metnin en sağındaki üçüncü paket kelimesinin indeks değeri olan 30 bulunur.

```
>>> metin.rfind('paket', 11, 40)
30
```

Üsteki kod satırında aramanın metin dizisi içinde 11. ve 40. karakterleri arasında sağdan yapılacağı belirtilmiş. Üsteki arama sonucunda ikinci 'paket' kelimesinin indeks değeri 30 bulunur.

5.6.5. join() Metodu

```
>>> '@'.join(['tpotur', 'gmail.com']) >>> ''.join(['t', 'u', 'n', 'a'])
'tpotur@gmail.com' 'tuna'
```

join() metodu karakter dizilerini belirtilen karakterlerle birleştirerek tek bir karakter dizisi oluşturur.

5.6.6. split() Metodu

```
>>> 'Ahmet Tuna POTUR'.split() >>> 'tpotur@gmail.com'.split('@')
['Ahmet', 'Tuna', 'POTUR'] ['tpotur', 'gmail.com']

>>> isimler_listesi = 'Tuna,Can,Emel,Oya,Ahmet,Mehmet'.split(',')
>>> isimler_listesi
['Tuna', 'Can', 'Emel', 'Oya', 'Ahmet', 'Mehmet']

>>> isimler_listesi = 'Ezgi\nKerem\nAlp\nAda\nMert\nFırat'.split('\n')
>>> isimler_listesi
['Ezgi', 'Kerem', 'Alp', 'Ada', 'Mert', 'Fırat']
```

split() metodu **join()** metodunun tersini yapar. Bir karakter dizisi içindeki karakterleri belirtilen karaktere göre ayırır.

5.6.7. strip(),rstrip() ve lstrip() Metodu

```
# Sağ ve sol boşluklar silinir
>>> ' Python ' .strip()
'Python'
# Sağ boşluklar silinir
>>> ' Python ' .rstrip()
' Python'
# Sol boşluklar silinir
>>> ' Python ' .lstrip()
'Python '
```

strip kesilesinin Türkçe karşılığı soymaaktır. **strip()** metodu karakter dizisinin başında ve sonunda bulunan beyaz boşlukları karakter dizisinden temizler. **rstrip()** metodu karakter dizisinin sadece sağındaki beyaz boşlukları, **lstrip()** solundaki beyaz boşlukları temizler.

```
# Boşluk karakteri dışında \t ve \n karakterleride silinir
>>> metin = '\t \t \n Merhaba Dünya \t \n \n'
>>> metin
'\t \t \n Merhaba Dünya \t \n \n'
>>> print(metin)
```

Merhaba Dünya

```
>>> metin = metin.strip()
>>> metin
'Merhaba Dünya'
```

Beyaz boşluk demek karakter dizisinde kullanılan **boşluk**, **satır başı(\n)**, **paragraf(\t)** gibi karakterlerdir. Beyaz boşluklar harfler ve sayılar ile yazılan karakterlerin başında ve sonunda bulunur.

```
# Girilen bilginin boşlukları silinir
>>> adSoyad = input("Ad Soyad: ")
Ad Soyad:          Ada Lovelace
>>> adSoyad
'          Ada Lovelace          '
>>> adSoyad = adSoyad.strip()
>>> adSoyad
'Ada Lovelace'
```

Kullanıcı girişi **strip()** metodunun en çok kullanıldığı yerlerden biridir. Kullanıcılar yanlışlıkla girdikleri bilginin başına ve sonuna boşluk girebilirler. **strip()** metodu ile kullanıcın girdiği fazla boşluklar temizlenir.

5.6.8. upper() ve isupper() Metotları

<pre># Tüm Harfleri Büyük Harf Yapar >>> 'merhaba dünya'.upper() 'MERHABA DÜNYA'</pre>	<pre># Tüm Harfleri Büyük Harf Yapar >>> 'Merhaba Dünya'.upper() 'MERHABA DÜNYA'</pre>
---	---

upper() metodu karakter dizisinin tüm karakterlerini büyük harfe çevirir.

<pre># Tüm harfler büyük mü? >>> 'MERHABA DÜNYA'.isupper() True</pre>	<pre># Tüm harfler büyük mü? >>> 'Merhaba Dünya'.isupper() False</pre>
--	---

isupper() metodu karakter dizisinin tüm karakterlerinin büyük harf olup olmadığına bakar. **isupper()** metodu karakter dizisi elemanlarının tümü büyük harf ise **True** değeri döner.

5.6.9. lower() ve islower() Metotları

<pre># Tüm Harfleri Küçük Harf Yapar >>> 'MERHABA DÜNYA'.lower() 'merhaba dünya'</pre>	<pre># Tüm Harfleri Küçük Harf Yapar >>> 'Merhaba Dünya'.lower() 'merhaba dünya'</pre>
---	---

lower() metodu karakter dizisinin tüm karakterlerini küçük harfe çevirir.

<pre># Tüm harfler küçük mü? >>> 'merhaba dünya'.islower() True</pre>	<pre># Tüm harfler küçük mü? >>> 'Merhaba Dünya'.islower() False</pre>
--	---

islower() metodu karakter dizisinin tüm karakterlerinin küçük harf olup olmadığına bakar.

5.6.10. title() ve istitle() Metotları

```
# kelimelerinin ilk harfini büyük yapar
>>> 'merhaba dünya'.title()      >>> 'MERHABA DÜNYA'.title()
'Merhaba Dünya'                  'Merhaba Dünya'
>>> 'veri yapıları ve algoritmalar'.title() >>> 'VERİ YAPILARI VE ALGORİTMALAR'.title()
'Veri Yapıları Ve Algoritmalar'   'Veri Yapıları Ve Algoritmalar'
```

title() metodu karakter dizisini başlıklarda olduğu gibi her kelimenin baş harfi büyük olacak şekilde değiştirir.

```
#Başlık şeklinde mi?                #Başlık şeklinde mi?
>>> 'Merhaba Dünya'.istitle()        >>> 'merhaba dünya'.istitle()
True                                  False
```

istitle() metodu karakter dizisinin başlık tipinde olup olmadığını kontrol eder.

5.6.11. capitalize() Metodu

```
# Karakter dizisinin sadece baş harfini büyük harf yapar
>>> 'merhaba dünya'.capitalize()    >>> 'MERHABA DÜNYA'.capitalize()
'Merhaba dünya'                    'Merhaba dünya'
```

capitalize() metodu karakter dizisinin sadece ilk harfini büyük harf yapar.

5.6.12. swapcase() Metodu

```
>>> 'merhaba dünya'.swapcase()      >>> 'MERHABA DÜNYA'.swapcase()
'MERHABA DÜNYA'                    'merhaba dünya'
>>> 'Merhaba Dünya'.swapcase()      >>> 'MeRhAbA DüNyA'.swapcase()
'mERHABA dÜNYA'                    'mErHaBa dÜnYa'
```

swapcase() metodu karakter dizisindeki büyük harfleri küçük harfleri ise büyük harfleri yapar.

5.6.13. isspace() Metodu

```
>>> '\t \n'                          >>> '\t \n Python \t \n'.isspace()
True                                  False
```

isspace() Metodu karakter dizisinin beyaz olup olmadığını bakar.

5.6.14. center(), rjust() ve ljust() Metotları

```
# Karakteri ortalar
>>> 'Python'.center(20)              >>> 'Python'.center(20, '*')
'      Python      '                '*****Python*****'
# Karakteri sağ yasla
>>> 'Python'.rjust(20)               >>> 'Python'.rjust(20, '*')
'      Python'                      '*****Python'
# Karakteri sola yasla
>>> 'Python'.ljust(20)               >>> 'Python'.ljust(20, '*')
'Python'                            'Python*****'
```

center(), rjust() ve ljust() metotlar iki parametre alır. Bu metotlar tek parametre ile kullanıldığında karakter dizisini ilk parametrede belirtilen boşluk değeri içinde ortalanmış, sağa yaslanmış veya sola yaslanmış olarak oluştururlar. İkinci parametreye karakter girilir. İki parametrelili kullanımda karakter dizisi ikinci parametrede girilen karakterin ortasına, sağına veya soluna yerleşir. **center()** metodu karakter dizisini ortalamak, **rjust()** sağ yaslamak ve **ljust()** sola yaslamak için kullanılır.

5.6.15. count() Metodu

```
#Karakter dizisi içinde aranan karakterlerin sayısı
>>> 'alafrangalaştıramayacaklardansalar'.count('a')
13
>>> 'alafrangalaştıramayacaklardansalar'.count('ala')
3
>>> 'alafrangalaştıramayacaklardansalar'.count('a',20)
5
>>> 'alafrangalaştıramayacaklardansalar'.count('a',5,30)
9
```

count() metodunun çalışma mantığı **index()** ve **find()** metodlarının çalışma mantığına benzer. **count()** metodunun ilk parametresine karakter dizisi girilir. **count()** metodu parametre olarak aldığı karakter dizisinin metin içinde kaç defa kullanıldığını hesaplar. **count()** metodunun ikinci parametresi arama yapılacak metnin başlangıç indeksini üçüncü parametresi bitiş indeksini belirtir.

5.6.16. isnumeric() Metodu

```
>>> '2018'.isnumeric()          >>> '2018 - 2017 Öğretim Yılı'.isnumeric()
True                             False
# karakter olarak tanımlanan float sayılar numeric değildir
>>> '3.14'.isnumeric()
False
```

isnumeric() metodu karakter dizisinin tüm karakterlerinin sayı olup olmadığına bakar. Eğer karakter dizisinin içinde 0 – 9 rakamları arasındaki sayılardan farklı olan bir karakter varsa **isnumeric()** metodu False sonuç verir. '3.14' gibi float sayıların sonucuda arada nokta karakteri bulunduğu için False çıkar.

5.6.17. endswith() ve startswith() Metotları

```
>>> metin = 'alacaklar: 1 paket makarna, 1 paket tuz, 2 paket şeker'
>>> metin.startswith('alacaklar')    >>> metin.startswith('paket')
True                                 False
>>> metin.endswith('şeker')          >>> metin.endswith('paket')
True                                 False
```

startswith() metodu aranan karakterlerin karakter dizisinin başında olup olmadığına bakar. **endswith()** metodu aranan karakterlerin karakter dizisinin sonunda olup olmadığına bakar.

```
>>> metin = 'alacaklar: 1 paket makarna, 1 paket tuz, 2 paket şeker'
>>> metin.startswith('paket',13)    >>> metin.endswith('şeker',13)
True                                 True
```

startswith() ve **endswith()** metodlarının ikinci parametreleri aramanın karakter dizisinin hangi indeksteki elemanından başlanarak yapılacağını belirtir.

```
>>> metin = 'alacaklar: 1 paket makarna, 1 paket tuz, 2 paket şeker'
>>> metin.startswith('tuz',36,48)    >>> metin.endswith('paket',36,48)
True                                 True
```

Bu metodların ikinci ve üçüncü parametreleri karakter dizisi içinde aramanın hangi indeksler arasında yapılacağını belirtir.

5.7. Karakter Dizilerine Eleman Ekleme (Karakter Birleşimi)

```
# ismin sonunda bir boşluk var
>>> a = "Ahmet "
>>> b = "Tuna "
>>> c = "POTUR"
>>> isim = a + b + c
>>> isim
'Ahmet Tuna POTUR'
```

```
>>> 'www'+'.'+'google'+'.'+'com'
'www.google.com'
```

Karakter dizileri birbirleriyle toplanabilir. Karakter dizileri toplandığında iki karakter dizisinin birleşiminden oluşan yeni bir karakter dizisi oluşur.

```
>>> başlık = "Ahmet " 'Tuna' " POTUR" # Çarpıştırarak birleştirme
>>> başlık
'Ahmet Tuna POTUR'
```

Tırnak ile ayrılan karakterler toplama sembolü kullanmadan bir birlerine çarpıştırılarak(bitiştirilerek) birleştirilebilir.

```
>>> metin = "Merhaba"
>>> metin += " Dünya" # karakterin başında bir boşluk var
>>> metin
'Merhaba Dünya'
```

Karakter dizileri üzerinde değişiklik yapılamaz. Fakat `metin += " Dünya"` işlemiyle karakter dizileri toplanarak karakter dizisinin sonuna eleman eklenebilir. Üstteki iki örneğe dikkat edilirse, karakterler birleştirildiği zaman anlam bütünlüğü sağlanması için karakter dizilerinin başında veya sonunda boşluk karakteri bırakmak gerekir.

```
>>> isim = 'T'
>>> isim += 'u'
>>> isim += 'n'
>>> isim += 'a'
>>> isim
'Tuna'
```

Karakter dizileri harf harf toplanarak birleştirilebilir.

```
# rakamlar birleşiyor
>>> rakamlar = '0'+'1'+'2'+'3'
>>> rakamlar
'0123'
```

```
>>> a = '35'
>>> a += '43'
>>> a
'3543'
```

Karakter toplamayı sayısal toplamayı karıştırmamak gerekir. Karakter dizileri toplandığında sonuç iki karakter dizisinin birleşimidir.

```
>>> 45 + '53'
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    45 + '53'
TypeError: unsupported operand type(s) for
+: 'int' and 'str'
```

```
# tip dönüşümü ile hata düzeltilir
>>> str(45)+'53'
'4553'
>>> 45+int('53')
98
```

5.8. Karakter Dizilerini Çarparak Tekrar Ettirme

```
>>> isim = "Tuna "
>>> isim*3
'Tuna Tuna Tuna '
>>> 'w'*3
'www'
```

```
>>> rakam = '7'
>>> rakam*7
'7777777'
>>> 'müdür '*3
'müdür müdür müdür '
```

* operatörü ile karakter dizileri çarpılarak tekrar ettirilebilir.

5.9. Dilimleme

```
>>> # Dilimleme
>>> metin = "Ada ilk bilgisayar programcısı olarak kabul edilir"
>>> metin[:]
'Ada ilk bilgisayar programcısı olarak kabul edilir'
>>> metin[::-1]
'rilide lubak karalo ısıcmargorp rayasiglib kli adA'
>>> metin[3:40:2]
' l igsyrypormııoaa a'
>>> metin[2::-1]
'adA'
>>> metin[38:]
'kabul edilir'
>>> metin[:3]
'A kiıy oaı akaldı'
```

Karakter dizileri anı demetler ve listeler gibi dilimlenir.

```
>>> adres1 = "www.teknoseyir.com"
>>> adres2 = "www.donanimhaber.com"
>>> adres3 = "www.python.org"
>>> adres4 = "www.zetcode.com"
>>> for adres in adres1, adres2, adres3, adres4:
    print("Adres: ", adres[4:-4])
```

```
Adres: teknoseyir
Adres: donanimhaber
Adres: python
Adres: zetcode
```

Döngüler konusu sonradan detaylı bir şekilde işlenecek fakat yeri gelmişken üstteki örneği yapmakta yarar var. Örnekte görüldüğü gibi döngüye giren web site adresi karakter dizilerinin `adres[4:-4]` ifadesiyle dilimlenerek ilk dört ve son dört karakterleri hariç yazdırılması gösterilmiş.

5.10. Karakter Dizileri Üzerinde Doğrudan Çalışmak

```
>>> "Merhaba Dünya"[4]
'a'
>>> "Merhaba Dünya"[8]
'D'
>>> "Merhaba Dünya"[8:]
'Dünya'
>>> "Merhaba Dünya"[1:12:2]
'ehb üy'
>>> "Merhaba Dünya".count('a')
3
>>> "Merhaba Dünya".index('ü')
9
```

Karakter dizileri listeler ve demetler gibi nesnel bir veri tipidir. Karakter dizi değerlerini aynı zamanda nesne olduğu için doğrudan kullanılabilir.

```
>>> adı = "Can"
>>> ders = "Fizik"
>>> notOrt = 75
>>> print("{} {} dersinden {} almıştır".format(adı, ders, notOrt))
Can Fizik dersinden 75 almıştır
```

Karakter dizileri üzerinde doğrudan çalışmak en çok karakterleri formatlarken kullanılır. Değişkenlerin taşıdıkları değerler karakter dizisi içinde `format()` metodu kullanılarak yazdırılır. Karakter dizisinin içinde kullanılan `{}` parantez `format()` metodu içinde bulunan değişkenin değerini sırasıyla alır.

5.11. in Operatörü

```
>>> 'Tuna' in 'Ahmet Tuna POTUR'
True
>>> 'Tuna' in 'Merhaba Dünya'
False
```

`in` operatörü kullanılarak bir karakter dizisi içinde başka bir karakter değerinin varlığı kontrol edebilir.

5.12. Kaçış Dizileri (Özel Karakterler)

5.12.1. Satır Başı (\n)

```
>>> ada = "Ada Lovelace sahip olduğu maddi kaynak sayesinde \nBabbage'in en büyük destekçilerinden biriydi. \nİlk algoritmayı tasarladığı için Ada \nİlk bilgisayar programcısı olarak kabul edilir."
```

```
>>> print(ada)
```

```
Ada Lovelace sahip olduğu maddi kaynak sayesinde
Babbage'in en büyük destekçilerinden biriydi.
İlk algoritmayı tasarladığı için Ada
İlk bilgisayar programcısı olarak kabul edilir.
```

Karakter dizisi içinde satır sonunu belirtmek için `\n` satır başı karakteri kullanılır. Python karakter dizisini yorumlamaya başladığı zaman `\n` karakterini görürse satırı sonlandırır ve devam eden karakterleri yeni satırdan yazmaya başlar. `\` karakteri `n` harfinin özel bir anlam kazanmasını sağlar.

```
>>> print("1.Satır \n2.Satır \n3.Satır \n4.Satır")
```

```
1.Satır
2.Satır
3.Satır
4.Satır
```

`\n` karakterinin kullanımını anlatmak için üstteki basitleştirilmiş örneği inceleyelim. `print()` fonksiyonu içine girilen karakter dışı `\n` karakteriyle 2,3 ve 4. Satır başları verilmiş.

5.12.2. Satır Başı Tab (\t)

```
>>> adı = "Can"
>>> boyu = "1.85"
>>> yaş = 15
>>> print("Adı\t:",adı,"\nBoyu\t:",boyu,"\nYaşı\t:",yaş)
Adı   : Can
Boyu  : 1.85
Yaşı  : 15
```

`\t` karakteri satır başı vermeye yarar. Kodun içinde metinlerin daha düzenli yerleşmesi için `\t` karakteri kullanılır.

```
>>> print("\tAra","\tAra","\tAra","\tAra")
Ara    Ara    Ara    Ara
```

Karakterler arasına bir tab karakteri kadar boşluk oluşturulması için üstteki örnek daha anlaşılır.

```
>>> print("Ara","Ara","Ara","Ara",sep="\t")
Ara    Ara    Ara    Ara
```

`sep` parametresi yardımıyla karakter arasına tab karakteri girilebilir.

```
>>> print(*"192837465",sep="\t")
1     9     2     8     3     7     4     6     5
```

`*` ve `sep="\t"` karakterleri kullanılarak bir karakter dizisi içindeki tüm karakterlerin arasına tab karakteri girilebilir.

5.12.3. Ters Bölü (\)

```
>>> print("İlk algoritmayı tasarladığı için\ Ada ilk bilgisayar programcısı olarak kabul edilir.")
İlk algoritmayı tasarladığı için Ada ilk bilgisayar programcısı olarak
kabul edilir.
```

Kodların düzgün görünmesi için yazdığınız her satırın en fazla **79 karakter** olması kabul görmüştür. Eğer yazdığınız bir satır **79 karakteri** aşıyorsa, aşan kısmı üsteki örnekte olduğu gibi **ters bölü(\)** operatörüyle alt satıra alınır. Örnekten anlaşılacağı gibi **ters bölü(\)** kod içinde bir değişiklik yapmaz sadece program satırlarının çok uzun görünmesini engeller. `print()` fonksiyonuna girilen değer **ters bölü(\)** ile satırlara ayrılrsa da `print()` fonksiyonu çıktısı tek satırdır.

```
>>> print('Ada'nın algoritması')      # ' karakteri iki kere kullanılmış

SyntaxError: invalid syntax
>>> # \ kullanılarak düzeltme          >>> # " kullanılarak düzeltme
>>> print('Ada\'nın algoritması')      >>> print("Ada'nın algoritması")
Ada'nın algoritması                  Ada'nın algoritması
```

Tek tırnak (') ile başlayan karakter dizilerinde ilk tırnaktan sonra iki kere daha tırnak kullanılırsa yazım hatası oluşur. Bu yazım hatasını düzeltmenin iki yolu vardır. Birinci yol ters bölü ile tırnak **\'** karakterini kullanmaktır. Ters bölü tırnak karakterinin karakter dizisini kapatma görevini engeller. İkinci yol karakter dizisini çift tırnak içinde tanımlamaktır.

```
>>> print("C:\nisan\harcama.txt")
C:
isan\harcama.txt
# \\ kullanılarak düzeltme          # r kullanılarak düzeltme
>>> print("C:\\nisan\harcama.txt")    >>> print(r"C:\nisan\harcama.txt")
C:\nisan\harcama.txt                C:\nisan\harcama.txt
```

Yazdığınız metinlerde **\n** karakteri kullanıldıysa Python bunu satır başı olarak yorumlar. Bu durumu düzeltmenin iki yolu vardır. Ters bölü karakteri iki kere kullanılırsa **\n** karakterinin satır başı özelliği devre dışı kalır. Karakter dizisinin başına **r** karakteri eklenirse dizi içindeki özel karakterler devre dışı kalır.

```
>>> print("C:\nisan\toplam.txt")
C:
isan      oplam.txt
# \\ kullanılarak düzeltme          # r kullanılarak düzeltme
>>> print("C:\\nisan\\toplam.txt")    >>> print(r"C:\nisan\toplam.txt")
C:\nisan\toplam.txt                C:\nisan\toplam.txt
```

\n karakterinde olduğu gibi karakter dizisinin içinde **\t** karakteri bulunuyorsa Python bunu satır başı olarak yorumlar. Bu durum **\n** örneğinde olduğu gibi ya **** karakteriyle ya da **r** operatörüyle düzeltilir.

5.13. print() Fonksiyonu

print() fonksiyonunu daha önce basit bir şekilde görmüştük. Bu bölümde **print()** fonksiyonunun görmediğimiz özelliklerine değinilecek. **print()** fonksiyonu parantez içine girilen değeri veya değerleri ekrana yazdırılır.

```
# bir parametrelili print fonksiyonu
>>> print("Adınız :")
Adınız :
>>> yaş = 16
>>> boy = 1.85
>>> adı = 'Can'
# virgülle ayrılarak girilen birden çok parametrelili print fonksiyonu
>>> print("Adınız :",adı)
Adınız : Can
# parametreler yazdırıldıklarında aralarında 1 boşluk olur
>>> print("Adınız:",adı,"Yaşınız:",yaş,"Boyunuz:",boy)
Adınız: Can Yaşınız: 16 Boyunuz: 1.85
```

`print()` fonksiyonuna istediğimiz sayıda parametre girişi yapabiliriz.

```
>>> print("Adınız :")
```

İfadesinde “Adınız” print() fonksiyonu parametresidir. Bu ifadede print() fonksiyonuna bir parametre girilmiş.

```
>>> print("Adınız:",adı,"Yaşınız:",yaş,"Boyunuz:",boy)
```

İfadesinde print() fonksiyonuna 6 adet parametre girişi yapılmış. Fonksiyonlar birden çok parametre ile kullanılıyorsa fonksiyona girilen parametreler virgül ile ayrılır. Birden çok parametre ile kullanılan print() fonksiyonun çıktısına dikkat ederseniz parametre değerlerinin bir boşlukla bir birinden ayrıldığı görülür.

5.13.1. print() Fonksiyonunun Parametreleri

print() fonksiyonunda şimdiye kadar parametre olarak sadece ekrana yazdırmak istediğimiz verileri kullandık. print() fonksiyonunda değerleri varsayılan olarak atanmış sep ve end parametreleri bulunmaktadır. print() fonksiyonunun sep ve end parametreleri ekrana yazdıracağı metnin düzenlememizi sağlar. sep parametresi için bir boşluk “ ”, end parametresi için bir satır “\n” değeri önceden tanımlandığı için print() fonksiyonun kullanımında bu parametreler için argüman girişi yapılmasına gerek yoktur.

5.13.1.1. sep Parametresi

```
# parametreler arası bir boşluk var
>>> print("http://","www.","google",".com")
http:// www. google .com
# parametreler arası boşluk kaldırıldı
>>> print("http://","www.","google",".com",sep="")
http://www.google.com
```

print() fonksiyonu kendisine verilen parametreleri ekrana yazdırırken ön tanımlı olarak parametreler arasında bir boşluk yerleştirir.

```
sep = " " # sep parametresine ön tanımlı olarak boş karakter atanmıştır
```

Üste görüldüğü gibi sep parametresine ön tanımlı olarak boşluk “ ” karakteri atanmıştır.

```
>>> print("http://","www.","google",".com",sep="")
```

Ekrana yazdırılan parametreler arasında boşluk olmaması için sep parametresine sep="" ifadesiyle boşluk olmayan karakter girişi yapılmalıdır. sep ifadesi İngilizcedeki separator(ayrıcı, ayraç) kelimesinin kısaltmasıdır.

```
>>> print("Prof","Ahmet Tezcen") # print()sep parametresiz kullanılır
Prof Ahmet Tezcen
>>> print("Prof","Ahmet Tezcen",sep=".") # sep parametresine . giriliyor
Prof.Ahmet Tezcen
```

Üsteki örnekte sep parametresine değer girmezseniz argümanlar aralarında bir boşluk bırakılarak yazılır. Eğer sep="." atamasıyla print() fonksiyonu argümanlarının arasına nokta“.” karakteri girilir.

```
>>> print(1,2,3,4,5,6,7,sep=" ",")
1, 2, 3, 4, 5, 6, 7
```

Üsteki örnekte argümanlar arasında virgül ve boşluk “,” karakteri yerleştirildiğinden sayılar ekrana virgül ile ayrılarak yazdırılmıştır.

```
>>> print("06","03","1981",sep = "/"")
06/03/1981
```

Kod içindeki gün ay ve yıl değerleri sep parametresi ile birleştirilerek ekrana yazdırılabilir.

5.13.1.2. end Parametresi

```
>>> isimler = ['Tuna', 'Can', 'Emel', 'Oya', 'Ahmet', 'Mehmet']
>>> for isim in isimler:      # isimler listesi için döngü kuruluyor
    print(isim)              # isimler elemanları tek tek yazdırılıyor
# print() fonksiyonu her elemandan sonra satır başı yapar
Tuna
Can
Emel
Oya
Ahmet
Mehmet
```

print() fonksiyonu ön tanımlı olarak, parametrelerin sonuna satır **başı karakteri** “\n” karakteri ekler. print() fonksiyonu önce aldığı argümanları ekrana yazdırır sonra bir satır boşluk vererek alt satıra geçer. Üsteki örnekte print() fonksiyonun bu özelliğini anlatabilmek için sonra göreceğimiz döngüler konusundan yararlandık. **for döngüsü** ile print() fonksiyonu her çağrıldığında liste elemanlarını sırasıyla ekrana yazdırıyor. print() fonksiyonu olduğu gibi kullanılırsa her değer ekrana yazdırıldıktan sonra bir satır boşluk verir ve elemanları alt alta yazdırır.

```
end = "\n" # end param. ön tanımlı olarak satır başı karakteri atanmıştır
```

end parametresi print() fonksiyonuna girilen parametrelerin sonuna neyin geleceğini belirler. **end** parametresine ön tanımlı olarak satır başı karakteri atanmıştır.

```
>>> isimler = ['Tuna', 'Can', 'Emel', 'Oya', 'Ahmet', 'Mehmet']
>>> for isim in isimler:      # isimler listesi için döngü kuruluyor
    print(isim,end=" ",)      # isimler elemanları tek tek yazdırılıyor
# liste elemanları virgülle ayrılarak sırasıyla yazdırılmış
Tuna, Can, Emel, Oya, Ahmet, Mehmet,
```

print() fonksiyonuna girilen argümanlar ekrana yazdırıldıktan sonra satır başı verilmesi istenmiyorsa **end** parametresinin değeri değiştirilebilir.

```
print(isim,end=" ",) # end parametresine virgül ve boşluk karakteri atanıyor
```

end parametresine virgül ve boşluk karakteri eklenirse “,” liste elemanları virgülle ayrılarak ve yan yana yazdırılır.

5.13.1.3. Yıldız Operatörü(*)

```
>>> print("Lüleburgaz")
Lüleburgaz
# * ile karakter verisi boşluk karakteriyle ayrılıyor
>>> print(*"Lüleburgaz")
L ü l e b u r g a z
```

print() fonksiyonuna parametre olarak girilen bir karakter dizisinin başına eklenen **yıldız(*)** operatörü, karakter dizisini tek tek öğelerine ayırır. **sep** parametresi kullanılmadan print() fonksiyonunda karakter dizisinin başına **yıldız(*)** parametresi eklenirse karakter arası bir boşluk bırakılarak yazdırılır.

```
# * ve sep ile karakterler arasına istenen karakter girilir
>>> print(*"Lüleburgaz",sep="-")      >>> print(*"Lüleburgaz",sep="*")
L-ü-l-e-b-u-r-g-a-z                  L*ü*l*e*b*u*r*g*a*z
>>> print(*"Lüleburgaz",sep=", ")      >>> print(*"Lüleburgaz",sep="-*-")
L, ü, l, e, b, u, r, g, a, z          L-*ü-*l-*e-*b-*u-*r-*g-*-a-*-z
```

print() fonksiyonunda **yıldız(*)** ve **sep** parametresi ile karakter arası istenen karakterler yazdırılır.

```
>>> print(*196375645, sep = ", ")
Traceback (most recent call last):
  File "<pyshell#9>", line 1, in <module>
    print(*196375645, sep = ", ")
TypeError: print() argument after * must be an iterable, not int
```

* operatörü sayılar için kullanılamaz. Python sayısal veriler ile * operatörü kullanılırsa hata verir.

```
>>> print(*str(196375645), sep = ", ")
1, 9, 6, 3, 7, 5, 6, 4, 5
```

str() fonksiyonuyla sayısal veriler karakter dizilerine dönüştürülürse * operatörü kullanılabilir.

```
# * ve sep parametresi liste ile kullanılabilir
>>> print(*[1, 3.14, 'T', 'Ahmet'], sep='/')
1/3.14/T/Ahmet
# * ve sep parametresi demet ile kullanılabilir
>>> print(*('B', 'u', 'r', 'g', 'a', 'z'), sep='&')
B&u&r&g&a&z
```

* operatörü karakter dizilerinin dışında liste ve demetler içinde kullanılabilir.

```
>>> ada = "Ada Lovelace ilk bilgisayar programcısı olarak kabul edilir."
>>> print(ada, "\n", "-"*len(ada), sep="")
Ada Lovelace ilk bilgisayar programcısı olarak kabul edilir.
-----
```

Ekrana yazdırmak istediğiniz metnin altına karakter sayısı kadar **tire(-)** karakteri eklemek istiyorsanız üsteki kodu kullanabilirsiniz. “\n” karakteri ada değişkeni içinde bulunan karakter dizisi yazıldıktan sonra paragraf başı verir. “-“*len(ada) işlemi ada değişkeni içindeki karakter sayısı kadar **tire(-)** karakterini ekrana basar. **sep** parametresinde varsayılan olarak boşluk karakteri bulunur. **sep=""** işlemiyle karakterler arası boşluklar alınır.

```
# \n kullanılmayınca alt satıra geçilmiyor
>>> print(ada, "\n", "-"*len(ada), sep="")
Ada Lovelace ilk bilgisayar programcısı olarak kabul edilir.-----
-----
```

Karşınıza böyle farklı bir kod çıkarsa kod üzerinde denemeler yaparak kodu daha kolay anlayabilirsiniz. Üsteki örnekte \n karakteri silinmiş. \n karakteri olmazsa **tire(-)** karakterleri alt satırdan başlamayacaktır.

```
# sep="" kullanılmayınca bir satır içten başlıyor
>>> print(ada, "\n", "-"*len(ada))
Ada Lovelace ilk bilgisayar programcısı olarak kabul edilir.
-----
```

sep parametresinin varsayılan değeri boşluk karakteridir. sep="" işlemi yapılmazsa, **tire(-)** karakterleri bir satır içten başlayacaktır.

```
>>> print(*"123", sep="\n")
1
2
3
```

* ve sep="\n" karakterleri kullanılarak bir karakter dizisi içindeki tüm karakterler alt alta yazılır.