İçindekiler

- Veri Manipülasyonu 101
  - Giriş
    - Neden NumPy?

  - Oluşturma ve Biçimlendirme İşlemleri
    - NumPy Array'i Oluşturmak
    - NumPy Array Özellikleri
    - Yeniden Şekillendirme (Reshaping)
    - Birleştirme (Concatenation)
    - Array Ayırma (Splitting)
    - Sıralama (Sorting)

  - Eleman İşlemleri
    - Index ile Elemanlara Erişmek
    - Array Alt Küme İşlemleri (Slicing)
    - Alt Küme Üzerinde İşlem Yapmak
    - Fancy Index ile Elemanlara Erişmek
    - Koşullu Eleman İşlemleri

  - Matematiksel İşlemler
    - Matematiksel İşlemler'e Giriş
    - NumPy ile İki Bilinmeyenli Denklem Çözümü

- Veri Manipülasyonu 201
  - Pandas Serileri
    - Pandas Giriş
    - Pandas Serisi Oluşturma
    - Eleman İşlemleri

- Pandas DataFrame
  - Pandas DataFrame Oluşturma
  - Eleman İşlemleri
  - Gözlem ve Değişken Seçimi: loc & iloc
  - Koşullu Eleman İşlemleri
  - Birleştirme (Join) İşlemleri
  - İleri Birleştirme İşlemleri

- Gruplama ve Toplulaştırma İşlemleri
  - Gruplama ve Toplulaştırma (Grouping & Aggregation)
  - Gruplama İşlemleri
  - Aggregate
  - Filter
  - Transform
  - Apply
  - Pivot Tablolar
  - Dış Kaynaklı Veri Okuma

In [ ]:

In [1]:
```
import numpy as np
import pandas as pd
import seaborn as sns
```

Veri Manipülasyonu 101

Giriş

In [1]:
```
print("a")
```

```
a
```

Merhaba

Herhangi başka şeyler de yazılabilir.

italik

kalın

In [2]:
```
Merhaba
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-2-52bbe095ee49> in <module>
----> 1 Merhaba

NameError: name 'Merhaba' is not defined
```

In [4]:
```
#Merhaba
```

In [3]:
```
"a"
```

Out[3]:
```
'a'
```

Neden NumPy?

İki diziyi çarpan programı ele alacağız.

Klasik Programlama'da

In [5]:
```
a = [1,2,3,4] #Listede bu değişken için arka planda 4 bilgi tutulur.
b = [2,3,4,5]
```

In [7]:
```
ab = []

for i in range(0, len(a)):
    ab.append(a[i]*b[i])

ab
```

Out[7]:
```
[2, 6, 12, 20]
```

NumPy'da

In [49]:
```
a = np.array([1,2,3,4]) #NumPy'da bu değişken için arka planda sadece 1 bilgi tutulur.
b = np.array([2,3,4,5])
```

In [50]:
```
a * b
```

Out[50]:
```
array([ 2,  6, 12, 20])
```

Görüldüğü üzere NumPy'da daha az kod ile bu işlemi gerçekleştirdik. Hem de bellekte daha az yer tutulmuş oldu.

Oluşturma ve Biçimlendirme İşlemleri

NumPy Array'i Oluşturmak

In [2]:
```
np.array([1,2,3,4,5])
```

Out[2]:
```
array([1, 2, 3, 4, 5])
```

In [3]:
```
a = np.array([1,2,3,4,5])
```

In [4]:
```
type(a)
```

Out[4]:
```
numpy.ndarray
```

In [5]:
```
np.array([3.14,4,2,13])
```

Out[5]:
```
array([ 3.14,  4.  ,  2.  , 13.  ])
```

NumPy listelerden farklı olarak verileri sabit tiple tuttuğu için hepsini float'a çevirdi. Bunu dilersek ayarlayadabiliriz.

In [7]:
```
np.array([3.14,4,2,13], dtype="int")
```

Out[7]:
```
array([ 3,  4,  2, 13])
```

Sıfırdan Array Oluşturma

In [9]:
```
np.zeros(10, dtype=int)
```

Out[9]:
```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

In [11]:

```
np.ones((3, 5), dtype=int)
```

Out[11]:
```
array([[1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1]])
```

In [12]:
```
np.full((3, 5), 3)

#Sadece 3'lerden oluşan 3x5'lik bir dizi oluşturduk.
```

Out[12]:
```
array([[3, 3, 3, 3, 3],
       [3, 3, 3, 3, 3],
       [3, 3, 3, 3, 3]])
```

In [14]:
```
np.arange(0, 31, 3)
```

Out[14]:
```
array([ 0,  3,  6,  9, 12, 15, 18, 21, 24, 27, 30])
```

In [15]:
```
np.linspace(0, 1, 10)
```

Out[15]:
```
array([0.        , 0.11111111, 0.22222222, 0.33333333, 0.44444444,
       0.55555556, 0.66666667, 0.77777778, 0.88888889, 1.        ])
```

In [16]:
```
np.random.normal(10, 4, (3,4))

#Normal dağılıma uygun matris oluşturabiliriz.
```

Out[16]:
```
array([[ 7.91665902, -0.59889243,  8.8391989 , 11.17366158],
       [14.13958833,  7.4649196 , 13.96844058,  3.87859815],
       [14.70408714,  4.95430674,  7.7710671 ,  9.21321217]])
```

In [17]:
```
np.random.randint(0, 10, (3,3))

#Rastgele int türünde matris oluşturabiliriz.
```

Out[17]:
```
array([[1, 1, 1],
       [2, 7, 6],
       [7, 7, 6]])
```

NumPy Array Özellikleri

- **ndim**: boyut sayısı

- **shape**: boyut bilgisi
- **size**: toplam eleman sayısı
- **dtype**: array veri tipi

In [2]:
```
np.random.randint(10, size = 10)
```

Out[2]:
```
array([0, 7, 0, 3, 0, 8, 9, 9, 7, 7])
```

In [3]:
```
a = np.random.randint(10, size = 10)
```

In [15]:
```
a
```

Out[15]:
```
array([1, 2, 4, 1, 3, 3, 8, 7, 5, 2])
```

In [4]:
```
a.ndim
```

Out[4]:
```
1
```

In [5]:
```
a.shape
```

Out[5]:
```
(10,)
```

In [6]:
```
a.size
```

Out[6]:
```
10
```

In [8]:
```
a.dtype
```

Out[8]:
```
dtype('int32')
```

In [9]:
```
b = np.random.randint(10, size = (3,5))
```

In [10]:
```
b
```

Out[10]:
```
array([[3, 7, 0, 1, 8],
```

```
       [7, 5, 0, 5, 9],
       [2, 7, 4, 4, 1]])
```

In [11]:
```
b.ndim
```

Out[11]:
```
2
```

In [12]:
```
b.shape
```

Out[12]:
```
(3, 5)
```

In [13]:
```
b.size
```

Out[13]:
```
15
```

In [14]:
```
b.dtype
```

Out[14]:
```
dtype('int32')
```

Yeniden Şekillendirme (Reshaping)

Neden İhtiyaç Duyulur?

Örneğin çalışmalarda fonksiyonlarımızın ya da döngülerimizin üretmiş olduğu çıktılar tek bir boyutta, tek bir array formunda gerçekleşebiliyor. Bunları bazen tek boyuttan iki boyuta, bazen de iki boyuttan tek boyuta indirmek ihtiyacı gerçekleşebiliyor. İşte bu sebeple bu yapısal dönüşümleri iyi kavramak gerekiyor. Buna benzer ihtiyaçlarla da **reshape** fonksiyonu ile başa çıkmış oluyoruz.

In [17]:
```
np.arange(1, 10)
```

Out[17]:
```
array([1, 2, 3, 4, 5, 6, 7, 8, 9])
```

In [19]:
```
np.arange(1, 10).reshape((3,3))
```

Out[19]:
```
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

In [20]:
```
a = np.arange(1, 10)
```

In [21]:
```
a
```

Out[21]:
```
array([1, 2, 3, 4, 5, 6, 7, 8, 9])
```

In [22]:
```
a.ndim
```

Out[22]:
```
1
```

Bazen tek boyutlu bir vektörü iki boyutlu matrisi çevirmek isteriz fakat tek boyutlu olan bilgisi de olduğu şekliyle kalsın. Yani tek boyut bilgilerini taşıyacak şekilde dönüştürme yapmak da isteyebiliriz. Bunu yapmak için:

In [25]:
```
b = a.reshape((1,9))
```

In [27]:
```
b
```

Out[27]:
```
array([[1, 2, 3, 4, 5, 6, 7, 8, 9]])
```

In [26]:
```
b.ndim
```

Out[26]:
```
2
```

Görüldüğü üzere görünüşte vektörle aynı özellikte fakat boyutlarıyla oynadığımız için boyutu iki oldu. Çıktıda da iki köşeli parantez olarak ekrana yazıldığını fark ediyoruz.

Birleştirme (Concatenation)

Tek Boyut İçin

In [29]:
```
x = np.array([1,2,3])
y = np.array([4,5,6])
```

In [30]:
```
np.concatenate([x, y])
```

Out[30]:
```
array([1, 2, 3, 4, 5, 6])
```

In [31]:
```
z = np.array([7,8,9])
```

```
#Birleştirilmiş diziye sonradan bir dizi de ekleyebiliriz.
```

In [32]:
```
np.concatenate([x, y, z])
```

Out[32]:
```
array([1, 2, 3, 4, 5, 6, 7, 8, 9])
```

İki Boyut İçin

In [33]:
```
a = np.array([[1,2,3],
              [4,5,6]])

#İki boyutlu dizi oluşturmak için çift köşeli parantez kullanılmalıdır.
```

In [34]:
```
np.concatenate([a, a])
```

Out[34]:
```
array([[1, 2, 3],
       [4, 5, 6],
       [1, 2, 3],
       [4, 5, 6]])
```

Sütun bazlı birleştirme yapmak istersek **axis** argümanı kullanılmalıdır. Varsayılan olarak 0'dır ve satır bazlı birleştirme yapar. Sütun bazlı yapmak için argümanı 1 yapmalıyız.

In [35]:
```
np.concatenate([a, a], axis = 1)
```

Out[35]:
```
array([[1, 2, 3, 1, 2, 3],
       [4, 5, 6, 4, 5, 6]])
```

Array Ayırma (Splitting)

Tek Boyut İçin

In [37]:
```
x = np.array([1,2,3,99,99,3,2,1])
```

In [38]:
```
np.split(x, [3, 5])
```

Out[38]:
```
[array([1, 2, 3]), array([99, 99]), array([3, 2, 1])]
```

Burada önce ayırmak istediğimiz diziyi, sonrasında hangi **indexlere** kadar ayırmak istiyorsak *-yazılan indexler dahil değil-* onları köşeli parantez içerisine yazmamız gerekiyor. Burada 3. index ve 5. index'e kadar ayırmak istedik. Son parçayı da otomatik olarak kendisi ayırdı.

Bu ayırdığımız dizileri kullanmak istersek kaç tane çıktı olduğunu belirledikten sonra o kadar değişkene **tek seferde** eşitlememiz gerekiyor. Bu, Python'ın kolaylıklarından bir tanesidir.

In [39]:
```
a,b,c = np.split(x, [3, 5])
```

In [40]:
```
a
```

Out[40]:
```
array([1, 2, 3])
```

In [41]:
```
b
```

Out[41]:
```
array([99, 99])
```

In [42]:
```
c
```

Out[42]:
```
array([3, 2, 1])
```

Görüldüğü üzere sorunsuz bir şekilde değişkenlere eşitlenmiş oldu.

İki Boyut İçin

In [43]:
```
m = np.arange(16).reshape(4,4)
m
```

Out[43]:
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
```

Diyelim ki bu iki boyutlu diziyi 2. satırdan itibaren *-yani yatay bazlı-* bölmek istediğimizi düşünelim. Bunun için **vsplit** fonksiyonu kullanılır.

In [44]:
```
np.vsplit(m, [2])
```

Out[44]:
```
[array([[0, 1, 2, 3],
        [4, 5, 6, 7]]),
 array([[ 8,  9, 10, 11],
        [12, 13, 14, 15]])]
```

In [45]:
```
ust, alt = np.vsplit(m, [2])
```

In [46]:
```
ust
```

Out[46]:
```
array([[0, 1, 2, 3],
       [4, 5, 6, 7]])
```

In [47]:
```
alt
```

Out[47]:
```
array([[ 8,  9, 10, 11],
       [12, 13, 14, 15]])
```

Dikey bir şekilde bölmek istersek de **hsplit** fonksiyonu kullanılır.

In [48]:
```
m
```

Out[48]:
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
```

In [49]:
```
np.hsplit(m, [2])
```

Out[49]:
```
[array([[ 0,  1],
        [ 4,  5],
        [ 8,  9],
        [12, 13]]),
 array([[ 2,  3],
        [ 6,  7],
        [10, 11],
        [14, 15]])]
```

In [50]:
```
sol, sag = np.hsplit(m, [2])
```

In [51]:
```
sol
```

Out[51]:
```
array([[ 0,  1],
       [ 4,  5],
       [ 8,  9],
       [12, 13]])
```

In [52]:
```
sag
```

Out[52]:
```
array([[ 2,  3],
       [ 6,  7],
```

```
       [10, 11],
       [14, 15]])
```

Sıralama (Sorting)

Tek Boyut İçin

In [54]:
```
v = np.array([2,1,4,3,5])
v
```

Out[54]:
```
array([2, 1, 4, 3, 5])
```

In [55]:
```
np.sort(v)
```

Out[55]:
```
array([1, 2, 3, 4, 5])
```

In [56]:
```
v
```

Out[56]:
```
array([2, 1, 4, 3, 5])
```

Orijinal hali sıralı değil. Bunu değiştirmek istersek NumPy kütüphanesinden değil de normal olan **sort** metodunu kullanmalıyız.

In [57]:
```
v.sort()
```

In [58]:
```
v
```

Out[58]:
```
array([1, 2, 3, 4, 5])
```

Görüldüğü üzere orijinal hali de değişmiş oldu.

İki Boyut İçin

In [61]:
```
m = np.random.normal(20, 5, (3,3))
```

In [62]:
```
m
```

Out[62]:
```
array([[22.58664685, 14.87461731, 20.50387183],
       [23.7495847 , 22.92208939, 17.92861138],
       [20.88148028, 25.26031998, 19.88054741]])
```

Bu dizinin satırlarını sıralamak istersek:

In [63]:
```
np.sort(m, axis = 1)
```

Out[63]:
```
array([[14.87461731, 20.50387183, 22.58664685],
       [17.92861138, 22.92208939, 23.7495847 ],
       [19.88054741, 20.88148028, 25.26031998]])
```

Sütunlarını sıralamak istersek:

In [64]:
```
np.sort(m, axis = 0)
```

Out[64]:
```
array([[20.88148028, 14.87461731, 17.92861138],
       [22.58664685, 22.92208939, 19.88054741],
       [23.7495847 , 25.26031998, 20.50387183]])
```

Eleman İşlemleri

Index ile Elemanlara Erişmek

In [3]:
```
a = np.random.randint(10, size = 10)
a
```

Out[3]:
```
array([4, 3, 7, 3, 6, 2, 3, 1, 0, 5])
```

In [4]:
```
a[0]
```

Out[4]:
```
4
```

In [5]:
```
a[-1]
```

Out[5]:
```
5
```

In [6]:
```
a[0] = 100
```

In [7]:
```
a
```

Out[7]:
```
array([100,   3,   7,   3,   6,   2,   3,   1,   0,   5])
```

In [8]:
```
m = np.random.randint(10, size = (3,5))
m
```

Out[8]:
```
array([[1, 0, 6, 4, 0],
       [4, 4, 9, 4, 9],
       [9, 7, 7, 0, 1]])
```

In [9]:
```
m[0,0]
```

Out[9]:
```
1
```

In [10]:
```
m[1,1]
```

Out[10]:
```
4
```

In [11]:
```
m[1,4]
```

Out[11]:
```
9
```

In [12]:
```
m[1,4] = 99
```

In [13]:
```
m
```

Out[13]:
```
array([[ 1,  0,  6,  4,  0],
       [ 4,  4,  9,  4, 99],
       [ 9,  7,  7,  0,  1]])
```

In [14]:
```
m[1,4] = 2.2
```

In [15]:
```
m
```

Out[15]:
```
array([[1, 0, 6, 4, 0],
       [4, 4, 9, 4, 2],
       [9, 7, 7, 0, 1]])
```

Görüldüğü üzere **NumPy dizileri için** ondalıklı bir sayı eklemek istememize rağmen sadece tamsayı kısmını ekledi. Bunun sebebi eğer **varolan** bir diziden farklı bir tip olarak veri eklemek istersek bunu kabul etmeyecektir. Dizinin yapısı baştan beri neyse ona göre ekleme yapacaktır. Fakat eğer **oluşturma esnasında** bir tane

farklı veri tipi ekleyeceksek dizi ona göre ayak uyduracaktır. Örneğin farklı olan veri tipi **float** ise bütün dizi **float** olarak değişecektir. Aksi halde sonradan eklenen başka bir veri tipi için bunu düzeltmeyecektir.

Array Alt Küme İşlemleri (Slicing)

Tek Boyut İçin

In [17]:
```
a = np.arange(20,30)
a
```

Out[17]:
```
array([20, 21, 22, 23, 24, 25, 26, 27, 28, 29])
```

In [18]:
```
a[0:3]
```

Out[18]:
```
array([20, 21, 22])
```

In [19]:
```
a[:3]
```

Out[19]:
```
array([20, 21, 22])
```

In [20]:
```
a[3:]
```

Out[20]:
```
array([23, 24, 25, 26, 27, 28, 29])
```

In [21]:
```
a[1::2]
```

Out[21]:
```
array([21, 23, 25, 27, 29])
```

In [22]:
```
a[0::2]
```

Out[22]:
```
array([20, 22, 24, 26, 28])
```

In [23]:
```
a[2::2]
```

Out[23]:
```
array([22, 24, 26, 28])
```

In [24]:
```
a[0::3]
```

Out[24]:
```
array([20, 23, 26, 29])
```

İki Boyut İçin

In [26]:
```
m = np.random.randint(10, size=(5,5))
m
```

Out[26]:
```
array([[9, 5, 9, 0, 9],
       [3, 6, 2, 7, 3],
       [5, 0, 0, 3, 9],
       [9, 8, 2, 6, 3],
       [9, 2, 7, 2, 3]])
```

In [27]:
```
m[: , 0]

#Bütün satırları seç, sonra 0. sütunu al demektir.
```

Out[27]:
```
array([9, 3, 5, 9, 9])
```

Not: Matris işlemlerinde öncesi satırı, sonrası sütunu temsil eder.

In [28]:
```
m[: , 1]
```

Out[28]:
```
array([5, 6, 0, 8, 2])
```

In [29]:
```
m[: , 4]
```

Out[29]:
```
array([9, 3, 9, 3, 3])
```

In [31]:
```
m
```

Out[31]:
```
array([[9, 5, 9, 0, 9],
       [3, 6, 2, 7, 3],
       [5, 0, 0, 3, 9],
       [9, 8, 2, 6, 3],
       [9, 2, 7, 2, 3]])
```

In [30]:
```
m[0 , :]
```

Out[30]:
```
array([9, 5, 9, 0, 9])
```

In [32]:
```
m[0]
```

Out[32]:
```
array([9, 5, 9, 0, 9])
```

In [33]:
```
m[1 , :]
```

Out[33]:
```
array([3, 6, 2, 7, 3])
```

In [34]:
```
m[0:2 , 0:3]
```

Out[34]:
```
array([[9, 5, 9],
       [3, 6, 2]])
```

In [35]:
```
m[: , :2]
```

Out[35]:
```
array([[9, 5],
       [3, 6],
       [5, 0],
       [9, 8],
       [9, 2]])
```

In [37]:
```
m
```

Out[37]:
```
array([[9, 5, 9, 0, 9],
       [3, 6, 2, 7, 3],
       [5, 0, 0, 3, 9],
       [9, 8, 2, 6, 3],
       [9, 2, 7, 2, 3]])
```

In [36]:
```
m[1:3 , :2]
```

Out[36]:
```
array([[3, 6],
       [5, 0]])
```

Alt Küme Üzerinde İşlem Yapmak

In [2]:
```
a = np.random.randint(10, size = (5,5))
a
```

Out[2]:
```
array([[0, 1, 2, 6, 4],
       [9, 1, 7, 3, 9],
       [4, 5, 9, 4, 1],
```

```
       [3, 8, 0, 7, 6],
       [8, 0, 3, 7, 7]])
```

In [3]:
```
alt_a = a[0:3 , 0:2]
alt_a
```

Out[3]:
```
array([[0, 1],
       [9, 1],
       [4, 5]])
```

In [4]:
```
alt_a[0,0] = 99999
alt_a[1,1] = 888
```

In [5]:
```
alt_a
```

Out[5]:
```
array([[99999,     1],
       [    9,   888],
       [    4,     5]])
```

In [6]:
```
a
```

Out[6]:
```
array([[99999,     1,     2,     6,     4],
       [    9,   888,     7,     3,     9],
       [    4,     5,     9,     4,     1],
       [    3,     8,     0,     7,     6],
       [    8,     0,     3,     7,     7]])
```

Görüldüğü üzere alt küme işlemleri yaptığımızda dizimizin ilk hali de değişmiş oldu. Fakat bazen bunun olmasını istemeyebiliriz. Bunun için **copy** metodu kullanılmalıdır.

In [7]:
```
m = np.random.randint(10, size = (5,5))
m
```

Out[7]:
```
array([[8, 1, 5, 2, 9],
       [4, 7, 5, 0, 0],
       [7, 8, 9, 0, 1],
       [2, 4, 4, 0, 9],
       [9, 4, 0, 6, 4]])
```

In [8]:
```
alt_m = m[0:3 , 0:2].copy()
alt_m
```

Out[8]:
```
array([[8, 1],
       [4, 7],
```

```
       [7, 8]])
```

In [9]:
```
alt_m[0,0] = 9999
alt_m
```

Out[9]:
```
array([[9999,    1],
       [   4,    7],
       [   7,    8]])
```

In [10]:
```
m
```

Out[10]:
```
array([[8, 1, 5, 2, 9],
       [4, 7, 5, 0, 0],
       [7, 8, 9, 0, 1],
       [2, 4, 4, 0, 9],
       [9, 4, 0, 6, 4]])
```

Görüldüğü üzere ana dizimizde hiçbir değişiklik olmadı.

Fancy Index ile Elemanlara Erişmek

Uyarı: Bu kavram, ilerleyen bölümlerde en önemli kavramlardan birisi olacak. Fancy, kelime anlamıyla fantastik, büyüleyici anlamlarına gelmektedir. Bize hem ileride göreceğimiz pandas dataframe'lerinde hem de numpy array'lerinde daha ileri düzey eleman seçme imkanları vermektedir. Bu, bizim mevcut bildiğimiz yaklaşımlardan biraz daha ileri seviyede ve varlığını tam anlamıyla kavradığımızda daha ileri düzeyde elemanlara erişmek ihtiyaçlarımızı karşılayacak olan bir yaklaşımdır. Özellikle fancy'ye dair bir şey gösterilmeyecek fakat arka taraftaki fancy yaklaşımı/mantığı çok iyi bir şekilde anlaşılırsa bunları ne zaman kullanmamız gerektiğini tam olarak kafamızda oturtmuş olacağız.

Tek Boyut İçin

In [11]:
```
v = np.arange(0, 30, 3)
v
```

Out[11]:
```
array([ 0,  3,  6,  9, 12, 15, 18, 21, 24, 27])
```

Normalde erişmek istediğimiz zaman aşağıdaki şekliyle yazarız:

In [12]:
```
v[1]
```

Out[12]:
```
3
```

In [13]:
```
v[3]
```

Out[13]:
```
9
```

In [14]:
```
v[5]
```

Out[14]:
```
15
```

In [15]:
```
[v[1], v[3], v[5]]
```

Out[15]:
```
[3, 9, 15]
```

Sorumuz şu: Elimizdeki 3 tane bilgi için bu şekilde erişim işlemi yapabildik. Fakat öyle ki, bir fonksiyonun veya döngünün çıktısı elimde yüzlerce index bilgisi taşıyor. Ve biz bu index bilgilerinin her birisini elimizdeki array'in içerisinden gidip yakalamak istiyoruz. Bunu nasıl yaparız?

İlk aklımıza döngü geliyor olabilir fakat bunu tek seferde yapmanın yolu fancy index'tir.

In [16]:
```
al_getir = [1,3,5]
```

In [17]:
```
v
```

Out[17]:
```
array([ 0,  3,  6,  9, 12, 15, 18, 21, 24, 27])
```

In [18]:
```
v[al_getir]
```

Out[18]:
```
array([ 3,  9, 15])
```

İşte bu şekilde çağırma işlemine fancy index denir. Yani biz bir listeye birçok değer yazdığımızda ve bunu **v[]** içerisine yazdığımızda bunu algılıyor ve gerekli değerleri getiriyor.

Not: Burada fancy indexin arka planda nasıl çalıştığı gösterilmiyor; sadece fancy indexin ne yaptığı gösteriliyor.

İki Boyut İçin

In [30]:
```
m = np.arange(9).reshape((3,3))
```

In [31]:
```
m
```

Out[31]:
```
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
```

In [32]:
```
eleman_1 = np.array([0,1]) #0. satır 1. sütundaki eleman
eleman_2 = np.array([1,2]) #1. satır 2. sütundaki eleman
```

In [33]:
```
m[eleman_1, eleman_2]
```

Out[33]:
```
array([1, 5])
```

Basit Index ile Fancy Index

In [34]:
```
m
```

Out[34]:
```
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
```

In [35]:
```
m[0, [1,2]]

#Burada basit index ile fancy indexi bir arada kullanmış olduk.
```

Out[35]:
```
array([1, 2])
```

Slice ile Fancy Index

In [41]:
```
m[0: , [1,2]]

#Burada da slice ve fancy indexi bir arada kullandık.
```

Out[41]:
```
array([[1, 2],
       [4, 5],
       [7, 8]])
```

Yani burada kavramları iyi anlayıp, problemlerle karşılaşıldığında ona göre çözüm üretmek gerekiyor.

Koşullu Eleman İşlemleri

In [42]:
```
v = np.array([1,2,3,4,5])
```

In [43]:
```
v > 5
```

Out[43]:
```
array([False, False, False, False, False])
```

In [44]:
```
v < 3
```

Out[44]:
```
array([ True,  True, False, False, False])
```

Bu şekilde koşulu sağlayanları gözlemlemiş olduk. Fakat bu koşulları sağlayan elemanları getirmek isteyebiliriz. İşte burada da fancy index devreye giriyor.

In [45]:
```
v[v < 3]
```

Out[45]:
```
array([1, 2])
```

In [46]:
```
v[v > 3]
```

Out[46]:
```
array([4, 5])
```

In [47]:
```
v[v >= 3]
```

Out[47]:
```
array([3, 4, 5])
```

In [48]:
```
v[v <= 3]
```

Out[48]:
```
array([1, 2, 3])
```

In [49]:
```
v[v == 3]
```

Out[49]:
```
array([3])
```

In [50]:
```
v[v != 3]
```

Out[50]:
```
array([1, 2, 4, 5])
```

In [51]:
```
v * 2
```

Out[51]:
```
array([ 2,  4,  6,  8, 10])
```

In [52]:
```
v / 5
```

Out[52]:
```
array([0.2, 0.4, 0.6, 0.8, 1. ])
```

In [53]:
```
v * 5 / 10
```

Out[53]:
```
array([0.5, 1. , 1.5, 2. , 2.5])
```

In [54]:
```
v ** 2
```

Out[54]:
```
array([ 1,  4,  9, 16, 25], dtype=int32)
```

Matematiksel İşlemler

Matematiksel İşlemler'e Giriş

In [1]:
```
v = np.array([1,2,3,4,5])
```

In [2]:
```
v - 1
```

Out[2]:
```
array([0, 1, 2, 3, 4])
```

In [3]:
```
v * 5
```

Out[3]:
```
array([ 5, 10, 15, 20, 25])
```

In [4]:
```
v / 5
```

Out[4]:
```
array([0.2, 0.4, 0.6, 0.8, 1. ])
```

In [5]:
```
v * 5 / 10 - 1
```

Out[5]:
```
array([-0.5,  0. ,  0.5,  1. ,  1.5])
```

Biz bu matematiksel işlemleri **ufunc** sayesinde yapabiliyoruz. Bunun anlamı, biz bir matematiksel işlem yaptığımız zaman numpy arka planda gerekli fonksiyonları çalıştırıyor.

In [6]:
```
np.subtract(v, 1)
```

```
#Örneğin ilk yaptığımız işlem bunu yapıyor.
```

Out[6]:
```
array([0, 1, 2, 3, 4])
```

In [7]:
```
np.add(v, 1)
```

Out[7]:
```
array([2, 3, 4, 5, 6])
```

In [9]:
```
np.multiply(v, 5)
```

Out[9]:
```
array([ 5, 10, 15, 20, 25])
```

In [10]:
```
np.divide(v, 5)
```

Out[10]:
```
array([0.2, 0.4, 0.6, 0.8, 1. ])
```

In [11]:
```
v ** 3
```

Out[11]:
```
array([  1,   8,  27,  64, 125], dtype=int32)
```

In [12]:
```
np.power(v, 3)
```

Out[12]:
```
array([  1,   8,  27,  64, 125], dtype=int32)
```

Görüldüğü üzere yaptığımız matematiksel işlemlerin arka planında bu fonksiyonlar çalışıyor.

In [13]:
```
v % 2
```

Out[13]:
```
array([1, 0, 1, 0, 1], dtype=int32)
```

In [14]:
```
np.mod(v, 2)
```

Out[14]:
```
array([1, 0, 1, 0, 1], dtype=int32)
```

In [15]:
```
np.absolute(np.array([-3]))
```

Out[15]:
```
array([3])
```

In [16]:
```
np.sin(360)
```

Out[16]:
```
0.9589157234143065
```

In [17]:
```
np.cos(180)
```

Out[17]:
```
-0.5984600690578581
```

In [18]:
```
v = np.array([1,2,3])
```

In [19]:
```
np.log(v)
```

Out[19]:
```
array([0.        , 0.69314718, 1.09861229])
```

In [20]:
```
np.log2(v)
```

Out[20]:
```
array([0.       , 1.       , 1.5849625])
```

In [21]:
```
np.log10(v)
```

Out[21]:
```
array([0.        , 0.30103   , 0.47712125])
```

In [22]:
```
?np
```

```
Type:        module
String form: <module 'numpy' from 'C:\\Users\\ertug\\anaconda3\\envs\\tf\\lib\\site-packages\\numpy\\__init__.py'>
File:        c:\users\ertug\anaconda3\envs\tf\lib\site-packages\numpy\__init__.py
Docstring:
NumPy
=====

Provides
  1. An array object of arbitrary homogeneous items
  2. Fast mathematical operations over arrays
  3. Linear Algebra, Fourier Transforms, Random Number Generation

How to use the documentation
----------------------------
Documentation is available in two forms: docstrings provided
```

```
with the code, and a loose standing reference guide, available from
`the NumPy homepage <https://www.scipy.org>`_.

We recommend exploring the docstrings using
`IPython <https://ipython.org>`_, an advanced Python shell with
TAB-completion and introspection capabilities.  See below for further
instructions.

The docstring examples assume that `numpy` has been imported as `np`::

  >>> import numpy as np

Code snippets are indicated by three greater-than signs::

  >>> x = 42
  >>> x = x + 1

Use the built-in ``help`` function to view a function's docstring::

  >>> help(np.sort)
  ... # doctest: +SKIP

For some objects, ``np.info(obj)`` may provide additional help.  This is
particularly true if you see the line "Help on ufunc object:" at the top
of the help() page.  Ufuncs are implemented in C, not Python, for speed.
The native Python help() does not know how to view their help, but our
np.info() function does.

To search for documents containing a keyword, do::

  >>> np.lookfor('keyword')
  ... # doctest: +SKIP

General-purpose documents like a glossary and help on the basic concepts
of numpy are available under the ``doc`` sub-module::

  >>> from numpy import doc
  >>> help(doc)
  ... # doctest: +SKIP

Available subpackages
---------------------
doc
    Topical documentation on broadcasting, indexing, etc.
lib
    Basic functions used by several sub-packages.
random
    Core Random Tools
linalg
    Core Linear Algebra Tools
fft
    Core FFT routines
```

```
polynomial
    Polynomial tools
testing
    NumPy testing tools
f2py
    Fortran to Python Interface Generator.
distutils
    Enhancements to distutils with support for
    Fortran compilers support and more.

Utilities
---------
test
    Run numpy unittests
show_config
    Show numpy build configuration
dual
    Overwrite certain functions with high-performance Scipy tools
matlib
    Make everything matrices.
__version__
    NumPy version string

Viewing documentation using IPython
-----------------------------------
Start IPython with the NumPy profile (``ipython -p numpy``), which will
import `numpy` under the alias `np`.  Then, use the ``cpaste`` command to
paste examples into the shell.  To see which functions are available in
`numpy`, type ``np.<TAB>`` (where ``<TAB>`` refers to the TAB key), or use
``np.*cos*?<ENTER>`` (where ``<ENTER>`` refers to the ENTER key) to narrow
down the list.  To view the docstring for a function, use
``np.cos?<ENTER>`` (to view the docstring) and ``np.cos??<ENTER>`` (to view
the source code).

Copies vs. in-place operation
-----------------------------
Most of the functions in `numpy` return a copy of the array argument
(e.g., `np.sort`).  In-place versions of these functions are often
available as array methods, i.e. ``x = np.array([1,2,3]); x.sort()``.
Exceptions to this rule are documented.
```

İstatistiksel Hesaplamalar

In [24]:
```
v
```

Out[24]:
```
array([1, 2, 3])
```

In [25]:
```
np.mean(v)
```

Out[25]: 2.0

In [26]:
```
v.sum()
```

Out[26]: 6

In [27]:
```
v.min()
```

Out[27]: 1

In [ ]:
```
np.mean(arr,axis=0) | Returns mean along specific axis
arr.sum() | Returns sum of arr
arr.min() | Returns minimum value of arr
arr.max(axis=0) | Returns maximum value of specific axis
np.var(arr) | Returns the variance of array
np.std(arr,axis=1) | Returns the standard deviation of specific axis
arr.corrcoef() | Returns correlation coefficient of array
```

NumPy ile İki Bilinmeyenli Denklem Çözümü

5 * x0 + x1 = 12

x0 + 3 * x1 = 10

Not: Aynı hücrede alt satıra inmek için çift space tuşuna basılmalı veya `<br>` komutu yazılmalıdır.

Bu denklemi NumPy'ın daha iyi anlaması için bilinmeyenlerin katsayılarını bir vektöre koymak, bunun sonucunda oluşan değerleri başka bir vektöre koymamız gerekiyor. En sonunda NumPy'ın **linarg.solve** komutunu kullanarak çözümü gerçekleştireceğiz.

In [2]:
```
a = np.array([[5,1], [1,3]])
b = np.array([12,10])
```

In [3]:
```
a
```

Out[3]:
```
array([[5, 1],
       [1, 3]])
```

In [4]:
```
b
```

Out[4]:
```
array([12, 10])
```

In [5]:
```
x = np.linalg.solve(a, b)
x
```

Out[5]:
```
array([1.85714286, 2.71428571])
```

Veri Manipülasyonu 201

Pandas Serileri

Pandas Giriş

- Panel Data'nın kısaltmasıdır.
- Veri manipülasyonu ve veri analizi için yazılmış açık kaynak kodlu bir Python kütüphanesidir.
- Ekonometrik ve finansal çalışmalar için doğmuştur.
- Temeli 2008 yılında atılmıştır.
- R DataFrame yapısını Python dünyasına taşımış ve DataFrame'ler üzerinde hızlı ve etkili çalışabilme imkanı sağlamıştır.
- Birçok farklı veri tipini okuma ve yazma imkanı sağlar.

Pandas Serisi Oluşturma

Değerleri ve indexleri beraber tutan ve gösteren tek boyutlu bir yapıdır. Boyut arttığında buna **DataFrame** denir.

In [3]:
```
pd.Series([1,2,3,4,5])
```

Out[3]:
```
0    1
1    2
2    3
3    4
4    5
dtype: int64
```

In [4]:
```
pd.Series([10,88,3,4,5])
```

Out[4]:
```
0    10
1    88
2     3
3     4
4     5
dtype: int64
```

In [5]:

```
seri = pd.Series([10,88,3,4,5])
```

In [6]:
```
type(seri)
```

Out[6]:
```
pandas.core.series.Series
```

In [7]:
```
seri.axes

#Index bilgisini gösterir.
```

Out[7]:
```
[RangeIndex(start=0, stop=5, step=1)]
```

In [8]:
```
seri.dtype
```

Out[8]:
```
dtype('int64')
```

In [9]:
```
seri.size
```

Out[9]:
```
5
```

In [10]:
```
seri.ndim
```

Out[10]:
```
1
```

Örneğin sıkça karşılaşılacağı üzere sadece değerlere erişmek istersek **values** fonksiyonunu kullanmalıyız.

In [11]:
```
seri.values
```

Out[11]:
```
array([10, 88,  3,  4,  5], dtype=int64)
```

In [12]:
```
seri.head()
```

Out[12]:
```
0    10
1    88
2     3
3     4
4     5
dtype: int64
```

In [13]:
```
seri.head(3)
```

Out[13]:
```
0    10
1    88
2     3
dtype: int64
```

In [14]:
```
seri.tail(3)
```

Out[14]:
```
2    3
3    4
4    5
dtype: int64
```

Index İsimlendirmesi

In [15]:
```
pd.Series([99,22,332,94,5])
```

Out[15]:
```
0     99
1     22
2    332
3     94
4      5
dtype: int64
```

In [16]:
```
pd.Series([99,22,332,94,5], index = [1,3,5,7,9])
```

Out[16]:
```
1     99
3     22
5    332
7     94
9      5
dtype: int64
```

In [17]:
```
pd.Series([99,22,332,94,5], index = ["a","b","c","d","e"])
```

Out[17]:
```
a     99
b     22
c    332
d     94
e      5
dtype: int64
```

In [18]:
```
seri = pd.Series([99,22,332,94,5], index = ["a","b","c","d","e"])
```

In [19]:
```
seri["a"]
```

Out[19]:
```
99
```

In [20]:
```
seri["a":"c"]
```

Out[20]:
```
a     99
b     22
c    332
dtype: int64
```

Sözlük Üzerinden Liste Oluşturma

In [26]:
```
sozluk = {"reg":10, "log":11, "cart":12}
```

In [27]:
```
seri = pd.Series(sozluk)
```

In [28]:
```
seri
```

Out[28]:
```
reg     10
log     11
cart    12
dtype: int64
```

İki Seriyi Birleştirerek Seri Oluşturma

In [29]:
```
pd.concat([seri, seri])
```

Out[29]:
```
reg     10
log     11
cart    12
reg     10
log     11
cart    12
dtype: int64
```

Eleman İşlemleri

In [31]:
```
a = np.array([1,2,33,444,75])
```

NumPy dizisinden de seri oluşturulabilir.

In [32]:
```
seri = pd.Series(a)
seri
```

Out[32]:
```
0      1
1      2
2     33
3    444
4     75
dtype: int32
```

In [33]:
```
seri[0]
```

Out[33]:
```
1
```

In [34]:
```
seri[0:3]
```

Out[34]:
```
0     1
1     2
2    33
dtype: int32
```

In [35]:
```
seri = pd.Series([121,200,150,99],
                 index = ["reg","loj","cart","rf"])
```

In [36]:
```
seri
```

Out[36]:
```
reg     121
loj     200
cart    150
rf       99
dtype: int64
```

In [37]:
```
seri.index
```

Out[37]:
```
Index(['reg', 'loj', 'cart', 'rf'], dtype='object')
```

In [38]:
```
seri.keys
```

Out[38]:
```
<bound method Series.keys of reg     121
loj     200
cart    150
```

```
rf       99
dtype: int64>
```

In [39]:
```
list(seri.items())
```

Out[39]:
```
[('reg', 121), ('loj', 200), ('cart', 150), ('rf', 99)]
```

Görüldüğü üzere serilerde eğer indexler sayı değil de string yani **anahtar** bazında ise onları getirmenin yolu **items()** fonksiyonudur.

In [40]:
```
seri.values
```

Out[40]:
```
array([121, 200, 150,  99], dtype=int64)
```

Eleman Sorgulama

In [41]:
```
"reg" in seri
```

Out[41]:
```
True
```

In [42]:
```
"a" in seri
```

Out[42]:
```
False
```

In [43]:
```
seri["reg"]
```

Out[43]:
```
121
```

Fancy Eleman

In [44]:
```
seri[["rf", "reg"]]
```

Out[44]:
```
rf      99
reg    121
dtype: int64
```

In [45]:
```
seri["reg"] = 130
```

In [47]:
```
seri["reg"]
```

Out[47]:
```
130
```

In [48]:
```
seri["reg":"loj"]
```

Out[48]:
```
reg    130
loj    200
dtype: int64
```

Pandas DataFrame

Pandas DataFrame Oluşturma

Pandas DataFrame, yapısal bir veri tipidir. Excel veri yapısına benzerdir. NumPy varken DataFrame'e ihtiyaç duyulmasının sebebi, NumPy, **fixtype** yani sabit veri tipli yapıya sahip olduğundan hem kategorik hem de sürekli değişkenler üzerinden işlem yapmaya konusunda pek başarılı değildir. Yani veri manipülasyonu ve veri analizi hususunda NumPy bize pek yardım edememektedir. Bu sebeplerden ötürü **Pandas DataFrame'e** ihtiyaç duyuldu.

Matematiksel anlamda vektör ve matris uzayında işlemler yapılacak olduğunda burada Pandas'ı bile kullanıyor olsak o zaten arka planda NumPy'ı kullanıyor. Dolayısıyla NumPy'ı biraz daha teorik yapılar için düşünebiliriz. Pandas'ı yani Pandas DataFrame'ini ise daha analitik anlamda makine öğrenmesi modellerine verecek olduğumuz veri setleri olarak düşünebiliriz.

In [2]:
```
l = [1,2,39,67,90]
l
```

Out[2]:
```
[1, 2, 39, 67, 90]
```

In [3]:
```
pd.DataFrame(l, columns = ["degisken_ismi"])
```

Out[3]:

	degisken_ismi
0	1
1	2
2	39
3	67
4	90

In [4]:
```
m = np.arange(1,10).reshape(3,3)
m
```

Out[4]:
```
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

In [5]:
```
pd.DataFrame(m, columns = ["var1","var2","var3"])
```

Out[5]:

	var1	var2	var3
0	1	2	3
1	4	5	6
2	7	8	9

DataFrame İsimlendirme

In [7]:
```
df = pd.DataFrame(m, columns = ["var1","var2","var3"])
df.head()
```

Out[7]:

	var1	var2	var3
0	1	2	3
1	4	5	6
2	7	8	9

In [8]:
```
df.columns
```

Out[8]:
```
Index(['var1', 'var2', 'var3'], dtype='object')
```

In [9]:
```
df.columns = ("deg1", "deg2", "deg3")
df
```

Out[9]:

	deg1	deg2	deg3
0	1	2	3
1	4	5	6
2	7	8	9

In [10]:
```
type(df)
```

Out[10]:
```
pandas.core.frame.DataFrame
```

In [11]:
```
df.axes #Index bilgileri
```

Out[11]:
```
[RangeIndex(start=0, stop=3, step=1),
 Index(['deg1', 'deg2', 'deg3'], dtype='object')]
```

In [14]:
```
df.shape #Satır-sütun bilgisi
```

Out[14]:
```
(3, 3)
```

In [15]:
```
df.ndim #Boyut sayısı
```

Out[15]:
```
2
```

In [16]:
```
df.size #Eleman sayısı
```

Out[16]:
```
9
```

In [17]:
```
df.values #DataFrame'deki değerler
```

Out[17]:
```
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

values fonksiyonu df'deki değerleri NumPy dizisine çevirir.

In [18]:
```
type(df.values)
```

Out[18]:
```
numpy.ndarray
```

Tip bilgisinden de bu anlaşılmaktadır.

In [19]:
```
df.head()
```

Out[19]:

	deg1	deg2	deg3
0	1	2	3

	deg1	deg2	deg3
1	4	5	6
2	7	8	9

In [21]:
```
df.tail(1)
```

Out[21]:

	deg1	deg2	deg3
2	7	8	9

In [22]:
```
a = np.array([1,2,3,4,5])
```

In [23]:
```
pd.DataFrame(a, columns=["deg1"])
```

Out[23]:

	deg1
0	1
1	2
2	3
3	4
4	5

Eleman İşlemleri

In [2]:
```
s1 = np.random.randint(10, size = 5)
s2 = np.random.randint(10, size = 5)
s3 = np.random.randint(10, size = 5)
```

In [3]:
```
sozluk = {"var1":s1, "var2":s2, "var3":s3}
sozluk
```

Out[3]:
```
{'var1': array([0, 8, 2, 6, 3]),
 'var2': array([5, 0, 7, 9, 0]),
 'var3': array([1, 1, 0, 9, 4])}
```

In [4]:
```
df = pd.DataFrame(sozluk)
df
```

Out[4]:

	var1	var2	var3
0	0	5	1
1	8	0	1
2	2	7	0
3	6	9	9
4	3	0	4

Görüldüğü üzere iç içe veri yapıları ile de DataFrame oluşturabiliyoruz.

In [5]:
```
df[0:1]
```

Out[5]:

	var1	var2	var3
0	0	5	1

In [6]:
```
df.index
```

Out[6]:
```
RangeIndex(start=0, stop=5, step=1)
```

In [7]:
```
df.index = ["a","b","c","d","e"]
```

In [8]:
```
df
```

Out[8]:

	var1	var2	var3
a	0	5	1
b	8	0	1
c	2	7	0
d	6	9	9
e	3	0	4

In [9]:
```
df["c":"e"]
```

Out[9]:

	var1	var2	var3
c	2	7	0
d	6	9	9
e	3	0	4

Silme

In [11]:
```
df.drop("a", axis=0)

# a indexi satırda olduğu için axis=0 argümanını kullandık. Sütun için axis=1 şeklinde kullanmamız gerekiyor.
```

Out[11]:

	var1	var2	var3
b	8	0	1
c	2	7	0
d	6	9	9
e	3	0	4

In [12]:
```
df
```

Out[12]:

	var1	var2	var3
a	0	5	1
b	8	0	1
c	2	7	0
d	6	9	9
e	3	0	4

Ana DataFrame'de bir değişiklik olmadı. Bunun için **inplace** argümanını kullanmamız gerekiyor.

In [13]:
```
df.drop("a", axis=0, inplace=True)
```

In [14]:
```
```

```
df
```

Out[14]:

	var1	var2	var3
b	8	0	1
c	2	7	0
d	6	9	9
e	3	0	4

Fancy Index

In [15]:
```
l = ["c","e"]
```

In [16]:
```
df.drop(l, axis=0)
```

Out[16]:

	var1	var2	var3
b	8	0	1
d	6	9	9

Görüldüğü üzere fancy index yardımıyla çoklu silme işlemi de yapabiliriz.

Değişkenler İçin

In [18]:
```
df
```

Out[18]:

	var1	var2	var3
b	8	0	1
c	2	7	0
d	6	9	9
e	3	0	4

In [17]:
```
"var1" in df

#Bu şekilde sorgulandığında sütunda olup olmadığını kontrol edecektir..
```

Out[17]:
```
True
```

In [19]:
```
l = ["var1","var4","var2"]
```

In [20]:
```
for i in l:
    print(i in df)
```

```
True
False
True
```

In [25]:
```
df
```

Out[25]:

	var1	var2	var3
b	8	0	1
c	2	7	0
d	6	9	9
e	3	0	4

In [27]:
```
df["var1"]

#Herhangi bir sütunu seçmek istediğimizde bu şekilde kullanmalıyız.
```

Out[27]:
```
b    8
c    2
d    6
e    3
Name: var1, dtype: int32
```

In [32]:
```
df["var4"] = df["var1"] / df["var2"]
```

In [33]:
```
df
```

Out[33]:

	var1	var2	var3	var4
b	8	0	1	inf
c	2	7	0	0.285714

	var1	var2	var3	var4
d	6	9	9	0.666667
e	3	0	4	inf

Değişken Silme

In [34]:
```
df.drop("var4", axis=1)
```

Out[34]:

	var1	var2	var3
b	8	0	1
c	2	7	0
d	6	9	9
e	3	0	4

In [35]:
```
df
```

Out[35]:

	var1	var2	var3	var4
b	8	0	1	inf
c	2	7	0	0.285714
d	6	9	9	0.666667
e	3	0	4	inf

In [36]:
```
df.drop("var4", axis=1, inplace=True)
```

In [37]:
```
df
```

Out[37]:

	var1	var2	var3
b	8	0	1
c	2	7	0
d	6	9	9

	var1	var2	var3
e	3	0	4

In [38]:
```
l = ["var1","var2"]
```

In [39]:
```
df.drop(l, axis=1)
```

Out[39]:

	var3
b	1
c	0
d	9
e	4

Gözlem ve Değişken Seçimi: loc & iloc

Pandas DataFrame'de en çok karşılaşılan sorun gözlem ve değişken seçimi sorunudur. Çünkü liste ve NumPy veri yapısından biraz farklıdır. **stackoverflow.com** sitesinde de Pandas DataFrame için en çok sorulan hatalardan bir tanesidir.

In [40]:
```
m = np.random.randint(1,30, size=(10,3))
df = pd.DataFrame(m, columns=["var1","var2","var3"])
df
```

Out[40]:

	var1	var2	var3
0	19	24	21
1	17	19	29
2	14	14	21
3	1	8	18
4	15	6	26
5	14	21	10
6	9	20	15
7	1	5	16

	var1	var2	var3
8	1	9	22
9	10	14	24

loc: Tanımlandığı şekliyle seçim yapmak için kullanılır.

In [41]:
```
df.loc[0:3]
```

Out[41]:

	var1	var2	var3
0	19	24	21
1	17	19	29
2	14	14	21
3	1	8	18

Bu şekilde kullanıldığında veri setinin ilk halindeki indexlemeye sadık kalacak şekilde yani nasıl bir indexleme varsa buna göre bir seçim imkanı veriyor.

iloc: Alışık olduğumuz indexleme mantığı ile seçim yapar.

In [42]:
```
df.iloc[0:3]
```

Out[42]:

	var1	var2	var3
0	19	24	21
1	17	19	29
2	14	14	21

loc fonksiyonu ile benzer bir işlem yaptığı gözlemleniyor. Fakat bu fonksiyonda sonuncu indexi almamaktadır.

Yani verilen indexlere sadık kalarak tanımlandığı şekli ile seçim yapmak istediğimizde **loc** fonksiyonunu kullanıyoruz ve girmiş olduğumuz indexlerin birebir eşleşecek şekilde yakalanmasını sağlıyor. **iloc** fonksiyonu ise klasik anlamda indexleme mantığı ile veri seti içerisindeki index yapısını görmezden gelerek -diğer ifadeyle sıfırlayarak- normal şekilde seçim imkanı sağlar.

In [43]:
```
df.iloc[0,0]
```

Out[43]: 19

In [44]:
```
df.iloc[:3 , :2]
```

Out[44]:

	var1	var2
0	19	24
1	17	19
2	14	14

In [45]:
```
df.loc[0:3 , "var3"]
```

Out[45]:
```
0    21
1    29
2    21
3    18
Name: var3, dtype: int32
```

Görüldüğü üzere 3'ü de dahil ederek ekrana yazdırdı.

In [46]:
```
df.iloc[0:3 , "var3"]
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
~\anaconda3\envs\tf\lib\site-packages\pandas\core\indexing.py in _has_valid_tuple(self, key)
    701             try:
--> 702                 self._validate_key(k, i)
    703             except ValueError as err:

~\anaconda3\envs\tf\lib\site-packages\pandas\core\indexing.py in _validate_key(self, key, axis)
   1368         else:
-> 1369             raise ValueError(f"Can only index by location with a [{self._valid_types}]")
   1370

ValueError: Can only index by location with a [integer, integer slice (START point is INCLUDED, END point is EXCLUDED), listlike of integers,
boolean array]

The above exception was the direct cause of the following exception:

ValueError                                Traceback (most recent call last)
<ipython-input-46-7abfd2d4a811> in <module>
----> 1 df.iloc[0:3 , "var3"]

~\anaconda3\envs\tf\lib\site-packages\pandas\core\indexing.py in __getitem__(self, key)
    871                     # AttributeError for IntervalTree get_value
    872                     pass
--> 873             return self._getitem_tuple(key)
    874         else:
```

```
    875             # we by definition only have the 0th axis

~\anaconda3\envs\tf\lib\site-packages\pandas\core\indexing.py in _getitem_tuple(self, tup)
   1441     def _getitem_tuple(self, tup: Tuple):
   1442
-> 1443         self._has_valid_tuple(tup)
   1444         try:
   1445             return self._getitem_lowerdim(tup)

~\anaconda3\envs\tf\lib\site-packages\pandas\core\indexing.py in _has_valid_tuple(self, key)
    702                 self._validate_key(k, i)
    703             except ValueError as err:
--> 704                 raise ValueError(
    705                     "Location based indexing can only have "
    706                     f"[{self._valid_types}] types"

ValueError: Location based indexing can only have [integer, integer slice (START point is INCLUDED, END point is EXCLUDED), listlike of integ
ers, boolean array] types
```

Görüldüğü üzere aynı kullanımı **iloc** için yaptığımızda hata ile karşılaştık. Bu, çok sık karşılaşılan bir hatadır. Eğer değişken ya da satırlarla ilgili mutlak bir değer işaretlemesi yapacaksak bu durumda **loc** fonksiyonunu kullanmalıyız. Normal indexli yaklaşım ile seçmek istiyorsak **iloc** fonksiyonunu kullanmamız gerekiyor.

In [47]:
```
df.iloc[0:3 , 1:3]
```

Out[47]:

	var2	var3
0	24	21
1	19	29
2	14	21

In [48]:
```
df.iloc[0:3]
```

Out[48]:

	var1	var2	var3
0	19	24	21
1	17	19	29
2	14	14	21

In [49]:
```
df.iloc[0:3]["var3"]
```

Out[49]:
```
0    21
1    29
```

```
2    21
Name: var3, dtype: int32
```

Özetle eğer verilen kurallara bağlı bir şekilde seçim yapılma ihtiyacı varsa (gözlem ya da değişken isimlendirmeleri açısından) **loc** fonksiyonu kullanılır. Eğer verilen isimlendirmelerden bağımsız **"klasik index mantığı ile"** seçim yapılmak isteniyorsa **iloc** fonksiyonu kullanılır.

Koşullu Eleman İşlemleri

In [50]:
```
m = np.random.randint(1,30, size=(10,3))
df = pd.DataFrame(m, columns=["var1","var2","var3"])
df
```

Out[50]:

	var1	var2	var3
0	28	4	27
1	6	15	24
2	29	21	3
3	9	20	4
4	7	23	9
5	16	16	4
6	26	24	8
7	12	18	14
8	4	7	23
9	27	17	2

In [51]:
```
df["var1"]
```

Out[51]:
```
0    28
1     6
2    29
3     9
4     7
5    16
6    26
7    12
8     4
9    27
Name: var1, dtype: int32
```

In [52]:
```
df["var1"][0:2]
```

Out[52]:
```
0    28
1     6
Name: var1, dtype: int32
```

In [53]:
```
df[0:2]
```

Out[53]:

	var1	var2	var3
0	28	4	27
1	6	15	24

In [54]:
```
df[0:2]["var1"]
```

Out[54]:
```
0    28
1     6
Name: var1, dtype: int32
```

In [55]:
```
df[0:2]["var1","var2"]
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
~\anaconda3\envs\tf\lib\site-packages\pandas\core\indexes\base.py in get_loc(self, key, method, tolerance)
   2897             try:
-> 2898                 return self._engine.get_loc(casted_key)
   2899             except KeyError as err:

pandas\_libs\index.pyx in pandas._libs.index.IndexEngine.get_loc()

pandas\_libs\index.pyx in pandas._libs.index.IndexEngine.get_loc()

pandas\_libs\hashtable_class_helper.pxi in pandas._libs.hashtable.PyObjectHashTable.get_item()

pandas\_libs\hashtable_class_helper.pxi in pandas._libs.hashtable.PyObjectHashTable.get_item()

KeyError: ('var1', 'var2')

The above exception was the direct cause of the following exception:

KeyError                                  Traceback (most recent call last)
<ipython-input-55-0efef18e406b> in <module>
----> 1 df[0:2]["var1","var2"]

~\anaconda3\envs\tf\lib\site-packages\pandas\core\frame.py in __getitem__(self, key)
   2904             if self.columns.nlevels > 1:
```

```
   2905                 return self._getitem_multilevel(key)
-> 2906             indexer = self.columns.get_loc(key)
   2907             if is_integer(indexer):
   2908                 indexer = [indexer]

~\anaconda3\envs\tf\lib\site-packages\pandas\core\indexes\base.py in get_loc(self, key, method, tolerance)
   2898                 return self._engine.get_loc(casted_key)
   2899             except KeyError as err:
-> 2900                 raise KeyError(key) from err
   2901
   2902         if tolerance is not None:

KeyError: ('var1', 'var2')
```

Bu şekilde değil de fancy yardımıyla bunu yapmamız gerekiyor.

In [56]:
```
df[0:2][["var1","var2"]]
```

Out[56]:

	var1	var2
0	28	4
1	6	15

In [57]:
```
df
```

Out[57]:

	var1	var2	var3
0	28	4	27
1	6	15	24
2	29	21	3
3	9	20	4
4	7	23	9
5	16	16	4
6	26	24	8
7	12	18	14
8	4	7	23
9	27	17	2

In [58]:
```
df.var1
```

Out[58]:
```
0    28
1     6
2    29
3     9
4     7
5    16
6    26
7    12
8     4
9    27
Name: var1, dtype: int32
```

Bu şekliyle de sütun seçme işlemi yapabiliriz.

In [59]:
```
df[df.var1 > 15]
```

Out[59]:

	var1	var2	var3
0	28	4	27
2	29	21	3
5	16	16	4
6	26	24	8
9	27	17	2

Köşeli parantezde yazmamızın sebebi df'in içerisine girmek istediğimizi ve bize değer(ler) döndürmesi gerektiğini belirtmemizdir.
Bu kodun anlamı, var1 değişkeninde 15'ten büyük olan değerleri filtrele ve ona göre getir demektir.
`df[df["var1"] > 15]` şeklinde de kullanılabilir fakat kolaylık açısından bu şekilde kullandık.

In [63]:
```
df[df.var1 > 15]["var1"]

#Bu koşulun içinden var1 değişkeninin getir demektir.
```

Out[63]:
```
0    28
2    29
5    16
6    26
9    27
Name: var1, dtype: int32
```

In [64]:
```
df[df.var1 > 15]["var2"]
```

Out[64]:
```
0     4
2    21
5    16
6    24
9    17
Name: var2, dtype: int32
```

In [68]:
```
df[(df.var1 > 15) & (df.var3 < 5)]
```

Out[68]:

	var1	var2	var3
2	29	21	3
5	16	16	4
9	27	17	2

Görüldüğü üzere birden fazla koşul da girilebilir.

In [70]:
```
df.loc[(df.var1 > 15), ["var1","var2"]]

#loc fonksiyonu bu şekilde de kullanılabilir.
```

Out[70]:

	var1	var2
0	28	4
2	29	21
5	16	16
6	26	24
9	27	17

In [72]:
```
df[df.var1 > 15][["var1","var2"]]

#Aynı çıktığı çağırmanın diğer yolu da bu şekildedir.
```

Out[72]:

	var1	var2
0	28	4
2	29	21
5	16	16

	var1	var2
6	26	24
9	27	17

Birleştirme (Join) İşlemleri

In [3]:
```
m = np.random.randint(1,30, size=(5,3))
df1 = pd.DataFrame(m, columns=["var1","var2","var3"])
df1
```

Out[3]:

	var1	var2	var3
0	18	2	9
1	20	29	6
2	4	8	15
3	23	14	21
4	29	11	6

In [4]:
```
df2 = df1 + 99
```

In [5]:
```
df2
```

Out[5]:

	var1	var2	var3
0	117	101	108
1	119	128	105
2	103	107	114
3	122	113	120
4	128	110	105

In [6]:
```
pd.concat([df1,df2])
```

Out[6]:

	var1	var2	var3

	var1	var2	var3
0	18	2	9
1	20	29	6
2	4	8	15
3	23	14	21
4	29	11	6
0	117	101	108
1	119	128	105
2	103	107	114
3	122	113	120
4	128	110	105

Görüldüğü üzere `concat()` fonksiyonu ile **alt alta** birleştirmiş olduk. Fakat indexlere baktığımız zaman tekrardan 0'dan başladığını görüyoruz. Öncelikle bunun çözümü için fonksiyonun içeriğine bir bakalım.

In [7]:
```
pd.concat?
```

```
Signature:
pd.concat(
    objs: Union[Iterable[~FrameOrSeries], Mapping[Union[Hashable, NoneType], ~FrameOrSeries]],
    axis=0,
    join='outer',
    ignore_index: bool = False,
    keys=None,
    levels=None,
    names=None,
    verify_integrity: bool = False,
    sort: bool = False,
    copy: bool = True,
) -> Union[ForwardRef('DataFrame'), ForwardRef('Series')]
Docstring:
Concatenate pandas objects along a particular axis with optional set logic
along the other axes.

Can also add a layer of hierarchical indexing on the concatenation axis,
which may be useful if the labels are the same (or overlapping) on
the passed axis number.

Parameters
----------
objs : a sequence or mapping of Series or DataFrame objects
```

```
    If a mapping is passed, the sorted keys will be used as the `keys`
    argument, unless it is passed, in which case the values will be
    selected (see below). Any None objects will be dropped silently unless
    they are all None in which case a ValueError will be raised.
axis : {0/'index', 1/'columns'}, default 0
    The axis to concatenate along.
join : {'inner', 'outer'}, default 'outer'
    How to handle indexes on other axis (or axes).
ignore_index : bool, default False
    If True, do not use the index values along the concatenation axis. The
    resulting axis will be labeled 0, ..., n - 1. This is useful if you are
    concatenating objects where the concatenation axis does not have
    meaningful indexing information. Note the index values on the other
    axes are still respected in the join.
keys : sequence, default None
    If multiple levels passed, should contain tuples. Construct
    hierarchical index using the passed keys as the outermost level.
levels : list of sequences, default None
    Specific levels (unique values) to use for constructing a
    MultiIndex. Otherwise they will be inferred from the keys.
names : list, default None
    Names for the levels in the resulting hierarchical index.
verify_integrity : bool, default False
    Check whether the new concatenated axis contains duplicates. This can
    be very expensive relative to the actual data concatenation.
sort : bool, default False
    Sort non-concatenation axis if it is not already aligned when `join`
    is 'outer'.
    This has no effect when ``join='inner'``, which already preserves
    the order of the non-concatenation axis.

    .. versionadded:: 0.23.0
    .. versionchanged:: 1.0.0

       Changed to not sort by default.

copy : bool, default True
    If False, do not copy data unnecessarily.

Returns
-------
object, type of objs
    When concatenating all ``Series`` along the index (axis=0), a
    ``Series`` is returned. When ``objs`` contains at least one
    ``DataFrame``, a ``DataFrame`` is returned. When concatenating along
    the columns (axis=1), a ``DataFrame`` is returned.

See Also
--------
Series.append : Concatenate Series.
DataFrame.append : Concatenate DataFrames.
DataFrame.join : Join DataFrames using indexes.
```

```
DataFrame.merge : Merge DataFrames by indexes or columns.

Notes
-----
The keys, levels, and names arguments are all optional.

A walkthrough of how this method fits in with other tools for combining
pandas objects can be found `here
<https://pandas.pydata.org/pandas-docs/stable/user_guide/merging.html>`__.

Examples
--------
Combine two ``Series``.

>>> s1 = pd.Series(['a', 'b'])
>>> s2 = pd.Series(['c', 'd'])
>>> pd.concat([s1, s2])
0    a
1    b
0    c
1    d
dtype: object

Clear the existing index and reset it in the result
by setting the ``ignore_index`` option to ``True``.

>>> pd.concat([s1, s2], ignore_index=True)
0    a
1    b
2    c
3    d
dtype: object

Add a hierarchical index at the outermost level of
the data with the ``keys`` option.

>>> pd.concat([s1, s2], keys=['s1', 's2'])
s1  0    a
    1    b
s2  0    c
    1    d
dtype: object

Label the index keys you create with the ``names`` option.

>>> pd.concat([s1, s2], keys=['s1', 's2'],
...           names=['Series name', 'Row ID'])
Series name  Row ID
s1           0         a
             1         b
s2           0         c
             1         d
```

```
dtype: object

Combine ``DataFrame`` objects with identical columns.

>>> df1 = pd.DataFrame([['a', 1], ['b', 2]],
...                    columns=['letter', 'number'])
>>> df1
  letter  number
0      a       1
1      b       2
>>> df2 = pd.DataFrame([['c', 3], ['d', 4]],
...                    columns=['letter', 'number'])
>>> df2
  letter  number
0      c       3
1      d       4
>>> pd.concat([df1, df2])
  letter  number
0      a       1
1      b       2
0      c       3
1      d       4

Combine ``DataFrame`` objects with overlapping columns
and return everything. Columns outside the intersection will
be filled with ``NaN`` values.

>>> df3 = pd.DataFrame([['c', 3, 'cat'], ['d', 4, 'dog']],
...                    columns=['letter', 'number', 'animal'])
>>> df3
  letter  number animal
0      c       3    cat
1      d       4    dog
>>> pd.concat([df1, df3], sort=False)
  letter  number animal
0      a       1    NaN
1      b       2    NaN
0      c       3    cat
1      d       4    dog

Combine ``DataFrame`` objects with overlapping columns
and return only those that are shared by passing ``inner`` to
the ``join`` keyword argument.

>>> pd.concat([df1, df3], join="inner")
  letter  number
0      a       1
1      b       2
0      c       3
1      d       4

Combine ``DataFrame`` objects horizontally along the x axis by
```

```
passing in ``axis=1``.

>>> df4 = pd.DataFrame([['bird', 'polly'], ['monkey', 'george']],
...                    columns=['animal', 'name'])
>>> pd.concat([df1, df4], axis=1)
  letter  number  animal    name
0      a       1    bird   polly
1      b       2  monkey  george

Prevent the result from including duplicate index values with the
``verify_integrity`` option.

>>> df5 = pd.DataFrame([1], index=['a'])
>>> df5
   0
a  1
>>> df6 = pd.DataFrame([2], index=['a'])
>>> df6
   0
a  2
>>> pd.concat([df5, df6], verify_integrity=True)
Traceback (most recent call last):
    ...
ValueError: Indexes have overlapping values: ['a']
File:      c:\users\ertug\anaconda3\envs\tf\lib\site-packages\pandas\core\reshape\concat.py
Type:      function
```

`ignore index` adında bir argüman olduğunu görüyoruz ve ön tanımlı şekli **False** olarak belirtilmiş. **True** yaptığımızda:

In [8]:
```
pd.concat([df1,df2], ignore_index=True)
```

Out[8]:

	var1	var2	var3
0	18	2	9
1	20	29	6
2	4	8	15
3	23	14	21
4	29	11	6
5	117	101	108
6	119	128	105
7	103	107	114
8	122	113	120
9	128	110	105

Görüldüğü üzere sorun düzeldi. İşte İngilizce bilmenin faydaları :)

In [11]:
```
df1.columns
```

Out[11]:
```
Index(['var1', 'var2', 'var3'], dtype='object')
```

In [12]:
```
df2.columns
```

Out[12]:
```
Index(['var1', 'var2', 'var3'], dtype='object')
```

In [13]:
```
df2.columns = ["var1","var2","deg3"]
```

In [14]:
```
df2
```

Out[14]:

	var1	var2	deg3
0	117	101	108
1	119	128	105
2	103	107	114
3	122	113	120
4	128	110	105

In [15]:
```
df1
```

Out[15]:

	var1	var2	var3
0	18	2	9
1	20	29	6
2	4	8	15
3	23	14	21
4	29	11	6

df2'nin bir sütununun ismini değiştirdik. Bakalım `concat` fonksiyonu çalışacak mı?

In [16]:
```
pd.concat([df1,df2])
```

Out[16]:

	var1	var2	var3	deg3
0	18	2	9.0	NaN
1	20	29	6.0	NaN
2	4	8	15.0	NaN
3	23	14	21.0	NaN
4	29	11	6.0	NaN
0	117	101	NaN	108.0
1	119	128	NaN	105.0
2	103	107	NaN	114.0
3	122	113	NaN	120.0
4	128	110	NaN	105.0

Görüldüğü üzere istenmeyen bir şekilde sonuç döndürdü. Zaten normal şartlar altında değişken isimleri farklı olduğu için aynı bilgiler taşımamaktadır ve birleştirme işlemi mantıksızdır.

In [17]:
```
pd.concat([df1,df2], join = "inner")
```

Out[17]:

	var1	var2
0	18	2
1	20	29
2	4	8
3	23	14
4	29	11
0	117	101
1	119	128
2	103	107
3	122	113
4	128	110

Bu argümanı kullandığımızda değişken isimleri aynı olanları birleştirdi.

Farklı olan değişkenlerin **anlam olarak** aynı verileri tuttuklarını ve dolayısıyla tek bir değişken olarak birleştirmek istediğimizi farz edelim:

In [26]:
```
pd.concat([df1,df2], join_axes=[df1.columns])
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-26-5fbe08d14da2> in <module>
----> 1 pd.concat([df1,df2], join_axes=[df1.columns])

TypeError: concat() got an unexpected keyword argument 'join_axes'
```

Not: Kursta hoca bu şekilde kullanıyor fakat `join_axes` argümanı sürümden dolayı kaldırılmış. Bunun yerine aşağıdaki kod kullanılmalıdır:

In [44]:
```
pd.concat([df1,df2], ignore_index=True).reindex(columns=df1.columns)
```

Out[44]:

	var1	var2	var3
0	18	2	9.0
1	20	29	6.0
2	4	8	15.0
3	23	14	21.0
4	29	11	6.0
5	117	101	NaN
6	119	128	NaN
7	103	107	NaN
8	122	113	NaN
9	128	110	NaN

Görüldüğü üzere değişken ismini sabit tutup ona göre birleştirme yapmak istediğimizde bu şekilde kullanılır.

In [45]:
```
pd.concat([df1,df2], ignore_index=True).reindex(columns=df2.columns)
```

Out[45]:

	var1	var2	deg3
0	18	2	NaN
1	20	29	NaN

	var1	var2	deg3
2	4	8	NaN
3	23	14	NaN
4	29	11	NaN
5	117	101	108.0
6	119	128	105.0
7	103	107	114.0
8	122	113	120.0
9	128	110	105.0

Aynı durumu df2 için de yapabiliriz. Burada değişken isimleri farklı olduğundan diğer verileri eklememize izin vermiyor. (Muhtemelen) fonksiyonun argümanlarını değiştirerek bu sorunu ortadan kaldırabiliriz.

İleri Birleştirme İşlemleri

Birebir Birleştirme

In [47]:
```
df1 = pd.DataFrame({'calisanlar': ['Ali','Veli','Ayse','Fatma'],
                    'grup': ['Muhasebe','Muhendislik','Muhendislik','IK']})
df1
```

Out[47]:

	calisanlar	grup
0	Ali	Muhasebe
1	Veli	Muhendislik
2	Ayse	Muhendislik
3	Fatma	IK

In [48]:
```
df2 = pd.DataFrame({'calisanlar': ['Ayse','Ali','Veli','Fatma'],
                    'ilk_giris': [2010,2009,2004,2019]})
df2
```

Out[48]:

	calisanlar	ilk_giris
0	Ayse	2010

	calisanlar	ilk_giris
1	Ali	2009
2	Veli	2004
3	Fatma	2019

In [49]:
```
pd.merge(df1,df2)
```

Out[49]:

	calisanlar	grup	ilk_giris
0	Ali	Muhasebe	2009
1	Veli	Muhendislik	2004
2	Ayse	Muhendislik	2010
3	Fatma	IK	2019

Görüldüğü üzere ortak değişken **"calisanlar"** olduğu için ona göre birleştirdi. Aynı zamanda **"calisanlar"** değişkeninde isimler başka yerlerde olmasına rağmen birebir olacak şekilde birleştirme işlemini yaptı. Bunu bir argümana yazmak istersek de:

In [50]:
```
pd.merge(df1,df2, on="calisanlar")
```

Out[50]:

	calisanlar	grup	ilk_giris
0	Ali	Muhasebe	2009
1	Veli	Muhendislik	2004
2	Ayse	Muhendislik	2010
3	Fatma	IK	2019

şeklinde kullanabiliriz.

Çoktan Teke (Many To One)

In [51]:
```
df3 = pd.merge(df1, df2)
```

In [52]:
```
df3
```

Out[52]:

	calisanlar	grup	ilk_giris
0	Ali	Muhasebe	2009
1	Veli	Muhendislik	2004
2	Ayse	Muhendislik	2010
3	Fatma	IK	2019

In [53]:
```
df4 = pd.DataFrame({'grup': ['Muhasebe','Muhendislik','IK'],
                    'mudur': ['Caner','Mustafa','Berkcan']})
df4
```

Out[53]:

	grup	mudur
0	Muhasebe	Caner
1	Muhendislik	Mustafa
2	IK	Berkcan

In [54]:
```
pd.merge(df3,df4)
```

Out[54]:

	calisanlar	grup	ilk_giris	mudur
0	Ali	Muhasebe	2009	Caner
1	Veli	Muhendislik	2004	Mustafa
2	Ayse	Muhendislik	2010	Mustafa
3	Fatma	IK	2019	Berkcan

Görüldüğü üzere burada **"grup"** değişkeni ortak olduğu için ona göre birleştirme işlemini yaptı.

Çoktan Çoka (Many To Many)

In [55]:
```
df5 = pd.DataFrame({'grup': ['Muhasebe','Muhasebe','Muhendislik','Muhendislik','IK','IK'],
                    'yetenekler': ['matematik', 'excel', 'kodlama', 'linux', 'excel', 'yonetim']})
df5
```

Out[55]:

	grup	yetenekler
0	Muhasebe	matematik

	grup	yetenekler
1	Muhasebe	excel
2	Muhendislik	kodlama
3	Muhendislik	linux
4	IK	excel
5	IK	yonetim

In [56]:
```
df1
```

Out[56]:

	calisanlar	grup
0	Ali	Muhasebe
1	Veli	Muhendislik
2	Ayse	Muhendislik
3	Fatma	IK

In [57]:
```
pd.merge(df1, df5)
```

Out[57]:

	calisanlar	grup	yetenekler
0	Ali	Muhasebe	matematik
1	Ali	Muhasebe	excel
2	Veli	Muhendislik	kodlama
3	Veli	Muhendislik	linux
4	Ayse	Muhendislik	kodlama
5	Ayse	Muhendislik	linux
6	Fatma	IK	excel
7	Fatma	IK	yonetim

Görüldüğü üzere yine **"grup"** değişkenine göre birleştirme işlemini yaptı. Bu tablo görünüşte pek doğru bir tablo değildir. Analitik bir işlem yapılacağı zaman da aynı şekilde sıkıntı verecektir fakat şimdilik sorun değildir.

Gruplama ve Toplulaştırma İşlemleri

Gruplama ve Toplulaştırma (Grouping & Aggregation)

Basit toplulaştırma fonksiyonları:

- count()
- first()
- last()
- mean()
- median()
- min()
- max()
- std()
- var()
- sum()

Burada yapacağımız yalnızca **yapısal** olarak fonksiyonları incelemektir. **Keşifçi Veri Analizi** bölümünde ise bu sonuçların ne anlama geldiğini veri görselleştirme tekniklerinden de yardım alarak analitik olarak yorumlayacağız.

In [2]:
```
df = sns.load_dataset("planets")
```

https://github.com/mwaskom/seaborn-data linkinden seaborn içerisindeki tüm verisetlerini görebiliriz.

In [3]:
```
df
```

Out[3]:

	method	number	orbital_period	mass	distance	year
0	Radial Velocity	1	269.300000	7.10	77.40	2006
1	Radial Velocity	1	874.774000	2.21	56.95	2008
2	Radial Velocity	1	763.000000	2.60	19.84	2011
3	Radial Velocity	1	326.030000	19.40	110.62	2007
4	Radial Velocity	1	516.220000	10.50	119.47	2009
...	...	...	...	...	...	...
1030	Transit	1	3.941507	NaN	172.00	2006
1031	Transit	1	2.615864	NaN	148.00	2007

	method	number	orbital_period	mass	distance	year
1032	Transit	1	3.191524	NaN	174.00	2007
1033	Transit	1	4.125083	NaN	293.00	2008
1034	Transit	1	4.187757	NaN	260.00	2008

1035 rows × 6 columns

In [4]:
```
df.head()
```

Out[4]:

	method	number	orbital_period	mass	distance	year
0	Radial Velocity	1	269.300	7.10	77.40	2006
1	Radial Velocity	1	874.774	2.21	56.95	2008
2	Radial Velocity	1	763.000	2.60	19.84	2011
3	Radial Velocity	1	326.030	19.40	110.62	2007
4	Radial Velocity	1	516.220	10.50	119.47	2009

In [6]:
```
df.shape
```

Out[6]:
```
(1035, 6)
```

In [14]:
```
df.mean()
```

Out[14]:
```
number              1.785507
orbital_period    2002.917596
mass                 2.638161
distance           264.069282
year              2009.070531
dtype: float64
```

Görüldüğü üzere bütün ddeğişkenlerin ortalamasını aldı. Belli bir sütunun ortalamasını almak istersek:

In [17]:
```
df["mass"].mean()
```

Out[17]:
```
2.6381605847953216
```

veya

In [18]:
```
df.mass.mean()
```

Out[18]:
```
2.6381605847953216
```

şeklinde kullanabiliriz.

In [19]:
```
df["mass"].count()
```

Out[19]:
```
513
```

In [23]:
```
df["mass"].min()
```

Out[23]:
```
0.0036
```

In [24]:
```
df["mass"].max()
```

Out[24]:
```
25.0
```

In [25]:
```
df["mass"].sum()
```

Out[25]:
```
1353.37638
```

In [26]:
```
df["mass"].std()
```

Out[26]:
```
3.8186166509616046
```

In [27]:
```
df["mass"].var()
```

Out[27]:
```
14.58183312700122
```

In [28]:
```
df.describe()
```

Out[28]:

	number	orbital_period	mass	distance	year
count	1035.000000	992.000000	513.000000	808.000000	1035.000000
mean	1.785507	2002.917596	2.638161	264.069282	2009.070531

	number	orbital_period	mass	distance	year
std	1.240976	26014.728304	3.818617	733.116493	3.972567
min	1.000000	0.090706	0.003600	1.350000	1989.000000
25%	1.000000	5.442540	0.229000	32.560000	2007.000000
50%	1.000000	39.979500	1.260000	55.250000	2010.000000
75%	2.000000	526.005000	3.040000	178.500000	2012.000000
max	7.000000	730000.000000	25.000000	8500.000000	2014.000000

In [29]:
```
df.describe().T #transpozunu alır.
```

Out[29]:

	count	mean	std	min	25%	50%	75%	max
number	1035.0	1.785507	1.240976	1.000000	1.00000	1.0000	2.000	7.0
orbital_period	992.0	2002.917596	26014.728304	0.090706	5.44254	39.9795	526.005	730000.0
mass	513.0	2.638161	3.818617	0.003600	0.22900	1.2600	3.040	25.0
distance	808.0	264.069282	733.116493	1.350000	32.56000	55.2500	178.500	8500.0
year	1035.0	2009.070531	3.972567	1989.000000	2007.00000	2010.0000	2012.000	2014.0

In [31]:
```
df.dropna().describe().T

#Eksik değerler atıldı.
```

Out[31]:

	count	mean	std	min	25%	50%	75%	max
number	498.0	1.734940	1.175720	1.0000	1.00000	1.000	2.0000	6.0
orbital_period	498.0	835.778671	1469.128259	1.3283	38.27225	357.000	999.6000	17337.5
mass	498.0	2.509320	3.636274	0.0036	0.21250	1.245	2.8675	25.0
distance	498.0	52.068213	46.596041	1.3500	24.49750	39.940	59.3325	354.0
year	498.0	2007.377510	4.167284	1989.0000	2005.00000	2009.000	2011.0000	2014.0

Gruplama İşlemleri

Gruplama işlemi, verisetinde yer alan kategorik değişkenlerinin gruplarının yakalanması ve bu grupların özelinde bazı işlemler yapılması demektir.

In [35]:
```
df = pd.DataFrame({'gruplar': ['A','B','C','A','B','C'],
                   'veri': [10,11,52,23,43,55]})
df
```

Out[35]:

	gruplar	veri
0	A	10
1	B	11
2	C	52
3	A	23
4	B	43
5	C	55

In [36]:
```
df.groupby("gruplar")
```

Out[36]:
```
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x0000020166E8E280>
```

Şu an gruplama işlemini yaptı fakat tek başına bir iş yapamadığından **"aggregation(toplulaştırma)"** fonksiyonlarına ihtiyaç vardır. Fonksiyonlarda herhangi biri kullanıldığında bize bir sonuç verecektir.

Yani **gruplar** değişkenini aldı ve grupladı. Fakat şu an diyor ki:
*Şu an **gruplar** değişkenini grupladım. Fakat ben `groupby` fonksiyonuyum. Ben tek başına çalışamam. Bana **aggregation(toplulaştırma)** fonksiyonu lazım. Çünkü bu grupdığım değişkene ne yapacağımı henüz bilmiyorum.*

Bunu demesine karşılık olarak hadi birkaç tane toplulaştırma fonksiyonu kullanalım:

In [37]:
```
df.groupby("gruplar").mean()
```

Out[37]:

	veri
gruplar	
A	16.5
B	27.0
C	53.5

In [38]:
```
df.groupby("gruplar").sum()
```

Out[38]:

gruplar	veri
A	33
B	54
C	107

Şimdi seaborn kütüphanesindeki **"planets"** veriseti üzerinden bu işlemleri yapalım:

In [39]:
```
df = sns.load_dataset("planets")
df.head()
```

Out[39]:

	method	number	orbital_period	mass	distance	year
0	Radial Velocity	1	269.300	7.10	77.40	2006
1	Radial Velocity	1	874.774	2.21	56.95	2008
2	Radial Velocity	1	763.000	2.60	19.84	2011
3	Radial Velocity	1	326.030	19.40	110.62	2007
4	Radial Velocity	1	516.220	10.50	119.47	2009

In [40]:
```
df.groupby("method")
```

Out[40]:
```
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x0000020166925130>
```

"method" değişkenini grupladık. Şimdi başka bir değişkeni seçip ona toplulaştırma fonksiyonlarını kullanalım:

In [41]:
```
df.groupby("method")["orbital_period"]
```

Out[41]:
```
<pandas.core.groupby.generic.SeriesGroupBy object at 0x00000201669258E0>
```

In [43]:
```
df.groupby("method")["orbital_period"].mean()
```

Out[43]:
```
method
Astrometry                         631.180000
Eclipse Timing Variations         4751.644444
Imaging                         118247.737500
```

```
Microlensing                         3153.571429
Orbital Brightness Modulation           0.709307
Pulsar Timing                        7343.021201
Pulsation Timing Variations          1170.000000
Radial Velocity                       823.354680
Transit                                21.102073
Transit Timing Variations              79.783500
Name: orbital_period, dtype: float64
```

In [44]:
```
df.groupby("method")["mass"].mean()
```

Out[44]:
```
method
Astrometry                           NaN
Eclipse Timing Variations       5.125000
Imaging                              NaN
Microlensing                         NaN
Orbital Brightness Modulation        NaN
Pulsar Timing                        NaN
Pulsation Timing Variations          NaN
Radial Velocity                 2.630699
Transit                         1.470000
Transit Timing Variations            NaN
Name: mass, dtype: float64
```

In [45]:
```
df.groupby("method")["orbital_period"].describe()
```

Out[45]:

method	count	mean	std	min	25%	50%	75%	max
Astrometry	2.0	631.180000	544.217663	246.360000	438.770000	631.180000	823.590000	1016.000000
Eclipse Timing Variations	9.0	4751.644444	2499.130945	1916.250000	2900.000000	4343.500000	5767.000000	10220.000000
Imaging	12.0	118247.737500	213978.177277	4639.150000	8343.900000	27500.000000	94250.000000	730000.000000
Microlensing	7.0	3153.571429	1113.166333	1825.000000	2375.000000	3300.000000	3550.000000	5100.000000
Orbital Brightness Modulation	3.0	0.709307	0.725493	0.240104	0.291496	0.342887	0.943908	1.544929
Pulsar Timing	5.0	7343.021201	16313.265573	0.090706	25.262000	66.541900	98.211400	36525.000000
Pulsation Timing Variations	1.0	1170.000000	NaN	1170.000000	1170.000000	1170.000000	1170.000000	1170.000000
Radial Velocity	553.0	823.354680	1454.926210	0.736540	38.021000	360.200000	982.000000	17337.500000
Transit	397.0	21.102073	46.185893	0.355000	3.160630	5.714932	16.145700	331.600590
Transit Timing Variations	3.0	79.783500	71.599884	22.339500	39.675250	57.011000	108.505500	160.000000

Bütün değişkenlerin sonuçlarını görmek istersek grupladıktan sonra hemen fonksiyonları kullanmamız yeterlidir.

In [46]:
```
df.groupby("method").mean()
```

Out[46]:

method	number	orbital_period	mass	distance	year
Astrometry	1.000000	631.180000	NaN	17.875000	2011.500000
Eclipse Timing Variations	1.666667	4751.644444	5.125000	315.360000	2010.000000
Imaging	1.315789	118247.737500	NaN	67.715937	2009.131579
Microlensing	1.173913	3153.571429	NaN	4144.000000	2009.782609
Orbital Brightness Modulation	1.666667	0.709307	NaN	1180.000000	2011.666667
Pulsar Timing	2.200000	7343.021201	NaN	1200.000000	1998.400000
Pulsation Timing Variations	1.000000	1170.000000	NaN	NaN	2007.000000
Radial Velocity	1.721519	823.354680	2.630699	51.600208	2007.518987
Transit	1.954660	21.102073	1.470000	599.298080	2011.236776
Transit Timing Variations	2.250000	79.783500	NaN	1104.333333	2012.500000

Aggregate

In [6]:
```
df = pd.DataFrame({'gruplar': ['A','B','C','A','B','C'],
                   'degisken1': [10,23,33,22,11,99],
                   'degisken2': [100,253,333,262,111,969]})
df
```

Out[6]:

	gruplar	degisken1	degisken2
0	A	10	100
1	B	23	253
2	C	33	333
3	A	22	262
4	B	11	111
5	C	99	969

In [3]:
```
df.groupby("gruplar")
```

Out[3]:
```
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x0000019B95FAA160>
```

In [4]:
```
df.groupby("gruplar").mean()
```

Out[4]:

gruplar	degisken1	degisken2
A	16	181
B	17	182
C	66	651

In [10]:
```
df.groupby("gruplar").aggregate(["min",np.median,max])

#ya da

df.groupby("gruplar").aggregate(["min","median","max"])
```

Out[10]:

	degisken1			degisken2		
	min	median	max	min	median	max
gruplar						
A	10	16	22	100	181	262
B	11	17	23	111	182	253
C	33	66	99	333	651	969

Görüldüğü üzere bizim belirlediğimiz istatistikleri getirmesini istersek **aggregate** fonksiyonunu kullanıyoruz.

Burada içerisine yazdığımız istatistikleri bütün değişkenler için aynı olacak şekilde uyguladı. Şimdi bunu bütün değişkenler için değil de bazı değişkenler için ayrı istatistikler göstermesini isteyelim.

In [11]:
```
df.groupby("gruplar").aggregate({"degisken1": "min",
                                 "degisken2": "max"})
```

Out[11]:

gruplar	degisken1	degisken2

gruplar	degisken1	degisken2
gruplar		
A	10	262
B	11	253
C	33	969

Görüldüğü üzere **degisken1** için **min** istatistiğini; **degisken2** için ise **max** istatistiğini gösterdik. Bunu çoğaltabiliriz.

In [13]:
```
df.groupby("gruplar").aggregate({"degisken1": "min",
                                 "degisken2": ["min","median"]})
```

Out[13]:

	degisken1	degisken2	
	min	min	median
gruplar			
A	10	100	181
B	11	111	182
C	33	333	651

In [14]:
```
df.groupby("gruplar").aggregate({"degisken1": ["min","mean","max"],
                                 "degisken2": ["min","median","max"]})
```

Out[14]:

	degisken1			degisken2		
	min	mean	max	min	median	max
gruplar						
A	10	16	22	100	181	262
B	11	17	23	111	182	253
C	33	66	99	333	651	969

Filter

Bu fonksiyon DataFrame üzerine ileri koşul işlemleri yapabilmemizi sağlıyor. Klasik koşul işlemlerinin ihtiyaçlarımızı karşılamadığı durumlarda kendimiz bir koşul

yazma istediğimizde **filter** fonksiyonunu kullanıyoruz.

In [1]:
```
df = pd.DataFrame({'gruplar': ['A','B','C','A','B','C'],
                   'degisken1': [10,23,33,22,11,99],
                   'degisken2': [100,253,333,262,111,969]})
df
```

Out[1]:

	gruplar	degisken1	degisken2
0	A	10	100
1	B	23	253
2	C	33	333
3	A	22	262
4	B	11	111
5	C	99	969

Koşulumuzu bir fonksiyon şeklinde yazalım:

In [2]:
```
def filter_func(x):
    return x["degisken1"].std() > 9
```

In [4]:
```
df.groupby("gruplar").std()
```

Out[4]:

	degisken1	degisken2
gruplar		
A	8.485281	114.551299
B	8.485281	100.409163
C	46.669048	449.719913

In [23]:
```
filter_func
```

Out[23]:
```
<function __main__.filter_func(x)>
```

In [22]:
```
filter_func(df)
```

Out[22]:
```
True
```

In [17]:
```
df.groupby("gruplar").filter(filter_func)
```

Out[17]:

	gruplar	degisken1	degisken2
2	C	33	333
5	C	99	969

Görüldüğü üzere **filter** fonksiyonuna kendi koşulumuzu yazdığımız fonksiyonu parantez içerisine yazdık ve koşula göre filtreledi.

Bu **filter** fonksiyonu, **True** veya **False** döndüren başka bir fonksiyon alması gerekiyor. `filter_func(df)` yazdığımızda bize **True** sonucunu döndüğü için fonksiyon sorunsuz çalışmaktadır ve istediğimiz sonucu döndürmektedir.

`df.groupby("gruplar").filter(filter_func)` şeklinde yazdık çünkü bu şekilde sıkıntısız çalışmaktadır ve bizden böyle istemektedir.
`df.groupby("gruplar").filter(filter_func(df))` yazıldığında hata vermektedir. Yani **filter** fonksiyonu direkt fonksiyonunun **kendisini** istemektedir.

Transform

Değişkenleri başka bir sayı aralığına dönüştürme işlemidir. Vektörel çalışan bir fonksiyondur. Değişkenler üzerinde Pandas'ta bulunmayan dönüştürme işlemlerini **filter** fonksiyonunda olduğu gibi kendimiz tanımlayabiliriz.

In [7]:
```
df = pd.DataFrame({'gruplar': ['A','B','C','A','B','C'],
                   'degisken1': [10,23,33,22,11,99],
                   'degisken2': [100,253,333,262,111,969]})
df
```

Out[7]:

	gruplar	degisken1	degisken2
0	A	10	100
1	B	23	253
2	C	33	333
3	A	22	262
4	B	11	111
5	C	99	969

In [2]:
```
df["degisken1"] * 9
```

Out[2]:
```
0     90
1    207
2    297
3    198
4     99
5    891
Name: degisken1, dtype: int64
```

Öncelikle kategorik değişkenden kurtulalım:

In [4]:
```
df_a = df.iloc[: , 1:3]
```

Sonra yapmak istediğimiz işlemi yazalım:

In [5]:
```
df_a.transform(lambda x: x-x.mean())
```

Out[5]:

	degisken1	degisken2
0	-23.0	-238.0
1	-10.0	-85.0
2	0.0	-5.0
3	-11.0	-76.0
4	-22.0	-227.0
5	66.0	631.0

Görüldüğü üzere tek seferlik fonksiyon oluşturmak için **lambda** ifadesini kullanarak her bir değerden ortalamayı çıkardık.
Başka bir işlem yapalım:

In [6]:
```
df_a.transform(lambda x: (x-x.mean()) / x.std())
```

Out[6]:

	degisken1	degisken2
0	-0.687871	-0.738461
1	-0.299074	-0.263736
2	0.000000	-0.015514
3	-0.328982	-0.235811
4	-0.657963	-0.704331

	degisken1	degisken2
5	1.973890	1.957853

Bu işleme standart normal dönüşüm denir.

Apply

Bu fonksiyon tıpkı tranform ve filter fonksiyonu gibi DataFrame'in değişkenlerinin üzerinde gezinme yeteneği olan ve aggregation yani toplulaştırma amacıyla kullanılabilecek olan bir fonksiyondur.

In [1]:
```
df = pd.DataFrame({'degisken1': [10,23,33,22,11,99],
                   'degisken2': [100,253,333,262,111,969]})
df
```

Out[1]:

	degisken1	degisken2
0	10	100
1	23	253
2	33	333
3	22	262
4	11	111
5	99	969

In [10]:
```
df.apply(np.sum)
```

Out[10]:
```
degisken1     198
degisken2    2028
dtype: int64
```

In [11]:
```
df.apply(np.mean)
```

Out[11]:
```
degisken1     33.0
degisken2    338.0
dtype: float64
```

Görüldüğü üzere fonksiyonun içerisine hazır fonksiyonlar kullanıldığı gibi kendimiz fonksiyon oluşturup onu da kullanabiliriz.

In [12]:
```
df = pd.DataFrame({'gruplar': ['A','B','C','A','B','C'],
                   'degisken1': [10,23,33,22,11,99],
```

```
                   'degisken2': [100,253,333,262,111,969]})
df
```

Out[12]:

	gruplar	degisken1	degisken2
0	A	10	100
1	B	23	253
2	C	33	333
3	A	22	262
4	B	11	111
5	C	99	969

In [13]:
```
df.groupby("gruplar").apply(np.sum)
```

Out[13]:

	gruplar	degisken1	degisken2
gruplar			
A	AA	32	362
B	BB	34	364
C	CC	132	1302

In [14]:
```
df.groupby("gruplar").apply(np.mean)
```

Out[14]:

	degisken1	degisken2
gruplar		
A	16.0	181.0
B	17.0	182.0
C	66.0	651.0

Pivot Tablolar

Verisetleri üzerinde bazı sütun işlemleri yaparak verisetini amaca uygun hale getirmek için kullanılan yapılardır. `groupby()` fonksiyonu ile karıştırılır. `groupby()` fonksiyonun çok boyutlu hali(birden fazla gruplu hali) olarak düşünülebilir.

In [7]:
```
titanic = sns.load_dataset('titanic')
titanic.head()
```

Out[7]:

	survived	pclass	sex	age	sibsp	parch	fare	embarked	class	who	adult_male	deck	embark_town	alive	alone
0	0	3	male	22.0	1	0	7.2500	S	Third	man	True	NaN	Southampton	no	False
1	1	1	female	38.0	1	0	71.2833	C	First	woman	False	C	Cherbourg	yes	False
2	1	3	female	26.0	0	0	7.9250	S	Third	woman	False	NaN	Southampton	yes	True
3	1	1	female	35.0	1	0	53.1000	S	First	woman	False	C	Southampton	yes	False
4	0	3	male	35.0	0	0	8.0500	S	Third	man	True	NaN	Southampton	no	True

In [19]:
```
titanic.groupby("sex")["survived"].mean()
```

Out[19]:
```
sex
female    0.742038
male      0.188908
Name: survived, dtype: float64
```

İlkel bir pivot işlemi yaptık. DataFrame şeklinde görülmesi için çift parantez koymalıyız.

In [23]:
```
titanic.groupby("sex")[["survived"]].mean()
```

Out[23]:

	survived
sex	
female	0.742038
male	0.188908

In [ ]:
```

```

In [25]:
```
titanic.groupby(["sex","class"])[["survived"]].aggregate("mean").unstack()
```

Out[25]:

	survived		
class	**First**	**Second**	**Third**
sex			

	survived		
class	First	Second	Third
sex			
female	0.968085	0.921053	0.500000
male	0.368852	0.157407	0.135447

Burada hem cinsiyet hem de "class" değişkenine göre gruplayarak hayatta kalıp kalmadıklarını inceledik.

`unstack()` fonksiyonu diğer grubun(class) yatay şekilde yazılması için kullanıldı. Yazılmadığı takdirde çıktı aşağıdaki gibi görülecektir:

In [30]:
```
titanic.groupby(["sex","class"])[["survived"]].aggregate("mean")
```

Out[30]:

		survived
sex	class	
female	First	0.968085
	Second	0.921053
	Third	0.500000
male	First	0.368852
	Second	0.157407
	Third	0.135447

In [29]:
```
titanic["class"].unique()
```

Out[29]:
```
['Third', 'First', 'Second']
Categories (3, object): ['Third', 'First', 'Second']
```

Bunu yapmanın daha kolay bir yolu var. Niye uğraştırıyorsun bizi hocam? :)

In [33]:
```
titanic.pivot_table("survived", index="sex", columns="class")
```

Out[33]:

	class	First	Second	Third
	sex			
female		0.968085	0.921053	0.500000

class	First	Second	Third
sex			
male	0.368852	0.157407	0.135447

Ya da aşağıdaki gibi de kullanılabilir:

In [34]:
```
titanic.pivot_table("survived", index=["sex","class"])
```

Out[34]:

		survived
sex	class	
female	First	0.968085
	Second	0.921053
	Third	0.500000
male	First	0.368852
	Second	0.157407
	Third	0.135447

Şimdi daha karmaşık bir pivot table oluşturalım. **age** değişkenini 2 grup halinde kategorik değişkene dönüştürelim, sonra pivot tabloya boyut(index) olarak ekleyelim:

In [39]:
```
titanic.age.head()
```

Out[39]:
```
0    22.0
1    38.0
2    26.0
3    35.0
4    35.0
Name: age, dtype: float64
```

Bölme işlemi için Pandas'ın `cut()` fonksiyonunu kullanacağız:

In [41]:
```
age = pd.cut(titanic["age"], [0, 18, 90])
age.head(10)
```

Out[41]:
```
0    (18.0, 90.0]
1    (18.0, 90.0]
2    (18.0, 90.0]
```

```
3    (18.0, 90.0]
4    (18.0, 90.0]
5             NaN
6    (18.0, 90.0]
7     (0.0, 18.0]
8    (18.0, 90.0]
9     (0.0, 18.0]
Name: age, dtype: category
Categories (2, interval[int64]): [(0, 18] < (18, 90]]
```

In [42]:

```
titanic.pivot_table("survived", ["sex",age], "class")
```

Out[42]:

	class	First	Second	Third
sex	**age**			
female	**(0, 18]**	0.909091	1.000000	0.511628
	(18, 90]	0.972973	0.900000	0.423729
male	**(0, 18]**	0.800000	0.600000	0.215686
	(18, 90]	0.375000	0.071429	0.133663

Görüldüğü üzere **class** değişkenini kategorilere bölerek pivot table'a index olarak eklemiş olduk.

Dış Kaynaklı Veri Okuma

.csv Okuma

In [50]:

```
pd.read_csv("reading_data/ornekcsv.csv", sep=";")

#Dosya başka bir klasörde olduğunda dosya yolunu girmemiz gerekiyor.
#Dosyalar ";" ile ayrıldığı için "sep" argümanına ";" koyuyoruz.
```

Out[50]:

	a	b	c
0	78	12	1.0
1	78	12	2.0
2	78	324	3.0
3	7	2	4.0
4	88	23	5.0
5	6	2	NaN

	a	b	c
6	56	11	6.0
7	7	12	7.0
8	56	21	7.0
9	346	2	8.0
10	5	1	8.0
11	456	21	8.0
12	3	12	88.0

.txt Okuma

In [51]:
```
pd.read_csv("reading_data/duz_metin.txt")
```

Out[51]:

	1 2
0	2 2
1	3 2
2	4 2
3	5 2
4	6 2
5	7 2
6	8 2
7	9 2
8	10 2

Fonksiyonun ön tanımlı "sep" argümanı "," olmasına rağmen boşluk ile ayrılan dosyayı da okuyabildi. Yani veriler boşluk ile ayrılsa bile sorunsuz bir şekilde okuyabiliyor.

.xlsx Okuma

In [52]:
```
pd.read_excel("reading_data/ornekx.xlsx")
```

Out[52]:

	a	b	c

	a	b	c
0	78	12	1.0
1	78	12	2.0
2	78	324	3.0
3	7	2	4.0
4	88	23	5.0
5	6	2	NaN
6	56	11	6.0
7	7	12	7.0
8	56	21	7.0
9	346	2	8.0
10	5	1	8.0
11	456	21	8.0
12	3	12	88.0

In [53]:
```
df = pd.read_excel("reading_data/ornekx.xlsx")
```

In [54]:
```
type(df)
```

Out[54]:
```
pandas.core.frame.DataFrame
```

In [55]:
```
df.head()
```

Out[55]:

	a	b	c
0	78	12	1.0
1	78	12	2.0
2	78	324	3.0
3	7	2	4.0
4	88	23	5.0

In [56]:
```
df.columns
```

Out[56]:
```
Index(['a', 'b', 'c'], dtype='object')
```

In [57]:
```
df.columns = ("A","B","C")
```

In [58]:
```
df
```

Out[58]:

	A	B	C
0	78	12	1.0
1	78	12	2.0
2	78	324	3.0
3	7	2	4.0
4	88	23	5.0
5	6	2	NaN
6	56	11	6.0
7	7	12	7.0
8	56	21	7.0
9	346	2	8.0
10	5	1	8.0
11	456	21	8.0
12	3	12	88.0

Github'dan hepsini değil de sadece tek bir verisetini indirmek için:

- İlgili verisetine tıklıyoruz.
- Sağ üstte "Raw" seçeneğine basıyoruz.
- Ekrana gelen "csv" formatındaki veriyi kopyalıyoruz.
- Jupyter Lab'da "+" işaretine tıklayıp "Other" bölümünden "Text"i seçiyoruz.
- Verileri yapıştırıp ismini değiştirerek kaydediyoruz.

In [60]:

```
tips = pd.read_csv("reading_data/data.txt")
tips.head()
```

Out[60]:

	total_bill	tip	sex	smoker	day	time	size
0	16.99	1.01	Female	No	Sun	Dinner	2
1	10.34	1.66	Male	No	Sun	Dinner	3
2	21.01	3.50	Male	No	Sun	Dinner	3
3	23.68	3.31	Male	No	Sun	Dinner	2
4	24.59	3.61	Female	No	Sun	Dinner	4

In [66]:

```
df[(df.var1 > 15)][[ "var1","var2"]]
```

Out[66]:

	var1	var2
2	28	28
4	21	25
6	25	6

In [70]:

```
m = np.arange(1,7).reshape((3,2))
pd.DataFrame(m, columns = ["var1","var2"])
```

Out[70]:

	var1	var2
0	1	2
1	3	4
2	5	6

In [71]:

```
df.var1.dtype
```

Out[71]:

```
dtype('int32')
```

In [ ]: