FIZ220_EST_UygulamaNotlari_05_NumPy_Dizileri_II

April 19, 2020

1 Uygulama Notları: 5

1.1 FİZ220 - Bilgisayar Programlama II | 16/04/2020

Matrisler II (NumPy Dizi (*numpy.ndarray*) Nesneleri)

- Matrislerin Özellikleri
  - size
  - shape
  - reshape
  - ndim
  - dtype
- Tipik Matrisler
  - zeros
  - ones
  - fill ile matrisi doldurmak
  - birim matris
  - rastgele matrisler
  - arange ile aralıklar
  - linspace
- Matris işlemleri
  - Dört temel işlem (eleman bazında)
  - Skaler ve Vektörel Çarpımlar
    - Skaler Çarpım
    - Vektörel Çarpım
  - Matrisin transpozesi, tersi ve determinantı
    - Transpozesi
    - Tersi
    - 'Tersimsisi'
    - Determinantı

Dr. Emre S. Taşcı, emre.tasci@hacettepe.edu.tr
Fizik Mühendisliği Bölümü
Hacettepe Üniversitesi

1.2 Matrislerin Özellikleri

Sıklıkla, elimize geçirdiğimiz bir NumPy dizisinin özelliklerini bilmek isteriz: boyu nedir, boyutu nedir, kimlerdendir, içinde sevdiceğimiz var mıdır vs. Bu bölümde temel bilgi komut ve metotlarını işleyeceğiz.

O halde, bir tane 2 boyutlu (4×5) matris tanımlayıp, işimize bakalım:

[1]:
```
import numpy as np
mat1 = np.
 ↪array([[0,1,2,3,4],[10,11,12,13,14],[20,21,22,23,24],[30,31,32,33,34]])
print(mat1)
```

```
[[ 0  1  2  3  4]
 [10 11 12 13 14]
 [20 21 22 23 24]
 [30 31 32 33 34]]
```

Fark ettiyseniz, değerler aynı zamanda satır ve sütun indislerini veriyor (ilk satırın (0 satırı) misal 2 indisli elemanını 02 yazamadığımdan, 2 ile yetiniyoruz ama artık *o kadar kusur kadı kudu* (©Yiğit Özgür 8).

1.2.1 size: Matrisin eleman sayısı

`size` metodu bize matrisin toplam eleman sayısını verir. Örneğimizdeki matrisimizin 20 elemanı var, o halde, "size" diye sorunce, "20" demesini bekliyoruz, bakalım:

[2]:
```
print("Matrisimizin eleman sayısı: ",mat1.size)
```

```
Matrisimizin eleman sayısı:  20
```

1.3 shape: Matrisimizin boyutları *(90-60-90)*

`shape` metodu matrisimizin "şeklini" yani boyutlarını bir demet (*tuple*) olarak döndürür:

[3]:
```
print("Matrisimizin boyutları: ",mat1.shape)
print("Matrisimizin ",len(mat1.shape)," adet boyutu var.")
print("Asıl (0 - satırlar) eksendeki 'kat' sayısı: ",mat1.shape[0])
print("İkincil (1 - sütunlar) eksendeki 'daire sayısı: ",mat1.shape[1])
```

```
Matrisimizin boyutları:  (4, 5)
Matrisimizin  2  adet boyutu var.
Asıl (0 - satırlar) eksendeki 'kat' sayısı:  4
İkincil (1 - sütunlar) eksendeki 'daire sayısı:  5
```

Burada dikkatinizi çektiyse, boyutların döndürüldüğü mat1.shape demetinin boyunu bulurken `size` metodunu değil, `len` komutunu kullandık. Bunun sebebi, `size`ın bir ndarray metodu olup, demetlere uygulanabilir olmaması. `len` komutu epey evrensel bir komut olup, hemen her değişkene sorulabilir. O halde niye `len` dururken, matrislerde `size` ile uğraşıyoruz? Hemen bakalım:

[4]:
```
print(len(mat1))
```

4

Gördüğünüz üzere, sadece ilk eksenin (satırların) sayısını veriyor, gayet de mantıklı zira Python bunu -teknik olarak- her birinde 5 elemanlı 1 boyutlu dizi olan, 4 elemanlı bir derleme olarak görüyor (matrisi nasıl tanımladığımızı hatırlayın: [[1. eleman: 1. dizi], [2. eleman: 2. dizi], …]). Bu yüzden, n-boyutlu matrislerin eleman sayısını `len` ile değil, `size` ile öğreniyoruz (bu konu sanırım bölüm sonlarında olan "bunları bilmeseniz de olur ama işte…" kısmına daha uygundu!.. 8)

(Casablanca, "Play it -again- Sam" (tamam, Bogart hiç *again* lafını söylemiyor aslında ama olsun))

1.4 reshape: yeniden şekillendir, Sam..

Matrisin şekli çok önemlidir zira o şekli yeniden ("re-") biçimlendirebiliriz ("shape") => `reshape`!

Elimizdeki matris (4x5)'ti, 20 elemanı vardı. Elemanları düzgünce dağıtabileceğimiz (yani bütün boyutlarının çarpımı 20 olduğu sürece) her boyuta çıkabiliriz. Çarpımı 20 olan sayılardan bazılarını sıralayalım:

- 1x20
- 2x10
- 2x2x5
- 20x1

Gördüğünüz üzere, yukarıdakilerden 1. ve 4. örnekler bir boyutlu matrisler (ilki satır, ikincisi sütun vektörü); 2. örnek 2 satırlı, 10 sütunlu iki boyutlu bir matris; 3. ilginç çünkü 3 boyut var. Tek tek deneyelim:

[5]:
```
print(mat1.reshape(1,20))
```

```
[[ 0  1  2  3  4 10 11 12 13 14 20 21 22 23 24 30 31 32 33 34]]
```

[6]:
```
print(mat1.reshape(2,10))
```

```
[[ 0  1  2  3  4 10 11 12 13 14]
 [20 21 22 23 24 30 31 32 33 34]]
```

[7]:
```
print(mat1.reshape(2,2,5))
print(mat1.reshape(2,2,5)[1,1,3])
```

```
[[[ 0  1  2  3  4]
  [10 11 12 13 14]]

 [[20 21 22 23 24]
  [30 31 32 33 34]]]
33
```

[8]:
```
print(mat1.reshape(20,1))
```

```
[[ 0]
 [ 1]
 [ 2]
 [ 3]
 [ 4]
 [10]
 [11]
 [12]
 [13]
 [14]
 [20]
 [21]
 [22]
 [23]
 [24]
 [30]
 [31]
 [32]
 [33]
 [34]]
```

(2x2x5) üç boyutlu matrisimizde [1,1,3] indisli elemanı istediğimizde, önce 1. eksende ilerleyip, 1 indisli diziye gittik:

```
[[20 21 22 23 24]   [30 31 32 33 34]]
```

dizisi. Sonra bu dizinin 1 indisli dizisine gittik:

```
[30 31 32 33 34]
```

dizisi. Sonra da, bu dizinin 3 indisli elemanına (yani 4. elemanına) gittik: 33

Bu vesileyle, çaktırmadan, 3 boyutlu bir diziyi nasıl oluşturabileceğimize dair bir yolu da keşfetmiş olduk: `reshape` metodu ile.

1.5 ndim: kısa yoldan kaç boyutlu bir matrisimiz olduğuna dair...

Elimizdeki matrisimizin kaç boyutlu olduğunu şeklinin kaç elemanı olduğunu sorarak öğrenebilmiştik:

[9]:
```
print("Matrisimizin ",len(mat1.shape)," adet boyutu var.")
```

```
Matrisimizin  2  adet boyutu var.
```

Bunu tek bir metotla da yapmak mümkün, bu iş için `ndim` metotu yardımımıza koşuyor:

[10]:
```
print("Matrisimizin ",mat1.ndim," adet boyutu var.")
```

```
Matrisimizin  2  adet boyutu var.
```

1.6 dtype: elemanlarının cinsi

Aşağıdaki üç diziye bir bakın – sizce aralarında çok fark var mı? 1. `dizi_1 = np.array([0,1,2])` 2. `dizi_2 = np.array([0,1,2.0])` 3. `dizi_3 = np.array([0,1.0,"iki"])`

...

Herhalde, 3.yü listeye koymasaydık, "fark yok" cevabı normal karşılanacaktı ama 3. işi bozduğundan, bu sefer 1. ile 2. arasında da bir bit yeniği çıkmasını bekliyoruz. Python'a verelim, bakalım o ne diyecek:

[11]:
```
dizi_1 = np.array([0,1,2])
dizi_2 = np.array([0,1,2.0])
dizi_3 = np.array([0,1.0,"iki"])

print("dizi_1:\n",dizi_1,"\n")
print("dizi_2:\n",dizi_2,"\n")
print("dizi_3:\n",dizi_3,"\n")
```

```
dizi_1:
 [0 1 2]

dizi_2:
 [0. 1. 2.]

dizi_3:
 ['0' '1.0' 'iki']
```

Hemen fark edenler kendilerine bir 10 puan yazsınlar bakalım! Detaylıca incelediğimizde, ilkinde sayılarımızın ondalık noktalarının olmadığını, ikincisinde diziyi girerken sadece 2'yi ondalıklı

girdiğimiz halde (o da hani "2.3" gibi değil, bildiğiniz "2.0" masumane şekliyleydi!) 0 ve 1'in kuyruklarına da ondalık noktasının takılmış olduğunu, üçüncü dizide ise hepsinin (0'ın ve 1.0'ın bile!) bir kelimeymişçesine kesme (') işaretleri arasına alınmış olduğunu görüyoruz. Bunun sebebi şu:

NumPy dizileri sadece tek bir eleman cinsini tutarlar. İlk dizimizde bütün elemanlarımız tam sayı, o yüzden tam sayı olarak tutulabiliyorlar; ikinci dizimizde iki tam sayı, bir tane ondalıklı sayı girdiğimizde, ondalıklı sayının varlığı (değerinden bağımsız olarak), dizinin elemanları tuttuğu cins olarak tam sayıyı değil, hepsini içerebilecek ondalıklı sayı cinsini seçmesini gerektiriyor (genel olarak: tam sayıları bilgi kaybı olmadan ondalıklı olarak tutabilsek de, tersi mümkün değil); üçüncüsünde ise, tam sayı olsun, ondalıklı sayı olsun, bunları string olarak tutabiliyoruz ama string değişkenlerini -genel olarak- sayı olarak ifade edemediğimizden, elemanların cinsi "en küçük ortak cins" olan string olarak tanımlanıyor.

Bir dizinin tuttuğu eleman cinsini de `dtype` metodu ile öğreniyoruz:

[12]:
```
print("dizi_1'in elemanlarının cinsi :",dizi_1.dtype,"\n")
print("dizi_2'nin elemanlarının cinsi:",dizi_2.dtype,"\n")
print("dizi_3'ün elemanlarının cinsi :",dizi_3.dtype,"\n")
```

```
dizi_1'in elemanlarının cinsi : int64

dizi_2'nin elemanlarının cinsi: float64

dizi_3'ün elemanlarının cinsi : <U32
```

Tipte yazılı kısım cinsi (integer: tam sayı; float: ondalıklı sayı; U: Unicode string) verirken, peşinden gelen sayı her bir eleman için hafızada ayrılan yer miktarını (bit cinsinden) verir.

Bir dizinin elemanlarının cinsini **zorla** bir başka cinse dönüştürmek mümkündür: bunu `astype` metotu ile yaparız:

[13]:
```
print(dizi_2.astype(int))
```

```
[0 1 2]
```

Ama bunun mümkün olmadığı yerde de zorlamanın pek bir alemi yok:

[14]:
```
print(dizi_3.astype(float))
```

```
␣
↪---------------------------------------------------------------------------

ValueError                                Traceback (most recent call␣
↪last)

<ipython-input-14-b930b903e208> in <module>
----> 1 print(dizi_3.astype(float))
```

```
ValueError: could not convert string to float: 'iki'
```

150 tane tam sayı elemanı olan bir diziye, 151. eleman olarak ondalıklı sayı eklediğimizde bütün elemanlarının tipi nasıl da tümden ondalıklı sayılara dönüşüyorsa, iki sayıyı topladığımızda da bu kural otomatikman uygulanır:

[15]:
```
# Tam sayı + tam sayı = tam sayı:
5 + 7
```

[15]:
```
12
```

[16]:
```
# Tam sayı + ondalıklı sayı = ondalıklı sayı:
5 + 7.5
```

[16]:
```
12.5
```

[17]:
```
# Çok mu barizdi? Öyleyse bir de şuna bakalım:
5 + 7.0
```

[17]:
```
12.0
```

1.7 Tipik Matrisler

Şimdi eğri oturup, doğru konuşalım: NumPy'da matris elemanlarını tanımlamak çoğu kez epey sıkıcı. Bu yüzden hayatı kolaylaştırıcı birkaç tipik hazır matris türü var.

1.7.1 zeros : sıfır sıfır sıfır...

Adı üzerinde, istediğimiz boyutta, tüm elemanları sıfır olan bir matris döndürür - yapmamız gereken, istediğimiz matris boyutunu belirlemekten ibaret:

[18]:
```
mat_0 = np.zeros([2,3,4])
print (mat_0)
```

```
[[[0. 0. 0. 0.]
  [0. 0. 0. 0.]
  [0. 0. 0. 0.]]

 [[0. 0. 0. 0.]
  [0. 0. 0. 0.]
  [0. 0. 0. 0.]]]
```

1.7.2 ones : bir bir bir... (nereye kadar gideceğiz böyle, haydi bakalım...)

Bu da, bütün elemanları 1 olan bir matris döndürür:

[19]:
```
mat_1 = np.ones([2,5,2])
print(mat_1)
```

```
[[[1. 1.]
  [1. 1.]
  [1. 1.]
  [1. 1.]
  [1. 1.]]

 [[1. 1.]
  [1. 1.]
  [1. 1.]
  [1. 1.]
  [1. 1.]]]
```

1.7.3 fill ile matrisi doldurmak

Bütün elemanları sıfır veya bir olan bir matrisi nasıl tanımlayacağımızı artık biliyoruz, peki bütün elemanlarının 2.7 olmasını istiyorsak? Bu durumda, iki-üç(-çok?) yolumuz var:

1. 0-matrisi oluşturup, bütün elemanlarına 2.7 ekleriz:

[20]:
```
mat_0 + 2.7
```

[20]:
```
array([[[2.7, 2.7, 2.7, 2.7],
        [2.7, 2.7, 2.7, 2.7],
        [2.7, 2.7, 2.7, 2.7]],

       [[2.7, 2.7, 2.7, 2.7],
        [2.7, 2.7, 2.7, 2.7],
        [2.7, 2.7, 2.7, 2.7]]])
```

2. 1-matrisi oluşturup, bütün elemanlarını 2.7 ile çarparız:

[21]:
```
mat_1 * 2.7
```

[21]:
```
array([[[2.7, 2.7],
        [2.7, 2.7],
        [2.7, 2.7],
        [2.7, 2.7],
        [2.7, 2.7]],

       [[2.7, 2.7],
        [2.7, 2.7],
```

```
        [2.7, 2.7],
        [2.7, 2.7],
        [2.7, 2.7]]])
```

3. Herhangi bir matris oluşturup (ya da hazır alıp), bütün elemanlarını 2.7 ile *doldururuz*:

[22]:
```
mat_n = np.array([[1,2,3.],[4,5,6]])
print(mat_n)
print("----------")
mat_n.fill(2.3)
print(mat_n)
```

```
[[1. 2. 3.]
 [4. 5. 6.]]
----------
[[2.3 2.3 2.3]
 [2.3 2.3 2.3]]
```

`fill` metodunu kullanırken dikkat etmeniz gereken iki şey var: * Metodu kullandığınızda doğrudan ilgili matrise etki eder, yeni bir matris döndürmez. * Elinizdeki matrisin cinsi ne ise, doldururken kullandığınız değerin cinsini matrisin cinsine döndürür.

İkinci noktanın nasıl çalıştığını görmek için üstteki örneği bir kez daha, bu kez "hilesiz, el çabukluğu kerametsiz", ağır çekimde izleyelim:

[23]:
```
mat_n = np.array([[1,2,3],[4,5,6]])
print(mat_n)
print("----------")
mat_n.fill(2.3)
print(mat_n)
```

```
[[1 2 3]
 [4 5 6]]
----------
[[2 2 2]
 [2 2 2]]
```

Hoppala! Yukarıdakinin *aynısını* yazdık ama bu sefer 2.3 yerine, 2 ile doldurdu… nereyi farklı girdik ki?.. (haydi bulun bakalım, size 1 dakika süre…)

….

Buldunuz mu? İki girişte de ilk satırdaki "3"e bakın. Evet, tam orası işte! İlk kodda "3." diyerek, çaktırmadan bütün diziyi ondalıklı tipine dönüştürmüşüz, ikincisinde hepsi tam sayı. Bu yüzden ikincisinde "2.3" ondalıklı sayısı ile doldur deyince matrisimize, "hiç kusura bakma, ben tam sayı matrisiyim, çok istiyorsa tam sayı kılığına girsin, 2 olarak eklerim" diyor, olayımız bundan ibaret. 8P 8)

1.7.4 birim matris

Birim matris, tanım itibarı ile -ve tabii ki boyutları ile uyumlu olmak şartıyla- bir matrisle çarpıldığı zaman, o matrisi değiştirmeyen matristir. Bunu da sağlamanın tek yolu, köşegeni boyunca "1" değerini almasıdır. "Identity" (birim, etkisiz) olarak isimlendirilip "I" harfi ile temsil edildiğinden, okunuşundan yola çıkıp, ufak bir kelime oyunu ile `eye` olarak tanımlanır:

[24]:
```
mat_birim_5x5 = np.eye(5,5)
print(mat_birim_5x5)
```

```
[[1. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0.]
 [0. 0. 1. 0. 0.]
 [0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 1.]]
```

[25]:
```
# Kare matris olması zorunluluğu yoktur:
mat_birim_3x5 = np.eye(3,5)
print(mat_birim_3x5)
```

```
[[1. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0.]
 [0. 0. 1. 0. 0.]]
```

1.7.5 rastgele matrisler

Bazen uğraşmak istemeyiz, "sen kafana göre doldur, ben sonra bakarım" deriz. Bu durumlarda numpy.random kütüphanesinin rand ve randint'i epey yardımcı olur:

[26]:
```
mat_rastgele_tamsayilar = np.random.randint(5,10,[2,3,4])
print(mat_rastgele_tamsayilar)
```

```
[[[6 8 9 5]
  [9 5 7 8]
  [9 9 9 5]]

 [[6 9 6 8]
  [8 5 7 6]
  [9 5 5 9]]]
```

[27]:
```
mat_rastgele_ondaliklar = np.random.rand(2,3,4)
print(mat_rastgele_ondaliklar)
```

```
[[[0.32050658 0.16444892 0.6934229  0.07682733]
  [0.67717982 0.99606652 0.55347651 0.71407072]
  [0.56696724 0.51968845 0.81804893 0.69267283]]

 [[0.57512085 0.19622693 0.849945   0.92691827]
```

```
[0.66443058 0.57485047 0.17344993 0.60702466]
[0.15546376 0.31921254 0.54711549 0.65001498]]]
```

Parametreler anlaşılıyor mu çağrıştan? randint'e ilk parametre olarak alt limiti (dahil), ikinci parametre olarak üst limiti (hariç), üçüncü parametre olarak da matrisimizin arzu ettiğimiz boyutunu belirtiyoruz (Bu arada, Python'da harikulade bir parametre sıralama opsiyonu var, birkaç haftaya göreceğiz inşallah 8)

rand'da ise parametre olarak sadece boyutu verebiliyoruz, o da bize, alışık olduğumuz üzere 0 ile 1 arasında (0 dahil, 1 hariç) sayılar üretiyor (teknik not: düzgün dağılımdan).

1.7.6 arange ile aralıklar

`arange` komutu bize bir boyutlu diziler verir ama elemanlarını istediğimiz adımla türetebildiğimiz için, peşine bir de `reshape` çekersek tadından yenmez olur. İlk parametre başlayacağı sayı (dahil), ikinci parametre biteceği sayı (hariç), üçüncü parametre de adım boyu olur:

[28]:
```
mat_aralik = np.arange(4,10.1,2.3)
print(mat_aralik)
```

```
[4.  6.3 8.6]
```

Gördüğünüz üzere, ne başlangıcın, ne bitişin, ne de adım boyunun tam sayı olmak gibi bir zorunluluğu yok. İleriye doğru gidebildiğimiz gibi, geriye doğru da gidebiliriz (Nasıl?.. Adım boyumuzu negatif alarak tabii ki!):

[29]:
```
mat_aralik_gerigeri = np.arange(10,2,-2)
print(mat_aralik_gerigeri)
```

```
[10  8  6  4]
```

[30]:
```
# Bir de arange + reshape kombosu yapalım, tam olsun:
mat_kombo_3x3 = np.arange(1,10).reshape(3,3)
print(mat_kombo_3x3)
```

```
[[1 2 3]
 [4 5 6]
 [7 8 9]]
```

Yukarıdaki örnekte nokta ardından nokta birleşik yazımı (metodun dönüşüne metot) kafanızı karıştırıyorsa, parantez içinde de güzel güzel belirtebilirsiniz:

[31]:
```
mat_kombo_3x3 = (np.arange(1,10)).reshape(3,3)
print(mat_kombo_3x3)
```

```
[[1 2 3]
 [4 5 6]
 [7 8 9]]
```

 (Selçuk Erdem, Karikatürler 1, s.72)

1.7.7 linspace: düzenli, disiplinli

Sayısal örneklerle haşır neşir olacağımız için bir fonksiyonun belli -ve hemen her zaman düzenl aralıklı- noktalarda değerini hesaplatmak istediğimizde arange yardıma koşar. Ama mesela doğrudan "23 ile 30 arasındaki değerleri hesaplayalım, 100 tane değer alalım" dersek, o zaman biraz zorlanıyoruz (ben zorlanıyorum en azından... `adım_boyu = (30 - 23) / 100`?..

Bu gibi durumlarda, bu işi yapan hazır linspace komutumuz var:

[32]:
```
mat_duzenli_guzel_aralik = np.linspace(23,30,100)
print(mat_duzenli_guzel_aralik)
print("----------------------")
print("Bu güzel, düzenli dizinin eleman sayısı: ",mat_duzenli_guzel_aralik.size)
```

```
[23.         23.07070707 23.14141414 23.21212121 23.28282828 23.35353535
 23.42424242 23.49494949 23.56565657 23.63636364 23.70707071 23.77777778
 23.84848485 23.91919192 23.98989899 24.06060606 24.13131313 24.2020202
 24.27272727 24.34343434 24.41414141 24.48484848 24.55555556 24.62626263
 24.6969697  24.76767677 24.83838384 24.90909091 24.97979798 25.05050505
 25.12121212 25.19191919 25.26262626 25.33333333 25.4040404  25.47474747
 25.54545455 25.61616162 25.68686869 25.75757576 25.82828283 25.8989899
 25.96969697 26.04040404 26.11111111 26.18181818 26.25252525 26.32323232
 26.39393939 26.46464646 26.53535354 26.60606061 26.67676768 26.74747475
 26.81818182 26.88888889 26.95959596 27.03030303 27.1010101  27.17171717
 27.24242424 27.31313131 27.38383838 27.45454545 27.52525253 27.5959596
 27.66666667 27.73737374 27.80808081 27.87878788 27.94949495 28.02020202
 28.09090909 28.16161616 28.23232323 28.3030303  28.37373737 28.44444444
 28.51515152 28.58585859 28.65656566 28.72727273 28.7979798  28.86868687
 28.93939394 29.01010101 29.08080808 29.15151515 29.22222222 29.29292929
 29.36363636 29.43434343 29.50505051 29.57575758 29.64646465 29.71717172
 29.78787879 29.85858586 29.92929293 30.        ]
----------------------
Bu güzel, düzenli dizinin eleman sayısı:  100
```

Yani, linspace ile tek yapmamız gereken başlangıcı (dahil), bitişi (**o da dahil**, aman dikkat!) ve toplamda kaç tane nokta istediğimiz belirtip, arkamıza yaslanmak!

1.8 Matris işlemleri (dört işlem + bir ters, bir düz)

1.8.1 Dört temel işlem (eleman bazında)

Matrislerimizi -birbirleriyle boyutları uyumlu olduğu sürece- dört temel işlemi kullanarak eleman bazında toplayabilir, çıkarabilir, çarpabilir ve hatta bölebiliriz.

[33]:
```
np.random.seed(220)
mat_a_3x2 = np.random.randint(1,10,[3,2])
print("mat_a_3x2:\n",mat_a_3x2)
print("-"*50)

mat_b_3x2 = np.random.randint(1,10,[3,2])
print("mat_b_3x2:\n",mat_b_3x2)
print("-"*50)

mat_c_2x4 = np.random.randint(1,10,[2,4])
print("mat_c_2x4:\n",mat_c_2x4)
print("-"*50)
```

```
mat_a_3x2:
 [[4 5]
 [9 3]
 [1 3]]
--------------------------------------------------
mat_b_3x2:
 [[7 1]
 [9 3]
 [6 6]]
--------------------------------------------------
mat_c_2x4:
 [[3 1 6 1]
 [3 1 1 7]]
--------------------------------------------------
```

[34]:
```
print("mat_a_3x2 + mat_b_3x2")
print(mat_a_3x2 + mat_b_3x2)
```

```
mat_a_3x2 + mat_b_3x2
[[11  6]
 [18  6]
 [ 7  9]]
```

[35]:
```
print("mat_a_3x2 * mat_b_3x2")
print(mat_a_3x2 * mat_b_3x2)
```

```
mat_a_3x2 * mat_b_3x2
[[28  5]
 [81  9]
 [ 6 18]]
```

[36]:
```
print("mat_a_3x2 / mat_b_3x2")
print(mat_a_3x2 / mat_b_3x2)
```

```
mat_a_3x2 / mat_b_3x2
```

```
[[0.57142857 5.        ]
 [1.         1.        ]
 [0.16666667 0.5       ]]
```

1.8.2 Skaler ve Vektörel Çarpımlar

Skaler Çarpım
Matrislerin skaler çarpımını ise `dot` fonksiyonu ile sağlarız:

[37]:
```
skaler_carpim_3x4 = np.dot(mat_a_3x2,mat_c_2x4)
print("skaler_carpim_3x4 = np.dot(mat_a_3x2,mat_c_2x4):")
print(skaler_carpim_3x4)
```

```
skaler_carpim_3x4 = np.dot(mat_a_3x2,mat_c_2x4):
[[27  9 29 39]
 [36 12 57 30]
 [12  4  9 22]]
```

Vektörel Çarpım
Vektörel çarpım ise `cross` fonksiyonu ile yapılmakta:

$$\hat{i} \times \hat{j} = \hat{k}$$

[38]:
```
i_hat = np.array([1,0,0])
j_hat = np.array([0,1,0])
k_hat = np.array([0,0,1])

print("i_hat x j_hat = ",np.cross(i_hat,j_hat))
print("-"*50)
print("mat_a_3x2 x mat_b_3x2 =",np.cross(mat_a_3x2,mat_b_3x2))
```

```
i_hat x j_hat =  [0 0 1]
--------------------------------------------------
mat_a_3x2 x mat_b_3x2 = [-31   0 -12]
```

1.8.3 Matrisin transpozesi, tersi ve determinantı

Transpozesi Matrislerimizin transpozesini `T` metodu ile alırız (Uzun uzadıya yazmak isterseniz transpose() komutunu kullanabilirsiniz)

[39]:
```
print(mat_a_3x2)
print("-"*50)
print(mat_a_3x2.T)
print()
print(np.transpose(mat_a_3x2))
mat_d_2x3x4 = np.random.randint(1,10,[2,3,4])
print("="*50)
```

```
print(mat_d_2x3x4)
print("-"*50)
print(mat_d_2x3x4.T)
```

```
[[4 5]
 [9 3]
 [1 3]]
--------------------------------------------------
[[4 9 1]
 [5 3 3]]

[[4 9 1]
 [5 3 3]]
==================================================
[[[9 6 2 9]
  [9 1 9 3]
  [5 1 1 6]]

 [[5 3 9 6]
  [7 2 9 2]
  [9 2 3 7]]]
--------------------------------------------------
[[[9 5]
  [9 7]
  [5 9]]

 [[6 3]
  [1 2]
  [1 2]]

 [[2 9]
  [9 9]
  [1 3]]

 [[9 6]
  [3 2]
  [6 7]]]
```

Tersi

Kare bir matrisin tersini `linalg` (*linear algebra*) kütüphanesinin `inv` (*inverse*) komutu ile alabiliriz:

[40]:
```
np.random.seed(220)
mat_e_3x3 = np.random.randint(1,10,[3,3])
mat_e_3x3_tersi = np.linalg.inv(mat_e_3x3)
print(mat_e_3x3)
print(mat_e_3x3_tersi)
```

```
[[4 5 9]
```

```
 [3 1 3]
 [7 1 9]]
[[-0.14285714  0.85714286 -0.14285714]
 [ 0.14285714  0.64285714 -0.35714286]
 [ 0.0952381  -0.73809524  0.26190476]]
```

Bildiğiniz üzere, bir matrisin tersi, o matrisle çarpıldığında, birim matrisini veren matristir, yani:

$$A \cdot A^{-1} = \mathbb{1}$$

Yukarıdaki hesabımızı kontrol edelim:

[41]:
```
print(np.dot(mat_e_3x3,mat_e_3x3_tersi))
```

```
[[ 1.00000000e+00  0.00000000e+00  0.00000000e+00]
 [ 0.00000000e+00  1.00000000e+00  1.11022302e-16]
 [-1.11022302e-16 -8.88178420e-16  1.00000000e+00]]
```

10^{-16} küçüklüğünü pratik olarak 0 alabileceğimiz için, çarpımın gayet güzel bir birim matris vermiş olduğunu görüyoruz! 8)

'Tersimsisi'
Bir matrisin tersi hesaplanırken, analitik olarak determinantından da faydalanır. Determinant ise sadece kare matrislerin bir özelliği olduğundan, bu nedenle kare olmayan matrislerin analitik olarak tersleri yoktur. Fakat, nümerik olarak kare olmayan (nxm)'lik bir matris ile çarpılıp, (nxn)'lik bir birim matrisi verebilecek (mxn)'lik bir matris bulunabilir. Kare olmayan bu ters matrisler de sanki-ters, 'tersimsi' (*pseudo-inverse*) olarak adlandırılıp, buradan kısaltmayla, yine `linalg` kütüphanesinin `pinv` komutu ile bulunur (yalnız tabii her matrisin tersi olacak diye bir garanti yoktur).

[42]:
```
np.random.seed(220)
mat_a_3x5 = np.random.randint(1,10,[3,5])
print("mat_a_3x5:")
print(mat_a_3x5)
print("-"*50)

print("mat_a_3x5_tersimsi:")
mat_a_3x5_tersimsi = np.linalg.pinv(mat_a_3x5)
print(mat_a_3x5_tersimsi)
print("-"*50)

print("mat_a_3x5 x mat_a_3x5_tersimsi =")
print(np.dot(mat_a_3x5,mat_a_3x5_tersimsi))
```

```
mat_a_3x5:
[[4 5 9 3 1]
 [3 7 1 9 3]
 [6 6 3 1 6]]
--------------------------------------------------
mat_a_3x5_tersimsi:
```

```
[[-0.00395615 -0.01876163  0.06817458]
 [-0.00167376  0.03052392  0.02884309]
 [ 0.11726905 -0.0413591  -0.03247369]
 [ 0.00768583  0.10084997 -0.07430697]
 [-0.05428559 -0.00789107  0.09827033]]
--------------------------------------------------
mat_a_3x5 x mat_a_3x5_tersimsi =
[[ 1.00000000e+00  2.60208521e-16  0.00000000e+00]
 [ 1.38777878e-16  1.00000000e+00  6.66133815e-16]
 [-5.55111512e-17 -2.08166817e-17  1.00000000e+00]]
```

Determinantı

Kare bir matrisin determinantını da şaşırtıcı olmayan bir biçimde, `linalg` kütüphanesinin `det` komutu ile elde ederiz:

[43]:
```
print("mat_e_3x3")
print(mat_e_3x3)
print("-"*50)
print("det(mat_e_3x3):")
print(np.linalg.det(mat_e_3x3))
```

```
mat_e_3x3
[[4 5 9]
 [3 1 3]
 [7 1 9]]
--------------------------------------------------
det(mat_e_3x3):
-41.999999999999986
```