# NUMPY

Numpy היא ספריית קוד פתוח בשפת פייתון.
הספרייה תומכת בפעולות מתמטיות על מטריצות ומערכים (וקטורים). לספרייה יש אוסף עשיר של פונקציות מתחומי האלגברה הלינארית והסטטיסטיקה, ומאפשרת לעשות חישובים בצורה מהירה יותר מהספריות הסטנדרטיות של פייתון, בעיקר חישובים על מבני נתונים דמויי וקטורים.
לרוב השימוש ב-numpy הוא לצורך חישובים מתחום האלגברה הלינארית כגון: מכפלות סקלריות של וקטורים, מכפלות של מטריצות, הכפלות לפי איברים ,מציאת פתרון למערכת משוואות, דטרמיננטות, מטריצה הופכית, ועוד הרבה מאוד,
כלים שהם אבני היסוד של תחומים כמו למידת מכונה, אינטליגנציה מלאכותית, סטטיסטיקה, תורת המשחקים, וכדו'.
במסמך הבא נסקור את עקרי הספרייה, כמובן שהספרייה ענקית ולא די במסמך אחד כדי לעבור עליה, מומלץ להסתכל על האתר הרשמי של הספרייה לעוד אינפורמציה אודותיה.

**התקנה-**
התקנת הספרייה משורת הפקודה: `pip install numpy`
לאחר ההתקנה נוכל להוסיף את כל הספרייה או להוסיף מודולים ספציפים ממנה.

## *פעולות אריתמטיות על מערך-*

כשרצינו לאסוף כמה אבייקטים באיזשהו מבנה נתונים דמוי וקטור היינו עושים את זה ברשימה או ב-tuple , ב-numpy יש לנו דבר חדש שנקרא מערך .
מערך הוא כמו tuple- הוא אובייקט immutable ששומר בתוכו אוסף של אובייקטים, אבל בעוד רשימה או tuple נועדו כדי לשמור על הנתונים, מערך משמש כדי לבצע עליהם פעולות מתמטיות כגון חיבור וקטורים, כפל וקטורים וכו'.
קריאה למערך תהיה דרך הפונקציה array, והיא מקבל רשימה או tuple מסוג מסויים ויוצר מערכך מאותו מבנה.
במידה והוא קיבל כמה אובייקטים מאותו הסוג הוא ימיר אותם לסוג הכי מורכב מבניהם שניתן להמיר אליו:

```
arr = np.array((1,2,3,'a'))
arr
```

```
array(['1', '2', '3', 'a'], dtype='<U11')
```

רשימה או tuple שאנחנו מבצעים עליהם אופרטור של חיבור או כפל מחזיר אובייקט חדש מאותו טיפוס שהוא שרשור של שני האופרנטים, לעומת זאת אופרטור על מערך ממש מבצע פעולה אריתמטית בין שני מערכים:

```
lis = [1,2,3]
print(f"list+list: {lis+[4,5,6]}")

arr = np.array((1,2,3)
print(f"arr + list: {arr + [4,5,6]}")
```

```
list+list: [1, 2, 3, 4, 5, 6]
arr + list: [5 7 9]
```

חיבור בין רשימה או tuple למערך מחזיר אובייקט מטיפוס מערך:

```
print(f"type(list+array) = {type(arr + [4,5,6])}")
```

```
type(list+array) = <class 'numpy.ndarray'>
```

כל פעולה אריתמטית שנבצע עם סקאלר תתבצע על כל איבר בנפרד.
וניתן גם לבצע פעולות אריתמטיות שלא קיימות ברשימה או בtuple, כגון חזקה או חילוק, וגם כאן הפעולה תבצע על כל איבר במערך בנפרד.
למרות שניתן להכניס כמה אובייקטים מטיפוסים שונים לאותו מערך, המהות של מערך הוא ייצוג של וקטור אלגברי, וכזכזה לא ניתן לחבר שני וקטורים עם גדלים שונים, אך יוצא דופן לכך הוא ווקטור בגודל אחד שנחשב לסקאלר:

```
print(arr/2)
print(arr**2)
print(arr*5)
print(arr + [5])

try:
    arr + [1,2] #=> ValueError

except ValueError as e:
    print(e)
```

```
[0.2 1. 1.5]
[1 4 9]
[ 5 10 15]
[6 7 8]
operands could not be broadcast together with shapes (3,) (2,)
```

חוץ מהאופרטורים הפשוטים יש מגוון עצום של פונקציות מתמטיות ,שלא קיימות בברירת מחדל של השפה, למשל פונקציות טריגונומטריות כמו sin, cos, tanh , או פונקציות מעריכיות כמו: לוגים, שורשים, ועוד:

```
print(np.tanh(arr))
print(np.exp(arr))
print(np.sqrt(arr))
print(np.log(arr))
```

```
[0.76159416 0.96402758 0.99505475]
[ 2.71828183  7.3890561  20.08553692]
[1.         1.41421356 1.73205081]
[0.         0.69314718 1.09861229]
```

פונקציות לינאריות על מערכים-
ביצוע פעולות אריתמטיות למשל כפל או חיבור מתבצע בין כל שני איברים הנמצאים באותו מיקום בשני המערכים, אבל לפעמים יש צורך בפעולות כמו כפל סקאלרי בין שני ווקטורים, אופציה אחת לעשות את זה היא ע"י סכמיה של התוצאה המתקבלת מכפל בין שני המערכים, אבל יש גם אלטרנטיבות אם זה נשמע לכם ארוך מידי, למשל אפשר להשתמש בפונקציה np.dot() שמקבלת שני מערכים שמכילים מספרים, ובאותו האורך, ומחזירה את המכפלה הסקלארית שלהם, ואם גם זה מרגיש לכם ארוך מידי, יש את האופרטור @ שמשמש למכפלה סקאלרית, אבל אישית לא הייתי ממליץ להשתמש באופרטור אלא אם ברור לכם שהנמען מכיר את הסימון הזה.
תזכורת: מכפלה סקאלרית מוגדרת כ- $ab=a^t b=\sum_{d=1}^{D}(a_d+b_d)$ :

```
arr2 = np.array([4,5,6])

print(np.sum(arr2*arr)) #=> 32
print(arr2.dot(arr)) #=> 32
print(arr2 @ arr)  #=> 32
```

דרך נוספת לייצג מכפלה סקלארית היא כך:

$$a^t b = \|a\| \bullet \|b\| \cos \Theta_{ab}$$

בשביל זה צריך לדעת את הנורמה של a ו-b (האורך של כל ווקטור, כלומר השורש של המכפלה הסקאלרית של הווקטור עם עצמו) וקוסינוס הזווית שביניהם, אבל למעשה אם אין לנו את אחד המרכיבים נוכל לגלות אותו בקלות ע"י שינוי סדר החישוב, למשל ננסה לגלות מהו Θ:

$$\cos\Theta = \frac{a^t b}{\|a\| \bullet \|b\|}$$

בשביל למצוא את הנורמה נוכל או לבצע שורש על המכפלה הסקלארית של הווקטור עם עצמו או להשתמש בפונקציה בנויה מראש של numpy הנמצאת תחת מרחב השם linalg.norm , linalg() :

```
a,b = arr,arr2
cosangle = a@b/(np.linalg.norm(a)*np.linalg.norm(b))
angle_degree= np.rad2deg(np.arccos(cosangle))
degree_sign= u'\N{DEGREE SIGN}'
print(f"{angle_degree}{degree_sign}")
```

```
12.933154491899135°
```

**מטריצות-**

ברמת העיקרון קיים אובייקט מטיפוס מטריצה בnumpy- אבל הוא פחות נפוץ, במקום עדיף להשתמש במערך של מערכים עם אותו האורך, numpy אפילו שומרת על הדפסה יפה של המטריצה:

```
arr = np.array([[1,2],[3,4]])
print(arr)
```

```
[[1 2]
 [3 4]]
```

Numpy מאפשר גישה לאינדקס ספציפי במטריצה בשני אופנים, או בצורה המקובלת שדומה לג'אווה- שכל אינדקס בתוך סוגרים מרובעים נפרדים, או בצורה שדומה יותר לc# - זוג אחד של סוגרים מרובעים שבתוכו אינדקס של השורה והאינדקס של העמודה מופרדות בפסיק:

```
print(arr[0][0])
print(arr[0,0])
```

```
1
1
```

והנה משהו שאפשר לבצע רק במערך של numpy, מה יקרה אם נרצה לקבל עמודה – ברשימה נצטרך לעבור בלולאה על כל הערכים באותו עמודה, לעומת זאת במערך ניתן לגשת לעמודה ישירות עם הסימון ':' במקום האינדקס של השורה ומספר העמודה:

```
arr[:,0]
```

```
array([1,3])
```

כל פונקציה אריתמטית שניתן לעשות על מערך חד ממדי, קרי וקטור, ניתן גם לבצע על מערך דו ממדי: כפל עם סקאלר, חיבור של מטריצה עם מטריצה אחרת מאותו סדר גובה, וכו'.
למטריצות גם יש כמה תכונות נוספות על ווקטורים, למשל שחלוף המטריצה שניתן לעשות עם המשתנה T של האובייקט(לכל מטריצה יש אובייקט שמור שהוא השחלוף שלה), מטריצה הופכית או דטרמיננטה ממרחב השם linalg :

```
print("inverse of arr: ")
print(np.linalg.inv(arr))
```

```
print("determinant of arr:")
print(np.linalg.det(arr))
print("arr Transpose:")
print(arr.T)
```

```
inverse of arr:
[[-2.   1. ]
 [ 1.5 -0.5]]
determinant of arr:
-2.0000000000000004
arr Transpose:
[[1 3]
 [2 4]]
```

מכפלה של מטריצות כברירת מחדל תעשה כמכפלה לפי איברים, אולם ניתן להכפיל מטריצות בצורה דומה להכפלה של ווקטורים עם הפונקציה dot, אבל יש לשים לב שהדרגה של העמודות של המטריצה הראשונה שווה לדרגה של השורות של המטריצה השנייה, אחרת תיזרק כשגיאה:

```
arr2 = np.array([[1,2,3],[4,5,6]])
arr.dot(arr2)
```

```
array([[ 9, 12, 15],
       [19, 26, 33]])
```

השווה בין מטריצות לא תעשה עם האופרטור == , היות ו-numpy מאוד רגיש לחישובים עד הנקודות הקטנות, במקום נרצה לעשות את החישוב בקירוב עם הפונקצייה allclose() שמוצאת האם המטריצות זהות עד כדי אפסילון,
למשל נבצע הפכיה למטריצה מסויימת ונכפיל אותה עם המטריצה המקורית, לכאורה אנחנו אמורים לקבל את מטריצה היחידה, ולכן אם נשווה את התוצאה למטריצת היחיד אנחנו אמורים לקבל "אמת" , אבל בפועל לא בטוח שזה מה שנקבל:

```
# a^(-1)*a == I ?
np.linalg.inv(arr).dot(arr) == np.array([[1,0],[0,1]])
```

```
array([[False,  True],
       [False,  True]])
```

אבל אם נשתמש בפונקציית allclose() לעומת זאת:

```
np.allclose(np.linalg.inv(arr).dot(arr) , np.array([[1,0],[0,1]]))
```

```
True
```

שימוש בספרייה לחישוב מערכת משוואות-
נניח יש לנו את התרגיל הבא:
כרטיס כניסה לפארק הוא 1.5 שקלים לילד, ולמבוגר הוא 4 שקלים.
ביום מסויים נכנסו לפארק 2200 אנשים, והסכום המצטבר של אותו יום היה 5050 שקלים.
כמה ילדים וכמה מבוגרים נכנסו לגן?

לפנינו שאלה קלאסית של שתי משוואות בשני נעלמים, נגדיר את מספר הילדים ב-$x_1$ ומספר המבוגרים ב-$x_2$.
ונקבל :

$$1.5x_1+4x_2=5050 \quad x_1+x_2=2200$$

נמיר את המשוואה למטריצות ונקבל:

$\begin{pmatrix} 1 & 1 \\ 1.5 & 4 \end{pmatrix}\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}=\begin{pmatrix} 2200 \\ 5050 \end{pmatrix}$ עכשיו נוכל להשתמש ב-numpy כדי לגלות את הפתרון של המשוואה.

השיטה הנאיבית תהיה להכפיל במטריצה ההופכית בשני האגפים, וזוהי אכן שיטה שתעבוד אם כי לא תמיד תיתן את התשובה המדוייקת ביותר והמהירה ביותר.

דרך יעילה ומדויקת יותר היא עם הפונקציה linalg.solve() שמקבלת את מטריצה המשוואות ווקטור הפתרונות , ומחזירה את ערכי ה-x שמקיימים את המערכת:

```
equations = np.array([[1,1],[1.5,4]])
solutions = np.array([2200,5050])
x_value = np.linalg.solve(equations,solutions)
print(f'x1 = {x_value[0]}, x2 = {x_value[1]}')
```

```
x1 = 1500.0, x2 = 700.0
```

כמובן שזאתי דוגמא לשאלה פשוטה מאוד, רוב הזמן נראה מקרים מסובכים יותר, מממדים גבוהים בהרבה, במקרים כאלה טוב שיש לנו פונקציות כמו solve().

יש מקרים בהם נרצה להשתמש במטריצה ספציפית , למשל מטריצת הזהות (מטריצת היחידה), או מטריצה שמלאה רק במספר אחד , numpy דאגו גם לזה וסיפקו לנו פונקציה שמאתחלת את המטריצה למטריצת אפסים עם zeros((rows ,columns)) , או שמאתחלת אותה למטריצת אחדות (שכולה אחדים) עם הפונקציה ones((rows ,columns)), ואם נרצה למלא את המטריצה במספר ששונה מאחד או מאפס פשוט נכפיל את מטריצת האחדות עם סקלר כלשהו.

חוץ מהשתיים האלה אפשר גם להגדיר את מטריצת היחידה בצורה ישירה עם הפונקציה eye(rows) .

תזכורת: מטריצת היחידה או מטריצת הזהות ,שלרוב נכתבת באות I, היא מטריצה שכולה אפסים למעט האלכסון הראשי שלה (i==j) שמלא באחדות, וכל מטריצה A שמוכפלת בה (שיכולה להיכפל) מקיימת: $A \times I = A$ .

```
arr_zeros = np.zeros((3,4))
arr_ones = np.ones((3,4))
arr_tens = arr_ones * 10
arr_identity = np.eye(3)

print('~~~ 3X4 matrix of zeros ~~~')
print(arr_zeros)
print('~~~ 3X4 matrix of ones ~~~')
print(arr_ones)
print('~~~ 3X4 matrix of tens ~~~')
print(arr_tens)
print('~~~ 3X3 Identity matrix ~~~')
print(arr_identity)
```

```
~~~ 3X4 matrix of zeros ~~~
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]
~~~ 3X4 matrix of ones ~~~
[[1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]]
~~~ 3X4 matrix of tens ~~~
[[10. 10. 10. 10.]
 [10. 10. 10. 10.]
 [10. 10. 10. 10.]]
~~~ 3X3 Identity matrix ~~~
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
```

**ערכים רנדומליים -**

יש מקרים בהם צריך לאתחל מטריצות עם ערך רנדומלי כלשהו, למשל בשביל לאתחל רשת נוירונים.
לnumpy יש מרחב שם שלם שמוקדש ליצירת מספרים רנדומלים , למשל בשביל לקבל מספר רנדומלי בין אפס לאחד נוכל להשתמש ב- random.random() ,או כדי לקבל מספר שלם בין מספר מסוים אחר נוכל להשתמש בפונקציה-
random.randint(from , to).
לרוב נרצה לאתחל מטריצה במספרים רנדומלים, אפשר להוסיף ארגומנט נוסף לפונקציה שקוראים לו size ומקבל גודל של מטריצה: random.random(from ,to , size = (row ,col)), או random.random(size = (row,col))
יותר מזה, לפעמים נרצה לקבוע מספר רנדומלי מתוך קבוצה של מספרים, נוכל להשתמש בפונקציה random.choice([...],size=(row,col)):

```
print(f'~~~ random: {np.random.random()} ~~~')
print(f'~~~ randint(1,100): {np.random.randint(1,100)} ~~~')
print(f'~~~ randint(1,100 size=(3,3)): ~~~ \n{np.random.randint(1,100, size=(3,3))}')
print(f'~~~ choice = [2,3,5,7,11,13],size=(3,3) ~~~\n{np.random.choice([2,3,5,7,11,13], size=(3,3))}')
```

```
~~~ random: 0.0908651100249237 ~~~
~~~ randint(1,100): 35 ~~~
~~~ randint(1,100 size=(3,3)): ~~~
[[76 57 17]
 [99 48 20]
 [10 58 89]]
~~~ choice = [2,3,5,7,11,13],size=(3,3) ~~~
[[ 3 11 13]
 [ 7 13  3]
 [ 5 13  2]]
```

יש מקרים בהם נרצה להיצמד להתפלגות מסוימת, למשל כשנרצה לסמלץ אירוע מהמציאות שקורה בהתפלגות ספציפית.
numpy תומכת בכמה התפלגויות build in בניהן התפלגות נורמלית (גאוסיאנית), התפלגות בינומית, התפלגות פואסונית, התפלגות אחידה ועוד הרבה:

```
# Normal Distribution:

'''The Normal Distribution is one of the most important distributions.
It is also called the Gaussian Distribution after the German mathematician Carl Friedrich Gauss.
It fits the probability distribution of many events, eg. IQ Scores, Heartbeat etc.
loc - (Mean) where the peak of the bell exists.
scale - (Standard Deviation) how flat the graph distribution should be.
size - The shape of the returned array: '''

normal_dis = np.random.normal(size=(3, 3))
print('~~~ normal distribution: ~~~')
print(normal_dis)
print("loc=0.5, scale=2, size=(3, 3):")
normal_dis = np.random.normal(loc=0.5, scale=2, size=(3, 3))
print(normal_dis,end='\n\n')


# Binomial Distribution
'''
Binomial Distribution is a Discrete Distribution.
It describes the outcome of binary scenarios, e.g. toss of a coin, it will either be head or tails.
It has three parameters:
n - number of trials.
p - probability of occurence of each trial (e.g. for toss of a coin 0.5 each).
```

```
size - The shape of the returned array.
'''
binomial_dis = np.random.binomial(n=10, p=0.5, size=(3,3))
print('~~~ binomial distribution: ~~~')
print("n=10, p=0.5, size=(3,3):")
print(binomial_dis,end='\n\n')




# Poisson Distribution
'''
Poisson Distribution is a Discrete Distribution.
It estimates how many times an event can happen in a specified time.
e.g. If someone eats twice a day what is probability he will eat thrice?
It has two parameters:
lam - rate or known number of occurences e.g. 2 for above problem.
size - The shape of the returned array.
'''
poisson_dis = np.random.poisson(lam=2, size=(3,3))
print('~~~ poisson distribution: ~~~')
print("lam=2, size=(3,3):")
print(poisson_dis,end='\n\n')



# Uniform Distribution
'''
Used to describe probability where every event has equal chances of occuring.
E.g. Generation of random numbers.
It has three parameters:
a - lower bound - default 0 .0.
b - upper bound - default 1.0.
size - The shape of the returned array.
'''
uniform_dis = np.random.uniform(size=(3, 3))
print('~~~ uniform distribution: ~~~')
print("size=(3,3):")
print(uniform_dis)
```

```
_____________________________________
~~~ normal distribution: ~~~
[[ 1.28366245  1.06644815  0.02160301]
 [-0.69632294 -1.93472226 -1.68291449]
 [-0.97862278  0.29254386  0.13799928]]
loc=0.5, scale=2, size=(3, 3):
[[-1.8203017  -2.09110571 -2.04192032]
 [ 0.8987513   3.79834804 -0.3864286 ]
 [ 1.27670998  1.76292947 -0.46554264]]

~~~ binomial distribution: ~~~
n=10, p=0.5, size=(3,3):
[[6 5 6]
 [2 4 4]
 [5 9 3]]

~~~ poisson distribution: ~~~
lam=2, size=(3,3):
[[1 2 2]
 [1 1 2]
 [1 3 2]]

~~~ uniform distribution: ~~~
size=(3,3):
[[0.54266188 0.16843016 0.76823649]
```

```
[0.12678897 0.21817468 0.14928719]
[0.43279256 0.2452473  0.10074541]]
```