# NUMPY

Numpy היא ספריית קוד פתוח לתכנות בשפה פייתון.   
הספרייה תומכת במטריצות ומערכים (וקטורים) ופעולות מתמטיות עליהן. לספרייה יש אוסף עשיר של פונקציות מתחומי האלגברה הלינארית והסטטיסטיקה, ומאפשרת לעשות חישובים בצורה מהירה יותר משימוש בטיפוסי נתונים לאוספים המגיעים עם פייתון.  
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.  
במסמך הבא נסקור את עקרי השימושי שיש לה ספרייה לספק וניתן למצוא עוד הרבה חומר על הספרייה מ[האתר הרשמי.](https://numpy.org/doc/stable/)  
   
  
  
**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 במערך או ב-tuple , ב-numpy יש לנו דבר חדש שנקרא מערך .  
מערך הוא כמו tuple- הוא אובייקט immutable ששומר בתוכו רשימה 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 הספרייה numpy באובייקט מערך, והוא 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 עליו 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, וככזה 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](https://numpy.org/doc/stable/reference/routines.math.html):

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- :

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:

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 שבניהם, אבל למעשה אם אין לנו את אחד המרכיבים נוכל לגלות אותו בקלות ע"י שינוי סדר החישוב, למשל ננסה לגלות מהו Θ:

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

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 מומלץ לשימוש, במקום עדיף להשתמש במערכך של מערכים עם אותו האורך, numpy אפילו שומר 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, המקום נרצה לעשות את החישוב בקירוב עם הפונקצייה allclose() שמוצאת האם בקירוב ממשי המטריצות זהות,   
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

טקסט