5-NumPy

By: Nourah almutlag

مقدمة

من المعروف أن لغة البايثون توفر العديد من المكتبات الرياضية ولكن هذه المكتبات تعتبر بسيطة وغير كافية لمعظم العمليات الحسابية في تحليل البيانات ومن هنا جاءت الحاجة لمكتبات توفر مزايا أفضل لإتمام العمليات الرياضية مثل مكتبة NumPy.

ماهي مكتبة NumPy ؟

تعتبر مكتبة **NumPy(Numerical Python)** من أهم المكتبات المستخدمة في الحوسبة العلمية (scientific computing) وتحليل البيانات وهي تعتبر الأساس الذي تم بناء العديد من المكتبات الرياضية عليه مثل مكتبة pandas حيث تم تطوير هذه المكتبة بالاعتماد على المفاهيم الخاصة بمكتبة NumPy لذا فإن تعلم المفاهيم الأساسية في هذه المكتبة مهم عند استخدام المكتبات الرياضية الأخرى.

- مكتبة NumPy هي مكتبة مفتوحة المصدر تم تطويرها من قبل Travis Oliphant في عام 2006.

مميزات مكتبة NumPy

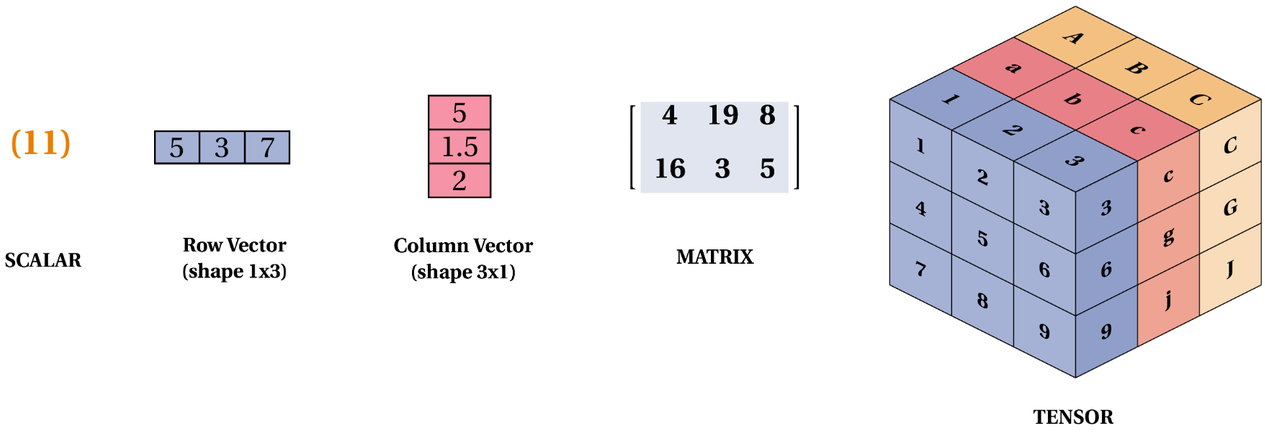
- توفر مكتبة NumPy المصفوفات (multidimensional array) بما في ذلك العمليات على هذه المصفوفات مثل: العمليات الرياضية والمنطقية (mathematical and logical operations) و shape manipulation و

العمليات الإحصائية (basic statistical) والجبر الخطي (basic linear algebra) وغيره.

- مكتبة NumPy تعتبر الأكثر استخدامًا لحساب المصفوفات متعددة الأبعاد والمصفوفات الكبيرة (multidimensional arrays and large arrays).
- تحتوي العديد من functions التي تتيح إجراء عمليات على المصفوفات بطريقة أكثر كفاءة وفعالية.
- إجراء عمليات حسابية عالية المستوى (high-level mathematical calculations).

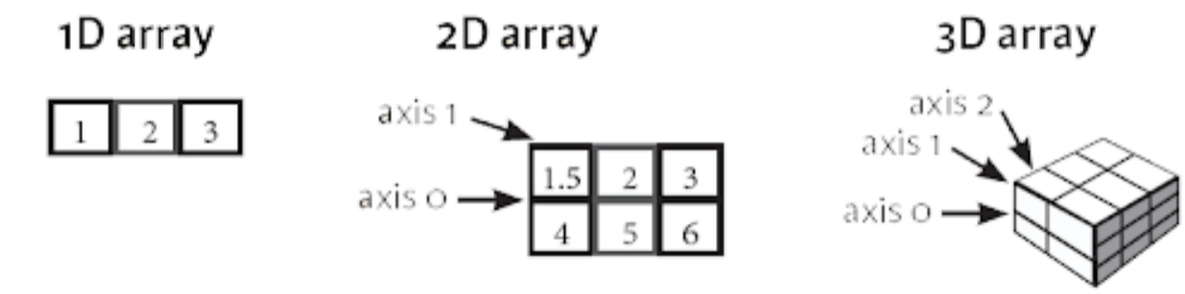
مفاهيم أساسية

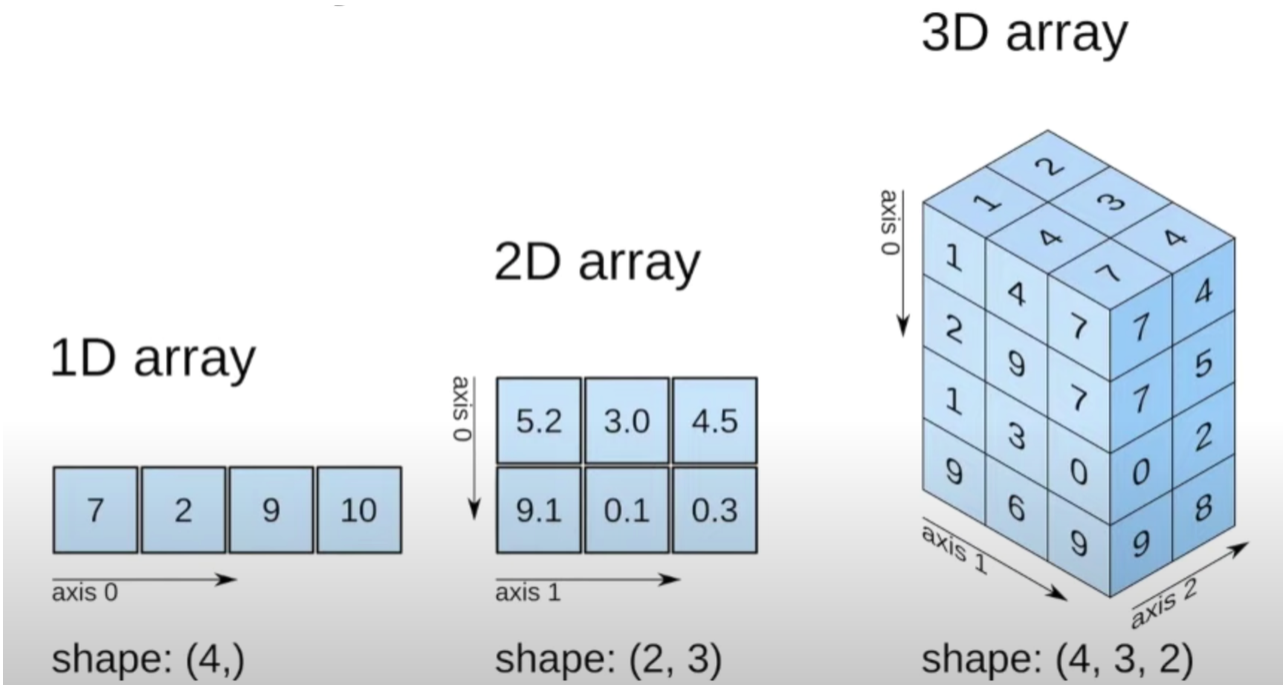
قبل التعرف على المصفوفات التابعة لمكتبة NumPy توجد لدينا أشكال مختلفة من وحدات التخزين وهي كالتالي:



Scalar, vector, matrix, tensor | Credit: refactored.ai

أشكال المصفوفات في مكتبة NumPy





ماهو ndarray object:

- تعتمد مكتبة NumPy على كائن ndarray object وهو اختصار لكلمة N-dimensional array، هذا الكائن يعتبر مصفوفة متجانسة ومتعددة الأبعاد (multidimensional homogeneous array) وعدد العناصر فيها محدد مسبقا. نقصد ب(متجانسة) أي أن جميع العناصر فيها من نفس النوع (type) ونفس الحجم (size).
- يعتبر حجم NumPy arrays ثابت، أي بمجرد تحديد الحجم وقت الإنشاء فلن نستطيع تغيير ذلك بعكس Python lists التي يمكن تغيير حجمها.

تحميل مكتبة NumPy

يمكنك تحميل مكتبة عن طريق استخدام أحد الأمرين:

```
conda install numpy # if you have conda distribution
pip install numpy   # if you don't have conda distribution
```

ملاحظة: إذا قمت بتنزيل برنامج Anaconda لن تحتاج لهذه الخطوة لأن هذه المكتبات تكون مثبتة بشكل تلقائي على جهازك.

التطبيق العملي

استدعاء مكتبة NumPy

```
import numpy as np
```

يتم الاستدعاء مباشرة ,و نستخدم as لوضع اسم مختصر فقط ويكون اي اختصار لكن المتعارفه بعد np لـnumpy.

إنشاء Array عن طريق مكتبة NumPy

```
a = np.array([1, 2, 3])
```

تم تعريف متغير a كـarray عن طريق ()np.array , ثم داخل ()np.array يتم كتابة المصفوفه و تكون ايضاً بداخل الاقواس المربعية [1,2,3], هنا تم إنشاء 1D-Array.

إنشاء List في Python

```
b = [1,2,3]
type(a)
type(b)
```

هنا نقارن بين array و list في python نلاحظ أن a من نوع`numpy.ndarray`، أما b من نوع`list` .

خصائص ndarray

أي كائن ndarray object يحمل عدة خصائص و هي كالتالي:

الخاصية	الوصف
ndarray.size	يحدد حجم المصفوفة وهو يشير لعدد العناصر في المصفوفة (حاصل ضرب عناصر shape).
ndarray.shape	يحدد عدد الأبعاد والعناصر بالمصفوفة عن طريق tuple يتكون من أرقام تحدد حجم كل بعد. على سبيل المثال: (n, m) يرمز n لعدد الصفوف و m لعدد الأعمدة.
ndarray.ndim	يحدد عدد الأبعاد للمصفوفة (dimensions) axes.
ndarray.itemsize	حجم bytes في كل عنصر في المصفوفة.
ndarray.dtype	يحدد نوع البيانات حيث أن كل كائن ndarray object مرتبط بنوع واحد من البيانات.
ndarray.data	يشير إلى buffer الذي يحتوي العناصر الحقيقية في المصفوفة.

لعرض نوع بيانات المصفوفة :

```
a.dtype
```

لعرض عدد أبعاد المصفوفة:

```
a.ndim
```

لعرض حجم المصفوفة :

```
a.size
```

لعرض شكل المصفوفة:

```
a.shape
```

لعرض حجم bytes في عناصر المصفوفة.

```
print(a.itemsize) #It defines the size in bytes of each item i
n the array
```

لعرض buffer الذي يحتوي العناصر الحقيقية في المصفوفة.

```
print(a.data)
```

مثال على المصفوفات من نوع 2D.

```
b = np.array([[1.3, 2.4],[0.3, 4.1]])
print('type = ', b.dtype)
print('number of dimensions = ', b.ndim)
print('shape = ', b.shape)
print('size = ', b.size)
print(b.itemsize)
print(b.data)
```

نلاحظ هنا طريقة انشاء مصفوفة من نوع 2D مشابة جداً لـ1D لكن نرى ان داخل اقواس الـarray اضفنا قوس مربع [] و داخل هذا القوس يتم كتابة كل row في قوس مربع [] وتم فصل كل row عن الاخر بفاصلة " , " .

لطباعة المصفوفة:

```
b
```

أنواع البيانات المدعومة في NumPy

نوع البيانات	الوصف
bool_	Boolean (true or false) stored as a byte
int_	Default integer type (same as C long; normally either int64 or int32)
intc	Identical to C int (normally int32 or int64)
intp	Integer used for indexing (same as C size_t; normally either int32 or int64)
int8	Byte (–128 to 127)

Integer (–32768 to 32767)	int16
Integer (–2147483648 to 2147483647)	int32
Integer (–9223372036854775808 to 9223372036854775807)	int64
Unsigned integer (0 to 255)	uint8
Unsigned integer (0 to 65535)	uint16
Unsigned integer (0 to 4294967295)	uint32
Unsigned integer (0 to 18446744073709551615)	uint64
Shorthand for float64	float_
Half precision float: sign bit, 5-bit exponent, 10-bit mantissa	float16
Single precision float: sign bit, 8-bit exponent, 23-bit mantissa	float32
Double precision float: sign bit, 11-bit exponent, 52-bit mantissa	float64
Shorthand for complex128	complex_
Complex number, represented by two 32-bit floats (real and imaginary compon ents)	complex64
Complex number, represented by two 64-bit floats (real and imaginary components)	complex128

إنشاء مصفوفة تحتوي على بيانات من نوع String.

```
g = np.array([['a', 'b'],['c', 'd']])
g
print(g.dtype)
print(g.dtype.name)
```

هنا تم إنشاء array من نوع 2D و البيانات من نوع string.

إنشاء مصفوفة تحتوي على بيانات من نوع complex

```
f = np.array([[1, 2, 3],[4, 5, 6]], dtype=complex)
f
```

هنا تم إنشاء متغير f كـarray من نوع 2D , نلاحظ هنا انه بإمكاننا تحديد نوع الـarray اثناء إنشائها بشكل مباشر من dtype .

لتحويل الـمصفوفة لنوع بيانات آخر.

```
n = np.array([['1', '2'],['3', '4']])
n.astype(int)
```

في المتغير n تم إنشاء array من نوع 2D و البيانات ارقام لكن من نوع string, حين نريد تحويل هذه الارقام الى نوع integer , نقوم بإستخدام astype و نحدد بداخلها النوع الذي نريد تغير البيانات اليه .

إنشاء المصفوفة

هناك عدة طرق لإنشاء NumPy arrays ومنها التالي:

- أولا: التحويل من Python structures مثل: (lists and tuples) عن طريق دالة (array().

مثال على إنشاء مصفوفة من نوع List.

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

في المتغير c تم إنشاء مصفوفة من نوع 2D و تم تعريف البيانات داخل هذه المصفوفة على هيئة list.

مثال على إنشاء مصفوفة من نوع Tuple.

```
d = np.array(((1, 2, 3),(4, 5, 6)))
d
```

في المتغير d تم إنشاء مصفوفة من نوع 2D و تم تعريف البيانات داخل هذه المصفوفة على هيئة tuple.

مثال على إنشاء مصفوفة من نوع Tuple و List.

```
e = np.array([(1, 2, 3), [4, 5, 6], (7, 8, 9)])
e
```

في المتغير e تم إنشاء مصفوفة من نوع 2D و تم تعريف البيانات داخل هذه المصفوفة على هيئة list و tuple.

- ثانيا: دوال Intrinsic NumPy array creation functions مثل: (arange, ones, zeros).

مثال على إنشاء مصفوفة تحتوي جميع بياناتها على صفر.

```
# a two-dimensional array 3x3
np.zeros((3, 3))
```

حين نريد إنشاء array لكن جميع عناصر هذه الـarray أصفار , نستخدم zeros() ,نحدد بداخلها حجم الـarray المراد إنشائها , حيث أن (n, m) يرمز n لعدد الصفوف و m لعدد الأعمدة.

مثال على إنشاء مصفوفة تحتوي جميع بياناتها على واحد.

```
np.ones((3, 3))
```

حين نريد إنشاء array لكن جميع عناصر هذه الـarray واحدات 1 , نستخدم ones() ,نحدد بداخلها حجم الـarray المراد إنشائها , حيث أن (n, m) يرمز n لعدد الصفوف و m لعدد الأعمدة.

مثال على إنشاء مصفوفة تحتوي على مجموعة متسلسلة من البيانات باستخدام دالة arange.

```
np.arange(0, 10)
np.arange(4, 10)
np.arange(0, 12, 3) #(start, end, gap)
np.arange(0, 6, 0.6)
```

حين نريد إنشاء array من نوع 1D و تكون بياناتها متسلسلة نستخدم arange(), نحدد بداخلها (start, end), وايضاً (start, end, gap).
نرى في المثال الاول سوف يتم إنشاء الـarray تحتوي على مجموعة متسلسلة تبدا من 0 و تنتهي عند 9.
اما في المثال الثاني سوف يتم إنشاء الـarray تحتوي على مجموعة متسلسلة تبدا من 4 و تنتهي عند 9.
و في المثال الثالث سوف يتم إنشاء الـarray تحتوي على مجموعة متسلسلة تبدا من 0 و تنتهي عند 11 لكن يتم تحديد gap هنا ,اي سوف يكون التسلسل يقفز 3 اعداد في كل مره .
و في المثال الاخير سوف يتم إنشاء الـarray تحتوي على مجموعة متسلسلة تبدا من 0 و تنتهي عند 5 و الـgap هنا تم تحديده و سوف يقفز 0.6 رقم في كل مره.

مثال على إنشاء مصفوفة تحتوي على مجموعة متسلسلة من البيانات باستخدام دالة linspace.

```
np.linspace(0,10,5)
```

توفر لنا numpy دالة مشابة جداً لـarange , اي انها سوف تنشئ array من نوع 1D و تكون بياناتها متسلسلة, نحدد بداخلها (start, end), وايضاً (start, end, gap).

> *ملاحظة: الفرق بين arange و linspace هو أن مخرجات دالة arange لا تشمل القيمة النهائية بينما دالة linspace تشمل القيمة النهائية.*

في المثال اعلاه سوف تبدا المصفوفة من 0 و تنتهي عند 10 و تقفز كل مرة 5 اعداد.

مثال على إنشاء مصفوفة باستخدام دالة random.

```
np.random.random(3) # The numbers obtained will vary with ever
y run
```

سيتم إنشاء array من نوع 1D بإرقام عشوائية بإستخدام random(), و يتم تحديد حجم الـarray بداخلها.

مثال على إنشاء multidimensional array باستخدام دالة random.

```
np.random.rand((3,3))
```

و هنا ايضاً يتم إنشاء array من نوع 2D و ايضاً بإرقام عشوائية بستخدام rand() و يتم تحديد حجم الـarray بداخلها ,حيث أن (n, m) يرمز n لعدد الصفوف و m لعدد الأعمدة.

rand() , random() جميعها تقوم بنفس الوظيفة .

مثال على إنشاء مصفوفة تحتوي متغيرات من نوع int باستخدام دالة random.

```
np.random.randint(2, size=10) # random 10 integers (from 0 to 1)
np.random.randint(5, size=(2, 4)) # random 2*4 array (from 0 to 4)
```

هنا ايضاً يتم إنشاء array بعناصر عشوائية لكن من نوع int , بإستخدام randint() و يتم تحديد بداخلها عدد العناصر و الـsize.
في الـarray الاولى سوف يتم إنشاء مصفوفة تتكون من رقمين , و في المثال تم تحديد 2 اي ان الرقمين هما 0 و1, و حجم المصفوفة 10 , سوف ينتج مصفوفة من 10 عناصر و جميع عناصرها تتكون من 0و1.

يمكن أيضا إنشاء المصفوفات عن طريق دوال أخرى مثل:

```
np.full((2,2),7) #(shape, fill_value)
np.full((4, 4), [1, 2, 3, 4])
np.eye(2) # 2-D array with ones on the diagonal and zeros else where
np.empty([2, 2], dtype=int) #(shape, dtype)
```

وايضاً توفر لنا numpy طرق اخرى , مثل حين نريد إنشاء مصفوفة جميع عناصرها تتكون من نفس العنصر اي من رقم معين فقط , فنستخدم full() و يتم تحديد حجم المصفوفة و الرقم او العنصر الذي سوف تتكون منه المصفوفة .

في المثال اعلاه (np.full((2,2),7 سوف يتم إنشاء مصفوفة من نوع 2D و جميع عناصرها تكون الرقم 7.
اما المثال ([np.full((4, 4), [1, 2, 3, 4 سوف يتم إنشاء مصفوفة من نوع 2D ايضاً, لكن هنا تم وضع list من 4 عناصر ,سوف يعوض بهذه الـlist في كل row.

وفي المثال الثالث (np.eye(2 تختلف الفكرة هنا , سوف يتم إنشاء مصفوفة من نوع 2D ,جميع عناصرها اصفار ماعدا القطر , اي أنها سوف تنتج مصفوفة قطرها 1 و باقي عناصرها 0.

اما المثال الاخير (np.empty([2, 2], dtype=int *هنا اسم الداله لايدل عليها , فهي لا تنشئ مصفوفة فارغة , بدل هني تنشئ بأرقام عشوائية لا نحدد (start , end) بل نحدد (shape, dtype) حجم المصوفة و نوع عناصر المصفوفة.*

- ثالثا: عن طريق (Replicating, joining, mutating) للمصفوفة الموجودة.

تغيير شكل المصفوفة (Shape Manipulation)

مثال مصفوفة بشكل one-dimensional arrays.

```
a = np.arange(0, 12)
```

رأينا هذا المثال في السابق , وهو إنشاء مصفوفة من نوع 1D وعناصرها مجموعة بيانات متسلسلة.

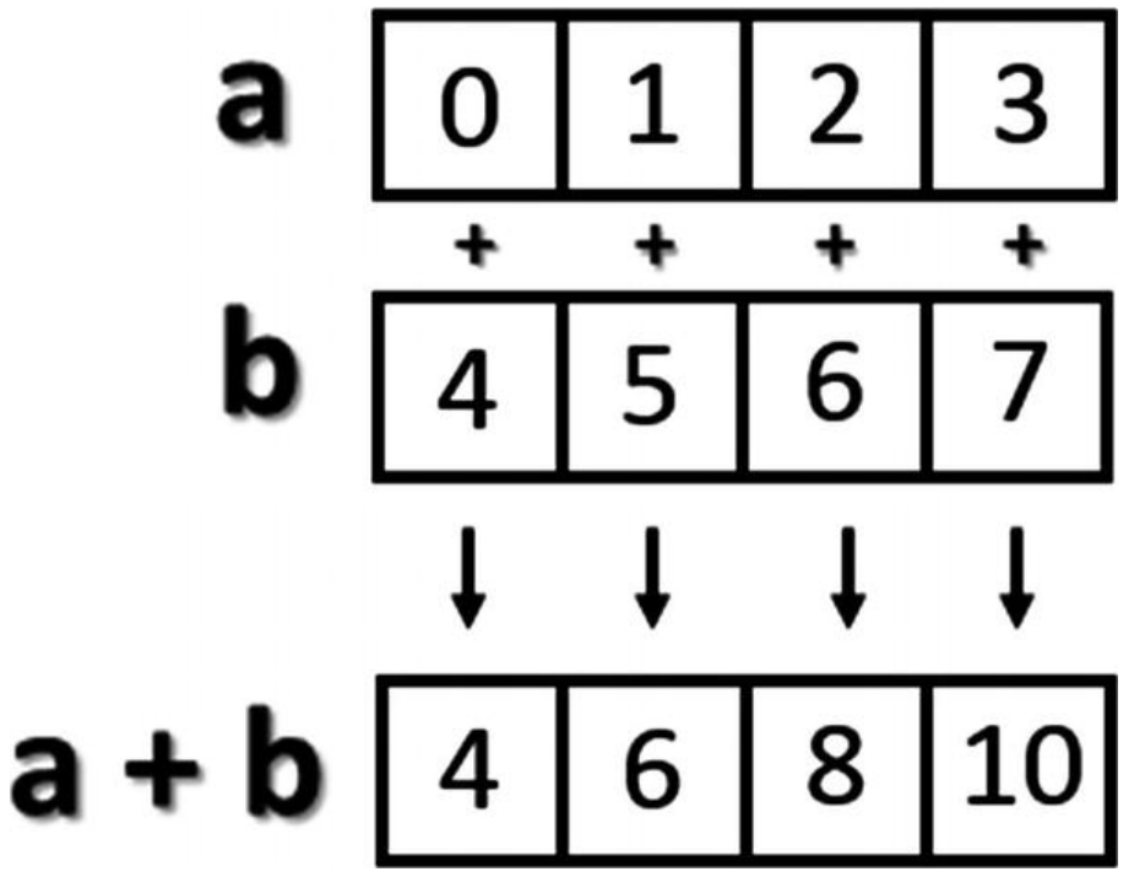
لتغيير المصفوفة السابقة إلى شكل two-dimensional arrays نستخدم دالة reshape أو خاصية shape.

```
a.reshape(3, 4)
a.shape = (3, 4)
```

نرى هنا طريقتين لتغير shape للمصفوفة, عن طريق استخدام reshape() و وضع الحجم الذي نريد تغير المصفوفة اليه, او بستخدام الطريقة الثانيه shape = ().

العمليات على المصفوفات

- العمليات الرياضية (Arithmetic Operators).




```
a = np.arange(4)
```

```
a+4
a*2
b = np.arange(4,8)
a + b
np.add(a, b)
a - b
np.subtract(a,b)
a * b
np.multiply(a, b)
a / b
np.divide(a,b)
```

لدنيا هنا اكثر من مثال ,تم إنشاء مصفوفة a بهذا الشكل array([0, 1, 2, 3])، لنرى كيف نطبق العمليات على هذه الـarray.
في a+4 هنا سوف يجمع 4 مع كل عنصر في هذه المصفوفة, و a*2 سوف يضرب 2 في كل عنصر ايضاً .

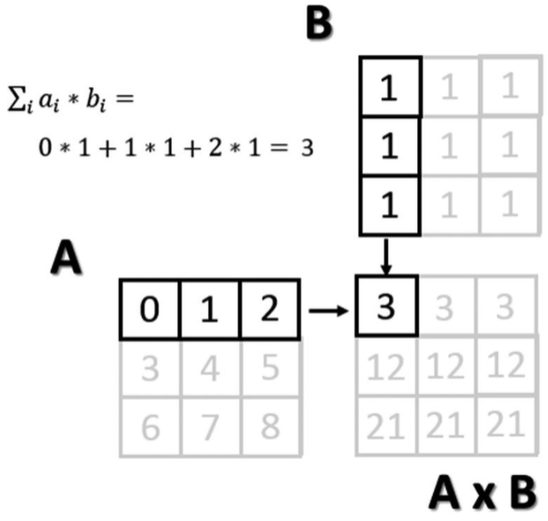
ايضاً توفر لنا numpy دوال للعمليات الرياضية للمصفوفات ,تم إنشاء مصفوفة b بهذا الشكلarray([4, 5, 6, 7])،
فمثلاً بشكل مباشر نستخدم هذه دالة add() للجمع بين المصفوفتين و سوف ينتج مصفوفه جديدة وهي ناتج الجمع
فيكون ناتج جمع a,b يساوي array([4, 6, 8, 10]).

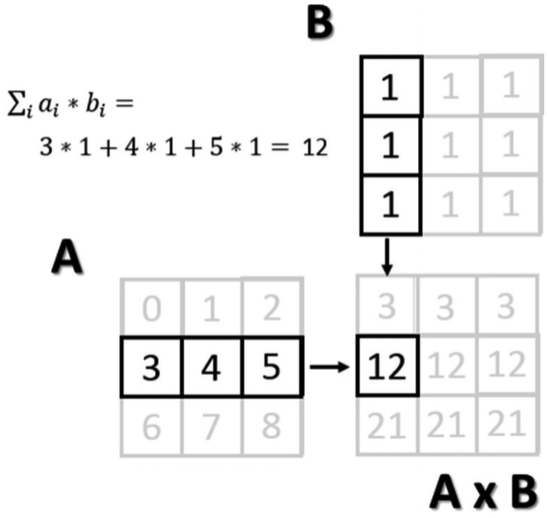
- ضرب المصفوفات (The Matrix Product).

```
A = np.arange(0, 9).reshape(3, 3)
B = np.ones((3, 3))
A * B
A.dot(B)
np.dot(A,B)
np.dot(B,A)
```

تم إنشاء مصفوفتين A , B جميعهم من نوع 2D ,لنرى الان طرق ضرب المصفوفات :
اما مباشرة A * B او عن طريق dot() طبعاً الطرق هذه تختلف عن (np.multiply(A, B) , في multiply() يقوم بضرب مصفوفتين حسب العنصر , اما dot()هو حاصل الضرب القياسي لمصفوفتين.

توضح لنا الصورة ادناه طريقة الضرب القياسي لمصفوفتين.





- عمليات (Increment and Decrement)

```
a = np.arange(4)
a += 1
a -= 1
a += 4
a *= 2
```

هنا تم إنشاء مصفوفة من نوع 1D , سوف نطبق عمليات Increment and Decrement على هذه الـarray, نرى اول عملية هنا a += 1 سوف تقوم بزيادة 1 لكل عنصر , و هنا a -=1 العكس سوف تنقص 1 من كل عنصر , a += 4 هنا سوف تقوم بزيادة 4 لكل عنصر , a *= 2 سوف تضرب كل عنصر في 2.

- دوال (Universal Functions (ufunc))

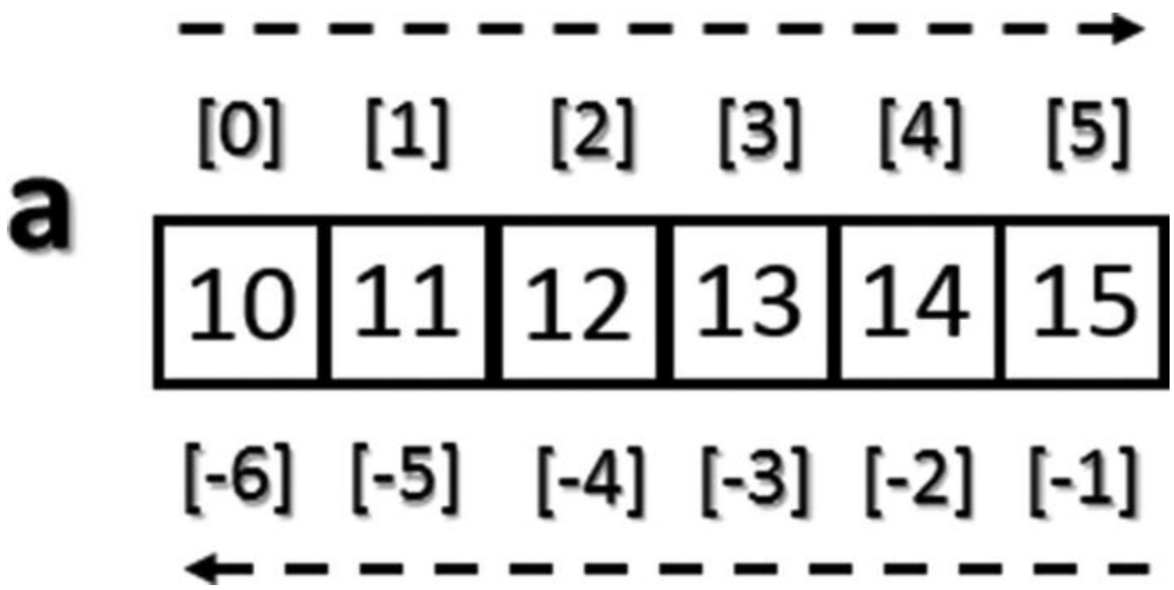
```
a = np.arange(1, 5)
np.sqrt(a)
np.log(a)
np.sin(a)
np.exp(a)
```

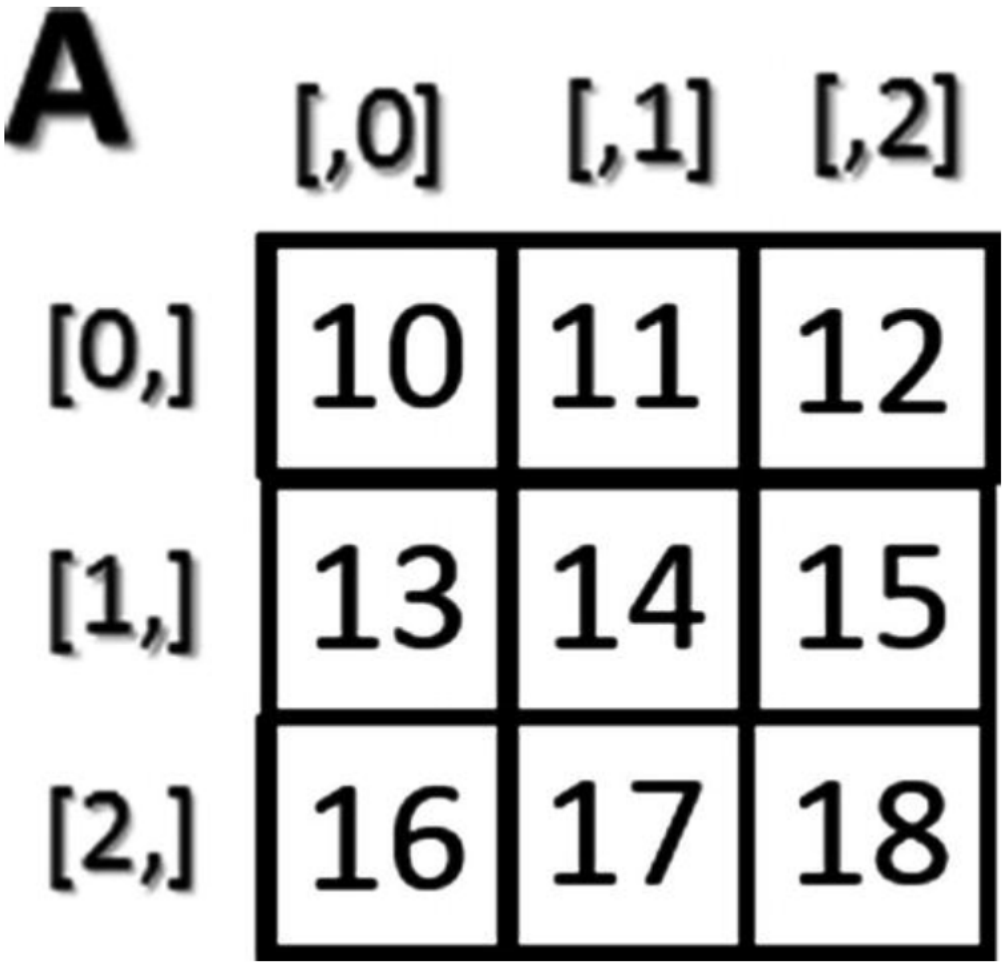
- دوال (Aggregate Functions)

```
a = np.array([3.3, 4.5, 1.2, 5.7, 0.3])
a.sum()
```

```
a.min()
a.max()
a.mean()
a.std()
```

التعامل مع (Indexing, Slicing, and Iterating):





أولا: Indexing

```
a = np.arange(10, 16)
a
a[4]
print(a[-1])
print(a[-6])
a[[1, 3, 4]] # two square brackets [[]]
A = np.arange(10, 19).reshape((3, 3))
A[1, 2]
A[2, 0]
A[2][0]
```

تم إنشاء مصفوفة بهذه الشكل array([10, 11, 12, 13, 14, 15]) , حين نريد ان نصل الى عنصر محدد في المصفوفة نصله بإستخدام الـindex, في المثال اعلاه لنفترض اننا نريد الوصول الى العنصر 14 نحدد الـindex لهذا العنصر وهو 4 فـa[4] سوف تطبع العدد الموجود في هذا الـindex وهو 14, اما حين نريد طباعة اخر عنصر في المصفوفة اياً كانت هذه المصفوفة فسوف نحدد الـindex انه 1- حيث ان 1- يعني اخر عنصر في المصفوفة فـ a[-1] تطبع اخر عنصر وهو 15, الان لنفترض اننا نريد طباعة اول index وهو 10 وهو يمثل 0 و يمثل -6 بالعكس فـ a[-6] و a[0] جميعهم يطبعوا 10.

اما في المصفوفة الاخرى فهي من نوع 2D وحين نريد نحدد index فلابد من two indexs لـrow و لـcolumn , نرى هنا ناتج المصفوفة A

```
array([[10, 11, 12],
       [13, 14, 15],
       [16, 17, 18]])
```

لنفترض اننا نريد الوصول الى العنصر 15 فسوف نحدد الصف 1 و العمود 2 فـ A[1, 2] سوف تطبع 15 , نريد الان العنصر 16 فنحدد الصف 2 و العمود 0 للوصول الى للعنصر 16 فـA[2, 0] سوف تطبع 16 , ايضا نستطيع كتابتها بهذا الشكل A[2][0] و سوف تعطينا نفس الناتج.

ثانيا: Slicing

```
a = np.arange(10, 16)
a[1:5] #[start index, final index]
a[1:5:2]  #[start index, final index, gap]
a[1:5:3]
a[::2] #[start index =0, final index = maximum index, gap = 2]
a[::3] #[start index =0, final index = maximum index, gap = 3]
a[:5:2] #[start index =0, final index = 5, gap = 2]
a[:5:] #[start index =0, final index = 5, gap = 1]
# two-dimensional array
A = np.arange(10, 19).reshape((3, 3))
A[0,:] #[row=0 , column=all]
A[:,0] #[row=all , column=0]
A[0:2, 0:2]
# not contiguous indexes >> specify an array of indexes.
A[[0,2], 0:2]
```

الفكرة الاساسية من slicing هي طباعة جزء معين من المصفوفة, في المصفوفة اعلاه ([array([10, 11, 12, 13, 14, 15])
لنفترض نريد طباعة 11و12و13و14 فقط فسوف نحدد index البداية و index النهاية فنريده يطبع من 1 الى 4 فـ[a[1:5] هنا تم تحديد *[start index, final index] و ايضاً نستطيع هنا استخدام gap ,في نفس المثال لنفترض اننا نريد طباعة* 11و12و13و14 لكن سوف نضيف الـgap و نريد تحديده بـ[a[1:5:3 3 سوف يكون الناتج ([array([11, 14]) .

لنرى الآن slicing لكن مع 2D مع المصفوفة A:

```
array([[10, 11, 12],
       [13, 14, 15],
       [16, 17, 18]])
```

لنفترض اننا نريد طباعة الصف الاول فقط من هذه المصفوفة , الصف الاول هنا هو في الـindex الذي يساوي 0 و جميع الاعمدة لاننا نريد طباعة كامل ف سوف نقول [A[0,:] و تعني الصف 0 و جميع الاعمدة فيكون الناتج ([array([10, 11, 12]).

اما في حين اننا نريد طباعة هذه الجزء فقط من المصفوفة A :

```
array([[10, 11],
       [13, 14]])
```

هنا نرى ان لدينا صفيين و عمودين اي اننا سوف نطبع من الصف 0 الى الصف 1 و من العمود 0 الى العمود 1 فـ يكون الحل [A[0:2, 0:2] حيث هنا تم تحديد الـindex لكل من rows , columns .

ثالثا: Iterating

استخدام loop في Python

```
for i in a:
    print(i)
```

بكل بساطة i سوف تمشي في الـarray التي هي a و في كل مرة تطبع العنصر , لكن نلاحظ ان a هنا من نوع 1D.

استخدام loop في Python مع two-dimensional array

```
for row in A:
    for item in row:
        print(item)
```

هنا الوضع مختلف بالنسبة لـ2D نحتاج هنا لكتابة for متداخلة واحدة لـ rows و الاخرى لـ items داخل كل row لنطبق على نفس المصفوفة A:

```
array([[10, 11, 12],
       [13, 14, 15],
       [16, 17, 18]])
```

سوف نعرف الـrow الاولى لتمشي على الصفوف و for اخرى لطباعة عناصر كل صف, في المثال اعلاه row سوف تكون تؤشر على الصف الاول وهو 0 ثم يتم إنشاء for اخرى بداخل الـfor الخاصة بـrow تقوم بتأشير على العناصر الصف الذي يؤشر عليه المؤشر row في الـfor الاولى و يطبع عناصر الصف الاول , و تستمر هكذا الى نهاية المصفوفة.

```
for item in A.flat:
    print(item)
```

هنا حل اخرى يقوم بطباعة عناصر المصفوفة 2D لكن بإختصار بإستخدام flat وهي تقوم بتسوية المصفوفة و هي عملية تقليل أبعاد المصفوفة فـA هنا كانت 2D مع استخدام A.flat تقل ابعادها و تصبح 1D .

استخدام loop في Numpy

```
# three arguments: the aggregate function, the axis on which t
o apply the iteration, and the array
# axis = 0(columns), axis = 1(rows)
np.apply_along_axis(np.mean, axis=0, arr=A)

np.apply_along_axis(np.mean, axis=1, arr=A)

def f2(x):
    return x/2

np.apply_along_axis(f2, axis=1, arr=A)
```

يوفر لنا numpy ايضاً apply_along_axis() وهي تقوم بتطبيق دالة على محور واحد فقط ,فهي تحتوي على الدالة التجميعية والمحور الذي سيتم تطبيق التكرار عليه والمصفوفة. هنا *(axis = 0(columns), axis = 1(rows* .
لكن ليش شرط ان تحتوي على دالة تجميعية (*aggregate function*) , فالإمكان ان تجتوي على function , في المثال اعلاه تم تعريف function وظيفتها تقسم العدد على 2 و نريد تطبيقها على جميع الصفوف لـA فقط بإستخدام apply_along_axis().

Conditions and Boolean Arrays

```
A = np.random.random((4, 4))
A
```

تم إنشاء مصفوفة عشوائية من نوع 2D :

```
array([[0.60350058, 0.48783745, 0.57010453, 0.5161902 ],
       [0.5346373 , 0.89991589, 0.65187429, 0.05345266],
       [0.42352431, 0.15311693, 0.82198236, 0.85773852],
       [0.62937089, 0.9137966 , 0.9897958 , 0.93737214]])
```

هنا لدينا شرط نريد فقط القيم التي اقل من 0.5 :

```
# you wanted to select all the values < 0.5
A < 0.5
```

نلاحظ ان الناتج هذا الشرط مصفوفة مكونه من False و True , فهو قام بإرجاع True للقيم التي حققت الشرط و False للقيم التيم لم تحقق الشرط .

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

لكن نحن نريد إرجاع القيم التي تحقق هذا الشرط, فنقوم بإستدعاء المصفوفه A مع الشرط السابق بهذا الشكل :

```
A[A < 0.5]
```

فنرى هنا انه تم إرجاع القيم التي حققت الشرط وليش false , true.

```
array([0.48783745, 0.05345266, 0.42352431, 0.15311693])
```

Array Manipulation

First: Joining Arrays

```
A = np.ones((3, 3))
A
```

تم إنشاء مصفوفة من نوع 2D جميع عناصرها 1:

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

```
B = np.zeros((3, 3))
```

```
B
```

وهنا تم إنشاء مصفوفة من نوع 2D جميع عناصرها 0 :

```
array([[0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.]])
```

نريد الان دمج المصفوفة A مع المصفوفة B بشكل رأسي , توفر numpy دوال عدة للدمج و من ضمنهم vstack(),
فنقوم بشكل مباشر تمرير المصفوفتين المراد دمجهم داخب vstack().

```
np.vstack((A, B)) # vertical stacking
```

```
array([[1., 1., 1.],
       [1., 1., 1.],
       [1., 1., 1.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.]])
```

نفس المثال السابق لكن نريد دمجهم بشكل افقي و ليس رأسي, ايضاً وفرت numpy ذلك عن طريق hstack().

```
np.hstack((A,B)) # horizontal stacking
```

ناتج الدمج الافقي للمصفوفتين :

```
array([[1., 1., 1., 0., 0., 0.],
       [1., 1., 1., 0., 0., 0.],
       [1., 1., 1., 0., 0., 0.]])
```

هنا تم إنشاء ثلاث مصفوفات من نوع 1D في كل من a,b,c .

```
a = np.array([0, 1, 2])
b = np.array([3, 4, 5])
c = np.array([6, 7, 8])
```

نريد دمج الثلاث مصفوفات لإنتاج مصفوفة جديدة من نوع 2D.

```
# np.column_stack and np.row_stack used with one-dimensional a
rrays
# which are stacked as columns or rows in order to form a new
two-dimensional array.
np.column_stack((a, b, c))
```

يوجد طريقتين و جميعهم ينتجون الحل ذاته , و هما *np.column_stack* و *np.row_stack* *يتم إستخدامهم مع المصفوفات من نوع 1D فقط لإنتاج مصفوفه جديدة من نوع 2D.*
يقومون بـstacking الاعمدة او الصفوف من أجل تكوين مصفوفة ثنائية الأبعاد جديدة.

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

نرى هنا row_stack وقمنا بتمرير المصفوفات الاحادية بداخلها.

```
np.row_stack((a, b, c))
```

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

Second: Splitting Arrays

```
A = np.arange(16).reshape((4, 4))
A
```

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

بعد ما تم إنشاء مصفوفة (4,4)من نوع 2D, نريد ان نقسمها المصفوفتين بشكل افقي لينتج لدنيا مصفوفتين بحجم (4,2) لكل مصفوفة.

```
[B,C] = np.hsplit(A, 2) #split the array horizontally
B # the 4x4 matrix A will be split into two 2x4 matrices.
```

نقوم بذلك بإستخدام hsplit() و نمرر بداخلها المصفوفة المراد فصلها و الأقسام بما أن الفصل افقي هو سوف يقسم الاعمدة ,سيتم تقسيم المصفوفة 4 × 4 إلى مصفوفتين 2 × 4, نرى هنا B:

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

```
C
```

و نرى هنا C:

```
array([[ 2,  3],
       [ 6,  7],
       [10, 11],
       [14, 15]])
```

نفس المثال السابق و لكن تقسيم المصفوفة عموديا, يكون بإستخدام vsplit() و بما انه عامودي سوف يقسم الصفوف ,مشابة جداً للمثال السابق hsplit() , و الاختلاف هنا سيتم تقسيم المصفوفة 4 × 4 إلى مصفوفتين 4 × 2:

```
[B,C] = np.vsplit(A, 2) #split the array vertically
B
```

نرى هنا B:

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

```
C
```

نرى هنا C:

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

```
# split() function, which allows you to split the array into n
onsymmetrical parts.
# arguments(array, indexes, axis=1(columns) and axis=0(rows))
[A1,A2,A3] = np.split(A,[1,3],axis=1)
A1
```

حين نريد تقسيم المصفوفة أجزاء غير متماثلة نستخدم split (), نقوم بتمرير المصفوفة المراد تقسيمها و الـindex و المحور نقصد هنا *(axis=1(columns) and axis=0(rows* فقط.

في المثال اعلاه تم تحديد المصفوفة A و الـ1,3] index] و المحور 1 يمثل الاعمدة , فسوف يقسم المصفوفة الى ثلاث مصفوفات بشكل غير متماثة .

```
array([[ 0],
       [ 4],
       [ 8],
       [12]])
```

```
A2
array([[ 1,  2],
       [ 5,  6],
       [ 9, 10],
       [13, 14]])
```

```
A3
array([[ 3],
       [ 7],
       [11],
       [15]])
```

نفس المثال السابق و لكن تم تغير المحور من 1 الى 0 و نقصد بالصفر الصفوف هنا .

```
[A1,A2,A3] = np.split(A,[1,3],axis=0)
A1
array([[0, 1, 2, 3]])
```

```
A2
array([[ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

```
A3
array([[12, 13, 14, 15]])
```

Copies or Views of Objects

Views

```
a = np.array([1, 2, 3, 4])
b = a
b
```

يتم إنشاء مصفوفة من نوع 1D في المتغير a و تم إنشاء متغير b يساوي a , لا يُنشئ نسخة مكررة من المصفوفة الرئيسية a هو يقوم فقط بإنشاء مرجع للمصفوفة الأصلية.

```
array([1, 2, 3, 4])
```

```
a[2] = 0
print(a)
print(b)
[1 2 0 4]
[1 2 0 4]
```

هنا تم تغير قيمة عنصر بإستخدام الـindex فـa[2] = 0 اي تم تغير قيمة index 2 الى 0 في المصفوفة a , سيتم ايضاً تغيره مباشرة في b ايضاً , لذلك ، إذا تم تغيير أي قيمة إلى المصفوفة الأصلية ، فإنها ستغير قيمة المصفوفة المنسوخة أيضًا.

```
c = a[0:2]
c
```

هنا سوف تنتج مصفوفة c التي هي جزء من مصفوفة a , فهنا استخدمنا slicing .

```
array([1, 2])
```

```
a[0] = 0
print(a)
print(c)
[0 2 0 4]
[0 2]
```

Copies

```
a = np.array([1, 2, 3, 4])
c = a.copy()
print(a)
print(c)
[1 2 3 4]
[1 2 3 4]
```

هنا بإستخدام copy() سوف يتم نسخ عناصر المصفوفة a في c .

```
a[0] = 0
```

```
print(a)
print(c)
[0 2 3 4]
[1 2 3 4]
```

لكن حين يتم تغير عنصر في a لا تتأثر به المصفوفة c ,إذا تم تعديل أي قيمة في المصفوفة الأصلية أو المنسوخة ، فلن يحدث أي تغيير على مصفوفة أخرى

مصادر إضافية

- **مكتبة Numpy.**

https://numpy.org/doc/stable/user/absolute_beginners.html

- **معلومات إضافية عن أنواع البيانات.**

https://numpy.org/doc/stable/reference/arrays.dtypes.html#arrays-dtypes

- **ملخص مكتبة Numpy ا NumPy Cheat Sheet.**

http://datacamp-community-prod.s3.amazonaws.com/ba1fe95a-8b70-4d2f-95b0-bc954e9071b0

- **مقدمة عن مكتبة Numpy.**

https://numpy.org/doc/stable/user/whatisnumpy.html

- **كتاب Python Data Analytics: With Pandas, NumPy, and Matplotlib By Fabio Nelli.**

https://learning.oreilly.com/library/view/python-data-analytics/9781484239131/

- **كتاب Python for Algorithmic Trading By Yves Hilpisch.**

https://learning.oreilly.com/library/view/python-for-algorithmic/9781492053347/