Introduction

Bases nécessaires à l'utilisation de fonctions universelles

- **Les fonctions universelles** permettent de travailler sur des ndarray élément par élément (Elementwise functions).
- **Documentation Numpy sur les fonctions universelles (ufunc) :** https://numpy.org/doc/stable/reference/ufuncs.html
- Le premier type de fonction évoqué sur la documentation de Numpy sont celles de type "**Broadcasting**". Nous allons commencer par étudier celles-ci.

Broadcasting

Le broadcasting (diffusion) est utilisée dans Numpy pour gérer des opérations avec des **ndarray de dimensions différentes.**
Commençons par instancier deux vecteurs de dimensions respectives (1,4), et (3,1). Vous pouvez constater que nous avons un vecteur ligne et un vecteur colonne.
Nous allons également instancier une **matrice 3x4**.

```
vect_ligne = np.array([1,2,3,4])
vect_colonne = np.array([[1], [2], [3]])
matrice = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
print("Dimensions du vecteur ligne :", vect_ligne.shape)
print("Dimensions du vecteur colonne :", vect_colonne.shape)
print("Dimensions de la matrice :", matrice.shape)
```

```
Dimensions du vecteur ligne : (4,)
Dimensions du vecteur colonne : (3, 1)
Dimensions de la matrice : (3, 4)
```

Si on essaie de procéder à une addition entre le vecteur ligne et la matrice, voici ce que l'on obtient.

```
vect_ligne + matrice
```

```
array([[ 2,  4,  6,  8],
       [ 6,  8, 10, 12],
       [10, 12, 14, 16]])
```

Le vecteur ligne vect_ligne est converti (broadcast) en une matrice de dimension 3x4 ressemblant à ceci np.array([1,2,3,4],[1,2,3,4],[1,2,3,4]).

Chaque **ligne de la matrice est ensuite additionnée avec le vecteur ligne**.
L'opérateur + est l'équivalent de la méthode np.add(), qui permet de faire un broadcast des ndarray de dimensions différentes.

```
np.add(vect_ligne, matrice)
```

```
array([[ 2,  4,  6,  8],
       [ 6,  8, 10, 12],
       [10, 12, 14, 16]])
```

Vous voyez que cela donne le **même résultat**.

Lorsqu'on effectue un broadcast, l'une des dimensions des deux ndarray doit être similaire pour que celui-ci ait lieu et qu'une opération élément par élément soit effectuée.
On peut tester une autre addition entre **notre vecteur colonne et notre matrice**.

```
vect_colonne = np.array([[1], [2], [3]])
matrice = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
np.add(vect_colonne, matrice)
```

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

Ici, l'opération **élément par élément** s'est effectuée **colonne par colonne**. Le broadcast a pu se faire car la matrice et le vecteur colonne possède **le même nombre de lignes**.
Le vecteur colonne est devenue une **matrice 3x4** (3 lignes et 4 colonnes), comme ceci : **np.array([1,1,1,1], [2,2,2,2], [3,3,3,3])**

Addition d'un vecteur ligne et d'une matrice

Addition du vecteur colonne et d'une matrice

Les règles du broadcasting

Les règles du broadcasting

- Les ndarrays ont tous **la même shape**
- Les ndarrays ont tous le **même nombre de dimension et la longueur de chaque dimension doit être soit similaire soit égale à 1.**
- Les ndarrays qui n'ont **pas assez de dimension** peuvent se voir **ajouter une dimension de longueur 1** pour satisfaire la deuxième propriété.

La notion de casting

Opérer un cast, c'est changer le type de tous les éléments d'un array.

Conversion d'int en float

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

```
print(vecteur.dtype)
print(matrice.dtype)
resultat = np.add(vecteur, matrice, dtype=np.float32)
print(resultat)
```

```
int32
int32
[[ 2.  4.  6.  8.]
 [ 6.  8. 10. 12.]]
```

Un point représentant la virgule des nombres décimaux a été ajouté à chacune des valeurs.

Conversion d'int en string

```
vecteur = np.array([1,2,3,4])
matrice = np.array([[1,2,3,4], [5,6,7,8]])
np.add(vecteur, matrice, dtype=np.str)
```

```
<ipython-input-26-d70938593b28>:4: DeprecationWarning: `np.str` is a deprecated alias for the builtin `str`. To silence this warning, use `str` by itself. Doing this will not modify any behavior and is safe. If you specifically wanted the numpy scalar type, use `np.str_` here.
Deprecated in NumPy 1.20; for more details and guidance: https://numpy.org/devdocs/release/1.20.0-notes.html#deprecations
  np.add(vecteur, matrice, dtype=np.str)
---------------------------------------------------------------------------
UFuncTypeError                            Traceback (most recent call last)
<ipython-input-26-d70938593b28> in <module>
      2 matrice = np.array([[1,2,3,4], [5,6,7,8]])
      3
----> 4 np.add(vecteur, matrice, dtype=np.str)

UFuncTypeError: ufunc 'add' did not contain a loop with signature matching types (dtype('<U'), dtype('<U')) -> dtype('<U')
```

Une erreur est levée car **numpy essaie d'abord de convertir tous les éléments des deux ndarrays en chaîne de caractères, et ensuite il essaie de procéder à l'addition**. Ce qui ne fonctionne pas, puisqu'on ne peut pas additionner des chaînes de caractères dans un ndarray.

Utiliser une fonction sur un axe spécifique

Créons une matrice de dimension 2x5.

```
matrice = np.array([[1,2,3,4,5], [6,7,8,9,10]])
```

A présent, chercheons à calculer la moyenne de chaque colonne de la matrice. On va utiliser la méthode np.mean (mean en anglais signifie moyenne), et on va lui passer en paramètre axis=0.

```
np.mean(matrice, axis=0)
```

```
array([3.5, 4.5, 5.5, 6.5, 7.5])
```

Ici on se retrouve avec une seule ligne, et **chaque colonne contient la moyenne des valeurs de chaque colonne de la matrice**.
résultat = [(1+6) / 2 = 3.5, (2+7) / 2 = 4.5, (3+8) / 2 = 5.5, (4+9) / 2 = 6.5, (5+10) / 2 = 7.5]

Essayons avec axis = 1.

```
np.mean(matrice, axis=1)
```

```
array([3., 8.])
```

Cette fois, nous avons la moyenne pour chaque ligne de la matrice. Notre matrice avait 2 lignes, donc on a 2 moyennes de calculées.

Quelques exemples de fonctions universelles

La fonction np.negative()

Inverse le signe de tous les éléments d'un ndarray.

Scalaire

```
scalaire = np.array(3)
np.negative(scalaire)
```

```
-3
```

Vecteur

```
vecteur = np.array([-3,-2,-1,0,1,2,3])
np.negative(vecteur)
```

```
array([ 3,  2,  1,  0, -1, -2, -3])
```

Matrice

```
matrice = np.array([[400, 401, 402, 403], [404, 405, 406, 407]])
np.negative(matrice)
```

```
array([[-400, -401, -402, -403],
       [-404, -405, -406, -407]])
```

Les signes ont bel et bien tous été inversés. Les nombres positifs sont devenus négatifs et les nombres négatifs sont devenus positifs.

La fonction np.power()

Le ndarray passé en premier paramètre est mis à la puissance du ndarray passé en second paramètre.

Scalaire et scalaire

```
scalaire = np.array(3)
puissance = np.array(3)
np.power(scalaire, puissance)
```

27

Vecteur et scalaire

```
vecteur = np.array([1,2,3,4,5,6,7,8,9,10])
puissance = np.array(2)
np.power(vecteur, puissance)
```

```
array([  1,   4,   9,  16,  25,  36,  49,  64,  81, 100], dtype=int32)
```

Le carré de toutes les valeurs contenues dans le vecteur ont été calculées.

Vecteur et vecteur

Note : pensez à respecter les règles du broadcasting

```
vecteur = np.array([1,2,3,4,5,6,7,8,9,10])
puissance = np.array([1,2,1,2,1,2,1,2,1,2])
np.power(vecteur, puissance)
```

```
array([  1,   4,   3,  16,   5,  36,   7,  64,   9, 100], dtype=int32)
```

Toutes les valeurs du vecteur ont été mis à la puissance des valeurs contenues dans le vecteur puissance élément par élément.

Matrice et scalaire

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

```
array([[  1,   8,  27],
       [ 64, 125, 216],
       [343, 512, 729]], dtype=int32)
```

Toutes les valeurs contenues dans la matrice ont été calculées à la puissance 3.

Matrice et vecteur

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

```
array([[  1,   4,  27],
       [  4,  25, 216],
       [  7,  64, 729]], dtype=int32)
```

Un broadcast est opéré sur le vecteur pour le transformer en matrice [[1,2,3], [1,2,3], [1,2,3]].
Ensuite chaque ligne de la matrice est parcourue, et chacune des valeurs de ces lignes est mis à la puissance de la valeur à l'indice correspondant dans la matrice de puissance.
Pour résumer, tous les éléments de la colonne 0 ont été mis à la puissance 1, la colonne 1 à la puissance 2 et la colonne 2 à la puissance 3.

Matrice et matrice

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

```
array([[       1,        4,       27],
       [     256,     3125,    46656],
       [  823543, 16777216, 387420489]], dtype=int32)
```

La fonction np.conjugate()

Scalaire

```
z1 = np.array(3 + 2j)
np.conjugate(z1)
```

```
(3-2j)
```

Vecteur

```
z1 = np.array(1+2j)
z2 = np.array(0-1j)
z3 = np.array(3)
z4 = np.array(-1-9j)
vc = np.array([z1, z2, z3, z4])
np.conjugate(vc)
```

```
array([ 1.-2.j,  0.+1.j,  3.-0.j, -1.+9.j])
```

Le conjugué de chaque élément a été calculé.

Matrice

```
z1 = np.array(2+3j)
z2 = np.array(0-3j)
z3 = np.array(1)
z4 = np.array(-10-2j)
matrice = np.array([[z1, z2], [z3, z4]])
np.conjugate(matrice)
```

```
array([[  2.-3.j,   0.+3.j],
       [  1.-0.j, -10.+2.j]])
```

La fonction np.around()

Permet d'arrondir les valeurs dans un ndarray avec un certain nombre de chiffres après la virgule.

Testons avec un ndarray de dimension 1

```
vect_a = np.array([1.3245, 3.2168, 9.8541, 6.3210, 1.0234, 7.850])
```
Arrondissons chaque valeur à 2 chiffres après la virgule grâce au paramètre decimals.

```
np.around(vect_a, decimals=2)
```

```
array([1.32, 3.22, 9.85, 6.32, 1.02, 7.85])
```

Testons avec un ndarray de dimension 2

```
mat_a = np.array([[1.213, 3.654, 6.654, 9.201, 4.403], [5.021, 5.558, 6.321, 7.013, 8.631]])
```

Arrondissons chaque valeur à 1 seul chiffre après la virgule

```
np.around(mat_a, decimals=1)
```

```
array([[1.2, 3.7, 6.7, 9.2, 4.4],
       [5. , 5.6, 6.3, 7. , 8.6]])
```