NUMPY ET LES TABLEAUX

À la fin de ce chapitre, je sais :

- ☞ distinguer les différentes structures de données
- ☞ choisir une structure de données adaptée à un algorithme
- ☞ importer la bibliothèque Numpy
- ☞ utiliser un tableau Numpy mono et multidimensionnel
- ☞ écrire des opérations élément par élément

Écrire un programme optimal en terme de complexité nécessite l'identification des structures de données utilisées très tôt dans le développement. En effet, le choix d'une structure de données plutôt qu'une autre, par exemple choisir une liste à la place d'un tableau ou d'un dictionnaire, peut rendre inefficace un algorithme selon le choix effectué.

Ce chapitre a pour but d'approfondir la définition des structures de données afin de permettre un choix éclairé. C'est pourquoi on définit d'abord ce qu'est un type de données abstrait en illustrant ce concept sur les listes, les tableaux, les piles et les files. Puis on fait le lien avec les implémentations possibles de ces types en structures de données, notamment en langage Python.

A Type abstrait de données et structure de données

■ **Définition 1 — Type abstrait de données (TAD).** Un type de données abstrait est une abstraction d'une structure de données qui ne se préoccupe pas de son implémentation sur une machine : sa structure interne est indiscernable, le type abstrait est vu de l'extérieur comme une boite noire.

Un TAD spécifie le quoi, c'est à dire le type de données contenues ainsi que les opérations qu'on faire dessus. Par contre, il ne spécifie pas comment dont les données sont stockées ni comment les opérations sont implémentées.

■ **Définition 2 — Structure de données.** Une structure de données est une mise en œuvre concrète d'un type abstrait, une implémentation d'un type abstrait sur dans un langage de programmation.

■ **Exemple 1 — Un entier.** Un entier est un TAD qui :

- contient une suite de chiffres [a] éventuellement précédés par un signe − ou +,
- fournit les opérations $+, -, \times, //, \%$.

Selon le langage, ce TAD entier est implémenté en machine par un type concret différent :

- `int` en Python,
- `Integer` ou `int` en Java,
- `char`, `short`, `int`, `uint`, `long int` en C,
- `int` en OCaml.

a. peu importe la base pour l'instant...

■ **Exemple 2 — Un booléén.** De la même manière, on peut définir un TAD qui désigne un booléen. Un booléen est un TAD qui :

- se note Vrai ou Faux,
- fournit les opérations logiques ET, OU, NON...

Selon le langage, ce TAD booléen est implémenté en machine par un type concret différent :

- `bool` valant `True` ou `False` en Python,
- `boolean` valant `true` ou `false` en Java,
- `bool` valant `1` ou `0` en C,
- `bool` valant `true` ou `false` en OCaml.

Un type abstrait de données est à une structure de donnée ce qu'un algorithme est à un programme. On spécifie un algorithme ou un type abstrait de données, mais on implémente un programme ou une structure de données.

Les exemples précédents de types abstrait de données étaient limités à des types simples. Mais il est possible de définir des types abstraits de données composés.

■ **Exemple 3 — Types abstraits de données composés.** Voici quelques types abstraits composés parmi les plus courants :

- liste,
- file,
- pile,
- arbre binaire,
- dictionnaire ou tableau associatif,
- ensemble,
- graphe.

R Il faut faire attention, car si les objets de type `list` en Python implémente le TAD liste, ce n'est pas la seule implémentation possible.

B TAD tableaux et listes

Pour les informaticiens, les listes et et les tableaux sont avant tout des TAD qui permettent de stocker de façon ordonnée des éléments.

■ **Définition 3 — TAD tableau.** Un TAD tableau représente une structure finie indiçable par des entiers. Cela signifie qu'on peut accéder à la lecture ou à l'écriture de n'importe quel élément directement en utilisant un indice, par exemple `t[3]`.

On peut décliner le TAD tableau de manière :

- statique : la taille du tableau est fixée, on ne peut pas ajouter ou enlever d'éléments.
- évolutive : la taille du tableau peut varier, on peut ajouter ou enlever des éléments. Dans ce cas, on parle de tableau dynamique.

■ **Définition 4 — TAD liste.** Un TAD liste représente une séquence finie d'éléments d'un même type qui possède un rang dans la séquence. Les données sont traitées séquentiellement, dans l'ordre du rang.

Un TAD liste est évolutif, c'est à dire qu'on peut ajouter ou enlever n'importe quel élément.

Les opérations sur un TAD liste sont :

- l'ajout ou suppression en début et/ou en fin de liste,
- l'accès à la fin de la liste.

La longueur d'une liste est le nombre d'éléments qu'elle contient. On dit qu'une liste est vide si elle ne contient aucun élément, sa longueur vaut zéro. Le début de la liste est désigné par le terme tête de liste (head), le dernier élément de la liste par la fin de la liste (tail).

Ce chapitre détaille les implémentations de trois du TAD : les tableaux (vus cette fois-ci comme structure de données concrète), les listes chaînées et les tableaux dynamiques (ou vecteurs, ou encore listes-tableaux).

C Implémentation des tableaux

a Implémentation d'un tableau statique

Dans sa version statique, un TAD tableau de taille fixe n est implémenté par un bloc de mémoire contiguë contenant n cases. Ces cases sont capables d'accueillir le type d'élément que contient le tableau.

Par exemple, pour un TAD tableau statique de cinq entiers codés sur huit bits, on alloue un espace mémoire de 40 bits subdivisés en cinq octets comme indiqué sur la figure 1. Dans

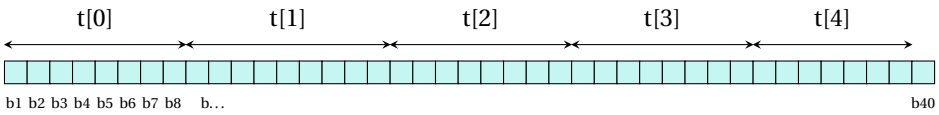

FIGURE 1 – Représentation d'un tableau statique en mémoire. Il peut représenter un tableau t de cinq entiers codés sur huit bits. On accède directement à l'élément i en écrivant $t[i]$.

la majorité des langages, l'opérateur [] permet alors d'accéder aux éléments, par exemple t[3]. Les éléments sont numérotés à partir de zéro : t[0] est le premier élément.

On peut estimer les coûts associés à l'utilisation d'un tableau statique comme le montre le tableau 1.

Opération	**Complexité**	**Raison**
Accès à un élément au début	$O(1)$	
Accès à un élément à la fin	$O(1)$	
Accès à un élément au milieu	$O(1)$	
Ajout d'un élément au début	$O(n)$	créer un nouveau tableau
Ajout d'un élément à la fin	$O(n)$	créer un nouveau tableau
Suppression d'un élément au début	$O(n)$	créer un nouveau tableau
Suppression d'un élément à la fin	$O(n)$	créer un nouveau tableau

TABLE 1 – Complexité des opérations associées à l'utilisation d'un tableau statique.

b Implémentation d'un tableau dynamique

Un tableau dynamique est implémenté par un tableau statique de taille n_{max} supérieure à la taille nécessaire pour stocker les données. Les n données contenues dans un tel tableau le sont donc simplement entre les indices 0 et $n-1$. Si la taille n_{max} n'est plus suffisante pour stocker toutes les données, on crée un nouveau tableau statique plus grand de taille kn_{max} et on recopie les données dedans.

Toute la subtilité des tableaux dynamiques réside dans la manière de gérer les nouvelles allocations mémoires lorsque le tableau doit être modifié.

R Les tableaux dynamiques sont parfois appelés vecteurs.

R Comme le montre le tableau 2, l'intérêt majeur du tableau dynamique est de proposer un accès direct constant comme dans un tableau statique tout en évitant les surcouts liés à l'ajout d'éléments.

Opération	Complexité	Raison
Accès à un élément au début	$O(1)$	
Accès à un élément à la fin	$O(1)$	
Accès à un élément au milieu	$O(1)$	
Ajout d'un élément au début	$O(n)$	décaler tous les éléments contigus
Ajout d'un élément à la fin	$O(1)$	amorti : il y a de la place ou pas
Suppression d'un élément au début	$O(n)$	décaler tous les éléments contigus
Suppression d'un élément à la fin	$O(1)$	amorti : il y a de la place, parfois trop

TABLE 2 – Complexité des opérations associées à l'utilisation d'un tableau dynamique.

R Le coût amorti signifie généralement le coût est constant. Mais, lorsqu'il n'y a plus de place, il faut bien créer la nouvelle structure adaptée au nombre d'éléments et cela a un coût linéaire $O(n)$. Donc ce coût amorti en $O(1)$ signifie c'est constant la plupart du temps mais que parfois cela peut être linéaire.

P En python le type `list` est implémenté par un tableau dynamique mais se comporte bien comme un TAD liste !

Cela a pour conséquence que :

- `L.pop()` et `L.append()` sont de complexité $O(1)$, donc supprimer ou ajouter en fin ne coûte pas cher,
- alors que `L.pop(0)` et `L.insert(0,elem)` sont de complexité $O(n)$ et donc supprimer ou ajouter en tête coûte cher.

Lorsqu'un algorithme doit supprimer ou ajouter en tête, il vaut mieux utiliser une autre structure de données qu'une `list` Python. Dans la bibliothèque `collections`, le type `deque` représente une liste sur laquelle les opérations d'ajout et de suppression en tête ou en fin sont en $O(1)$.

R Rechercher un élément dans un tableau statique ou dans un tableau dynamique présente donc une complexité dans le pire des cas linéaire en $O(n)$: il faut nécessairement balayer tous les éléments si l'élément recherché se trouve en dernière position.

★ D Implémentations des listes

a Listes simplement chaînées

Un élément d'une liste simplement chaînée est une cellule constituée de deux parties :

- la première contient une donnée, par exemple un entier pour une liste d'entiers,
- la seconde contient un pointeur, c'est à dire une adresse mémoire, vers un autre élément (l'élément suivant) ou rien.

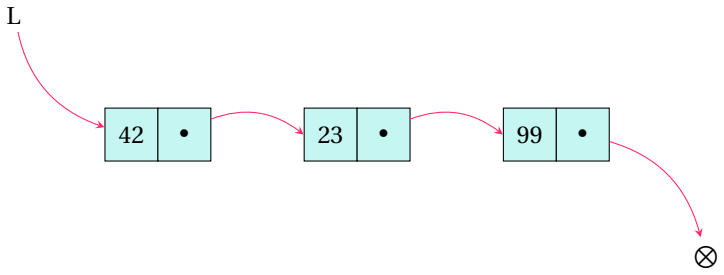

FIGURE 2 – Représentation d'une liste simplement chaînée d'entiers L. L pointe sur le premier élément de la liste. Le dernier pointeur ne pointe sur rien.

Une liste simplement chaînée se présente donc comme une succession d'éléments composites, chacun pointant sur le suivant et le dernier sur rien. En général, la variable associée à une liste simplement chaînée n'est qu'un pointeur vers le premier élément.

Opération	**Complexité**	**Raison**
Accès à un élément au début	$O(1)$	L pointe sur le premier élément
Accès à un élément à la fin	$O(n)$	accès séquentiel
Accès à un élément au milieu	$O(n)$	accès séquentiel
Ajout d'un élément au début	$O(1)$	L pointe sur le premier élément
Ajout d'un élément à la fin	$O(n)$	accès séquentiel
Suppression d'un élément au début	$O(1)$	L pointe sur le premier élément
Suppression d'un élément à la fin	$O(n)$	accès séquentiel

TABLE 3 – Complexité des opérations associées à l'utilisation d'une liste simplement chaînée.

b Listes doublement chaînées

Un élément d'une liste doublement chaînée est une cellule constituée de trois parties :

- la première contient un pointeur vers l'élément précédent,
- la deuxième contient une donnée,
- la troisième contient un pointeur vers l'élément suivant.

Une liste doublement chaînée enregistre dans sa structure un pointeur vers le premier élément et un pointeur vers le dernier élément. Ainsi on peut toujours accéder directement à la tête et à la fin de liste. Par contre, c'est un peu plus lourd en mémoire et plus difficile à implémenter qu'une liste simplement chaînée. Le tableau 4 recense les coûts associés aux opérations sur les listes doublement chaînées.

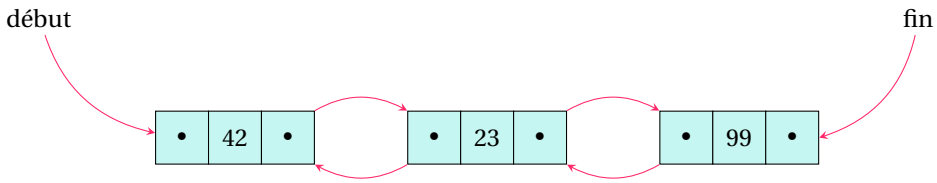

FIGURE 3 – Représentation d'une liste doublement chaînée d'entiers L. On conserve un pointeur sur le premier élément et un autre sur le dernier élément de la liste.

Opération	**Complexité**	**Raison**
Accès à un élément au début	$O(1)$	pointeur sur le premier élément
Accès à un élément à la fin	$O(1)$	pointeur sur le dernier élément
Accès à un élément au milieu	$O(n)$	accès séquentiel
Ajout d'un élément au début	$O(1)$	pointeur sur le premier élément
Ajout d'un élément à la fin	$O(1)$	pointeur sur le dernier élément
Suppression d'un élément au début	$O(1)$	pointeur sur le premier élément
Suppression d'un élément à la fin	$O(1)$	pointeur sur le dernier élément

TABLE 4 – Complexité des opérations associées à l'utilisation d'une liste doublement chaînée.

R La recherche d'un élément dans une liste simplement ou doublement chaînée présente donc une complexité dans le pire des cas linéaire en $O(n)$: il faut nécessairement balayer tous les éléments si l'élément recherché se trouve en dernière position.

E Bilan de complexités

Le tableau 5 propose une synthèses des coûts des opérations sur les structures listes et tableaux en fonction de leurs implémentations.

Opération	Tableau statique	Liste chaînée	Liste doublement chaînée	Tableau dynamique
Accès à un élément au début	$O(1)$	$O(1)$	$O(1)$	$O(1)$
Accès à un élément à la fin	$O(1)$	$O(n)$	$O(1)$	$O(1)$
Accès à un élément au milieu	$O(1)$	$O(n)$	$O(n)$	$O(1)$
Ajout d'un élément au début	$O(n)$	$O(1)$	$O(1)$	$O(n)$
Ajout d'un élément à la fin	$O(n)$	$O(n)$	$O(1)$	$O(1)$ amorti
Suppression d'un élément au début	$O(1)$	$O(1)$	$O(1)$	$O(n)$
Suppression d'un élément à la fin	$O(n)$	$O(1)$	$O(1)$	$O(1)$ amorti

TABLE 5 – Complexité des opérations associées à l'utilisation des listes et des tableaux.

F Pourquoi Numpy? --→ HORS PROGRAMME

P Numpy est un bibliothèque logicielle Python dédiée au calcul numérique. Elle n'est pas explicitement au programme. Certaines épreuves de concours en interdisent son usage quand d'autres l'exigent explicitement. Il faut rester **vigilant** à la lecture de l'épreuve.

STOP Le paragraphe précédent est important pour l'épreuve d'informatique!

Le cœur du langage Python ne dispose pas d'un type tableau **statique**, c'est-à-dire de taille fixe. Les concepteurs de ce langage ont créé un type hybride qui se comporte à la fois comme une liste et un tableau : la liste Python. L'implémentation est réalisée grâce à un tableau **dynamique**, c'est-à-dire un tableau dont la taille évolue en fonction du besoin au cours du programme. Néanmoins, cette évolution de taille a un impact sur la rapidité des opérations. La liste Python est très souple et (trop) pratique : on peut la tordre pour faire, au moins en apparence, à peu près ce que l'on veut, même si cela n'est pas toujours efficace. Cependant, elle n'est pas adaptée au calcul numérique : les listes imbriquées ne sont pas des tableaux multidimensionnels et trop de flexibilité rend le code propice à des bogues à répétitions difficiles à cerner.

Pour le calcul numérique, il est important de pouvoir utiliser des tableaux statiques pour représenter des vecteurs, desămatrices ou des tableaux à n dimensions car les opérations qu'on souhaite réaliser se font soit élément par élément (calcul vectoriel) soit par matriciellement. Or, ces calculs vectoriels ou matriciels ne sont cohérents que si les tailles des opérandes correspondent. La taille fixe permet de garantir la cohérence des calculs effectués : on ne peut pas par inadvertance ajouter un élément à un tableau statique. D'un point de vue machine, **l'espace mémoire alloué peut rester le même tout au long du calcul**, le chargement en mémoire est donc plus facile à gérer, les modifications et les accès plus simples et plus rapides.

Les plus-values de Numpy, dans ce contexte, sont les suivantes :

1. le type `array` est un tableau statique multidimensionnel. En machine, tous les éléments sont stockés de manière contigüe, à plat : l'aspect multidimensionnel n'est qu'une vi-

sion du tableau procurée par le génie logiciel, du sucre syntaxique. C'est pourquoi il est très facile de changer de redimensionner un tableau Numpy car il n'est pas modifié en mémoire par cette opération.

2. les types simples Numpy sont plus riches que les types de base Python : on peut par exemple utiliser des types entiers non signés (np.uint8) pour stocker une information sur un octet plutôt que d'utiliser un entier signé sur 64 bits, ou utiliser des flottants simple précision plutôt que double. Gain de place, gain de temps également dans les opérations.
3. les calculs vectoriel (élément par élément) et matriciel proposés par Numpy présentent deux avantages :
   - les opérations vectorielles `+,-,*,/,...` sont pré-compilées en langage machine et donc le calcul est exécuté très rapidement par l'interpréteur Python car il n'est pas interprété. Les codes compilés utilisent les instructions vectorielles des processeurs, des instructions capables de réaliser plusieurs opérations du même type (addition par exemple) sur des opérandes différentes en un seul cycle d'horloge.
   - la syntaxe proposée via la surcharge d'opérateur permet de s'affranchir de l'utilisation des boucles et d'écrire une formule vectorielle avec la même syntaxe qu'une formule scalaire. C'est à cette condition d'écriture d'ailleurs que les calculs peuvent être accélérés par les opérations pré-compilées. Par exemple, ci-dessous, on a opéré une multiplication par deux pour chaque élément de X, puis une soustraction avec un élément de `B` : zéro pour la première colonne, un pour la seconde, deux pour la dernière.

     ```
     import numpy as np
     X = np.arange(9.0).reshape((3, 3)) # Matrice 3x3
     # array([[0., 1., 2.],
     #        [3., 4., 5.],
     #        [6., 7., 8.]])
     B = np.arange(3.0) # vecteur (1,3)
     # array([0., 1., 2.])
     Y = 2*X - B # matrice 3x3
     # résultat :
     #       array([[ 0.,  1.,  2.],
     #              [ 6.,  7.,  8.],
     #              [12., 13., 14.]])
     ```

   - L'opérateut `@` garantit que la syntaxe d'un calcul matriciel est identique à l'expression mathématique. Par exemple :

     ```
     import numpy as np
     A = np.array([[1, 2], [3, 4]])
     B = np.array([[5, 6], [7, 8]])
     produit = A @ B
     ```

Vocabulary 1 — Element-wise ⇜⇝ élément par element

Numpy est très utilisé par les scientifiques du monde entier. Les raisons principales sont les suivantes :

1. Numpy est open source,
2. Numpy est optimisée avec un cœur compilé en langage C (compilé, pas interprété donc plus rapide),
3. Numpy s'interface facilement avec d'autres langage de calcul scientifique (C et Fortran notamment),
4. Numpy procure les `array`, une structure de données de type tableau multidimensionnel de dimensions fixes. Cela complète habilement le langage Python qui ne propose que des listes qui sont implémentées par des tableaux dynamiques.
5. Numpy propose des opérations sur les tableaux élément par élément. Cela permet à la fois d'éviter d'écrire des boucles et de paralléliser les opérations.
6. Numpy permet l'écriture de calculs matriciels.
7. Ces deux derniers points font que les formules mathématiques apparaissent écrites quasi-naturellement dans le code.

G Importation

On peut importer comme on le désire la bibliothèque Numpy. Cependant une convention très utilisée fait qu'on l'importe souvent comme ceci :

```
import numpy as np

t = np.zeros(42)
```

H Créer des vecteurs, des matrices ou des tableaux

■ **Définition 5 — Vecteur numpy.** Le terme *vecteur* désigne des `array numpy` de dimension 1.

■ **Définition 6 — Matrice numpy .** Le terme *matrice* désigne les `array` numpy de dimension deux.

Lorsque le tableau possède plus de deux dimensions, on parle de tableau.

On peut créer, comme le code 1 le montre, un vecteur, une matrice ou un tablau numpy à partir :

- du constructeur np.array(),
- d'une liste ou d'une liste imbriquée,
- de fonctions spéciales en précisant les dimensions du tableau et la valeur d'initialisation des cases du tableau,
- d'un tableau existant : la tableau créé aura les mêmes dimensions. On précise la valeur à laquelle on veut initialiser les cases.

Code 1 – Créer des tableaux Numpy

```
import numpy as np

v1 = np.array([1, 2, 3])
print(v1.shape, v1)  # (3,) [1 2 3]
v2 = np.array([[10], [20], [30]])
print(v2.shape, v2)  # (3, 1) [[10] [20] [30]]

v3 = np.zeros((1, 3))
print(v3.shape, v3)  # (1, 3) [[0. 0. 0.]]
v4 = np.ones((1, 7))
print(v4.shape, v4)  # (1, 7) [[1. 1. 1. 1. 1. 1. 1.]]

m1 = np.zeros((5, 5))
print(m1.shape, m1)
# (5, 5)
# [[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.]]

m1 = np.ones((2, 3))
print(m1.shape, m1)
# (2, 3)
# [[1. 1. 1.]
# [1. 1. 1.]]

t0 = np.zeros((42, 21, 84, 3))
print(t0.shape)  # (42, 21, 84, 3)

t1 = np.zeros_like(v1)
print(t1.shape, t1)  # (3,) [0 0 0]
t2 = np.ones_like(v3)
print(t1.shape, t1)  # (1, 3) [[1. 1. 1.]]
t3 = np.full_like(m1, 42)
print(t1.shape, t1)
# (2, 3)
#  [[42. 42. 42.]
#  [42. 42. 42.]]

s = np.zeros((4, 4, 500)).shape
print(s[0], s[1], s[2])  # 4 4 500

r = np.arange(0, 1, 0.1) # like range but for float !
print(r)  # [0.  0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9]
time = np.linspace(0, 1, 11)
print(time) # [0.  0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1. ]
```

P Comme le montre la fin du code 1, la fonction `shape` renvoie un tuple qui spécifie les dimensions du tableau. Comme c'est un tuple, on peut accéder à la taille de chaque dimension par l'opérateur `[]`. Pour un matrice, par exemple, on obtient le nombre de lignes et le nombre de colonnes.

I Accéder aux éléments d'un tableau

Les cases d'un `array` numpy sont numérotées à partir de zéro, comme pour les listes Python. Le tronçonnage ainsi que l'indexage négatif sont également disponibles comme le montre le code 2.

P Cependant, la syntaxe de la manipulation des tableaux multidimensionnels est différente de celle des listes imbriquées : une virgule sépare les indices des dimensions différentes. Il faut rester vigilant et ne pas confondre ces syntaxes.

STOP Le paragraphe précédent est important pour l'épreuve d'informatique!

Code 2 – Accéder aux éléments d'un tableau numpy

```
import numpy as np

m = np.array([[1, 2], [3, 4]])
print(m)
# [[1 2]
#  [3 4]]
print(m[0, 0], m[0, 1], m[1, 0], m[1, 1])  # 1 2 3 4
# slicing
print(m[:, 0])  # [1 3]
print(m[:, 1])  # [2 4]
print(m[0, :])  # [1 2]
print(m[1, :])  # [3 4]

# negative indexing
print(m[-1, -1])
# reversing
print(m[::-1])
# assignment
m[0, :] = 42
print(m)
# [[42 42]
#  [ 3  4]]

# resizing
m = m.reshape((1, 4))
print(m)  # [[1 2 3 4]]
```

P Comme on peut le voir à la fin du code 2, on peut facilement redimensionner un tableau, avec ses éléments dedans. Cela vient du fait que numpy stocke toujours les éléments d'un tableau de manière contiguë en mémoire et ce, quelle que soit les dimensions du tableau.

La bibliothèque stocke aussi avec le tableau ses dimensions apparentes pour l'utilisateur. L'accès aux éléments au travers ses dimensions `t[3,2]` n'est donc qu'une commodité (du sucre syntaxique) fournie par le génie logiciel de la bibliothèque. Un indice réel est calculé par numpy à partir de [3,2] pour accéder à la case l'élément dans la zone contiguë en mémoire.

Vocabulary 2 — Syntactic sugar ⇜⇝ Sucre syntaxique. Élément de la syntaxe d'un langage visant à faciliter l'utilisation, l'écriture et la lecture associées à un concept. Cela adoucit le travail humain.

J Opérations élément par élément

On a très souvent besoin d'appliquer les mêmes formules à tous les éléments d'un tableau, à toute une série de données. Par exemple, pour filtrer un signal, le moyenner, pour changer de couleur tous les pixels d'une image ou tout simplement pour calculer tous le prix TTC de tous les éléments d'une liste de prix HT.

Le succès de numpy repose principalement sur la capacité à opérer des calculs sur un tableau comme si on les effectuait sur une variable scalaire (cf. code 3). **Il devient alors inutile de faire une boucle pour traiter tous les éléments du tableau** et l'écriture et la lecture des formules physiques et mathématiques s'en trouvent facilitées :

Code 3 – Opérer élément par élément

```
import numpy as np

x = np.ones(4)
print(x + x)  # [2. 2. 2. 2.]
print(x - x)  # [0. 0. 0. 0.]
print(4 * x * x / 5)  # [0.8 0.8 0.8 0.8]
print(x / x)  # [1. 1. 1. 1.]

r = np.arange(0, 1, 0.3)
print(r)  # [0.  0.3 0.6 0.9]
s = np.pi * r ** 2
print(s)  # [0. 0.28274334 1.13097336 2.54469005]

time = np.linspace(0, 1, 11)
print(time)  # [0.  0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1. ]
signal = 5 * np.cos(2 * np.pi * 7 * time)
print(signal)
# [ 5.         -1.54508497 -4.04508497  4.04508497  1.54508497 -5.
#   1.54508497  4.04508497 -4.04508497 -1.54508497  5.        ]
```

P Les opérateur arithmétiques `+,-,*,/,*` utilisés dans le code 3 sont des opérateurs qui agissent sur des tableaux numpy. En informatique, on dit alors qu'on a surchargé les opérateurs `+,-,*,/,*` pour qu'ils puissent agir sur d'autres données que les `int` ou les `float`.

K Opérations matricielles

Numpy possède également la multiplication matricielle comme le montre le code 4. C'est l'opérateur `@` qui permet de réaliser cette opération. Il est également très facile de faire de l'algèbre linéaire [1] avec le module `numpy.linalg`.

Code 4 – Calcul matriciel

```
import numpy as np

A = np.array([[1, 2], [3, 4]])
X = np.array([[.2], [.1]])
B = np.array([[.1, .2, .3], [.4, .5, .6]])
U = np.array([[0.1], [0.2], [0.3]])

Xp = A @ X + B @ U
print(Xp.shape)  # (2, 1)
print(Xp)
# [[0.54]
# [1.32]]
```

L Types de données

La bibliothèque numpy fournit également des types de données supplémentaires à Python. Ce sont en fait des types utilisés par le langage C qui sous-tend les calculs. On y trouve notamment les types `int` non signés désignés par `uint` ou les `float` sur 32 bits.

De nombreuses données concrètes sont représentées par des entiers non signés. C'est le cas par exemple des pixels d'une image dont on code généralement l'intensité sur huit bits, c'est à dire par une valeur comprise entre 0 et $2^8 - 1 = 255$. Numpy propose le type `np.uint8` pour représenter ce type de données.

Code 5 – Types de données numpy

```
import numpy as np

unsigned_integer = np.uint8(42)
print(unsigned_integer) # 42
image = np.zeros((1024, 512), dtype=np.uint8)
image[:] = 237
print(image)
#[[237 237 237 ... 237 237 237]
```

1. Calculer une matrice inverse par exemple.

```
# [237 237 237 ... 237 237 237]
# [237 237 237 ... 237 237 237]
# ...
# [237 237 237 ... 237 237 237]
# [237 237 237 ... 237 237 237]
# [237 237 237 ... 237 237 237]]
```

R Utiliser le type de données le plus adapté à la nature de la donnée est très important : cela permet d'économiser radicalement l'espace mémoire et d'accélérer les calculs.

M Autres fonctions

Numpy regorge de fonctionnalités. N'hésitez pas à consulter la documentation en ligne. Cette bibliothèque est complétée par `scipy`, bibliothèque scientifique.

- fonctions numpy standards à connaître : `min`, `max`, `std`, `mean`, `sum`,
- `unique` : permet de ne conserver qu'un seul exemplaire de chaque valeur dans un tableau,
- `argmax`, `argmin` : permet de récupérer l'indice du maximum ou du minimum d'un tableau,
- `concatenate` agrège deux tableaux,
- le module `numpy.random` et notamment `shuffle` pour mélanger un tableau.

■ **Exemple 4 — Normalisation d'un ensemble de données par colonne.** On suppose qu'on dispose d'un ensemble de données sous la forme d'un tableau numpy de dimention `(n,m)`. Chaque colonne représente un paramètre que l'on souhaite normaliser : centrer en zéro et d'écart type égal à un.

Pour chaque échantillon x_{ij} de la ligne i et colonne j, on veut :

$$x_{ij}^{norm} = \frac{x_{ij} - \mu_j}{\sigma_j}$$

où μ_j et σ_j sont les moyennes et écart-type de la colonne j.

On peut réaliser ceci en une seule instruction :

```
import numpy as np

def normalized(data):
    return (data — np.mean(data, axis=0)) / np.std(data, axis=0)


data = np.random.random((100, 5))
data = normalized(data)

print(np.max(data, axis=0), np.min(data, axis=0), np.mean(data, axis=0), np.std(data,
    axis=0))
```