UE12P26 · MINES PARIS · PE 2026

Python numérique

le poly

NumPy pandas matplotlib

Tout ce qu'il faut pour les quiz et les devoirs, et un peu plus.

Rappels Python · Le tableau NumPy · Vectoriser · La table pandas · Trier, regrouper, croiser · Le temps et les figures

Sommaire

1 Rappels Python 4
- 1.1 Types de base, opérateurs 4
- 1.2 f-strings 5
- 1.3 Indexation et slicing 6
- 1.4 Fonctions 7
- 1.5 Instructions et blocs 8
- 1.6 Containers 8
- 1.7 Références partagées 9
- 1.8 Itérations 10
- 1.9 Modules et fichiers 11
- 1.10 Classes 12

2 NumPy — le tableau 14
- 2.1 Créer un tableau 14
- 2.2 Anatomie : `shape`, `dtype` et la mémoire 15
- 2.3 `dtype` : choisir la taille des entiers 16
- 2.4 Indexation et slicing 17
- 2.5 `reshape`, vues et copies 18
- 2.6 Vectorisation : jamais de boucle `for` 19
- 2.7 Les images : des tableaux `(H, W, 3)` 19

3 NumPy — vectoriser 22
- 3.1 Agrégations et `axis` 22
- 3.2 Masques booléens 23
- 3.3 Broadcasting 25
- 3.4 Aléatoire : `np.random.default_rng` 26
- 3.5 Indexation par un tableau d'entiers 27
- 3.6 Monte-Carlo, convergence, Mandelbrot (D2) 28
- 3.7 La panoplie des tableaux de coordonnées (vers D6) 30
- 3.8 Divers à connaître 31

4 pandas — la table de données 33
- 4.1 Anatomie d'une DataFrame 33
- 4.2 Lire et écrire un CSV 34
- 4.3 Premier regard sur une table 35
- 4.4 `loc` et `iloc` 35
- 4.5 Masques et sélections 36
- 4.6 Créer, supprimer, convertir des colonnes 37

4.7 Valeurs manquantes : `NaN` 37
4.8 Modifier : la règle d'or 38
4.9 Les gestes mis bout à bout 38

5 pandas — trier, regrouper, croiser 40

5.1 Trier 40
5.2 `groupby` : éclater, agréger, recoller 41
5.3 `pivot_table` : le tableau croisé 43
5.4 `pd.cut` : découper en intervalles 44
5.5 `merge` : aligner deux tables sur une clé 44
5.6 `concat` : empiler 45
5.7 Champions et podiums : `idxmax`, `nlargest` 46
5.8 Construire une DataFrame par programme 47
5.9 Les briques mises bout à bout 47

6 Le temps et les figures 50

6.1 Dates : `datetime64`, `Timestamp`, `to_datetime` 50
6.2 Durées : `to_timedelta` et `.dt` 51
6.3 Séries temporelles : `DatetimeIndex` 52
6.4 `resample` vs `rolling` 54
6.5 matplotlib : figure et axes 56
6.6 `plot`, `scatter` et les valeurs manquantes 58
6.7 Grilles de sous-figures 59
6.8 Tracer depuis pandas : `df.plot` 60
6.9 Pour finir 61

CHAPITRE 1

Rappels Python

Vous avez tous écrit du Python en prépa, plus ou moins récemment et plus ou moins volontiers. Ce chapitre n'est donc pas un cours de programmation, mais une remise en route : nous allons y revoir les mécanismes du langage sur lesquels s'appuient tout le reste du poly, les quiz et les devoirs. À l'usage, trois d'entre eux font trébucher une fois sur deux et méritent qu'on s'y arrête : la différence entre `/` et `//`, la borne de fin exclue dans les *slices*, et le fait qu'une affectation ne copie jamais rien.

1.1 Types de base, opérateurs

Python est *typé dynamiquement*, c'est-à-dire qu'on n'annonce jamais le type d'une variable : il est déduit de ce qu'on range dedans (au besoin, `type(x)` permet de le vérifier). De plus, tout y est *objet* : les nombres et les chaînes bien sûr, mais aussi les fonctions et les modules, que l'on pourra donc ranger dans une variable ou passer en argument comme n'importe quelle autre valeur.

```
a = 10          # int
b = 3.14        # float
c = True        # bool (True se comporte comme 1)
s = "texte"     # str ('...' ou "..." au choix)
```

Passons à l'arithmétique. Là où beaucoup de langages n'ont qu'une division, Python en a deux, et les confondre donne un résultat faux sans le moindre message d'erreur !

```
25 / 10, 25 // 10, 25 % 10, 2 ** 10
```

```
RÉSULTAT  (2.5, 2, 5, 1024)
```

La barre oblique simple est la division « mathématique » : `25 / 10` vaut `2.5`, et surtout `10 / 5` vaut `2.0` et non `2`. La double barre `//` donne quant à elle le quotient entier, `%` le reste de la division, et `**` la puissance.

PIÈGE

`/` renvoie *toujours* un flottant, même entre entiers. La division entière est `//`, le reste `%`. Pour tester la parité d'un nombre, on écrira donc : `x % 2 == 0`.

Un flottant, justement, n'est qu'une approximation : Python le code sur 64 bits (norme IEEE 754), or la plupart des nombres décimaux n'ont pas d'écriture exacte en binaire, de la même façon qu'un tiers n'a pas d'écriture décimale exacte. Voyez plutôt :

```
1 0.2 + 0.1, 0.2 + 0.1 == 0.3
```

```
RÉSULTAT (0.30000000000000004, False)
```

Cette différence est la conséquence inévitable de l'arrondi, et non un bug de Python.

À RETENIR

`0.2 + 0.1 == 0.3` vaut `False` : on ne teste jamais l'égalité de deux flottants avec `==`. On demande plutôt s'ils sont *proches*, avec `math.isclose(a, b)` (ou `np.isclose` sur des tableaux).

Côté comparaisons, nous disposons de `==`, `!=`, `<`, `<=` et leurs symétriques ; côté logique de `and`, `or` et `not`, écrits en toutes lettres et non en symboles. Notez un agrément propre à Python : les comparaisons peuvent se « chaîner », ce qui permet d'écrire `0 <= x < 10` comme on le ferait au tableau.

PIÈGE

Ce confort s'arrête aux scalaires : le chaînage, `and`, `or` et `not` sont *interdits sur les tableaux numpy et les Series pandas*. On y utilisera `&`, `|` et `~` sur des masques, cf. §3.3.

1.2 f-strings

Dès qu'il s'agit d'afficher un résultat lisible (une valeur dans un `print`, un titre de figure, une légende), nous passerons par une *f-string*. Le principe tient en trois signes : un `f` devant les guillemets, les expressions entre accolades, et un format optionnel après un deux-points.

```python
import math
print(f"pi = {math.pi:.2f}, sin = {math.sin(1):.3f}")
```

```
RÉSULTAT  pi = 3.14, sin = 0.841
```

Retenez bien que ce qui est entre accolades est du vrai code Python, évalué au moment de la construction de la chaîne : `{math.sin(1)}` appelle bien la fonction. Le suffixe `:.2f` demande deux chiffres après la virgule ; nous retrouverons ce mini-langage de format plus tard, pour les axes des figures.

1.3 Indexation et slicing

Toutes les séquences de Python (chaînes, listes, tuples, et plus tard tableaux numpy et Series pandas) s'indexent de la même manière : ce que nous voyons ici resservira donc à l'identique au chapitre 2. Les indices commencent à `0`, et les indices négatifs comptent depuis la fin, ce qui évite d'écrire `t[len(t) - 1]` pour accéder au dernier élément.

```python
chaine = 'abcdefghij'
chaine[0], chaine[-1]
```

```
RÉSULTAT  ('a', 'j')
```

Au-delà d'un élément isolé, on peut extraire une tranche (un *slice*) avec la syntaxe `debut:fin:pas`. Attention au point qui revient dans tous les quiz : **la borne `fin` est exclue**, comme dans `range` !

```python
chaine[1:4], chaine[::2], chaine[::-1]
```

```
RÉSULTAT  ('bcd', 'acegi', 'jihgfedcba')
```

Les trois termes sont optionnels : `debut` vaut `0` par défaut, `fin` la longueur, et `pas` vaut `1`. C'est ce qui explique les écritures idiomatiques ci-dessus : `[::2]` prend une case sur deux depuis le début, et un pas négatif parcourt la séquence à l'envers, si bien que `[::-1]` la renverse entièrement.

La convention « fin exclue » peut sembler arbitraire, mais elle a une vertu à l'usage : la fin d'une tranche étant exactement le début de la suivante, les tranches s'emboîtent sans se recouvrir ni laisser de trou (`s[:3] + s[3:] == s`, quel que soit l'indice de coupe), et la longueur d'un slice se calcule immédiatement : `fin - debut`.

> **À RETENIR**
>
> `t[debut:fin:pas]` : `debut` inclus, `fin` *exclue*. `t[-1]` est le dernier élément, `t[::-1]` la séquence renversée.

1.4 Fonctions

Une fonction se définit avec `def`, son corps est indenté, et la première chaîne du corps sert de documentation (le *docstring*) : c'est elle que renverra `help`.

```
def poly(x, a=1):
    """le docstring documente la fonction"""
    return a * x**2 + 1
```

Le paramètre `a` a ici une valeur par défaut : il devient *optionnel*, et on peut le nommer à l'appel plutôt que de compter les positions.

```
poly(3), poly(3, a=2)
```

```
RÉSULTAT  (10, 19)
```

`return` arrête immédiatement la fonction et renvoie la valeur indiquée. Si l'exécution atteint la fin du corps sans rencontrer de `return`, la fonction renvoie `None`. C'est légal, et cela explique la plupart des `None` inattendus que vous verrez s'afficher dans vos notebooks !

> **À RETENIR**
>
> Sans `return`, une fonction renvoie `None`. Les variables créées dans le corps sont *locales* : leur portée s'arrête à la fin de la fonction.

Notez également qu'une fonction peut accepter un nombre variable d'arguments : dans `def f(fixe, *args)`, le nom `args` reçoit le reste des arguments sous forme de tuple, et l'étoile joue symétriquement à l'appel, où `f(*liste)` « déballe » la liste. Enfin, quand une fonction ne peut pas faire son travail, elle lève une exception plutôt que de renvoyer une valeur bidon : `raise ValueError("...")` interrompt l'exécution et fait remonter l'erreur dans la pile des appels, et `try: ... except Exception as e: ...` permet de la rattraper.

1.5 Instructions et blocs

Python n'a ni accolades ni `begin/end` : c'est l'*indentation* (quatre espaces par niveau) qui délimite les blocs de `if`, `elif`, `else`, `for`, `while`, `def`, `with`… Elle fait donc partie de la syntaxe du langage, et une indentation incohérente est une erreur de syntaxe, pas une faute de goût.

Dans un test, Python ne réclame pas un booléen : il accepte n'importe quelle valeur et lui demande si elle est « vraie ». La règle est simple : tout ce qui est vide ou nul est faux.

```
1 bool(0), bool(""), bool([]), bool(None), bool([0])
```

```
RÉSULTAT (False, False, False, False, True)
```

Sont donc *falsy* : `0`, `0.0`, `""`, `[]`, `set()`, `{}` et `None` ; tout le reste vaut `True`. On en profite pour écrire `if liste:` plutôt que `if len(liste) > 0:`. Dans les boucles enfin, `break` sort de la boucle *la plus imbriquée* (celle qui contient directement l'instruction), tandis que `continue` abandonne l'itération en cours et passe directement à la suivante.

1.6 Containers

Python fournit quatre structures de données de base, et le savoir-faire consiste à choisir la bonne. La `list` est le réflexe par défaut : une collection ordonnée qu'on remplit au fur et à mesure. Le `tuple` sert quand la collection ne doit pas changer, typiquement un couple de coordonnées ou le retour multiple d'une fonction. Le `dict` associe des clés à des valeurs, dès qu'on veut retrouver une information par son nom plutôt que par sa position. Le `set`, enfin, dédoublonne et teste très rapidement l'appartenance.

type	littéral	propriétés
`list`	`[1, 2, 3]`	ordonnée, modifiable ; `append`, `pop`, `+` concaténe, `in`, slicing
`tuple`	`(1, 2)` ; 1 élément : `(1,)`	comme une liste, mais *non modifiable*
`dict`	`{'a': 1, 'b': 2}`	associations clé → valeur ; `d['a']`, `d['c'] = 3` ajoute, `del d['a']`, `in` teste les clés, `.items()`
`set`	`{1, 2, 3}` ; vide : `set()`	sans doublon ni ordre ; `add`, `remove`, `in` très rapide

Deux de ces littéraux sont contre-intuitifs, et reviennent régulièrement dans les quiz :

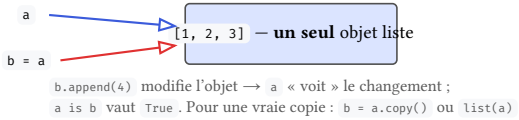
PIÈGE

`(1,)` est un tuple, `(1)` est l'entier `1` : c'est la virgule qui fait le tuple, pas les parenthèses ! De même, `{}` est un dictionnaire vide ; l'ensemble vide s'écrit `set()`.

Le pendant naturel des tuples est l'affectation multiple, ou *unpacking* (déballage) : `a, b = 10, 20` initialise deux variables d'un coup, `a, b = b, a` les échange sans variable temporaire, et le même mécanisme rend très lisible le parcours d'un dictionnaire : `for cle, val in d.items(): ...`.

1.7 Références partagées

Nous arrivons à la notion qui produit les bugs les plus déroutants du chapitre. En effet, une variable, un paramètre de fonction ou une case de liste ne *contiennent* pas un objet : ce sont des *références* qui pointent vers lui. Ainsi l'affectation et l'appel de fonction *ne copient jamais* l'objet : ils font seulement pointer un nom de plus vers le même endroit.



Vérifions-le ensemble : nous créons une liste, nous l'affectons à un second nom, puis nous modifions l'objet à travers ce second nom.

```
a = [1, 2, 3]
b = a               # même objet !
b.append(4)
print(a, a is b)
```

RÉSULTAT `[1, 2, 3, 4] True`

La liste `a` a changé sans qu'on l'ait touchée ! Il n'y a jamais eu qu'une seule liste, désignée par deux noms. Pour en obtenir une indépendante, il faut le demander explicitement, avec `a.copy()` ou `list(a)`. Notez au passage l'opérateur `is`, qui ne fait pas la même chose que `==` : `==` compare les *valeurs* (deux listes de même contenu sont égales), tandis que `is` demande s'il s'agit du *même objet* en mémoire.

```python
c = a.copy()
c.append(5)
a, c is a
```

```
RÉSULTAT  ([1, 2, 3, 4], False)
```

MUTABLE OU PAS ?

Le partage n'est un souci que pour les objets *modifiables* : `list`, `dict`, `set`, et les `ndarray` numpy (une fonction qui modifie son argument tableau modifie l'original : d'où le `positions.copy()` du devoir D6). Nombres, chaînes et tuples sont immuables, aucun risque. `==` compare les *valeurs*, `is` teste si c'est le *même objet*.

1.8 Itérations

Contrairement à d'autres langages, la boucle `for` de Python ne « compte » pas : elle *parcourt*. On lui donne un itérable (une liste, un dictionnaire, un fichier ouvert, un `range`) et elle en sort les éléments un par un.

```python
for item in container: ...
```

Quand on a besoin de l'indice *en plus* de l'élément, on n'écrit pas `range(len(liste))` : on emballe plutôt l'itérable dans `enumerate`, qui produit des couples `(indice, élément)` que l'on déballe directement dans l'en-tête de la boucle.

```python
for i, lettre in enumerate("abc"):
    print(i, lettre)
```

```
RÉSULTAT  0 a
1 b
2 c
```

Et pour parcourir deux séquences en parallèle, la bonne réponse est `zip`, qui les apparie élément par élément et s'arrête à la plus courte.

```python
list(zip([1, 2, 3], "abc"))
```

```
RÉSULTAT  [(1, 'a'), (2, 'b'), (3, 'c')]
```

Reste le cas de `range(n)`, qui énumère `0 … n-1` (fin exclue, toujours !) sans jamais construire la liste en mémoire : c'est le bon outil pour répéter quelque chose n fois, mais le mauvais pour parcourir un container.

Terminons par un outil que nous utiliserons beaucoup : les *compréhensions*, qui construisent un container en une seule expression, en itérant sur un autre.

```
[x**2 for x in range(5)]     # liste  [0, 1, 4, 9, 16]
{x: x**2 for x in range(3)}  # dict   {0: 0, 1: 1, 2: 4}
[x for x in t if x % 2 == 0] # avec filtre
```

La première fabrique une liste, la deuxième un dictionnaire (notez les deux-points entre la clé et la valeur), et la troisième filtre au passage grâce à la clause `if` finale. Enfin, les compréhensions peuvent s'emboîter, ce qui donne une façon très compacte de construire un tableau à deux dimensions, un motif qui vous servira tel quel dans le devoir D2 :

```
np.array([[f(n, s) for s in range(5)] for n in ns])
```

1.9 Modules et fichiers

1.9.1 Importer un module

La bibliothèque standard et les bibliothèques scientifiques s'utilisent toutes de la même façon : on importe un module, puis on qualifie les noms qu'on en tire.

```
import math                # -> math.pi, math.sin
from math import pi        # importe juste le nom pi
import numpy as np         # alias : LA convention du cours
```

Je vous recommande de préférer `import math` puis `math.sin` à `from math import sin` : sur une ligne de calcul un peu longue, savoir d'où vient chaque fonction vaut largement les cinq caractères économisés. Font exception les alias consacrés (`np` pour numpy, `pd` pour pandas, `plt` pour `matplotlib.pyplot`), qu'on écrit toujours ainsi.

1.9.2 Lire et écrire un fichier

Un fichier ouvert est une ressource : il faut le refermer, y compris si une erreur survient entre-temps. Plutôt que d'y penser à chaque fois, on confie ce soin au mot-clé `with`, qui referme le fichier automatiquement à la sortie du bloc.

```
with open("resultats.txt", "w") as f:
    print("ligne", file=f)
```

Le second argument précise le mode : `"w"` pour écrire (attention, le fichier est écrasé !), et rien du tout pour lire, qui est le mode par défaut. À la lecture, l'objet fichier est lui-même itérable, et l'itérer donne les lignes une par une, sans jamais charger tout le fichier en mémoire.

```python
with open(DATA) as f:
    for line in f: ...
```

> **À RETENIR**
>
> Toujours `with open(...) as f:` : le fichier se referme tout seul, même en cas d'erreur. Itérer sur `f` donne les lignes une à une.

1.10 Classes

Nous n'écrirons pas beaucoup de classes dans ce cours, mais nous en manipulerons sans arrêt : un `ndarray`, une `DataFrame`, une figure matplotlib sont des objets, et savoir lire une définition de classe vous aidera à comprendre leur documentation.

Une classe se déclare avec `class`, et sa méthode `__init__` (le *constructeur*) est appelée automatiquement à la création d'un objet ; son rôle est d'attacher à l'objet ses attributs.

```python
class User:
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

Les autres méthodes se définissent dans le même bloc indenté. Certaines sont ordinaires, comme `birthday` ci-dessous ; d'autres portent des noms entourés de doubles soulignés et sont appelées par Python lui-même : par ex. `__repr__` dit comment l'objet doit s'afficher.

```python
class User:
    # ... suite du même bloc
    def birthday(self):
        self.age += 1
    def __repr__(self):
        return f"{self.name}, {self.age} ans"
```

Pour créer une instance, on appelle la classe comme une fonction, avec les arguments de `__init__` moins le premier : `User("Martin", 22)` déclenche `__init__(self=objet créé, "Martin", 22)`.

```
u = User("Martin", 22)
u.birthday()
u
```

```
RÉSULTAT  Martin, 23 ans
```

Reste à expliquer ce `self`, premier paramètre de chaque méthode et jamais fourni à l'appel. En clair, `self` est l'objet lui-même : `u.birthday()` est une écriture abrégée de `User.birthday(u)`. C'est la règle générale des méthodes (`objet.methode(args)` signifie `Type.methode(objet, args)`), et c'est pour cela que `"abc".upper()` fonctionne.

EN PRATIQUE

Aide en ligne : la complétion `Tab`, `Shift-Tab` sur un nom de fonction dans Jupyter, `pd.read_csv?`, ou encore `help(f)`. Des réflexes à prendre avant les devoirs !

CHAPITRE 2

NumPy — le tableau

Une matrice, une série de mesures, une image ou un son semblent être des objets très différents. Pourtant, du point de vue de la mémoire de l'ordinateur, il s'agit à chaque fois de la même chose, c'est-à-dire d'un *tableau multi-dimensionnel* de nombres. Ce type de tableau n'existe pas dans Python de base : c'est la bibliothèque `numpy` qui le fournit, sous le nom de `numpy.ndarray` (*n-dimensional array*). Retenez bien que tout le calcul scientifique en Python, `pandas` et `matplotlib` compris, repose sur cette brique.

Deux propriétés définissent le `ndarray`, et nous allons voir qu'elles expliquent à la fois sa vitesse et ses pièges : il est *homogène*, c'est-à-dire que toutes les cases ont le même type, et ses éléments sont *de taille fixe* (le nombre d'octets par case ne change jamais).

Commençons par les imports : on écrit toujours ces deux lignes de la même façon, car c'est LA convention du cours, et je vous demande de la respecter dans tous vos rendus.

```
import numpy as np
import matplotlib.pyplot as plt
```

2.1 Créer un tableau

À l'usage, un tableau se crée le plus souvent de trois façons : en convertissant une liste Python, en demandant un tableau « tout fait » d'une forme donnée, ou en générant une suite de valeurs. Le tableau ci-dessous rassemble les constructeurs du cours ; notez que les trois derniers reviendront sans cesse dans les devoirs.

`np.array(liste)`	convertit une liste (de listes) ; forme et type déduits
`np.zeros((2, 3))`	rempli de `0.` ; *flottants* par défaut, `dtype=` pour choisir

`np.ones(shape)`, `np.empty(shape)`	rempli de `1.` / *non initialisé* (contenu quelconque)
`np.arange(2, 20, 3)`	de 2 à 20 *exclu*, pas 3 → `[2 5 8 11 14 17]` : 6 éléments
`np.linspace(0, 1, 5)`	5 valeurs équidistantes, *bornes incluses* → `[0. 0.25 0.5 0.75 1.]`
`np.random.randint(0, 10, size=(3, 4))`	entiers dans `[0, 10[` ; borne sup. *exclue*, paramètre `size=` (pas `shape=`)
`np.indices((n, m))`	`I, J = np.indices((n, m))` : `I[i, j] == i`, `J[i, j] == j`

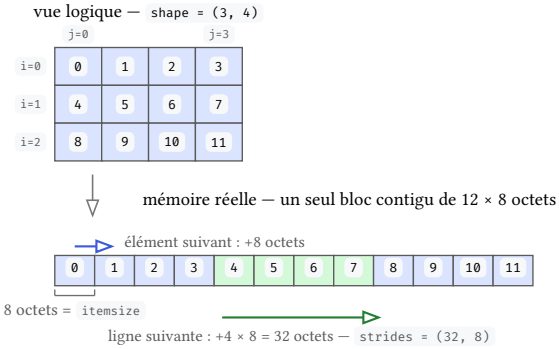
Remarquez la logique des bornes : `arange` s'arrête *avant* la borne haute, exactement comme le `range` de Python, et on lui donne un pas. `linspace` inclut quant à lui les deux bornes, et on lui donne cette fois un *nombre de points*. C'est donc `linspace` que nous utiliserons pour échantillonner une fonction à tracer.

> **À RETENIR**
>
> `np.array([1, 2, 3])` renvoie un objet de type `numpy.ndarray` : il n'existe pas de type `numpy.array`, c'est le nom de la fonction de construction. `arange` exclut la borne haute (comme `range`), `linspace` inclut les deux bornes.

2.2 Anatomie : `shape`, `dtype` et la mémoire

`numpy` doit sa vitesse à l'organisation de la mémoire : un tableau y est *un seul bloc d'octets contigu*, dans lequel toutes les cases ont la même taille. Ainsi, pour passer d'une case à la suivante, il suffit d'avancer d'un nombre fixe d'octets, un simple décalage que l'on appelle l'*offset* : il n'y a ni recherche ni détour à faire, et c'est cette régularité qui rend tout le reste possible.

La figure suivante montre les deux faces d'un même tableau : d'un côté la grille `(3, 4)` que vous voyez, de l'autre le ruban de 96 octets que la machine voit.



La vue logique n'est qu'un habillage : `reshape` peut ainsi la changer sans déplacer la moindre donnée, puisqu'il suffit de réinterpréter le même ruban. Les `strides` (en clair, « de combien d'octets sauter pour avancer d'un cran dans chaque dimension ») servent de table de conversion entre les deux mondes.

Interrogeons un tableau sur son anatomie :

```
1 tab = np.arange(24).reshape(2, 3, 4)
2 tab.shape, tab.ndim, tab.size, tab.dtype, tab.itemsize, tab.nbytes
```

```
RÉSULTAT  ((2, 3, 4), 3, 24, dtype('int64'), 8, 192)
```

Chacun de ces attributs répond à une question simple. `shape` donne la forme du tableau (un tuple, avec une valeur par dimension) ; `ndim` donne le nombre de dimensions, c'est-à-dire `len(shape)` ; `size` compte les éléments en tout ; `dtype` indique le type, commun à toutes les cases ; `itemsize` précise le nombre d'octets occupés par chaque élément. Enfin `nbytes` boucle la boucle : c'est le produit `size × itemsize`, soit la taille mémoire totale, ici $24 \times 8 = 192$ octets.

2.3 `dtype` : choisir la taille des entiers

Le `dtype` décide de ce qu'une case *peut* contenir. Avec n bits on représente 2^n valeurs : des entiers signés dans $[-2^{n-1}, 2^{n-1} - 1]$, ou non signés dans $[0, 2^n - 1]$. Ainsi `int8` couvre $-128 \ldots 127$, et `uint8` couvre $0 \ldots 255$, ce qui en fait le type naturel des images, dont les pixels prennent justement leurs valeurs entre 0 et 255.

Pour changer de type, on convertit avec `astype` : `t.astype(np.float64)` renvoie une *copie indépendante* convertie ; notez que l'original conserve son type.

Que se passe-t-il alors quand un calcul produit une valeur qui ne tient pas dans le type ? C'est le piège le plus classique du cours :

NUMPY CALCULE À TAILLE CONSTANTE

Le type d'un tableau ne grandit jamais pendant un calcul : le résultat *déborde* silencieusement, en arithmétique modulo 2^n.

```
t = np.array([100, 50], dtype=np.int8)
t + t                    # 200 > 127 : déborde !
```

RÉSULTAT `array([-56, 100], dtype=int8)`

```
im = np.array([250, 10], dtype=np.uint8)
im + 10                  # 260 % 256 = 4
```

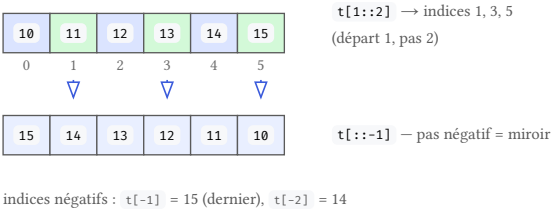
RÉSULTAT `array([ 4, 20], dtype=uint8)`

En effet, $200 - 256 = -56$ en `int8`, et $260 \bmod 256 = 4$ en `uint8`. Pour calculer juste, on convertit donc *avant* le calcul (`t.astype(int)`), puis on revient au besoin au format image avec `np.clip(..., 0, 255).astype(np.uint8)`.

À l'usage, la règle du cours est simple : restez sur le type que `numpy` infère tout seul (`int64` ou `float64`), et réservez `uint8` aux images, en gardant ce piège en tête chaque fois que vous *calculez* sur une image.

2.4 Indexation et slicing

L'accès aux éléments reprend les conventions de Python revues au chapitre 1 (indices à partir de 0, indices négatifs depuis la fin, slices `debut:fin:pas` avec la borne `fin` exclue) et les généralise à toutes les dimensions. Le schéma ci-dessous les résume sur un tableau 1D.



En dimension supérieure, on donne *un indice par dimension, séparés par des virgules* : `t[i, j]`. L'écriture `t[i][j]` fonctionne aussi, cependant elle construit un tableau intermédiaire pour rien : prenez le pli de la virgule dès maintenant.

```python
t = np.arange(12).reshape(3, 4)
t[1, -1], t[-1][0], t[0]        # élément, élément, ligne entière
```

```
RÉSULTAT  (7, 8, array([0, 1, 2, 3]))
```

Notez que donner *moins* d'indices que de dimensions renvoie un sous-tableau : ici `t[0]` est la première ligne entière.

Chaque dimension accepte aussi un slice, et l'on peut alors *affecter* une valeur à toute une sélection d'un coup, la valeur étant répétée automatiquement (elle est « broadcastée », nous y reviendrons au chapitre suivant) :

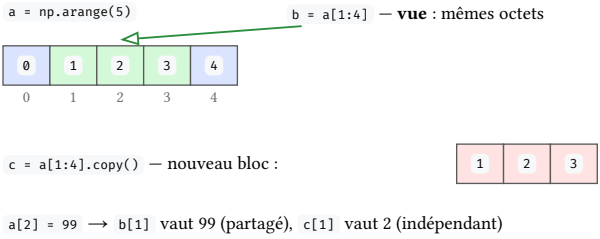
```python
t = np.zeros((3, 3), dtype=int)
t[1:, 1:] = 7                   # le bloc en bas à droite
t.sum()
```

```
RÉSULTAT  28
```

Combinée aux pas, cette écriture dessine des motifs entiers sans une seule boucle : ainsi `t[:, ::2] = 1` peint une colonne sur deux (des rayures verticales), tandis que `t[0] = 1` et `t[-1] = 1` tracent un cadre. Je vous laisse inventer le reste : c'est tout l'esprit du devoir D1.

2.5 `reshape`, vues et copies

Nous arrivons à la conséquence la plus déroutante du modèle mémoire : *slicer ou reshaper un tableau numpy ne copie rien.* On obtient certes un nouvel objet, avec sa propre indexation, cependant il regarde *le même bloc d'octets* que l'original : on dit que c'est une *vue* (*view*). Ainsi, modifier l'un revient à modifier l'autre.



Vérifions-le ensemble, en quatre lignes :

```python
a = np.arange(6)
b = a.reshape(2, 3)     # b est une vue sur les données de a
b[0, 0] = 100
a[0]
```

RÉSULTAT `100`

Si chaque slice copiait ses données, extraire une colonne d'une image de plusieurs millions de pixels coûterait une allocation énorme, et le code serait inutilisable à l'usage. `numpy` alloue donc le moins de mémoire possible, et c'est à vous de demander une copie quand vous en voulez une, avec `.copy()`.

Ajoutons trois compléments utiles au quotidien. Tout d'abord, `reshape` accepte un `-1` qui signifie « calcule cette dimension pour moi » : sur 24 éléments, `reshape(4, -1)` donne `(4, 6)`, `reshape(-1)` aplatit tout, et `reshape(-1, 1)` fabrique une colonne. Ensuite, `np.shares_memory(a, b)` indique si deux tableaux partagent leurs octets, ce qui est bien pratique pour vérifier ce que l'on manipule. Enfin, `plt.imread` peut renvoyer un tableau *non modifiable* : le cas échéant, on le duplique avec `im = im.copy()` avant d'y toucher.

2.6 Vectorisation : jamais de boucle `for`

Comment appliquer une fonction à tous les éléments d'un tableau ? Contrairement au réflexe hérité des listes Python, on ne « parcourt » pas le tableau : on applique la fonction *au tableau entier*, et ainsi `y = np.sin(x)` calcule le sinus des dix millions de valeurs de `x` d'un coup. La boucle existe bien, cependant elle tourne *à l'intérieur* de `numpy`, en C, sur un bloc mémoire régulier : elle est dix à cent fois plus rapide qu'un `for` Python, et le code est au passage plus court et plus lisible.

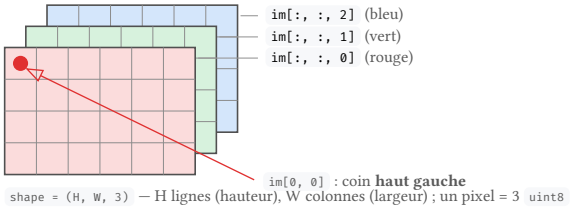
Ces fonctions élément par élément (`np.sin`, `np.exp`, `np.abs`, ainsi que tous les opérateurs `+ - * / ** %`) portent un nom : ce sont les *ufunc* (*universal functions*). Retenez ce terme, il vous sera utile pour chercher dans la documentation.

> **À RETENIR**
>
> La règle du cours, et la consigne de tous les devoirs : **aucune boucle Python sur les éléments d'un tableau**. Tout se fait par slicing, opérations vectorisées, agrégations et broadcasting.

2.7 Les images : des tableaux `(H, W, 3)`

Les images sont le terrain de jeu idéal pour mettre en pratique tout ce qui précède : quand on se trompe, l'erreur se voit à l'écran ! Une image couleur est un tableau

`uint8` de forme `(hauteur, largeur, 3)` : pour chaque pixel, trois valeurs entre 0 et 255 dosent le rouge, le vert et le bleu. C'est le principe de la synthèse additive : `(0, 0, 0)` donne du noir, `(255, 255, 255)` du blanc, et `(255, 255, 0)` (rouge plus vert) du jaune.



Chargeons une image et jouons avec le slicing : chaque ligne ci-dessous est une opération d'image complète, sans aucune boucle.

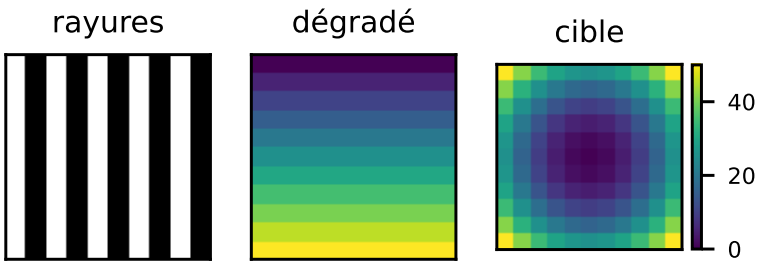
```
im = plt.imread("data/les-mines.jpg")   # -> ndarray (533, 800, 3) uint8
im.shape[0]           # hauteur (lignes) ; im.shape[1] : largeur
im[:, :, 1]           # le canal vert : un tableau 2D (H, W)
```

```
im[:10, :10]          # le coin 10x10 en haut à gauche
im[::2, ::2]          # une ligne et une colonne sur deux
im[:, ::-1]           # miroir gauche-droite ; im[::-1] : haut-bas
```

```
im[2:5, 6:10] = (220, 30, 30)   # peindre un rectangle (broadcasting)
255 - im                        # le négatif
```

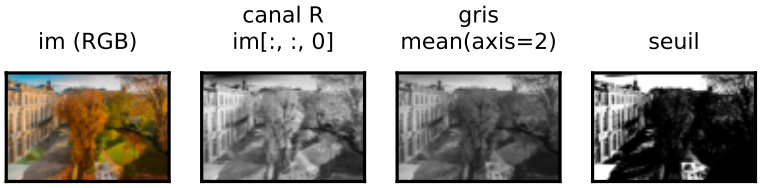
Donnons quelques repères pour les devoirs. Pour créer une image noire de 100 pixels de haut sur 200 de large, on écrira `np.zeros((100, 200, 3), dtype=np.uint8)` : *les lignes d'abord*, car c'est la convention qui piège tout le monde. Notez par ailleurs que certaines images ont un quatrième canal, dit *alpha*, qui code la transparence de `0` (transparent) à `255` (opaque) : c'est le format RGBA.

Côté affichage, `plt.imshow(im)` suffit pour une image couleur. Un tableau 2D (niveaux de gris) n'emporte en revanche pas sa couleur : il faut préciser la table avec `cmap`, typiquement `plt.imshow(gris, cmap="gray")`. On habille la figure avec `plt.title("...")` et `plt.colorbar()`, *avant* `plt.show()`. Pour écrire sur disque, on utilise `plt.imsave("fig.png", im)` ; attention cependant au chemin du retour, car `plt.imread` d'un PNG renvoie des *flottants entre 0 et 1* avec un canal alpha, donc une forme `(H, W, 4)`, et non des `uint8`.



`plt.imshow` + `plt.colorbar()` sur trois motifs sans boucle : rayures par slicing à pas 2 (`t[:, ::2] = 1`), dégradé `I` et cible `(I-5)² + (J-5)²` sur `I, J = np.indices(...)` .

Deux outils viennent compléter la panoplie des motifs. `np.outer(a, b)` construit le tableau 2D des produits `a[i] * b[j]` (sa variante `np.add.outer` fait de même avec les sommes) : une grille entière naît ainsi de deux vecteurs. Quant à `np.repeat(t, k, axis=...)`, il duplique `k` fois chaque élément le long d'un axe ; appliqué deux fois à une image, une fois par axe, il la met à l'échelle.



Le TP images du cours : la photo RGB, un canal isolé `im[:, :, 0]`, le passage en gris par `mean(axis=2)`, et un seuillage par masque booléen.

CHAPITRE 3

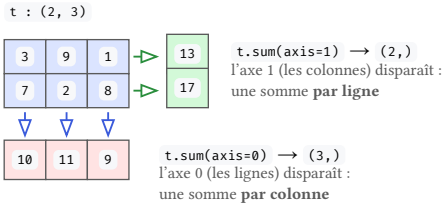
NumPy — vectoriser

Le chapitre précédent a présenté l'objet `ndarray` : un bloc de mémoire, une forme, un type. Dans celui-ci, nous allons apprendre les gestes qui permettent de s'en servir *sans jamais boucler sur les éléments* : agréger le long d'un axe, sélectionner par un masque, ou encore combiner des tableaux de formes différentes. Ces réflexes suffisent à traiter le devoir D2 (Monte-Carlo, marches aléatoires, Mandelbrot) ainsi que D6 (le problème à n corps).

3.1 Agrégations et `axis`

Agréger, c'est réduire plusieurs valeurs à une seule : une somme, une moyenne, un maximum. La question qui revient à chaque quiz est de savoir *sur quoi* porte la réduction.

Les fonctions d'agrégation `sum`, `mean`, `min`, `max`, `std`, `argmin`, `argmax`, `all`, `any`, `cumsum`… existent chacune en deux exemplaires équivalents, une fonction (`np.sum(t)`) et une méthode (`t.sum()`) : prenez celle qui se lit le mieux. Sans argument, elles écrasent *tout* le tableau et renvoient un scalaire. Le paramètre `axis` limite la réduction à une seule dimension, comme le montre le diagramme ci-dessous.



À RETENIR

L'axe que l'on donne est celui qui disparaît. Pour `t` de forme `(2, 3, 5)` : `t.sum(axis=0)` ⟶ `(3, 5)`, `t.sum(axis=1)` ⟶ `(2, 5)`, `t.sum(axis=2)` ⟶ `(2, 3)`. On peut en donner plusieurs d'un coup : `axis=(0, 1)` ⟶ `(5,)`.

En dimension 2, on se trompe souvent, car l'intuition dit le contraire de la règle : `axis=0` agrège *le long des lignes* et laisse un résultat *par colonne*, tandis que `axis=1` agrège le long des colonnes et laisse un résultat *par ligne*. En cas de doute, le bon réflexe consiste à vérifier la forme du résultat plutôt que le sens des flèches. Ainsi, sur une image `(H, W, 3)`, `im.mean(axis=2)` moyenne les trois canaux et renvoie le gris `(H, W)`.

Vérifions cela sur un tableau où les deux axes ne racontent pas la même chose.

```
t = np.array([[3, 9, 1],
              [7, 2, 8]])
print(t.min(axis=0), t.argmin(axis=1))
```

RÉSULTAT `[3 2 1] [2 1]`

`t.min(axis=0)` compare les deux lignes case à case et rend une valeur par colonne ; `t.argmin(axis=1)` travaille quant à elle dans chaque ligne et rend non pas la valeur mais son *indice* : `2` pour la première ligne, `1` pour la seconde.

Les fonctions en `arg…` réservent un piège de plus : appelées *sans* `axis`, `argmax()` et `argmin()` renvoient l'indice dans le tableau *aplati*. Sur un `(3, 4)`, elles peuvent ainsi répondre `7`, qui n'est ni une ligne ni une colonne ; pour retrouver les coordonnées, on utilisera `np.unravel_index(t.argmax(), t.shape)`.

Notez enfin que `cumsum` est l'intruse de la famille, car elle n'agrège rien : `t.cumsum(axis=...)` renvoie *tous les totaux intermédiaires*, donc un tableau de même forme que l'entrée (`[2, 3, 1]` devient `[2, 5, 6]`). C'est elle qui transforme une suite de déplacements en une suite de positions dans le devoir D2.

3.2 Masques booléens

En numpy, une comparaison comme `t > 5` ne répond pas `True` ou `False` : elle renvoie un tableau de booléens de même forme que `t`. C'est ce qu'on appelle un *masque*, et c'est l'objet qui remplace le `if` dans un code vectorisé : au lieu de tester les éléments un par un, on dessine la carte de ceux qui conviennent, puis on s'en sert.

Le premier usage est la sélection : placé entre crochets, un masque extrait les éléments marqués `True`, à plat.

```
t[t > 5]                # les éléments > 5, à plat
```

Le deuxième usage est le comptage : `True` valant 1 et `False` 0, sommer un masque compte les cases qui conviennent, et le moyenner donne une proportion (c'est ainsi que nous estimerons π dans le devoir D2). Quand seule la réponse globale importe, `np.any` et `np.all` évitent de compter pour rien.

```
(t > 5).sum()           # combien d'éléments > 5
(t > 5).mean()          # quelle proportion
np.any(t == 0)          # au moins un zéro ?  np.all : tous ?
```

Le troisième usage est la modification : un masque placé *à gauche* du `=` désigne les cases où écrire, et numpy y écrit directement.

```
t[t % 2 == 1] = -1      # les impairs deviennent -1
```

Enfin, `np.where` fabrique un tableau neuf en choisissant élément par élément : c'est en clair le `if` / `else` vectorisé. Attention, avec un seul argument, `np.where(t > 0)` renvoie tout autre chose, à savoir les *indices* des cases vraies.

```
np.where(t > 0, t, 0)   # t là où t est positif, 0 ailleurs
```

COMBINER DES CONDITIONS

Sur des tableaux, `and` / `or` / `not` et les comparaisons chaînées sont *interdits* et provoquent l'erreur « `ValueError: The truth value of an array … is ambiguous` ». On utilise à la place les opérateurs *bit-à-bit* `&` (et), `|` (ou), `~` (non), en *parenthésant chaque condition*, car leur priorité est plus forte que celle des comparaisons :

```
(t < 0) | (t > 100)     # OK
t > 2 & t < 8           # ValueError ! lu t > (2 & t) < 8
(t >= 25) & (t < 75)    # OK
```

Ce message se reconnaît du premier coup d'œil : Python a demandé à un tableau entier s'il était vrai, ce qui n'a pas de sens. Le coupable est toujours un `and`, un `or` ou un `3 <= t < 7` déguisé (nous avions annoncé cette restriction dès le chapitre 1).

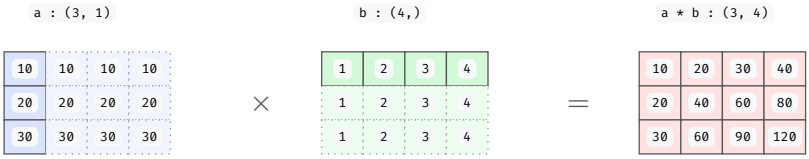
SÉLECTION PAR MASQUE = COPIE

Contrairement au slicing (vue), `s = t[t > 3]` est une *copie* : modifier `s` ne touche pas `t`. Pour modifier l'original, gardez la sélection à gauche du `=` : `t[t > 3] = 0`.

Les masques se composent avec les agrégations : sur un masque en plusieurs dimensions, un `.all(axis=...)` exige que la condition soit vraie partout le long d'un axe, un `.any(axis=...)` qu'elle le soit au moins une fois, et on peut enchaîner les deux. Une question comme « toutes les coordonnées sont-elles nulles au même instant, à au moins un instant ? » se traduit ainsi en deux agrégations, sans aucune boucle ; je vous laisse choisir les axes, le devoir D2 vous y fera penser.

3.3 Broadcasting

Le *broadcasting* (diffusion) décrit la façon dont numpy combine deux tableaux de formes différentes. La règle est courte : les formes sont comparées *de droite à gauche* et, à chaque position, elles doivent être *égales* ou bien *l'une des deux doit valoir 1* ; cette dimension est alors « étirée » virtuellement, sans aucune allocation. Si une seule position échoue, toute l'opération échoue avec un `ValueError`.



les cases en pointillé n'existent pas en mémoire : chaque dimension de taille 1 est « étirée » virtuellement

Une colonne multipliée par une ligne engendre une matrice entière, en une seule ligne de code.

```
x = np.array([2, 5, 10])
x.reshape(-1, 1) * x        # colonne (3,1) x ligne (3,) -> (3,3)
```

RÉSULTAT `[[4 10 20] [10 25 50] [20 50 100]]` : la grille de tous les produits

Le cas d'école du quiz oppose deux situations qui se ressemblent. D'un côté, `(4, 3) + (3,)` *fonctionne* : en partant de la droite, 3 rencontre 3, la dimension manquante se complète, et la ligne de 3 est ajoutée à chacune des 4 lignes. De l'autre, `(3, 4) + (3,)` *échoue*, car à droite 4 rencontre 3 ; le vecteur ne demandait pourtant qu'à être ajouté colonne par colonne, cependant il faut le dire explicitement en lui

donnant la forme `(3, 1)`, avec `reshape(-1, 1)` ou `[:, np.newaxis]` (dont `None` est l'alias). Un scalaire, quant à lui, se diffuse toujours : `t + 1` ne pose aucune question.

> **À RETENIR**
>
> On compare les formes *de droite à gauche* : une dimension de taille 1 s'étire, une dimension absente se complète, et tout le reste doit coïncider. Le réflexe pour fabriquer une colonne : `col[:, None]`.

Trois applications de ce mécanisme reviennent dans les devoirs. La première construit la grille complexe de Mandelbrot (D2) : une ligne de parties réelles `(1, nx)` ajoutée à une colonne de parties imaginaires `(ny, 1)` donne le plan complet d'un coup.

```
c = x[None, :] + 1j * y[:, None]        # (ny, nx)
```

La deuxième calcule *toutes les différences deux à deux* : en opposant un tableau vu en ligne et le même vu en colonne, on obtient la matrice des écarts `D[i, j] = x[j] - x[i]`. Ce motif se généralise à des tableaux de coordonnées, et c'est lui qui ouvre le problème à n corps du devoir D6.

```
D = x[None, :] - x[:, None]             # (N, N)
Delta = P[:, None, :] - P[:, :, None]   # (2, N, N)
```

La troisième, plus ludique, énumère toutes les sommes de deux dés : `d[:, None] + d[None, :]` produit la grille de toutes les issues, qu'un masque suffit ensuite à compter.

3.4 Aléatoire : `np.random.default_rng`

Toutes les simulations du cours partent de tirages au sort : les points de Monte-Carlo, les directions des marcheurs, ou encore les conditions initiales de D6. La façon moderne de les produire consiste à fabriquer d'abord un *générateur*, puis à lui demander des tableaux entiers ; c'est le paramètre `size=` qui décide de la forme, et il accepte un tuple.

```
rng = np.random.default_rng(0)          # graine 0 : reproductible
rng.uniform(-1, 1, size=(n, 2))         # flottants uniformes
rng.integers(0, 4, size=(K, n))         # entiers dans {0, 1, 2, 3}
```

La graine (*seed*) fixe la suite des tirages : deux exécutions donnent exactement les mêmes valeurs. C'est une exigence des devoirs : sans graine, les cellules de test ne

vérifient rien. Notez que l'on croise aussi l'ancienne interface, encore très répandue en ligne : `np.random.randint(0, 10, size=...)` tire des entiers de 0 à 9 *inclus*.

3.5 Indexation par un tableau d'entiers

Jusqu'ici, nous avons mis entre crochets des entiers, des slices et des masques. On peut aussi y mettre un *tableau d'entiers* : numpy remplace alors chaque indice par la valeur correspondante, comme on consulterait une table.

```
table = np.array([10, 20, 30])
table[np.array([0, 2, 2, 1])]
```

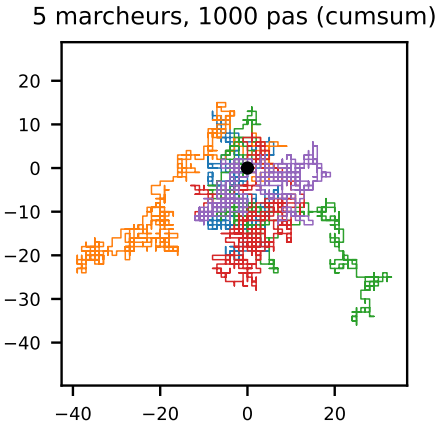
RÉSULTAT `[10 30 30 20]`

Notez que le résultat prend la forme du tableau d'*indices* (et non celle de la table), et que rien n'interdit de demander deux fois le même élément. Si la table a plusieurs dimensions, chaque indice ne rapporte plus un nombre mais *la ligne* entière.

```
directions = np.array([[1, 0], [-1, 0], [0, 1], [0, -1]])
directions[np.array([0, 3])]
```

RÉSULTAT `[[1 0] [0 -1]]`

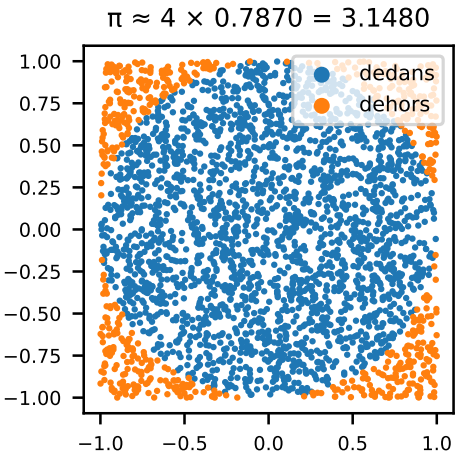
C'est le mécanisme des marches aléatoires du devoir D2 : une table de déplacements indexée par des numéros tirés au hasard fabrique tous les pas d'un coup, puis une somme cumulée le long de l'axe du temps les transforme en positions. Deux lignes suffisent pour cinq cent mille pas, à condition de choisir le bon axe !



Cinq marches aléatoires : indexation par tableau d'entiers + somme cumulée, aucune boucle.

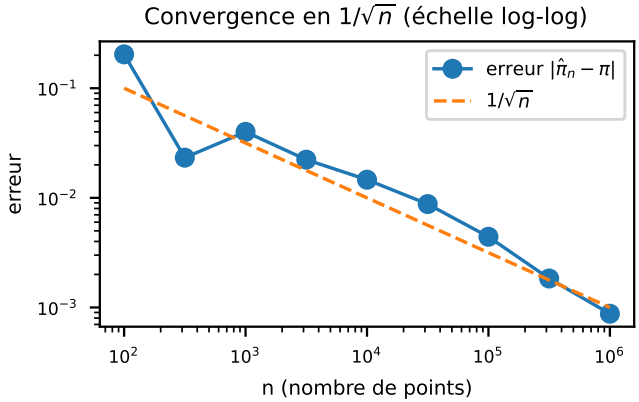
3.6 Monte-Carlo, convergence, Mandelbrot (D2)

Une méthode de *Monte-Carlo* estime une quantité en tirant beaucoup de points au hasard et en comptant ceux qui satisfont une condition. Ici, nous tirons des points du carré et nous regardons s'ils tombent dans le quart de disque : le masque fait le comptage, et la proportion multipliée par 4 donne une estimation de π.



Un masque sépare les points du disque des autres ; deux `plt.scatter` (masque et `~masque`), `plt.axis("equal")` pour un disque rond.

La vitesse à laquelle l'estimation s'améliore importe plus que la valeur trouvée. Sur des axes logarithmiques, une loi de puissance devient une droite dont la pente est l'exposant, et la lecture est alors sans appel : gagner une décimale de précision coûte cent fois plus de points.



`plt.loglog` : l'erreur suit $1/\sqrt{n}$, une droite de pente $-1/2$ en log-log ; `np.logspace(2, 6, 9)` échantillonne les exposants.

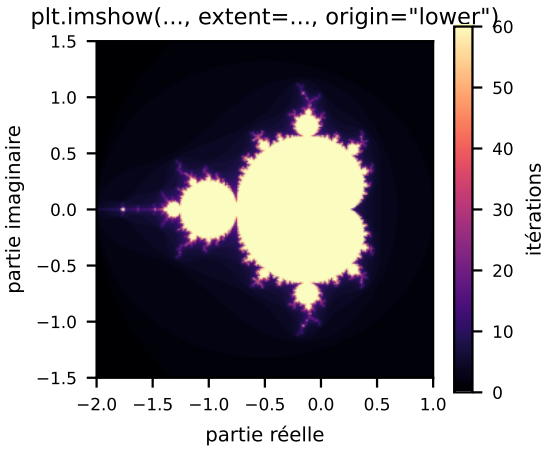
3.6.1 Le schéma vectorisé de Mandelbrot

L'ensemble de Mandelbrot demande, pour chaque point c du plan, d'itérer $z \leftarrow z^2 + c$ et de compter les itérations passées avant que $|z|$ dépasse 2. La tentation est grande d'écrire deux boucles imbriquées, une sur les pixels et une sur les itérations. Or nous n'en garderons qu'*une seule*, celle sur les itérations, car chaque étape dépend de la précédente et aucun broadcasting ne permet de remonter le temps. Les pixels sont en revanche indépendants les uns des autres : ils avancent donc tous ensemble, dans la même expression.

```
z = np.zeros_like(c); n_iter = np.zeros(c.shape, dtype=int)
for k in range(max_iter):
    z = z**2 + c                    # tous les points d'un coup
    n_iter += np.abs(z) <= 2        # True compte pour 1
```

Dans la dernière ligne, `np.abs(z) <= 2` est un masque de la taille de l'image, qui vaut `True` là où le point n'a pas encore divergé : l'ajouter à `n_iter` n'incrémente donc que les compteurs des points encore « vivants ». À la sortie, `n_iter` vaut `max_iter` pour les points de l'ensemble, et seulement quelques unités pour ceux qui explosent tout de suite. En clair, tester et compter sont ici une seule et même opération, et c'est ce qui nous dispense d'un `if` par pixel.

Les points qui divergent produisent des `RuntimeWarning` (overflow, invalid) sans gravité : `with np.errstate(over="ignore", invalid="ignore"):` les fait taire, à condition bien sûr d'avoir compris d'où ils viennent.



`plt.imshow(m, extent=[xmin, xmax, ymin, ymax], origin="lower")` : axes gradués en coordonnées, pas en pixels ; `plt.colorbar(label="itérations")`.

3.7 La panoplie des tableaux de coordonnées (vers D6)

Le projet final (des corps qui s'attirent et que l'on regarde orbiter) se joue entièrement avec les outils de ce chapitre, appliqués à des tableaux de coordonnées. Nous présentons ici les pièces du jeu ; leur assemblage constitue le devoir lui-même.

Il faut d'abord savoir empiler des vecteurs : deux vecteurs `(N,)` de coordonnées x et y se recollent en un seul tableau `(2, N)`, dont la ligne 0 contient les x et la ligne 1 les y :

```
1 np.vstack([[1, 2, 3], [4, 5, 6]])
```

RÉSULTAT `[[1 2 3] [4 5 6]]` ; `np.stack` fait de même, en laissant choisir l'axe

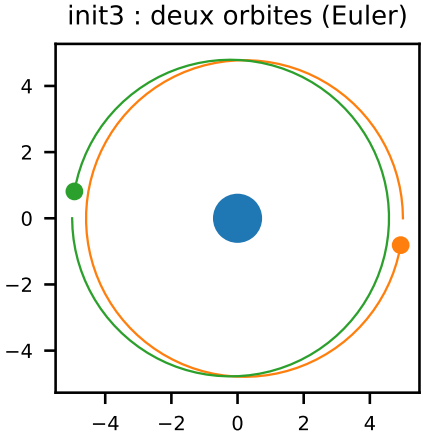
Il faut ensuite savoir mesurer des longueurs : `np.linalg.norm(t, axis=k)` calcule la norme euclidienne le long de l'axe `k`, qui disparaît comme dans toute agrégation. C'est `np.sqrt((t**2).sum(axis=k))`, en plus court. Sur un tableau `(2, N)` de coordonnées, `axis=0` renvoie ainsi les `N` longueurs des vecteurs.

Il faut enfin savoir neutraliser une diagonale : dans une matrice de couples, les termes `[i, i]` (« i avec lui-même ») sont souvent indésirables, typiquement avant une division. Pour ce faire, `np.fill_diagonal(mat, valeur)` écrit sur toute la diagonale, *en place* : la fonction ne renvoie rien, elle modifie directement la matrice.

EN PRATIQUE

Une division par une distance nulle guette tout calcul de forces entre paires. Plutôt qu'un test, mettez l'infini sur la diagonale : `1 / np.inf` vaut proprement `0.0`, et le terme fantôme disparaît du total.

Avec ces pièces et celles vues plus haut (les différences deux à deux, la règle de droite à gauche, une somme sur le bon axe), la force résultante sur chaque corps s'écrit sans aucune boucle ; je vous laisse trouver dans quel ordre les assembler. Seule l'intégration en temps reste une vraie boucle, car chaque pas dépend du précédent. Une précaution pour finir : pensez à **copier** les tableaux reçus en argument avant de les modifier, faute de quoi votre fonction abîmera les données de l'appelant (c'est le passage par référence, cf. §1.7).



Ce qu'on obtient à la fin de D6 : les trajectoires de trois corps (`ax.set_aspect("equal")` pour des orbites rondes).

3.8 Divers à connaître

Terminons par quatre outils que l'on rencontre vite en ligne, et sur lesquels il vaut mieux avoir un avis avant de s'en servir.

3.8.1 `@np.vectorize` : la commodité, pas la vitesse

Ce décorateur rend appelable sur un tableau une fonction scalaire, typiquement une fonction écrite avec un `if` que l'on ne sait pas exprimer autrement. Attention cependant : il ne compile rien, il boucle élément par élément. On y gagne donc en confort d'écriture, mais *pas en vitesse*.

3.8.2 `out=` : calculer sans allouer

Les *ufunc* acceptent un paramètre `out=` qui désigne le tableau où écrire le résultat : ainsi `np.multiply(t, 2, out=t)` double `t` **directement dans `t`**, sans tableau temporaire. L'économie de mémoire est réelle, cependant la lisibilité en souffre : réservez cette écriture aux cas où l'économie compte vraiment.

3.8.3 `np.clip` : plafonner, mais au bon moment

`np.clip(t, 0, 255)` ramène toutes les valeurs dans l'intervalle donné. Le souci classique est de le dégainer trop tard : clipper *après* un débordement `uint8` ne répare rien, car l'information est déjà perdue dans le calcul modulo. L'ordre correct est donc le suivant : on convertit d'abord, on calcule, on clippe, et on reconvertit (cf. §2.3).

3.8.4 `%timeit` : mesurer plutôt que croire

Dans un notebook, `%timeit expr` mesure le temps d'exécution d'une expression en la relançant plusieurs fois et en moyennant les mesures ; `%%timeit` en première ligne mesure la cellule entière. Avant d'affirmer qu'une version est plus rapide qu'une autre, mesurez : c'est le seul argument recevable dans une discussion sur la performance.

CHAPITRE 4

pandas — la table de données

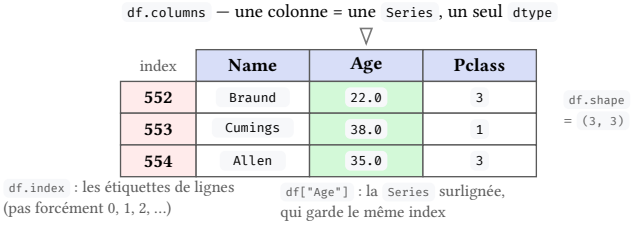
Les données de la vraie vie ne se présentent presque jamais sous forme de matrices : ce sont des *tables*, c'est-à-dire une observation par ligne (un passager du Titanic, un marathon, une région), plusieurs informations par observation, et des colonnes de types différents (des nombres, des textes, des dates). Or c'est précisément ce que `numpy`, homogène par construction, ne sait pas bien représenter. `pandas`, qui est bâti au-dessus de `numpy`, apporte ce qui manque : la table hétérogène et, surtout, son *indexation*.

Deux types vont nous suffire pour tout faire : la table, `DataFrame`, et la colonne (ou la ligne), `Series`. On importe par convention :

```
1 import pandas as pd
```

4.1 Anatomie d'une DataFrame

Le schéma ci-dessous est la carte mentale à garder devant les yeux pendant tout le cours : une grille de valeurs, encadrée par deux index.



Une `DataFrame` se comprend comme *un dictionnaire de colonnes* : les clés sont les noms de colonnes (`df.columns`), et les valeurs sont des objets `Series`. Chaque `Series` porte des valeurs d'un seul `dtype` (une colonne d'âges est en `float64`, une colonne

de noms en `object`), ainsi que *le même index de lignes que la table* (`df.index`) : c'est cet index commun qui garantit que les colonnes restent alignées entre elles.

Arrêtons-nous un instant sur le vocabulaire, car il structure tout le chapitre : les *indices* comptent les positions à partir de 0, tandis que les *index* sont les étiquettes que l'on a choisies (noms de colonnes, identifiants de lignes). Tant qu'on n'a rien choisi, pandas fabrique un `RangeIndex` (0, 1, 2…) et les deux notions coïncident ; en revanche, dès que l'on indexe par `PassengerId`, elles divergent. Notez enfin que `df.shape` renvoie `(lignes, colonnes)`, et que `len(df)` compte les *lignes* (`shape[0]`).

> **`df["age"]` vs `df[["age"]]`**
>
> Une *clé* renvoie la colonne, donc une **`Series`** : `df["Age"]`. Une *liste de clés* renvoie quant à elle une sous-table, donc une **`DataFrame`**, même si la liste ne compte qu'une seule colonne : `df[["Age"]]`. La nuance compte dès qu'une fonction attend l'un ou l'autre. Notez que `df.Age` est un raccourci de lecture commode (mais impossible si le nom contient un espace).

4.2 Lire et écrire un CSV

Un fichier CSV (*comma-separated values*) est la forme la plus simple que puisse prendre une table : une observation par ligne, et des valeurs séparées par un caractère convenu, qui n'est d'ailleurs pas toujours une virgule. D'où le premier réflexe que je vous demande d'acquérir : **regarder le fichier** (avec `%cat` ou VS Code) avant de le charger, afin de repérer le séparateur, l'entête et les éventuelles lignes de commentaires.

```
df = pd.read_csv("f.csv", sep=";", index_col="id")
df = pd.read_csv(DATA, sep="\t", names=["city", "year", ...])
df.to_csv("out.csv", sep=";", index=False)
```

Voici les paramètres qui servent tout le temps :

`sep=`	le séparateur (`,` par défaut, souvent `;` ou `\t` à l'usage)
`index_col=`	colonne à utiliser comme index des lignes
`names=`	les noms de colonnes quand le fichier n'a *pas d'entête* (sinon la 1re ligne de données est prise pour l'entête !)
`header=None`	dire explicitement qu'il n'y a pas d'entête
`skiprows=3`	sauter des lignes de commentaires
`index=False`	(`to_csv`) ne pas écrire la colonne d'index

Notez que l'index peut aussi se choisir après coup : `df = df.set_index("id")` (ou `inplace=True`), et l'opération inverse `df.reset_index()` retransforme l'index en colonne ordinaire (nous nous en resservirons avant un `merge`). Attention cependant à un détail qui tombe régulièrement au quiz : `set_index` *consomme* la colonne. Ainsi, sur une table `(2, 2)` de colonnes `id` et `v`, `df.set_index("id")` a pour forme `(2, 1)` et pour seule colonne `['v']`.

4.3 Premier regard sur une table

Face à un fichier inconnu, nous jouerons toujours la même gamme : les premières lignes, la structure, puis les statistiques.

```
df.head(5); df.tail(5)      # premières / dernières lignes
df.info()      # par colonne : nom, nb de valeurs NON MANQUANTES, dtype
df.describe()  # colonnes numériques : count, mean, std, min, quartiles, max
df.dtypes      # le type de chaque colonne
```

`info()` est le plus dense des trois : en une seule sortie, il donne le nom, le compte de valeurs *non manquantes* et le `dtype` de chaque colonne. C'est donc lui qui révèle d'un coup d'œil les colonnes trouées et les nombres lus comme du texte.

Pour explorer une colonne en particulier, nous disposons de deux outils. Le premier, `value_counts()`, compte les occurrences de chaque valeur :

```
df["Embarked"].value_counts()
```

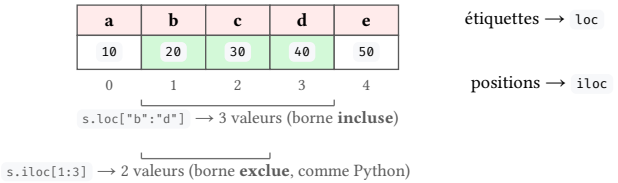
> RÉSULTAT `S 644` `C 168` `Q 77` : une `Series` indexée par les valeurs distinctes, comptées par ordre décroissant (`normalize=True` : en proportions)

Le second, `unique()`, renvoie le *tableau* des valeurs distinctes ; pour obtenir leur *nombre*, on écrira `len(df["Pclass"].unique())`, ou directement `nunique()`. Ne confondez pas cette dernière avec `count()`, qui compte les valeurs *non manquantes*.

4.4 `loc` et `iloc`

Contrairement à `numpy`, où `tab[i]` désigne une ligne, `df[x]` désigne une *colonne* : on ne peut donc pas écrire `df[ligne, colonne]`. Les accès en deux dimensions passent par deux accesseurs dédiés, qui prennent leurs arguments dans l'ordre naturel *(ligne, colonne)* : `df.loc[...]` travaille avec les *étiquettes*, tandis que `df.iloc[...]` travaille avec les *positions* (« i comme *integer* »).

Ainsi `df.loc[5, "Name"]` lit le nom de la ligne *d'index* 5 (qui n'est pas forcément la sixième ligne !), tandis que `df.iloc[5, 3]` lit la sixième ligne, quatrième colonne.



À RETENIR

Le slice `loc` *inclut* la borne haute (on désigne des étiquettes, il n'y a pas de « juste avant » naturel), tandis que le slice `iloc` l'*exclut*, comme partout en Python : `s.loc["b":"d"]` donne 3 valeurs, `s.iloc[1:3]` en donne 2. « Les lignes 10 à 12 » en comptant à partir de 1, c'est donc `df.iloc[9:12]`. Notez que `iloc` accepte les indices négatifs (`df.iloc[-1, 0]`), et que la colonne d'index ne compte pas dans les positions.

Dans chaque direction, on peut mettre au choix une étiquette, une liste, un slice ou un masque, et rien n'interdit de mélanger les genres, comme dans `df.loc[df["Age"] > 70, ["Name", "Sex"]]`. La dimension du résultat suit la sélection : une sélection 2D renvoie une `DataFrame`, une sélection 1D une `Series`, et une sélection 0D la valeur elle-même.

4.5 Masques et sélections

Le filtrage fonctionne comme en `numpy` : une comparaison vectorisée fabrique une `Series` de booléens (un masque), et indexer la table par ce masque conserve les lignes à `True`.

```
df[df["Sex"] == "female"]                    # filtrer des lignes
df[(df["Sex"] == "female") & (df["Age"] > 60)]  # & | ~  + parenthèses
```

Les règles de combinaison sont les mêmes qu'au chapitre précédent (§3.2) : `&`, `|`, `~`, avec chaque condition entre parenthèses, et jamais `and` / `or`, qui lèvent l'erreur « `ValueError: The truth value of a Series is ambiguous` ».

Pour filtrer sur du texte, les colonnes de chaînes offrent l'accesseur `.str`, qui applique les méthodes des chaînes à toute la colonne d'un coup :

```
df[df["Name"].str.contains("Mrs")]     # aussi .str.replace, .str.lower...
```

Notez que le même mécanisme existe pour les dates (`.dt` , cf. chapitre 6) et pour les catégories (`.cat`).

4.6 Créer, supprimer, convertir des colonnes

La table étant un dictionnaire de colonnes, *ajouter* une colonne revient à une simple affectation de clé, le plus souvent à partir d'un calcul sur les colonnes existantes :

```
df["Family"] = df["SibSp"] + df["Parch"] + 1
```

Pour *supprimer* une colonne, `drop` renvoie une nouvelle table (ou modifie la table sur place avec `inplace=True`) :

```
df = df.drop(columns=["Cabin"])
```

Pour *convertir* enfin, `astype` change le type d'une colonne propre ; quand la colonne est sale (des nombres mêlés de texte), `pd.to_numeric` avec `errors="coerce"` convertit ce qui peut l'être et transforme le reste en `NaN` :

```
s = pd.Series(["12", "x", "7"])
pd.to_numeric(s, errors="coerce").sum()
```

RÉSULTAT `19.0` : le `"x"` devient `NaN` , ignoré par `sum`

4.7 Valeurs manquantes : `NaN`

Les trous dans les données sont représentés par `NaN` (*Not a Number*). Première surprise : leur simple présence suffit à changer le type d'une colonne ! En effet, `NaN` est un flottant :

```
pd.Series([1, 2, None]).dtype
```

RÉSULTAT `dtype('float64')` : un `NaN` force le passage des entiers en flottants

Pour localiser ces trous, `isna()` construit un masque de même forme que la table (son contraire est `notna()`). Combiné aux sommes, ce masque fournit tous les diagnostics utiles :

```
df.isna().sum()       # nb de NaN PAR COLONNE (axis=0 par défaut)
df.isna().sum(axis=1) # par ligne ;  total : .sum().sum()
```

Pour traiter les valeurs manquantes, deux gestes sont possibles : les remplacer (`s.fillna(0)`) ou les supprimer (`s.dropna()`). Notez que les agrégations (`sum` , `mean` …) les ignorent de toute façon.

PIÈGE

`fillna` et `dropna` *ne modifient pas* l'objet : elles renvoient une nouvelle série. Ainsi, `s.fillna(0)` seul sur sa ligne ne fait rien : il faut réaffecter le résultat, avec `s = s.fillna(0)`.

4.8 Modifier : la règle d'or

Il nous reste à aborder la question qui fâche : comment modifier *une partie* d'une table ? L'erreur classique consiste à sélectionner d'abord, puis à modifier ensuite :

```
sub = df[df["Pclass"] == 1]
sub["Age"] = 0            # la colonne de df n'est PAS modifiée !
```

Ce `df[...]` suivi d'un `[...] =` est un *chaînage d'indexation* : la sélection intermédiaire se comporte comme une copie, et pandas proteste avec un `SettingWithCopyWarning` (« *A value is trying to be set on a copy of a slice from a DataFrame* »). Même quand « ça marche », c'est par accident : on ne peut pas compter sur le résultat.

La bonne écriture fait la sélection et l'affectation *en un seul* `loc` :

```
df.loc[df["Pclass"] == 1, "Age"] = 0
```

À RETENIR

Pour modifier une sous-partie d'une table, tout doit tenir dans un seul `df.loc[lignes, colonnes] = valeur`. Et si l'on veut vraiment travailler sur une copie indépendante, on la demande explicitement : `sub = df[df["Pclass"] == 1].copy()`.

4.9 Les gestes mis bout à bout

Pour fixer les idées, déroulons ensemble la gamme complète sur un exemple : un relevé de qualité de l'air, une ligne par mesure, avec une station, une date et une concentration. Le fichier vient d'un capteur, sans entête et avec un séparateur exotique ; le premier réflexe est de le regarder, puis de nommer les colonnes soi-même :

```
df = pd.read_csv("releves.txt", sep="|",
                 names=["station", "annee", "pm10"])
```

Viennent ensuite les sélections, avec un masque simple, un masque combiné, puis des positions :

```
df[df["annee"] == 2024]
df[(df["station"] == "Opéra") & (df["pm10"] > 50)]
df.iloc[:5]                       # les 5 premières mesures
```

Puis viennent les extractions, où la nuance entre `Series` et `DataFrame` refait surface, ainsi que les décomptes :

```
df.loc[df["station"] == "Opéra", "pm10"]            # -> une Series
df.loc[df["station"] == "Opéra", ["annee", "pm10"]]  # -> une DataFrame
df["station"].unique()            # un ndarray de stations
df["pm10"].mean()
df["annee"].value_counts().sort_index()   # mesures par année
```

C'est mot pour mot la gamme du devoir D3, appliquée à d'autres données. Le devoir y ajoute des *durées* de course (`"2:06:29"`) : retenez bien que ce ne sont pas des dates, et leur traitement (`pd.to_timedelta` et l'accesseur `.dt`) vous attend en §6.2.

CHAPITRE 5

pandas — trier, regrouper, croiser

Dans le chapitre précédent, nous avons appris à *lire* une table : la charger, la regarder, en extraire des lignes et des colonnes. Dans celui-ci, nous allons apprendre à la faire *parler*. En effet, les questions que l'on pose à de vraies données ont presque toujours la même forme : « combien, en moyenne, par catégorie ? », « qui arrive en tête ? », « que donne le croisement de deux critères ? », « et si on rapprochait ce fichier de cet autre ? ». pandas y répond avec un petit nombre d'opérations (trier, regrouper, croiser, découper en tranches, fusionner) que l'on enchaîne comme des briques. Elles font la matière de la séance 4 et l'ossature entière du devoir D4.

5.1 Trier

Trier répond au plus court à la question « quelles sont les trois plus grandes valeurs ? », et c'est surtout un préalable obligatoire dès que l'on manipule des séries temporelles (cf. §6). Nous disposons de deux méthodes, selon que l'on trie sur les *valeurs* d'une ou plusieurs colonnes, ou sur les *étiquettes* de l'index.

```
1 df.sort_values(by=["Pclass", "Age"], ascending=[True, False])
```

On donne à `sort_values` une colonne, ou une liste de colonnes : l'ordre est alors lexicographique, c'est-à-dire que le 2e critère ne sert qu'à départager les ex æquo du premier. `ascending` suit la même logique : un booléen unique pour tout le tri, ou un booléen par critère comme ici (classe croissante, et à l'intérieur de chaque classe, âge décroissant). La méthode jumelle `df.sort_index()` réordonne quant à elle les lignes *selon l'index* ; c'est ce que l'on fait après un `groupby` quand on veut retrouver l'ordre alphabétique des clés plutôt que l'ordre des valeurs agrégées.

Que deviennent les valeurs manquantes, qu'un tri ne sait par définition pas comparer aux autres ? Vérifions :

```
pd.Series([3, None, 1, 2]).sort_values().index
```

RÉSULTAT `Index([2, 3, 0, 1], dtype='int64')` : 1, 2, 3, puis le `NaN`

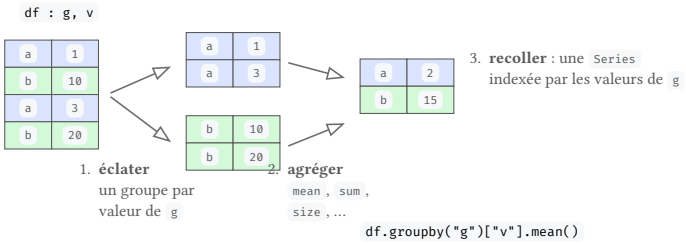
À RETENIR

Les `NaN` sont rangés *en fin de tri*, que l'ordre soit croissant ou décroissant : en effet, ils ne sont ni les plus petits ni les plus grands, ils sont mis de côté. Au besoin, `na_position="first"` les ramène devant.

Notez que le tri conserve l'index : une ligne ne perd pas son identité parce qu'on l'a déplacée, et ses étiquettes la suivent. Si l'on veut vraiment des positions 0…n−1 après coup, `reset_index(drop=True)` renumérote. Enfin, pour savoir si une série est *déjà* triée sans avoir à la trier, `s.is_monotonic_increasing` répond par un booléen (avec sa jumelle `is_monotonic_decreasing`) : c'est ce que vérifient les `assert` du devoir D4.

5.2 `groupby` : éclater, agréger, recoller

Dès qu'une question contient le mot « par » (par classe, par sexe, par région, par filière), c'est un `groupby`. L'opération tient en trois temps, que le diagramme ci-dessous déroule : on éclate la table en sous-tables selon la clé, on agrège chacune séparément, puis on recolle les résultats en une seule ligne par groupe. Notez que l'on n'écrit que le deuxième temps : pandas se charge des deux autres.



Le premier réflexe à acquérir est de comprendre que `df.groupby("Sex")` ne calcule *rien* du tout. En effet, l'objet renvoyé, un `DataFrameGroupBy`, n'est que la *description de la partition* : « voici comment je découperais la table si on me le demandait ». Il faut encore dire *quoi* agréger et *comment*, c'est-à-dire choisir une colonne puis appeler une fonction d'agrégation.

```
df.groupby("Pclass")["Age"].mean()
```

Le résultat est une `Series` indexée par les valeurs distinctes de `Pclass` : une étiquette par groupe, avec l'âge moyen du groupe en face. Deux variantes comptent les lignes au lieu de les moyenner ; prenez garde, car elles ne donnent pas le même nombre :

```
df.groupby("Pclass").size()          # taille de chaque groupe
df.groupby("Pclass")["Age"].count()  # Age renseignés dans le groupe
```

À RETENIR

`size()` compte *toutes* les lignes du groupe, tandis que `count()` ne compte que les valeurs *non manquantes* de la colonne choisie. L'écart entre les deux mesure donc exactement les trous de cette colonne dans chaque groupe.

Notez que l'on n'est pas limité à une agrégation par appel : `agg` en accepte une liste, et renvoie alors une table au lieu d'une série :

```
df.groupby("g")["v"].agg(["mean", "max"])
```

RÉSULTAT une `DataFrame` : une ligne par groupe, *une colonne par fonction*

Les agrégations s'écrivent comme de simples chaînes de caractères, et le catalogue est large : `"mean"`, `"sum"`, `"size"`, `"count"`, `"min"`, `"max"`, `"std"`, `"median"`, `"first"`, `"nunique"`… Il reste une conséquence du « recollage » à connaître, car elle surprend tout le monde au moins une fois.

PIÈGE

Après un `groupby`, la clé passe dans l'*index* du résultat, et non dans ses colonnes : `g["Sex"]` lève un `KeyError`, alors que `g["female"]` fonctionne très bien. C'est gênant dès que l'on veut fusionner, tracer ou exporter le résultat.

```
df.groupby("Pclass", as_index=False)["Fare"].mean()
```

`as_index=False` demande à pandas de laisser `Pclass` en colonne ordinaire, sous un banal `RangeIndex` ; `reset_index()` obtient le même effet après coup. Retenez bien ce geste : c'est celui qui précède presque tous les `merge` (§5.5).

Rien n'empêche par ailleurs de grouper sur *plusieurs critères*, en passant une liste de clés : `df.groupby(["Pclass", "Sex"]).size()` renvoie une `Series` dont l'index est un *MultiIndex* à deux niveaux, c'est-à-dire dont les étiquettes sont des tuples. On y accède naturellement par tuple, avec `g.loc[(1, "female")]` ; on peut aussi ne donner

que le premier niveau, `g.loc[1]`, et l'on récupère alors la sous-série de toute la première classe. En cas de doute, `g.index.nlevels` répond `2`.

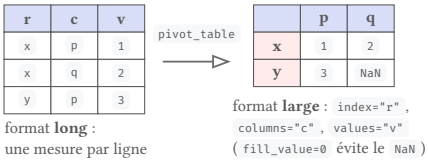
Une série à deux niveaux se lit cependant assez mal : c'est une longue colonne de tuples. Pour y remédier, `unstack()` prend un niveau d'index et le transforme en *colonnes*, ce qui donne enfin une table à deux dimensions, lisible et traçable. Par défaut, c'est le *dernier* niveau qui bascule ; `t.unstack(0)` bascule le premier.

```
t = df.set_index(["pays", "date"])["cas"].unstack()
```

RÉSULTAT lignes = `pays`, colonnes = les valeurs de `date` ; forme `(2, 2)`, colonnes `['d1', 'd2']`

5.3 `pivot_table` : le tableau croisé

Quand la question porte sur *deux* axes à la fois (le taux de survie par classe *et* par sexe, typiquement), l'enchaînement d'un `groupby` à deux clés puis d'un `unstack` finit par être fastidieux. `pivot_table` fait les deux d'un coup, avec des noms d'arguments qui décrivent ce que l'on veut obtenir plutôt que la mécanique employée.



format **long** : une mesure par ligne

format **large** : `index="r"`, `columns="c"`, `values="v"` (`fill_value=0` évite le `NaN`)

```
df.pivot_table(values="Survived", index="Pclass", columns="Sex")
```

Quatre arguments sont à retenir : `values` désigne ce que l'on met dans les cases ; `index` et `columns` désignent les critères qui deviennent respectivement les lignes et les colonnes ; `aggfunc`, enfin, dit comment agréger les valeurs qui tombent dans une même case. Or `aggfunc` vaut *la moyenne par défaut* : ainsi, avec une colonne `Survived` codée en 0/1, l'appel ci-dessus donne directement un tableau de *taux* de survie, sans que l'on ait rien demandé !

> **À RETENIR**
>
> `pivot_table` = `groupby` sur deux clés + `unstack`, en un seul appel. Le résultat est le même ; la différence est d'intention : `groupby` + `unstack` décrit la mécanique, `pivot_table` décrit le tableau que l'on veut obtenir.

Deux arguments de confort reviennent constamment dans le devoir D4. `fill_value=0` remplace les cases vides (celles dont la combinaison ligne × colonne

n'existe pas dans les données, comme le nucléaire en Bretagne), qui vaudraient sinon `NaN`. Quant à `aggfunc="sum"`, il sert même lorsqu'il n'y a qu'une valeur par case : la moyenne, elle, convertirait tous les entiers en flottants sans raison.

5.4 `pd.cut` : découper en intervalles

Le taux de survie « par âge » n'a aucun sens valeur par valeur : en effet, on n'a pas assez de passagers de 37 ans exactement. Par *tranche* d'âge, en revanche, la question devient intéressante. `pd.cut` fabrique cette colonne de tranches à partir d'une colonne continue, et l'on retombe alors sur un `groupby` ordinaire.

```
df["classe"] = pd.cut(df["Age"], bins=[0, 18, 65, 100],
                      labels=["jeune", "adulte", "senior"])
```

Retenez bien que les bornes de `bins` sont des *poteaux* et non des cases : $n+1$ bornes délimitent n intervalles, ici `(0, 18]`, `(18, 65]` et `(65, 100]`. La notation dit tout : borne basse *exclue*, borne haute *incluse*. Ce qui tombe hors des bornes extrêmes devient `NaN` ; ainsi, un âge de 102 ans ne serait dans aucune tranche.

> **PIÈGE**
>
> `labels` doit compter *un élément de moins* que `bins`, sans quoi pandas refuse. Attention aussi à l'oubli classique : une borne basse à `0` exclut la valeur `0` elle-même ; commencez à `-1` ou `-0.001` quand les zéros comptent.

Le résultat est de dtype `category`, et il est ordonné (« jeune » vient avant « adulte »), ce qui permet ensuite les comparaisons et un tri sensé. On l'exploite comme n'importe quelle colonne catégorielle : `value_counts()` pour les effectifs, ou `df.groupby("classe", observed=True)["Survived"].mean()` pour l'agrégat par tranche. L'option `observed=True` évite un avertissement et écarte les catégories restées vides.

5.5 `merge` : aligner deux tables sur une clé

Les données réelles arrivent rarement en un seul fichier : les productions d'un côté, les populations de l'autre, et une colonne commune pour les rapprocher. `merge` est le `JOIN` de SQL : il aligne deux tables *par la valeur d'une clé*, et non par leur position.

`a`

k	x
1	10
2	20
3	30

`b`

k	y
2	7
3	8
4	9

`a.merge(b, on="k")`

k	x	y
2	20	7
3	30	8

`inner` (défaut) : clés communes

`how="outer"`

k	x	y
1	10	NaN
2	20	7
3	30	8
4	NaN	9

toutes les clés, `NaN` aux absents ; `how="left"` : toutes les clés de `a`

```
pd.merge(eleves, notes, on="id")   # ou eleves.merge(notes, on="id")
```

Reste à décider quoi faire des clés qui n'existent que d'un côté : c'est le rôle de `how`, et personne ne peut trancher ce choix à votre place.

```
a.merge(b, on="k", how="left")
```

`"inner"` est le défaut : on ne garde que les clés présentes des deux côtés, ce qui veut dire que l'on *perd* silencieusement les autres. `"left"` et `"right"` gardent toutes les clés d'un côté et remplissent d'en face avec des `NaN` ; `"outer"` garde l'union des clés. Enfin, lorsque la clé ne porte pas le même nom des deux côtés, on précisera `left_on=` et `right_on=` ; lorsqu'elle est dans l'index, `left_index=True`.

EN PRATIQUE

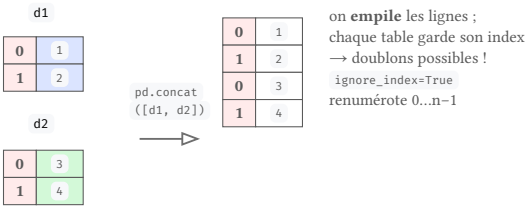
Une `Series` ne se fusionne pas directement : `merge` réclame deux `DataFrame`. `reset_index()` fait la conversion : l'ancien index devient une vraie colonne, utilisable comme clé de fusion.

EFFET MULTIPLICATIF

Si une clé apparaît p fois à gauche et q fois à droite, le merge produit $p \times q$ lignes pour cette clé : `a` (k=1,1) × `b` (k=1,1,1) donne *6 lignes* ! Comparez donc les `shape` avant et après : un merge qui grossit a trouvé des doublons.

5.6 `concat` : empiler

Là où `merge` aligne en largeur sur une clé, `concat` empile bêtement en hauteur des tables de même structure. C'est l'outil des données livrées en morceaux (un fichier par mois, un export par région), et celui des lignes de total que l'on ajoute au bas d'un tableau.



`pd.concat([df1, df2])` empile les lignes ; l'opération est n-aire, c'est-à-dire que la liste peut contenir autant de tables que l'on veut, et `axis=1` colle en largeur plutôt qu'en hauteur. Les colonnes absentes d'un côté sont remplies de `NaN` : c'est souvent le signe que l'on a empilé des choses qui n'allaient pas ensemble.

> **PIÈGE**
>
> *Chaque table garde son index* : `pd.concat([d1, d2])` produit ainsi un index `0, 1, 0, 1`. Les doublons d'étiquettes ne sont pas une erreur pour pandas, cependant `r.loc[0]` renvoie alors *toutes* les lignes d'étiquette 0, et non une seule. `ignore_index=True` renumérote proprement de 0 à n−1.

Un usage classique est l'ajout d'une ligne de synthèse au bas d'une table. Comme `concat` n'empile que des tables, il faut d'abord fabriquer une `DataFrame` d'une seule ligne à partir de la série des totaux : `to_frame()` fait la conversion, puis la transposition `.T` bascule la colonne obtenue en ligne.

5.7 Champions et podiums : `idxmax`, `nlargest`

La question « quelle région produit le plus ? » demande un *nom*, et non une valeur. Cette distinction est une source d'erreurs classique, et pandas la matérialise par deux méthodes différentes.

```
serie.idxmax()      # l'ÉTIQUETTE du max (idxmin : celle du min)
serie.max()         # la VALEUR du max
```

Appliquée à une `DataFrame`, `croise.idxmax()` travaille colonne par colonne et renvoie une `Series` : pour chaque colonne, l'étiquette de ligne où elle est maximale. Une seule ligne de code suffit donc à répondre à « pour chaque filière, quelle région est championne ? ». Et quand on veut un podium plutôt qu'un seul vainqueur :

```
serie.nlargest(3)      # les 3 plus grandes, triées, avec leur index
df.nlargest(3, "Fare") # même idée sur une colonne d'une table
```

`nsmallest` fait l'inverse. Notez enfin que sur une série indexée par des dates, `idxmax()` renvoie directement le `Timestamp` du pic : nous nous en servirons dans le devoir D5 pour dater le jour le plus fréquenté.

5.8 Construire une DataFrame par programme

Avant de lancer une idée sur 40 000 lignes, il vaut mieux la tester sur quatre ! D'où l'intérêt de savoir fabriquer une petite table à la main, en choisissant la forme qui colle à la façon dont les données se présentent.

```
pd.DataFrame({"x": [1, 2, 3], "y": [4, 5, 6]})     # les COLONNES
pd.DataFrame([{"x": 1, "y": 4}, {"x": 2, "y": 5}]) # les LIGNES
```

La première forme est la plus courante : un *dictionnaire de colonnes*, dont chaque clé donne un nom de colonne et chaque valeur la liste des cellules. La seconde est une *liste de lignes*, c'est-à-dire des enregistrements arrivant un par un, façon JSON. Deux variantes complètent le tableau : partir d'un `ndarray` numpy en lui collant des étiquettes, ou ne fabriquer qu'une seule colonne.

```
pd.DataFrame(tab_numpy, index=..., columns=...)  # un ndarray
pd.Series([10, 20], index=["a", "b"])            # une colonne
```

Dans tous les cas, les colonnes doivent avoir la même longueur, et `index=` fixe au besoin les étiquettes de lignes ; sans lui, pandas pose son `RangeIndex` habituel.

5.9 Les briques mises bout à bout

Pour voir comment ces opérations s'enchaînent, déroulons ensemble une petite étude complète : trois librairies d'une même enseigne, un fichier `ventes.csv` avec une ligne par couple magasin × rayon (`magasin`, `rayon`, `ca_keuros`), et un second fichier avec la surface de chaque magasin. La même mécanique, dans le même ordre, vous resservira pour le devoir D4, sur d'autres données et d'autres questions.

1. Charger et agréger. Première question : combien vend chaque magasin, tous rayons confondus ?

```
df = pd.read_csv("ventes.csv", sep=";")
total = (df.groupby("magasin")["ca_keuros"].sum()
           .sort_values(ascending=False))
```

`total` est une `Series` indexée par le nom du magasin, du plus gros vendeur au plus petit. Elle nous servira de fil rouge jusqu'à la fin.

2. Croiser. La question suivante porte sur deux axes : les ventes par magasin *et* par rayon. C'est le tableau croisé, avec les deux arguments de confort que nous venons de voir.

```
croise = df.pivot_table(values="ca_keuros", index="magasin",
                        columns="rayon", aggfunc="sum",
                        fill_value=0)
```

Nous obtenons trois lignes, quatre colonnes, et des zéros (et non des `NaN`) là où la combinaison n'existe pas dans le fichier : la gare ne vend pas de jeux.

3. Vérifier la cohérence. Deux calculs indépendants portant sur les mêmes données doivent se recouper : la somme d'une ligne du tableau croisé, c'est le total du magasin. Vérifions-le plutôt que de l'espérer, en prenant garde au fait que les deux objets ne sont pas triés dans le même ordre :

```
(croise.sum(axis=1) == total.loc[croise.index]).all()
```

Le `.loc[croise.index]` réordonne `total` (qui est trié par valeurs) dans l'ordre du tableau croisé, trié par nom : sans lui, on comparerait des magasins différents ligne à ligne ! Ce réflexe de réalignement avant comparaison sert dans toutes les études.

4. Fusionner. Le chiffre d'affaires brut favorise mécaniquement les grands magasins ; pour comparer ce qui est comparable, nous le rapportons au mètre carré, et pour ce faire, nous rapprochons les deux fichiers par la colonne `magasin` :

```
par_m2 = total.reset_index().merge(surfaces, on="magasin")
par_m2["ca_par_m2"] = par_m2["ca_keuros"] / par_m2["surface_m2"]
```

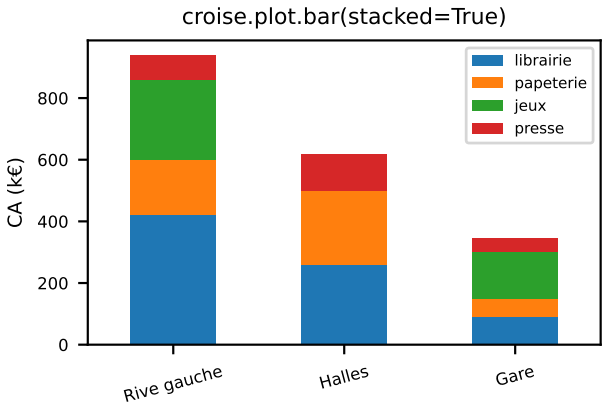
Le `reset_index()` est indispensable : `total` est une `Series`, or `merge` réclame deux `DataFrame`. Le résultat mérite un coup d'œil au `shape` : on doit toujours trouver trois lignes, sans quoi un magasin se serait perdu en route.

5. Classer et exporter. Reste à ranger les magasins en catégories lisibles, puis à écrire le résultat sur le disque :

```
par_m2["classe"] = pd.cut(par_m2["ca_par_m2"], bins=[0, 2, 5, 1e9],
                          labels=["faible", "moyen", "fort"])
par_m2.to_csv("rendement.csv", sep=";", index=False)
```

La borne haute `1e9` sert de poteau « et tout le reste » : sans elle, les valeurs au-delà de la dernière borne sortiraient des intervalles et récolteraient un `NaN`. Quant à `index=False`, il évite d'écrire dans le CSV une colonne de numéros de ligne dont personne n'a l'usage.

L'étude se termine par une figure, qui montre d'un coup d'œil ce que trois tableaux disent moins bien : la taille de chaque magasin *et* la composition de ses ventes.



`croise.plot.bar(stacked=True)` : une barre par ligne d'index, une couleur par colonne, empilées ; puis `ax.set_ylabel(...)`, `plt.tight_layout()`, `plt.savefig(...)`.

CHAPITRE 6

Le temps et les figures

Ce dernier chapitre réunit les deux gestes qui closent une analyse : lire une table *dans le temps* (par jour, par mois, en tendance), puis la *montrer*. Les deux se croisent en permanence dans les devoirs, où l'on finit toujours par tracer une série. Nous ajoutons donc aux imports habituels celui de matplotlib :

```
import matplotlib.pyplot as plt
```

6.1 Dates : `datetime64`, `Timestamp`, `to_datetime`

Tant qu'une date reste une *chaîne de caractères*, on ne peut presque rien en faire : ni la soustraire à une autre, ni en extraire le mois, ni découper la série « de janvier à mars ». Le premier geste, face à une colonne de dates, est donc de la convertir en type temporel.

pandas distingue deux types temporels, qu'il ne faut jamais confondre. Un *instant* (`np.datetime64`, exposé côté pandas comme `Timestamp`) répond à la question « quand ? ». Une *durée* (`np.timedelta64`, ou `Timedelta`) répond quant à elle à « combien de temps ? ». Leur arithmétique découle du bon sens : la différence de deux instants est une durée (`TS - TS → TD`), un instant décalé d'une durée reste un instant (`TS ± TD → TS`), le quotient de deux durées est un nombre (`TD // TD → int`) et le reste une durée (`TD % TD → TD`). En revanche, `TS + TS` ne veut rien dire, et pandas le refuse.

Vérifions sur deux dates du semestre :

```
d = pd.Timestamp("2026-10-20") - pd.Timestamp("2026-09-08")
type(d).__name__, d.days
```

```
RÉSULTAT  ('Timedelta', 42)
```

Remarquez que la soustraction n'a pas renvoyé un nombre de jours, mais un objet `Timedelta`, dont l'attribut `.days` vaut 42. C'est ce détour par un type dédié qui

rend l'arithmétique des dates fiable : les mois de longueurs inégales et les années bissextiles sont gérés pour nous.

À l'usage, les dates arrivent d'un CSV sous forme de texte, et c'est `to_datetime` qui convertit toute la colonne d'un coup :

```
df["date"] = pd.to_datetime(df["date"], format="%Y-%m-%d")
```

Le résultat est une `Series` de dtype `datetime64`. Notez que l'opération ne touche *ni l'index ni le reste de la table* : d'où la réaffectation explicite de la colonne. Préciser `format=` n'est pas obligatoire (pandas devine tout seul), mais je vous le recommande : c'est plus rapide, et cela évite les lectures surprenantes d'un jour pris pour un mois. Les codes usuels sont `%Y` (année sur quatre chiffres), `%m` (mois), `%d` (jour), et `%H:%M:%S` pour l'heure. Enfin, une date invalide ne fait pas échouer la conversion : elle devient `NaT` (*Not a Time*), le `NaN` des dates.

Une fois la colonne convertie, l'accesseur `.dt` (l'équivalent, pour les dates, de ce que `.str` est pour le texte) donne accès à tous les attributs temporels :

```
s.dt.year, s.dt.month, s.dt.day        # entiers
s.dt.dayofweek                         # 0 = lundi ... 6 = dimanche
s.dt.day_name()                        # "Tuesday" (en anglais !)
```

PIÈGE

Les *attributs* de `.dt` s'écrivent sans parenthèses (`s.dt.year`), les *méthodes* avec (`s.dt.day_name()`). Notez aussi que `day_name()` répond en anglais (`"Tuesday"`, jamais `"mardi"`), quelle que soit la langue du système.

Lorsque les dates sont passées en index, les mêmes attributs sont disponibles directement, sans `.dt` : `velos.index.year`, `.month`, `.dayofweek`. C'est la porte d'entrée des profils temporels de la section suivante.

6.2 Durées : `to_timedelta` et `.dt`

Un temps de course comme `"2:06:29"` ressemble à une heure, mais n'en est pas une : c'est une *durée*, et rien ne dit quel jour elle a été réalisée. Cette confusion est le piège classique du devoir D3 ; pandas fournit donc une fonction de conversion bien distincte de `to_datetime` :

```
df["duration"] = pd.to_timedelta(df["duration"])   # -> Timedelta
```

La colonne obtenue contient des `Timedelta`, sur lesquels on peut trier, faire une moyenne ou calculer un écart. Cependant, une durée n'est pas un nombre : pour

tracer une courbe ou comparer des ordres de grandeur, on la ramène le plus souvent à des secondes, ou bien on la décompose unité par unité.

```
df["duration"].dt.total_seconds()   # en secondes (float)
df["duration"].dt.components        # table hours/minutes/...
```

`total_seconds()` renvoie un flottant, directement exploitable par numpy et matplotlib. `.components` renvoie quant à lui une table avec une colonne par unité (`days`, `hours`, `minutes`, `seconds`…), bien commode pour réafficher « 2 h 06 » sans arithmétique manuelle.

PIÈGE

Passer une telle colonne à `parse_dates=` de `read_csv` serait un contresens : `"2:06:29"` est une durée, pas une date, et pandas y verrait une heure de la journée courante (c'est le piège du devoir D3 !). La bonne fonction est `to_timedelta`, appliquée après la lecture.

Enfin, la division entière et le modulo fonctionnent sur les durées, ce qui évite souvent le détour par les secondes :

```
une_heure = np.timedelta64(1, "h")
d // une_heure      # nombre d'heures entières (int)
d % une_heure       # ce qui reste (Timedelta)
```

Le quotient donne les heures pleines, et le reste ce qu'il faudra convertir en minutes : c'est la décomposition que l'on écrirait à la main, mais sans risque d'erreur d'unité.

6.3 Séries temporelles : `DatetimeIndex`

Une colonne de dates est déjà utile. Cependant, pandas réserve ses meilleurs outils temporels (la sélection par période, `resample`) aux séries dont *l'index* est fait de dates. Le geste standard tient en deux appels enchaînés :

```
s = (df.set_index("date")["comptage"]   # index -> DatetimeIndex
       .sort_index())                   # et on TRIE par date
```

`set_index` promeut la colonne `date` au rang d'index, et la sélection de colonne réduit la table à une seule série. Ne sautez pas le tri : la sélection par intervalle suppose un index ordonné, et l'oublier donne des résultats vides ou aberrants, sans message d'erreur !

Quand il faut fabriquer un axe de temps de toutes pièces (pour simuler, ou pour réaligner une série trouée), `date_range` produit un index régulier :

```
idx = pd.date_range("2020-01-01", periods=100, freq="D")
```

On obtient ici cent jours consécutifs à partir du 1er janvier 2020 ; `freq=` accepte les mêmes codes que `resample` (`"D"`, `"W"`, `"ME"`…).

Avec un `DatetimeIndex`, `loc` accepte des *chaînes partielles*. Ainsi, `"2020-02"` désigne tout février et `"2020"` toute l'année, sans que l'on ait à écrire la moindre borne. Et comme c'est du `loc`, les *bornes sont incluses* :

```
len(s.loc["2020-01-10":"2020-01-20"]), len(s.loc["2020-02"])
```

RÉSULTAT `(11, 29)` : 11 jours bornes incluses ; tout février 2020 (bissextile)

Onze jours et non dix : du 10 au 20 *inclus*, contrairement aux tranches de listes Python. Et vingt-neuf jours en février, puisque 2020 est bissextile : pandas connaît le calendrier !

6.3.1 Diagnostiquer une série qui arrive sale

Les données de terrain arrivent rarement propres : on y trouve des dates en double (deux exports concaténés), des valeurs aberrantes (un capteur en panne qui enregistre `0` ou une valeur folle), des jours sans mesure. Trois outils permettent de *voir* ces défauts ; or c'est bien le diagnostic qui compte, avant toute réparation.

Pour les doublons, `duplicated()` (sur une série ou sur un index) renvoie un masque booléen, avec `False` à la première apparition d'une valeur et `True` aux suivantes :

```
pd.Index([3, 1, 3, 3]).duplicated()
```

RÉSULTAT `[False, False, True, True]` : combiné à `~`, il isole les premières occurrences

Pour les valeurs aberrantes, ce sont les masques du chapitre 4 qui travaillent : `s == 0`, `s > 3 * s.median()`… Une affectation par masque transforme alors ces valeurs en `NaN`, c'est-à-dire en manquants *déclarés* : c'est bien préférable à des zéros qui écrasent silencieusement les moyennes.

Pour les trous enfin, `interpolate()` remplace chaque `NaN` par une valeur déduite de ses voisins, linéairement par défaut :

```
pd.Series([10, np.nan, np.nan, 40.0]).interpolate()
```

RÉSULTAT `[10.0, 20.0, 30.0, 40.0]` : l'alternative à `fillna` quand les données ont un ordre naturel

Et comme toujours, on vérifie plutôt que l'on suppose : `describe()` pour la vraisemblance des valeurs, `isna().sum()` pour les trous restants, `s.index.is_monotonic_increasing` pour l'ordre. Le devoir D5 vous fera dérouler ce diagnostic en entier sur une vraie série ; ce sera à vous de choisir l'ordre des réparations.

Une série propre se lit ensuite volontiers en *variations* plutôt qu'en valeurs absolues. `pct_change()` calcule la variation relative d'une valeur à la suivante :

```
1 pd.Series([100, 110, 99]).pct_change()
```

RÉSULTAT `[NaN, 0.10, -0.10]` : première case `NaN`, ×100 pour des pourcentages

La première case est toujours `NaN` : en effet, il n'y a pas de valeur précédente à laquelle se comparer.

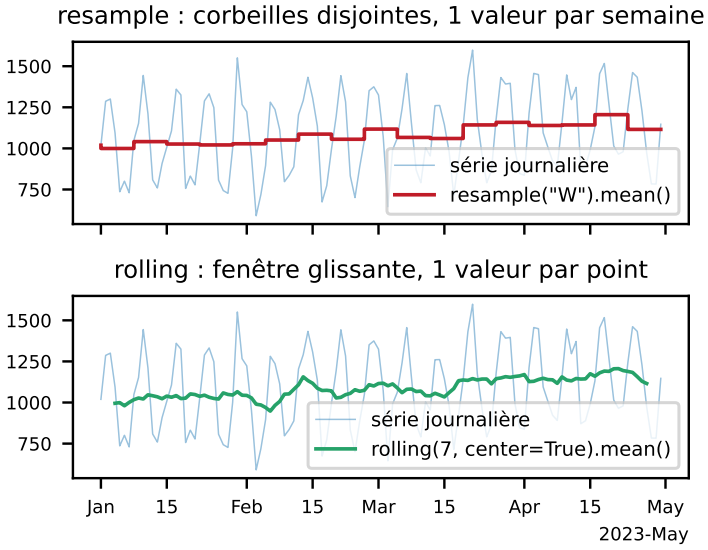
6.4 `resample` vs `rolling`

Deux verbes se ressemblent et sont pourtant opposés. Pour les distinguer une fois pour toutes, regardons ce qu'ils font au *nombre de valeurs*.

À RETENIR

`resample` découpe le temps en *corbeilles disjointes* : une valeur par corbeille, la série raccourcit. `rolling` fait glisser une *fenêtre* : une valeur par point, la série garde sa longueur. Les deux ne font que grouper : on agrège ensuite (`.mean()`, `.sum()`…).

La figure ci-dessous met les deux mécanismes côte à côte : à gauche des tranches qui ne se chevauchent pas, à droite une fenêtre qui avance d'un cran à chaque fois.



`resample` s'écrit comme un `groupby` temporel, avec un code de fréquence, et exige un index fait de dates :

```
s.resample("W").mean()    # moyenne par semaine
s.resample("ME").mean()   # par mois (Month End)
s.resample("YE").sum()    # total par année
```

Les codes se lisent comme des abréviations : `"W"` pour *week*, `"ME"` pour *month end* (fin de mois ; c'était `"M"` avant pandas 2.2), `"YE"` pour *year end*. Retenez que le choix de l'agrégation compte autant que celui de la fréquence : une moyenne journalière et un total mensuel ne racontent pas la même histoire.

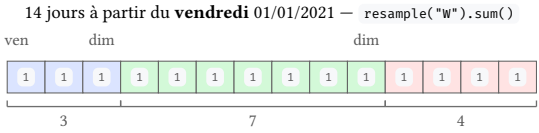
`rolling` s'écrit, lui, avec un nombre de points, et sert à lisser une courbe trop agitée pour être lisible :

```
s.rolling(7).mean()                # fenêtre de 7 valeurs
s.rolling(7, center=True).mean()   # fenêtre centrée
```

LES BORDS

`rolling(7)` a besoin de 7 valeurs : les *6 premières* sorties sont donc `NaN` (`[nan, nan, 2.0, 3.0, 4.0]` pour `rolling(3)` sur 1…5) ; avec `center=True`, les `NaN` se répartissent entre les deux extrémités. Quant aux semaines `"W"` de `resample`, elles se terminent le *dimanche* : les corbeilles des bords sont incomplètes.

Ces bords incomplets expliquent bien des comptages qui semblent faux. Le diagramme suivant les rend visibles sur quatorze jours commençant un vendredi.



les semaines `"W"` se terminent le **dimanche** → corbeilles incomplètes aux bords : `[3, 7, 4]`

6.4.1 Profils : `groupby` sur l'index

Pour répondre à la question « y a-t-il plus de vélos le mardi que le dimanche ? », on ne découpe plus le temps en tranches successives : on *regroupe* des jours qui ne se suivent pas. C'est un `groupby` ordinaire (§5.2), appliqué à un attribut de l'index.

```
velos.groupby(velos.index.dayofweek).mean()  # profil hebdo
velos.groupby(velos.index.month).mean()      # profil mensuel
```

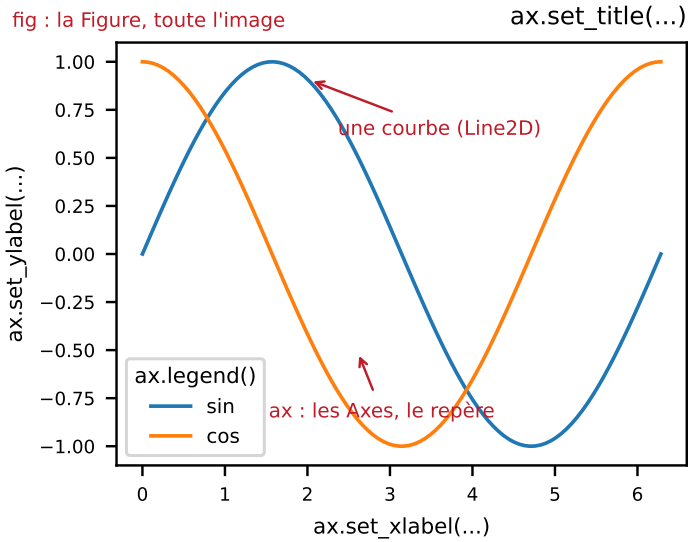
Le premier regroupe par jour de la semaine (`0` = lundi … `6` = dimanche), le second par mois (`1` à `12`). En croisant deux attributs, on obtient une série à double index, que `unstack` remet à plat en tableau à deux entrées, avec une ligne par mois et une colonne par année :

```
par_mois = velos.groupby([velos.index.year,
                          velos.index.month])
par_mois.mean().unstack(0)     # lignes = mois, colonnes = années
```

Sous cette forme, la table se trace directement : une courbe par année, superposées mois par mois.

6.5 matplotlib : figure et axes

Avant de tracer quoi que ce soit, il nous faut distinguer deux objets que l'on confond souvent au début, et dont dépend tout le vocabulaire de matplotlib. La figure ci-dessous les nomme.



Dans `fig, ax = plt.subplots(figsize=(8, 4))`, `fig` est la *figure* entière, c'est-à-dire l'image que l'on sauvegardera dans un fichier, tandis que `ax` est un *repère* (`Axes`) dessiné dessus, avec ses deux échelles, son titre et sa légende. Une figure peut porter plusieurs repères : c'est tout l'objet des grilles de sous-figures présentées plus loin.

Tout l'habillage (titre, étiquettes d'axes, légende, sauvegarde) s'écrit ensuite de deux façons strictement équivalentes. Le style *pyplot* s'adresse implicitement au « repère courant » : c'est le plus court, idéal pour une figure unique. Le style *objet* nomme quant à lui explicitement le repère visé : c'est le seul praticable dès qu'il y en a plusieurs. Les deux colonnes ci-dessous produisent exactement la même image.

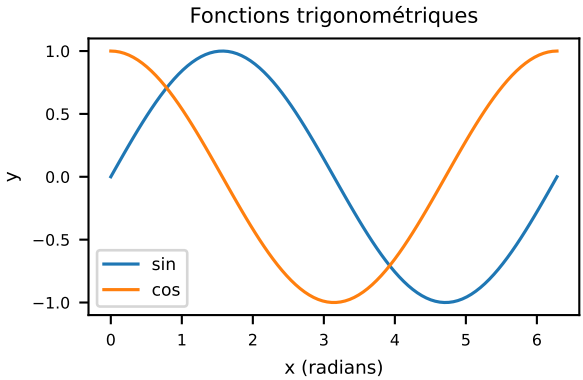
```python
# style pyplot (repère courant)
plt.plot(x, y1, label="sin")
plt.plot(x, y2, label="cos")
plt.title("Titre")
plt.xlabel("x")
plt.ylabel("y")
plt.legend()
plt.savefig("fig.png", dpi=120)
plt.show()
```

```python
# style objet (explicite)
fig, ax = plt.subplots()
ax.plot(x, y1, label="sin")
ax.plot(x, y2, label="cos")
ax.set_title("Titre")
ax.set_xlabel("x")
ax.set_ylabel("y")
ax.legend()
fig.savefig("fig.png", dpi=120)
```

La correspondance est mécanique : `plt.title` devient `ax.set_title`, `plt.xlabel` devient `ax.set_xlabel`. Seule la sauvegarde change de destinataire (`fig.savefig`), puisque l'on écrit l'image entière et non un repère. Je vous conseille de choisir un style et de vous y tenir dans une même cellule : les mélanger fonctionne, mais rend le code impossible à relire.

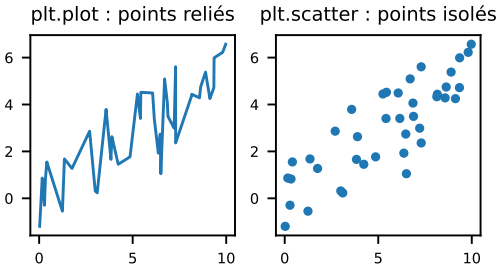
> **À RETENIR**
>
> `plt.legend()` affiche les `label=` passés aux tracés : sans `label`, pas de légende, et sans `legend()`, rien ne s'affiche non plus. Enfin, `plt.savefig(...)` se place *avant* `plt.show()` : `show` referme la figure, et après lui on sauverait un fichier vide.

Voici ce que donnent ces quelques lignes : deux courbes, des axes nommés, une légende.

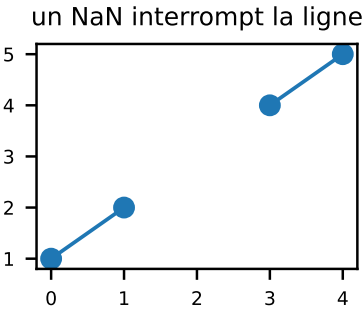


6.6 `plot`, `scatter` et les valeurs manquantes

À chaque intention sa fonction. `plot` relie les points dans l'ordre où on les donne : c'est une *courbe*, et elle suppose que cet ordre a un sens (le temps, en général). `scatter` pose au contraire des points indépendants : c'est un *nuage*, où l'ordre n'a aucune importance et où l'on cherche une corrélation.



`plt.plot` relie les points par des segments ; `plt.scatter` trace des points isolés.



un `NaN` *interrompt* la ligne : deux segments séparés, pas d'interpolation.

Sur la seconde figure, le `NaN` au milieu des ordonnées *coupe* la ligne en deux segments. matplotlib n'interpole jamais à notre place, et c'est heureux : le trou reste visible au lieu d'être maquillé par un segment inventé.

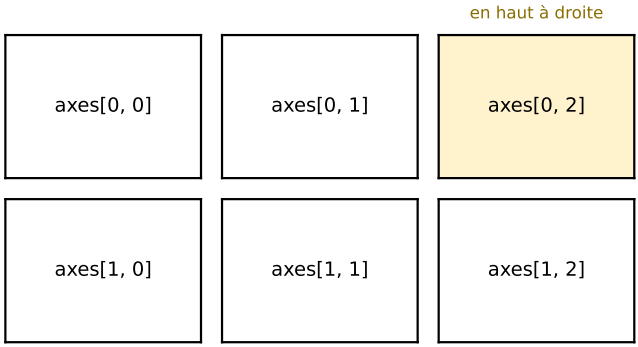
Quelques options reviennent constamment, et se combinent librement :

```
plt.plot(x, y, "r--", marker="o", lw=0.5, alpha=0.4)
plt.axis("equal")   # repère orthonormé : les cercles sont ronds
plt.loglog(x, y)    # deux axes logarithmiques
```

La chaîne `"r--"` condense couleur et style de trait, `lw` règle l'épaisseur et `alpha` la transparence (précieuse quand les points se superposent). `plt.axis("equal")` s'impose pour une trajectoire ou une figure géométrique : sans lui, un cercle s'affiche en ellipse. `plt.loglog` change la lecture même du graphique : une loi de puissance y devient une droite, dont la pente donne l'exposant. Enfin, `plt.xlim` / `plt.ylim` bornent les axes et `plt.xticks` en fixe les graduations.

6.7 Grilles de sous-figures

Comparer deux tracés, c'est d'abord les mettre côte à côte. Pour ce faire, `plt.subplots` sait produire d'un coup une grille entière de repères partageant la même figure.




```
fig, axes = plt.subplots(2, 3)   # axes : tableau numpy (2, 3)
axes[0, 2].plot(x, y)            # case en haut à droite
```

`axes` est un *tableau numpy* de forme `(2, 3)` : on y accède comme dans n'importe quel tableau, la ligne d'abord puis la colonne. La case en haut à droite est donc `axes[0, 2]`, et non `axes[2, 0]` !

Quand la grille n'a qu'une ligne ou qu'une colonne, il est plus lisible de *déballer* le tableau directement dans des noms parlants :

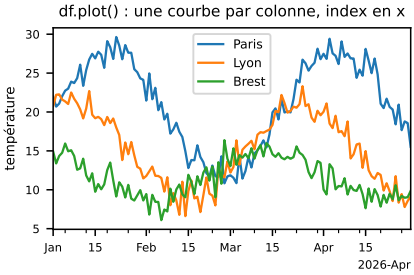
```
fig, (ax_haut, ax_bas) = plt.subplots(2, 1, figsize=(11, 8))
```

On termine toujours de la même façon : `tight_layout` ajuste les espacements pour que les étiquettes ne soient ni coupées ni superposées, puis on enregistre.

```
fig.tight_layout()
fig.savefig("serie.png")
```

6.8 Tracer depuis pandas : `df.plot`

Passer par matplotlib pour tracer une table que l'on a déjà en main serait fastidieux. pandas propose donc un raccourci qui fait le travail d'habillage tout seul.



`df.plot()` trace *une courbe par colonne numérique*, prend l'index en abscisse et construit la légende à partir des noms de colonnes. D'où l'intérêt d'un index de dates soigné : il devient l'axe du temps sans que l'on ait un mot de plus à écrire.

Le résultat reste du matplotlib ordinaire : on peut enchaîner avec `plt.title(...)` ou n'importe quel réglage vu plus haut.

```
df.plot()                  # une courbe par colonne numérique
s.plot(marker="o")         # une Series aussi
```

L'attribut `.plot` porte aussi des méthodes qui changent le type de tracé, sans jamais changer la logique d'appel :

```
df.plot.bar(); df.plot.barh()   # barres verticales/horizontales
croise.plot.bar(stacked=True)   # barres empilées
df.plot.scatter(x="Volume", y="Close")
```

Et l'on peut envoyer le tracé dans un repère déjà créé, ce qui réconcilie `df.plot` avec les grilles de sous-figures :

```
profil.plot.bar(ax=ax_bas)   # dans un repère existant
df.hist(); df.boxplot()      # histogrammes, boîtes à moustaches
```

PIÈGE

Pour un nuage de points depuis une table, la bonne écriture est `df.plot.scatter(x=..., y=...)`. `df.plot(x=, y=)` *relie* les points, et `df.scatter` n'existe pas !

Pour aller plus loin, sachez que `seaborn` (`import seaborn as sns`) offre des visualisations statistiques plus riches (`relplot`, `pairplot`, `displot`), avec la même logique d'appel `data=`, `x=`, `y=`, `hue=`.

6.9 Pour finir

Les devoirs ne demandent rien d'autre que ce qui précède : il s'agit de choisir la bonne brique au bon moment, et de vous demander à chaque étape *pourquoi* le résultat a la tête qu'il a. Et pour toute question, comme d'habitude, vous pouvez me contacter par email.