COURS PYTHON TSI2

SOMMAIRE

Chapitre 1

Recursion and sorting recursion

■ Objectifs

Les incontournables :

savoir

Résumé de cours

■ Récurrence et fonctions récursives

Nous savons que l'on peut définir des suites en ne donnant pas une formule explicite pour u_n mais une formule de récurrence (et le premier terme de la suite). Par exemple, même si l'on ne sait pas en trouver une formule explicite, il existe une unique suite vérifiant

$$u_0 = 1 \text{ et } \forall n \in \mathbb{N},\ u_{n+1} = u_n^2 - n.$$

Si l'on veut écrire une fonction qui calcule le terme de rang n de cette suite, on peut écrire une boucle, dans laquelle on commence par calculer le terme de rang 1, puis le terme de rang 2...
On dit que la méthode est **impérative**.

```
In [1]: def u(n):
   ...:     x = 1
   ...:     for k in range(1,n+1):
   ...:         # calcul de u(k)=u^2(k-1) -(k-1)
   ...:         x = x**2 -k+1
   ...:     return x
In [2]: u(7),u(12)
Out[2]: (10, 310822058031477121387888456118 65)
```

On voit vu la tête de `u(12)` que l'on monte vite. Ici, nous avons détaillé la succession de calculs à faire pour arriver à u_n. Mais on peut faire autrement.

```
In [4]: def uRec(n):
   ...:     if n == 0 :
   ...:         return 1
   ...:     else :
   ...:         return uRec(n-1)**2 -(n-1)
   ...:
In [5]: uRec(7),uRec(12)
Out[5]: (10, 310822058031477121387888456118 65)
```

Une telle fonction est dite **récursive**, i.e. que dans la définition de la fonction, la fonction elle-même apparaît. Cette fois-ci, nous ne détaillons aucun calcul mais nous allons développer le cheminement pour calculer $uRec(4)$. Pour calculer u_4, la fonction s'appelle elle-même pour calculer u_3. Elle se réappelle encore pour calculer u_2, puis idem avec u_1 et enfin avec u_0. Elle connaît la réponse pour u_0 c'est 1. Elle renvoie cette valeur ce qui permet de calculer u_1. Après ce calcul, elle renvoie u_1 et on peut enfin calculer u_4.

On peut comparer le temps d'execution de la fonction `u` et de la fonction `uRec`
Pour cela, on use de la fonction prédefinie `time` que vous connaissez déjà de `perf_counter`

```
In [7]: from time import perf_counter as pc
In [8]: for n in range(2,10) :
   ...:     t = pc()
   ...:     u(n)
   ...:     print(pc() - t)

2.6999999818144715e-06 4.500000045482011e-06
2.7000000955013093e-06 2.499999936844688e-06
2.900000026784255e-06 3.199999923708674e-06
3.6000000136482413e-06 4.3999999661537e-06

In [9]: for n in range(2,10) :
   ...:     t = pc()
   ...:     uRec(n)
   ...:     print(pc() - t)

2.6000000161729986e-06 3.6000000136482413e-06
4.900000021734741e-06 5.600000008598727e-06
5.399999963628943e-06 6.199999916134402e-06
6.20000002982124e-06 7.3999999585794285e-06
```

Rien de bien significatif.

```
In [11]: for n in range(20,30) :
   ...:     t = pc()
   ...:     u(n)
   ...:     print(pc() - t)

0.00013369999999213178 0.0003566000000319036
0.00101859999995213 0.0034633999999869047
0.009808200000065881 0.03159629999993285
0.09322210000004816 0.26550889999998617
0.7767928999999185 2.3187297999999146

In [12]: for n in range(20,30) :
   ...:     t = pc()
   ...:     uRec(n)
   ...:     print(pc() - t)
0.00013879999994514947 0.0003527000000076441
0.0010068000000273969 0.0030861999999842737
0.010145299999976487 0.028480800000011186
0.08652330000006714 0.26422969999998713
0.7755059999999503 2.32723390000001
```

Encore rien de significatif. En fait les méthodes impératives ou récursives peuvent être comparables dans certains cas et dans d'autres il vaut mieux l'une que l'autre. Cela dépendra de l'algorithme.

Mise en œuvre : de l'exercice 01 à l'exercice 06

■ fonctions définies sur les listes

Les fonctions récursives que nous avons vues jusqu'à présent étaient définies sur $\mathbb{N}$, ce qui permettait de justifier leur terminaison :
• on commençe par traiter le (ou les) cas de base, pour lesquels la fonction renvoie directement une valeur (en général $n = 0$ et $n = 1$, voire d'autres situations plus compliquées) ;
• dans les autres cas, la fonction se réappelle elle-même, avec un argument strictement inférieur à celui de l'appel précédent.

Puisque l'argument diminue strictement, on finira, lors des appels récursifs, par tomber sur un des cas de base, pour lesquels la fonction renverra directement une valeur, ce qui permettra, par remontées successives, de terminer le calcul.

Puisque l'argument diminue strictement, on finira, lors des appels récursifs, par tomber sur un des cas de base, pour lesquels la fonction renverra une valeur, ce qui permettra, par remontées successives, de terminer le calcul.

Une autre situation dans laquelle il est commode de définir des fonctions récursives est celle des listes :
• on commence par traiter le(s) cas de base (en général, la liste vide, éventuellement le cas d'une liste de longueur 1, voire d'autres cas) ;
• dans le cas général, la fonction se réappelle elle-même, avec pour argument une nouvelle liste, dont la longueur est strictement inférieure à celle de l'appel précédent.

Dans ce cas aussi, on est assuré de la terminaison de la fonction.

Exemple : Écrivons une fonction récursive qui calcule la somme des termes d'une liste (de nombres).
• si la liste est vide, la somme est nulle ; on renvoie alors la valeur 0;
• sinon, il suffit de calculer (récursivement) la somme des termes de la sous-liste obtenue en retirant le dernier élément, et d'ajouter ce dernier élément.
On utilise $l[-1] = l[len(l) - 1]$ pour définir le dernier élément d'une liste et $l[: -1]$ pour définir la liste emputée de son dernier élément. On tape alors `sommesTermesRec(l)`

```
In [13]: def sommeTermesRec(l):
    ...:     if len(l) == 0:
    ...:         return 0
    ...:     else:
    ...:         return l[-1] + sommeTermesRec(l[:-1])
In [14]: sommeTermesRec([1,4,-4,7])
Out[14]: 8
```

Remarque : Cette façon d'écrire la fonction est un peu maladroite. En effet, en Python, l'appel à `L[:-1]` crée une copie de la liste `l` (privée de son dernier terme). Il y a donc ici autant de copies créés que d'appels récursifs. Il est préférable de ne créer aucune copie, en écrivant une nouvelle fonction (récursive) qui calcule la somme entre deux indices `deb` et `fin` À chaque appel récursif, on remplace l'indice `fin` par son prédécesseur. Le cas terminal est celui d'une plage vide, c'est-à-dire celui où `fin < debut` On tape :

```
In [15]: def sommeEntreRec(l,deb,fin):
    ...:     if fin < deb:
    ...:         return 0
    ...:     else :
    ...:         return l[fin] + sommeEntreRec(l,deb,fin-1)
```

La terminaison est now assurée par le fait que, lors des appels récursifs, la quantité `fin-deb` décroît strictement. Elle finira donc par devenir strictement négative, ce qui est le cas terminal.
La somme totale des éléments de la liste s'obstient en tapant :

```
In [16]: def NewsommeTermesRec(l):
    ...:     sommeEntreRec(l,0,len(l)-1)
In [17]: NewsommeTermesRec([1,4,-4,7])
Out[17]: 8
```

Mise en œuvre : de l'exercice 07 à l'exercice 09

■ fonctions définies sur d'autres types

L'argument d'une fonction récursive peut être plus compliqué qu'un entier ou une liste. Nous avons vu déjà quelques cas : `f(a,b)` de l'exercice 06 ou `sommeEntreRec(l,deb,fin)`
Dans ces deux exemples, cependant seul un des paramètres variait lors des appels récursifs (`b` remplacé par `b-1` ou `fin` par `fin-1`). La terminaison était alors facilement établie.

Exemples : 1. Un premier exemple
Que dire de la fonction suivante :

```
In [26]: def s(a,b) :
    ...:     if b == 0:
    ...:         return a
    ...:     else :
    ...:         return s(a+1,b-1)
```

Cette fois ci, les deux arguments évoluent et `a` augmente ! Ce n'est pas grave car le cas terminal est `b=0` et le deuxième argument diminue strictement à chaque appel récursif. Si `b` est un entier positif, la fonction se terminera.

Si l'on suppose que `a` et `b` sont deux entiers positifs, que renvoie `s(a,b)` ?

```
In [28]: s(2,4), s(2024,2025)
Out[28]: (6, 4049)
```

2. Nombre de chemins de déplacement d'un pion

Un pion se déplace sur un échiquier dont les cases sont numérotées.

Le départ est situé au point de coordonnées $(0,0)$. À chaque déplacement, le pion se déplace d'une et d'une seule case, soit vers la droite, soit vers le haut. On cherche à savoir combien de chemins distincts permettent de rejoindre le point de coordonnées (n,p) avec n et p deux entiers positifs connus.

• dans le cas général, les chemins qui aboutissent en (n,p) se divisent en deux catégories : ceux qui proviennent du point $(n,p-1)$ et ceux qui proviennent du point $(n-1,p)$. Ce raisonnement s'applique si $n,p \geqslant 1$.
On en déduit que $f(n,p) = f(n,p-1) + f(n-1,p)$.

• dans le cas où $n = 0$, il y a un seul type de chemin qui se termine en $(0,p)$: c'est de provenir de $(0,p-1)$ (du moins si $p \geqslant 1$). On en déduit $f(0,p) = f(0,p-1)$;

• de même, si $p = 0$, on a : $f(n,0) = f(n-1,0)$;

• enfin, il y a un unique chemin qui rermine en $(0,0)$: celui qui consiste à effectuer 0 déplacements.

```
In [29]: def nbChemins(n,p):
    ...:     if n == 0 and p == 0:
    ...:         return 1
    ...:     elif n == 0:
    ...:         return nbChemins(0,p-1)
    ...:     elif p == 0 :
    ...:         return nbChemins(n-1,0)
    ...:     else :
    ...:         return nbChemins(n-1,p)+nbChemins(n,p-1)
```

À la main calculer `nbChemins(1,1)` et `nbChemins(2,2)`

```
In [34]: nbChemins(4,4), nbChemins(4,7), nbChemins(8,10)
Out[34]: (70, 330, 43758)

In [35]: nbChemins(12,14)
Out[35]: 9657700
```

Pour `nbChemins(12,14)` ce ne fut pas instantané.

Comment justifier la terminaison de la fonction ? Cette fois-ci, au cours des appels récursifs, les deux paramètres évoluent, et aucun ne décroît strictement à chaque étape (parfois, le paramètre p est remplacé par lui-même ; parfois c'est n). Il faut cette fois ci s'intéresser à la quantité $s = n+p$. Celle-ci décroît strictement (d'une unité) à chaque appel récursif. Comme le cas $n = p = 0$ est l'unique cas de base, on finira par tomber sur ce cas lors des appels récursifs, et la fonction se terminera.

Remarque : on peut simplifier un peu la définition (et le nombre d'appels récursifs) en remarquant que si $n = 0$ alors il existe un unique chemin menant de $(0,0)$ à $(0,p)$, qui consiste à se déplacer vers le haut à chaque étape. On a donc $f(0,p) = 1$ (et de même $f(n,0) = 1$).

On obtient :

```
In [36]: def New_nbChemins(n,p):
    ...:     if n == 0 or p == 0 :
    ...:         return 1
    ...:     else :
    ...:         return New_nbChemins(n-1,p)+New_nbChemins(n,p-1)
    ...:

In [37]: New_nbChemins(4,4), New_nbChemins(4,7), New_nbChemins
   (8,10)
Out[37]: (70, 330, 43758)
```

Exercice d'application : Montrer que dans l'exemple précédent, le nombre de chemins est

$$f(n,p) = \binom{n+p}{n} = \binom{n+p}{p}.$$

3. Exponentiation rapide

Le problème est de calculer x^n avec $n \in \mathbb{N}$ le plus efficacement possible.

La solution qui vient à l'esprit est de calculer $x \times x \times ... \times x$ ce qui se fait au prix de $n-1$ multiplications.

Mais dans le cas où $n = 8$, on peut écrire : $x^8 = (x^2)^4 = (x^2)^{2\times 2} = ((x^2)^2)^2$.

Ce calcul ne nécessite que trois élévations au carré au lieu de sept multiplications dans la méthode classique.

De façon générale, si n est un nombre pair, on a utilisé la formule : $x^{2p} = (x^2)^p$.

Le problème est donc ramené de l'exposant $n = 2p$ à celui de l'exposant p qui est deux fois plus petit.

Dans le cas où n est impair, alors posons $n = 2p+1$ et : $x^{2p+1} = x(x^2)^p$. On tape :

```
In [38]: def puissRapide(x,n):
    ...:     if n == 0 :
    ...:         return 1
    ...:     elif n % 2 == 0:
    ...:         return puissRapide(x*x,n//2)
    ...:     else:
    ...:         return x * puissRapide(x*x, n//2)

In [39]: puissRapide(3,2) , puissRapide(5,15)
Out[39]: 9   30517578125
```

On peut donner une forme avec deux cas terminaux : $n = 0$ et $n = 1$.

On tape :

```
In [41]: def New_puissRapide(x,n):
    ...:     if n == 0 :
    ...:         return 1
    ...:     elif n == 1 :
    ...:         return x
    ...:     elif n % 2 == 0:
    ...:         return New_puissRapide(x*x,n//2)
    ...:     else:
    ...:         return x * New_puissRapide(x*x, n//2)

In [42]:  New_puissRapide(3,2),  New_puissRapide(5,15)
Out[42]: (9, 30517578125)
```

La terminaison est assurée par le fait que, lors de chaque appel récursif, l'exposant décrôit strictement, donc finit par tomber sur 0, pour lequel la fonction renvoie une valeur.

Cet algorithme est très intéressant car il permet de passer d'une complexité linéaire (en l'exposant) pour le calcul des puissances à une complexité logarithmique. En effet, à chaque appel récursif, l'exposant est au moins divisé par deux, et deux multiplications au plus sont effectuées.
À l'issue de p appels récursifs, l'argument est donc au plus égal à $n/2^p$.
On tombe sur un cas terminal dès que ce nombre est strictement inférieur à 2.
On cherche donc le plus petit entier p tel que :

$$\frac{n}{2^p} < 2 \text{ et } \frac{n}{2^{p-1}} \geqslant 2.$$

Cela donne : $p \leqslant \log_2(n)$.

4. Calcul de PGCD : algorithme d'Euclide

Rappelons que si `a` et `b` sont deux entiers positifs ou nuls (l'un au moins étant non nul), le PGCD de `a` et de `b` est le plus grand diviseur commun à `a` et `b`. La remarque intéressante pour le calculer est que les diviseurs communs à `a` et `b` sont les mêmes que les diviseurs communs à `a-b` et `b`

Donc : `PGCD(a,b) = PGCD(a-b,b)`

Par une récurrence immédiate, on a :

$$\forall q \in \mathbb{N} \text{ avec } \texttt{a-qb} \geqslant 0, \texttt{PGCD(a,b) = PGCD(a-qb,b)}.$$

Le plus grand possible est le quotient de la division euclidienne de `a` par `b` Alors `a-bq` est le reste `r` de cette division euclidienne et : `PGCD(a,b) = PGCD(r,b).`

Dans le cas où `b=0`, les diviseurs communs à `a` et `b=0` sont les diviseurs de `a` Le plus grand est donc `a`

Enfin, par convention, $PGCD(0,0) = 0$ et donc la formule `PGCD(a,0)=a` est encore vraie pour `a=0`

```
In [43]: def PGCD(a,b):
    ...:     if b == 0 :
    ...:         return a
    ...:     else :
    ...:         return PGCD(a%b,b)
```

Vérifions la terminaison de la fonction. Le cas terminal est `b=0` Or, dans notre codage, `b` n'est jamais modifié, donc s'il est initialement non nul, on ne tombera jamais sur le cas terminal.
Donc la fonction boucle sans fin...
Pour corriger cela, on remarque que : `PGCD (a,b) = PGCD(b,a)` et on tape alors :

```
In [44]: def GutPGCD(a,b):
    ...:     if b == 0 :
    ...:         return a
    ...:     else :
    ...:         return GutPGCD(b,a%b)
```

Cette fois la fonction se termine car `b` est remplacé par le reste de la division de `a` par `b` qui est strictement inférieur à `b` On finit par tomber sur le cas terminal.

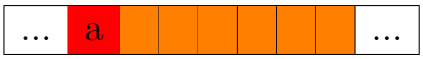
Exercice Faire tourner `PGCD(28,35)` à la main. Avec l'ordi, on trouverait :

```
In [45]: GutPGCD(28,35)
Out[45]: 7
```

■ Un premier tri récursif : Algorithme de tri rapide

Pour trier la sous-liste $l[i..j]$, l'algorithme de tri rapide consiste à :

1. choisir un élément pivot a (cellule rouge) dans la sous-liste $l[i..j]$ (plage rouge + orange). On choisit par exemple le premier `l[i]`



2. permuter les éléments de la sous-liste (symbolysée plus bas par les couleurs yellow, red et blue) de sorte que, après permutation, tous les éléments inférieurs à a (plage yellow) soient situés avant a, tous les éléments supérieurs à a soient situés après a (plage blue).

3. relancer le point 1 sur les sous-listes $l[i..k-1]$ et $l[k+1..j]$ si elles sont de longueur strictement supérieure à 1.

Commençons par écrire une fonction *partition* qui partitionne une liste entre les indices *deb* et fin, suivant l'élément a d'indice *deb*. Nous aurons besoin que cette fonction renvoie la nouvelle position de a, pour que l'on sache quelles sont les deux sous-listes à trier à l'étape suivante.
Pour écrire cette fonction, examinons les éléments l_i, pour $i \in [\![deb,\ fin]\!]$, en maintenant l'invariant de boucle suivant :

- $deb \leqslant m \leqslant i \leqslant fin$
- $\forall k \in [\![deb,\ m]\!],\ l_k \leqslant a$
- $\forall k \in [\![m+1,\ i]\!],\ l_k > a$.

Et passons maintenant à `partition(l,deb,fin)` qui est la fonction Python correspondante.

```
In [46]: def partition(l,deb,fin):
    ...:     a=l[deb] ; m =deb
    ...:     for i in range(deb+1,fin+1):
    ...:         if l[i] <= a:
    ...:             m += 1
    ...:             l[i],l[m] = l[m],l[i]
    ...:     l[deb],l[m] = l[m], l[deb]
    ...:     return m
```

Mise en œuvre : exercice 10

Remarque : On peut taper `return (m,l)` pour voir comment `l` est transformé. C'est ce que l'on fera à l'exercice 10.

Écrivons maintenant la fonction récursive qui trie le tableau entre les indices `deb` et `fin`

```
In [56]: def triEntre(l,deb,fin):
    ...:     if deb < fin:
    ...:         m = partition(l,deb,fin)
    ...:         triEntre(l,deb,m-1)
    ...:         triEntre(l,m+1,fin)
```

Il reste enfin la fonction principale, qui appelle la fonction récursive. On tape :

```
In [57]: def triRapide(l):
    ...:     triEntre(l,0,len(l)-1)
    ...:     return l
    ...:
In [58]: triRapide([3,-1,2,3,5,4,8,0,4,2,5,-5])
Out[58]: [-5, -1, 0, 2, 2, 3, 3, 4, 4, 5, 5, 8]
```

Complexité du tri rapide :

On peut montrer que le pire des cas est celui où, à chaque étape de partition, une des deux sous-listes est vide (et l'autre de taille $n-1$). La complexité maximale vérifie donc la relation de récurrence :

$$C_{max}(n) = C_{max}(n-1) + n - 1.$$

D'où l'on tire (comme $C_{max}(1) = 0$) que :

$$C_{max}(n) = \sum_{k=2}^{n}(k-1) = \frac{n(n-1)}{2} \sim \frac{n^2}{2}.$$

A l'inverse, le cas le plus favorable est celui, où, à chaque étape, les deux sous-listes résultant de la partition ont la même taille (à une unité près). Calculons la complexité dans ce cas.

Pour ce faire, nous nous limiterons au cas où la liste est de longueur $n = 2^p - 1$.

Notons alors

$$x_p = C_{min}(2^p - 1).$$

Si $p \geqslant 2$, la partition de la liste se fait au prix de $2^p - 2$ comparaisons et produit de deux sous-listes de tailles égales à $2^{p-1} - 1$.

On a donc la formule :

$$x_p = 2x_{p-1} + 2^p - 2 \Rightarrow x_p - 2 = 2(x_{p-1} - 2) + 2^p.$$

On divise tout par 2^p et l'on a, en posant $y_p = \dfrac{x_p - 2}{2^p}$,

$$y_p = y_{p-1} + 1.$$

C'est une suite arithmétique classique et comme $x_1 = 0$, $y_1 = -1$.

On a : $y_p = p - 1 + y_1 = p - 2$ puis $x_p = (p - 2)2^p + 2$.

Comme ici $n = 2^p - 1$, $2^p \sim n$ et $p \sim \log_2(n)$. On obtient donc :

$$C_{min}(n) \sim n \log_2(n).$$

Nous admettrons que ce résultat reste vrai même lorsque n est quelconque. On résume :

Proposition. Les complexités dans le pire des cas et dans le meilleur des cas du tri rapide sont respectivement :

$$C_{min}(n) \in O(n \log n) \text{ et } C_{max}(n) \in O(n^2)$$

Ce tri n'est pas très intéressant.

Notons que ce tri n'est pas stable : lors de l'étape de partition, un élément de même clé que le pivot dans la plage $l[deb..fin]$ sera permuté avec le pivot.

■ Un second tri récursif : Algorithme de tri fusion

Le principe de ce tri est de découper la liste en deux, de trier (récursivement) chaque partie, et de fusionner ces deux sous-listes pour obtenir une liste triée.

Commençons par écrire une fonction de fusion de deux listes l_1 et l_2, déjà triées.

Plus précisément, puisque notre fonction doit trier la liste l (et non renvoyer une autre liste, dont les éléments sont ceux de l mais triès.

Pour cela, nous créons une liste auxiliaire aux, dans laquelle nous réalisons le tri, avant de recopier le résultat dans la liste l.

On tape alors en Python la fonction `fusion(l,deb,mil,fin)` suivante.

```
In [59]: def fusion(l,deb,mil,fin):
    ...:     aux = []
    ...:     i,j = deb, mil+1
    ...:     while i <= mil and j <= fin :
    ...:         if l[i] < l[j]:
    ...:             aux.append(l[i]) ; i += 1
    ...:         else :
    ...:             aux.append(l[j]) ; j += 1
    ...:     while i <= mil :
    ...:         aux.append(l[i]) ; i += 1
    ...:     while j <= fin :
    ...:         aux.append(l[j]) ; j += 1
    ...:     for k in range(deb,fin + 1) :
    ...:         l[k] = aux[k-deb]

# On ecrit la fonction qui trie l entre deb et fin

In [60]: def triFusionEntre(l,deb,fin):
    ...:     if deb < fin :
    ...:         m = (deb+fin)//2
    ...:         triFusionEntre(l,deb,m)
    ...:         triFusionEntre(l,m+1,fin)
    ...:         fusion(l,deb,m,fin)

# Puis on tape la fonction principale

In [61]: def triFusion(l):
    ...:     triFusionEntre(l,0,len(l)-1)
    ...:     return l

In [62]: triFusion([-2,2,3,7,9,10,1,8,1,4])
Out[62]: [-2, 1, 1, 2, 3, 4, 7, 8, 9, 10]
```

Mise en oeuvre : Exercice 11

Proposition. La complexité (dans tous les cas) du tri fusion vérifie : $C(n) \in O(nlogn)$

Notons enfin que ce tri est stable.

Chapitre 2

Dictionary

■ Objectifs

Les incontournables :

savoir

Résumé de cours

■ Introduction

Les dictionnaires permettent de mémoriser des données. Ces données peuvent être des résultats intermédiaires qui sont produits par des calculs et qui doivent être réutilisés ensuite.
Les dictionnaires sont des types construits. Rappelons les types principaux de Python.
Le type `tuple` : C'est une liste ordonnée d'objets délimitée par des parenthèses. Ces objets sont indéxés par des entiers.

```
In [1]: a = (1,2,3) ; a = a + (5,6)
In [2]: a
Out[2]: (1, 2, 3, 5, 6)
```

On peut concaténer des `tuples`.

Le type `list` : C'est une liste ordonnée d'objets délimitée par des crochets. Ces objets sont indéxés par des entiers.

```
In [1]: l=[3,1,4]; l.append(5)
In [2]: l
Out[2]: [3, 1, 4, 5]
```

Le type `dict` : C'est une liste non ordonnée d'objets délimitée par des accolades, appelée dictionnaire. Ces objets sont accessibles par des clés (au lieu d'un indice). Une clé peut être n'importe quel objet, un `tuple` mais pas une liste ou un autre dictionnaire. On construit un dictionnaire en associant des successions de couples, chaque couple est constitué d'un premier élément, appelé clé, `keys` en Python et un second appelé valeur, `values` en Python.

```
In [1]: d={0:1,'login':'lppr','pass':'lppr','
   proc':64}
In [2]: d['login']
Out[2]: 'lppr'
In [3]: for cle in d.keys(): print(cle)
Out[3]:  0
         login
         pass
         proc
```

```
In [1]: for val in d.values(): print(val)
Out[1]:  1
         lppr
         lppr
         64
In [2]: for cle,val in d.items(): print(cle,val)
Out[2]: 0 1
        login lppr
        pass lppr
       proc 64
In [3]: valeur = d.pop('proc') ; valeur
Out[3]: 64
```

■ Implémentation

Une liste en Python est implémentée de manière à ce que certaines opérations soient très efficaces, c'est-à-dire avec une complexité en temps constante. Ces opérations sont par exemple : obtenir la longueur d'une liste, accéder à un élément et modifier un élément. La suppression et l'insertion d'un élément en fin de liste ont aussi un coût que l'on peut considérer comme constant.

Les dictionnaires ne sont pas des séquences ce qui siginifie que les éléments n'ont pas une place, repérée par un indice entier, qui permet d'ordonner l'ensemble pour le parcourir et par la même occasion accéder à un élément particulier en utilisant son indice. Dans un dictionnaire, un élément n'a ni prédécesseur ni successeur. L'objectif est de disposer d'opérations similaires à celles énumérées ci-dessus pour les listes, avec la même efficacité.

Il est important de noter que les éléments d'un dictionnaire n'ont pas à être ordonnés. Donc pour le parcours et l'affichage des clés ou des couples (clé, valeur), il n'y a pas d'ordre prévisible.

Il existe différentes manières d'implémenter un dictionnaire.

Exemple 1

Par exemple, si le dictionnaire **a pour clés des entiers naturels**, on peut utiliser un tableau. Si un indice est égal à une clé, on écrit la valeur correspondante, sinon on écrit une valeur par défaut. Dans ce cas, la complexité en espace est en $O(N)$, où N est la plus grande clé.

Rappelons le principe d'un tableau qui peut être utilisé par exemple pour stocker les adresses en mémoire des éléments d'une liste en Python.
On accède à un élément par son indice. Les adresses des éléments sont enregistrées sur des mots de taille fixe (8 ou 4 octets) de manière séquentielle. Ainsi, si l'on connait l'adresse du début, on peut obtenir l'adresse d'un élément quelconque en calculant par une opération simple sur l'indice la place où se trouve cette adresse. On prévoit un tableau assez grand pour pouvoir ajouter quelques éléments.

Illustration par la figure 1

Exemple 2

Plaçons nous dans le cas général où **les clés ne sont pas toutes des entiers naturels**. Une implémentation classique d'un dictionnaire utilise une **table de hachage**. On dispose d'une **fonction de hachage**, c'est-à-dire une fonction qui à une clé associe un entier appelé **valeur de hachage**

de la clé. Cet entier est utilisé pour calculer l'indice d'un tableau où est placée la clé.
On peut ainsi stocker suivant l'indice calculé pour chaque clé, l'adresse de la clé puis celle de la valeur et stocker la clé et la valeur aux adresses indiquées. Cela revient à utiliser une liste ordonnée suivant les indices calculés pour les clés avec successivement des clés et des valeurs et des places vides. Pour trouver un élément, on calcule l'indice, on trouve l'adresse de la clé et on vérifie. Ceci est exécuté en temps constant. On peut aussi stocker la valeur de hachage.

On doit donc disposer d'une fonction f avec $f(e) = p$ qui calcule la place p d'un élément e. Deux éléments distincts devant se trouver à deux places distinctes, la fonction f doit être bijective. Ce point a une première conséquence : si une clé `x` est mutable, sa place va changer à chaque mutation. Donc on ne peut utiliser comme clé que des objets non mutables, plus précisément non récursivement mutables.

Illustration par la figure 2

Exemple 3

Une autre possibilité est de stocker suivant l'ordre de création, la valeur de hachage calculée pour la clé, l'adresse de la clé puis celle de la valeur et stocker la clé et la valeur aux adresses indiquées. Cela revient à utiliser une liste ordonnée suivant l'ordre de création avec successivement une valeur de hachage, une adresse de clé et une adresse de valeur et ainsi de suite, sans place inoccupée. On complète alors avec un tableau d'octets où chaque octet, à la place dont l'indice est calculé pour une clé, donne la position dans la liste des trois éléments valeur de hachage, clé, valeur. Les octets ne correspondant à aucune valeur de hachage valent -1 pour indiquer une place vide et -2 pour un élément supprimé.
Donnons un exemple où seules les adresses des clés sont représentées.

Considérons le dictionnaire

```
In [1]: {97: "a", 101:"e", 114:"r", 111: " o"}
```

et un tableau `T` de huit octets supposé être la capacité du dictionnaire. ce tableau `T` va donner les positions des clés dans le dictionnaire.
Soit h la fonction de hachage et une clé `c` qui est la $k^{\text{ème}}$ insérée alors si `p` est la position calculée avec `p = h(c) modulo 8` alors `T[p] = k`.
Si l'on ne considère que les clés, le tableau des données contient dans l'ordre $97, 101, 114, 111, 0, 0, 0, 0$.
L'élément 97 a pour indice 0, l'élément 101 a pour indice 1, l'élément 114 a pour indice 2 et l'élément 111 a pour indice 3.
On choisit pour h la fonction identité (ce qui est logique, on verra plus loin).
Alors `h(97) modulo 8` est `97 modulo 8` soit 1. Alors $T[1] = 0$.
Puis `h(101) modulo 8` est `101 modulo 8` soit 5. Alors $T[5] = 1$.
Puis `h(114) modulo 8` est `114 modulo 8` soit 2. Alors $T[2] = 2$.
Enfin, `h(111) modulo 8` est `111 modulo 8` soit 7. Alors $T[7] = 3$.
Le tableau T contient dans l'ordre :

$$-1, 0, 2, -1, -1, 1, -1, 3$$

En effet, $T[0]$ par exemple n'est pas affecté et est une place vide et cet octet ne correspond à aucune donnée. On peut mettre 255 à la place de -1 (car $2^8 = 256$).
Pour accéder à la clé 101, comme sa position `p = h(101) modulo 8` vaut 5, on regarde dans T à la position 5 et on y lit la valeur 1 qui est la position de 101 dans le dictionnaire.

Si l'on supprime par exemple la clé 97, on remplace dans T à l'indice 1 la valeur 0 par -2 (ou 254).

Fonctions de hachage

Comment obtenir une bonne fonction de hachage ? Cette fonction, notée h doit permettre un calcul rapide de l'image d'une clé. Les indices calculés en général par h(clé) modulo n, où n est la taille du tableau T doivent être tous distincts.
En pratique, il est très souvent impossible d'avoir une correspondance bijective entre les clés et les indices. Lorsque le même indice est obtenu pour deux clés, on parle de **collision**.
Pour résumer, on procède en deux étapes :

Étape 1. Une fonction h de hachage associe un entier à chaque clé. Elle code donc les clés. En Python, la fonction est `hash`

```
In [1]: hash(55)
Out[1]: 55
In [2]: hash('pass') ; hash("login")
Out[2]: -8560737331547147328
    -4537507453712144586
In [3]: hash(2**61-1) ; hash(-1)
Out[3]: 0   -2
```

• Si i est un entier différent de -1 et si $-2^{61}+1 < i < 2^{61}-1$ alors `hash(i)` vaut i.
• Si x est un flottant égal à p/q alors $hash(x)$ vaut $(int(p*M/q))\,\%\,M$ avec $M = 2^{61}-1$.

```
In [1]: hash(0.1)
Out[1]: 230584300921369408
In [2]: (int(1*(2**61-1)/10)) % (2**61 -1)
Out[2]: 230584300921369408
```

• Si la clé est une chaîne de caractères la valeur de hachage est calculée avec une part de hasard à chaque nouvelle session et a pour valeur un entier qui s'écrit sur 64 bits.

Étape 2 Il faut ensuite une fonction qui réduise chaque entier pour obtenir un entier appartenant à $[\![0, n-1]\!]$, où n est la taille du tableau (la capacité du dictionnaire).
En Python, on utilise `hash(c) % n`

■ Gestion des collisions

En Python deux objets distincts de même valeur ont la même valeur de hachage :

```
In [1]: a=(3,2)
In [2]: b=(3,2)
In [3]: a is b
Out[3]: False
In [4]: hash(a) == hash(b)
Out[4]: True
```

Mais deux objets dont les valeurs ne vérifient pas le critère d'égalité peuvent aussi avoir la même valeur de hachage :

```
In [1]: hash(0)
Out[1]: 0
In [2]: hash(2**61 -1)
Out[2]: 0
```

Il faut donc trouver des solutions à ce problème de collision.
Pour gérer une collision, on a plusieurs méthodes. En cas de collision, on peut calculer une autre place ou prendre la première place libre qui suit. On commence par prendre un dictionnaire plus long. Une méthode de redimensionnement est utilisée par Python qui prévoit des dictionnaires dont pas plus des deux tiers de la capacité est utilisée. Si l'on atteint les deux tiers de la capacité et qu'un élément doit être ajouté, la capacité du dictionnaire est multipliée par 2.
Par exemple :
de 1 à 5 éléments, la capacité est 8 car $(3/2) \times 5 = 7.5$ et $(3/2) \times 6 = 9$.
Jusqu'à 10 éléments, la capacité est 16 car $(3/2) \times 10 = 15$ et $(3/2) \times 11 = 16.5$
etc.
La capacité n est ainsi une puissance de 2.
Si la taille est $n = 2^p$, alors $h \,\%\, n$ et $h \,\&\, (n-1)$ ont la même valeur.

```
In [1]: 556 % 16
Out[2]: 12
In [2]: 556 & 15
Out[2]: 12
```

En Python, avec un dictionnaire de capacité 8, en cas de collision à une place i, la nouvelle place est calculée avec la formule :

$$i = (5 * i + (h // (2 * *5)) + 1) \,\%\, 8$$

où h a pour valeur `hash(c)` si `c` est la clé.
Si les clés sont des entiers i strictement inférieurs à 32, l'entier h est i et `h // (2**5)` vaut 0 et la formule se réduit à : $i = (5 * i + 1) \,\%\, 8$.
Ainsi si $i = 4$, les 7 valeurs successives sont :

```
In [1]: ind=4
In [2]: for k in range(1,8) :
            ind=(5*ind+1)%8; print(ind)
Out[2]: 5
        2
        3
        0
        1
        6
        7
```

On parcourt ainsi les huit places en sachant que quatre au moins sont disponibles.

Exemple 4

Illustrons un problème de collision. Soit le dictionnaire :

```
In [1]: d = { "i": 19, "n": 14, "f": 6 , " o ": 15}
```

Nous utilisons une liste de longueur 8 pour stocker un dictionnaire de cinq éléments maximum.

```
In [1]: def dic(couples):
          liste = 8 * [None]
          for cle, val in couples:
              liste[hash(cle)%8]=(cle,val)
          return liste
```

```
In [1]: dic((("i",19),("n",14),("f",6), ("o",15)))
    [None, None, ('n', 14), None, None, None, ('o', 15), None]
```

On voit que le couple `("f",6)` par exemple a disparu car `("o",15)` a été le dernier affecté en liste[6] car `hash("f")%8` et `hash("0")%8` sont les mêmes.
On modifie la fonction `dic` pour gérer les collisions.

```
In [1]: def dic(couples):
            liste = 8* [None]
            for cle, val in couples :
                i = hash(cle) % 8
                while liste[i] is not None :
                    i = (5*i +1) % 8
                liste[i] = (cle,val)
            return liste

In[2]:  dic((("i",19),("n",14),("f",6), ("o",15)))
Out[2]: [None,None,('n',14),None,('o', 15),None,('i', 19),('f',
    6)]
```

La place d'indice 7 est occupée après l'insertion de `("f",6)` et la nouvelle place calculée pour insérer `("o",15)` est

```
In [1]: (5*7+1)%8
Out[1]: 4
```

C'est bien l'indice 4.

■ Manipulation

Construction

Pour l'instant, on a crée un dictionnaire en plaçant entre des accolades des couples (clé,valeur) séparés par des virgules, chaque clé et sa valeur associée étant séparées par deux points. On peut construire un dictionnaire par **compréhension** :

```
In [1]: d  = {x : x**2 for x in range(1,5)}
In [2]: d
Out[2]: {1: 1, 2: 4, 3: 9, 4: 16}
```

Ou aussi par **insertion**. On crée un dictionnaire vide puis on insère les éléments par une boucle.

```
In [1]: d = {}
In [2]: for x in range(1,5):
            d[x] = x**2
In [3]: d
Out[3]: {1: 1, 2: 4, 3: 9, 4: 16}
In [4]: d[3]
Out[4]: 9
```

Au passage `d[x]` renvoie la valeur dont `x` est la clé.
De même que les éléments d'une liste peuvent être des listes, les éléments d'un dictionnaire peuvent être des dictionnaires :

```
In [1]: pays = {"France":{"capitale": "Paris",
                          "population": 68014000,
                          "superficie": 643800.0},
                "Portugal" : {"capitale": "Lisbonne",
                              "population": 10302674,
                              "superficie":92300.0},
                "Italie" :{"capitale": "Rome",
                           "population" : 60359546,
                           "superficie" : 301336.0}}
In [2]: pays["France"]["population"]
Out[2]: 68014000
In [3]: pays["France"]
Out[3]: {'capitale': 'Paris', 'population': 68014000, 'superficie
    ': 643800.0}
```

Utilisation

Accès aux éléments

Deux méthodes donnent accès aux clés ou aux valeurs, ce sont les méthodes `keys` et `values`. Et la méthode `items` donne accès à l'ensemble des couples. Le mieux c'est le faire fonctionner.

```
In [1]: d = {"true":"vrai", "false" : "faux", "and" : "et", "or "
    : "ou"}
In [2]: d.keys()
Out[2]: dict_keys(['true', 'false', 'and', 'or '])
In [3]: d.values()
Out[3]: dict_values(['vrai', 'faux', 'et', 'ou'])
In [4]: d.items()
Out[4]: dict_items([('true', 'vrai'), ('false', 'faux'), ('and',
   'et'), ('or ', 'ou')])
```

L'accès à une valeur s'obtient comme avec les listes. La différence est qu'il faut préciser la clé à la place de l'indice.

```
In [1]: d["true"]
Out[1]: 'vrai'
```

Appartenance

le mot clé permet de tester l'appartenance d'une clé à un dictionnaire par l'appartenance d'une valeur.

```
In [1]: "vrai" in d, "and" in d
Out[1]: (False, True)
```

Boucles

On peut itérer avec une boucle `for` sur un dictionnaire, la variable d'itération est une clé.

```
In [1]: cles=[]
In [2]: for obj in d:
            cles.append(obj)

In [3]: cles
Out[3]: ['true', 'false', 'and', 'or ']
```

Il est possible avec une boucle `for` d'itérer sur les clés, sur les valeurs ou sur les couples (clé, valeur) à l'aide des objets `d.keys()`, `d.values()` et `d.items()`

```
In [1]: val = []
In [2]: for newobj in d.values():
            val.append(newobj)
In [3]: val
Out[3]: ['vrai', 'faux', 'et', 'ou']
```

Propriété

On ne peut pas modifier la taille d'un dictionnaire durant une itération sans un message d'erreur.

```
In [1]: dd = {0:0, 1:-5, 2:4}
In [2]: for c in dd:
            dd[c+1] = dd[c] + 1

Traceback (most recent call last):
File "<ipython-input-22-04e6fc6c3560>", line 1, in <module>
    for c in dd:
RuntimeError: dictionary changed size during iteration
```

Pourtant, l'opération se fait quand-même.

```
In [1]: dd
Out[1]: {0: 0, 1: 1, 2: 2}
```

Nombre d'éléments

La fonction `len` renvoie la longueur d'un dictionnaire c'est-à-dire le nombre de couples (clé, valeur).

```
In [1]: len(d),len(dd)
Out[1]: (4, 3)
```

Suppression

Pour supprimer un élément, nous utilisons la fonction `del`

Copie

On use de `d.copy()`. La vigilance s'impose, comme avec les listes, puisque les dictionnaires sont des objets mutables.

```
In [1]: d2 = d.copy()
In [2]: d2
Out[2]: {'true': 'vrai','false': 'faux','and': 'et', 'or ': 'ou'}
In [3]: del d2["false"]
In [4]: d
Out[4]: {'true': 'vrai','false': 'faux', 'and': 'et','or ': 'ou'}
In [5]: d2 = d
In [6]: del d2["false"]
In [7]: d
Out[7]: {'true': 'vrai', 'and': 'et', 'or ': 'ou'}
```

Donc `d` a perdu `"false"` alors qu'on l'a enlevé qu'à `d2`.
Autre exemple où le même pb survient.

```
In [1]: d3 = {"true": ["vrai", "vraie"]}
In [2]: d4 = d3.copy()
In [3]: d4
Out[3]: {'true': ['vrai', 'vraie']}
In [4]: d4["true"][1] = "vrais"
In [5]: d3
Out[5]: {'true': ['vrai', 'vrais']}
In [6]: d4
Out[6]: {'true': ['vrai', 'vrais']}
```

■ Applications dans le langage Python

Les dictionnaires sont des objets au coeur du fonctionnement du langage Python. Lorsqu'on exécute un programme, plusieurs dictionnaires sont mobilisés en arrière plan, même si le programme ne contient aucune définition explicite de dictionnaire.

Espace de noms

Les noms appartenant à un espace de noms quelconque sont les clés d'un dictionnaire. Au démarrage de l'interpréteur, le module `__builtins__` est chargé. Ce module contient tous les objets que nous pouvons utiliser directement dans le programme et le dictionnaire `__builtins__.__dict__` lié à ce module contient tous les identifiants liés à ces objets comme `help,len,True,max` etc. Donnons un exemple (la barre `__` devant et après `builtins` et `dict` est constituée de deux fois le tiret sous la touche 8.

```
In [1]:  __builtins__.__dict__['min'](5,2,3)
Out[1]:  2
```

Le contenu des modules que nous importons est représenté par un dictionnaire.
Si nous importons `math` avec `import math` alors nous obtenons le contenu, comme les fonctions disponibles par `dir(math)`. Ceci revient à demander la liste des clés du dictionnaire `math.__dict__`
Si l'on tape ces deux commandes, la dernière écrit les mêmes fonctions que la première mais avec la syntaxe d'un dictionnaire.

```
In [1]: dir(math)
Out[1]: ['__doc__',
         '__loader__',
         '__name__',
         '__package__',
         '__spec__',
         'acos',
         'acosh',
         'asin',
         'asinh',
          ...
         'tau',
         'trunc']
```

```
In [1]: math.__dict__
Out[1]:  {'__name__': 'math',
   '__doc__': 'This module provides access to the mathematical
      functions\ndefined by the C standard.',
 '__package__': '',
 '__loader__': _frozen_importlib.BuiltinImporter,
 '__spec__': ModuleSpec(name='math', loader=<class '
    _frozen_importlib.BuiltinImporter'>, origin='built-in'),
 'acos': <function math.acos(x, /)>,
 'acosh': <function math.acosh(x, /)>,
 'asin': <function math.asin(x, /)>,
 ...
  'e': 2.718281828459045,
 'tau': 6.283185307179586,
 'inf': inf,
 'nan': nan}
```

Tous les identifiants des variables et des fonctions que nous définissons sont stockés dans un dictionnaire que nous obtenons avec `globals()`. Tapez le et vous verrez apparaître tout notre passé du jour.
Les paramètres d'une fonction et les variables définies dans le corps d'une fonction sont stockés dans un dictionnaire créé à l'exécution de la fonction et supprimé à la fin de l'exécution. On peut observer son contenu en ajoutant dans le corps de la fonction `print(locals()`

```
In [1]: def f(x,y):
          z = x+y
          print(locals())
In [2]: f(3,4)
Out[2]: {'x': 3, 'y': 4, 'z': 7}
```

Notion de portée
Définissons une fonction `abs` dans un programme en cours. Le nom `abs` appartient alors à l'espace de nom local de ce programme. Donc si l'on utilise `abs` dans ce programme, c'est l'espace local qui est examiné en premier et donc cette fonction est utilisée en premier. Mais la fonction `abs` connue (la valeur absolue) du module `__builtins__` n'a pas été modifiée.

```
In [1]:def abs(x):
          return x
In [2]: abs(-5)
Out[2]: -5
In [3]:__builtins__.abs(-5)
Out[3]:  5
```

Notion d'affectation
Écrire par exemple $x = 300$ et `globals()["x"]`$= 300$ est équivalent. On ajoute au dictionnaire la clé `x` associée à la valeur 300. Autrement dit, on ajoute au dictionnaire le couple (identité ou adresse de l'objet de type `str` et de valeur $'x'$, identité de l'objet de type `int` et de valeur `300`). Si l'on écrit ensuite $x = 500$, on remplace l'identité de l'objet de type `int` et de valeur 300 par celui de valeur 500. Le premier objet n'est pas modifié mais `x` est lié à un nouvel objet.

Exécution d'une fonction
Tapons :

```
In [1]: def f(f):
          return 2*f
In [2]: f(5)
Out[2]: 10
```

La clé `f` (le nom de la fonction) est ajoutée au dictionnaire `globals()` et liée à l'objet de type `function` et de valeur le code de la fonction.
Lors de l'exécution, un dictionnaire local lié à la fonction est créé. Ce dictionnaire contient pendant l'exécution le nom `f` (celui du paramètre), associé à la valeur 5.

```
In [1]: def f(f):
          return 2*f, locals()
In [2]: f(5)
Out[2]: (10, {'f': 5})
```

Pour finir, les dictionnaires sont aussi utilisés dans des opérations de comptage et pour l'implémentation des graphes. Ils sont utilisés aussi pour la modélisation des jeux ou le traitement de données en tables etc.

Chapitre 3

Graphs

■ Objectifs

Les incontournables :

savoir

Résumé de cours

VOIR FEUILLES PDF

diagbox

Chapitre 4

Database

■ Objectifs

Les incontournables :

savoir

Résumé de cours

■ Organisation des données. Modèle relationnel

I. Introduction. Type de données

La quantité de données utilisables dans notre vie est gigantesque et se mesure en gigaoctet $= 10^6$ octets, téraoctet $= 10^9$ octets ou pétaoctet $= 10^{12}$ octets. L'utilisation de bases de données relationnelles permet le stockage de données ainsi que l'organisation des mises à jour et des consultations par un grand nombre d'utilisateurs.

Parlons un peu du type de données. Les formats peuvent être différents : des nombres, des textes, des images, des vidéos. Pour les stocker, il faut choisir comment les coder afin de les enregistrer sur un support numérique, donc définir un type de données par exemple des types numériques, des types textes, des types dates.
Chaque système de gestion des bases de données, ou SGBD, présente quelques différences sur les types disponibles mais en propose suffisamment afin de pouvoir utiliser dans chaque cas le plus économique en besoin au niveau de la mémoire et limiter ainsi la taille de la base de données.
Les **types numériques** se décomposent en **types entiers** ou non et dans ce cas, on se limite au **type flottant**. Une représentation des nombres en machine a été étudié en TSI1 avec des mots de taille fixe $(8, 16, 32, 64$ bits). Les SGBD peuvent utiliser des représentations différentes pour certains types de nombres afin de pouvoir les stocker de manière exacte et effectuer des calculs exacts. Par exemple, pour un nombre décimal, on peut coder en binaire chaque chiffre de son écriture. Avec des mots de même taille pour chaque chiffre, nous aurons besoin de 4 bits par mots, en effet le plus grand nombre est 9 et s'écrit 1001 en binaire et par exemple $68,3$ est codé 0110 1000 0011.
Pour les textes également, on se restreint en général à un **type chaîne de caractères**.
Concernant les dates, il existe des types précis mais on se limite à des chaînes de caractères du type `aaaa-mm-jj`, ce qui est suffisant pour pouvoir effectuer des comparaisons avec l'ordre lexicographique. Une autre façon est d'utiliser des nombres entiers représentant le nombre de secondes écoulées depuis l'origine (souvent 1970-01-01) et la date souhaitée.

II. Schéma relationnel. Vocabulaire

Des notions mathématiques comme le langage ensembliste, les relations, et la logique sont à la base du modèle.

Considérons par exemple dans un lycée des données concernant les él'èves, les professeurs et les administratifs. On reprèsente l'ensemble des données, comme le nom, le prénom, la date de naissance, le numéro de téléphone de chaque personne dans un tableau à deux dimensions appelé **relation ou table**.

Une colonne du tableau est appelée **attribut ou champ** par exemple la colonne nommée `nom`. L'ensemble des colonnes caractérisent la relation. On peut noter l'analogie avec une loi conjointe d'un couple (X, Y) de variables aléatoires discrètes finies. Chaque personne correspondrait à une valeur de X et chaque attribut correspondrait à une valeur de Y.

Chaque ligne de la table ou relation est un **enregistrement**, on dit aussi une **instance** ou `tuple` ou encore `n-uplet` (à condition qu'il y ait n colonnes). Chacun de ces tuples est unique et caractérise donc une personne donnée. On part du principe que c'est impossible d'avoir le même nom, le même prénom, la même date de naissance et le même numéro de phone pour deux personnes différentes (à moins qu'il y ait eu piratage des données mais on suppose être dans un monde honnête).

Chaque valeur dans une colonne a un type unique pour cette colonne avec des valeurs limites (par exemple 8 caractères max etc.). On dit que les valeurs permises dans une colonne appartiennent à un **domaine**.
L'obligation d'appartenance d'une valeur à un domaine s'appelle une **contrainte d'intégrité**. Plus précisément, il s'agit ici d'une **intégrité de domaine**.
Le **produit cartésien** des domaines $D_1, D_2, ..., D_n$ noté $D_1 \times D_2 \times ... \times D_n$ est l'ensemble des `n-uplets` $(v_1, v_2, ..., v_n)$ tels que, pour tout i, $v_i \in D_i$.
Une relation est donc un sous-ensemble du produit cartésien de domaines.
Exemple : supposons les domaines $D_1 = \{a, b, c\}$ et $D_2 = \{x, y\}$.

D_1 \ D_2	x	y
a	(a, x)	(a, y)
b	(b, x)	(b, y)
c	(c, x)	(c, y)

Une relation est par exemple $R = \{(a, x), (b, x), (c, y)\}$ ou encore

C_1	C_2
a	x
b	x
c	y

, où la colonne C_1 est constituée de valeurs du domaine D_1 et la colonne C_2 est constituée de valeurs du domaine D_2.

III. Notion de clé dans un schéma relationnel

• Dans toute relation, un attribut (c'est-à-dire une colonne) ou un groupe d'attributs, doit permettre d'identifier de manière unique un enregistrement (c'est-à-dire une ligne).
On l'appelle une **clé candidate**
Par exemple, si l'on reprend l'exemple plus haut, la colonne C_2 ne peut pas être une clé car il y a deux fois x et par contre la colonne C_1 est bien une clé candidate.

• S'il y a plusieurs clés candidates, on en privilégie une, nommée la **clé primaire**. Choisir une clé primaire est une contrainte d'intégrité imposée à la relation. Dans l'exemple précédent, on n'a pas le choix, la colonne C_1 est la seule clé primaire possible.

• Certaines relations peuvent contenir un attribut dont les valeurs permises sont uniquement celles prises par la clé primaire d'une autre relation. Cet attribut est alors appelé **clé étrangère**. Une clé étrangère est une **contrainte d'intégrité référentielle** (elle fait référence à un attribut d'une autre table). Elle relie deux tables de manière cohérente.

Les clés primaires et les clés étrangères permettent de spécifier des contraintes dans les bases de données et d'effectuer des opérations de jointures entre les tables, opérations présentées dans le chapitre suivant.

La présence d'une clé primaire est capitale pour assurer que les lignes sont toutes distinctes. Une contrainte d'unicité est donc imposée : chaque valeur prise par l'attribut ou le groupe d'attributs, sur lequel est posé une contrainte de clé primaire est unique.

IV. Exemples détaillés

Reprenons l'exemple d'un établissement scolaire et considérons une base de données qui sert à stocker des informations utiles pour l'établissement scolaire. Plusieurs relations sont possibles.

Pour représenter le concept d'élève, nous utilisons une relation **Élève**. Les attributs (ou les colonnes) de cette relation par exemple sont **INE** (numéro d'identification), **Nom**, **Prénom**, **Âge**, **Sexe**, **Classe**. Chaque élève est représenté par un enregistrement (une ligne).

Pour représenter le concept de classe, nous utilisons une relation **Classe** avec pour attributs **Nom**, **Effectif**, **Salle**.
Il y a bien entendu des liens entre l'ensemble des élèves et l'ensemble des classes.
Un élève donné fait parti d'une classe particulière.
Ce lien fonctionne dans les deux sens : une classe contient plusieurs élèves.

Les données peuvent être modélisées d'une autre manière.
On ajoute une relation **Personne** pour représenter les personnes appartenant à la communauté scolaire. Si cette relation dispose d'attributs comme le nom, le prénom, etc., on peut supprimer ces atrributs de la relation **Élève** et ne garder que le numéro INE et la classe. Pour des relations comme **Professeurs**, **Administratif**, **Service**, on peut remplacer l'attribut **INE** par un attribut ayant pour valeur le numéro de sécurité sociale par exemple.

Un schéma de relation se présente sous cette forme :

Relation(attribut 1, attribut 2, ..., attribut N)
Clé primaire : **attribut 1**
Clé étrangère : **attribut N** en référence à la clé primaire d'une autre relation.

Par exemple :
Élève (INE, Nom, Prénom, Âge, Sexe, Classe)
Les attributs **INE** et **Âge** sont de type entier, les autres de type chaîne de caractères.
Clé primaire : **INE**
Clé étrangère : **Classe** en référence à la clé primaire d'une autre relation.

Ce schéma peut aussi se noter :
Élève (INE, Nom, Prénom, Àge, Sexe, #Classe)
L'attribut sur lequel est posé une contrainte de clé primaire est souligné et l'attribut sur lequel est posé une contrainte de clé étrangère est précédé du caractère #

On peut représenter le schéma ainsi :

INE
Nom
Prénom
Âge
Sexe
#Classe

Conception d'une relation

Si à une valeur d'un attribut A correspond une et une seule valeur d'un attribut B, on dit que l'attribut B est en **dépendance fonctionnelle** de l'attribut A.
Par exemple, à un identifiant d'élève correspond un unique nom d'élève.

De manière générale, tous les attributs ne contiennent qu'une seule information et sont en dépendance fonctionnelle de la clé primaire.
Si une clé primaire est constituée de plusieurs attributs, les autres attributs sont en dépendance fonctionnelle de l'intégrabilité de la clé primaire et pas seulement d'une partie de celle-ci. De plus, tous les attributs sont en d¡'ependance fonctionnelle uniquement de la clé primaire.

Organisation des données
Les données peuvent être représentées dans un tableau comme ci-après.

INE	NOM	Prénom	Âge	Sexe	Classe
1	Parker	Bonnie			2
2	Barrow	Clyde			4
3	Laurel	Stanley			1
...					
50	Hardy	Oliver			
...					

Remarque : Ce tableau ne doit pas être interprété comme celui obtenu dans un tableur.
Un tableur permet d'effectuer diverses actions sur ces données : des calculs, des schémas, des graphiques.
Ici l'attribut **INE** est un identifiant numérique qui permet d'identifier chaque ligne de manière unique. Cet attribut **INE** peut donc être défini comme clé primaire.

Pour la deuxième relation nommée **Classe**, on a le tableau :

Nom	Effectif	Salle
PSI1	45	$B307$
PSI2	42	C206
PC	38	$B311$
TSI2	29	C204
PT	41	B309
...	...	...

Pour distinguer les attributs **Nom** des relations **Élève** et **Classe**, nous pouvons utiliser la notation pointée comme **Élève.Nom** et **Classe.Nom**
L'attribut **Nom** de la relation **Classe** est une clé primaire. Nous souhaitons poser une contrainte de clé étrantère sur l'attribut **Classe** de la relation **Élève**.
Or, pour effectuer des recherches et pour procéder à d'éventuelles modifications, il est plus efficace d'utiliser un nombre entier plutôt qu'une chaîne de caractères. On ajoute donc une colonne supplémentaire, un nouvel attribut nommé **Id**. Il s'agit d'une clé artificielle ou clé de substitution (surrogate key) qui permet d'identifier une ligne quelconque.

Id	Nom	Effectif	Salle
1	PSI1	45	$B307$
2	PSI2	42	C206
3	PC	38	$B311$
4	TSI2	29	C204
5	PT	41	B309
...	...	...	

Les valeurs de la clé étrangère **Classe** de la relation **Élève** sont alors des valeurs de la clé primaire **Id** de la relation **Classe**.

Avec cette représentation, les modifications sont beaucoup plus simple à gérer que que si toutes les données étaient dans un tableau unique. Ainsi si par exemple, les élèves de la classe PSI1 changent de salle, il n'y a qu'une seule valeur à modifier dans le tableau **Classe**.

La notion de contrainte est capitale. Supposons que dans la table **Classe** un enregistrement, ou une ligne, contient la valeur **7** pour l'identifiant **Id** et que cette valeur **7** est celle d'un ou plusieurs enregistrements pour l'attribut **Classe** dans la table **Élève**.
Si par erreur, la suppression dans la table **Classe** de l'enregistrement avec l'identifiant **7** est demandée, cela va provoquer une erreur car cela conduirait à une violation de la contrainte de référence ou contrainte de clé étrangère, de **Élève.Classe** vers **Classe.Id**
En pratique, on peut contourner ce problème en imposant une suppression en cascade. Dans ce cas toutes les lignes concernées de la table **Élève** sont supprimées. On peut aussi imposer que les lignes concernées ne soient pas supprimées mais qu'une valeur par défaut remplace la valeur **7**.

V. Bases de données relationnelles et SGBD

Un SGBD est un système de gestion de base de données. L'informaticien Charles Bachman est le premier concepteur d'un SGBD moderne.

Pour accéder à une base de données, on utilise donc un SGBD. Ce peut être un composant logiciel, comme SQLite qui est une bibliothéque avec une interface directement intégrable dans un programme. D'autres SGBD comme MySQL et PostgreSGL fonctionnent avec un serveur.

Il y a deux types de systèmes de gestion de base de données :

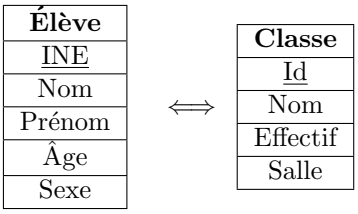
- des systèmes libres comme MySQL, PostgreSQL, SQLite ;
- des systèmes propriétaires comme Oracle Database ou encore SQL Server de Microsoft ou encore Access de la suite Microsoft Office

Fonction d'un SGBD

Le système doit assurer la persistance des données. Cela consiste à garder en mémoire des versions antérieures lorsque des modifications sont effectuées.
Le système doit gérer des accès concurrents et la sécurité.
Il gère les droits et privilèges d'un utilisateur.
L'un des attendus principaux est un accès aux données efficace. Cette efficacité est conditionnée par la manière de stocker les différentes clés primaires. L'utilisation d'une clé de substitution (surrogate key) apporte une simplification pour la recherche. On peut utiliser plutôt qu'une structure linéaire (pour la recherche des données) une structure d'arbre, en particulier un arbre binaire (Btree en anglais). C'est la structure utilisée par les SGBD. Enfin, les SGBD utilisent aussi des tables de hachage pour classifier les données.

VI. Entités-Associations

Reprenons l'exemple plus haut. Un élément qui joue un rôle dans une relation par exemple **Élèves** est appelé **entité**. C'est ici un élève particulier qui possède des attributs servant à le décrire. Des liens ou relations existent entre **Élève** et **Classe**. La relation entre deux entités de ces schémas s'appelle une **association**. Le type d'association est décrit par un titre comme **Appartenir** puisqu'un élève appartient à une classe ou **Contenir** puisqu'une classe contient un élève.



Le lien entre une entité et une association est caractérisé par un couple qui indique le nombre minimum de fois et le nombre maximum de fois qu'une entité peut participer à l'association.

Le couple est **1-1** si chaque entité participe exactement une fois à la relation.
Le couple est **1-n** si chaque entité participe au moins une fois à la relation.
Le couple est **0-n** si certaines entités ne participent pas une fois à la relation.

Pour notre exemple, on a le couple **1-1** entre **Élève** et **Appartenir** signifie que chaque élève appartient à exactement une classe. On a le couple **1-n** entre **Appartenir** et **Classe** qui signifie que chaque classe contient au moins un élève.
On peut présenter le modèle entité-association en parlant d'associations **1-***, **1-1**, ***-*** entre deux entités. Par exemple, nous avons une association **1-*** entre **Élève** et **Classe**.

■ Langage SQL (Structured Query Language)

I. Introduction

Le langage SQL (Structured Query Language = langage de requêtes structurées) permet aux utilisateurs de communiquer avec une base de données. C'est un langage déclaratif qui décrit ce que l'on souhaite faire avec une requête : une demande adressée à un SGBD.
Plus précisément, une instruction en SQL traduit une interrogation, une mise à jour, une insertion, une suppressio, une création etc.

Les instructions en SQL ont une syntaxe assez simple, plus libre qu'en Python.

Déjà contrairement à Python, SQL ne fait pas la différence entre majuscule et minuscule. Ceci dit, la convention veut que les mots prédéfinis du langage soient en majuscule (par exemple SELECT qui est le premier de tous) et les noms de tables ou de colonnes (de la table) soient en minuscule (à part éventuellement la première lettre). Les espaces, les retours à la ligne et l'indentation n'ont aucune valeur syntaxique mais participent à la clarté de la requête (donc à utiliser dans une copie en particulier le jour du concours).

L'écriture des valeurs doivent respecter des règles (liées aux types ou domaines) : un flottant s'écrit comme en Python avec un point pour la virgule, une chaîne de caractères s'écrit comme en Python avec des guillemets ou des apostrophes et par exemple la chaîne de caractères `'Ta␣gueule!'` est différent de `'ta␣gueule!'`.
Enfin, on termine une requête ou instruction par un point-virgule qui sert de séparateur entre deux instructions. Ce n'est pas obligatoire s'il n'y a qu'une instruction. Là aussi cela rappelle Python où dans une ligne, on peut mettre plusieurs instructions indépendantes separées par un point-virgule.

II. Les requêtes d'interrogation dans une table

Nous allons nous intéresser à cinq cas : **projection** (c'est-à-dire récupérer et sortir de la table des colonnes ou attributs), **champ calculé** (c'est-à-dire faire des calculs dans des colonnes donnés), **renommage de colonne**, **tri** (c'est-à-dire ordonner les lignes de la table dans un sens imposé), **selection de lignes**.

1. La projection

Reprenons un des exemples plus haut avec la table **Classe**

Id	Nom	Effectif	Salle
1	PSI1	45	$B307$
2	PSI2	42	C206
3	PC	38	$B311$
4	TSI2	29	C204
5	PT	41	B309
...	...	...	

Tapons **SELECT Nom FROM Classe**, on obtient :

Nom
PSI1
PSI2
PC
TSI2
PT
...

On peut demander le contenu de plusieurs colonnes en séparant les noms par des virgules :

Tapons **SELECT Nom, Salle FROM Classe**, on obtient :

Nom	Salle
PSI1	B307
PSI2	C206
PC	B311
TSI2	C204
PT	B309
...	...

Pour obtenir toutes les colonnes, on utilise le symbole *****

Retenir les syntaxes suivantes qui permettent de récupérer une ou plusieurs colonnes. :

SELECT Attribut1 FROM Table récupère la colonne **Attribut1** de **Table**

SELECT Attribut1 , Attribut2 FROM Table récupère les colonnes **Attribut1** et **Attribut2** de **Table**

SELECT * FROM Table récupère toutes les colonnes de **Table**

SELECT DISTINCT Attribut1 FROM Table récupère la colonne **Attribut1** de **Table** en supprimant les doublons.

Remarque La commande **SELECT ... FROM ...** s'appelle une projection car on peut faire l'analogie avec une projection vectorielle, l'espace vectoriel E est la table et l'image de la projection est constitué des colonnes récupérées qu'on peut voir comme un sous-espace vectoriel de E.

2. Le champ calculé

Un champ est un synonyme d'attribut ou de colonne ici. Le mieux est un exemple.

Relation : **Notes (Id, Maths, Physique, Informatique, SI)** avec **Id** pour clé primaire.
Les autres champs sont de type flottant et représentent des notes.

Supposons que la commande **SELECT Maths FROM Notes** renvoie :

Maths
14
12.5
8

Alors la commande **SELECT Maths * 3 FROM Notes** renvoie :

Maths*3
42
37.5
24

On peut aussi faire des opérations entre plusieurs champs (à condition qu'ils soient de type flottant)

La requête **SELECT Id, (Maths + Physique) / 2 FROM Notes** renvoie deux colonnes de **Notes**, la première est la colonne des identifiants des élèves et la seconde fait la moyenne ligne par ligne des notes de maths et de physique de chaque élève défini par son identifiant.

La requête **SELECT Id, 2*3 FROM Notes ;** renvoie deux colonnes, la première est la colonne **Id** de **Notes** et la seconde colonne (qui conserve le même nombre de lignes que la colonne **Id** est constituée de la valeur 6 sur chaque ligne.

Remarque : Ces opérations sur les champs utilisent les opérateurs classiques +, -, * et /
On verra plus loin qu'on peut les prolonger par les fonctions d'agrégation.

3. Renommage des colonnes

Dans l'exemple de la relation **Notes**, on a crée un nouveau champ appelé **Maths*3**, on peut décider de lui donner un autre nom par exemple, on décide de l'appeler **Total_Maths** et de même la colonne **(Maths + Physique)/2**, on veut la nommer **Moyenne**.

On tape alors les requêtes SQL :

SELECT Maths * 3 AS Total_Maths FROM Notes ;
SELECT (Maths + Physique) / 2 AS Moyenne FROM Notes ;

On a donc introduit une nouvelle fonction prédéfinie **AS** qui ressemble à `as` de Python.

4. Le tri

L'opération **SELECT** renvoie des colonnes dont les lignes respectent l'ordre initial. Ainsi une table est un ensemble d'enregistrements qui ne sont pas ordonné à moins que l'on impose un ordre. C'est l'objet de la suite. Le tri est une opération qui permet de classer les enregistrements selon un ou plusieurs critères. On utilise la commande prédéfinie **ORDER BY** qui a deux arguments le premier est **ASC** pour un tri dans l'ordre croissant et le second est **DESC** pour un tri dans l'ordre décroissant. Par défaut l'ordre est croissant et donc **ASC** n'est pas obligatoire (maintenant on peut le mettre pour la clarté). Enfin, l'ordre pris en compte est celui des lignes d'une ou de plusieurs colonnes données. Encore une fois, c'est beaucoup plus clair avec notre exemple que nous vous rappelons.

Relation : **Notes (Id, Maths, Physique, Informatique, SI)** avec **Id** pour clé primaire.
Les autres champs sont de type flottant et représentent des notes.

Nous désirons obtenir les identifiants des élèves classés suivant l'ordre décroissant des notes de maths. On tapera :

SELECT Id FROM Notes ORDER BY Maths DESC

On remarquera l'emplacement des différentes commandes, c'est une chronologie de syntaxe à respecter.

Nous désirons obtenir les identifiants des élèves classés suivant l'ordre croissant des notes de maths. On tapera :

SELECT Id FROM Notes ORDER BY Maths ;

Ici par défaut, c'est bien **ASC**

Plus subtil : nous désirons obtenir les identifiants des élèves classés suivant l'ordre croissant des notes de maths et s'il y a égalité entre deux notes de maths, on classera selon l'ordre croissant des notes de physique. On tapera :

SELECT Id FROM Notes ORDER BY Maths, Physique ;

C'est ce que l'on appelle l'ordre lexicographique (qui permet de créer un ordre dans $\mathbb{C}$ au passage mais c'est un autre sujet). On peut pousser ce process jusqu'au bout.

SELECT Id FROM Notes ORDER BY Maths, Physique, Informatique, SI, Id ;

5. La selection de lignes de la table

Avec la fonction **SELECT**, on obtient toute une colonne.

Supposons que l'on ne désire que certaines lignes de cette colonne.

α) Supposons d'abord que l'on veuille la partie d'une colonne qui va d'une ligne donnée à une autre ligne donnée. Ce sont les clauses **LIMIT** et **OFFSET** qui permettent de préciser les numéros des lignes voulues. On place ces fonctions à la fin de la requête SQL. Attention comme en Python on commence l'indice de la ligne par **0** et ainsi la commande **OFFSET 0** signifie à partir de la première ligne et par exemple la commande **LIMIT 5 OFFSET 8** signifie que l'on prend 5 lignes à partir de la neuvième ligne. Tapons la requête SQL :

SELECT Id, Maths FROM Notes LIMIT 2 OFFSET 1 ;

Elle renvoie la deuxième et troisième ligne des colonnes **Id** et **Maths**

On peut demander des lignes triées. Par exemple :

SELECT Maths FROM Notes ORDER BY Physique LIMIT 2 OFFSET 1 ;

Cette requête renvoie la deuxième et troisième ligne de la colonne **Maths** dont les lignes ont été préalablement triées selon les notes de physique.
On comprend que **ORDER BY** doit être logiquement avant **LIMIT OFFSET**

β) Supposons que l'on veuille non pas un certain nombre de lignes consécutives mais choisies selon un critère précis, on utilise alors la clause **WHERE** Donnons un exemple.
Considérons la relation **table** suivante.

x	y	z
-1	-1	-2
-1	0	-1
-1	1	0
0	-1	1
0	0	0
0	1	1
1	-1	-1
1	0	0
1	1	2

Tapons **SELECT x , y FROM Table WHERE z $>$ 0** On obtient :

x	y
0	-1
0	1
1	1

Il faut retenir que **WHERE** est suivi d'une expression booléenne qui est formée de :

de noms de colonnes ;

d'opérateurs mathématiques $+$, $-$, $*$, $/$;

d'opérations de comparaison $<$, $<=$, $>=$, $<>$;

d'opérateurs logiques **NOT**, **AND**, **OR**

Les opérateurs **AND** et **OR** traduisent une intersection et une réunion respectivement.
Par exemple supposons le schéma relationnel **Table** constitué de trois colonnes **col1**, **col2** et **col3** toutes de type flottant. Il s'agit de renvoyer cette table en ne gardant que les lignes telles que sur chaque ligne de **Table**, on ait la contrainte :

2*col1 + col2 <100 et (col 3 <> 1000 ou col3 = col1)

On doit donc taper :

SELECT col1, col2, col 3 FROM Table WHERE (2*col1 + col2 < 100) AND (col3<>1000 OR col3 = col1) ;

Remarque : attention, l'égalité est = et non == comme en Python et la négation de l'égalité est <> et non ! = comme en Python.

Autre exemple, prenons la relation **Élève (Id, Nom, Prénom, Adresse, Ville, Tel)** avec la clé primaire **Id** et écrivons une requête pour avoir les noms et prénoms des élèves qui habitent à Nice. On tapera :

SELECT Nom, Prénom FROM Élève WHERE Ville = "Nice" ;

Écrivons une requête pour avoir les noms et prénoms des élèves qui habitent à Nice ou à Cannes.

On tapera :

SELECT Nom, Prénom FROM Élève WHERE Ville = "Nice" OR Ville = "Cannes" ;

Écrivons une requête pour avoir le nom et le numéro de téléphone des élèves dont la première lettre du nom est comprise entre 'A' et 'M' bornes comprises.

On tapera :

SELECT Nom, Tel FROM Élève WHERE Nom >= "A" AND Nom < "N" ;

On admettra que l'ordre lexicographique est valable pour les chaînes de caractères.

γ) On peut faire des opérations logiques appliquées à des morceaux de tables : **UNION** pour l'union, **INTERSECT** pour l'intersection et **EXCEPT** pour la différence. Les morceaux de tables doivent être compatibles (même nombre et mêmes types d'attributs).

Exemple : Soit la relation **Élève (Id, Nom, Prénom, Adresse, Ville, Tel, #Numprof)** qui donne la liste des élèves ayant cours une heure précise.

La clé étrangère **Numprof** permet de savoir quel est le professeur de l'élève à cette ligne. On veut récupérer le nom et le prénom des élèves qui ont cours avec le professeur ayant pour **Numprof** la valeur **7** et dont l'identifiant **Id** est inférieur à **30**. On tapera :

SELECT Nom, Prénom FROM Élève WHERE Numprof = 7
INTERSECT
SELECT Nom , Prénom FROM Élève WHERE Id < 30 ;

Comme les opérations portent sur des ensembles, les doublons sont automatiquement éliminés.

III. Fonctions d'agrégation

Nous nous bornons aux cinq principales : **COUNT, SUM, AVG, MIN, MAX**

Ce sont des fonctions qui s'appliquent à des colonnes d'un schéma relationnel et qui se placent juste après **SELECT** dans la requête SQL.
Elles permettent de faire des calculs statistiques basiques.

Prenons pour tout le paragraphe le schéma suivant :

Météo (Id, Lieu, Année, Mois, Tmin, Tmax, Précipitations)

1. La fonction COUNT

Cette fonction sert à compter des enregistrements, donc les colonnes ne sont pas nécessairement de type flottant ou int pour l'utiliser.

Par exemple, écrivons la requête SQL pour obtenir le nombre de mois en 2021 pour lesquels les précipitations sont supérieures à 20 mm à Antibes.

Ce nombre de mois doit être renommé **Nombre_de_mois** dans la requête.

SELECT COUNT(Mois) AS Nombre_de_mois FROM Météo WHERE Lieu = "Antibes" AND Année = 2021 AND Precipitations > 20 ;

Remarque : On peut remplacer **COUNT(Mois)** par **COUNT(*)** car il s'agit simplement de compter le nombre de lignes qui vérifient la condition **WHERE**

2. La fonction SUM

Cette fonction s'applique à une colonne de valeurs numériques et fait la somme des lignes sélectionnées de cette colonne.

Par exemple, écrivons la requête SQL pour obtenir la quantité de pluies tombée en 2021 à Antibes.
Cette quantité de pluie doit être renommé **Précipations_annuelles** dans la requête.

SELECT SUM(Précipitations) AS Précipations_annuelles FROM Météo WHERE Lieu = "Antibes" AND Année = 2021 ;

3. La fonction AVG

Cette fonction sert à calculer la moyenne (en anglais average) des valeurs d'un champ numérique.

Par exemple, écrivons la requête SQL pour obtenir la moyenne des températures maximales en 2021 à Antibes.
Cette moyenne doit être renommée **Moyenne_Tmax** dans la requête.

SELECT AVG(Tmax) AS Moyenne_Tmax FROM Météo WHERE Lieu = "Antibes" AND Année = 2021

4. Les fonctions MIN et MAX

Elles permettent d'obtenir la valeur minimale et la valeur maximale d'un champ numérique.

Par exemple, écrivons la requête SQL pour obtenir la valeur maximale des précipitations en 2021 à Antibes.
Cette valeur maximale doit être renommé **Max_précipitations** dans la requête.

SELECT MAX(Précipitations) AS Max_précipitations FROM Météo WHERE Lieu = "Antibes" AND Année = 2021 ;

IV. Jointure

La jointure permet de mettre en relation des tables, par l'intermédiaire des liens qui existent en particulier entre la clé primaire de l'une et la clé étrangère de l'autre.
La jointure est une opération de selection car elle permet de ne retenir par exemple que les enregistrements pour lesquels la valeur de la clé primaire d'une table est égal à la valeur de la clé étrangère d'une autre table.

Au niveau de la syntaxe, l'opérateur **JOIN** exprime la jointure entre deux tables et la clause **ON** permet de préciser le critère de jointure. Par exemple :

table1 JOIN table2 ON table1.attribut_a = table2.attribut_b

Attention, les tables concernées peuvent contenir des colonnes dont les noms sont identiques. On évite de se mélanger les pinceaux en utilisant la syntaxe **table.colonne** qu'on conseille d'utiliser systématiquement pour la clarté visuelle de la requête.

Exemple : Utilisons les deux tables suivantes.

Relation 1 : **Professeur (Id, Nom, Prénom, Tel, Salle)**

Relation 2 : **Élève (Id, Nom, Prénom, Adresse, Ville, Tel, #Numprof)**

On constate que **Numprof** est une clé étrangère de la table **Élève** en référence avec la clé primaire **Id** de la table **Professeur**

La requête est d'obtenir le nom des élèves qui ont cours avec Madame Hopper (connue sous le nom de "Hopper" dans la colonne **Nom** de **Professeur** et la salle où a lieu ce cours.
Comme plusieurs colonnes ont la même dénomination dans les deux tables, on va utiliser aussi la syntaxe **table.colonne**

On tape la requête SQL :

SELECT Élève.Nom , Professeur.Salle FROM Élève JOIN Professeur ON Élève.Numprof = Professeur.Id WHERE Professeur.Nom = "Hopper" ;

Cet exemple est assez complet.
Notez que l'on selectionne ici une colonne dans chaque table avec **SELECT** puis on signale que l'on a **JOIN** les deux tables et que l'on assimile les colonnes **Numprof** d'une table et **Id** de l'autre avec **ON** et enfin **WHERE** permet de ne garder que les lignes concernant Madame Hopper.

V. Renommage de tables

Le renommage permet de raccourcir et de simplifier l'écriture d'une requête. On renomme les tables et on utilise le nouveau nom (l'alias) devant les champs dans la requête. On use du mot **AS** (comme pour le renommage de colonnes). Il est facultatif mais pour la clarté de l'écriture autant l'utiliser.
Par exemple si l'on veut que **Table** devienne son alias **T**, on tape **Table AS T** ou **Table T**
On peut créer l'alias au moment de **JOIN**. Par exemple :

Table1 AS t1 JOIN Table2 As t2 ON t1.attribut_a = t2.attribut_b

Reprenons l'exemple précédent des aventures de Madame Hopper.

On décide de renommer **e** la table **Élève** et de renommer **p** la table **Professeur** et on veut obtenir le nom des élèves qui ont cours avec Madame Hopper et la salle où a lieu ce cours. Et tout ça dans la même requête.

SELECT e.Nom , p.Salle FROM Élève AS e JOIN Professeur AS p ON e.Numprof = p.Id WHERE p.Nom = "Hopper" ;

Remarque : On remarque que le choix des tables arrivant en premier dans l'ordre d'exécution, les alias sont crées au niveau de **JOIN** et s'utilisent donc au moment de **SELECT** qui arrive ensuite dans l'exécution. Attention, l'ordre d'écriture donne l'impression que l'on utilise les alias avant de les définir. C'est normal. C'est simplement que l'ordre d'écriture n'est pas l'ordre d'exécution.

VI. Groupement et filtrage

Les fonctions d'agrégation peuvent être utilisées sur un groupe de données.
On utilise deux clauses **GROUP BY** et **HAVING**

1. La clause GROUP BY

Le mieux c'est un exemple.

Considérons le schéma relationnel : **Frais (Id, nom, prix, #id_marque)** qui donne des données sur une liste de produits frais (leur identification (clé primaire), leur nom, leur prix et leur marque (clé étrangère), les champs **Id** et **id_marque** sont de type entier, le champ **nom** est une chaîne de caractères et le champ **prix** est de type flottant.

On a par ailleurs le schéma relationnel **Marque (id, nom)** avec **id** pour clé primaire qui est un tableau donnant les identifiants des marques et les noms des marques.

On va écrire une requête donnant pour chaque marque le nom de la marque et le nombre de produits en vente (qui sont sur le schéma relationnel **Frais** bien entendu).

Comme **nom** apparaît pour **Marque** et **Frais**, on va utiliser les notations **Table.attribut**
De plus, comme deux tables entrent en jeu on va sortir un **JOIN** comme on dit.

La requête SQL est :

```
SELECT Marque.nom, COUNT(Frais.id) FROM Frais JOIN Marque
ON id_marque = Marque.id GROUP BY Marque.nom;
```

Ainsi **GROUP BY** précise le (ou les) attribut(s) utilisés pour le groupement. Ici c'est l'attribut **nom** de **Marque**

On va écrire une requête donnant pour chaque marque le nom de la marque et le prix de son produit le plus cher (qui sont sur le schéma relationnel **Frais** bien entendu).
On modifie un peu la requête précédente.

La requête SQL est :

```
SELECT Marque.nom, Max(Frais.prix) FROM Frais JOIN Marque
ON id_marque = Marque.id GROUP BY Marque.nom;
```

On va écrire maintenant une requête donnant pour chaque marque le nom de la marque et le prix moyen de leurs produits coûtant strictement plus de 10 euros (qui sont sur le schéma relationnel **Frais** bien entendu).
On modifie encore un peu la requête précédente.

La requête SQL est :

```
SELECT Marque.nom, AVG(Frais.prix) FROM Frais JOIN Marque
ON id_marque = Marque.id WHERE Frais.prix > 10 GROUP BY Marque.nom;
```

On remarque ici que **WHERE** est avant **GROUP BY**

2. La clause HAVING

Le mieux encore c'est de reprendre l'exemple précédent avec pour nouvelle requête d'obtenir le nom des marques et le prix moyen de leurs produits si ce prix moyen est supérieur strictement à 10 euros
On modifie la requête précédente. La clause **WHERE** disparaît et est remplacée par **HAVING** mais qui se place maintenant après **GROUP BY**

La requête SQL est :

```
SELECT Marque.nom, AVG(Frais.prix) FROM Frais JOIN Marque
ON id_marque = Marque.id GROUP BY Marque.nom HAVING AVG(Frais.prix)>10;
```

Pour distinguer **WHERE** et **HAVING**, on dira que la clause **WHERE** filtre avant une agrégation et la clause **HAVING** filtre après les groupements.

On peut utiliser **WHERE** et **HAVING** ensemble.

Par exemple prenons la relation **Notes (Date, Exercice , Points)**

```
SELECT Date, Exercice, AVG(Points) FROM Notes
WHERE Date >= "2022-01-01" GROUP BY Date, Exercice
HAVING AVG(Points)>10 ORDER BY Date;
```

VII. Ordre de la composition d'une requête

Il faut bien distinguer l'ordre d'écriture d'une requête et l'ordre d'exécution qui ne sont pas du tout les mêmes.

1. Ordre d'écriture d'une requête SQL

Les seules clauses obligatoires sont **SELECT** et **FROM**, les autres sont à placer à la demande selon la requête voulue. Dans l'ordre d'apparition sur la scène :

SELECT expressions (composées avec des noms de colonnes, séparées par des virgules)

FROM tables (séparées par le mot **JOIN** ou par des virgules)

ON conditions (si jointure avec **JOIN**)

WHERE conditions

GROUP BY attributs (séparés par des virgules)

HAVING conditions

ORDER BY attributs

LIMIT

OFFSET

2. Ordre d'exécution d'une requête

Cet ordre est important car il permet de batir la requête sur le brouillon (avant de la mettre en ordre d'écriture sur votre copie).

▷ **Étape 1 : choix des tables**

FROM : où trouver les informations, dans quelles tables ;

JOIN : s'il y a plusieurs tables à traiter.

▷ **Étape 2 : choix des lignes**

ON : sélection des lignes à traiter si jointure avec **JOIN** ;

WHERE : sélection des lignes à traiter ;

GROUP BY : pour effectuer des regroupements sur les lignes à traiter ;

HAVING : pour ajouter des conditions sur les groupes obtenus.

▷ **Étape 3 : choix des colonnes**

SELECT : choisir les colonnes.

▷ **Étape 4 : Traitement des colonnes**

MIN, MAX, ..., +, -, ... : traiter les colonnes avec des fonctions ou des opérateurs ;

DISTINCT : supprimer les doublons ;

ORDER BY : ordonne les résultats ;

LIMIT OFFSET : ne conserver que certaines lignes de la réponse.

Conséquence de l'ordre d'execution sur le renommage d'une table ou une colonne

Si l'on renomme une table dans la clause **FROM** qui s'execute en premier, ce renommage est disponible pour toute la suite.

Si l'on renomme une colonne dans l'instruction **SELECT**, il faut faire attention.

On peut écrire sans problème **SELECT Id AS n FROM Classe ;**

Par contre ce renommage n'est pas disponible pour **WHERE** par exemple qui est traitée avant.

Ainsi la requête **SELECT Id AS n FROM Classe WHERE n>5 ;** ne fonctionnera pas car **n** est inconnu au moment de la clause **WHERE** qui intervient après **FROM** mais avant **SELECT** dans l'ordre d'exécution.
La solution pour résoudre ce problème est d'écrire **WHERE Id>5** dans la requête précédente.