II Bases de données et langage SQL

29 août 2025

Dans ce chapitre, nous nous intéresserons à l'organisation de données au sein d'une *base de données*, ainsi qu'à la manière de récupérer des informations d'une telle base par des requêtes, que nous écrirons dans le langage SQL.

Nous prendrons comme exemple le jeu Pokemon : la base de données avec laquelle nous travaillerons contiendra les informations des pokémons de la première génération. Afin de conserver une base de données d'une taille raisonnable, nous ne traiterons cela que partiellement.

La base de données est téléchargeable sur mon site (`solnon.fr`). Vous devrez la télécharger et exécuter dessus les requêtes écrites dans ce cours (ainsi que les exercices que vous écrirez). Pour ce faire, je vous conseille d'utiliser le logiciel *DB browser for sqlite*[1].

1 Généralités et vocabulaire sur les bases de données

1.1 Vocabulaire

Les données que nous manipulerons seront contenues dans des *tables*, de la même manière que dans un tableur (*cf.* le fichier `pokebase.ods`).

Exemple 1.1.1 (Aperçu de la table `POKEMONS`)**.**
Voici un aperçu de quelques lignes de la première de nos tables.

numero	nom	evolution	pv
...	...	...	...
4	"Salamèche"		39
5	"Reptincel"	4	58
6	"Dracaufeu"	5	78
...	...	...	...

1. `https://sqlitebrowser.org/`

Par exemple, le pokémon numéro 5 a pour nom `"Reptincel"`, peut s'obtenir en évoluant le pokémon numéro 4, et a 58 points de vie.

Remarque 1.1.2.
Le fait d'avoir des cases vides est légèrement hors programme. Cela n'a aucune incidence sur notre travail, nous ne nous en soucierons donc pas.

Exemple 1.1.3 (Aperçu de la table `TYPES`)**.**
Voici un aperçu de quelques lignes de la deuxième de nos tables.

numero	type
...	...
5	"feu"
6	"feu"
6	"vol"
...	...

Par exemple, le pokémon numéro 6 a deux types : le type `"feu"` et le type `"vol"`.

Exemple 1.1.4 (Aperçu de la table `FAIBLESSES`)**.**
Voici un aperçu de quelques lignes de la troisième de nos tables.

type	faible_contre
...	...
"plante"	"glace"
"feu"	"eau"
"feu"	"roche"
...	...

Par exemple, le type feu est faible à la fois contre le type eau et contre le type roche.

Exemple 1.1.5 (Aperçu de la table `EQUIPES`)**.**
Voisi un aperçu de quelques lignes de la dernière de nos tables.

dresseur	numero	niveau	pv_bonus	date_rencontre
...	...	...	...	...
"Pierre"	95	12	37	"1998-05-12"
"Ondine"	120	18	44	"1998-06-22"
"Ondine"	121	21	65	"1998-06-22"
...	...	...	...	...

Par exemple, on voit que la dresseuse Ondine a dans son équipe (au moins) deux pokémons : le pokémon numéro 120 (niveau 18, 44 points de vie en plus que ses points de vie de base) et numéro 121 (niveau 21, 65 points de vie en plus que ses points de vie de base). La rencontre avec le héros du jeu a eu lieu le 22 juin 1998.

Une ligne sera parfois appelé un *enregistrement*. Le nom d'une colonne sera parfois appelé un *attribut*. Le *schéma* d'une table est la donnée des attributs de cette table.

Exemple 1.1.6.
Dans la table `POKEMONS`, pour l'enregistrement numéro 4, l'attribut `nom` a pour valeur `"Salamèche"`.
Le schéma de la table `POKEMONS` est donc (`numero`,`nom`,`evolution`,`pv`).

Le *domaine* d'un attribut est l'ensemble des valeurs que peuvent prendre cet attribut. Comme tout le temps en informartique, on considérera que ces valeurs sont typées : on manipulera les types usuels : entiers (type `INTEGER`), flottants (type `REAL`), chaînes de caractères (type `VARCHAR`, écrites avec des guillemets), voire des booléens. Les dates pourront être représentées par des chaînes de caractères ou des entiers, même s'il existe les types `DATE` (format AAAA-MM-JJ) et `DATETIME` (format AAAA-MM-JJ HH :MM :SS), que l'on pourra considérer comme des chaînes de caractères. Remarquons que ces derniers formats se comparent naturellement, et que l'ordre lexicographique correspond à l'ordre temporel.

Exemple 1.1.7.
Le domaine de l'attribut `numero` est $\{1, 2, \ldots, 151\}$.

Définition 1.1.8 (Clef primaire)**.**
Une *clef primaire* d'une table est la donnée d'un ou plusieurs attributs permettant de caractériser tout enregistrement de cette table.
Dans le cas d'un unique attribut, la clef primaire sera dite *simple*. Sinon, elle sera dite *composée*.

Remarque 1.1.9.
La plupart du temps, les éléments de la base de données seront définis par un identifiant (ex : numéro client, numéro INSEE, référence d'un produit). Un tel identifiant sera donc pour nous une clef primaire (simple).

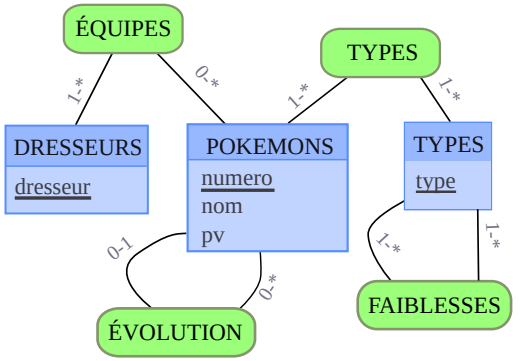
Exercice 1.1.10.
Déterminer les clefs primaires des tables de notre exemple.

Définition 1.1.11 (Clef étrangère)**.**
Une *clef étrangère* d'une table est un attribut de cette table correspondant à une clef primaire (ou une sous-partie, dans le cas d'une clef primaire composée) d'une autre table.

Exercice 1.1.12.
Déterminer les clefs étrangères présentes dans notre exemple.

1.2 Schéma Entité-Association

On représente parfois la structure des données que l'on manipule dans un *diagramme entité-association* (on peut aussi parler de *modèle* entité-association). On distinge alors :
— les entités, qui sont les objets que l'on manipule ;
— les associations, qui sont les relations que l'on exprime entre ces objets.
Dans notre exemple sur les pokémons :
— les entités sont les pokémons, les types de pokémons, les dresseurs ;
— les associations sont :
  — les équipes (association reliant les pokémons et les dresseurs) ;
  — les évolutions (association reliant les pokémons à eux-mêmes) ;
  — les types (association reliant les pokémons aux types) ;
  — les faiblesses des types (association reliant les types à eux-mêmes).
Voici une représentation de ce schéma.



Sur les arêtes reliant les entités aux associations, on a indiqué des cardinalités, sous la forme

cardinalité minimale – cardinalité maximale.

La cardinalité minimale indique le nombre de fois minimum où une entité apparaît dans cette association. Elle peut valoir :
— 0 : l'entité peut ne pas apparaître dans l'association ;
— 1 : l'entité apparaît au moins une fois dans l'association.
La cardinalité maximale indique le nombre de fois maximum où une entité apparaît dans cette association. Elle peut valoir :
— 1 : l'entité peut ne pas apparaître plus d'une fois dans l'association ;
— $*$ (parfois N) : l'entité peut apparaître plus d'une fois dans l'association.

Exemple 1.2.1.
La cardinalité $1-*$ reliant l'entité `DRESSEURS` à l'association `ÉQUIPES` indique que chaque dresseur participe au moins une fois à la relation.

La cardinalité $0-*$ reliant l'entité `POKEMONS` à l'association `ÉQUIPES` indique que chaque pokémons peut ne pas être présent dans aucune équipe, ou bien être plusieurs fois dans une (ou des) équipe.

La cardinalité $0-1$ reliant l'entité `POKEMONS` à l'association `ÉVOLUTION` indique que chaque pokémon provient au plus d'un autre pokémon, par évolution. Cela signifie que cette association peut être exprimée comme une clef étrangère dans la table des pokémons (il suffit d'indiquer le pokémon dont vient l'évolution).

Remarque 1.2.2.
Toute cardinalité du type $1-1$ dans une relation binaire (ou $0-1$, même si cela est légèrement hors programme) indique que l'on peut exprimer cette relation par l'utilisation d'une clef étrangère dans la table de l'entité. Il n'y a donc pas besoin d'écrire une table pour exprimer cette association.

Exercice 1.2.3.
On s'intéresse aux étudiants de CPGE d'un lycée, et des classes du lycée. Pour chaque étudiant, on veut connaître son nom et son prénom. Pour chaque classe, on veut connaître le nom de son professeur responsable. Bien entendu, on veut connaître la classe de chaque étudiant.

Écrire un diagramme entité-associations modélisant ceci, avec les cardinalités associées. Préciser les tables associées, leurs schémas et les types des données.

Exercice 1.2.4.
Toutes les informations de notre base sont-elles bien résumées dans notre modèle entité-association ? Si ce n'est pas le cas, proposer une correction à ce modèle.

2 Requêtes SQL

Une requête SQL est un script, qui respecte la norme SQL. Exécutée par un moteur SQL sur une base de données (au format adapté par le moteur), elle renvoie une nouvelle table.

La requête est analysée par le moteur *avant* d'être exécutée. Les opérations ne seront donc pas toujours exécutées dans l'ordre d'écriture de la requête, comme vous en avez l'habitude dans Python.

Remarquons enfin que l'indentation n'est pas signifiante en SQL. Toutefois, il est primordial d'indenter correctement les blocs SQL, afin d'en assurer la lisibilité.

2.1 Projection : instruction `SELECT ... FROM`

Le bloc de base d'une requête SQL est de la forme :

```
1 SELECT col1,...,coln
2 FROM table ;
```

où `col1,...,coln` sont les noms des colonnes/attributs que l'on désire obtenir, et où `table` est le nom de la table sur laquelle on travaille. Le point virgule signifie la fin de la requête.

Exemple 2.1.1.
On peut récupérer les noms des pokémons de notre base par la requête suivante.

```
3 SELECT nom
4 FROM POKEMONS ;
```

On peut aussi récupérer toute la table des types par la requête suivante.

```
5 SELECT *
6 FROM TYPES ;
```

Exercice 2.1.2.
Récupérer la colonne des différents types de pokémons.

Une table peut contenir plusieurs fois la même ligne. Par exemple, la requête précédente a dû vous afficher un grand nombre de fois chaque type. Pour n'obtenir qu'une seule fois chaque ligne, on utilisera l'instruction `SELECT DISTINCT`.

Exemple 2.1.3.
On récupère donc la liste des types par l'instruction suivante.

```
7 SELECT DISTINCT type
8 FROM TYPES ;
```

2.2 Sélection : instruction `WHERE`

Pour ne récupérer que les lignes vérifiant une certaine condition booléenne, on utilisera le mot clef `WHERE`, de la manière suivante.

```sql
SELECT col1,...,coln
FROM table
WHERE condition ;
```

Cette condition booléenne s'écrira naturellement en utilisant l'égalité, les comparaisons usuelles (`<`, `<=` *etc.*) sur les données numériques, ainsi qu'avec les opérateurs booléens `AND`, `OR` et `NOT`.

Exemple 2.2.1.
Pour obtenir le numéro de Pikachu, on utilise la requête suivante.

```sql
SELECT numero
FROM POKEMONS
WHERE nom = "Pikachu";
```

Pour obtenir les noms des pokémons dont les numéros vont de 10 (inclus) à 20 (exclus), on utilise la requête suivante.

```sql
SELECT nom
FROM POKEMONS
WHERE 10 <= numero
      AND
      numero < 20 ;
```

2.3 Jointure : instruction `JOIN ... ON`

Les requêtes précédentes avaient peu d'intérêt, notamment car elles ne portaient que sur une table. Pour mener une requête portant sur plusieurs tables à la fois, on réalise une *jointure* de ces tables. On utilisera la syntaxe suivante.

```sql
SELECT col1,...,coln
FROM table1
JOIN table2 ON critere
WHERE condition
```

On pourra enchaîner les `JOIN` afin de joindre plusieurs tables.

Dans les cas (simples) qui sont à votre programme, cela revient peu ou prou à mettre bout à bout les tables manipulées. Vous devrez alors indiquer les colonnes qui devront correspondre entre ces tables. La plupart du temps on le signifie par une égalité. Deux tables différentes pouvant partager un même nom de colonne, on lèvera l'ambiguïté par la syntaxe `table.colonne` : cela signifie que l'on considère la colonne `colonne` de la table `table` (et pas d'une autre). Ainsi, le critère de jointure (au programme) aura la forme suivante.

```
table1.colonne1 = table2.colonne2
```

Exemple 2.3.1.
Afin d'afficher les noms des pokémons et leurs types (certains pokémons peuvent deux types), on utilisera la requête suivante.

```sql
SELECT nom, type
FROM POKEMONS
JOIN TYPES ON POKEMONS.numero = TYPES.numero ;
```

Pour obtenir les pokemons de type `feu`, on utilisera la requête suivante.

```sql
SELECT nom
FROM POKEMONS
JOIN TYPES ON POKEMONS.numero = TYPES.numero
WHERE type = 'feu';
```

On pourra enchaîner les jointures avec des instructions `JOIN ... ON` successives.

Exercice 2.3.2.
Écrire une requête permettant d'obtenir la liste des couples (nom,type), où type est une faiblesse du pokémon de ce nom.

On voudra parfois effectuer une jointure d'une table avec elle-même, afin de dédoubler les informations de cette table et de pouvoir mener des recherches plus complexes : on parle alors d'*autojointure*.

Comme la table (et ses colonnes) apparaîtra deux fois dans la requête, il y aura une ambiguïté. On lèvera alors cette ambiguïté en donnant un *alias* à chaque table, avec le mot clef `AS`. Remarquons que l'on peut aussi donner des alias aux colonnes par le même procédé.

Exemple 2.3.3.
On cherche tous les couples de pokémons qui évoluent l'un en l'autre. On va donc réaliser une autojointure sur la table `POKEMONS`, on demandant à ce que le numéro de la première copie (le pokémon avant évolution) corresponde à l'attribut `evolution` de la deuxième copie. On écrit cela comme suit.

```sql
SELECT P1.nom AS avant, P2.nom AS apres
FROM POKEMONS AS P1
JOIN POKEMONS AS P2 ON P1.numero = P2.evolution ;
```

Exercice 2.3.4.
Obtenir la table des triplets (avant,milieu,après) des noms des pokémons évoluant trois fois, les uns dans les autres.

Exercice 2.3.5.
Déterminer les noms des pokémons ayant à la fois le type `eau` et le type `glace`.

2.4 Fonctions de calcul `COUNT`, `MIN` et cie

On peut réaliser certains calculs immédiatement sur les colonnes. Voici la liste des opérations au programme (les noms sont assez transparents) :

- `MIN(col)` : calcule le minimum de la colonne (ordre numérique ou lexicographique) ;
- `MAX(col)` : calcule le maximum de la colonne (*idem*) ;
- `SUM(col)` : calcule la somme des éléments de la colonne (uniquement pour des colonnes numériques) ;
- `AVG(col)` : calcule la moyenne des éléments de la colonne (*idem*) ;
- `COUNT(col)` : compte le nombre de lignes de la colonne (on écrira le plus souvent `COUNT(*)` pour compter le nombre de lignes d'une table).

Ces commandes sont à utiliser juste après le `SELECT`, dans la liste des colonnes, et créent des colonnes *ad-hoc*.

Exemple 2.4.1.
Pour déterminer le nombre de points de vie maximal d'un pokémon, on utilisera la requête suivante.

```
34 SELECT MAX(pv)
35 FROM POKEMONS;
```

Pour récupérer la table des pokémons qui ont ce maximum de points de vie, on peut utiliser une *requête imbriquée*, comme suit.

```
36 SELECT *
37 FROM POKEMONS
38 WHERE pv = (SELECT MAX(pv)
39               FROM POKEMONS
40             );
```

En effet, une table à une ligne et une colonne numérique pourra être interprétée comme un nombre. Nous verrons une écriture plus simple de cette requête dans la partie suivante, sous l'hypothèse de l'unicité d'un tel enregistrement.

Exercice 2.4.2.
Combien de pokémons existe-t-il dans la base ? Combien de pokémons ont le type eau ? Quelle est la moyenne des points de vie des pokémons ? Déterminer le pokémon ayant le moins de points de vie.

2.5 Agrégation et filtrage : instructions `GROUP BY` et `HAVING`

On peut vouloir utiliser les fonctions précédentes non pas sur la table toute entière, mais uniquement sur des sous-groupes particuliers : on dit alors que l'on *agrège* les données. Pour cela, on utilise le mot-clef `GROUP BY`, en indiquant l'attribut par rapport auquel on désire agréger les données. Le calcul va alors se faire pour chaque valeur de cet attribut.

Exemple 2.5.1.
On désire compter pour chaque type le nombre de pokémons de ce type. On va alors réaliser la jointure des deux tables `POKEMONS` et `TYPES`, utiliser la fonction `COUNT`, et regrouper le calcul selon l'attribut `type`.

```
41 SELECT type, COUNT(*) as nbpoke
42 FROM TYPES
43 JOIN POKEMONS ON TYPES.numero=POKEMONS.numero
44 GROUP BY type ;
```

Exercice 2.5.2.
Obtenir la moyenne des points de vie des pokémons de chaque type. Déterminer le type ayant la moyenne la plus élevée.

On peut ensuite vouloir sélectionner les lignes de ce résultat agrégé : on parle de *filtrage*. Cela ne peut se faire par un `WHERE`, car la condition du `WHERE` porte sur les données *avant* agrégation. Pour filtrer les données agrégées, on utilise le mot-clef `HAVING`.

Exemple 2.5.3.
Pour obtenir les types partagés par au moins dix pokémons, on peut reprendre le code précédent et rajouter un `HAVING`.

```
45 SELECT type, COUNT(*) as nbpoke
46 FROM TYPES
47 JOIN POKEMONS ON TYPES.numero=POKEMONS.numero
48 GROUP BY type
49 HAVING nbpoke >= 10 ;
```

On peut aussi écrire la requête suivante.

```
50 SELECT type
51 FROM TYPES
52 JOIN POKEMONS ON TYPES.numero=POKEMONS.numero
53 GROUP BY type
54 HAVING COUNT(*) >= 10 ;
```

Remarque 2.5.4.
La confusion entre `WHERE` et `HAVING` est une cause fréquente d'erreur. Je ne peux que vous encourager à bien retenir ce point de cours.

Exercice 2.5.5.
Déterminer les pokémons qui ont deux types différents.

Exercice 2.5.6.
Déterminer les types pour lesquels il existe au moins un pokémon ayant 100 points de vie, ou plus.

Exercice 2.5.7.
Déterminer la liste des types partagés par au moins 5 pokémons dont les numéros sont inférieurs ou égaux à 100.

2.6 Tris et cie

Il est possible d'effectuer un tri (selon une colonne) avec le mot clef `ORDER BY`, suivi du nom de la colonne, ainsi que du type de tri (`ASC` pour un tri par ordre croissant, `DESC` par ordre décroissant).

Exemple 2.6.1.
Pour trier les pokémons par ordre alphabétique, on peut donc écrire la requête suivante.

```
55 SELECT *
56 FROM POKEMONS
57 ORDER BY nom ASC ;
```

De manière plus avancée, on peut indiquer plusieurs colonnes de tri, afin d'indiquer comment trier les *ex-æquo* du premier critère.

Exemple 2.6.2.
Pour trier les pokémons par nombre de points de vie décroissant, et en cas d'égalité par ordre alphabétique, on utilisera la requête suivante.

```
58 SELECT *
59 FROM POKEMONS
60 ORDER BY pv DESC, nom ASC ;
```

On peut aussi obtenir les `n` premières lignes d'une table par la commande `LIMIT n`, et obtenir ces `n` premières lignes de la table obtenue en ayant sauté les `p` premières lignes de la table (donc, les lignes numérotées de $n+1$ à $n+p$ dans la table de départ) par l'utilisation successives des commandes `LIMIT n` et `OFFSET p`

Remarque 2.6.3.
Attention, la commande `OFFSET` ne peut s'utiliser sans `LIMIT`.

Remarque 2.6.4.
Ces commandes peuvent bien entendu s'utiliser sans tri. La plupart du temps, leur utilisation ne sera intéressante que sur des données triées.

Exemple 2.6.5.
Pour obtenir les dix premiers pokémons, par ordre alphabétique, on utilisera la requête suivante.

```
61 SELECT *
62 FROM POKEMONS
63 ORDER BY nom ASC
64 LIMIT 10 ;
```

Pour obtenir la liste des pokémons situés du 21e au 30e rang, par ordre alphabétique, on utilisera cette requête-ci.

```
65 SELECT *
66 FROM POKEMONS
67 ORDER BY nom ASC
68 LIMIT 10
69 OFFSET 20 ;
```

Exercice 2.6.6.
Quel est le 42e pokémon (par ordre alphabétique) ?

Exercice 2.6.7.
Écrire deux requêtes différentes afin d'obtenir la liste des 10 derniers pokémons (par ordre alphabétique), donnée dans l'ordre alphabétique.

2.7 Opérations ensemblistes : `UNION`, `INTERSECT`, `EXCEPT`, produit cartésien.

On peut réaliser les opérations ensemblistes classiques (union, intersection, différence) avec les mots-clefs respectifs (`UNION`, `INTERSECT`, `EXCEPT`). Ces mots clefs se placent entre deux tables ayant le même schéma, donc entre deux `SELECT`.

Exemple 2.7.1.
Pour obtenir les noms des pokémons qui sont de type `plante` ou de type `poison`, on utilisera la requête suivante.

```
70 SELECT nom FROM POKEMONS
71 JOIN TYPES ON POKEMONS.numero = TYPES.numero
```

```
72 WHERE type = 'plante'
73
74 UNION
75
76 SELECT nom FROM POKEMONS
77 JOIN TYPES ON POKEMONS.numero = TYPES.numero
78 WHERE type = 'poison'
79 ;
```

Pour obtenir ceux des pokémons qui sont de type `plante` sans être pour autant de type `poison`, on utilisera la requête suivante.

```
80 SELECT nom FROM POKEMONS
81 JOIN TYPES ON POKEMONS.numero = TYPES.numero
82 WHERE type = 'plante'
83
84 EXCEPT
85
86 SELECT nom FROM POKEMONS
87 JOIN TYPES ON POKEMONS.numero = TYPES.numero
88 WHERE type = 'poison'
89 ;
```

Pour effectuer le produit cartésien de deux tables (*i.e.* une jointure des tables sans condition), on indiquera les deux tables juste après le `FROM`, séparées d'une virgule.

Remarque 2.7.2.
Autant que faire se peut, on évitera cette syntaxe.

Exemple 2.7.3.
Pour effectuer le produit cartésien des tables `TYPES` et `FAIBLESSES`, on utilisera la requête suivante.

```
90 SELECT *
91 FROM TYPES, FAIBLESSES ;
```

2.8 Opérations numériques : $+$, $-$, $*$ et $/$.

On peut effectuer les opérations usuelles sur les colonnes numériques. Le programme demandant explicitement de ne pas rentrer dans les détails de la division d'entiers, nous ne nous aventurerons pas dans cette voie.

Exemple 2.8.1.
Pour calculer les points de vie réels de chaque pokémon de chaque dresseur, on additionne donc les points de vie supplémentaires de chaque pokémon aux points de vie de base de ce type de pokémon.

```
92 SELECT dresseur, nom, pv + pv_bonus as pv_tot
93 FROM EQUIPES
94 JOIN POKEMONS ON EQUIPES.numero = POKEMONS.numero ;
```

Exercice 2.8.2.
Calculer les rapports entre le bonus de points de vie et le niveau des pokémons de la table `EQUIPE`. Quel pokémon a le meilleur rapport ?

3 Exercices

Exercice 3.0.1.
Quels sont les types faibles contre eux-mêmes ?

Exercice 3.0.2.
Écrire une requête donnant les noms des pokémons ayant une faiblesse contre eux-mêmes (exemple : Bulbizarre est de type `plante` et du type `poison`, or le type `plante` a une faiblesse contre le type `poison`).

Exercice 3.0.3.
Obtenir la liste des pokémons qui peuvent évoluer en plusieurs autres pokémons.

Exercice 3.0.4.
Écrire deux requêtes différentes permettant chacune d'obtenir la liste des pokémons ayant deux types différents.

Exercice 3.0.5.
Vérifier que les dresseurs ont chacun au plus six pokémons dans leur équipe.

Exercice 3.0.6.
Calculer la moyenne des niveaux des pokémons de chaque dresseur. Quels dresseurs ont le meilleur niveau moyen dans leur équipe ?

Exercice 3.0.7.
Pour chaque dresseur, calculer la somme des points de vie (totaux) de chaque pokémon.

Exercice 3.0.8.
Déterminer le type de pokémon le plus présent parmi les pokémons des dresseurs.

Exercice 3.0.9.
Calculer les moyennes par dresseur des points de vie des pokémons de type `eau` (pour les dresseurs qui en possèdent). Déterminer ceux pour lesquels cette moyenne vaut au moins 100.