Chapitre 4 Manipuler des données

4.1 Les principes des packages dplyr

Le but de dplyr est d'identifier et de rassembler dans un seul package les outils de manipulation de données les plus importantes pour l'analyse des données. Ce package rassemble donc des fonctions correspondant à un ensemble d'opérations élémentaires (ou *verbes*) qui permettent de :

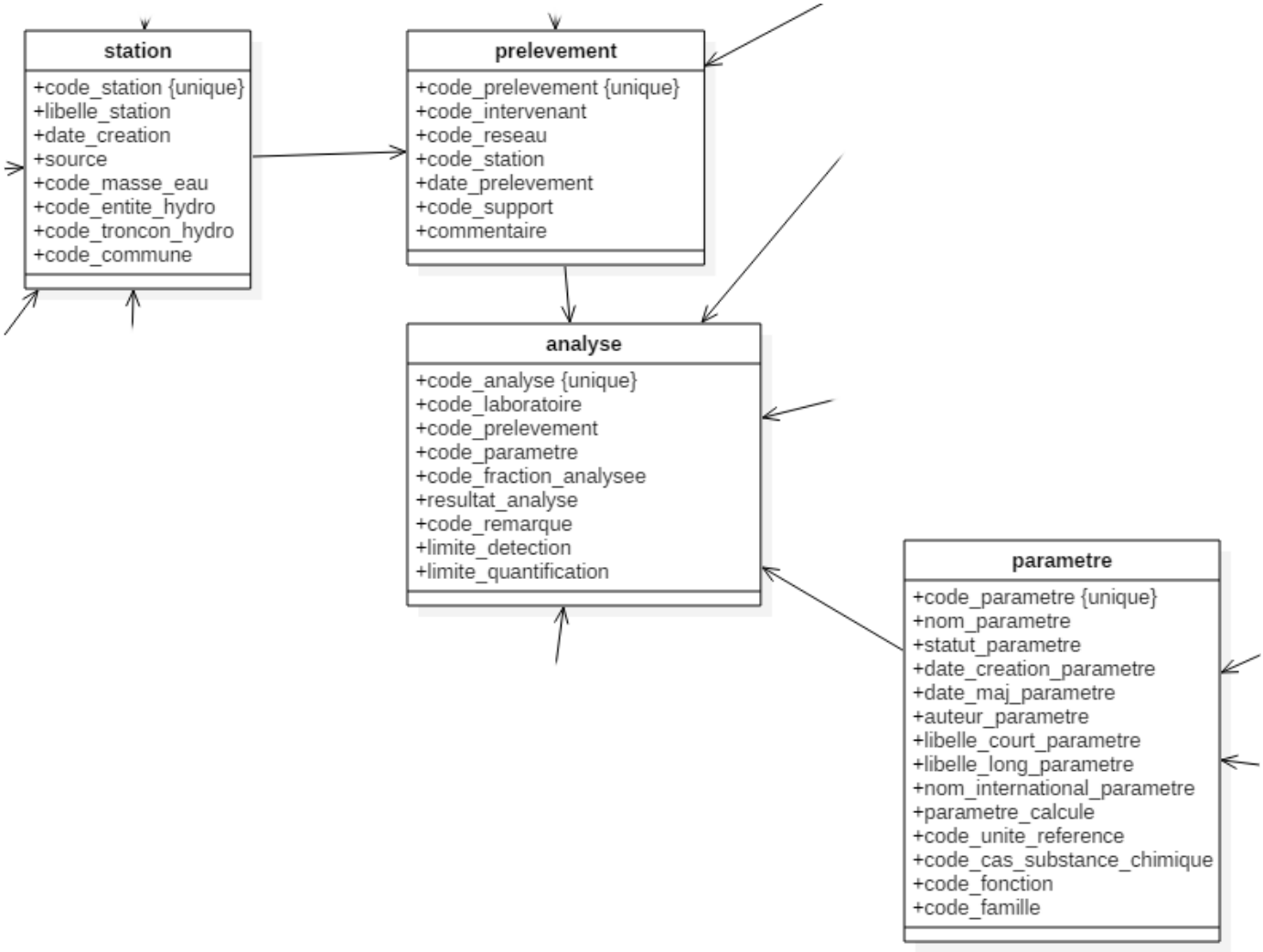
- Sélectionner un ensemble de variables : ***select()***
- Sélectionner un ensemble de lignes : ***filter()***
- Ajouter/modifier/renommer des variables : ***mutate()*** ou ***rename()***
- Produire des statistiques aggrégées sur les dimensions d'une table : ***summarise()***
- Trier une table : ***arrange()***
- Manipuler plusieurs tables : ***left_join()***, ***right_join()***, ***full_join()***, ***inner_join()***…

D'appliquer cela sur des données, quel que soit leur format : data frames, data.table, base de données sql, big data…

D'appliquer cela en articulation avec **group_by()** qui change la façon d'interpréter chaque fonction : d'une interprétation *globale* sur l'ensemble d'une table, on passe alors à une approche *groupe par groupe* : chaque groupe étant défini par un ensemble des modalités des variables défini dans l'instruction **group_by()**.

4.2 Présentation des données

On va travailler sur ce module principalement à partir des données sitadel en date réelle estimée et à partir des données de qualité des eaux de surface.



4.3 Chargement des données

```
load (file = "data/FormationPreparationDesDonnées.RData")
```

4.4 Les verbes clefs de dplyr pour manipuler une table

4.4.1 Sélectionner des variables : select()

Nous allons ici sélectionner un ensemble de variables de la table des prélèvements

```
prelevementb <- select (prelevement, date_prelevement, code_prelevement,
                        code_reseau, code_station)
datatable (prelevementb)
```

Show 10 entries Search:

	date_prelevement	code_prelevement	code_reseau	code_station
1	1992-07-08	1	ARS	044000110
2	1992-12-15	2	ARS	044000110
3	1993-04-21	3	ARS	044000110
4	1993-11-25	4	ARS	044000110
5	1994-04-26	5	ARS	044000110
6	1995-04-11	6	ARS	044000110
7	1996-09-23	8	ARS	044000110
8	1997-04-22	11	ARS	044000110
9	1997-05-25	12	ARS	044000110
10	1997-09-23	13	ARS	044000110

Showing 1 to 10 of 22,224 entries

Previous 1 2 3 4 5 … 2223 Next

```
prelevementb <- select (prelevement, -commentaire)
names (prelevementb)
```

```
## [1] "code_prelevement" "code_intervenant" "code_reseau"
## [4] "code_station"     "date_prelevement" "code_support"
```

select() possède ce qu'on appelle des helpers qui permettent de gagner du temps dans l'écriture de notre select. A partir du moment où les conventions de nommage sont correctement effectuées, cela permet de gagner également en reproductibilité d'une année sur l'autre.

Exemple d'usage : récupérer toutes les variables qui commencent par "code" :

```
prelevementb <- select (prelevement, starts_with ("code_"))
```

Exemple d'usage : récupérer toutes les variables qui commencent par "code" et date_prelevement :

```
prelevementb <- select (prelevement, starts_with ("code_"), one_of ("date_prelevement"))
```

4.4.2 Trier une table : arrange()

```
prelevementb <- arrange (prelevementb, date_prelevement)
```

4.4.3 Renommer une variable : rename()

```
prelevementb <- rename (prelevementb, date_p = date_prelevement)
```

On peut aussi directement renommer une variable dans l'opération **select()**

```
prelevementb <- select (prelevement, date_p = date_prelevement, code_prelevement,
                        code_reseau, code_station)
```

4.4.4 Filter une table : filter()

On va ici récupérer les analyses produites par l'ARS

```
ars <- filter (prelevement, code_reseau == "ARS")
```

L'exemple ci-dessus n'exerce un filtre que sur une condition unique.

Pour des conditions cumulatives (toutes les conditions doivent être remplies), le “&” ou la “,”

```
ars <- filter (prelevement, code_reseau == "ARS", code_intervenant == "44")
```

Pour des conditions non cumulatives (au moins une des conditions doit être remplie), le “|”

```
ars <- filter (prelevement, code_reseau == "ARS" | code_reseau == "FREDON")
```

Si une condition non cumulative s’applique sur une même variable, privilégier un test de sélection dans une liste avec le %in%

```
ars <- filter (prelevement, code_reseau %in% c ("ARS", "FREDON"))
```

Pour sélectionner des observations qui ne répondent pas à la condition, le !

Toutes les observations ayant été réalisées par un autre réseau que l’ARS

```
non_ars <- filter (prelevement, !(code_reseau == "ARS"))
```

4.4.5 Modifier/rajouter une variable : mutate()

mutate() est le verbe qui permet la transformation d’une variable existante ou la création d’une nouvelle variable dans le jeu de données.

Création de nouvelles variables

```
prelevementb <- mutate (prelevementb,
                        code_prelevement_caract = as.character (code_prelevement),
                        code_reseau_fact = as.factor (code_reseau))
```

Modification de variables existantes

```
prelevementb <- mutate (prelevementb,
                        code_prelevement = as.character (code_prelevement),
                        code_reseau = as.factor (code_reseau))
```

mutate() possède une variante, **transmute()**, qui fonctionne de la même façon que **mutate()**, mais ne garde que les variables modifiées ou crées par le verbe.

4.4.6 Extraire un vecteur : pull()

pull() permet d'extraire une variable d'un dataframe comme un vecteur

```
stations_de_la_table_prelevement <- pull (prelevement, code_station)
stations_de_la_table_prelevement <- unique (stations_de_la_table_prelevement)
```

4.5 La boîte à outils pour créer et modifier des variables avec R

4.5.1 Manipuler des variables numériques

Vous pouvez utiliser beaucoup de fonction pour créer des variables avec **mutate()**.

- Les opérations arithmétiques : +,-,**,/,^*
- Arithmétique modulaire : *%/%* (division entière) et *%%* (le reste), où x == y * (x %/% y) + (x %% y)
- Logarithmes : *log()*, *log2()*, *log10()*
- Navigations entre les lignes : *lead()* et *lag()* qui permettent d'avoir accès à la valeur suivante et précédente d'une variable.

```
x <- sample (1:10)
lagx <- lag (x)
leadx <- lead (x)
lag2x <- lag (x, n = 2)
lead2x <- lead (x, n = 2)
cbind (x = x, lagx = lagx, lag2x = lag2x, leadx = leadx, lead2x = lead2x)
```

```
##        x lagx lag2x leadx lead2x
##  [1,]  3   NA    NA     1      9
##  [2,]  1    3    NA     9      7
##  [3,]  9    1     3     7      4
##  [4,]  7    9     1     4     10
##  [5,]  4    7     9    10      2
##  [6,] 10    4     7     2      5
##  [7,]  2   10     4     5      6
##  [8,]  5    2    10     6      8
##  [9,]  6    5     2     8     NA
## [10,]  8    6     5    NA     NA
```

- opérations cumulatives ou glissantes :
    - R fournit des fonctions pour obtenir **opérations cumulatives** les somme, produit, minimum et maximum cumulés, dplyr fournit l'équivalent pour les moyennes : *cumsum()*, *cumprod()*, *cummin()*, *cummax()*, *cummean()*
    - Pour appliquer des **opérations glissantes**, on peut soit créer l'opération avec l'instruction *lag()*, soit exploiter le package **RcppRoll** qui permet d'exploiter des fonctions prédéfinies.

Exemple de somme glissante sur un pas de 2 observations.

```
x <- sample (1:10)
cumsumx <- cumsum (x)
rollsumx <- roll_sum (x, n=2)
rollsumx
```

```
## [1]  6 11  9  7  6  9 17 18 17
```

La fonction `roll_sumr()` fait en sorte d'obtenir un vecteur de même dimension que l'entrée x

```
rollsumrx <- roll_sumr (x, n=2)
rollsumrx
```

```
##  [1] NA  6 11  9  7  6  9 17 18 17
```

```
length(rollsumrx) == length(x)
```

```
## [1] TRUE
```

Nous pouvons obtenir une matrice des différenntes valeurs calculées :

```
cbind (x = x, cumsumx = cumsum (x), rollsumx = rollsumx, rollsumrx = roll_sumr (x, n=2
```

```
##        x cumsumx rollsumx rollsumrx
##  [1,]  1       1        6        NA
##  [2,]  5       6       11         6
##  [3,]  6      12        9        11
##  [4,]  3      15        7         9
##  [5,]  4      19        6         7
##  [6,]  2      21        9         6
##  [7,]  7      28       17         9
##  [8,] 10      38       18        17
##  [9,]  8      46       17        18
## [10,]  9      55        6        17
```

- Comparaisons logiques : <, <=, >, >=, *!=*
- Rangs : **min_rank()** devrait être la plus utile, il existe aussi notamment **row_number()**, **dense_rank()**, **percent_rank()**, **cume_dist()**, **ntile()**.
- **coalesce (x, y)** : permet de remplacer les valeurs manquantes de x par celle de y

- ***variable = ifelse (condition (x), valeursi, valeursinon)*** permet d'affecter *valeursi* ou *valeursinon* à *variable* en fonction du fait que *x* répond à *condition*. Exemple : création d'une variable résultat pour savoir si le résultat de nos analyses est bon ou non.

```
analyseb <- mutate (analyse,
                    resultat_ok = ifelse (code_remarque %in% c (1,2,7,10),
                                          yes = TRUE, no = FALSE))
```

- ***case_when()*** permet d'étendre la logique de ifelse à des cas plus complexes. Les conditions mises dans un ***case_when()*** ne sont pas exclusives. De ce fait, il faut pouvoir déterminer l'ordre d'évaluation des conditions qui y sont posées. Cet ordre s'effectue de bas en haut, c'est à dire que la dernière condition évaluée (celle qui primera sur toutes les autres) sera la première à écrire. Exemple: On va ici recalculer des seuils fictifs sur les analyses.

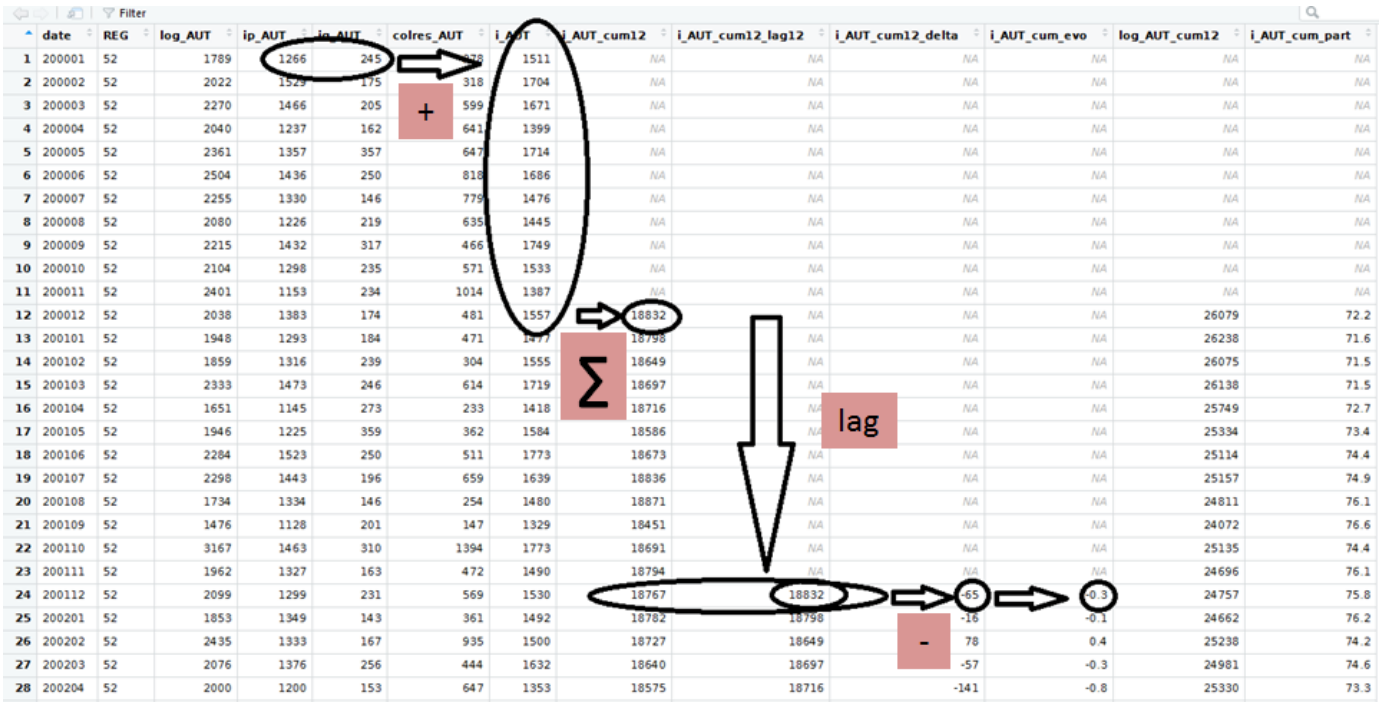
```
analyseb <- mutate (analyse, classe_resultat_analyse = case_when (
                        resultat_analyse == 0  ~ "1",
                        resultat_analyse <= 0.001  ~ "2",
                        resultat_analyse <= 0.01  ~ "3",
                        resultat_analyse <= 0.1  ~ "4",
                        resultat_analyse > 0.1  ~ "5",
                        TRUE ~ ""
                    ))
```

4.5.2 Exercice : Les données mensuelles sitadel

A partir du fichier sitadel de février 2017 (ROES_201702.xls), sur la région Pays de la Loire (code région 52), livrer un fichier contenant pour chaque mois, pour les logements individuels (i_AUT = ip_AUT + ig_AUT) :

- le cumul des autorisations sur 12 mois glissants(i_AUT_cum12)
- le taux d'évolution du cumul sur 12 mois (i_AUT_cum_evo, en %)
- la part de ce cumul dans celui de l'ensemble des logements autorisés (log_AUT), en pourcentage

```
sitadel <- read_excel ("data/ROES_201702.xls", sheet = "AUT_REG",
                       col_types = c ("text","text","numeric","numeric","numeric","numeric"
```



	date	REG	log_AUT	ip_AUT	ip_AUT_	colres_AUT	i_AUT	i_AUT_cum12	i_AUT_cum12_lag12	i_AUT_cum12_delta	i_AUT_cum_evo	log_AUT_cum12	i_AUT_cum_part
1	200001	52	1789	1266	245	[illegible]	1511	NA	NA	NA	NA	NA	NA
2	200002	52	2022	1529	175	318	1704	NA	NA	NA	NA	NA	NA
3	200003	52	2270	1466	205	599	1671	NA	NA	NA	NA	NA	NA
4	200004	52	2040	1237	162	641	1399	NA	NA	NA	NA	NA	NA
5	200005	52	2361	1357	357	647	1714	NA	NA	NA	NA	NA	NA
6	200006	52	2504	1436	250	818	1686	NA	NA	NA	NA	NA	NA
7	200007	52	2255	1330	146	779	1476	NA	NA	NA	NA	NA	NA
8	200008	52	2080	1226	219	635	1445	NA	NA	NA	NA	NA	NA
9	200009	52	2215	1432	317	466	1749	NA	NA	NA	NA	NA	NA
10	200010	52	2104	1298	235	571	1533	NA	NA	NA	NA	NA	NA
11	200011	52	2401	1153	234	1014	1387	NA	NA	NA	NA	NA	NA
12	200012	52	2038	1383	174	481	1557	18832	NA	NA	NA	26079	72.2
13	200101	52	1948	1293	184	471	1477	18798	NA	NA	NA	26238	71.6
14	200102	52	1859	1316	239	304	1555	18649	NA	NA	NA	26075	71.5
15	200103	52	2333	1473	246	614	1719	18697	NA	NA	NA	26138	71.5
16	200104	52	1651	1145	273	233	1418	18716	NA	NA	NA	25749	72.7
17	200105	52	1946	1225	359	362	1584	18586	NA	NA	NA	25334	73.4
18	200106	52	2284	1523	250	511	1773	18673	NA	NA	NA	25114	74.4
19	200107	52	2298	1443	196	659	1639	18836	NA	NA	NA	25157	74.9
20	200108	52	1734	1334	146	254	1480	18871	NA	NA	NA	24811	76.1
21	200109	52	1476	1128	201	147	1329	18451	NA	NA	NA	24072	76.6
22	200110	52	3167	1463	310	1394	1773	18691	NA	NA	NA	25135	74.4
23	200111	52	1962	1327	163	472	1490	18794	NA	NA	NA	24696	76.1
24	200112	52	2099	1299	231	569	1530	18767	18832	-65	-0.3	24757	75.8
25	200201	52	1853	1349	143	361	1492	18782	18798	-16	-0.1	24662	76.2
26	200202	52	2435	1333	167	935	1500	18727	18649	78	0.4	25238	74.2
27	200203	52	2076	1376	256	444	1632	18640	18697	-57	-0.3	24981	74.6
28	200204	52	2000	1200	153	647	1353	18575	18716	-141	-0.8	25330	73.3

4.5.3 Manipuler des dates

Parmi l'ensemble des manipulations de variables, celle des dates et des heures est toujours une affaire complexe. Le framework tidyverse propose le package ***lubridate*** qui permet de gérer ces informations de façon cohérente.

- gestion des dates :

```
dmy ("jeudi 21 novembre 2017")
dmy ("21112017")
ymd ("20171121")
```

- gestion des dates/heures :

```
dmy_hms ("mardi 21 novembre 2017 9:30:00")
```

- combien de jours avant Noël ?

```
dmy ("25 décembre 2018") - dmy ("16 avril 2018")
```

- le jour de la semaine d'une date :

```
wday (dmy ("19012038"), label = TRUE)
```

La fonction make_date et make_datetime vous permettent de transformer un ensemble de variables en un format date ou date heure. Utile par exemple lorsque l'on a une variable pour l'année, le mois et le jour.

Exercice : convertir les données de la table exercice pertinentes au format date.

4.5.4 Manipuler des chaînes de caractères

Le package *stringr* compile l'ensemble des fonctions de manipulation de chaînes de caractère utiles sur ce type de données.

On peut diviser les manipulations de chaîne de caractère en 4 catégories :

- manipulations des caractères eux-mêmes
- gestion des espaces
- opérations liées à la langue
- manipulations de "pattern", notamment des expressions régulières.

4.5.4.1 Manipulations sur les caractères

Obtenir la longueur d'une chaîne

```
str_length ("abc")
```

```
## [1] 3
```

Extraire une chaîne de caractère

str_sub prend 3 arguments : une chaîne de caractère, une position de début, une position de fin. Les positions peuvent être positives, et dans ce cas, on compte à partir de la gauche, ou négatives, et dans ce cas on compte à partir de la droite.

```
x <- c ("abcdefg", "hijklmnop")
str_sub (string = x, start = 3, end = 4)
```

```
## [1] "cd" "jk"
```

```
str_sub (string = x, start = 3, end = -2)
```

```
## [1] "cdef"   "jklmno"
```

str_sub peut être utilisé pour remplacer un caractère

```
str_sub (x, start = 3, end = 4) <- "CC"
x
```

```
## [1] "abCCefg"   "hiCClmnop"
```

4.5.4.2 Gestion des espaces

la fonction **str_pad()** permet de compléter une chaîne de caractère pour qu'elle atteigne une taille fixe. Le cas typique d'usage est la gestion des codes communes Insee.

```
code_insee <- 1001
str_pad (code_insee, 5, pad = "0")
```

```
## [1] "01001"
```

On peut choisir de compléter à gauche, à droite, et on peut choisir le "pad". Par défaut, celui-ci est l'espace.

La fonction inverse de **str_pad()** est **str_trim()** qui permet de supprimer les espaces aux extrémités de notre chaîne de caractères.

```
proust <- "   Les paradoxes d'aujourd'hui sont les préjugés de demain.  "
str_trim (proust)
```

```
## [1] "Les paradoxes d'aujourd'hui sont les préjugés de demain."
```

```
str_trim (proust, side = "left")
```

```
## [1] "Les paradoxes d'aujourd'hui sont les préjugés de demain.  "
```

Les **expressions régulières** permettent la détection de “patterns” sur des chaîne de caractères.

```
txt <- c ("voiture","train", "voilier", "bus", "avion", "tram", "trotinette")
str_detect (string = txt, pattern = "^tr") # les éléments qui commencent pas les lettre
```

```
## [1] FALSE  TRUE FALSE FALSE FALSE  TRUE  TRUE
```

```
txt [str_detect (string = txt, pattern = "^tr")]
```

```
## [1] "train"      "tram"       "trotinette"
```

```
str_detect (string = txt, pattern = "e$") # les éléments qui terminent par la lettre e
```

```
## [1]  TRUE FALSE FALSE FALSE FALSE FALSE  TRUE
```

```
txt [str_detect (string = txt, pattern = "e$")]
```

```
## [1] "voiture"    "trotinette"
```

4.5.4.3 Opérations liées à la langue

Ces différentes fonctions ne donneront pas le même résultat en fonction de la langue par défaut utilisée. La gestion des majuscules/minuscules :

```
proust <- "Les paradoxes d'aujourd'hui sont LES préjugés de Demain."
str_to_upper (proust)
```

```
## [1] "LES PARADOXES D'AUJOURD'HUI SONT LES PRÉJUGÉS DE DEMAIN."
```

```
str_to_lower (proust)
```

```
## [1] "les paradoxes d'aujourd'hui sont les préjugés de demain."
```

```
str_to_title (proust)
```

```
## [1] "Les Paradoxes D'aujourd'hui Sont Les Préjugés De Demain."
```

La gestion de l'ordre :

```
x <- c ("y", "i", "k")
str_order (x)
```

```
## [1] 2 3 1
```

```
str_sort (x)
```

```
## [1] "i" "k" "y"
```

Suppression des accents (base::iconv) :

```
proust2 <- "Les paradoxes d'aujourd'hui sont les préjugés de demain ; et ça c'est embêta
iconv (proust2, to = "ASCII//TRANSLIT")
```

```
## [1] "Les paradoxes d'aujourd'hui sont les prejuges de demain ; et ca c'est embetant"
```

4.5.5 Manipuler des variables factorielles (=qualitatives)

Les fonctions du module forcats permettent de modifier les modalités d'une variable factorielle, notamment :

- Changer les modalités des facteurs et/ou leur ordre
- Regrouper des modalités

On va ici utiliser cette fonction pour modifier le tri des stations en fonction de leur fréquence d'apparition dans la table "prelevement""

forcats permet beaucoup d'autres possibilités de tri :

- manuellement des facteurs (***fct_relevel()***);
- en fonction de la valeur d'une autre variable (***fct_reorder()***);
- en fonction de l'ordre d'apparition des modalités (***fct_inorder()***).

Consulter la doc du module pour voir toutes les possibilités très riches de ce module.

En quoi ces fonctions sont utiles ?

Elles permettent notamment :

- lorsqu'on fait des graphiques, d'afficher les occurences les plus importantes d'abord ;
- de lier l'ordre d'une variable en fonction d'une autre (par exemple les code Insee des communes en fonction des régions).

Exemple : ordonner les modalités d'un facteur pour améliorer l'aspect d'un graphique

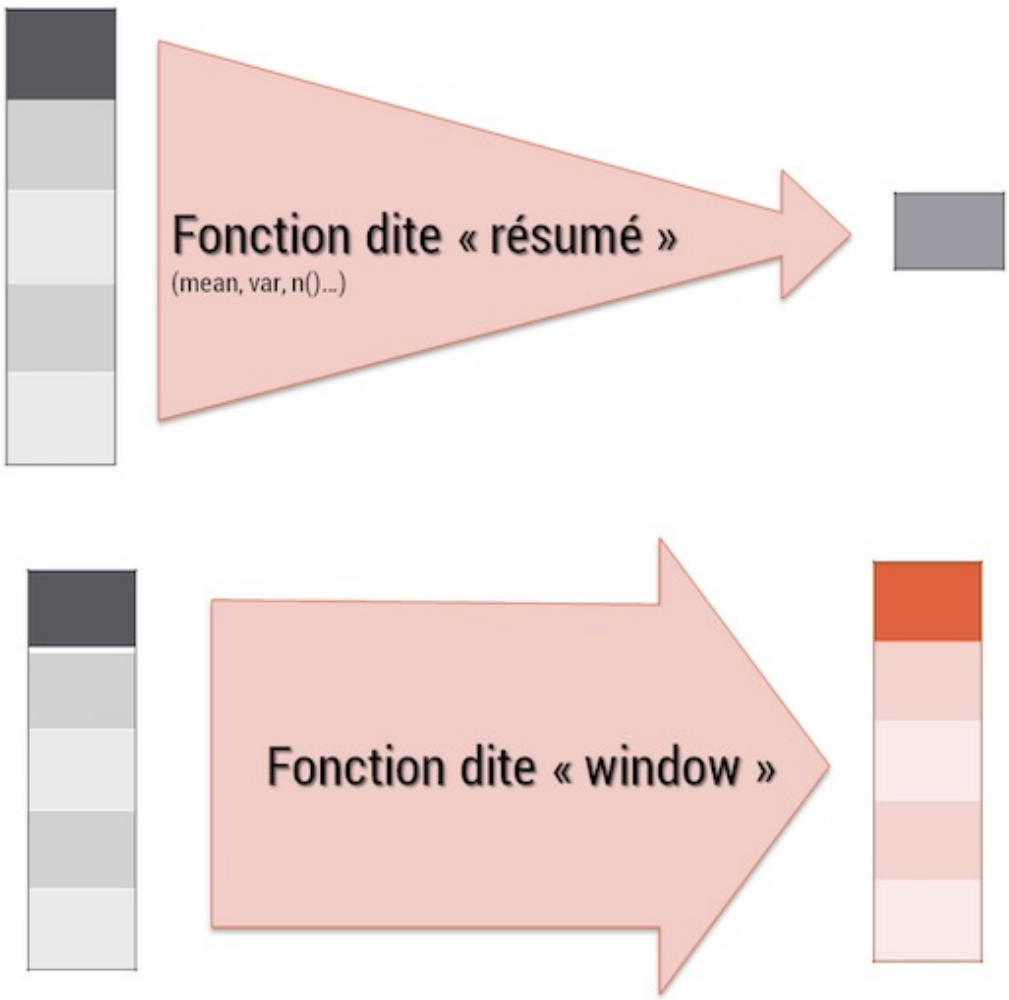
```
library (ggplot2)
library (forcats)
num <- c (1, 8, 4, 3, 6, 7, 5, 2, 11, 3)
cat <- c (letters [1:10])
data <- data.frame (num, cat)

ggplot (data, aes (x = cat, num)) +
  geom_bar (stat = "identity") +
    xlab (label = "Facteur") + ylab (label ="Valeur")
```

```
ggplot (data, aes (x = fct_reorder (cat, -num), num)) +
  geom_bar (stat = "identity") +
  xlab (label = "Facteur ordonné") + ylab (label ="Valeur")
```

4.6 Aggréger des données : summarise()

La fonction ***summarise()*** permet d'aggréger des données, en appliquant une fonction sur les variables pour construire une statistique sur les observations de la table. **summarise()** est une fonction dite de "résumé". À l'inverse de **mutate()**, quand une fonction summarise est appelée, elle retourne une seule information. La moyenne, la variance, l'effectif…sont des

informations qui condensent la variable étudiée en une seule information.



La syntaxe de summarise est classique. Le resultat est un dataframe

```
summarise (exercice,
           mesure_moyenne = mean (resultat_analyse, na.rm = T))
```

On peut calculer plusieurs statistiques sur une aggrégation

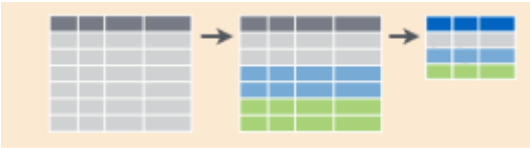
```
summarise (exercice,
           mesure_moyenne = mean (resultat_analyse, na.rm = T),
           mesure_total = sum (resultat_analyse, na.rm = T)
           )
```

4.6.1 Quelques fonctions d'aggrégations utiles

- compter : *n()*
- sommer : *sum()*

- compter des valeurs non manquantes *sum(!is.na())*
- moyenne : *mean()*, moyenne pondérée : *weighted.mean()*
- écart-type : *sd()*
- médiane : *median()*, quantile : *quantile(.,quantile)*
- minimum : *min()*, maximum : *max()*
- position : *first()*, *nth(., position)*, *last()*

4.7 Aggréger des données par dimension : group_by()

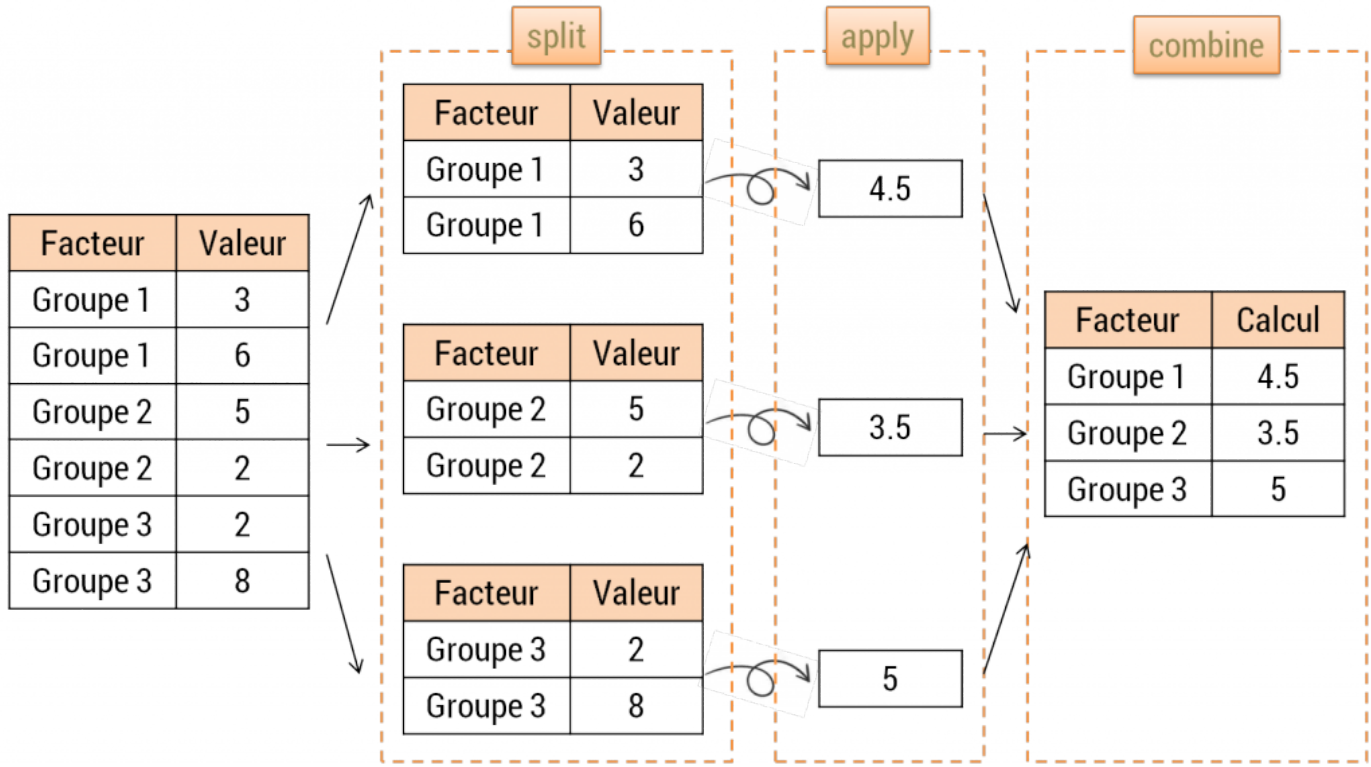


Summarise est utile, mais la plupart du temps, nous avons besoin non pas d'aggréger des données d'une table entière, mais de construire des aggrégations sur des sous-ensembles : par années, départements… La fonction ***group_by()*** va permettre *d'éclater* notre table en fonction de dimensions de celle-ci.

Ainsi, si on veut construire des statistiques agrégées non sur l'ensemble de la table, mais pour chacune des modalités d'une ou de plusieurs variables de la table. Il faut deux étapes :

- Utiliser prélablement la fonction ***group_by()*** pour définir les variables sur lesquelles on souhaite aggréger les données.
- Utiliser ***summarise()*** ou ***summarise_XX()*** sur la table en sortie de l'étape précédente

Découper un jeu de données pour réaliser des opérations sur chacun des sous-ensembles afin de les restituer ensuite de façon organisée est appelée stratégie du split – apply – combine schématiquement, c'est cette opération qui est réalisée par dplyr dès qu'un ***group_by()*** est introduit sur une table.



Exemple pour calculer les statistiques précédentes par mois :

```
exercice <- mutate (exercice,
                    annee = year (date_prelevement))

paran <- group_by (exercice, annee)

summarise (paran,
           mesure_moyenne = mean (resultat_analyse, na.rm = T),
           mesure_total = sum (resultat_analyse, na.rm = T)
          )
```

```
## # A tibble: 26 x 3
##    annee mesure_moyenne mesure_total
##    <dbl>          <dbl>        <dbl>
##  1 1991.         0.0981         4.32
##  2 1992.         0.137          8.33
##  3 1993.         0.123          6.14

##  4 1994.         0.0684         4.72
##  5 1995.         0.0803         6.99
##  6 1996.         0.0915         6.86
##  7 1997.         0.0529         5.14
##  8 1998.         0.131         46.5
##  9 1999.         0.0547        89.7
## 10 2000.         0.118        191.
## # ... with 16 more rows
```

Pour reprendre des traitements “table entière”, il faut mettre fin au ***group_by()*** par un ***ungroup()***

4.8 Le pipe

Le pipe est la fonction qui va vous permettre d’écrire votre code de façon plus lisible pour vous et les utilisateurs. Comment ? En se rapprochant de l’usage usuel en grammaire.

`verbe(sujet,complement)` devient `sujet %>% verbe(complement)`

Quand on enchaîne plusieurs verbes, l’avantage devient encore plus évident :

`verbe2(verbe1(sujet,complement1),complement2)` devient `sujet %>% verbe1(complement1) %>% verbe2(complement2)`

En reprenant l'exemple précédent, sans passer par les étapes intermédiaires, le code aurait cette tête :

```
summarise (
  group_by (
    mutate (
      exercice,
      annee = year (date_prelevement)
            ),
      annee
            ),
          mesure_moyenne = mean (resultat_analyse, na.rm = T),
          mesure_total = sum (resultat_analyse, na.rm = T)
          )
```

```
## # A tibble: 26 x 3
##    annee mesure_moyenne mesure_total
##    <dbl>          <dbl>        <dbl>
##  1 1991.         0.0981         4.32
##  2 1992.         0.137          8.33
##  3 1993.         0.123          6.14
##  4 1994.         0.0684         4.72
##  5 1995.         0.0803         6.99
##  6 1996.         0.0915         6.86
##  7 1997.         0.0529         5.14
##  8 1998.         0.131         46.5
##  9 1999.         0.0547        89.7
## 10 2000.         0.118        191.
## # ... with 16 more rows
```

Avec l'utilisation du pipe (raccourci clavier CTrl + Maj + M), il devient :

```
exercice %>%
  mutate (annee = year (date_prelevement)) %>%
  group_by (annee) %>%
    summarise (mesure_moyenne = mean (resultat_analyse, na.rm = T),
               mesure_total = sum (resultat_analyse, na.rm = T))
```

```
## # A tibble: 26 x 3
##    annee mesure_moyenne mesure_total
##    <dbl>          <dbl>        <dbl>
##  1 1991.         0.0981         4.32
##  2 1992.         0.137          8.33
##  3 1993.         0.123          6.14
##  4 1994.         0.0684         4.72
##  5 1995.         0.0803         6.99
##  6 1996.         0.0915         6.86
##  7 1997.         0.0529         5.14
##  8 1998.         0.131         46.5
##  9 1999.         0.0547        89.7
## 10 2000.         0.118        191.
## # ... with 16 more rows
```

4.9 La magie des opérations groupées

L'opération **group_by()** que nous venons de voir est très utile pour les aggrégations, mais elle peut aussi servir pour créer des variables ou filtrer une table, puisque **group_by()** permet de traiter notre table en entrée comme *autant de tables séparées* par les modalités des variables de regroupement.

4.10 Exercice

Sur les données "sitadel", effectuer les opérations suivantes en utilisant l'opérateur %>% :

- les mêmes calculs que ceux réalisés sur la région 52, mais sur chacune des régions
- les aggrégations par année civile pour chacune des régions, puis leur taux d'évolution d'une année sur l'autre (exemple : (val2015-val2014)/val2014)

4.11 Exercice

Sur les données "FormationPreparationDesDonnées.RData", table "exercice" :

- calculer le taux de quantification pour chaque molécule (code_parametre), chacune des année : nombre de fois où elle a été retrouvée (code_remarque=1) sur le nombre de fois où elle a été cherchée (code_remarque = 1,2,7 ou 10)
    - créer la variable “annee”
    - créer la variable de comptage des présences pour chaque analyse (1=présent, 0=absent)
    - créer la variable de comptage des recherches pour chaque analyse (1=recherchée, 0=non recherchée)
    - pour chaque combinaison année x code_parametre, calculer le taux de quantification
- trouver pour chaque station, sur l’année 2016, le prélèvement pour lequel la concentration cumulée, toutes substances confondues, est la plus élevée (~ le prélèvement le plus pollué)
    - filtrer les concentrations quantifiées (code_remarque=1) et l’année 2016
    - sommer les concentrations (resultat_analyse) par combinaison code_station x code_prelevement
    - ne conserver que le prélèvement avec le concentration maximale

4.12 Les armes non conventionnelles de la préparation des donnéees

Nous venons de voir les verbes de manipulation d’une table les plus fréquents de dplyr. Ces verbes sont pour la plupart déclinés dans des versions encore plus puissantes, que l’on pourrait appeler conditionnelles. Dans l’univers dplyr, ces verbes sont appelés des *scoped variants*

- **xx_at()**, ou xx est l’un des verbes précédents, permet d’appliquer une opération sur un ensemble de variables définies
- **xx_if()**, ou xx est l’un des verbes précédents, permet d’appliquer une opération sur toutes les variable de la table en entrée remplissant une condition particulière
- **xx_all()**, ou xx est l’un des verbes précédents, permet d’appliquer une opération sur toutes les variables de la table en entrée

La syntaxe diffère un peu sur ces versions. On peut la globaliser ainsi : fonction(***selectiondevariables***,***funs(opérationaréalisersurcesvariables)***) La sélection de variable diffère ensuite des fonctions :

- **xx_at()**, on donne une liste de variables
- **xx_if()**, on donne une condition que doivent remplir ces variables
- **xx_all()**, on prend toutes les variables

Exemple sur l'exercice sur les données sitadel.

```
sitadel <- read_excel ("data/ROES_201702.xls", "AUT_REG") %>%
  group_by (REG) %>%
  mutate_if (is.numeric, funs (cumul12 = roll_sumr (., n = 12))) %>%
  mutate_at (vars (ends_with ("cumul12")), funs (evo = 100 * . / lag (., 12) - 100)) %>%
  mutate_at (vars (ends_with ("cumul12")), funs (part = 100 *./ log_AUT_cumul12))
```

Les verbes ayant ces variantes sont les suivants : select(), arrange(), rename(), filter(), mutate(), transmute(), group_by(), summarise().