

# Bulletin de la Dialyse à Domicile

## Initiation à la manipulation de données avec le package dplyr

Claire Della Vedova

1087 chemin de Sainte Roustagne, 04100 MANOSQUE, France

### Résumé

NDLR : Le RDPLF a pour but principal d'être une aide pour permettre aux équipes de dialyse à domicile d'évaluer leurs pratiques cliniques et également conduire des études à partir d'exports anonymisés des données qu'elles saisissent. A cette fin, depuis juin 2019, un article de formation à l'utilisation du logiciel Libre R est publié trimestriellement à chaque parution du Bulletin de la Dialyse à Domicile. Le but est de permettre à toutes les équipes de réaliser des statistiques de bases et visualiser rapidement leur données.

Le premier article de cette série d'initiation était consacré au téléchargement et à l'installation du logiciel R sur les ordinateurs Macintosh et PC : <https://doi.org/10.25796/bdd.v2i2.20513>.

Le second article était consacré à la visualisation graphiques des données statistiques avec le package Esquisse, simple d'utilisation et nécessitant peu d'apprentissage : <https://doi.org/10.25796/bdd.v2i3.21313>.

Le troisième article était consacré à la visualisation graphiques avec le Package ggplot2 :

La formation totale se fait sur 15 mois, au rythme d'un article par trimestre à chaque parution du BDD. Cela laissera largement le temps d'assimiler et tester les connaissances acquises entre chaque article. Pour ceux qui souhaiteront aller plus vite, ils pourront aller sur le blog (<https://statistique-et-logiciel-r.com/>).

Dates des prochaines parutions :

- article 6 (Septembre 2020) : la réalisation d'analyses descriptives (paramètres statistiques et graphs) sous forme de dashboard avec le package flexboard

Mots clés : Statistiques, épidémiologie, RDPLF, Logiciel libre R, cours de statistiques

<https://doi.org/10.25796/bdd.v2i4.52303>

Le quatrième article expliquait comment générer un rapport d'analyse automatiquement, et de façon dynamique, en utilisant "Rmarkdown".

Ce cinquième article est consacré à la manipulation des données avec le package dplyr : <https://doi.org/10.25796/bdd.v3i1.54523>

Comme dans les numéros précédent un fichier exemple, tiré de la base de données du RDPLF sera utilisé.

Claire Della Vedova est Ingénieure en biostatistique / data analyste, Elle utilise quotidiennement le logiciel R pour analyser des données. Elle a travaillé pendant plus de 15 ans dans les domaines de l'environnement et de la santé, et a formé de nombreux étudiants et chercheurs à l'utilisation de R. Elle anime depuis novembre 2017 le blog Statistique et Logiciel R dont le but est d'aider les débutants à mieux appréhender les méthodes statistiques classiques et à utiliser le logiciel R plus efficacement, notamment au travers de tutoriels : <https://statistique-et-logiciel-r.com/>.

## 1. Introduction

L'analyse de données nécessite bien souvent de savoir manipuler des tableaux de données, par exemple pour sélectionner des colonnes, filtrer des lignes, ou encore calculer des paramètres sur un sous-groupe d'observations.

Sous R, toutes ces manipulations peuvent facilement être réalisées, en employant le package dplyr (qui appartient, comme le package ggplot2, au super package tidyverse).

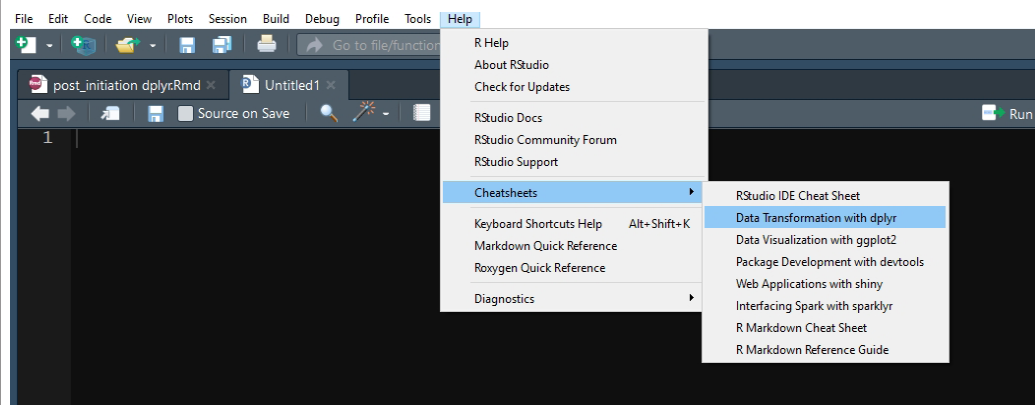
Les principales fonctions à connaître, pour débiter dans la manipulation de données sont :

- `select()` : pour sélectionner des colonnes,
- `filter()` : pour sélectionner des lignes,
- `mutate()` : pour créer des variables,
- `summarise()` : pour résumer des données, en calculant des paramètres descriptifs,
- `group_by()` : pour regrouper les données (avant de calculer un paramètre descriptif par exemple).

Ces cinq fonctions peuvent s'utiliser les unes après les autres, en utilisant le symbole `%>%` (pipe en anglais), que l'on pourrait traduire par "et puis".

Nous allons les explorer une à une à travers quelques exemples.

Le package dispose d'une cheat sheet, elle peut être téléchargée à partir de R Studio et de l'onglet Help → Cheatsheets → Data Transformation with dplyr.



# Data Transformation with dplyr : : CHEAT SHEET



dplyr functions work with pipes and expect tidy data. In tidy data:



## Summarise Cases

These apply **summary functions** to columns to create a new table of summary statistics. Summary functions take vectors as input and return one value (see back).

**summary function**

**summarise**(data, ...) Compute table of summaries. `summarise(mtcars, avg = mean(mpg))`

**count**(x, ..., wt = NULL, sort = FALSE) Count number of rows in each group defined by the variables in ... Also **tally**(). `count(iris, Species)`

**VARIATIONS**  
**summarise\_all()** - Apply funs to every column.  
**summarise\_at()** - Apply funs to specific columns.  
**summarise\_if()** - Apply funs to all cols of one type.

## Group Cases

Use **group\_by()** to create a "grouped" copy of a table. dplyr functions will manipulate each "group" separately and then combine the results.

`mtcars %>%  
 group_by(cyl) %>%  
 summarise(avg = mean(mpg))`

**group\_by**(data, ..., add = FALSE) Returns copy of table grouped by ...  
`g_iris <- group_by(iris, Species)`

**ungroup**(x, ...) Returns ungrouped copy of table.  
`ungroup(g_iris)`

## Manipulate Cases

**EXTRACT CASES**  
 Row functions return a subset of rows as a new table.

**filter**(data, ...) Extract rows that meet logical criteria. `filter(iris, Sepal.Length > 7)`

**distinct**(data, ..., keep\_all = FALSE) Remove rows with duplicate values.  
`distinct(iris, Species)`

**sample\_frac**(tbl, size = 1, replace = FALSE, weight = NULL, env = parent.frame()) Randomly select fraction of rows.  
`sample_frac(iris, 0.5, replace = TRUE)`

**sample\_n**(tbl, size, replace = FALSE, weight = NULL, env = parent.frame()) Randomly select size rows. `sample_n(iris, 10, replace = TRUE)`

**slice**(data, ...) Select rows by position.  
`slice(iris, 10:15)`

**top\_n**(x, n, wt) Select and order top n entries (by group if grouped data). `top_n(iris, 5, Sepal.Width)`

**Logical and boolean operators to use with filter()**

<	<=	is.na()	%in%		xor()
>	>=	!is.na()	!	&	

See ?base::Logic and ?Comparison for help.

**ARRANGE CASES**

**arrange**(data, ...) Order rows by values of a column or columns (low to high), use with **desc()** to order from high to low.  
`arrange(mtcars, mpg)`  
`arrange(mtcars, desc(mpg))`

**ADD CASES**

**add\_row**(data, ..., before = NULL, after = NULL) Add one or more rows to a table.  
`add_row(faithful, eruptions = 1, waiting = 1)`

## Manipulate Variables

**EXTRACT VARIABLES**  
 Column functions return a set of columns as a new vector or table.

**pull**(data, var = -1) Extract column values as a vector. Choose by name or index.  
`pull(iris, Sepal.Length)`

**select**(data, ...) Extract columns as a table. Also **select\_if()**.  
`select(iris, Sepal.Length, Species)`

**Use these helpers with select()**  
 e.g. `select(iris, starts_with("Sepal"))`

**contains**(match) **num\_range**(prefix, range) **one\_of**(...) **matches**(match) **starts\_with**(match)

**MAKE NEW VARIABLES**  
 These apply **vectorized functions** to columns. Vectorized funs take vectors as input and return vectors of the same length as output (see back).

**vectorized function**

**mutate**(data, ...) Compute new column(s).  
`mutate(mtcars, gpm = 1/mpg)`

**transmute**(data, ...) Compute new column(s), drop others.  
`transmute(mtcars, gpm = 1/mpg)`

**mutate\_all**(tbl, funs, ...) Apply funs to every column. Use with **funs()**. Also **mutate\_if()**.  
`mutate_all(faithful, funs(log(), log2()))`  
`mutate_if(iris, is.numeric, funs(log()))`

**mutate\_at**(tbl, cols, funs, ...) Apply funs to specific columns. Use with **funs()**, **vars()** and the helper functions for **select()**.  
`mutate_at(iris, vars(-Species), funs(log()))`

**add\_column**(data, ..., before = NULL, after = NULL) Add new column(s). Also **add\_count()**, **add\_tally()**. `add_column(mtcars, new = 1:32)`

**rename**(data, ...) Rename columns.  
`rename(iris, Length = Sepal.Length)`



RStudio® is a trademark of RStudio, Inc. • CC BY SA RStudio • Info@rstudio.com • 844-448-1212 • rstudio.com • Learn more with browseVignettes(package = c("dplyr", "tidbtle")) • dplyr 0.7.0 • tidbtle 1.2.0 • Updated: 2019-08

## 2. Avant de commencer

### 2.1 Organiser son travail

Avant de commencer à explorer les fonctions du package dplyr je vous recommande de créer un dossier de travail dédié à votre analyse de données, et de l'associer à un projet R.

Vous trouverez plus d'information sur cette étape dans le premier article de cette série dédiée à l'utilisation du logiciel R <https://www.bdd.rdpf.org/index.php/bdd/article/view/20513>.

Les données utilisées dans les exemples ci-dessous, sont les mêmes que celles des articles précédents, elles sont téléchargeables au format csv, à cette adresse : <https://www.rdpf.org/exempleR/FichierExempleStat.csv>

Une fois téléchargé, placez le fichier csv dans le dossier "data" de votre projet R, puis utilisez la commande suivante pour les importer dans R :

```
mydata <- read.csv2(«data/FichierExempleStat.csv»)
```

## 2.2 Charger le package dplyr

Pour charger le package dplyr, vous pouvez employer l'une des deux commandes suivantes :

```
library(dplyr)  
library(tidyverse)
```

## 2.2 Gérer les éventuels conflits de packages

Un conflit de package peut survenir lorsque deux packages contiennent des fonctions différentes mais portant le même nom. C'est le cas, par exemple, de la fonction `select()` du package `dplyr`, et de la fonction `select()` du package `MASS`. En cas de conflits de packages, R ne sait pas quelle fonction choisir. Au mieux, il vous renverra une erreur dont le message est rarement intelligible, au pire, il ne fera rien : il n'exécutera pas la commande et vous ne le signalera pas !

Pour éviter cette situation, je vous recommande de spécifier, en amont de vos scripts, que vous souhaitez utiliser les fonctions `select()` et `filter()` du package `dplyr`, en utilisant le package `conflicted` et sa fonction `conflict_prefer()`, comme ceci :

```
library(conflicted)  
conflict_prefer(«select», «dplyr»)  
conflict_prefer(«filter», «dplyr»)
```

Remarque : à ma connaissance les fonctions `mutate()`, `summarise()` et `group_by` ne sont pas (actuellement) concernées par un éventuel conflit de package.

## 3. Sélectionner des variables

Pour créer un sous-jeu de données en ne sélectionnant que certaines variables, nous employons la fonction `select()`.

### 3.1 Par le nom

Si nous souhaitons faire un nouveau fichier de données, avec uniquement les variables `PAYS`, `sexe`, `Charlson`, `Taille` et `Poids`, il suffit de les indiquer en argument de la fonction `select()` :

```
mydata2 <- mydata %>%  
  select(PAYS, sexe, Charlson, Taille, Poids)  
# vérification des variables présentes dans le jeu créé  
head(mydata2)  
  
##           PAYS  sexe Charlson Taille Poids  
## 1     FRANCE    F         3    132  30.0  
## 2     TUNISIE    F         3    131  32.5  
## 3 LUXEMBOURG    M         5    140  33.0  
## 4     FRANCE    F         6    142  35.0  
## 5     FRANCE    F         6    150  35.0  
## 6     FRANCE    F         4    145  37.0
```

### 3.2 Désélectionner le nom

Il est également possible de désélectionner des variables, en employant le symbole “-” devant celles-ci :

```
mydata3 <- mydata %>%  
  select(-code.post, -Age.1ere.DP,-Charlson_modif, -Type.de.DP)  
  
# verification  
head(mydata3)  
  
##          PAYS  sexe Charlson Taille Poids  
## 1    FRANCE    F         3    132   30.0  
## 2    TUNISIE    F         3    131   32.5  
## 3 LUXEMBOURG    M         5    140   33.0  
## 4    FRANCE    F         6    142   35.0  
## 5    FRANCE    F         6    150   35.0  
## 6    FRANCE    F         4    145   37.0
```

### 3.3 Sélection par nom partiel

Il est encore possible de sélectionner des variables en utilisant un motif de lettres contenu dans le nom des variables. Par exemple, si nous souhaitons sélectionner toutes les variables contenant le motif “Charlson” :

```
mydata4 <- mydata %>%  
  select(contains(«Charlson»))  
  
# verification  
head(mydata4)  
  
##   Charlson Charlson_modif  
## 1         3                3  
## 2         3                2  
## 3         5                5  
## 4         6                5  
## 5         6                5  
## 6         4                3
```

Nous pouvons aussi employer les fonctions `starts_with()` et `ends_with()`. Par exemple, pour sélectionner les variables se terminant par le motif de lettres “DP” :

```
mydata5 <- mydata %>%  
  select(ends_with(«DP»))  
# vérification  
head(mydata5)  
  
##   Age.1ere.DP      Type.de.DP  
## 1       20.13      DPCA  
## 2       50.74      DPCA  
## 3       18.86 DPA quotidienne  
## 4       50.86 DPA quotidienne  
## 5       54.07      DPCA  
## 6       53.89      DPCA
```

### 3.4 Sélection par classe de variable

En complément de la fonction `select()`, le package `dplyr` dispose d'une fonction `select_if()` qui permet, par exemple de sélectionner des variables en fonction de leur classe (numérique ou factoriel).

Ainsi, si nous souhaitons, sélectionner toutes les variables numérique d'un data frame, nous pouvons employer la commande suivante :

```
mydata_num <- mydata %>%  
  select_if(is.numeric)  
  
head(mydata_num)  
  
## Age.1ere.DP Charlson Charlson_modif Taille Poids  
## 1      20.13      3           3    132  30.0  
## 2      50.74      3           2    131  32.5  
## 3      18.86      5           5    140  33.0  
## 4      50.86      6           5    142  35.0  
## 5      54.07      6           5    150  35.0  
## 6      53.89      4           3    145  37.0
```

De même pour les variables catégorielles :

```
mydata_fact <- mydata %>%  
  select_if(is.factor)  
  
head(mydata_fact)  
  
## code.post      PAYS  sexe      Type.de.DP  
## 1    56100      FRANCE  F        DPCA  
## 2     1006      TUNISIE  F        DPCA  
## 3     2540 LUXEMBOURG  M  DPA quotidienne  
## 4     3200      FRANCE  F  DPA quotidienne  
## 5     31603      FRANCE  F        DPCA  
## 6     74374      FRANCE  F        DPCA
```

## 4. Filtrer des lignes

La fonction `filter()` permet de filtrer des lignes, notamment en fonction:

- d'une variable numérique
- d'une variable catégorielle (facteur)

### 4.1 En fonction d'une variable numérique

Par exemple, si nous souhaitons conserver uniquement les patients dont l'âge de la première DP est supérieur à 65 ans :

```
sup65 <-mydata %>%  
  filter(Age.1ere.DP>65)  
  
head(sup65)  
##   code.post   PAYS  sexe Age.1ere.DP Charlson Charlson_modif   Type.de.DP  
## 1   25000 FRANCE   F     86.50      7          3          DPCA  
## 2   83100 FRANCE   F     83.75      7          3          DPCA  
## 3   72037 FRANCE   F     71.58      7          4 DPA quotidienne  
## 4   97448 FRANCE   F     76.88      5          2 DPA quotidienne  
## 5    3200 FRANCE   F     69.25      4          2          DPCA  
## 6   54504 FRANCE   F     86.53      7          3          DPCA  
##   Taille Poids  
## 1   147    38  
## 2   139    39  
## 3   157    39  
## 4   145    40  
## 5   152    40  
## 6   155    40
```

Nous pouvons également employer un intervalle de valeurs, comme ceci :

```
entre55_65 <- mydata %>%  
  filter(between(Age.1ere.DP, 55, 65))  
  
head(entre55_65 )  
##   code.post   PAYS  sexe Age.1ere.DP Charlson Charlson_modif   Type.de.DP  
## 1   59037 FRANCE   F     56.06      3          2          DPCA  
## 2    1200 BELGIQUE F     63.29      8          6 DPA quotidienne  
## 3    4500 BELGIQUE F     57.30      4          3 DPA quotidienne  
## 4   44402 FRANCE   F     58.72      6          5 DPA quotidienne  
## 5   31603 FRANCE   F     61.35      4          2          DPCA  
## 6   29000 FRANCE   F     63.62      6          4          DPCA  
##   Taille Poids  
## 1   135  42.0  
## 2   157  43.0  
## 3   167  43.0  
## 4   162  44.0  
## 5   145  44.5  
## 6   157  45.0
```

#### 4.2 En fonction d'une variable catégorielle

Si nous souhaitons réaliser un sous-jeu de données, ne contenant que les observations relatives à la France :

```
df_Fr <- mydata %>%
  filter(PAYS==»FRANCE»)

head(df_Fr)

## code.post PAYS sexe Age.1ere.DP Charlson Charlson_modif Type.de.DP
## 1 56100 FRANCE F 20.13 3 3 DPCA
## 2 3200 FRANCE F 50.86 6 5 DPA quotidienne
## 3 31603 FRANCE F 54.07 6 5 DPCA
## 4 74374 FRANCE F 53.89 4 3 DPCA
## 5 25000 FRANCE F 86.50 7 3 DPCA
## 6 83100 FRANCE F 83.75 7 3 DPCA
## Taille Poids
## 1 132 30
## 2 142 35
## 3 150 35
## 4 145 37
## 5 147 38
## 6 139 39
```

Pour sélectionner plusieurs pays, il est nécessaire de créer le vecteur de pays, et d'employer le symbole %in%, comme ceci :

```
df_Su_Be_Lu <- mydata %>%
  filter(PAYS %in% c(«SUISSE», «BELGIQUE», «LUXEMBOURG»))

head(df_Su_Be_Lu)

## code.post PAYS sexe Age.1ere.DP Charlson Charlson_modif Type.de.DP
## 1 2540 LUXEMBOURG M 18.86 5 5 DPA quotidienne
## 2 1211 SUISSE F 28.61 2 2 DPCA
## 3 1200 BELGIQUE F 17.38 3 3 DPA quotidienne
## 4 1011 SUISSE F 47.00 3 3 DPA quotidienne
## 5 1200 BELGIQUE F 63.29 8 6 DPA quotidienne
## 6 1211 SUISSE F 77.76 6 3 DPA quotidienne
## Taille Poids
## 1 140 33
## 2 154 41
## 3 147 42
## 4 150 43
## 5 157 43
## 6 157 43
```

#### 4.3 En fonction d'un double critère.

Par exemple si nous souhaitons filtrer les femmes de plus de 170 cm



```
Fsup170 <- mydata %>%
  filter(sexe==>F) %>%
  filter(Taille>170)

head(Fsup170)
```

##	code.post	PAYS	sexe	Age.1ere.DP	Charlson	Charlson_modif	Type.de.DP
## 1	59385	FRANCE	F	64.90	4	2	DPA quotidienne
## 2	75020	FRANCE	F	20.73	2	2	DPA quotidienne
## 3	83100	FRANCE	F	58.74	15	14	DPA quotidienne
## 4	35400	FRANCE	F	77.03	7	4	DPCA
## 5	62408	FRANCE	F	46.60	3	3	DPCA
## 6	5000	TUNISIE	F	21.63	2	2	DPCA

##	Taille	Poids
## 1	175	45
## 2	172	48
## 3	172	52
## 4	172	54
## 5	171	56
## 6	171	58

A noter que nous pouvons également employer la syntaxe suivante :

```
Fsup170bis <- mydata %>%
  filter(sexe==>F & Taille>170 )
```

#### 4.4 En fonction d'une position

Pour filtrer des lignes en fonction de leur position (index), nous pouvons employer la fonction slice().

Par exemple, pour ne conserver que les lignes 5 à 10 du jeu de données :

```
mydata_5to10 <- mydata %>%
  slice(5:10)

mydata_5to10
```

##	code.post	PAYS	sexe	Age.1ere.DP	Charlson	Charlson_modif	Type.de.DP
## 1	31603	FRANCE	F	54.07	6	5	DPCA
## 2	74374	FRANCE	F	53.89	4	3	DPCA
## 3	20420	MAROC	F	22.25	2	2	DPCA
## 4	25000	FRANCE	F	86.50	7	3	DPCA
## 5	10000	MAROC	F	18.22	2	2	DPA quotidienne
## 6	83100	FRANCE	F	83.75	7	3	DPCA

##	Taille	Poids
## 1	150	35
## 2	145	37
## 3	148	37
## 4	147	38
## 5	136	39
## 6	139	39

Pour filtrer des lignes, non contiguës, en fonction de leur position, il est nécessaire d'employer la concaténation, comme dans l'exemple ci-dessous :

```
mydata_multiple <- mydata %>%
  slice(c(2:4),59,c(111:113))
mydata_multiple

## code.post      PAYS  sexe Age.1ere.DP Charlson Charlson_modif      Type.de.DP
## 1      1006      TUNISIE  F      50.74      3      2      DPCA
## 2      2540 LUXEMBOURG  M      18.86      5      5 DPA quotidienne
## 3      3200      FRANCE   F      50.86      6      5 DPA quotidienne
## 4      10000     MAROC   M      33.62      2      2      DPCA
## 5      1020      BELGIQUE F      28.62      8      8 DPA quotidienne
## 6      75014     FRANCE  M      17.85      2      2      DPCA
## 7      46005     FRANCE  F      46.52      2      2 DPA quotidienne
## Taille Poids
## 1      131      32.5
## 2      140      33.0
## 3      142      35.0
## 4      159      44.0
## 5      165      46.0
## 6      165      46.0
## 7      168      46.0
```

#### 4.5 Liste des principaux opérateurs

Les principaux opérateurs qui peuvent être employés avec la fonction filter() sont résumés ci-dessous :

Opérateur	Signification
<	strictement inférieur
<=	inférieur ou égal
>	strictement supérieur
>=	supérieur ou égal
==	exactement égal
!=	différent
x y	x et/ou y
xor(x,y)	x ou y (intersection exclue)
x & y	x et y
is.na(x)	test si x est NA
%in%	compris dans (un vecteur de choix)

#### 5. Créer une nouvelle variable

La fonction mutate() permet de créer une nouvelle variable dans un data.frame.

## 5.1 Une variable numérique

Par exemple, si nous souhaitons créer une nouvelle variable taille, exprimée cette fois en mètres :

```
mydata <- mydata %>%  
  mutate(Taille_m=Taille/100)  
  
head(mydata)
```

##	code.post	PAYS	sexe	Age.1ere.DP	Charlson	Charlson_modif	Type.de.DP
## 1	56100	FRANCE	F	20.13	3	3	DPCA
## 2	1006	TUNISIE	F	50.74	3	2	DPCA
## 3	2540	LUXEMBOURG	M	18.86	5	5	DPA quotidienne
## 4	3200	FRANCE	F	50.86	6	5	DPA quotidienne
## 5	31603	FRANCE	F	54.07	6	5	DPCA
## 6	74374	FRANCE	F	53.89	4	3	DPCA

##	Taille	Poids	Taille_m
## 1	132	30.0	1.32
## 2	131	32.5	1.31
## 3	140	33.0	1.40
## 4	142	35.0	1.42
## 5	150	35.0	1.50
## 6	145	37.0	1.45

Si nous souhaitons, à présent, créer une variable IMC à partir du poids et de la taille des patients, en divisant le poids par la taille (exprimée en mètre) :

```
mydata <- mydata %>%  
  mutate(IMC = Poids/(Taille_m))  
  
head(mydata)
```

##	code.post	PAYS	sexe	Age.1ere.DP	Charlson	Charlson_modif	Type.de.DP
## 1	56100	FRANCE	F	20.13	3	3	DPCA
## 2	1006	TUNISIE	F	50.74	3	2	DPCA
## 3	2540	LUXEMBOURG	M	18.86	5	5	DPA quotidienne
## 4	3200	FRANCE	F	50.86	6	5	DPA quotidienne
## 5	31603	FRANCE	F	54.07	6	5	DPCA
## 6	74374	FRANCE	F	53.89	4	3	DPCA

##	Taille	Poids	Taille_m	IMC
## 1	132	30.0	1.32	22.72727
## 2	131	32.5	1.31	24.80916
## 3	140	33.0	1.40	23.57143
## 4	142	35.0	1.42	24.64789
## 5	150	35.0	1.50	23.33333
## 6	145	37.0	1.45	25.51724

## 5.2 Une variable catégorielle

En couplant la fonction ifelse(), à la fonction mutate(), il est par exemple possible de catégoriser une variable numérique. Cette fonction ifelse() prend trois arguments :

- la condition
- la valeur que prend la variable si la condition est satisfaite
- la valeur que prend la variable sinon.

Par exemple si nous souhaitons créer une variable “Stature” qui prend pour valeur “petite” si la taille est inférieure ou égale 166 cm, et “grande” sinon, nous pouvons employer les commandes suivantes :

```
mydata <- mydata %>%
  mutate(Stature = ifelse(Taille<=166, «Petite», «Grande»))

# visualisation du résultat
sample_n(mydata,10)
## code.post PAYS sexe Age.1ere.DP Charlson Charlson_modif Type.de.DP
## 1 38701 FRANCE M 71.59 9 6 DPA quotidienne
## 2 11100 FRANCE F 50.75 4 3 DPCA
## 3 13698 FRANCE M 46.96 2 2 DPCA
## 4 84902 FRANCE M 63.92 5 3 DPA quotidienne
## 5 62408 FRANCE M 81.37 2 -2 DPCA
## 6 54504 FRANCE M 78.86 6 3 DPCA
## 7 14033 FRANCE F 68.18 5 3 DPCA
## 8 4000 TUNISIE F 23.99 2 2 DPA quotidienne
## 9 94275 FRANCE F 63.95 4 2 DPCA
## 10 4000 BELGIQUE M 51.81 3 2 DPA quotidienne
## Taille Poids Taille_m IMC Stature
## 1 175 86 1.75 49.14286 Grande
## 2 160 82 1.60 51.25000 Petite
## 3 170 87 1.70 51.17647 Grande
## 4 176 82 1.76 46.59091 Grande
## 5 172 87 1.72 50.58140 Grande
## 6 180 53 1.80 29.44444 Grande
## 7 148 53 1.48 35.81081 Petite
## 8 155 52 1.55 33.54839 Petite
## 9 165 106 1.65 64.24242 Petite
## 10 175 88 1.75 50.28571 Grande
```

### 5.3 Transformer une variable

La fonction mutate peut également permettre de modifier une variable existante, plutôt que d’en créer une autre. Pour cela, ce sont surtout les fonctions mutate\_if() et mutate\_at() qui sont employées.

Par exemple, pour centrer et réduire les variables numériques de notre jeu de données, nous pouvons coupler les fonctions mutate\_if() et scale() qui permet le centrage réduction, comme ceci :

```
mydata_num_cr <- mydata %>%
  mutate_if(is.numeric, scale)
head(mydata_num_cr)
## code.post PAYS sexe Age.1ere.DP Charlson Charlson_modif
## 1 56100 FRANCE F -2.7237972 -1.12386300 -0.3901693
## 2 1006 TUNISIE F -0.8685743 -1.12386300 -0.9416180
## 3 2540 LUXEMBOURG M -2.8007698 -0.33322084 0.7127281
## 4 3200 FRANCE F -0.8613013 0.06210025 0.7127281
## 5 31603 FRANCE F -0.6667484 0.06210025 0.7127281
## 6 74374 FRANCE F -0.6776579 -0.72854192 -0.3901693
## Type.de.DP Taille Poids Taille_m IMC Stature
## 1 DPCA -3.490473 -2.724799 -3.490473 -2.479468 Petite
## 2 DPCA -3.592773 -2.565011 -3.592773 -2.232003 Petite
## 3 DPA quotidienne -2.672075 -2.533053 -2.672075 -2.379127 Petite
## 4 DPA quotidienne -2.467476 -2.405222 -2.467476 -2.251172 Petite
## 5 DPCA -1.649077 -2.405222 -1.649077 -2.407428 Petite
## 6 DPCA -2.160576 -2.277392 -2.160576 -2.147836 Petite
```

La fonction `mutate_at()` est employé pour préciser les variables numériques que nous souhaitons centrer réduire:

```
mydata_num_cr2 <- mydata %>%
  mutate_at(«Age.1ere.DP», scale)
head(mydata_num_cr2)
```

##	code.post	PAYS	sexe	Age.1ere.DP	Charlson	Charlson_modif	Type.de.DP
## 1	56100	FRANCE	F	-2.7237972	3	3	DPCA
## 2	1006	TUNISIE	F	-0.8685743	3	2	DPCA
## 3	2540	LUXEMBOURG	M	-2.8007698	5	5	DPA quotidienne
## 4	3200	FRANCE	F	-0.8613013	6	5	DPA quotidienne
## 5	31603	FRANCE	F	-0.6667484	6	5	DPCA
## 6	74374	FRANCE	F	-0.6776579	4	3	DPCA

##	Taille	Poids	Taille_m	IMC	Stature
## 1	132	30.0	1.32	22.72727	Petite
## 2	131	32.5	1.31	24.80916	Petite
## 3	140	33.0	1.40	23.57143	Petite
## 4	142	35.0	1.42	24.64789	Petite
## 5	150	35.0	1.50	23.33333	Petite
## 6	145	37.0	1.45	25.51724	Petite

#### 5.4 Liste des fonctions utilisables

Le package `dplyr` propose de nombreuses fonctions très utiles qui peuvent être employées dans une fonction `mutate()` pour transformer ou créer des variables :

### Vector Functions

TO USE WITH MUTATE ()

**mutate()** and **transmute()** apply vectorized functions to columns to create new columns. Vectorized functions take vectors as input and return vectors of the same length as output.

vectorized function

OFFSETS

`dplyr::lag()` - Offset elements by 1  
`dplyr::lead()` - Offset elements by -1

CUMULATIVE AGGREGATES

`dplyr::cumall()` - Cumulative all()  
`dplyr::cumany()` - Cumulative any()  
`dplyr::cummax()` - Cumulative max()  
`dplyr::cummean()` - Cumulative mean()  
`dplyr::cummin()` - Cumulative min()  
`dplyr::cumprod()` - Cumulative prod()  
`dplyr::cumsum()` - Cumulative sum()

RANKINGS

`dplyr::cume_dist()` - Proportion of all values <=

`dplyr::dense_rank()` - rank w ties = min, no gaps  
`dplyr::min_rank()` - rank with ties = min  
`dplyr::ntile()` - bins into n bins  
`dplyr::percent_rank()` - min\_rank scaled to [0,1]  
`dplyr::row_number()` - rank with ties = "first"

MATH

`+`, `-`, `*`, `/`, `^`, `%/%`, `%%` - arithmetic ops  
`log()`, `log2()`, `log10()` - logs  
`<`, `<=`, `>`, `>=`, `!=`, `==` - logical comparisons  
`dplyr::between()` - x >= left & x <= right  
`dplyr::near()` - safe == for floating point numbers

MISC

`dplyr::case_when()` - multi-case if\_else()

```
iris %>% mutate(Species = case_when(
  Species == "versicolor" ~ "versi",
  Species == "virginica" ~ "virgi",
  TRUE ~ Species))
```

`dplyr::coalesce()` - first non-NA values by element across a set of vectors  
`dplyr::if_else()` - element-wise if() + else()  
`dplyr::na_if()` - replace specific values with NA  
`pmax()` - element-wise max()  
`pmin()` - element-wise min()  
`dplyr::recode()` - Vectorized switch()  
`dplyr::recode_factor()` - Vectorized switch() for factors

## 6 Calculer un paramètre

### 6.1 De manière globale

La fonction summarise() permet de calculer un paramètre descriptif, comme une moyenne, la valeur minimum, la valeur maximum etc... Voici un exemple de commande qui peut être employée pour calculer la moyenne, la médiane, le min, le max et l'écart type de la Taille :

```
mydata %>%
  summarise(moy=mean(Taille, na.rm=TRUE),
            med=median(Taille, na.rm=TRUE),
            min=min(Taille, na.rm=TRUE),
            max=max(Taille, na.rm=TRUE),
            ecart_type=sd(Taille, na.rm=TRUE))

##           moy med min max ecart_type
## 1 166.1201 166 100 196 9.775194
```

### 6.2 Par sous-groupes

Un des grands intérêts de cette fonction summarise(), réside dans son couplage avec la fonction group\_by(). La combinaison de ces fonctions permet alors d'obtenir l'estimation de paramètres pour différents sous-groupes de données.

Ainsi, si nous souhaitons obtenir les paramètres précédents, en fonction des pays, nous pouvons employer en amont la syntaxe group\_by(PAYS)%>% , comme ceci :

```
mydata %>%
  group_by(PAYS) %>%
  summarise(Nb=n(),
            moy=mean(Taille, na.rm=TRUE),
            med=median(Taille, na.rm=TRUE),
            min=min(Taille, na.rm=TRUE),
            max=max(Taille, na.rm=TRUE),
            ecart_type=sd(Taille, na.rm=TRUE))

## # A tibble: 6 x 7
##   PAYS           Nb moy   med   min  max ecart_type
##   <fct>         <int> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1 BELGIQUE       244 167. 168  120  194  9.24
## 2 FRANCE       2639 166. 166  100  196  9.74
## 3 LUXEMBOURG    10 171. 174  140  195  16.5
## 4 MAROC         99 164. 165  126  186  10.3
## 5 SUISSE        33 167. 168  150  185  10.3
## 6 TUNISIE       107 166. 168  131  190  10.1
```

Nous pouvons utiliser plusieurs variables dans la fonction group\_by(), par exemple le pays et le sexe, comme ceci :

```
mydata %>%
  group_by(PAYS, sexe) %>%
  summarise(Nb=n(),
            moy=mean(Taille, na.rm=TRUE),
            med=median(Taille, na.rm=TRUE),
            min=min(Taille, na.rm=TRUE),
            max=max(Taille, na.rm=TRUE),
            ecart_type=sd(Taille, na.rm=TRUE))

## # A tibble: 12 x 8
## # Groups: PAYS [6]
##   PAYS      sexe   Nb   moy   med   min   max ecart_type
##   <fct>    <fct> <int> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1 BELGIQUE  F     91  161.  162   140   177    6.90
## 2 BELGIQUE  M    153  171.  170   120   194    8.44
## 3 FRANCE    F   1056  159.  159   120   186    7.48
## 4 FRANCE    M   1583  171.  170   100   196    8.12
## 5 LUXEMBOURG F     2  158.  158.  151   164    9.19
## 6 LUXEMBOURG M     8  175.  176   140   195   16.4
## 7 MAROC     F    55  159.  160   126   178    9.07
## 8 MAROC     M    44  171.  170   154   186    7.87
## 9 SUISSE    F    13  158.  157   150   172    6.63
## 10 SUISSE   M    20  173.  174.  155   185    7.54
## 11 TUNISIE  F    48  161.  161   131   184    9.10
## 12 TUNISIE  M    59  171.  172   145   190    8.35
```

Et bien entendu, si nous le souhaitons, nous pouvons stocker ces paramètres dans une nouvelle table de données, qui pourra ensuite, par exemple, être utilisée pour créer des visualisations. Pour cela, il nous suffit d'utiliser une assignation (avec une flèche, en amont) :

```
new_table <- mydata %>%
  group_by(PAYS, sexe) %>%
  summarise(Nb=n(),
            moy=mean(Taille, na.rm=TRUE),
            med=median(Taille, na.rm=TRUE),
            min=min(Taille, na.rm=TRUE),
            max=max(Taille, na.rm=TRUE),
            ecart_type=sd(Taille, na.rm=TRUE))

head(new_table)

## # A tibble: 6 x 8
## # Groups: PAYS [3]
##   PAYS      sexe   Nb   moy   med   min   max ecart_type
##   <fct>    <fct> <int> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1 BELGIQUE  F     91  161.  162   140   177    6.90
## 2 BELGIQUE  M    153  171.  170   120   194    8.44
## 3 FRANCE    F   1056  159.  159   120   186    7.48
## 4 FRANCE    M   1583  171.  170   100   196    8.12
## 5 LUXEMBOURG F     2  158.  158.  151   164    9.19
## 6 LUXEMBOURG M     8  175.  176   140   195   16.4
```

### 6.3 Liste des fonctions utilisables dans summarise():

Voici une liste de fonctions qui peuvent être utilisée dans la fonction summarise() pour résumer de l'information :

## Summary Functions

### TO USE WITH SUMMARISE ()

**summarise()** applies summary functions to columns to create a new table. Summary functions take vectors as input and return single values as output.

summary function

### COUNTS

**dplyr::n()** - number of values/rows  
**dplyr::n\_distinct()** - # of uniques  
**sum(!is.na())** - # of non-NA's

### LOCATION

**mean()** - mean, also **mean(!is.na())**  
**median()** - median

### LOGICALS

**mean()** - Proportion of TRUE's  
**sum()** - # of TRUE's

### POSITION/ORDER

**dplyr::first()** - first value  
**dplyr::last()** - last value  
**dplyr::nth()** - value in nth location of vector

### RANK

**quantile()** - nth quantile  
**min()** - minimum value  
**max()** - maximum value

### SPREAD

**IQR()** - Inter-Quartile Range  
**mad()** - median absolute deviation  
**sd()** - standard deviation  
**var()** - variance

## 7. Conclusion

J'espère que cet article vous aura convaincu, à la fois sur les capacités du package dplyr à manipuler les données, mais aussi sur sa facilité d'utilisation.

Notez, néanmoins, que les fonctions select(), filter(), mutate(), et summarise() offrent de nombreuses autres possibilités, qu'il est impossible de détailler dans un seul article.

Si vous souhaitez explorer ces possibilités, vous pouvez consulter le blog de suzan Baert, notamment les pages :

- Data Wrangling Part 1: Basic to Advanced Ways to Select Columns <https://suzan.rbind.io/2018/01/dplyr-tutorial-1/>
- Data Wrangling Part 2: Transforming your columns into the right shape <https://suzan.rbind.io/2018/01/dplyr-tutorial-2/>



[rbind.io/2018/02/dplyr-tutorial-2/](https://rbind.io/2018/02/dplyr-tutorial-2/)

- Data Wrangling Part 3: Basic and more advanced ways to filter rows <https://suzan.rbind.io/2018/02/dplyr-tutorial-3/>
- Data Wrangling Part 4: Summarizing and slicing your data <https://suzan.rbind.io/2018/04/dplyr-tutorial-4/>

*Open Access* : cet article est sous licence Creative commons CC BY 4.0 : <https://creativecommons.org/licenses/by/4.0/deed.fr>

Vous êtes autorisé à :

*Partager* — copier, distribuer et communiquer le matériel par tous moyens et sous tous formats

*Adapter* — remixer, transformer et créer à partir du matériel pour toute utilisation, y compris commerciale.

Cette licence est acceptable pour des œuvres culturelles libres.

L'Offrant ne peut retirer les autorisations concédées par la licence tant que vous appliquez les termes de cette licence. selon les conditions suivantes :

*Attribution* — Vous devez créditer l'Œuvre, intégrer un lien vers la licence et indiquer si des modifications ont été effectuées à l'Œuvre. Vous devez indiquer ces informations par tous les moyens raisonnables, sans toutefois suggérer que l'Offrant vous soutient ou soutient la façon dont vous avez utilisé son Œuvre. <http://creativecommons.org/licenses/by/4.0/>.