R pour ma grand-mère

Vincent Bonhomme

2024-04-13

Table of contents

Préface

À propos de l'ouvrage

La vocation de cet ouvrage est de fournir un point de départ rapide mais solide aux principaux usages de R, un environnement complet et open-source pour l'analyse de données.

L'auteur a tenté d'écrire la ressource qu'il aurait aimé avoir à ses débuts, dans l'esprit de ses deux références chéries : *R pour les débutants* d'Emmanuel Paradis et *R for data science* d'Hadley Wickham. Une sélection de références plus complètes et/ou plus spécifiques, sont listées. Un rapide tour d'internet vous permettra d'en trouver davantage.

Ce document sert de compagnon à une formation pour des absolu · e · s débutant · e · s sur une durée présentielle de 3 jours. Son intention est d'autonomiser ces utilisateur · trice · s tout en espérant être utiles aux non-néophytes voire aux plus assidu · e · s.

À propos de l'auteur

Je suis biologiste de l'évolution de formation et aujourd'hui consultant, formateur et chercheur associé à l'UMR ISEM à Montpellier. J'utilise R quotidiennement depuis 2006 et je suis l'auteur de nombreux packages dont Momocs, pour l'analyse de forme.

Je suis disponible pour de la consultance et des formations sur à peu près tous les "niveaux" d'utilisation de R, depuis le tout début jusqu'au packaging et la contruction d'interface conviviales ala shiny.

En *side project*, j'ai confondé Cévennette qui conçoit et fabrique des toilettes sèches à séparation, sans odeur et sans copeaux, pour les particuliers, les événements et les collectivités.

Si vous avez envie de faire pipi ou si vous cherchez un formateur ou un consultant en R, voici mes contacts :

- www.vincentbonhomme.fr
- bonhomme.vincent@gmail.com

Code source

- Si vous le lisez en pdf, il est disponible en ligne ici : https://vbonhomme.github.io/R-pour-ma-grand-mere.
- Si vous le lisez en ligne, il est disponible en pdf ici
- Ce document Quarto est écrit en Rmarkdown et hébergé par Github: [https://github.com/vbonhomme/R-pour-ma-grand-mere]
- Si vous êtes à l'aise sur Github, vous pouvez directement modifier le contenu que j'approuverai (ou non), pour corriger une erreur, une coquille, etc. Si vous souhaitez simplement suggérer une modification ou signaler un problème, vous pouvez très simplement ouvrir un "ticket" dans l'onglet "Issues".

Licence

Conventions

Les fonctions et packages seront mentionnées dans cette `police`, avec des parenthèses pour les noms de fonctions, sans parenthèses pour les packages : `ggplot()` et `ggplot2`.

Le code en ligne utilisera `une police de ce type`; les blocs ressemblent à celui ci :

```
# this is a comment
3+4
```

```
[1] 7
```

Le résultat, textuel ou graphique, s'affiche à proximité et vous pouvez copier-coller ce code dans R (ou cliquer sur le bouton en haut à droite du cadre) et obtenir le même résultat sur votre machine. Tout ce qui suit un `#` est un commentaire - lisez-les ! - et est ignoré par R.

Parfois, une maxime de sagesse populaiRe est insérée dans un bloc comme ci-dessous. Promis, elles ne réfèrent pas toutes au sous-vêtements.

> Les statistiques sont comme les petites culottes : elles montrent le superflu et cachent l'essentiel.

---

1 Installer un environnement R

Je préconise l'installation conjoint de R et de RStudio son IDE (environnement de travail intégré). Vous pouvez faire sans RStudio mais je vous le déconseille.

1.1 R

L'installation de R se passe généralement sans encombres, en quelques minutes.

Le plus simple est de choisir votre système d'exploitation depuis la page officielle :

https://cran.rstudio.com/.

Pour Windows et Mac, il suffit de télécharger et d'installer l'archive. Pour Mac, comme indiqué sur la page en lien, il vous faudra probablement également installer XQuartz. Sous Linux, vous serez guidés pour utiliser votre gestionnaire de paquets préféré.

1.2 RStudio

Pour installer RStudio, version bureau, récupérez l'installeur depuis sa page d'accueil et laissez vous porter :

https://posit.co/download/rstudio-desktop/

1.3 Vérifier votre installation

Une fois que vous avez installé R puis RStudio, ouvrez RStudio.

Dans la fenêtre "console", là où vous voyez une invite de commande `>` tapez 1+1 puis `<Entrée>`. Voilà, vous avez une installation de R+RStudio qui fonctionne.

1.4 Installer les packages

Les besoins de ce document se limitent aux packages du tidyverse que vous pouvez installer en tapant à la console la commande suivante :

```
install.packages("tidyverse")
```

2 Une introduction en 3 minutes

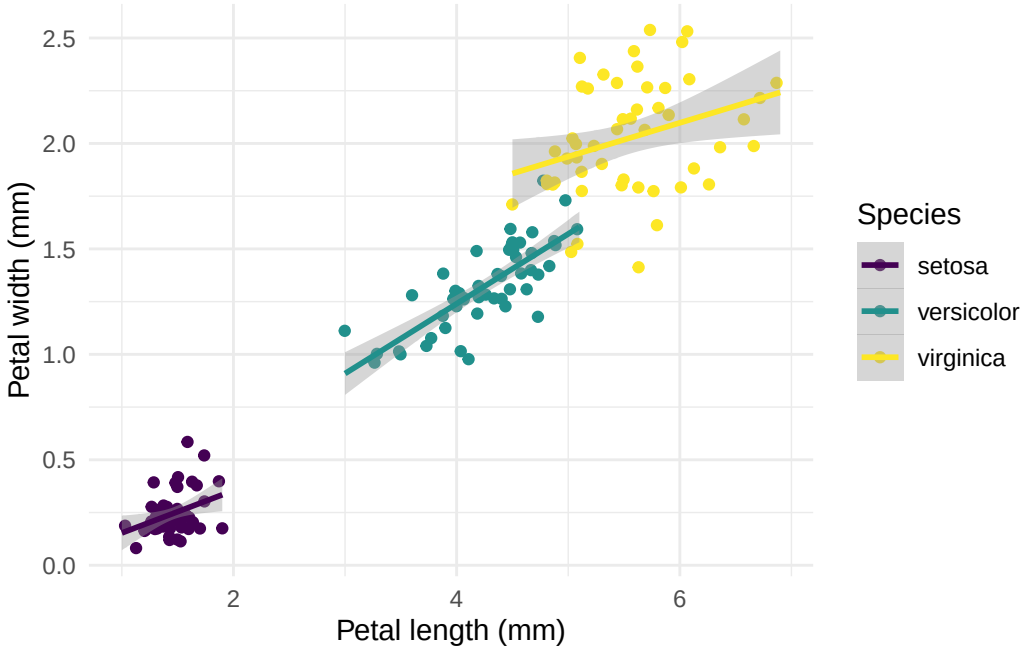
Le bloc de code ci-dessous combine tous les aspects que nous allons aborder dans ce document : utilisation de packages, du pipe %>%, préparation de données issues d'un tableau, production et customisation d'un graphe, création d'une fonction, calculs combinés sur listes de données, etc.

```
# dependencies
library(tidyverse)

# data tidying
iris2 <- iris %>%
  as_tibble() %>%
  select(pl=Petal.Length, pw=Petal.Width, sp=Species)
iris2
```

```
# A tibble: 150 x 3
      pl    pw sp
   <dbl> <dbl> <fct>
 1   1.4   0.2 setosa
 2   1.4   0.2 setosa
 3   1.3   0.2 setosa
 4   1.5   0.2 setosa
 5   1.4   0.2 setosa
 6   1.7   0.4 setosa
 7   1.4   0.3 setosa
 8   1.5   0.2 setosa
 9   1.4   0.2 setosa
10   1.5   0.1 setosa
# i 140 more rows
```

```
# a quick graph
ggplot(iris2) +
  aes(pl, pw, col=sp) +
  geom_jitter() +
  geom_smooth(method="lm", formula="y~x") +
```

```
  scale_color_viridis_d() +
  guides(colour=guide_legend("Species")) +
  xlab("Petal length (mm)") + ylab("Petal width (mm)") +
  theme_minimal()
```




```
# a little helper function to get adjusted R2 of a linear model
lm_adj_r2 <- function(x) summary(lm(pl~pw, data=x))$adj.r.squared

# group-wise statistics
iris2 %>%
  nest(data=c(pl, pw)) %>%
  mutate(adj_r2=map_dbl(data, lm_adj_r2))
```

```
# A tibble: 3 x 3
  sp         data              adj_r2
  <fct>      <list>             <dbl>
1 setosa     <tibble [50 x 2]> 0.0914
2 versicolor <tibble [50 x 2]> 0.611
3 virginica  <tibble [50 x 2]> 0.0851
```

3 Premiers pas et concepts-clés

3.1 Arithmétique

En premier lieu, R est une calculette. Vous pouvez copier-coller les blocs de code directement dans votre console R :

```
# back to school
1+2
```

[1] 3

```
3-4
```

[1] -1

```
2/3
```

[1] 0.6666667

```
2*2
```

[1] 4

```
# beyond +, -, /, *
2^4
```

[1] 16

```
sqrt(9) # equivalent to 9^(1/2)
```

[1] 3

```
7%%3
```

[1] 1

```
# precedence rules apply
(1.5-2)*4
```

[1] -2

```
1.5-2*4
```

[1] -6.5

Tous les opérateurs arithmétiques courants sont disponibles :

- `+`, `-`, `*`, `/` pour l'arithmétique de base
- `^` pour élever au carré et `sqrt` pour la racine carrée
- `%%` pour le modulo, etc.

Vous venez d'utiliser - peut-être sans le savoir -, votre première fonction : `sqrt`, pour *square root*. Les fonctions ont souvent des noms explicites et sont, généralement, suivies de parenthèses dans lesquelles glisser une ou des valeurs.

En réalité, toutes les opérations ci-dessus sont des fonctions, y compris un banal `+`[1]. Nous y reviendrons mais cela nous vaut la première maxime de sagesse populaire à garder dans un coin de la tête :

> En R, tout est fonction.

Vous pouvez également utiliser des parenthèses pour "emboiter" des opérations dans le bon ordre. Sans parenthèses, les règles de précédence[2] classiques s'appliquent comme dans l'exemple ci-dessus : une multiplication sera calculée avant une addition.

3.2 Variables et assignation

Comme on pouvait s'en douter, R a une mémoire et c'est donc plus qu'une calculette, c'est une calculatrice.

Pour stocker une valeur dans une **variable** nous utilisons l'opérateur d'**assignation** : `<-`[3]

```
plop <- 3
plop^2
```

```
[1] 9
```

[1] essayez donc `` `+`(1, 2) ``
[2] voir `?Syntax`
[3] `` `<-`(toto, 4); toto*2 `` puisqu'on te dit que tout est fonction !

D'autres opérateurs d'assignation existent (`->`, `=`, `<<-`, etc.) mais je vous conseille - calmement mais fermement - de vous en tenir au bon vieux `<-`[4].

Naturellement, si vous assignez une nouvelle valeur à une variable, celle-ci est remplacée :

```
plop <- 2
plop+3
```

```
[1] 5
```

Et bien entendu vous pouvez combiner les variables :

```
plip <- 7
plop*plip
```

```
[1] 14
```

Vous pouvez lister les variables existantes avec la fonction `ls()`. L'onglet "Environnement" dans RStudio permet aussi de les visualiser, plus convivialement et de façon plus détaillée.

Pour afficher la valeur d'une variable, et plus généralement d'un objet, il suffit de taper son nom dans la console et d'appuyer sur `<Entrée>`. En coulisses, R appelle alors automatiquement la fonction `print`.

R a tendance à faire beaucoup de choses en coulisses. Dans l'ensemble, cela se traduit par un caractère raisonnablement intuitif pour vous. Mais si une opération aussi naturelle qu'une assignation est une hérésie majeure pour un esprit informaticien orthodoxe, habitué à déclarer une variable et son type *avant* d'y assigner une valeur.

3.3 Bien nommer ses variables

> Mal nommer les choses c'est ajouter au malheur du monde – Camus

À propos du nommage de vos variables, soyez explicites mais compacts et évitez : les caractères spéciaux, les majuscules, les points (utilisez `_`) et les noms réservés comme `pi`. Ici aussi, l'anglais aide bien.

[4] L'assignation `->` est acceptable dans certains contextes, notamment en combinaison avec un pipe `%>%`. `=` est plutôt réservés aux arguments de fonctions ou, éventuellement, à des méta-paramètres en début de script. `<<-` est en revanche proscrit car il assigne dans l'environnement global. Utilisé dans une fonction, il remplacera dans l'environnement global la valeur de cette variable, si elle existe. Si vous pensez en avoir besoin, c'est généralement (>99.9%) que votre script/fonction est mal fichue.

```r
# good names
mod1
mod1_spain
N_perm

# bad names
my.model # . is for methods
MY_MODEL # calm down
My_Model # Camel Case works but not favoured
my_model_after_lda_a_and_data_subset2_flavourB # headaches and typos garanteed

# invalid or error-prone names
34_data
pi <- 4    # works but very bad idea
print <- 2 # same
&italy
```

Ce qui nous amène à une deuxième maxime populaire:

> There are only two hard things in Computer Science: cache invalidation and naming things. – Phil Karlton

Pour assigner plus d'une valeur à une variable, la fonction `c`, pour *concatenate*, est votre amie :

```r
toto <- c(1, 2, 3, 4, 5)
toto*2
```

```
[1]  2  4  6  8 10
```

Jusqu'ici toutes les variables assignées étaient des scalaires, c'est à dire des variables ne comprenant qu'une valeur. `toto` n'est pas un scalaire mais un *vecteur* de nombres, qui peut s'écrire en ligne, c'est à dire en une seule et unique *dimension*. Je pose ça là, nous y reviendrons bien vite.

Quelques fonctions bien utiles pour visualiser et décrire des variables en R :

```r
length(toto)
```

```
[1] 5
```

```r
head(toto, 2) # show the first 2 values
```

```
[1] 1 2
```

```r
tail(toto, 3) # show the last 3 values
```

```
[1] 3 4 5
```

Nous voyons que certaines fonctions acceptent plus d'une seule valeur au sein de leurs parenthèses : on parle d'**argument**. Quand plus d'un seul argument est **passé** à une fonction, on les sépare par des virgules. Certaines fonctions peuvent/doivent aussi être appelées à vide, comme on l'a vu avec `ls()`.

3.4 Séquences régulières

Vous m'avez vu (me) taper "à la main", des séries de nombres telles que `c(1, 2, 3, 4, 5)`. Même l'esprit le plus endormi constatera une certaine régularité dans cette séquence des premiers entiers naturels. Jusqu'à 5, on peut imaginer - et encore -, la taper à la main, mais imaginons que nous ayons besoin d'aller jusqu'à 100 ou même à 37427. Doit bien y avoir quelqu'un · e qui a pensé à une fonction pour faire ça non ? Ceci nous amène à une double maxime, peut-être les deux plus importantes de toute cette formation.

> Si tu penses que tu es en train de faire quelque chose de répétitif et/ou stupide, il existe à coup sûr une façon plus intelligente de faire

De façon plus compacte :

> Un · e bon · ne programmeur · se est une grosse feignasse

R possède toutes les fonctions dont on peut rêver pour générer séquences régulières et nombres aléatoires. Commençons par les séquences régulières et la versatile fonction `seq` qui prend au minimum deux arguments pour le point de départ et le point d'arrivée :

```r
seq(1, 10)
```

```
 [1]  1  2  3  4  5  6  7  8  9 10
```

Générér une séquence d'entiers naturels est une opération si banale en programmation que l'on peut faire encore pus court avec l'opérateur `:`:

```
1:5
```

```
[1] 1 2 3 4 5
```

```
-1:4
```

```
[1] -1  0  1  2  3  4
```

De la même façon, vous pouvez répéter une ou des valeurs avec `rep` :

```
tonton <- 1:5
rep(tonton, 2)
```

```
 [1] 1 2 3 4 5 1 2 3 4 5
```

```
rep(tonton, each=3)
```

```
 [1] 1 1 1 2 2 2 3 3 3 4 4 4 5 5 5
```

À ce point de votre existence, vous devez vous poser les questions suivantes :

- faut-il se rappeler de tout ça ?
- ces histoires de paramètres de fonction, attends… ah oui, le Monsieur a dit d'apeller ça des *arguments*, c'est bien joli mais on les trouve où ?
- et d'ailleurs il doit bien y avoir une documentation quelque part pour tout ce bazar ?
- quand est-ce qu'on fait une pause ?

Vous vous posez les bonnes questions et il est temps de faire un interlude, et non des moindres, l'interlude : `?`

3.5 Interlude clavier

- Toutes les commandes tapées depuis l'ouverture de R/RStudio sont dans votre console. Pour l'effacer, pressez `<Ctrl> + <L>`. Vos objets sont conservés.
- Pour naviguer dans votre historique, côté console, pressez les flèches `<Haut>` et `<Bas>`.
- Pour compléter un nom de fonction ou d'argument, pressez `<Tab>`

- Une fois que vous avez la fonction qui vous intéresse, positionnez-vous au sein de ses parenthèses et pressez `<Tab>` de nouveau : la liste des arguments apparaît avec la portion de doc consacrée.
- Dans RStudio, allez à peu près tout tester dans le menu `Code`. Vous y trouverez des fonctions très utiles ("Reindent lines", "Reformat code" par exemple). À droite de la commande se trouve aussi les raccourcis clavier. Utilisez-les !

3.6 Fonctions

Les fonctions sont généralement abordées plus tard mais je crois non seulement en vous mais aussi qu'elles doivent être démystifiées précocément.

3.6.1 Que sont les fonctions ?

Les fonctions sont des unités de code qui font quelque chose d'utile. Le plus souvent on envoie une valeur et on en récupère une autre. `sqrt()` par exemple renvoie la racine carrée de la valeur *passée* en *argument*.

Les arguments, séparés par des virgules, définissent les "options" de la fonction concernée. Une fonction peut avoir zéro, un, plusieurs et même un nombre indéfini d'argument.

3.6.2 Écrire ses fonctions : `function`

Vous pouvez définir vos propres fonctions avec la fonction `function`. On va encapsuler une portion de code entre des accolades `{` et la nommer. Définissons une fonction qui ajoute `3` à un argument que l'on va appeller `x` (vous pouvez essayer avec `y` ou `toto` :

```
plus3 <- function(x) {
  x+3
}

plus3(5)
```

```
[1] 8
```

```
plus3(1:3)
```

```
[1] 4 5 6
```

Les fonctions permettent de ne pas copier-coller bêtement du code, de dupliquer des lignes. À terme, cela vous permettra d'avoir du code non dupliqué, moins propice à des erreurs de frappes. Aussi, si vous changez d'avis, vous pourrez changer la définition de fonction et, chaque fois qu'elle sera appelée en aval, le nouveau comportement sera appliqué. Nous reviendrons sur ces bonnes pratiques à la toute fin de cette formation.

3.6.3 Notions d'environnement

La valeur `x` ci-dessus, dans la définition de fonction, est une variable utilisée uniquement dans l'**environnement** de la fonction `plus3`. Un environnement est un espace de travail mémoire isolé du reste du monde et qui n'existe que durant l'exécution de la fonction.

Autrement dit, si la variable `x` est déja assignée dans l'**environnement global**, c'est à dire en dehors de la fonction, dans la console si on veut, et bien ce `x` global ne sera ni modifié, ni même utilisé. Un exemple :

```
x <- 45
plus3(5)
```

```
[1] 8
```

Naturellement vous pouvez tout de même utiliser votre `x` global comme argument de la fonction `plus3`. Ce `x` global va être utilisé localement par la fonction puis vous être retourné :

```
plus3(x)
```

```
[1] 48
```

3.6.4 Documentation des fonctions : `?`

Toute fonction déjà disponible en R ou un package a forcément une page d'aide dédiée à laquelle on accède avec : `?nom_de_la_fonction`. Revenons à notre fonction `seq` bien pratique pour créer des séquences régulières. En tapant `?seq` on accède au contenu suivant :

```
Sequence Generation

Description
Generate regular sequences. seq is a standard generic with a default method. seq.int is a pri

## Default S3 method:
```

```
seq(from = 1, to = 1, by = ((to - from)/(length.out - 1)),
    length.out = NULL, along.with = NULL, ...)

seq.int(from, to, by, length.out, along.with, ...)

seq_along(along.with)
seq_len(length.out)

Arguments
[...]

Details
[...]

Value
[...]

See also
[...]

Examples
[...]
```

Toutes les pages d'aides ont la même structure, et les sections suivantes :

- **Description** : ce que la fonction fait
- **Usage** : la fonction "déployée" c'est à dire avec tous ses arguments. Parfois plusieurs fonctions sont regroupées dans une même age d'aide, comme c'est le cas pour `seq`. Ces fonctions peuvent être des variantes avec des noms différents ou des **méthodes** c'est à dire des fonctions au comportement différent selon le type d'objet sur lequel elles opèrent.
- **Arguments** : un descriptif de tous les arguments disponibles. La classe et le format de chacun d'eux est mentionnée.
- **Details** : souvent un remède à l'insomnie mais les subtilités d'implémentation sont là, souvent cachées au détour d'une phrase.
- **Value** : ce que la fonction retourne.
- **References** : où s'en référer si vous n'en avez pas assez
- **See Also** : fonctions connexes, très pratique pour enrichir son vocabulaire et trouver son bonheur.
- **Examples** : peut être la plus utile de toutes avec ses exemples d'utilisation que vous pouvez copier-coller ou même appeller directement depuis la console avec `example("nom_de_la_fonction")`. Vous pouvez essayer `example("plot")` par exemple.

- En pied de page, vous avez également une information qui sera utile plus tard : le package et sa version dans lequel se trouve être cette fonction. Pour `seq`, on est dans le package `base` dont toutes les autres fonctions sont indexées dans le lien “Index”.

Certaines pages d’aide, surtout pour le langage lui-même, sont plutôt des résumés du fonctionnement et sont un peu moins intuitives à trouver, par exemple `?Arithmetic`, `?Special`, `Syntax`. D’autres fonctions, par exemple pour les opérateurs, doivent être encadrées de guillemets arrières (`), par exemple `?`+`` ou la mise en abyme de `?`?``. Enfin, il existe d’autres ressources comme les “vignettes”, plus conviviales, surtout pour les packages les plus récents. Nous y reviendrons.

Les pages d’aide sont souvent compactes et obscures mais l’information que vous cherchez est probablement là. On apprend beaucoup à lire ces pages d’aide même si à première vue cette littérature n’est guère attrayante.

Enfin, la variante `??(quoiquoiquoi)`, raccourci de `help.search("quoiquoiquoi")` permet de chercher toutes les occurences de `quoiquoiquoi` dans *toutes* les pages d’aide de R.

3.6.5 Arguments : noms, positions et valeurs par défaut

Après avoir consulté `?seq` on peut par exemple préciser le point de départ (`from`), le point d’arrivée (`to`), le pas (`by`) et la longueur totale du vecteur à créer (`length.out`).

Vous constaterez que `from` est défini avec une valeur par défaut (`from=1`). Ainsi, si vous omettez sa valeur et ne spécifiez que `to`, `from` prendra sa valeur par défaut. Ces deux commandes sont donc équivalentes :

```
seq(from=1, to=5)
```

```
[1] 1 2 3 4 5
```

```
seq(to=5)
```

```
[1] 1 2 3 4 5
```

Vous pouvez également abréger le nom des arguments, moyennant que l’abbréviation soit univoque, c’est à dire que le nom de l’argument que vous abrégez ne soit pas identique à celui d’un autre argument. Ainsi `from` et `length.out` peuvent être abrégés en `fr` et `length`:

```
seq(fr=0, to=2, length=5)
```

```
[1] 0.0 0.5 1.0 1.5 2.0
```

Ce n'est jamais une bonne idée mais si vous utilisez le nom complet ou une abbréviation des arguments vous pouvez changer leur ordre. Ainsi `seq(length=5, to=2, fr=0)` sera équivalent à la commande précédente.

Vous pouvez même omettre le nom des arguments comme on l'a fait dans les sections précédentes sans le mentionner. Dans ce cas, les arguments sont passés positionnellement et doivent être renseignés dans l'ordre tel que défini dans la section 'Usage' de leur documentation:

```
seq(-3, 4, 12)
```

```
[1] -3
```

3.7 Concept de recyclage

```
toto <- 1:5
toto*3
```

```
[1]  3  6  9 12 15
```

Il n'aura pas échappé à votre sagacité que dans le précédent exemple, ou une multiplicatio est opérée entre deux objets de tailles différentes (cinq valeurs et une seule), R vous a compris et a multiplié *chaque* élément de `toto` par 3.

C'est l'idée, omniprésente, de **recyclage**. Ça ne parait pas grand chose mais c'est souvent bien pratique et, hélas, quelques fois glissant. Par exemple, si vous multipliez deux vecteurs non-conformes, c'est à dire ni de même longueur, ni multiples l'un de l'autre, des effets indésirables peuvent se manifester. Ou pire encore, rester cachés.

```
toto <- c(1, 2, 3, 4)
tata <- c(5, 4, 3)
toto*tata
```

```
Warning in toto * tata: longer object length is not a multiple of shorter object
length
```

```
[1]  5  8  9 20
```

De nos jours, R a tendance à émettre des Warnings quand un recyclage exotique est impliqué. Lisez les messages et autres warnings ! Celui-ci est plutôt explicite mais si vous n'y comprenez goutte, copiez-collez le message dans un moteur de recherche.

3.8 Indexation `[` : saisir et changer des valeurs

Indexer une ou des valeurs c'est sélectionner un sous-ensemble de valeurs dans une variable pour en faire quelque chose.

L'opérateur d'indexation est le crochet droit : `[`, qui tel Dupont et Dupond vont par paires. À gauche du crochet, la variable; à l'intérieur l'indice ou les indices.

```
tutu <- c(7, 12, 2, 5, 4)
tutu[1]
```

```
[1] 7
```

```
tutu[3]
```

```
[1] 2
```

```
tutu[length(tutu)] # take the last value of tutu, no matter tutu' length
```

```
[1] 4
```

```
tutu[c(2, 4)]
```

```
[1] 12  5
```

L'indexation peut se combiner avec l'assignation si on ne veut pas seulement saisir les données mais en faire quelque chose :

```
tutu
```

```
[1]  7 12  2  5  4
```

```r
tutu[c(2, 3)] <- c(-1, -3)
tutu
```

```
[1]  7 -1 -3  5  4
```

```r
tutu[c(2, 3)] <- 0.5  # indexing, assignation and recycling combined!
tutu
```

```
[1] 7.0 0.5 0.5 5.0 4.0
```

Ce type d'indexation est dit "positive" : l'indice réfère aux positions que l'on *veut*.

Bien pratique, l'opération d'indexation "négative" sélectionne ce que l'on ne veut *pas*.

```r
tutu
```

```
[1] 7.0 0.5 0.5 5.0 4.0
```

```r
tutu[-1]
```

```
[1] 0.5 0.5 5.0 4.0
```

```r
tutu[-c(1, 3)] <- pi
tutu
```

```
[1] 7.000000 3.141593 0.500000 3.141593 3.141593
```

Et comme vous vous en doutiez, on peut également utiliser une variable pour indexer :

```r
toto <- c(6, 5, 4, 3)
tata <- c(2, 3)
toto[tata]
```

```
[1] 5 4
```

```
toto[-tata]
```

```
[1] 6 3
```

Le concept d'indexation est absolument central en R, et en programmation en général.

L'indexation peut varier à la marge avec des indices dans plusieurs dimensions comme nous le verrons plus loin (`iris[2, 3]`), voire même des doubles crochets (`[[`), mais le comportement présenté ici reste invariable. Youpi !

3.9 Opérateurs de comparaison et logiques

L'indexation est une belle occasion de parler des opérateurs de comparaison, très utiles pour filter vos données. L'idée est qu'on teste d'abord une condition dont on peut se servir pour indexer

```
tonton <- c(4, 8, 2, 9, 1, 3, 5)
test <- tonton < 5  # tests a condition
test
```

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

```
which(test) # returns indices of TRUE
```

```
[1] 1 3 5 6
```

```
# this can be used to index
tonton[which(test)]
```

```
[1] 4 2 1 3
```

```
# or more directly
tonton[test] # filter elements of tonton lower than 5
```

```
[1] 4 2 1 3
```

```r
# conditions can be combined
tonton[tonton < 5 & tonton >=2] # same with lower than 5 AND higher or equal to 2
```

```
[1] 4 2 3
```

```r
tata <- c(7:1)
tonton[tata %in% c(3, 4, 5)] # takes the elements of tonton for which those of that are in
```

```
[1] 2 9 1
```

Voilà une liste de tous les opérateurs de comparaison (`?Comparison`) :

opérateur	signification
`<`	strictement inférieur
`<=`	inférieur ou égal
`>`	strictement supérieur
`>=`	supérieur ou égal
`==`	égal
`!=`	différent
`%in%`	dans l'ensemble

Et à y être celle pour les opérateurs logiques (`?Logic`) sur lesquels nous reviendrons :

opérateur	signification
`!`	NOT
`&`	AND (élément par élément)
`&&`	AND
`|`	OR (élément par élément)
`||`	OR
`xor(x, y)`	OR (exclusif)

Parfois on peut également avoir besoin de `any`, `all`[5].

[5] `%notin%` et `none` n'existent pas en R mais on peut facilement les composer avec `!(a %in% b)` et `!(all(...))`

3.10 Classes d'objets

3.10.1 `class`

Accrochez-vous à votre voisin · e, nous abordons un concept clé. Jusqu'ici nous n'avons manipulé que des nombres, avec ou sans assignation à une variable. D'autres **classes** d'objets existent en R.

On peut accéder à la **classe** d'un vecteur avec la fonction `class`. On voit que les chaînes de caractères sont des `character` pour R:

```
class(toto)
```

```
[1] "numeric"
```

Imaginons que nous mesurions des individus et que nous enregistrions leurs prénoms, sexe, stature et si, oui ou non, ils ont subi une formation à R. Ces quatres variables auront des natures différents :

- `prenom` : sera plutôt des lettres
- `stature` : sera un nombre décrivant leur taille
- `sexe` : sera une étiquette pouvant prendre une et une seule des valeurs suivantes `{femme/homme/autre}`
- `formation` : sera un descripteur de type `vrai/faux` que l'on traduira en `TRUE/FALSE`.

Nous avons déja enregistré trois individus :

prenom	stature	genre	formation
Hildegarde	178	femme	TRUE
Jean-Jacques	163	homme	FALSE
Victor	184	autre	TRUE

3.10.2 `character`

Tentons de créer la première variable, c'est à dire la première colonne, à la main :

```
prenom <- c("Hildegarde", "Jean-Jacques", "Victor")
prenom
```

```
[1] "Hildegarde"   "Jean-Jacques" "Victor"
```

R ne fait pas d'histoires et nous a crée un vecteur de chaînes de caractères !

3.10.3 `numeric`

Créons maintenant le vecteur `stature`, on sait faire :

```
stature <- c(178, 163, 184)
stature
```

```
[1] 178 163 184
```

```
class(stature)
```

```
[1] "numeric"
```

Tous les vecteurs que nous avons crée jusqu'ici, `prenom` mis à part, était donc des `numeric`. Précisions que des variantes de `numeric` existent : `double`, `integer`, etc. mais vous n'aurez peut être jamais à vous en soucier.

3.10.4 `factor`

La colonne `genre` est un peu particulière puisque elle est une chaîne de caractères mais elle ne peut prendre que des valeurs définies, à savoir une et une seule valeur de l'ensemble : `{homme, femme, autre}`. La classe `factor` est là pour ça et la fonction `factor` permet de créer notre variable `sexe` :

```
genre <- factor(c("femme", "homme", "autre"))
class(genre)
```

```
[1] "factor"
```

```
genre
```

```
[1] femme homme autre
Levels: autre femme homme
```

Votre œil aiguisé aura détecté deux différences par rapport à `prenom` : l'absence de guillemets autour de chaque valeur et une ligne supplémentaire qui indique les valeurs possibles, les `levels` de ce vecteur.

Petite digression : les facteurs en R sont très pratiques mais assez piégeux. Nous aurons l'occasion d'y revenir mais avant ça, évacuons les dernier · e · s informaticien · ne · s pur sang de la salle. Imaginons qu'un nouveau `level`, une nouvelle catégorie, doive être créée pour la variable genre, disons `licorne`. Tentons l'opération candidement :

```
genre2 <- c(genre, "licorne")
genre2
```

```
[1] "2"       "3"       "1"       "licorne"
```

```
class(genre2)
```

```
[1] "character"
```

Malédiction (apparente)[6] que nous expliquerons plus tard.

3.10.5 `logical`

Laissons reposer les facteurs pour l'instant et continuons notre création de variable avec la colonne `formation`:

```
formation <- c(TRUE, FALSE, TRUE)
class(formation)
```

```
[1] "logical"
```

```
formation
```

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

Voici une classe très utile, les `logical`, souvent issus de tests et de comparaisons logiques, comme survolé précédemmet par exemple :

[6] Il y a encore plus "drôle": `c(genre, 0)*2`

```
  stature > 180
```

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

Nous avons donc crée nos quatre colonnes :

```
  prenom
```

```
[1] "Hildegarde"   "Jean-Jacques" "Victor"
```

```
  stature
```

```
[1] 178 163 184
```

```
  genre
```

```
[1] femme homme autre
Levels: autre femme homme
```

```
  formation
```

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

Nous avons vu autant de classes différentes (`character`, `numeric`, `factor`, `logical` respectivement) et, bonne nouvelle, on a quasiment fait le tour des classes ! Il nous en reste deux, très voisines : `list` et `data.frame`.

3.10.6 `list`

Une `liste` est un vecteur dont les éléments peuvent être de classes et de longueurs différentes, dont d'autres listes. Autrement dit, c'est la structure de données universelle en R :

```
  list(toto="A", tata=c(1, 3))
```

```
$toto
[1] "A"

$tata
[1] 1 3
```

```r
list(prenom, stature)
```

```
[[1]]
[1] "Hildegarde"   "Jean-Jacques" "Victor"

[[2]]
[1] 178 163 184
```

Vous pouvez comparer le résultat à `c(prenom, stature)`. Observez aussi au passage les doubles crochets droits (`[[`), dont vous n'aurez bien plus peur.

3.10.7 `data.frame`

Un type de liste bien pratique, devenu archi central en R moderne, est le `data.frame` qui est une liste dont la double particularité est que tous ses éléments sont nommés et de même longueur. Une façon un peu alambiquée de dire qu'il nous aura fallu tout cela pour réinventer en R le tableau excel ![7]. Nous y reviendrons abondamment.

Comme vous pouvez vous en douter, personne n'importe les données comme cela, on préférera lire directement une table `.csv` ou `.xlsx`. Minute papillon, on y vient.

```r
data <- data.frame(prenom=prenom,
                   stature=stature,
                   genre=genre,
                   formation=formation)
data
```

```
       prenom stature genre formation
1   Hildegarde     178 femme      TRUE
2 Jean-Jacques     163 homme     FALSE
3       Victor     184 autre      TRUE
```

```r
class(data)
```

```
[1] "data.frame"
```

[7] Ne dites à personne que j'ai écrit ça !

3.10.8 `is.*` et `as.*`

Les fonctions `is.*` permettent de tester les classes :

```
is.logical(formation)
```

```
[1] TRUE
```

```
is.factor(prenom)
```

```
[1] FALSE
```

```
is.data.frame(data)
```

```
[1] TRUE
```

Et les fonctions `as.*` permettent de convertir les classes, lorsque la conversion est pertinente :

```
as.factor(prenom)
```

```
[1] Hildegarde   Jean-Jacques Victor
Levels: Hildegarde Jean-Jacques Victor
```

```
as.character(genre)
```

```
[1] "femme" "homme" "autre"
```

3.11 Indexation multiple `[,`

Revenons à notre `data.frame` `data`, un objet en deux dimensions : il possède des lignes et des colonnes. L'indexation vue précédemment sur des vecteurs, des objets en une seule dimension, fonctionne dans le même esprit mais il nous faut renseigner les indices pour chaque dimension. Si l'un des indices n'est pas renseigné, toutes les positions concernées sont retournées mais il ne faut pas oublier pour autant la virgule !

Par convention la première dimension est celle des lignes, la deuxième celle des colonnes. Quelques exemples ci-dessous qui ne font qu'extraire les données. Nous pourrions aussi les modifier avec l'opérateur d'assignation `<-` comme vu précédemment.

```
data[1, ] # first row
```

```
      prenom stature genre formation
1 Hildegarde     178 femme      TRUE
```

```
data[, 1] # first column
```

```
[1] "Hildegarde"   "Jean-Jacques" "Victor"
```

```
data[-2, ] # everything but the second row
```

```
      prenom stature genre formation
1 Hildegarde     178 femme      TRUE
3     Victor     184 autre      TRUE
```

```
data[c(1, 3), -3] # first and third row, all columns but the third
```

```
      prenom stature formation
1 Hildegarde     178      TRUE
3     Victor     184      TRUE
```

Vous aurez noté qu'en ne sélectionnant qu'une colonne, par exemple `data[, 1]` ci-dessus, on perd la nature de `data.frame` pour revenir à la classe d'origine de la colonne. Souvent pratique, parfois glissant. Pour ne pas perdre la classe d'origine, il suffit d'ajouter `drop=FALSE` à votre opération d'indexation, après tous les indices :

```
data[, 2, drop=FALSE]
```

```
  stature
1     178
2     163
3     184
```

Les colonnes uniques peuvent aussi se sélectionner avec l'opérateur `$` qui permet d'accéder à un élément de liste, pourvu qu'il soit nommé. Les `data.frame`s étant des listes nommées, cela fonctionne :

```
data$stature
```

```
[1] 178 163 184
```

Les doubles crochets droits `[[` permettent eux aussi de sélectionner positionnellement, ou nominativement, l'élément de liste concerné. Les deux commandes ci-dessous seront équivalentes :

```
data[[3]]
```

```
[1] femme homme autre
Levels: autre femme homme
```

```
data[["genre"]]
```

```
[1] femme homme autre
Levels: autre femme homme
```

3.12 Indexation de liste : `[` versus `[[`

L'indexation avec une simple ou une double paire de crochets est souvent source d'incompréhension d'autant que sur des vecteurs le résultat est le même :

```
toto <- 5:3
toto
```

```
[1] 5 4 3
```

```
toto[2]
```

```
[1] 4
```

```
toto[[2]]
```

```
[1] 4
```

Gardez à l'esprit que la double paire de crochets droits (`[[`) ne s'utilise que sur des listes (`data.frame` y compris donc).

Pour expliquer simplement la différence entre `[` et `[[`, disons qu'une liste est un train de marchandise sans locomotive avec un ou plusieurs wagons.

- `[` permet de sélectionner un wagon, le résultat est toujours un train, certes minima**liste**
- `[[` permet de sélectionner le **contenu** du wagon qui n'est donc plus un train mais une vache, une palette ou des voyageurs :

```
tata <- list(wagon1="a", wagon2=1:3)
# [ picks a list element
tata[2]
```

```
$wagon2
[1] 1 2 3
```

```
class(tata[2])
```

```
[1] "list"
```

```
# [[ picks a list element AND drops the list
tata[[2]]
```

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

```
class(tata[[2]])
```

```
[1] "integer"
```

Notons que `$` est équivalent à `[[`:

```
tata$wagon2
```

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

3.13 `matrix`

Par souci de complétude, mentionnons les matrices qui sont des tableaux rectangulaires de nombres. Elles se créent avec la fonction `matrix` et en spécifiant les valeurs de remplissage et le nombre de lignes et/ou de colonnes :

```
m <- matrix(c(3, 1, 9.2, 6, 7, 0), nrow=2)
m
```

```
     [,1] [,2] [,3]
[1,]    3  9.2    7
[2,]    1  6.0    0
```

Les `matrix` et les `data.frame` possèdent de nombreux points communs. On peut notamment accéder à leurs dimensions, noms de lignes et colonnes avec les mêmes fonctions :

```
dim(m)
```

```
[1] 2 3
```

```
nrow(m)
```

```
[1] 2
```

```
ncol(m)
```

```
[1] 3
```

```
rownames(m) <- c("plop", "plip")
rownames(m) # idem for colnames
```

```
[1] "plop" "plip"
```

Une opération courante sur des matrices consiste à faire des calculs marginaux, par exemple calculer la somme par colonnes. La famille `apply` permet ce type de calcul. On va renseigner trois arguments (voir `?apply` et ses exemples) : l'objet sur lequel travailler; la dimension sur laquelle calculer (`1` pour les lignes, `2` pour les colonnes); et enfin la fonction à appliquer, sans parenthèses :

```
apply(m, 2, sum)
```

```
[1]  4.0 15.2  7.0
```

3.14 `array`

Par souci de complétude, l'idée de matrice se généralise dans des dimensions supérieures à 2. Pour le dire autrement, une `matrix` est un `array` à deux dimensions.

Les `array` se créent dans le même esprit : les valeurs de remplissage d'abord, puis on précise les dimensions. Ci-dessous, un array de deux tranches de matrices de `2x3`.

```
a <- array(data=c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12), dim=c(2, 3, 2))
a
```

```
, , 1

     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6

, , 2

     [,1] [,2] [,3]
[1,]    7    9   11
[2,]    8   10   12
```

L'indexation fonctionne ici aussi. Par exemple si l'on veut la valeur de la première ligne, troisième colonne, deuxième tranche :

```
a[1, 3, 2]
```

```
[1] 11
```

3.15 Fonctions utiles

3.15.1 Sur `numeric`

Quelques fonctions utiles pour décrire des vecteurs numériques (`numeric`) :

```
x <- c(5, 4, 3, 2, 1)
length(x)
min(x)
max(x)
range(x)  # shortcut for c(min(x), max(x))
median(x) # shortcut for quantile(x, probs=0.5)
sum(x)
mean(x)   # average
sd(x)     # standard deviation
var(x)    # variance
```

```
[1] 5
[1] 1
[1] 5
[1] 1 5
[1] 3
[1] 15
[1] 3
[1] 1.581139
[1] 2.5
```

3.15.2 Sur `factor`

```
f1 <- factor(c("apple", "banana", "banana", "pear", "grape", "grape"))
f2 <- factor(c("yellow", "yellow", "yellow", "green", "red", "green"))

length(f1)  # length
levels(f1)  # levels, as character
nlevels(f1) # number of levels, shortcut for length(level(f1))
table(f1)   # count occurences
table(f1, f2) # cross-tabulate factors
```

```
[1] 6
[1] "apple"  "banana" "grape"  "pear"
```

```
[1] 4
f1
 apple banana  grape   pear
     1      2      2      1
        f2
f1       green red yellow
  apple      0   0      1
  banana     0   0      2
  grape      1   1      0
  pear       1   0      0
```

3.15.3 Sur `character`

```r
bla1 <- "tonton"
bla2 <- "tata"

nchar(bla1) # count characters
paste(bla1, bla2, sep=" et ") # see also paste0
toupper(bla1) # convert to upper case. see also tolower()
gsub("o", "i", bla1) # replace all 'o' with 'i' in bla1. see ?grep
substr(bla1, 2, 4) # character from positions 2 to 4
abbreviate(bla1) # abbreviate
```

```
[1] 6
[1] "tonton et tata"
[1] "TONTON"
[1] "tintin"
[1] "ont"
tonton
"tntn"
```

Les derniers exemples montrent des manipulations de chaînes de caractères. Le package `stringr` remplace avantageusement ces approches "historiques". Nous y reviendrons.

3.15.4 Sur `data.frame`

```r
dim(data)
nrow(data)
ncol(data) # equivalent to length(data)
summary(data) # a summary, column-wise
```

```r
head(data) # show only the first rows
tail(data) # show only the last rows
# View(data) # show an interactive viewer for your data.frame
str(data)  # show the structure of your data.frame
```

```
[1] 3 4
[1] 3
[1] 4
    prenom             stature        genre   formation
 Length:3           Min.   :163.0   autre:1   Mode :logical
 Class :character   1st Qu.:170.5   femme:1   FALSE:1
 Mode  :character   Median :178.0   homme:1   TRUE :2
                    Mean   :175.0
                    3rd Qu.:181.0
                    Max.   :184.0
        prenom stature genre formation
1   Hildegarde     178 femme      TRUE
2 Jean-Jacques     163 homme     FALSE
3       Victor     184 autre      TRUE
        prenom stature genre formation
1   Hildegarde     178 femme      TRUE
2 Jean-Jacques     163 homme     FALSE
3       Victor     184 autre      TRUE
'data.frame':   3 obs. of  4 variables:
 $ prenom   : chr  "Hildegarde" "Jean-Jacques" "Victor"
 $ stature  : num  178 163 184
 $ genre    : Factor w/ 3 levels "autre","femme",..: 2 3 1
 $ formation: logi  TRUE FALSE TRUE
```

graphics:::plot.data.frame

```r
set.seed(2329)
runif(10, -1, 1)
```

```
 [1]  0.34341687 -0.89818619  0.86519883  0.05414381 -0.95204424  0.42169961
 [7]  0.09957463  0.27190793 -0.84168929  0.72023581
```

3.16 Générer des nombres aléatoires

3.16.1 Au sein d'une séquence existante

La fonction `sample` permet d'échantillonner au sein d'un vecteur existant. Vous devenz préciser ce vecteur, puis le nombre de tirages à effectuer, avec ou sans remise.

Si vous n'avez pas de dé, en voilà un :

```
sample(1:6, size=1)
```

```
[1] 6
```

Avec `size=3, replace=TRUE` vous pouvez même jouer au 421. Ou encore générer un tirage de loto sans les boules qui s'agitent :

```
sample(1:49, size=6, replace=FALSE)
```

```
[1]  9 47 36 14  4 34
```

Chaque élément du vecteur a autant de chance de sortir qu'un autre. Si vous préférez le scrabble, vous pouvez aussi utiliser `sample` mais en calibrant l'argument `probs` sur la fréquence des lettres dans la langue française. N'allez donc pas vous taper l'alphabet à la main, jeter un oeil à `letters` et `LETTERS`.

3.16.2 Distributions existantes

Au delà des vecteurs existants, vous pouvez générer des nombres aléatoires issus d'une distribution. Toutes les distributions disponibles sont listées dans la bien nommée `?Distriutions`.

Ces fonctions sont nommées de la façon suivante `{r, p, q, d}nom_abregé_distrib`. La première lettre désigne la variante désirée des différentes fonctions pour une distribution donnée, selon que l'on veuille générer des nombres, la densité de probabilité, les quantiles associés, etc.

Pour générer 10 nombres aléatoires compris entre -1 et 1 on peut par exemple :

```
runif(10, 0, 1) # see ?runif
```

```
[1] 0.1193610 0.4918118 0.7741212 0.3069952 0.7218488 0.4027434 0.4372506
[8] 0.8760479 0.8498670 0.4134920
```

Dans le même esprit on peut tirer 1000 nombres issus d'une distribution normale de moyenne 5 et d'écart-type 3 avec la commande suivante :

```
x <- rnorm(1e3, 5, 3)
mean(x)
```

```
[1] 5.028203
```

```
sd(x)
```

```
[1] 3.014893
```

`1e3` est la notation dite "ingénieur" parfaitement comprise par R. Ici, on a 1 suivi de 3 zéros, soit `10^3` soit `1000`. Vous constaterez également que la moyenne est à peu près de 5 et l'écart type à peu près égal à 3.

Si vous réexécutez cette commande, vous aurez un autre vecteur avec des valeurs différentes mais également à peu près centré sur 5.

```
y <- rnorm(1e3, 5, 3)
mean(x) - mean(y)
```

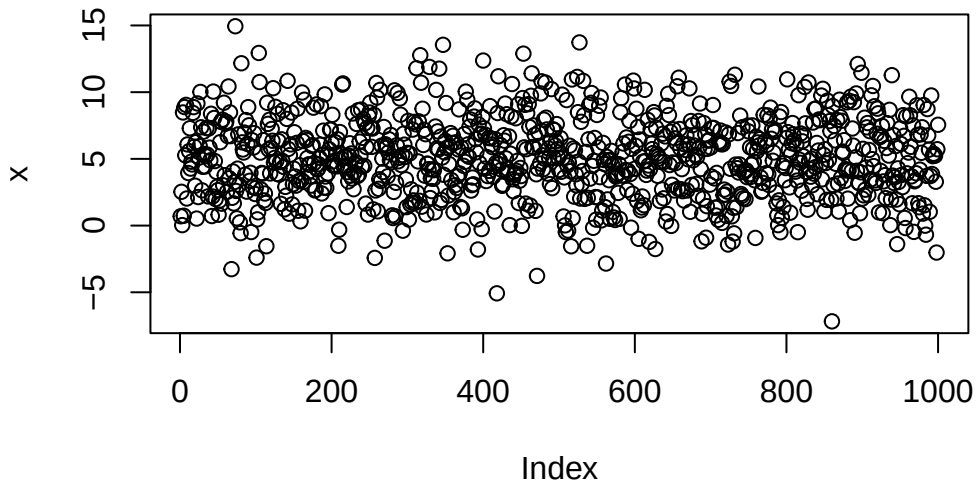
```
[1] -0.01312834
```

Si vous désirez générer des nombres aléatoires certes, mais les mêmes, par exemple d'une session à l'autre ou (c'est la même chose) sur l'ordinateur de votre collègue, c'est possible avec la fonction `set.seed`.

3.17 Premiers graphes

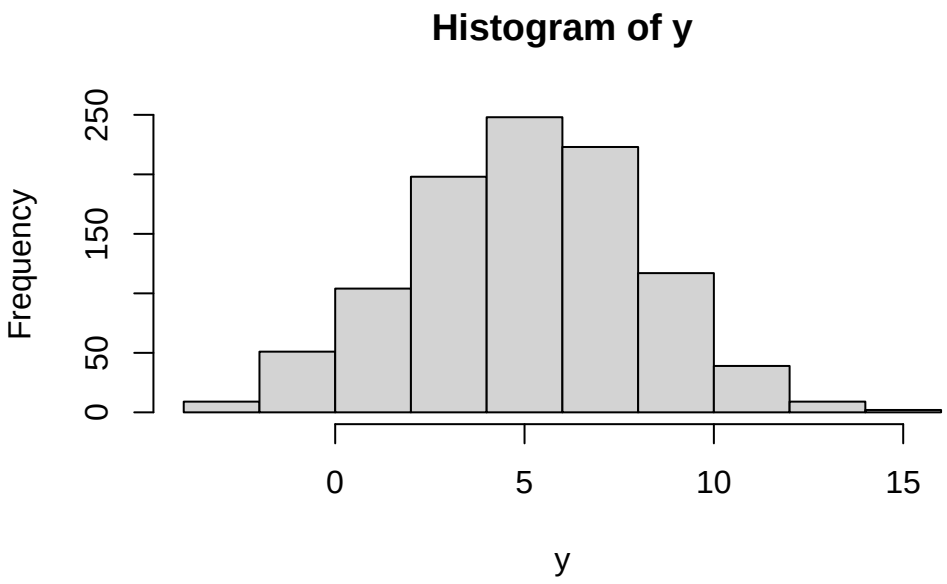
Décrire un vecteur de nombres avec `mean` et `sd` mais faire des jolis graphes c'est mieux. La commande `plot` est la fonction de base pour ce faire :

```
plot(x)
```



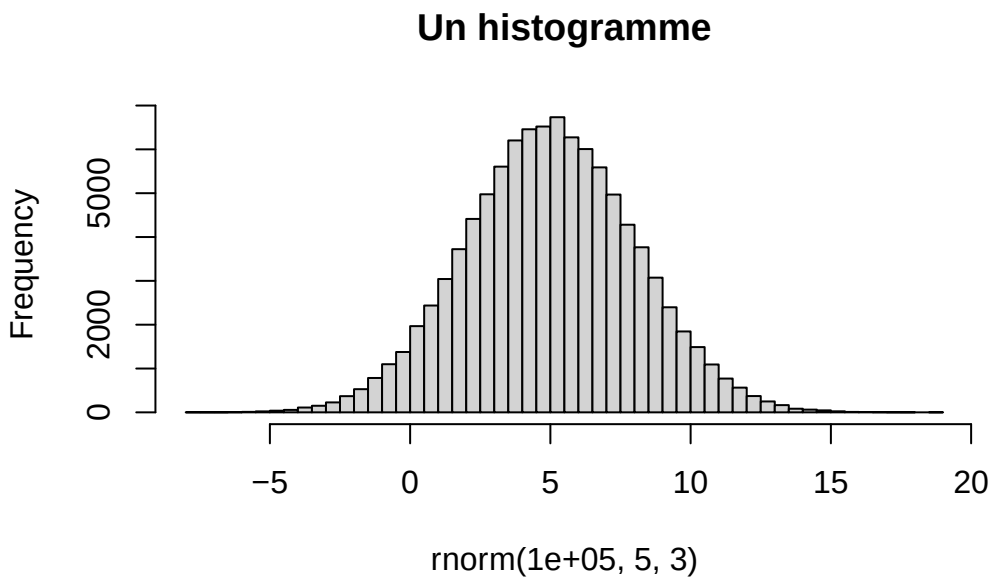
R a compris que chaque valeur du vecteur `x` devait être plottée sur l'axe de ordonnées et en a déduit que l'axe des abscisses devait être la série de 1 à 1000. Ici un histogramme serait plus approprié pour décrire la distribution de notre vecteur de nombres aléatoires :

```
hist(y)
```



On voit qu'il est bien centré sur `5`. Si l'on augmente le nombre de valeurs générées ainsi que le nombres d'intervalles (`breaks` en anglais), on retrouve une bien belle gaussienne :

```
hist(rnorm(1e5, 5, 3), breaks=50, main="Un histogramme")
```



Les graphiques de base de R sont un peu surannés depuis l'avènement de `gpplot2` que nous verrons en détail plus loin mais ils ont encore leur mot à dire, même s'il ne s'exprimeront pas longuement ici.

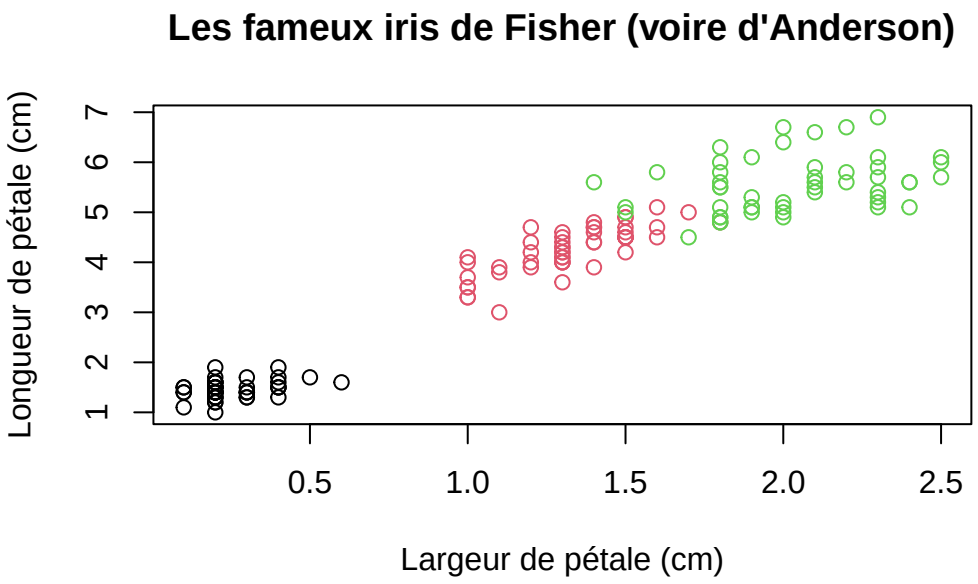
Voyons tout de même ce que l'on peut faire avec `iris`, l'un des nombreux jeux de données livrés avec R[8].

```
head(iris)
```

```
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          5.1         3.5          1.4         0.2  setosa
2          4.9         3.0          1.4         0.2  setosa
3          4.7         3.2          1.3         0.2  setosa
4          4.6         3.1          1.5         0.2  setosa
5          5.0         3.6          1.4         0.2  setosa
6          5.4         3.9          1.7         0.4  setosa
```

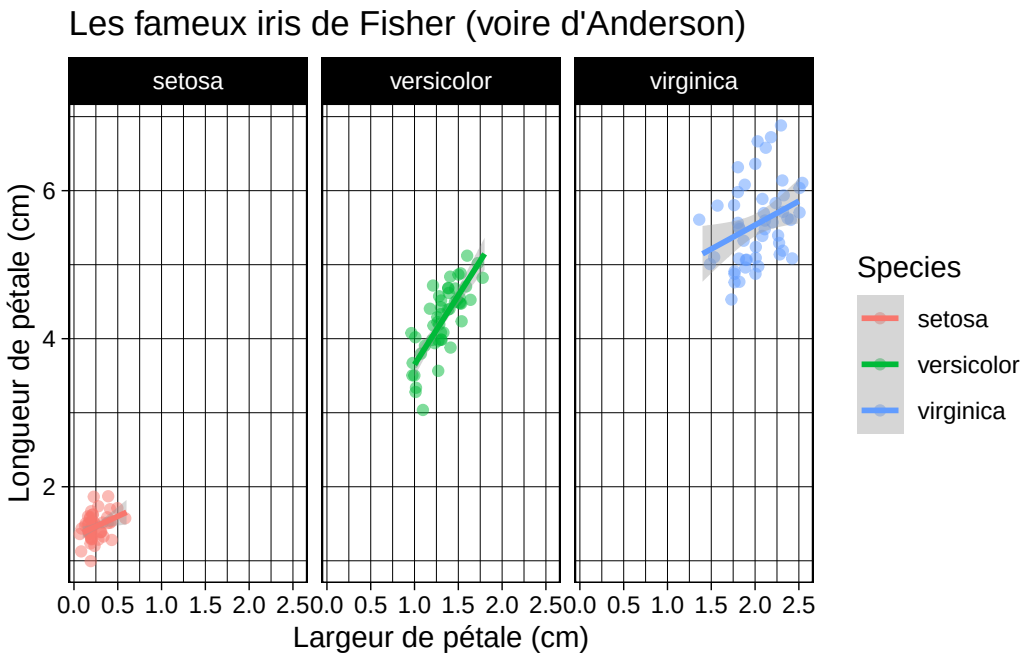
```
plot(Petal.Length~Petal.Width, data=iris, col=Species,
     xlab="Largeur de pétale (cm)", ylab="Longueur de pétale (cm)",
     main="Les fameux iris de Fisher (voire d'Anderson)")
```

[8] voir `?datasets`.



En petit aguichage `ggplot2` voilà ce qu'on peut obtenir dans le même temps :

```
library(ggplot2)
ggplot(iris) +
  aes(x=Petal.Width, y=Petal.Length, col=Species) +
  geom_jitter(alpha=0.5) +
  geom_smooth(method="lm", formula="y~x") +
  facet_grid(~Species) +
  xlab("Largeur de pétale (cm)") +
  ylab("Longueur de pétale (cm)") +
  ggtitle("Les fameux iris de Fisher (voire d'Anderson)") +
  theme_linedraw()
```



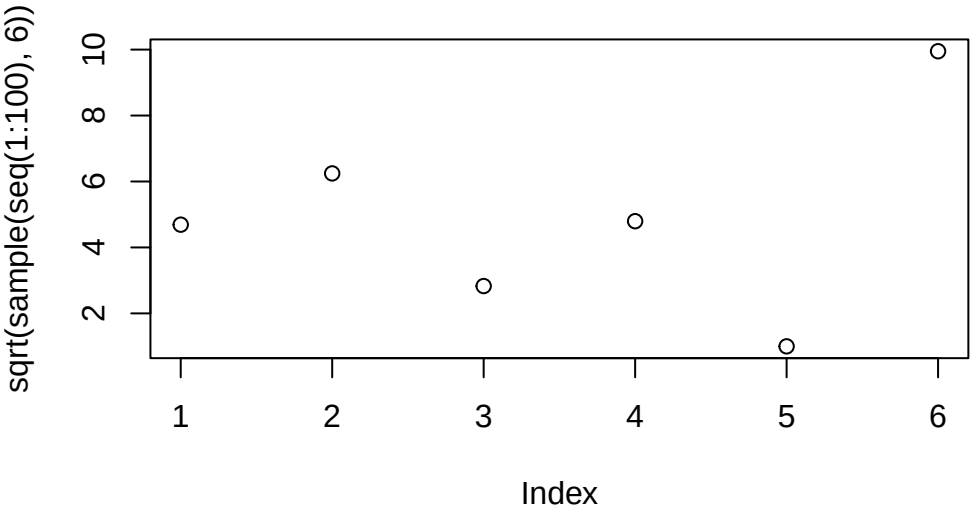
3.18 Un mot sur les packages

3.19 L'opérateur *pipe* %>%

L'opérateur pipe permet de passer, de gauche à droite, le résultat d'une fonction à une seconde fonction, puis à une troisième. Ce pipe est détaillé dans un chapitre mais sa compréhension devrait être intuitive.

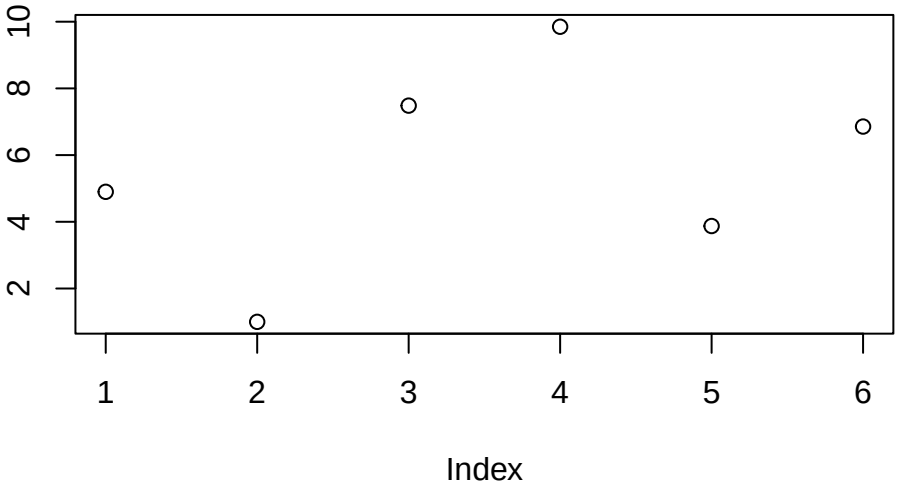
Comparez par exemple ces deux lignes pourtant strictement équivalentes:

```
library(magrittr)
plot(sqrt(sample(seq(1:100), 6)))              # road to burnout
```




```
seq(1:100) %>% sample(6) %>% sqrt() %>% plot() # let's breathe
```

.



Les packages du R moderne, en premier lieu ceux du `tidyverse` en ont fait une idée centrale de leur design et il est peu dire que nous autres mortel · le · s en profitons tous les jours.

3.20 Trucs et astuces pour R et RStudio

- Toutes les commandes tapées depuis l'ouverture de R/RStudio sont dans votre console. Pour l'effacer, pressez `<Ctrl> + <L>`. Vos objets sont conservés.
- Pour naviguer dans votre historique, côté console, pressez les flèches `<Haut>` et `<Bas>`.
- Pour compléter un nom de fonction ou d'argument, pressez `<Tab>`

Le flux classique de travail est le suivant :

1. Vous voulez faire quelque chose
2. Vous bidouillez dans la console
3. Vous êtes satisfait · e de votre commande
4. Vous la sauvez dans un script
5. Repartez à 1.

Au fur et à mesure de votre avancée, votre script va se remplir. Demain, dans 6 mois, ou sur un autre ordinateur, vous pourrez refaire "tourner" vos analyses et avoir strictement les mêmes résultats. On parle de **reproductibilité**. Gage de science sérieuse et, pour vous, de sérenité. Pour ces raisons, je vous conseille de ne pas enregistrer votre environnement de travail quand vous fermez R ou RStudio.

Un script peut être un fichier texte ou `.R`. RStudio gère bien les différents scripts en affichant dans la même fenêtre votre console, vos scripts, vos graphes, etc.

- De temps à autre, faites tout "retourner". RStudio a un raccourci pour cela : `Run > Restart R and run all`.
- Vous pouvez également faire tourner un autre script depuis le script en cours avant `source`. Bien pratique par exemple pour mettre toutes vos fonctions dans un script et les analyses à proprement parler dans un autre.

RStudio a pléthore raccourcis et fonctionnalités bien pratiques par exemple pour formater votre code selon les standards en cours (`Code > Reformat Code`) ou simplement pour réindenter proprement votre code. L'indentation est le décalage par rapport à la marge gauche du script. Cette indentation est purement esthétique en R, contrairement à Python par exemple. Elle est bien pratique pour sauter une ligne qui serait trop longue, ce qui est déconseillé.

3.21 Pour la suite

Si vous lisez ces lignes c'est que vous avez survécu jusqu'ici, bravo. Vous avez fait le plus dur, vous parlez déjà R. Le reste est une longue promenade en faux plat qui tourne autour de ces idées. Bien sûr, votre vocabulaire va s'enrichir, votre syntaxe sera de plus en plus concise et vous passerez ainsi de plus en plus de temps à la plage.

Bien sûr, vous aurez envie de mettre le feu à l'ordinateur, votre bureau et l'intégralité du monde open-source. N'en faites rien, asseyez vous en tailleur et méditez la sentence de notre maitre à tou · te · s, Hadley Wickham :

> Frustration is typical and temporary

L'échec, l'incompréhension, la frustration sont la matérialisation de l'apprentissage. Restez calmes, gardez votre sang-froid, tapez votre problème dans Google, lisez Stack Overflow et faites vous confiance. Perséverez. Je crois en vous.

4 Éléments de programmation

4.1 Control flow

Faisons comme Houellebecq et pompons Wikipédia qui écrit mieux que nous :

> En programmation informatique, une structure de contrôle est une instruction particulière d'un langage de programmation impératif pouvant dévier le flot de contrôle du programme la contenant lorsqu'elle est exécutée. Si, au plus bas niveau, l'éventail se limite généralement aux branchements et aux appels de sous-programme, les langages structurés offrent des constructions plus élaborées comme les alternatives (if, if–else, switch…), les boucles (while, do–while, for…) ou encore les appels de fonction.

`if`, `else`, `ifelse`, `for`, `while`, `next`, `break`

En termes plus directs, le *control flow* dont disposent à peu près tous les langages de programmation, permet d'adapter le comportement d'un code en fonction des circonstances.

4.2 `if`

Imaginons par exemple que nous voulions écrire une fonction qui imprime à la console si un nombre est positif ou négatif. La structure `if` permet de tester une condition logique et, généralement, d'agir en conséquence. `?Control` nous apprend que son patron général est le suivant :

```
if (condition) expr
```

`condition` sera un test logique qui, s'il est vrai, exécutera la ou les commandes `expr`. Si la commande est unique vous pouvez l'écrire en ligne. Si elle est multiple, on l'embrassera d'accolades :

```
if (condition) {
expr1
expr2
etc.
}
```

Notre fonction pourrait ressembler à :

```
signe1 <- function(x){
  if (x > 0) {
    cat(x, "est positif")
  }
  if (x < 0) {
    cat(x, "est négatif")
  }
}

signe1(-1)
```

```
-1 est négatif
```

```
signe1(5)
```

```
5 est positif
```

Ici, nous omettons le classique `return` car la fonction ne retourne rien, elle se contente d'émettre un message dans la console avec `cat()` ce qui est très différent. Du reste, vous pouvez presque toujours omettre `return` car une fonction retourne simplement la dernière valeur de son code. Ci-dessous, `signe1()` ne retourne rien car `toto` est `NULL` (sauf si vous l'avez assigné avant naturellement).

```
toto <- signe1(1)
```

```
1 est positif
```

```
toto
```

```
NULL
```

4.3 `else`

Plutôt que d'empiler les `if`, vous pouvez utiliser `else` quand si ce n'est pas un cas, c'est forcément l'autre. Comme cela (à première vue) semble être le cas ici :

```
signe2 <- function(x){
  if (x > 0) {
    cat(x, "est positif")
  } else {
    cat(x, "est négatif")
  }
}

signe2(-1)
```

```
-1 est négatif
```

```
signe2(5)
```

```
5 est positif
```

En réalité ici, on a une complication supplémentaire quand `x=0`, que n'est pas prévu par `signe1` et pire encore par `signe2` (essayez donc `signe1(0)` et `signe2(0)`). Les `if/else` peuvent être emboités et l'indentation de code se révèle particulièrement utile. (`Code > Reindent lines` dans RStudio, ou `<Ctrl>+A, <Ctrl>+I`, en remplaçant par sous Mac).

```
signe3 <- function(x){
  if (x==0) {
    cat(x, "n'est ni négatif ni positif")
  } else {
    if (x > 0) {
      cat(x, "est positif")
    }
    if (x < 0) {
      cat(x, "est négatif")
    }
  }
}
```

```
signe3(0)
```

```
0 n'est ni négatif ni positif
```

```
signe3(-1)
```

```
-1 est négatif
```

```
signe3(5)
```

```
5 est positif
```

Ce comportement de choix multiples se généralise au delà de deux avec `switch`.

Vous pouvez tester plusieurs choses à la fois mais le résultat doit être un `logical` de longueur 1, c'est à dire soit `TRUE` soit `FALSE`. Si par exemple vous voulez tester si un nombre est positif ET inférieur à 10 alors vous utiliserez un patron de ce genre pour `cond` :

```
if ((x > 0) & (x < 10))
```

Finis les messages à la console, nous allons désormais retourner des nombres, en l'occurence `-1` si `x` est négatif, `1` sinon (zéro y compris):

```
signe4 <- function(x) {
  if (x<0)
    -1
  else
    1
}

signe4(-1)
```

```
[1] -1
```

```
signe4(0)
```

```
[1] 1
```

```
signe4(1)
```

```
[1] 1
```

Au passage, un exemple d'omission d'accolades lorsqu'une seule ligne est à exécuter. Cette structure aussi compacte avec un seul `if` et un seul `else`, et surtout une seule valeur retournée peut s'écrire de façon plus compacte avec `ifelse` suivante le patron : `ifelse(cond, expr_ifTRUE, expr_ifFALSE)` dans ce goût là : `signe5 <- function(x) ifelse(x<0, -1, 1)`

À ce moment de votre existence, vous vous dites "génial, je vais pouvoir balancer un vecteur à `signe5` et aller à la plage". Modérez votre enthousiasme :

```
signe4(-1:1)
```

```
Warning in if (x < 0) -1 else 1: the condition has length > 1 and only the first
element will be used
```

```
[1] -1
```

Cette commande aurait pu marcher, via un recyclage dans votre dos, mais ce n'est pas le cas, ce qui nous donne - heureux hasard - une transition rêvée vers `for`. Notez bien que je vous montre `for` mais que normalement vous ne devriez presque jamais en avoir besoin grâce à `map` et ses variantes, dans le package `purrr`.

4.4 `for`

Parcourir les valeurs d'un vecteur, d'une matrice, etc. et agir avec ces valeurs est une tâche très courante en programmation. `for` permet de balayer un vecteur donné et d'assigner temporairement cette valeur à une autre variable, généralement appelée `i` :

```
for (i in 1:5) {
  print(i^2)
}
```

```
[1] 1
[1] 4
[1] 9
[1] 16
[1] 25
```

Le code ci-dessus devrait être transparent. Une précision toutefois : je suis obligé de forcer l'impression à la console avec `print` (une variante moins subtile de `cat`) sinon ce qu'il se passe dans les accolades de `for` y resterait, sans conséquence visible.

Si vous voulez utiliser les résultats d'un calcul au sein d'une boucle `for`, ce qui est le cas le plus fréquent, vous devez l'assigner à un objet compatible préalablement crée. De plus, on utilise généralement un vecteur avec toutes les valeurs à balayer par `i`. Plutôt que d'utiliser cette chose : `1:length(x)`, on préferera le plus court et souvent plus générique `seq_along(x)`

```
x <- 1:5
res <- vector("numeric", length=length(x))
for (i in 1:length(x)){
  res[i] <- x[i]^2
}
res
```

```
[1]  1  4  9 16 25
```

Au risque de me répéter, vous ne devriez pas avoir trop besoin de `for` si vous maîtrisez `map`, qui est plus compact, plus explicite et le plus souvent plus rapide :

```
purrr::map_dbl(1:5, ~.x^2)
```

```
[1]  1  4  9 16 25
```

4.5 `while` et al.

Je ne m'attarde pas sur les autres structures de contrôles, bien moins utilisées, très bien décrites ailleurs et plutôt compréhensibles si vous avez survécu jusqu'ici : `while`, `next`, `repeat`, `break`, etc.

4.6 Fonctions

Les fonctions sont des unités de code qui remplissent une fonction donnée qui n'existe nulle part dans R[1]. Elles sont au coeur de R, et plus largement de tous les environnements de programmation. Paraxalement et même si elles peuvent effrayer au début, il n'y a pas grand chose à connaître sur les fonctions !

[1] ou plus probablement vous n'avez tout exploré ici :-)

- Définir une fonction plutôt que copier-coller son code à chaque fois que vous en avez besoin est non seulement bien pratique mais aussi plus sûr : si vous avez une modification à faire vous la faites une fois dans la définition de fonction, pas partout.
- Idéalement une fonction fait une seule chose mais bien ! Faire des fonctions qui font aussi le café est un tropisme de débutant · e · s mais aue l'on perd vite. Oubliez donc `all_my_PhD_results()` et décomposez vos fonctions. Vos collaborateurs et vous-même dans six mois (c'est pareil), vous en sauront gré.
- Idéalement toujours, sauf pour les cas les plus triviaux, une fonction est documentée sur le modèle de la doc de R: c'est à dire son fonctionnement, les différents arguments, ce qu'elle retourne, des détails éventuels et des exemples. Commentez également l'intérieur de vos fonctions, ce sera utile pour tous y compris vous-même et tout de suite.
- Pour les scripts les plus complexes, vous pouvez regrouper vos fonctions dans un script indépendant, et le `source` r en début de votre script d'analyse ou de votre document Quarto[2].

  Ne soyez pas timides, crééz vos fonctions à chaque fois que vous répétez le même code

Passons aux choses sérieuses.

Les fonctions sont définies avec la fonction `fonction` suivant le patron :

```
nom_fonction <- function(arg1=default1, arg2=default2, ...) {
  # instructions
  return()
}
```

- Une fonction peut avoir zéro, un, plusieurs, ou même un nombre indéfini d'arguments.

```
# 0 argument
say_hi <- function() cat("hi there")
# try it: say_hi()

# pass arguments to internal functions with `...`
my_boxplot <- function(x, y, ...){
  boxplot(x, y, ...)
}
# try it: my_boxplot(iris$Petal.Length, iris$Species, col="grey50")
```

[2]pour donner un ordre de grandeur, sur mes cinq derniers projets de consulting, tous les scripts finaux dépassent les 500 lignes mais sourcent des scripts qui font plus de 1000 lignes, dont presque la moitié de commentaires !

- Une fonction a des arguments nommés. Quand c'est possible avec une valeur par défaut. Ces valeurs peuvent être calculées "à la volée" sur d'autres arguments (voir `by` dans `?seq`).
- Une fonction peut utiliser une fonction comme argument (`FUN` dans `apply` par exemple)
- Une fonction peut retourner une fonction[3]
- Le code d'une fonction est accessible, la plupart du temps, en tapant le nom de la fonction sans parenthèse :

```
say_hi
```

```
function() cat("hi there")
```

- Une fonction ne peut retourner qu'un seul objet. Si vous avez plusieurs, il vous faut en passer par une liste, si possible nommée : `return(list(res1=... res2=...))`
- Idéalement, une fonction peut parer à toutes les situations avec un message d'erreur informatif ou a minima une gestion de l'erreur. Je vous mets à l'aise, ce n'est presque jamais le cas : `plot("a")` n'est pas vraiment explicite et ne point pas le vrai problème.
- Quand vous commencez à avoir un joli paquet de fonctions, il est temps de penser à créer un package, pour vous-même ou pour le monde et dans un premier temps de lire la section consacrée plus loin.

4.7 Un mot sur les méthodes

Les fonctions peuvent avoir un comportement différent selon la classe de l'argument sur lequel elles opèrent. Un exemple trivial est `summary` qui retourne des quantiles quand on lui passe un `numeric`, et un descriptif plus sommaire quand on lui passe un `character` :

```
summary(1:5)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
      1       2       3       3       4       5 
```

```
summary(letters[1:5])
```

[3] Je vous laisse avec Hadley, et chercher du paracétamol

```
   Length     Class      Mode
        5 character character
```

`summary` est une *méthode* plus qu'une fonction, une *façon* de faire une chose (un résumé en l'occurence) sur une variable qui *dépend* de la classe de cette variable.

Voyons d'abord comment déclarer une méthode. Nous allons l'appeler `shout`, pour crier en anglais. Quand `shout` sera appelée sur un `character` elle le passera en majuscules; sur un `numeric` elle élèvera au carré.

```
shout <- function(x) {
  UseMethod("shout", x)
}

shout.default <- function(x) {
  stop("'shout' not defined on this class")
}

shout.character <- function(x) {
  toupper(x)
}

shout.numeric <- function(x) {
  x^2
}

# shout(iris) # try this on your machine
shout("let's be quiet")
```

```
[1] "LET'S BE QUIET"
```

```
shout(1:5)
```

```
[1]  1  4  9 16 25
```

C'est bien joli tout ça, mais n'aurait-on pas pu caler un bon vieux `is.numeric` et tout déclarer dans une fonction. Oui, bravo, vous avez raison (j'ai créé un monstre). Mais pas vraiment (du calme jeune padawan) car les méthodes peuvent faire bien plus que cela, et notamment donner une saveur "orientée objet" à R.

Il existe plusieurs systèmes de déclaration de méthodes en R, et plus largement de programmation orientée objet, on peut citer `S3`, `S4`, `R6`, etc. Le présent document ne s'attardera pas plus sur les méthodes mais les lignes ci-dessous démineront probablement quelques mystères de R.

Pour afficher toutes les méthodes, on utilisera la fonction `methods` qui peut s'appeler soit sur le nom soit sur la classe :

```
  methods("shout")
```

```
[1] shout.character shout.default   shout.numeric
see '?methods' for accessing help and source code
```

```
  methods(class="character")
```

```
 [1] all.equal                as.data.frame            as.Date
 [4] as.POSIXlt               as.raster                coerce
 [7] coerce<-                 formula                  getDLLRegisteredRoutines
[10] Ops                      shout
see '?methods' for accessing help and source code
```

```
  # try: methods("plot") # graphs, graphs, everywhere
  # or even: methods("summary") # !!!
```

Comme on l'a vu précédemment, pour accéder au code d'une méthode on pourrait imaginer qu'il suffise, comme pour toute fonction de taper son nom à la console, sans parenthèses. Ici, le nom ne suffit pas, il faut également le suffixe de classe :

```
  shout # not what I wanted but still, we know it's a method not a bare function
```

```
function(x) {
  UseMethod("shout", x)
}
<bytecode: 0x7f8316bc8d80>
```

```
  shout.character
```

```
function(x) {
  toupper(x)
}
```

```
shout.numeric
```

```
function(x) {
  x^2
}
```

4.8 Un mot sur les packages

C'est un exercice très instructif, valorisant, valorisable et devenu quasiment facile de nos jours.

La référence absolue est *R packages* d'Hadley Wickham, qui a le bon goût d'être libre : https://r-pkgs.org/

Les tables de la loi sont plus dures mais ce sont les lois : https://cran.r-project.org/doc/manuals/r-release/R-exts.html

5 Modélisation statistique : `lm`

Ce document introduit l'aspect le plus basique de modélisation mais qui inspire ou dont dérivent tous les autres : le modèle linéaire aka `lm`.

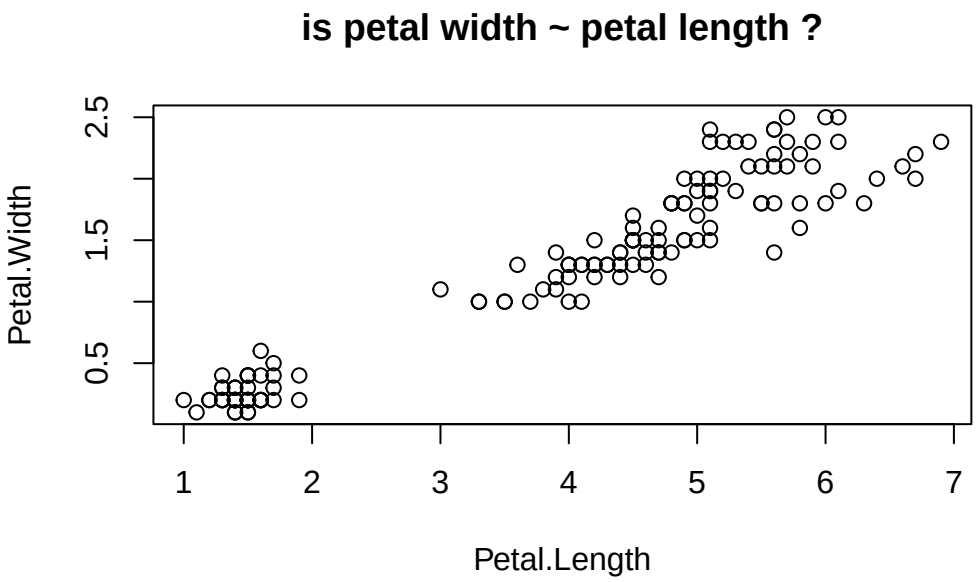
Nous allons nous contenter du jeu de données `iris` et en particulier d'analyser la relation, si elle existe, entre la longueur et la largeur des pétales.

Avant de faire des modèles, faites des graphes !

```
head(iris)
```

```
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          5.1         3.5          1.4         0.2  setosa
2          4.9         3.0          1.4         0.2  setosa
3          4.7         3.2          1.3         0.2  setosa
4          4.6         3.1          1.5         0.2  setosa
5          5.0         3.6          1.4         0.2  setosa
6          5.4         3.9          1.7         0.4  setosa
```

```
plot(Petal.Width ~ Petal.Length, data=iris,
     main="is petal width ~ petal length ?")
```




```
mod <- lm(Petal.Width ~ Petal.Length, data=iris)
mod
```

```
Call:
lm(formula = Petal.Width ~ Petal.Length, data = iris)

Coefficients:
 (Intercept)  Petal.Length
     -0.3631        0.4158
```

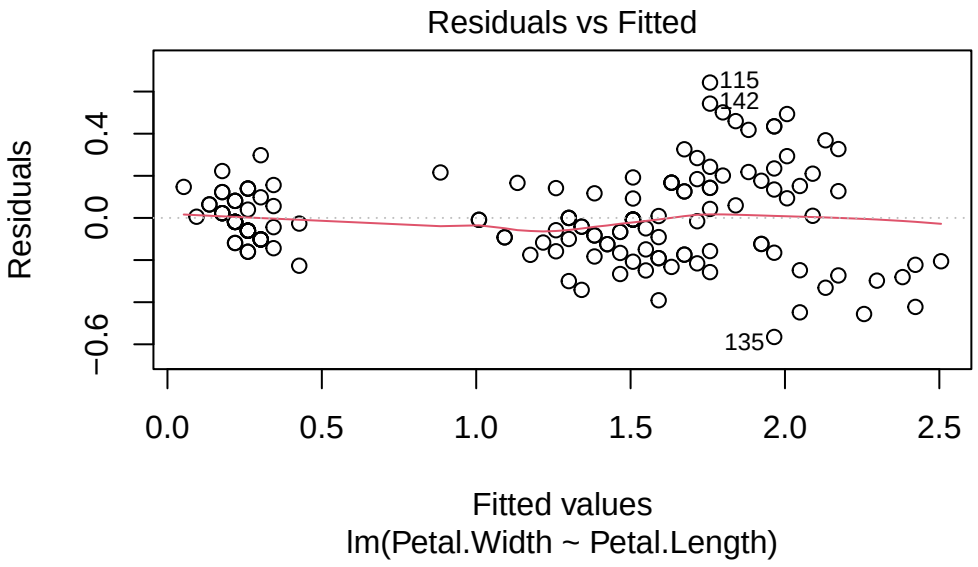
```
summary(mod)
```

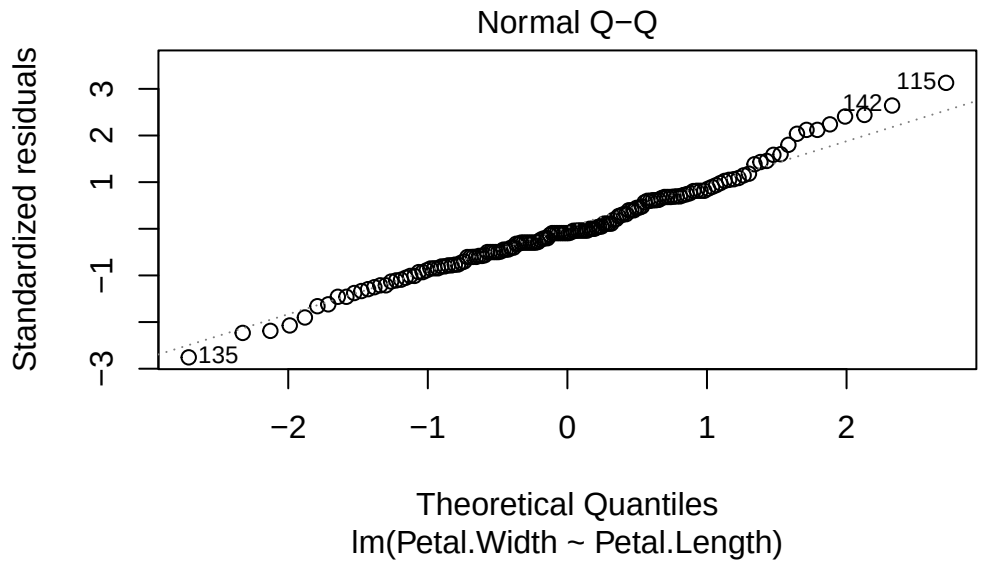
```
Call:
lm(formula = Petal.Width ~ Petal.Length, data = iris)

Residuals:
     Min       1Q   Median       3Q      Max
-0.56515 -0.12358 -0.01898  0.13288  0.64272
```

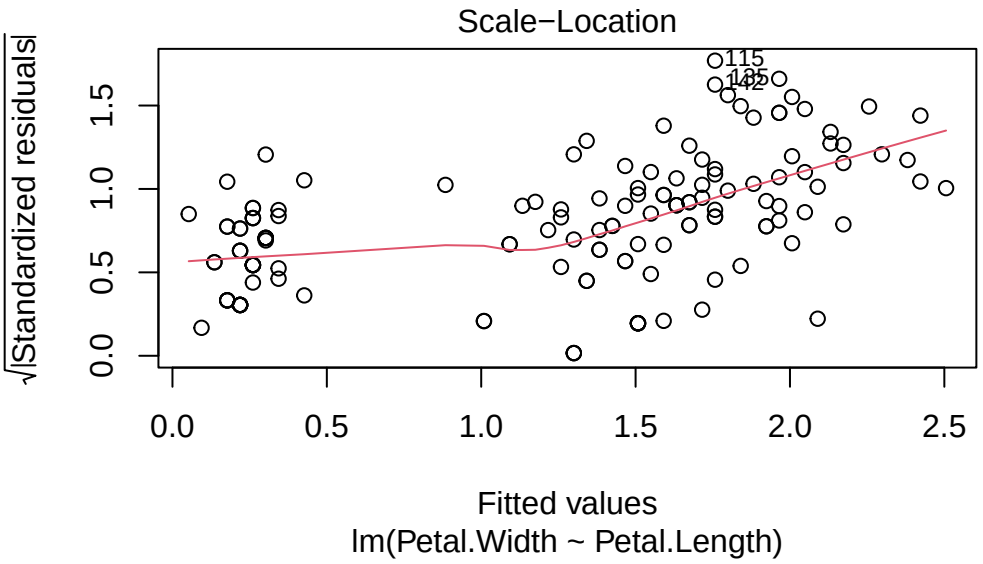
```
Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept)  -0.363076   0.039762  -9.131  4.7e-16 ***
Petal.Length  0.415755   0.009582  43.387  < 2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.2065 on 148 degrees of freedom
Multiple R-squared:  0.9271,    Adjusted R-squared:  0.9266
F-statistic:  1882 on 1 and 148 DF,  p-value: < 2.2e-16
```

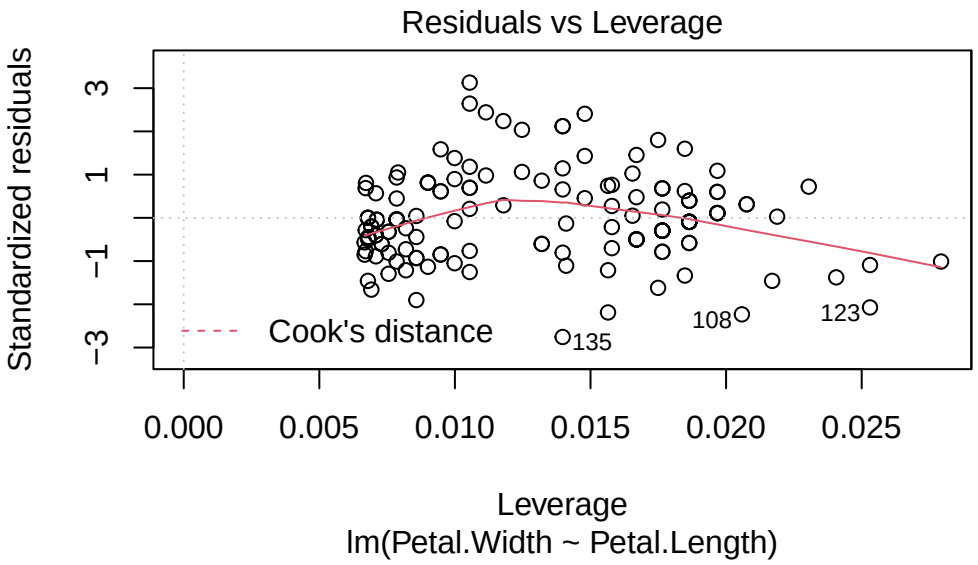
5.0.1 formula

```
plot(mod)
```




Normal Q−Q
Standardized residuals
−3
−1
1
2
3
115
142
135
−2
−1
0
1
2
Theoretical Quantiles
lm(Petal.Width ~ Petal.Length)


Scale−Location
√|Standardized residuals|
0.0
0.5
1.0
1.5
115
135
142
0.0
0.5
1.0
1.5
2.0
2.5
Fitted values
lm(Petal.Width ~ Petal.Length)


Residuals vs Leverage
Standardized residuals
3
1
−1
−3
0.000
0.005
0.010
0.015
0.020
0.025
Leverage
lm(Petal.Width ~ Petal.Length)
Cook's distance
135
108
123

```
#predict()
```

6 Ceci n'est pas qu'un opérateur : %>% et `magrittr`

Il est peu dire que cet opérateur[1] a révolutionné R, lorsqu'il y a été importé, d'abord dans le package `magrittr` sous sa forme `%>%`. Il est désormais inclus dans le R de "base" sous sa forme `|>` mais nous n'utiliserons que la version historique `%>%`, que je trouve plus lisible, plus facile à taper (`<Maj> + <Ctrl/Cmd> + <M>` dans RStudio) et parce que les années aidant, je deviens conservateur.

L'idée du pipe est issue de la composition de fonctions en mathématiques. Plutôt que d'écrire :

`h(g(f(x)))` on peut déplier cet emboitement de fonctions et écrire `(h  g  f)(x)`

En langage R, plutôt que d'écrire `h(g(f(x)))` on écrira : `x %>% f() %>% g() %>% h()`[2]. Cette écriture est non seulement plus lisible mais elle se lit également de gauche à droite, dans le sens conventionnel de notre partie du monde.

6.1 %>% vs |>

D'abord introduit par le package `magrittr` le forward pipe est désormais dans le R "de base" depuis la version 4.1.0.

Le pipe `|>` tend désormais à être préféré à `%>%` comme on le lira sur le (blog du tidyverse)[, très complet)[https://www.tidyverse.org/blog/2023/04/base-vs-magrittr-pipe/]. Ce dernier mérite néanmoins, avec ses autres copains de `magrittr`, d'être détaillé.

Si la vignette de `magrittr` est très bien faite (`vignette("magrittr")`), j'en livre ici une introduction rapide.

[1] À prononcer à l'anglaise hein : "payepeu"
[2] Vous pouvez également omettre les parenthèses si vos fonctions sont passées sans argument.

6.2 %>%

Dans les grandes lignes, %>% et |> fonctionnent de la même façon. J'ai tendance à continuer d'utiliser %>% qui est chargé automatiquement par le tidyverse. Pour les autres opérateurs, ou si vous n'utilisez pas le tidyverse vous pouvez en disposer avec :

```
library(magrittr)
```

Venons en aux faits.

```
runif(100) %>% mean()
```

```
[1] 0.4823802
```

6.3 Le . pour customiser le forward

6.4 %T>%

Utilisation ?magr

6.5 '%$%

6.6 %<>%

Utilisation Critiques

7 Importer ses données

7.1 Bonnes pratiques

Dans une vaste majorité des cas, vous remplissez et gérez vos données à partir d'un tableur. Que vous utilisiez *MS Excel*, *Open Office*, *Libre Office*, *Google Sheets* ou même un bloc notes, je ne saurai trop vous conseiller de respecter les quelques règles ci-dessous qui vous faciliteront la vie ensuite.

> Soignez vos données sources

Je liste ci-dessous les grands principes, issus de l'excellent article par Karl Broman et Kara Woo[1]. Je les amende à la marge mais allez de ce pas lire cet article !

1. Bien nommer votre fichier

- `data_2024-04-23` est une excellente base. Un horodatage est utile si d'autres versions venaient à advenir, vous n'auriez que le nom du fichier source à changer côté R.
- faites des sauvegardes, en plusieurs lieux, y compris de vos vieux fichiers.

2. Bien nommer vos colonnes

- Je vous conseille l'anglais et les minuscules partout.
- Utilisez des noms concis (vous allez les taper souvent) mais informatifs (on ne peut pas se permettre de malentendu).
- Pas d'espaces pour les noms de colonnes, utilisez `_` à la place. Par exemple : `arm_length`, `arm_width`, `leg_length`, etc.
- Vous pouvez utiliser un patron pour les "familles" de variables par exemple = `geo_city`, `geo_country`, `geo_lat`, `geo_lon`. Cela vous permettra de les sélectionner facilement avec `dplyr` et `tidyselect`.

3. Soyez homogènes partout, tout le temps

- Pour chaque colonne ayant un système de nommage, par exemple un facteur, soyez homogènes. Si vous sexez des mésanges, restez dans l'ensemble `{F, M, NA}`. Chaque mésange aura une (et une seule) valeur.

[1] https://doi.org/10.1080/00031305.2017.1375989

- Idem pour les colonnes décrivant des variables continues, choisissez une unité (du système international) et tenez vous-y ! Par exemple : `182.1` (cm); `174.2` (cm); etc.
- La validation des données de votre tableur est utile, à la saisie comme après.
- Ces recommandations valent particulièrement pour les "ID" partagés entre fichiers. Si vous avez opté pour un système, tenez vous en à ce système (ou changez tout).
- Utilisez le `.` comme séparateur décimal

4. Une seule valeur par cellule

- Les commentaires doivent, éventuellement, être faits ailleurs que dans la cellule. Pas non plus de codage par le formatage (couleur de cellule ou de texte, graisse et italique, etc.)
- Pas non plus de cellules vides. Si vous avez des données manquantes, il est préférable de l'expliciter, par exemple avec `NA`.

5. Exporter en fichier texte

- Exportez votre tableur en `.txt` ou `.csv` (qui est fondamentalement du texte) ou n'importe quel autre fichier pouvant s'ouvrir avec un simple bloc-notes.
- L'import direct de fichier `xlsx` (et variantes) est possible mais non souhaitable.

7.2 Import

Cette section utilise le petit jeu de données "data.csv" que vous pouvez télécharger ici.

N'hésitez pas à créer votre propre jeu de données ou bien à en utiliser un que vous auriez sous la main.

Je commence par les fonctions de base de R mais sachez que RStudio fait très bien le job, alors ne restez pas trop bloqués sur la section `read.table`, surtout là pour la démystifier.

7.2.1 `read.table`

La fonction "historique" pour l'import de données est `read.table` qui a provoqué des tonneaux de larmes et qui, pour une entrée en matière est, il est vrai, un peu aride. Et pourtant, il n'y a que quelques arguments à préciser si votre fichier tableur, devenu texte, suit les conseils de la section précédente.

Le principal point d'achoppement est peut-être le premier argument (`file`) qui définit le chemin vers votre fichier. Un chemin peut être spécifié de façon relative ou absolue :

- un chemin absolu a comme référence la racine de votre système de fichiers (`C:\` ou `~` selon que vous soyiez respectivement sur Windows ou sur un OS plus décent)
- un chemin relatif a comme référence le dossier de travail actuel de R, accessible avec `getwd()` et modifiable par `setwd()`.

Si par exemple, j'ai récupéré mon fichier `data.csv` (voir plus haut) et qu'il gît sur mon Bureau (j'ai un Mac) le chemin absolu serait : `"~/Desktop/data0.csv"` ; Alors que mon chemin relatif serait relatif à : `/Users/vbonhomme/Research/R-pour-ma-grand-mere`.

Mon conseil est le suivant : utilisez les "Projets" RStudio. L'icône bleue en haut à droite ou "File > New project". Puis, dans ce dossier qui contiendra vos scripts, vos données, vos résultats, votre papier, etc. créez un dossier "data". De cette façon, votre *working directory* sera votre dossier général et ils vous suffira d'utiliser ce type de chemin : `data/data.csv`.

`read.table` accepte ensuite d'autres arguments, les plus souvent utilisés étant :

- `sep` pour définir le caractère qui définit le saut de champ (de colonne). Si vous utilisez des tabulations, ce que je ne conseille pas, vous pouvez le mentionner avec `\t` ;
- `dec` : idem pour le marqueur de la décimale (idéalement `.`, la valeur par défaut, ou `,` que je vous déconseille)
- `header` : `TRUE`/`FALSE` selon que vos colonnes aient des entêtes ou non ;
- `skip` : pour sauter, éventuellement, des lignes en début de fichier.

Avec `data.csv` on peut donc importer son contenu via :

```
x <- read.table("data/data.csv", sep=";", header=TRUE)
x
```

```
        prénom  sexe stature     R
1   Hildegarde femme     168  TRUE
2 Jean-Jacques homme     170 FALSE
3       Victor autre     184 FALSE
```

7.2.2 `RStudio`

La commande "Environment > Import dataset" de RStudio vous simplifie grandement cette tâche avec un aperçu en temps réel de l'import via le package `readr`. Une fois que vous êtes satisfait · e, vous pouvez copier-coller le code qui vous sera retourné à l'issu de l'import. Royal !

7.2.3 `readr`

Le package `readr` du tidyverse fait la même chose mais vous donne plus de contrôle sur l'import. Le package est bien documenté ici.

7.3 Export

Vous pourriez aussi avoir besoin d'écrire des fichiers `.csv`, `.txt`, etc. Sachez qu'à côté de `read.table`, `read_csv`, etc. existent leur cousines destinées à l'écriture : `write.table`, `write_csv` qui fonctionnent de la même façon.

Pour avoir un fichier qui s'ouvre "facilement" dans un tableur je vous conseille :

```
write.table(votre_objet, row.names=FALSE, col.names=TRUE)
```

7.4 `.rda`

Si vous voulez sauver/lire des fichiers R, dans le format natif de R, les commandes `save`/`load` sont vos amies. Le format `rda` (ou `.RData`, c'est la même chose) est compressé et optimisé. Vous pouvez ainsi sauver un jeu de données comme cela :

```
save(iris, file="iris.rda")
load("iris.rda")
```

Cette approche est par exemple très utile dans le cas d'un calcul qui serait très long dans un script (qui se terminerait par un `save`) et serait rappelé à la vie par un autre script (qui commencerait par un `load`).

7.5 Autres I/Os

Pour vous ouvrir les shakras, sachez qu'il existe quantité d'autres approches pour importer/exporter des données.

- `readlines` lit des fichiers textes bruts
- le package `feather` utilise un format performant et interopérable de données avec Python
- le package `xlsx` permet de lire/écrire des fichiers `xlsx` (berk)
- le package `httr` permet de travailler avec l'http et notamment de faire des requêtes `GET` et `POST`
- `yaml` permet de parser des fichiers `.yaml`

8 Manipulation de données avec `dplyr`

8.1 `tibble`

8.2 `rename`

8.3 `select`

8.4 `tidyselect`

8.5 `filter`

8.6 `mutate`

8.7 `transmute`

8.8 `summarise`

8.9 `group_by`

8.10 `rowwise`

8.11 `join`

8.12 `nest`

8.13 `cheat sheet`

9 Nettoyer ses données avec `tidyr`

The goal of tidyr is to help you create tidy data. Tidy data is data where:

- Each variable is a column; each column is a variable.
- Each observation is a row; each row is an observation.
- Each value is a cell; each cell is a single value.

9.1 `tibble`

9.2 `pivot_longer/pivot_wider`

9.3 `separate/unite`

9.4 `expand`

better expand.grid

10 Graphiques avec `ggplot2`

10.1 Rationale

Il est peu dire que le package `ggplot2` a changé la face de la représentation graphique en R.

Le package de base, `graphics`[1] permet certes de tout faire mais laborieusement. D'autres packages (`lattice` par exemple) permettent une création plus proche de l'utilisateur · trice, moins proches des "primitives" (segments, points, etc.) graphiques.

La force de `ggplot2`, d'abord écrit par Hadley Wickham, est d'implémenter la *Grammar of Graphics* de Leland Wilkinson qui dissocie les données de leurs représentations, de façon déclarative. Nous déclarerons ainsi à un jeu de données, des modes de représentation (des `geom_`) qui s'appuieront eux-mêmes sur des déclarations d'axes des abscisses, des ordonées, des couleurs, des tailles etc. (via `aes`), nous préciserons les axes (`coord_`), les paramètres de style `theme`.

Nous obtiendrons finalement un graphe que nous pourrons modifier à soit et qui sera même, en soit, une machine à faire d'autres graphes.

`ggplot2` s'installe classiquement avec `install.packages("ggplot2")` mais il est compris dans le tidyverse que vous avez du installer précédemment avec `install.packages("tidyverse")`.

Pour charger `ggplot2`, il suffit de taper `library(ggplot2)` ou encore `library(tidyverse)`.

10.2 Un premier graphe

`ggplot2` travaille sur des `data.frame`s (ou des `tibble`s mais c'est la même chose). Tout jeu de données qui n'est pas un `data.frame` sera converti ou tenté de l'être avec `fortify`. Le plus simple pour ne pas avoir de surprises étant de convertir vos données en un `data.frame` en bonne et due forme, vous-mêmes.

Nous allons utiliser `iris` qui est déjà un `data.frame` mais que nous allons, pour la cosmétique, convertir en `tibble` et en renommer les colonnes

[1] `?graphics`

```r
library(tidyverse)
```

```r
iris2 <- iris %>%
  as_tibble() %>%
  rename(pl=Petal.Length, pw=Petal.Width,
         sl=Sepal.Length, sw=Sepal.Width, sp=Species)
iris2 # iris, as tibble and with more compact column names
```

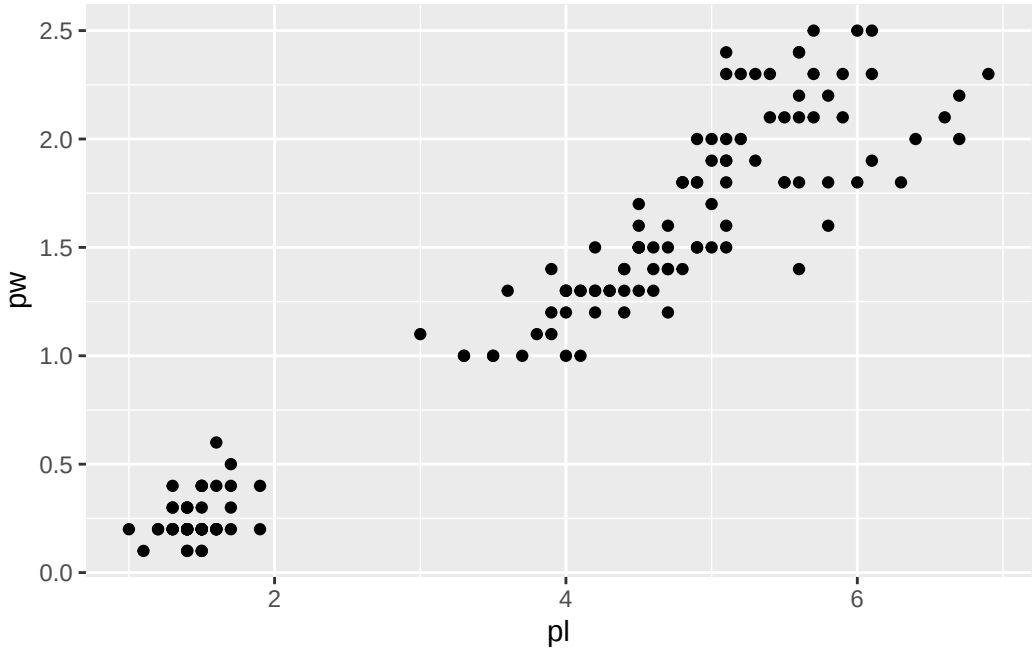
```
# A tibble: 150 x 5
      sl    sw    pl    pw sp
   <dbl> <dbl> <dbl> <dbl> <fct>
 1   5.1   3.5   1.4   0.2 setosa
 2   4.9   3     1.4   0.2 setosa
 3   4.7   3.2   1.3   0.2 setosa
 4   4.6   3.1   1.5   0.2 setosa
 5   5     3.6   1.4   0.2 setosa
 6   5.4   3.9   1.7   0.4 setosa
 7   4.6   3.4   1.4   0.3 setosa
 8   5     3.4   1.5   0.2 setosa
 9   4.4   2.9   1.4   0.2 setosa
10   4.9   3.1   1.5   0.1 setosa
# i 140 more rows
```

Les étapes suivantes sont toujours les mêmes :

1. déclarer le `data.frame`à utiliser (`iris2` ici)
2. déclarer quelles colonnes doivent être utilisées (nous voulons la largeur de pétale `pw` en fonction, en `y` en somme, de la longueur de pétale `pw`).
3. déclarer le mode de représentation (ici un nuage de points `geom_point`, mais quantité d'autres `geom` existent comme les histogrammes, les boxplots, etc.)

`ggplot2` utilise l'opérateur `+` pour assembler ces différentes couches, après la fonction `ggplot()` qui initie le graphe.

```r
iris2 %>%
  ggplot() +
  aes(x=pl, y=pw) +
  geom_point()
```



Vous trouverez parfois la déclaration du `data.frame` `aes` au sein de `ggplot()` comme cela : `ggplot(iris2, aes(x=pw, y=pl))`. J'ai tendance à tout éclater comme ci-dessus, mais à vous de voir.

10.3 Un deuxième `geom` et un sacrifice

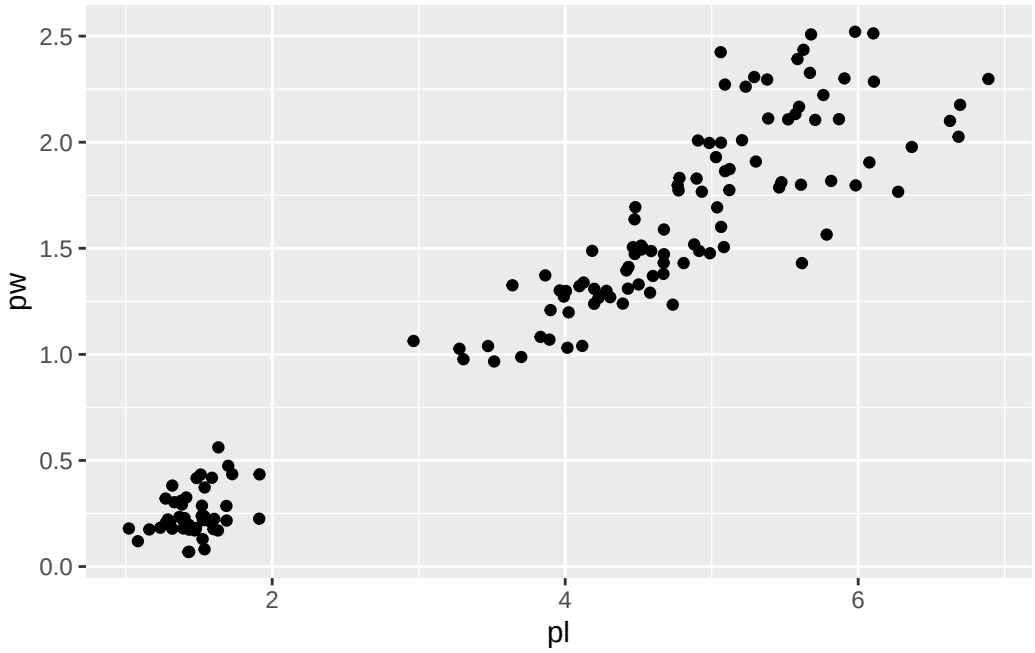
Si vous vous amusiez à compter le nombre de points représentés, ou plus intelligemment `iris2 %>% select(pl, pw) %>% unique() %>% nrow()`, vous réaliseriez que 42 points ne sont pas visibles. La faute n'en incombe nullement à `ggplot2` mais à la précision des instruments de mesure au moment de l'acquisition de données ! Plus exactement, ils sont visibles mais superposés.

Cette mise en garde nous donne l'occasion de présenter un deuxième `geom` qui introduit suffisamment de bruit dans les données pour que tous les points soient représentés tout en gardant la relation générale entre les deux variables `pw` et `pl`. En d'autres termes, nous sacrifions l'exactitude pour gagner en fidélité du nombre de points effectivements présents.

La représentation visuelle de l'information quantitative est souvent affaire de sacrifices et elle est un domaine de recherche à part entière. Offrez-vous ou faites vous offrir la bibliographie complète d'Edward Tufte !

Voilà un `iris2` avec ses 150 iris :

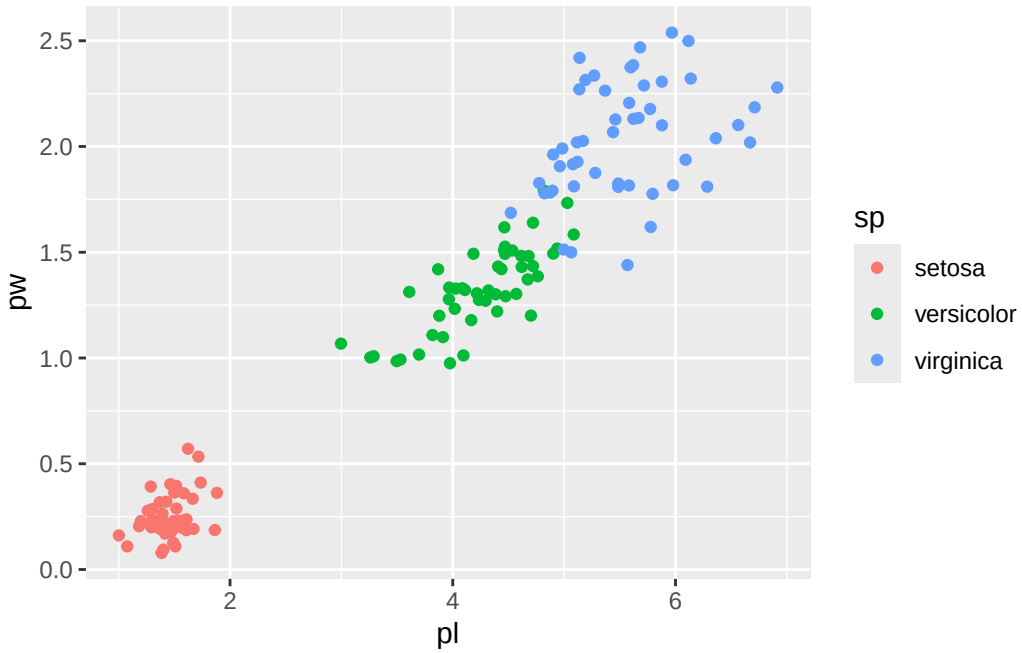
```
iris2 %>%
  ggplot() +
  aes(pl, pw) +
  geom_jitter()
```



10.4 `aes` : d'autres variables sur le même graphe

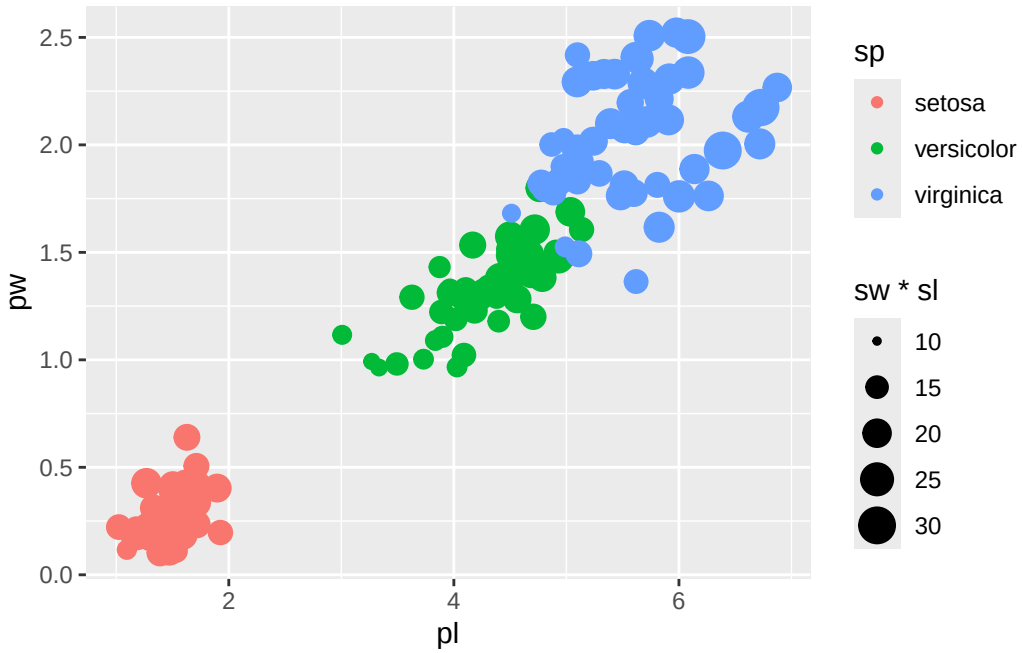
`aes` règle les *aesthetics* de votre `ggplot`. Si par exemple vous voulez associer chaque espèce à une couleur, il vous suffit de rajouter `col=sp` dans `aes()` et la légende est automatiquement générée :

```
iris2 %>% ggplot() +
  aes(pl, pw, col=sp) +
  geom_jitter()
```



Dans le même esprit, si vous vouliez rajouter les dimensions des sépales comme encodage des tailles de point, c'est dans `aes()` que ça se passe. Vous pouvez même créer à la volée, dans `aes` même, des opérations sur les colonnes de `iris2`. Une "pseudo-aire", faisant fi de la forme mais intégrant la longueur et la largeur des sépales peut se créer comme suit :

```
iris2 %>% ggplot() +
  aes(pl, pw, col=sp, size=sw*sl) +
  geom_jitter()
```

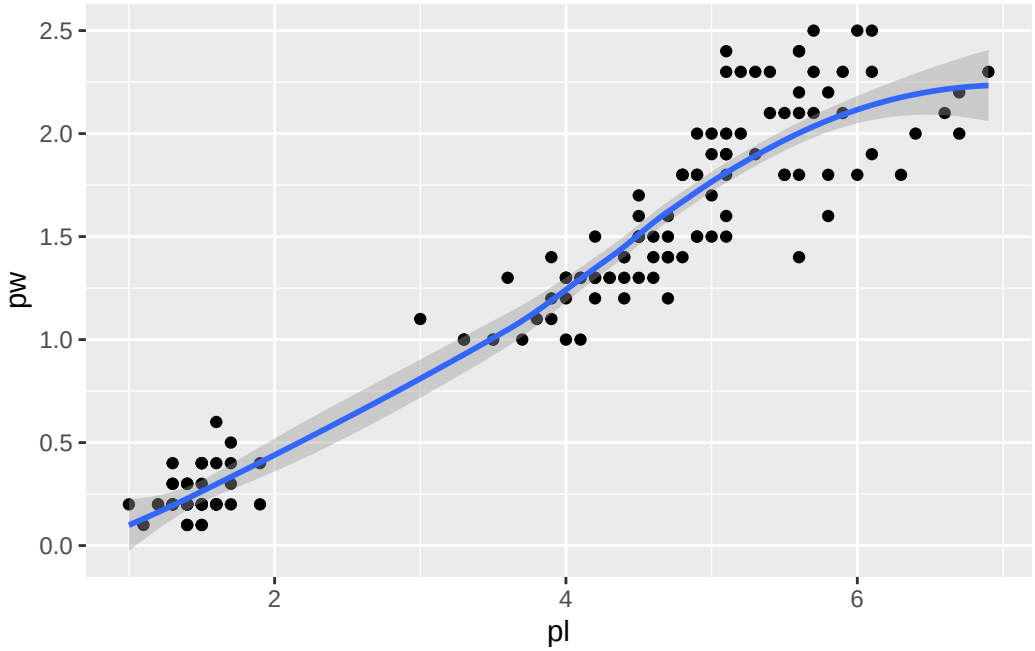


10.5 Tendances et modèles statistiques

La production de graphes est centrale dans l'exploration de données et `ggplot2` en fait une tâche simple. `stat_smooth` va ajouter une courbe de tendance aux données représentées.

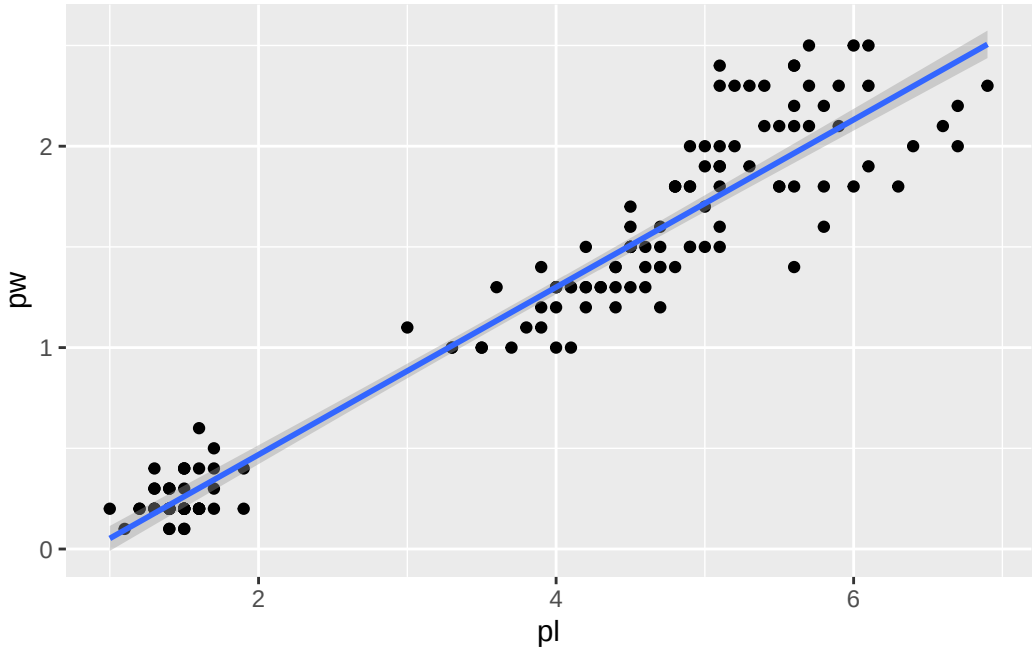
```
gg <- iris2 %>% ggplot() + aes(pl, pw) + geom_point()
gg + stat_smooth()
```

```
`geom_smooth()` using method = 'loess' and formula = 'y ~ x'
```



Deux choses. D'abord, `ggplot2` vous avertit que la courbe ajoutée utilise `loess`, c'est à dire un ajustement polynomial local calé sur `y~x`. Si ces messages vous ennuient, il vous suffit de préciser à `geom_smooth` d'utiliser ces modèles (`geom_smooth(method="loess", formula="y~x")`). Ensuite, peut être vous aviez plutôt en tête un bon vieux modèle linéaire avec `lm` plutôt que ce `loess` certes flatteur mais qui ne sied peut-être pas à votre esprit naturellement parcimonieux. Bonne nouvelle, c'est très simple :

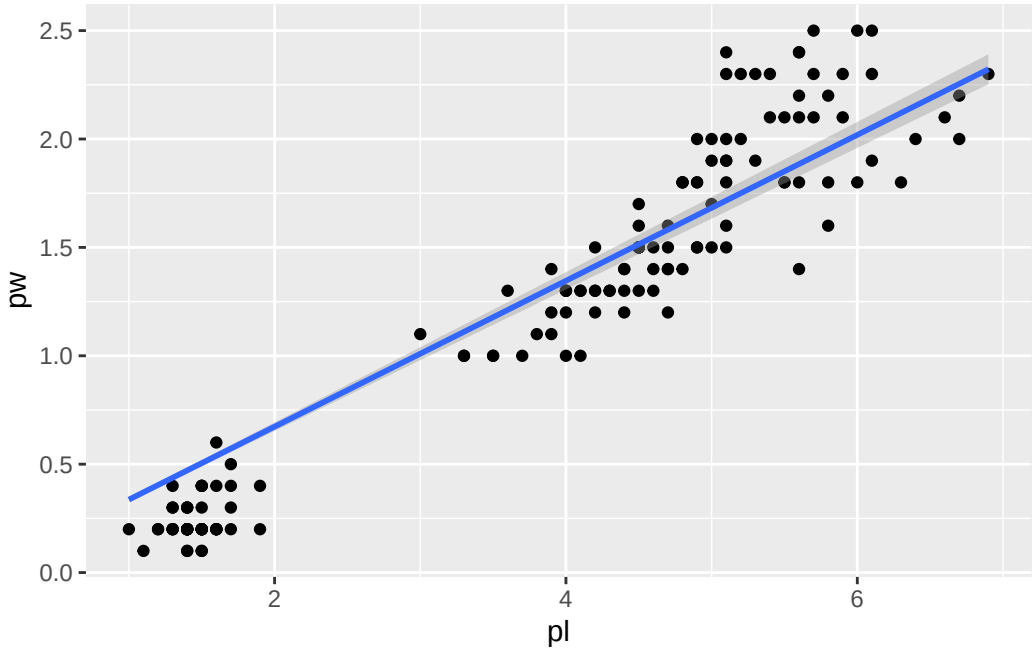
```
gg + geom_smooth(method="lm", formula="y~x")
```



Après avoir contemplé ce graphe, vous vous dites que ce modèle linéaire, moyennant que la longueur de pétale soit inférieure à 2 vous prédirait des largeurs de pétales négatives. Ce qui vous en conviendrez n'a guère de sens, ni pour un · e biologiste, ni même pour un · e statisticien · ne (c'est dire !).

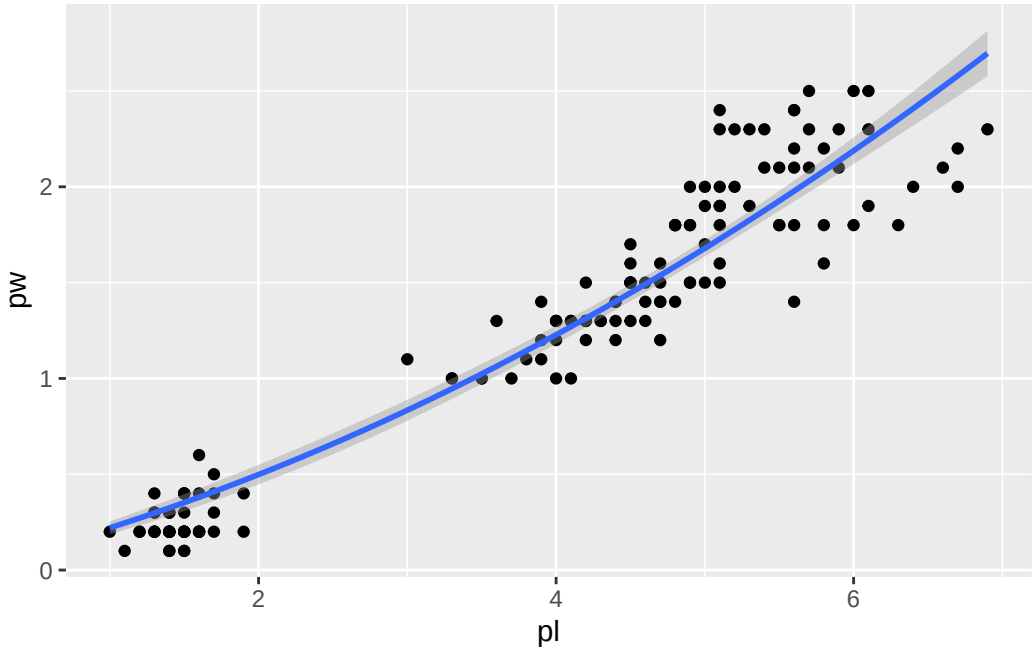
Vous l'avez deviné si vous avez déjà utilisé `lm()`, ça se passe dans dans la spécification de la `formula`. Pour forcer le modèle à ne pas avoir d'intercept (ou plutôt un intercept égal à 0), vous pouvez spécifier `-1` dans la formule. Je sais que ce n'est pas très intuitif mais vous consulterez avec volupté `?formula` pour plus d'informations.

```
gg + geom_smooth(method="lm", formula="y~x-1")
```



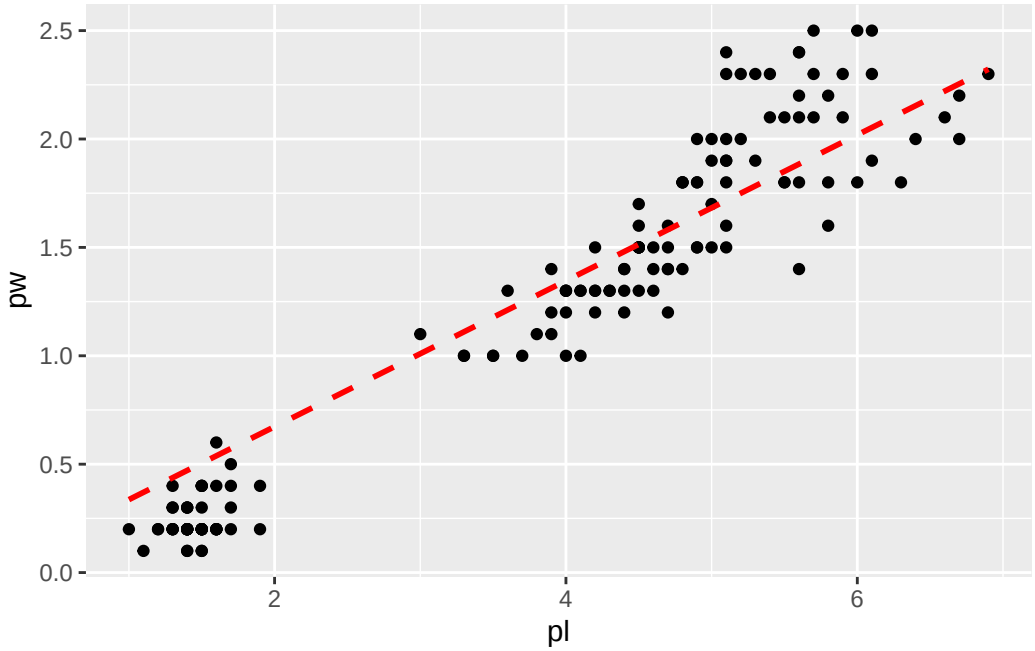
Dans le même esprit si nous voulions forcer la relation à être quadratique, nous aurions pu :

```
gg + geom_smooth(method="lm", formula="y~I(x^2)+x-1")
```



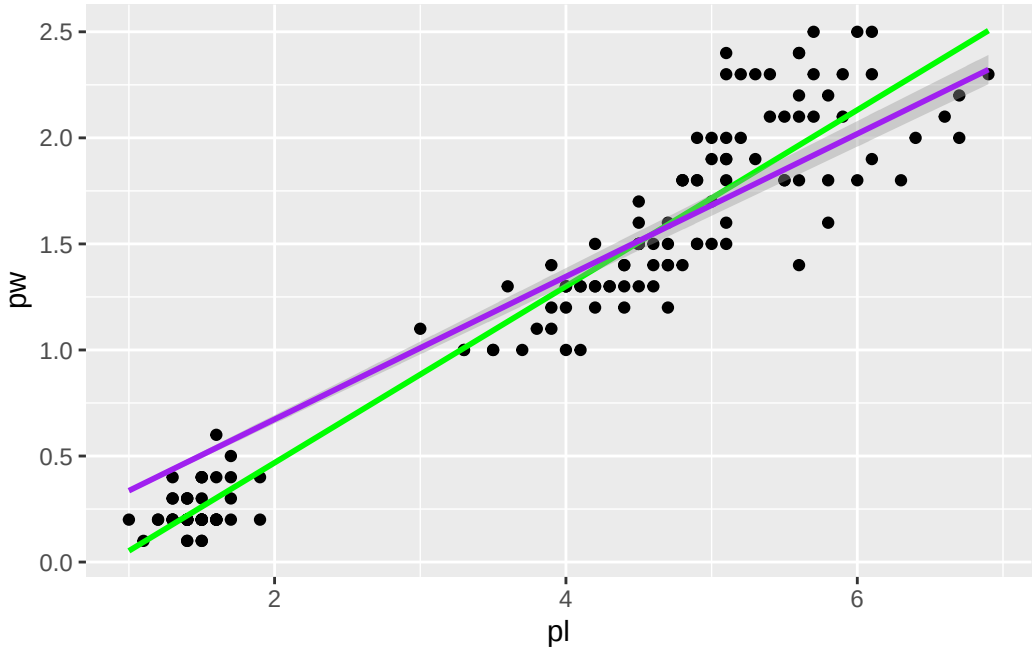
La couleur de la droite (ou de la courbe) de régression, ainsi que la présence ou non d'un intervalle de confiance peuvent se préciser, toujours dans `geom_smooth`. Par exemple :

```
gg + geom_smooth(method="lm", formula="y~x-1", se=FALSE, col="red", linetype="dashed")
```



Ces paramètres `se` et `colour` sont documentés dans `?geom_smooth` et ne s'appliquent qu'à ce `geom`, pas à l'échelle globale du `ggplot` ni même des autres `geoms`. Par exemple :

```
gg +
  geom_smooth(method="lm", formula="y~x", col="green", se=FALSE) +
  geom_smooth(method="lm", formula="y~x-1", col="purple") # no intercept
```



Un petit rappel à la doxa ne faisant jamais de mal :

> ce n'est pas parce que produire des droites/courbes de tendance est facile que vous êtes autorisé · e à le faire

Si votre modèle n'est pas "significatif", vous n'êtes pas vraiment autorisé · e à le représenter. Naturellement vous pouvez le faire en cachette mais dans un article vous allez vous faire tomber dessus par le · a post-doc qui se tape la review !

Il n'y a aucun moyen à ma connaissance de le faire avec `ggplot2`, ni même de récupérer les modèles créés en interne par ce dernier mais vous aurez, quoiqu'il en soit, besoin de les explorer par ailleurs. Recréeons le modèle sans intercept :

```
mod <- lm(pw~pl-1, data=iris2)
mod
```

```
Call:
lm(formula = pw ~ pl - 1, data = iris2)

Coefficients:
    pl
0.3365
```

```r
summary(mod)
```

```
Call:
lm(formula = pw ~ pl - 1, data = iris2)

Residuals:
     Min       1Q   Median       3Q      Max
-0.48446 -0.24366 -0.13699  0.08642  0.68379

Coefficients:
   Estimate Std. Error t value Pr(>|t|)
pl 0.336511   0.005063   66.46   <2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.2573 on 149 degrees of freedom
Multiple R-squared:  0.9674,    Adjusted R-squared:  0.9672
F-statistic:  4417 on 1 and 149 DF,  p-value: < 2.2e-16
```

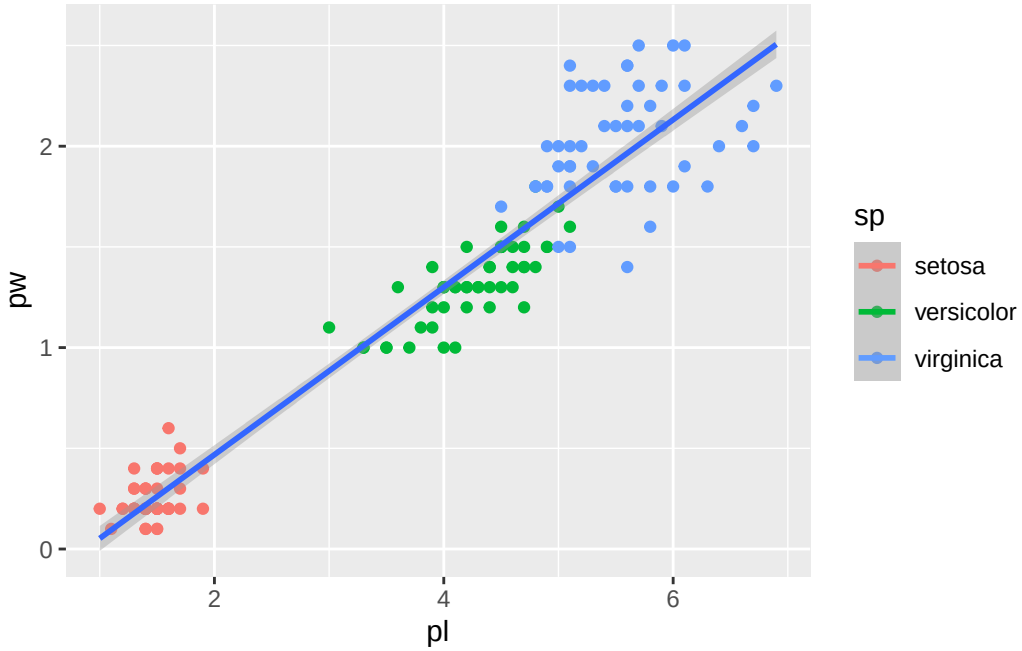
Sans surprise, le modèle linéaire est "significatif" et le(s) R2 excellents. Vous pouvez donc représenter ces modèles.

Le clou du spectacle désormais : si nous avons déclaré une couleur par espèce d'iris, le reste suit. Le reste ici étant un modèle linéaire. Tadan !

```r
gg <- iris2 %>%
  ggplot() +
  aes(pl, pw, col=sp) +
  geom_point() +
  geom_smooth(method="lm", formula="y~x")
```

Notons que si vous ne voulez PAS que le reste suive, en termes plus formels que `geom_smooth()` n'hérite pas des paramètres globaux de `aes`, il vous suffit de déclarer un `aes` local au paramètre `mapping`, le premier argument de tous les `geom` :

```r
iris2 %>%
  ggplot() +
  aes(pl, pw, col=sp) +
  geom_point() +
  geom_smooth(mapping = aes(col=NULL), method="lm", formula="y~x")
```

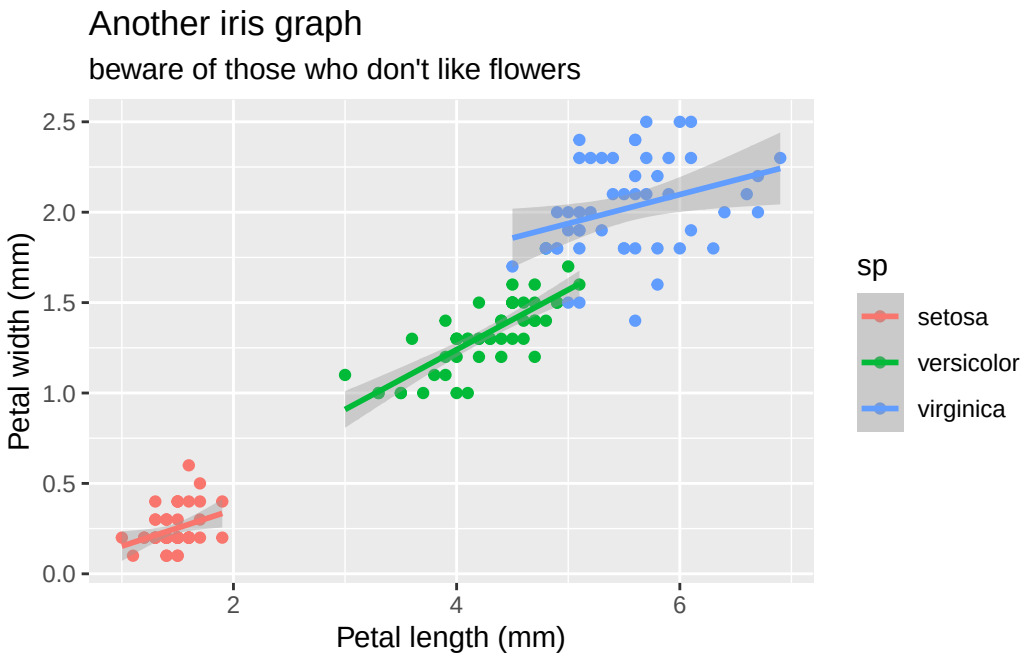


10.6 Interlude cosmétique : `labs`, `theme` et `scale_`

Puisque je sens que vous commencez à tomber éperdument amoureux · se de `ggplot2`, il est temps d'enfoncer le clou avec un peu de cosmétique.

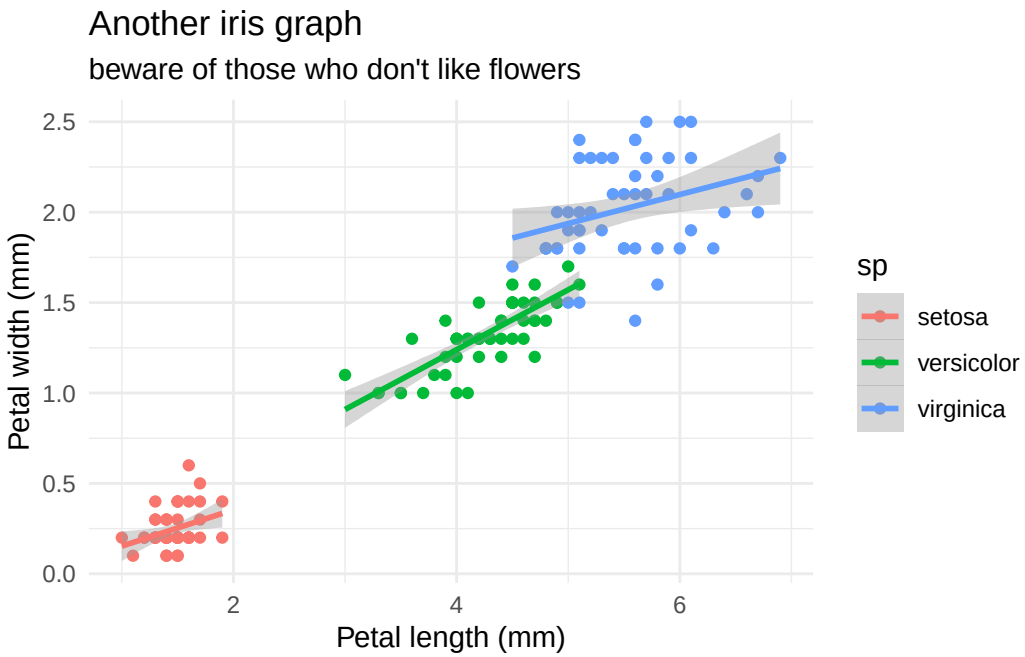
Le nom des axes ainsi que les titres et sous-titres sont faciles à modifier :

```
gg <- gg +
  xlab("Petal length (mm)") +
  ylab("Petal width (mm)") +
  labs(title="Another iris graph", subtitle = "beware of those who don't like flowers")
gg
```



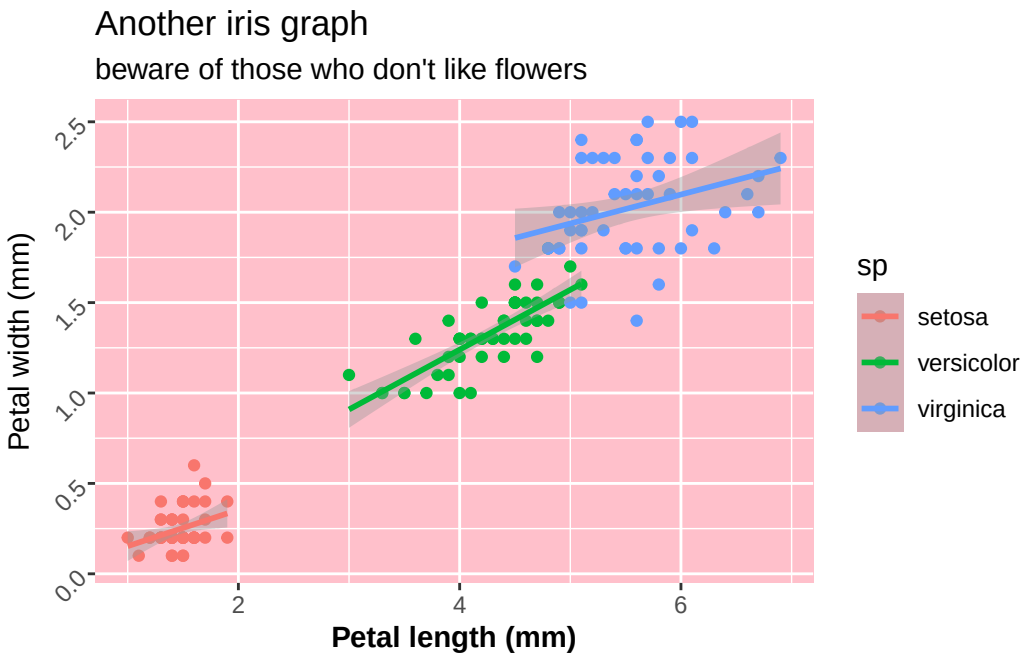
Le "thème" général des graphes peut être modifié. Vous pouvez essayer les autres, tapez `theme_` puis pressez `<Tab>`.

```
gg + theme_minimal()
```



Chaque élément individuel peut être modifié. Sachez que c'est un terrain glissant, susceptible de vous désociabiliser. Avec un peu de sueur on fait exactement ce que l'on veut, y compris pire :

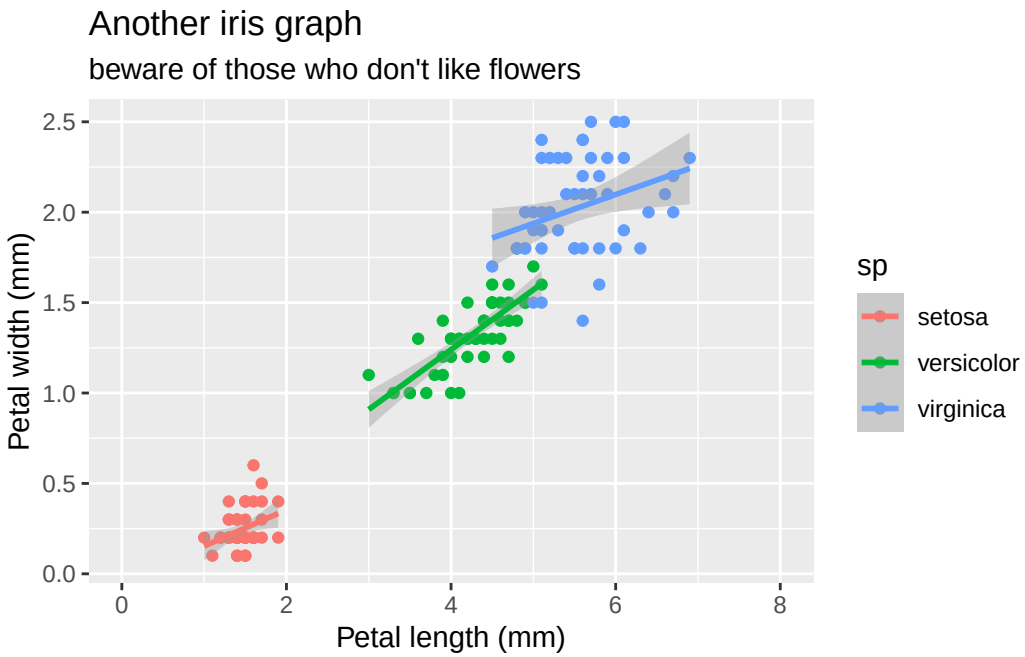
```
gg + theme(axis.title.x = element_text(face="bold"),
           axis.text.y = element_text(angle=45),
           panel.background = element_rect(fill="pink"))
```



Si l'un des thèmes vous convient et/ou que vous en avez customisé un, vous pouvez le définir pour tous les graphes avec `theme_set`.

Enfin, et l'on s'écarte un peu de la pure cosmétique, vous pouvez ajuster les systèmes de représentation des axes, à commencer par leurs limites :

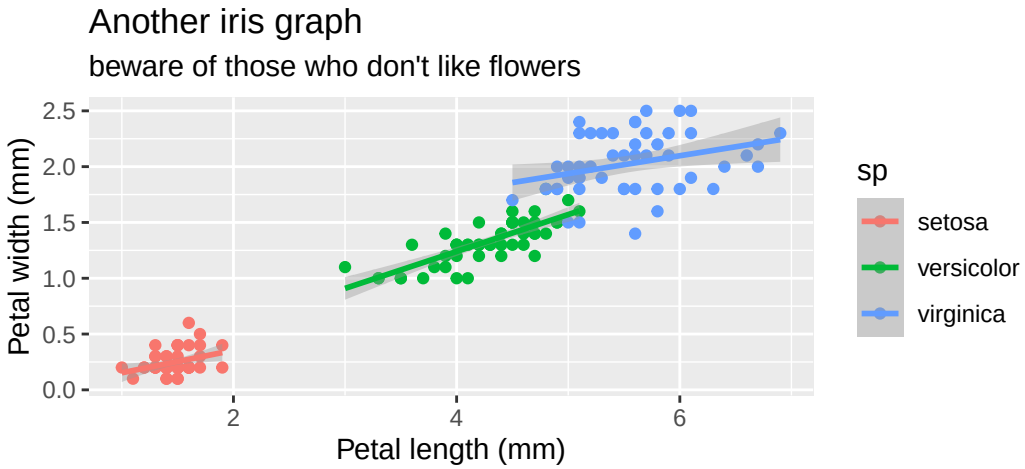
```
gg + scale_x_continuous(limits=c(0, 8))
```



Pour la seule étendue, on préferera `xlim()` et `ylim()` mais la fonction `scale_x_continuous` fait beaucoup plus.

Vous pouvez même changer le système de coordonnées. Si par exemple vous représentez un plan factoriel d'ACP, vous aurez besoin de contraindre l'aspect `y/x` de telle façon qu'un centimètre sur `x` à l'écran, représente également un centimètre sur `y`.

```
gg + coord_equal()
```

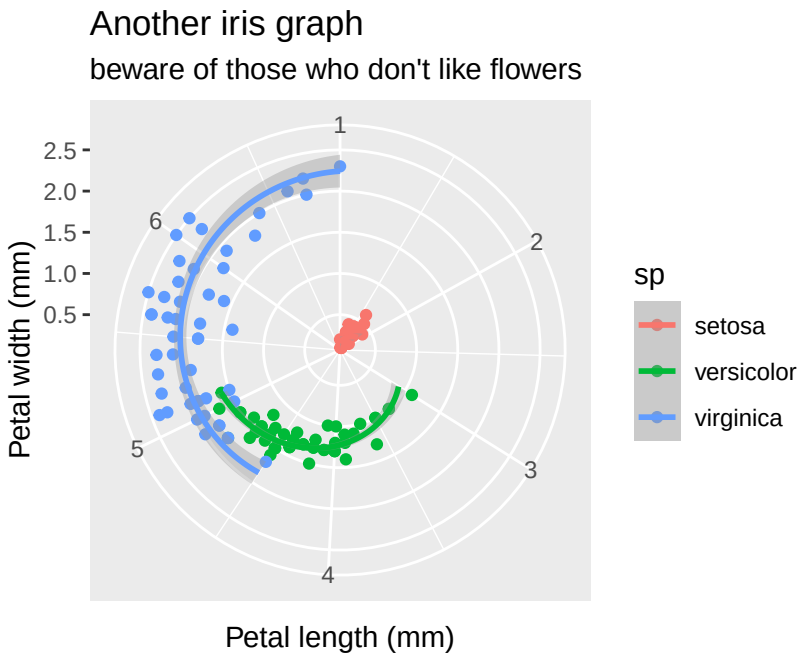


Les régressions sont moins flatteuses [2] mais on a bel et bien un plan euclidien.

Enfin, vous pétez les plombs et vous vous lancez dans l'étude des vents, ou quoique ce soit d'autre avec des coordonées polaires. La famille `coord_*` vous permet de changer carrément le système de coordonées :

```
gg + coord_polar()
```

[2] petite filouterie : si vous voulez convaincre visuellement, arrangez vous pour que vos regressions aient un angle de 45°



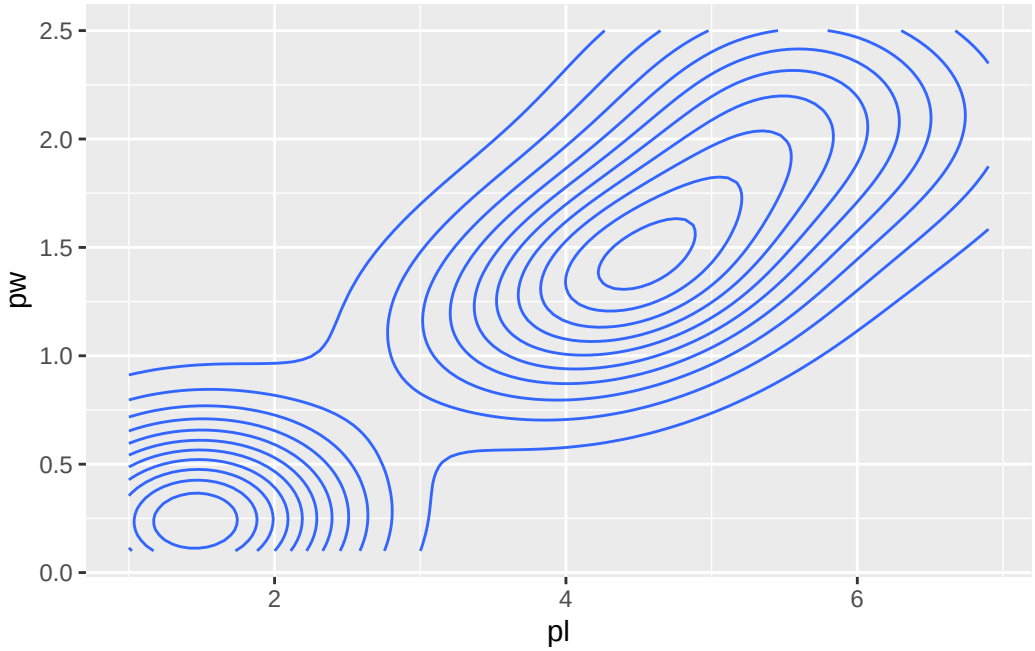
`coord_flip` vous sera aussi utile si vous désirez culbuter le graphe et passer les `x` en `y` et vice-versa.

10.7 `geom` (suite) : deux variables continues

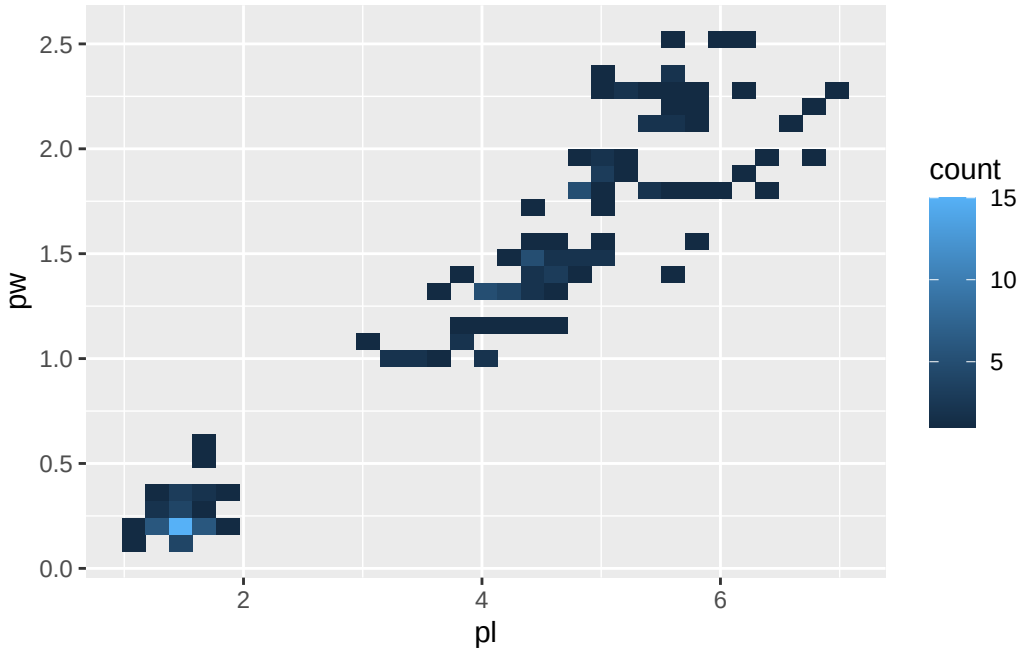
Moult autres `geom`s sont disponibles dans `ggplot2` et sont présentés graphiquement ici.

Pour continuer sur deux variables continues, on pourra rajouter facilement des courbes ou autres éléments de densité. Toutes les opérations kernel sont faites pour vous donc vous n'avez pas vraiment à vous en soucier. 150 points c'est un peu limite pour ces fonctions mais restons fidèles à `iris` :

```
gg <- iris2 %>% ggplot() + aes(pl, pw)
gg + geom_density2d()
```

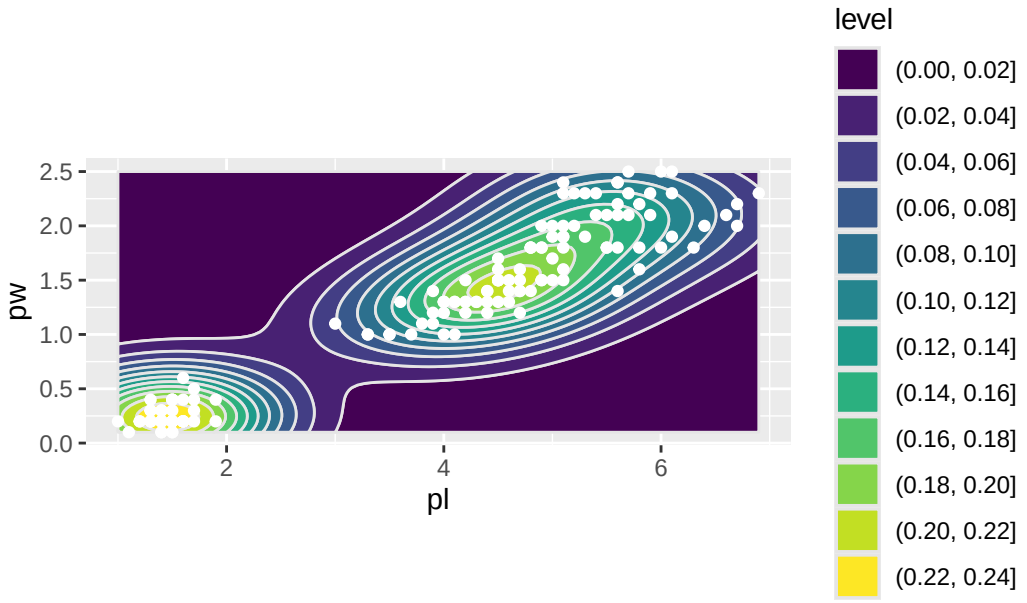



```
gg + geom_bin_2d()
```



Naturellement, vous pouvez rajouter les points. L'ordre de représentation est celui des déclarations. Si vous passez `geom_point` après `geom_density2d`, les points seront représentés au-dessus des courbes de densité :

```
gg + geom_density2d_filled(col="grey90") + geom_point(col="white") + coord_equal()
```



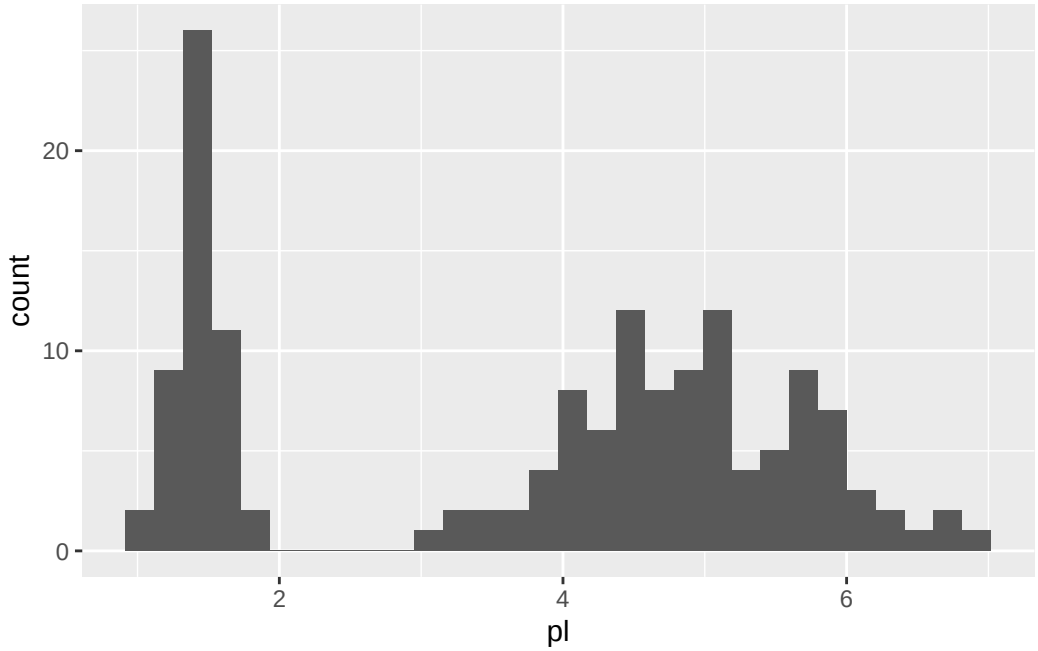
Classieux n'est-ce pas ?

10.8 `geom` (suite) : une seule variable continue

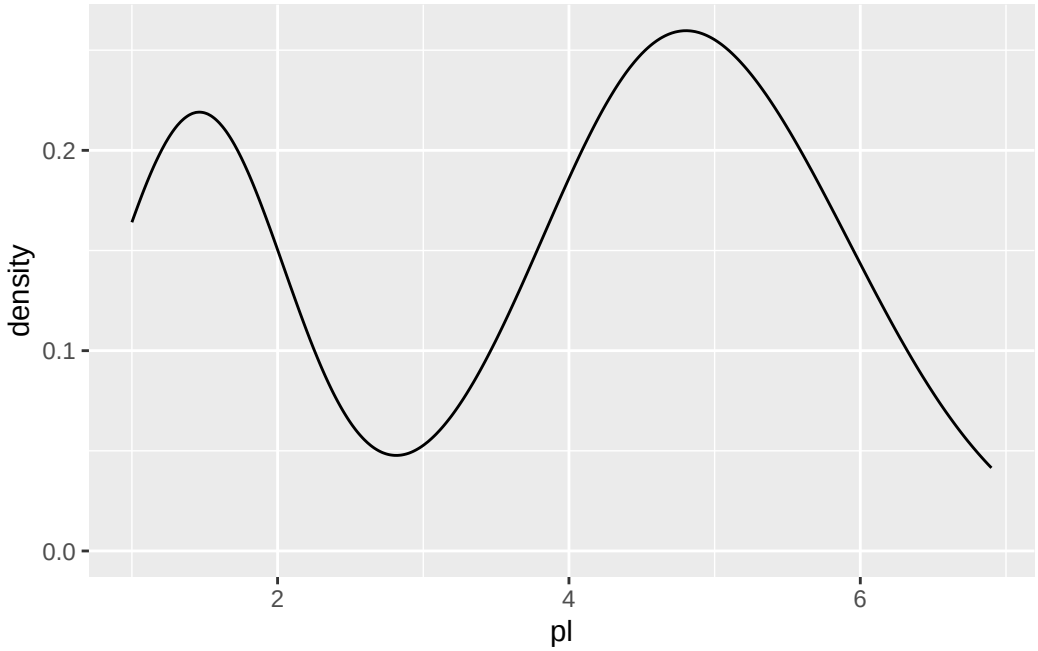
Quand vous n'avez qu'une seule variable continue c'est vraisemblablement que vous vous intéressez à sa distribution, c'est à dire que vous voulez un histogramme ou sa densité, selon que vous vouliez binner vos données ou les garder continues. Quelques exemples ci-dessous et je vous laisse explorer en autonomie.

```
gg <- iris2 %>% ggplot() + aes(pl)
gg + geom_histogram()
```

```
`stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```

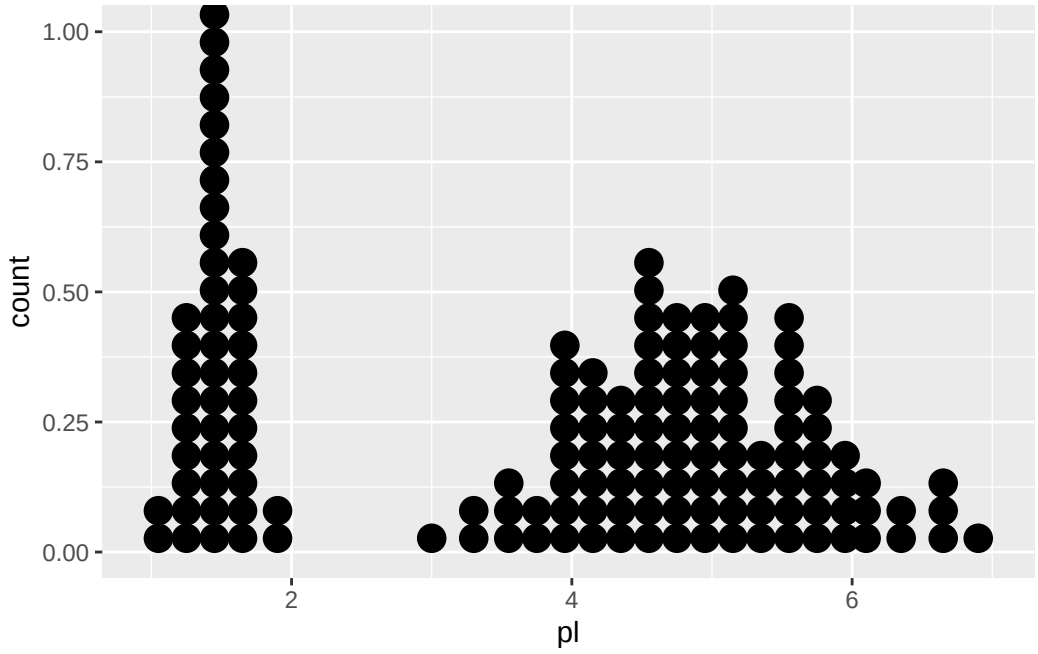



```
gg + geom_density()
```



```
gg + geom_dotplot()
```

```
Bin width defaults to 1/30 of the range of the data. Pick better value with
`binwidth`.
```

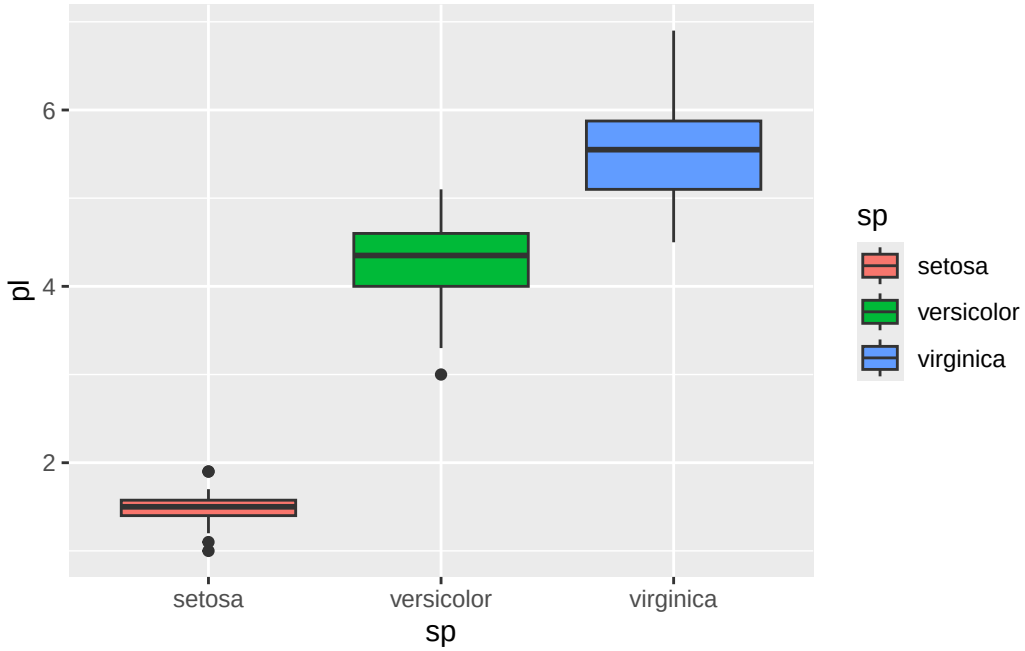


Quand `ggplot2` et plus généralement `R` vous enquiquine avec un message c'est souvent pour votre bien.

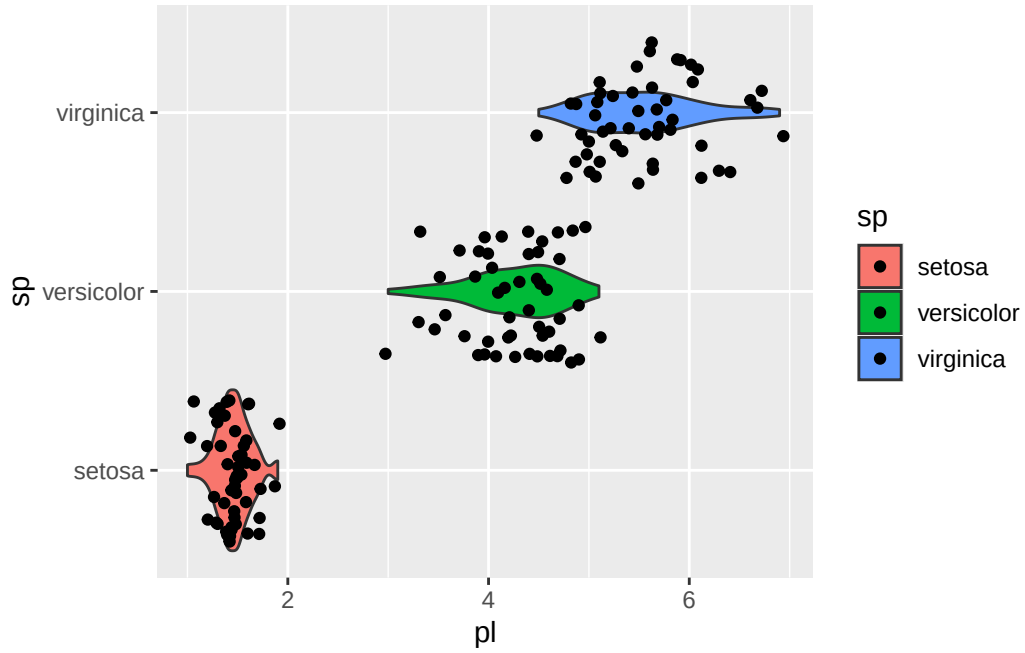
10.9 `geom` (suite) : une variable continue et un facteur

`ggplot2` a tout ce qu'il vous faut pour représenter des boxplots et variantes :

```
gg <- iris2 %>% ggplot() + aes(x=sp, y=pl, fill=sp)
gg + geom_boxplot()
```



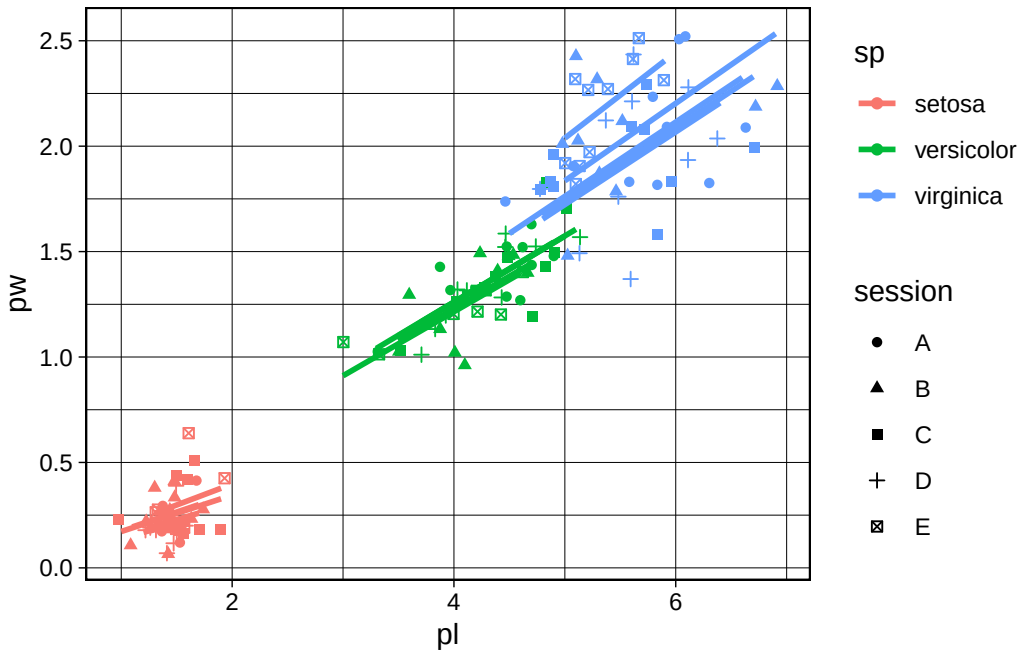

```
gg + geom_violin() + geom_jitter() + coord_flip()
```



10.10 Les sous-graphes avec `facet_`

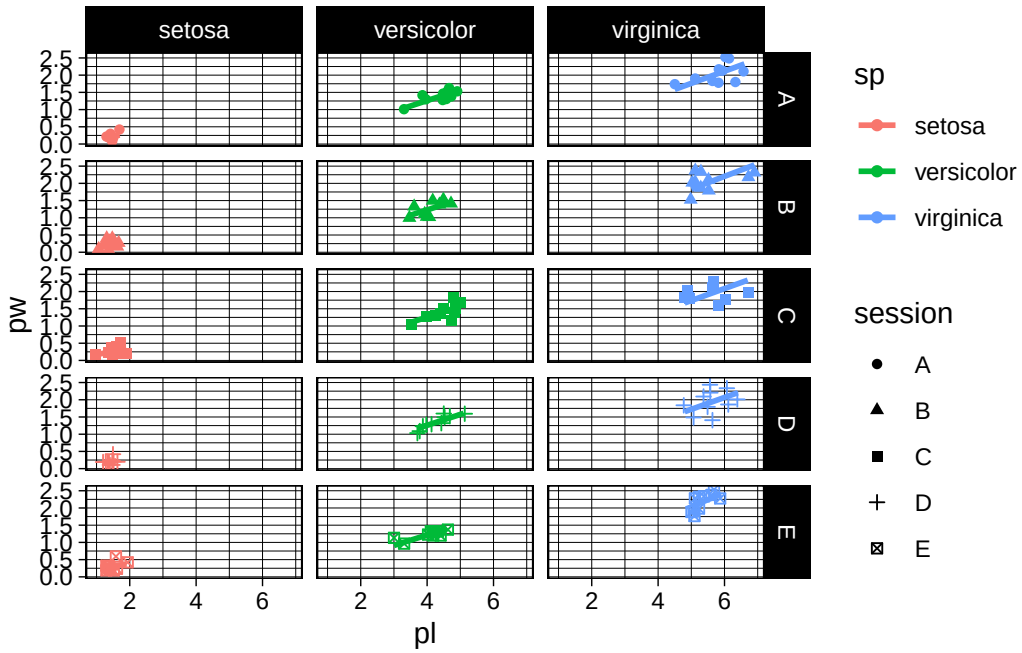
Une famille de fonction très puissante en `ggplot2` est celle des `facet_` qui vous permettent de faire des `sous-graphes` facilement. Pour que l'exemple soit encore plus aiguisé, nous allons créer une autre colonne facteur dans `iris2` simulant cinq sessions de terrain, de mesure de 10 iris par espèce par session. Puis nous allons créer un graphe de base, l'occasion de montrer l'utilisation de `shape`.

```
iris2 <- iris2 %>% mutate(session=factor(rep(rep(LETTERS[1:5], each=10), 3)))
gg <- iris2 %>% ggplot() +
  aes(pl, pw, col=sp, shape=session) +
  geom_jitter() + geom_smooth(method="lm", formula="y~x-1", se=FALSE) +
  theme_linedraw()
gg
```

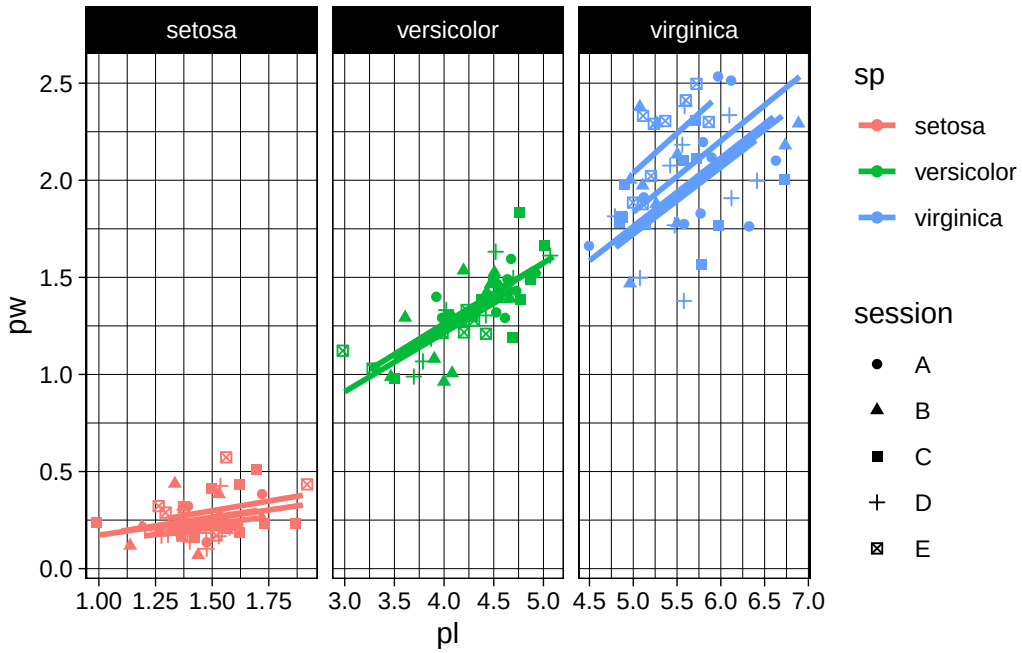


Nous conviendrons qu'on y voit goutte. Surtout entre sessions. `facet_grid` est votre ami. Vous pouvez spécifier qui va en ligne, qui va en colonne et si les échelles doivent être fixes ou peuvent être libres. Quelques exemples :

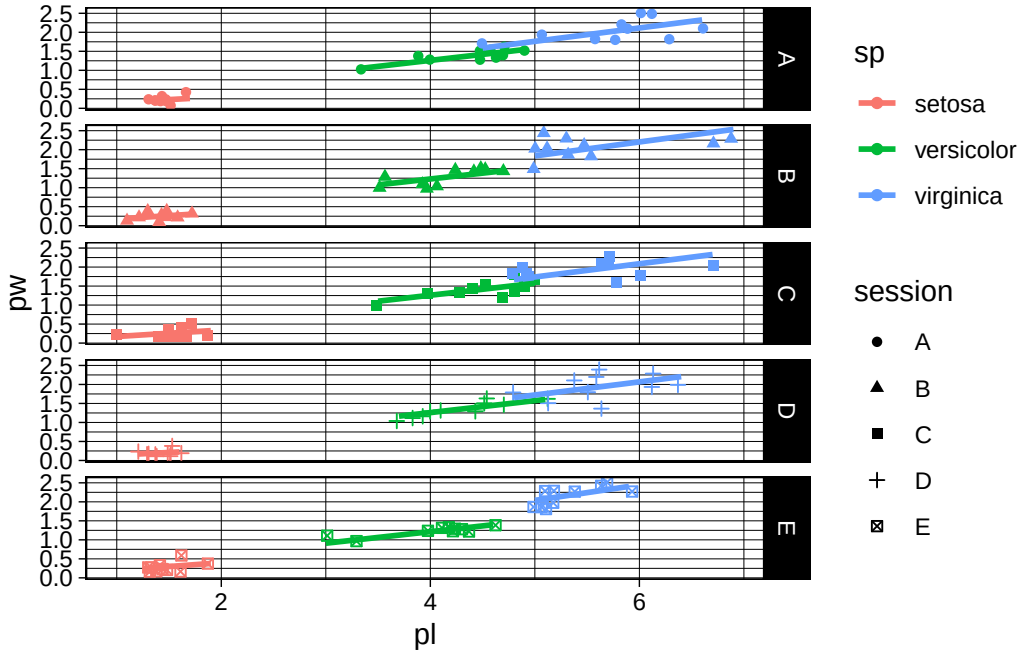
```
gg + facet_grid(session~sp)
```




```
gg + facet_grid(.~sp, scales="free")
```



```
gg + facet_grid(session~., scales="fixed")
```

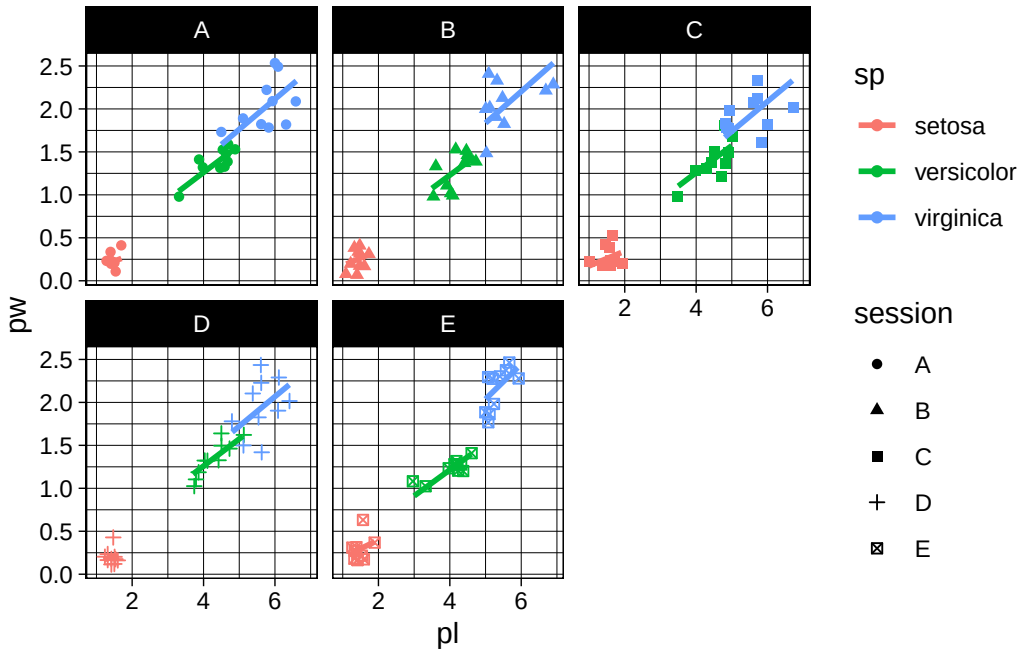


Pratique ! Notez que quand vous ne voulez qu'une dimension, vou spouvez omettre l'un des membres à gauche ou à droite et le remplacer par un `.` [3]

`facet_wrap` est plus lâche dans sa définition et ne veut qu'une seule colonne, dont les sous-graphes correspondant seront simplement enroulés selon des dimensions plaisantes, que vous pouvez spécifier :

```
gg + facet_wrap(~session, nrow=2)
```

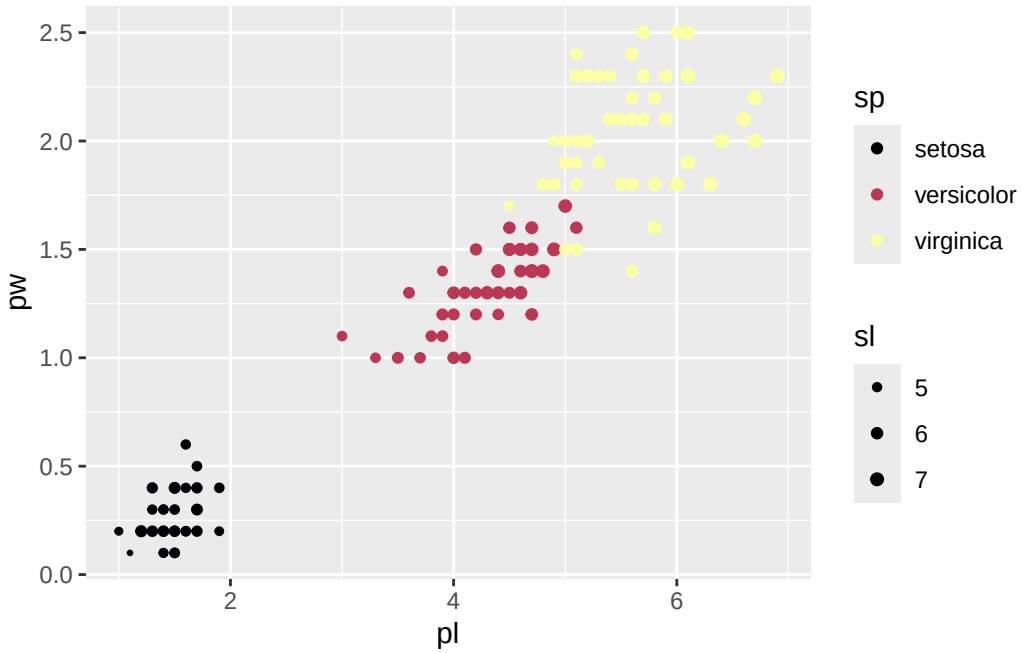
[3] et `~sp` sans point à gauche fonctionnerait aussi.



10.11 Interlude cosmétique : `scale_` (suite) et `guides`

Si vous ne vous pâmez pas devant les couleurs par défaut, il est naturellement possible de les changer avec une des fonctions `scale_color_*`. Il en va de même pour les autres modes de représentation définis dans `aes`.

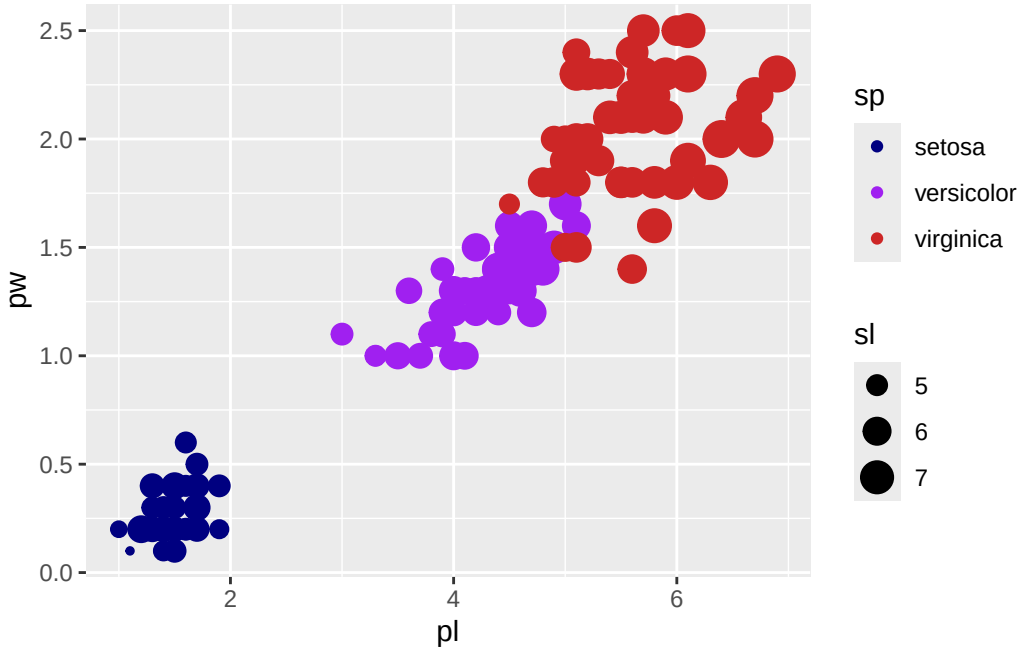
```
gg <- iris2 %>% ggplot() + aes(pl, pw, col=sp, size=sl) + geom_point()
gg + scale_color_viridis_d(option = "B") + scale_size(range=c(0.5, 2))
```



Je vous laisse explorer les `scale_` en autonomie. Oui, il y a moyen d'y passer des journées. Le choix des couleurs est un sujet central car il faut penser à tout le monde : les daltoniens, les imprimantes noir et blanc. Et encore plus largement la perception des couleurs par l'œil et le cerveau humains.

Vous pouvez également ne pas vous en référer à des palettes pré-construites et fixer vos propres couleurs avec un vecteur nommé et l'une des fonctions `scale_*_manual` :

```
sp_cols <- c("setosa"="navyblue", "versicolor"="purple", "virginica"="firebrick3")
gg + scale_color_manual(values=sp_cols)
```

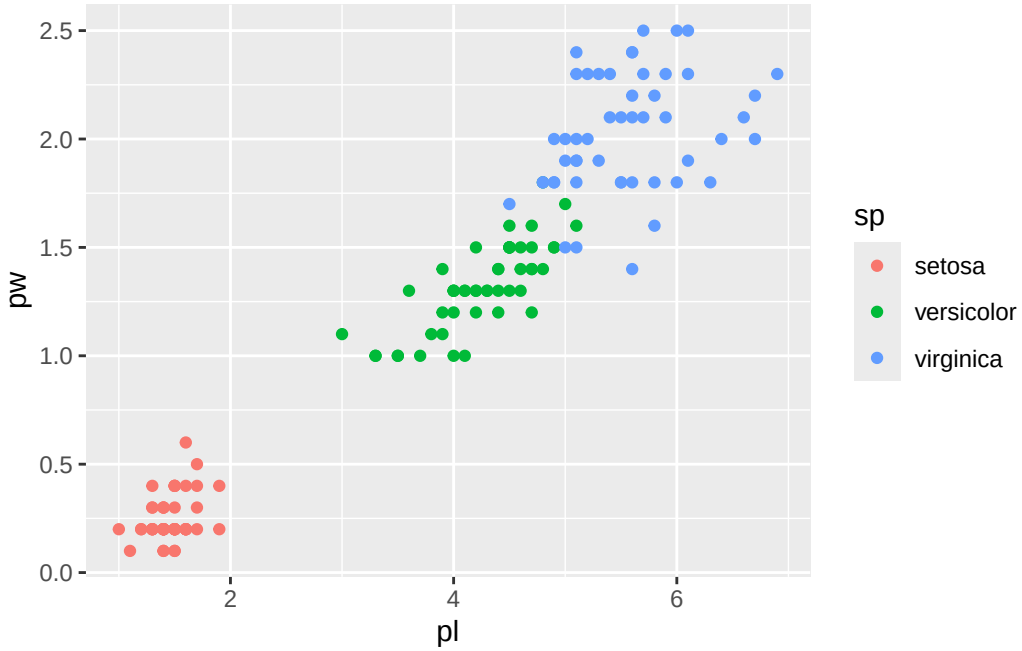


10.12 %+% : une fabrique à graphes

Plus haut, je vous ai promis que les graphes `ggplot2` étaient en soit des machines à faire des graphes[4]. Imaginons que nous disposions d'un autre jeu de données se présentant de la même façon, typiquement avec les mêmes colonnes utilisées par un `ggplot` que vous auriez construit avec amour et simplicité :

```
gg <- ggplot(iris2) + aes(pl, pw, col=sp) + geom_point()
gg
```

[4] on pourrait parler de fonctions, voire de fonction factories



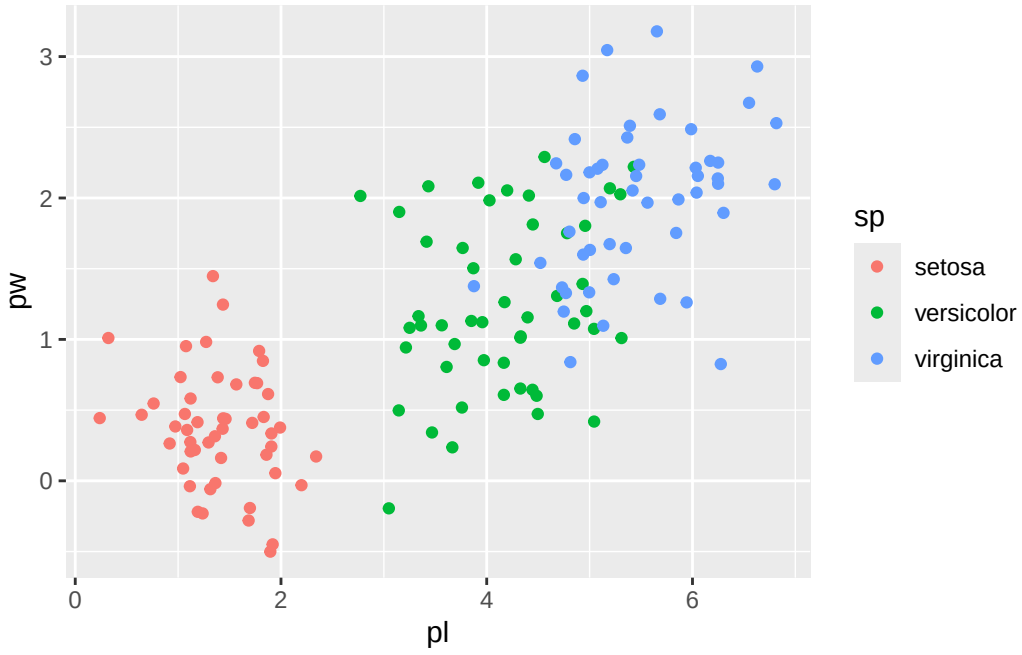
Et voilà un autre jeu de données, avec un peu de bruit gaussien pour tout le monde :

```
iris_bis <- iris2 %>% mutate(across(c(pl, pw), ~.x+rnorm(.x, sd=0.5)))
iris_bis
```

```
# A tibble: 150 x 6
      sl    sw    pl      pw sp     session
   <dbl> <dbl> <dbl>   <dbl> <fct>  <fct>
 1   5.1   3.5 1.91   0.335  setosa A
 2   4.9   3   1.31  -0.0591 setosa A
 3   4.7   3.2 1.30   0.271  setosa A
 4   4.6   3.1 0.762  0.547  setosa A
 5   5     3.6 0.973  0.385  setosa A
 6   5.4   3.9 1.02   0.734  setosa A
 7   4.6   3.4 1.19   0.415  setosa A
 8   5     3.4 1.12   0.208  setosa A
 9   4.4   2.9 1.12   0.582  setosa A
10   4.9   3.1 1.57   0.682  setosa A
# i 140 more rows
```

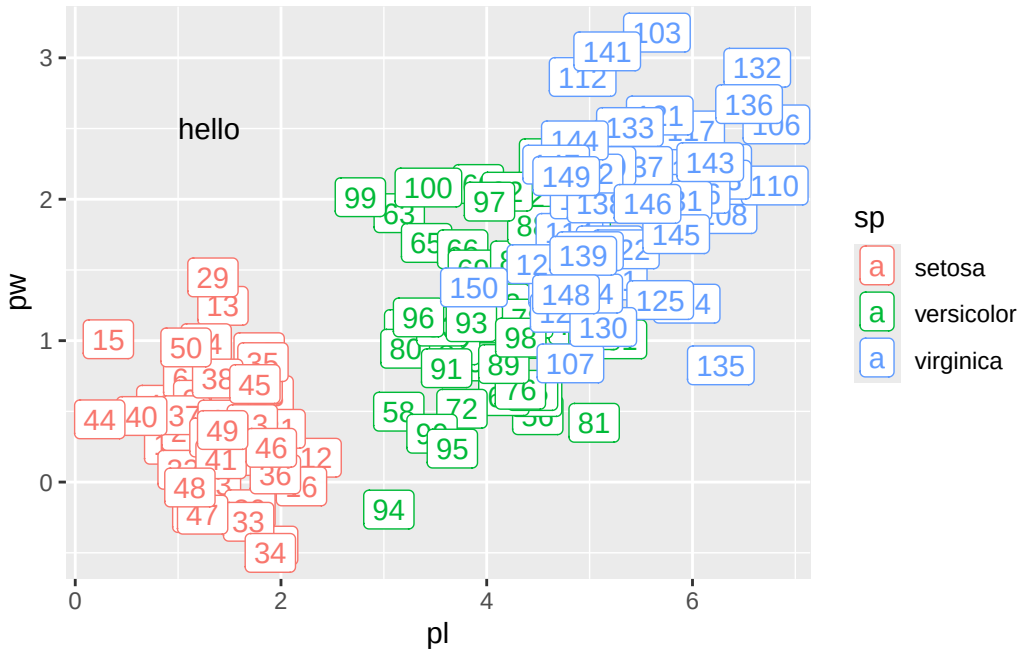
Nous pouvons remplacer le jeu de données utilisé initialement par `gg` (c'est à dire `iris2`) par votre nouveau jeu de données (c'est à dire `iris_bis`), en utilisant l'opérateur `%+%` :

```
gg %+% iris_bis
```



Vous pouvez même modifier le ggplot ainsi obtenu. Ci-dessous, je pousse le bouchon jusqu'à manipuler au tout dernier moment `iris_bis` pour ajouter un numéro d'individu séquentiel et ajouter une étiquette avec `geom_label`, une variante de `geom_text`, tous deux bien utiles. À y être, je rajouter aussi une petite annotation, alignée à gauche sur le point `(1; 2.5)`.

```
gg2 <- gg %+% mutate(iris_bis, i=1:n()) +
  geom_label(aes(label=i)) +
  annotate("text", x=1, y=2.5, label="hello", hjust=0)
gg2
```



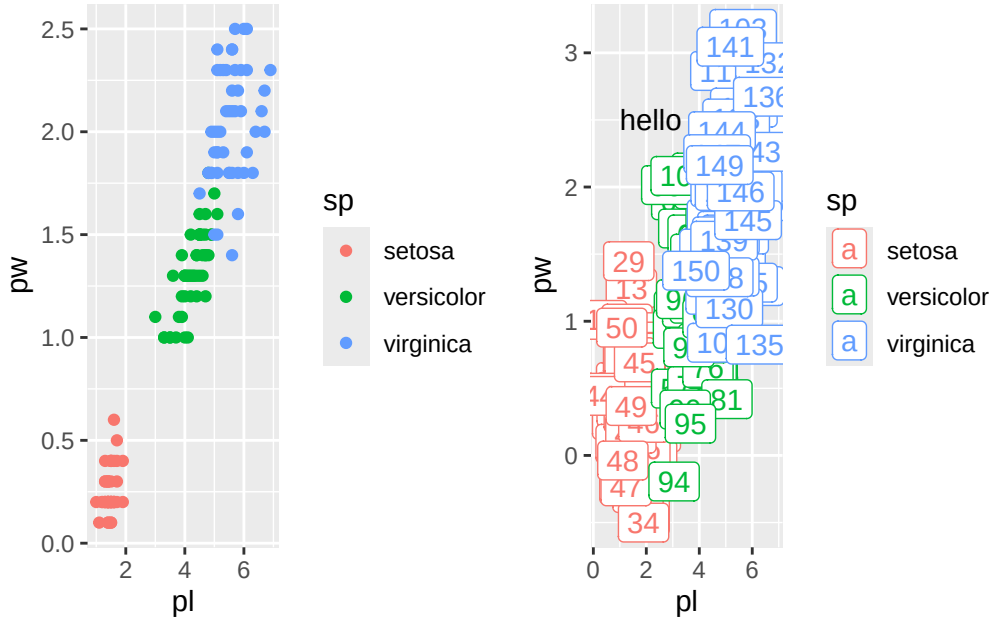
10.13 Un package bien utile : `patchwork`

Quantité de packages existent pour étendre encore les fonctionnalités de `ggplot2`. Nous pouvons citer :

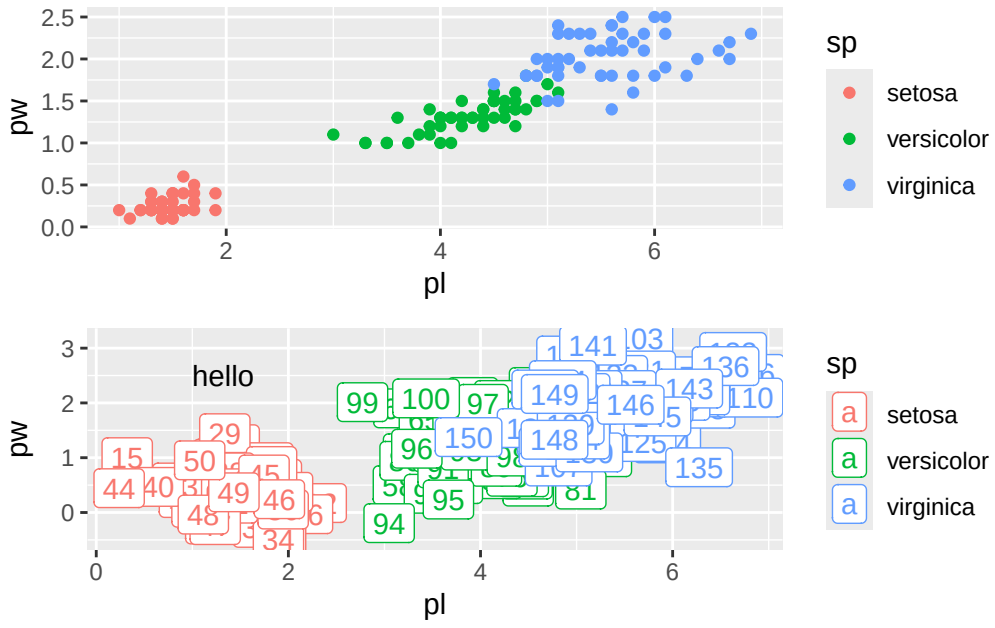
- `plotly` : pour des graphes interactifs
- `ggmap` : gestion des données spatiales et des projections
- `ggrepel` : pour avoir des étiquettes de données non juxtaposées
- `ggdendro`: representer des dendrogrammes et autres arbres
- etc.

Nous ne présenterons que `patchwork`: qui permet l'assemblage intuitif de graphes. Son fonctionnement est simplissime. Si vous avez plusieurs `ggplot` vous pouvez les assembler, comme des "méta-facet" en quelque sorte. Les opérateurs `+` et `/` construisent un seul `ggplot` juxta- ou superposés :

```
library(patchwork)
gg + gg2
```


pw
pl
sp
setosa
versicolor
virginica
hello
a
a
a

```
gg / gg2
```


pw
pl
sp
setosa
versicolor
virginica
hello
a
a
a

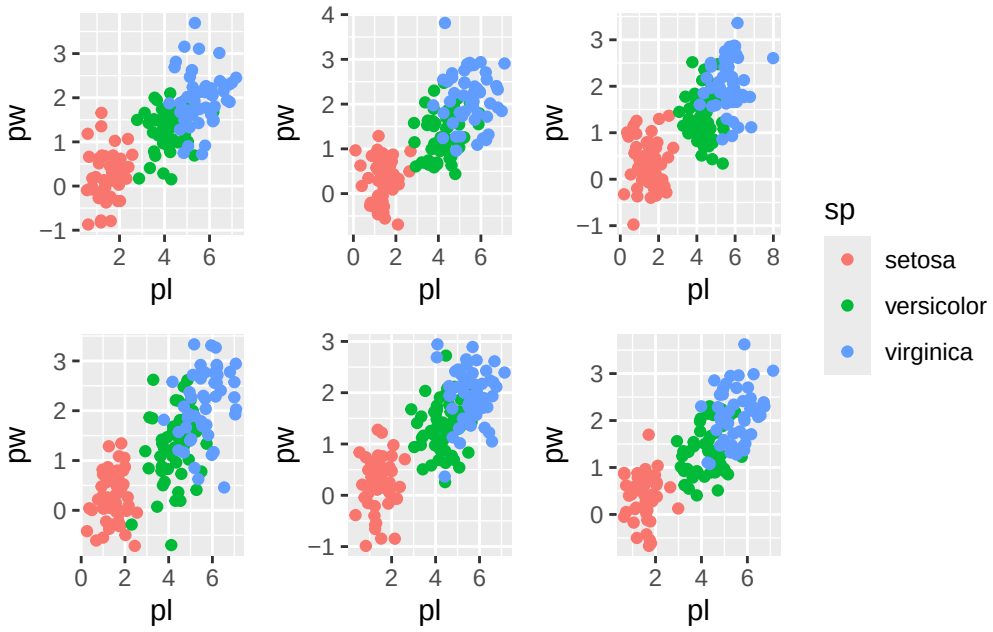
`plot_layout` et `plot_annotation` sont très utiles pour des compositions plus sophistiquées.

Enfin, si vous avez non pas deux mais une quantité de graphes, par exemple dans une liste, `wrap_plots` fait le job.

Imaginons que je vous ai convaincu · e de ne désormais plus faire de terrain et de simuler vos données. Imaginons que vous simuliez `K`, disons 6, sessions de mesures.

Le `map` ci-dessous fera l'objet de tout le chapitre suivant. Ici, un simple aperçu de sa puissance.

```
K=6
# simulate data
iris6 <- map(seq_len(6),
             ~iris2 %>% mutate(across(c(pl, pw), ~.x+rnorm(.x, sd=0.5))))
# build the ggs
gg6 <- map(iris6, ~ gg %+% .x)
# patchwork them
wrap_plots(gg6, ncol=3) + plot_layout(guides="collect")
```



10.14 Sauvez vos créations avec ggsave

Une fois que vous êtes satisfait · e de votre `ggplot`, vous pouvez le sauver vers un `pdf`, `eps`, `jpg`, `png`, etc. facilement avec la fonction `ggsave`.

Il vous suffit d'appeller `ggsave` avec le nom de fichier et son extension correspondant au format que vous voulez sauver. Vous pouvez aussi spécifier la largeur et la hauteur de sortie. Par défaut `ggsave` sauve le dernier graphe produit mais vous pouvez spécifier l'argument `plot` de `ggsave`.

```
ggsave("my_plot_1.pdf", gg, width=12, height=8)
```

10.15 Considérations post-liminaires

`ggplot2` peut-être frustrant dans les premiers temps. Comme vous le savez la frustration est typique mais temporaire. `ggplot2` mérite définitivement de s'y accrocher et les larmes de hargne seront bientôt remplacées par des cris de joie et des apéros en terrasse grâce au temps gagné.

11 Manipulation de listes avec purrr

Si vous n'avez besoin que de `purrr`, `library(purrr)` charge le package. Si vous utilisez tout le tidyverse, `purrr` sera chargée après :

```
library(tidyverse)
```

11.1 Les listes c'est la vie

```
z <- list(wagon1="pomme", wagon2=1:3, wagon3=c(TRUE, FALSE))
length(z)
```

```
[1] 3
```

```
names(z)
```

```
[1] "wagon1" "wagon2" "wagon3"
```

```
str(z)    # structure of z
```

```
List of 3
 $ wagon1: chr "pomme"
 $ wagon2: int [1:3] 1 2 3
 $ wagon3: logi [1:2] TRUE FALSE
```

```
# View(z) # RStudio Viewer
z         # simple print method
```

```
$wagon1
[1] "pomme"

$wagon2
[1] 1 2 3

$wagon3
[1]  TRUE FALSE
```

Ne soyez pas effrayé · e par les listes : elles sont simplement des conteneurs pouvant accueillir des objets de classes différentes (ou non) et de longueur différentes (ou non).

On peut se figurer une liste comme un train dont les éléments seraient des wagons dont chacun contiendrait des conteneurs plus petits qui eux contiennent des éléments :

```
  --  wagon 1  --        --  wagon 2  --       --  wagon 3  --
[    ["pomme"]    ] -- [    [ 1:3 ]     ] - [    [ c(TRUE, FALSE) ]    ]
```

Une liste s'indexe de plusieurs façons, positionnellement ou avec le nom de se(s) élément(s)-wagons s'ils sont nommés :

```
z[2]        # using positionnal index
```

```
$wagon2
[1] 1 2 3
```

```
z["wagon2"] # using ["name"]
```

```
$wagon2
[1] 1 2 3
```

```
class(z[2])
```

```
[1] "list"
```

Notons que la syntaxe `z$wagon2` fonctionne aussi.

Si l'on utilise les doubles crochets, on accède directement au contenu des caisses dans les wagons. Ces derniers ont perdu leurs natures de "wagon", ne sont plus des listes :

```
  z[[2]]        # using positionnal index
```

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

```
  z[["wagon2"]] # using ["name"]
```

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

```
  class(z[[2]])
```

```
[1] "integer"
```

Les assignations `list[index] <-` et `list[[index]] <-` fonctionnent comme à l'acoutumée.

11.2 `map` à la vanille

Le package `purrr` s'articule autour de la famille `map` dont il existe plusieurs variantes dont la première à voir est `map` tout court.

L'idée est simple : appliquer une fonction à tous les éléments d'une liste et retourner une liste. La liste sera le premier argument de `map`, la fonction le second. Le nom de fonction est passé sans parenthèses :

```
  map(z, class)
```

```
$wagon1
[1] "character"

$wagon2
[1] "integer"

$wagon3
[1] "logical"
```

11.3 `map_*` et ses autres parfums

Dans l'exemple ci-dessus, si l'on ne veut pas de liste mais un vecteur (quand cela est possible), on peut utiliser `unlist` :

```
z %>% map(class) %>% unlist()
```

```
     wagon1      wagon2      wagon3
"character"   "integer"   "logical"
```

Mais quand la classe de sortie d'une fonction map est homogène et connue, on utilisera plutôt les variantes de map de la forme `map_*`. `*` pouvant prendre les valeurs `int` et `dbl` pour retourner des `numeric` sous forme d'entiers ou de doubles, `lgl` pour les vectuers logiques, `df`, `dfr` et `dfc` pour retourner des `data.frame`, éventuellement combinés par `r`ow ou `c`olonnes.

```
z %>% map_chr(class)
```

```
     wagon1      wagon2      wagon3
"character"   "integer"   "logical"
```

```
list(1:2, 4:9) %>% map_dbl(length)
```

```
[1] 2 6
```

11.4 `~` et `\(x)` : les fonctions anonymes sont vos amies

Dans `purrr` et plus largement dans le tidyverse, on peut déclarer des fonctions anonymes, "à la volée" c'est à dire qu moment où l'on en a besoin. `purrr` mobilise l'opérateur `~` et la variable `.x`.

R "de base" a repris l'idée dans une saveur un peu différente avec `\(x)`. Ces trois approches sont équivalentes :

```
x <- 1:5 # create a vector
# option 1 : using a named function
square <- function(x) x^2
map_dbl(x, square)
```

```
[1]  1  4  9 16 25
```

```
# option 2 : purrr style anonymous function
map(x, ~.x^2)
```

```
[[1]]
[1] 1

[[2]]
[1] 4

[[3]]
[1] 9

[[4]]
[1] 16

[[5]]
[1] 25
```

```
# option 3 : R base anonymous function
map(x, \(x) x^2)
```

```
[[1]]
[1] 1

[[2]]
[1] 4

[[3]]
[1] 9

[[4]]
[1] 16

[[5]]
[1] 25
```

11.5 `map2` et généralisation `pmap`

Imaginons que vous ayez deux listes `z1` et `z2` et que chaque élément de `z1` doivent être élevé à la puissance de chaque élément de `z2`. `purrr` généralise `map` avec deux listes avec les fonctions `map2_*` :

```
z1 <- list(4:5, c(3, 2, 4.25), 1:3)
z2 <- list(2, 1, 3)
map2(z1, z2, \(x, y) x^y)
```

```
[[1]]
[1] 16 25

[[2]]
[1] 3.00 2.00 4.25

[[3]]
[1]  1  8 27
```

Si vous avez plus de trois listes, vous pouvez utiliser `pmap_`, comme suit. Le premier argument sera une liste de toutes vos listes. Imaginons qu'après avoir élevé à la puissance de `z2` nous voulions ajouter une valeur dans `z3` :

```
z3 <- list(10, -5, 0)
pmap(list(z1, z2, z3), \(x, y, z) x^y + z)
```

```
[[1]]
[1] 26 35

[[2]]
[1] -2.00 -3.00 -0.75

[[3]]
[1]  1  8 27
```

11.6 Opérations sur listes

11.7 cheat sheet

12 Le reste du tidyverse au pas de course : forcats, stringr, lubridate et readr

12.1 forcats

12.2 stringr

12.3 lubridate

12.4 readr

13 Dictionnaire

Ci-dessous, le vocabulaire de base pour bien démarrer en R. Nous n'avons pas tout vu mais vous savez comment trouver de l'aide pas vrai ?

13.1 Environnement

- `<-`, `assign` : assigne une valeur
- `%>%`/`|>` : *forward pipe* `g(f(x))` devient `x %>% f %>% g`
- `get/setwd` : accède et changer l'emplacement de travail
- `here::here` est une alternative plus générique
- `ls` : liste les objets dans l'environnement
- `rm(list=ls())` : efface tous les objets mais votre ordinateur peut prendre feu[1]
- `object.size` : taille d'un objet
- `install.packages`/`library` : installe et charge des packages
- `citation` : citer les gentils gens
- `sessionInfo` : informations de session, notamment le versionnage

13.2 Arithmétique

opérateur	signification
`+`	addition
`-`	soustraction
`/`	division
`*`	multiplication
`%%`	modulo
`%/%`	division euclidienne
`^`	puissance

[1] https://twitter.com/hadleywickham/status/940021008764846080

13.3 Mathématiques

fonction	signification
`log`	logarithme (népérien)
`log10`	logarithme (décimal)
`exp`	exponentielle
`sum`	somme
`min`	minimum
`max`	maximum
`range`	étendue, raccourci pour `c(min(x), max(x))`
`median`	médiane
`quantile`	quantile (voir l'argument `probs`)
`summary`	un peu de tout ce qui précède
`round`	arrondi classique
`signif`	garde `n` chiffres
`floor`	arrondi inférieur
`ceiling`	arrondi supérieur
`var`	variance
`sd`	écart-type
`cor(x, y)`	corrélation
`cov(x, y)`	covariance
`cos` et al.	trigonométrie
`?Complex`	pour la gestion des complexes

13.4 Valeurs spéciales

- `NA` pour les valeurs manquantes. `na.omit` et les arguments `na.rm` aident à filtrer ces valeurs.
- `NULL` : ensemble vide
- `-Inf/Inf` : $\pm\infty$
- `pi` et autres constantes (`?Constants`), `letters`
- Beaucoup de jeux de données disponibles nativement, voir `?datasets` ## Arithmétique

13.5 Comparaison

opérateur	signification
`<`	strictement inférieur

opérateur	signification
`<=`	inférieur ou égal
`>`	strictement supérieur
`>=`	supérieur ou égal
`==`	égal
`!=`	différent
`%in%`	dans l'ensemble

13.6 Tests logiques

opérateur	signification
`!`	NOT
`&`	AND (élément par élément)
`&&`	AND
`|`	OR (élément par élément)
`||`	OR
`xor(x, y)`	OR (exclusif)
`all`	teste si uniquement des TRUE
`any`	teste si l'une des conditions au moins est TRUE

13.7 Control flow

- Toute la clique habituelle (`?Control`) : `if`, `else`, `ifelse`, `for`, `while`, `next`, `break`, etc.

13.8 Fonctions

```
nom_fonction <- function(arg1=default1, arg2=default2, ...) {
  # instructions
  return()
}
```

- Les arguments et leurs valeurs par défaut sont optionnels.
- `return` n'est pas obligatoire, la dernière ligne est renvoyée. Ne rien retourner est possible
- `list()` pour des retours de plus d'une valeur
- fonctions anonymes, souvent créées à la volée: `function(.x) .x*2` ou `\.x .x+2`. Souvent passé en `formula` à `map` (argument `.f`), par exemple : `~x+3`

13.9 Vecteurs

- `[i]/[i]<-` : accède et modifie les i-ème(s) position d'un vecteur
- `c` : concatène des valeurs et crée un vecteur
- `names/names<-` : accède et change les noms d'un vecteur
- `sort` : ordonne un vecteur
- `order` : ordonne un vecteur mais retourne les indices
- `rev` : retourne un vecteur du dernier au premier élément
- `unique` : retourne les valeurs distinctes
- `table` : compte les valeurs d'un ou de plusieurs vecteurs

13.10 Séquences régulières et aléatoires

- `seq` : séquences régulières
- `a:b` : raccourci pour `seq(a, b)`
- `rep` : répète un vecteur ou ses éléments
- `runif/rnorm/rbinom` : nombres aléatoires issus d'une distribution uniforme, normale, binomiale. Voir `?Distributions`

13.11 Matrices

- `[i, j]` + `[i, j]<-` : accède et modifie les valeurs aux `i`-ième(s) et `j`-ième(s) indices (lignes puis colonnes)
- `matrix/data.frame` : création
- `is/as.matrix` et `is/as.data.frame` : test et conversion de type
- `col/rownames` et `col/rownames<-` : accéder et définir row/colnames
- `t` : transposition
- `%*%` : multiplication matricielle (`matrix` only)
- `apply` : opération marginale
- `row/colSums` : raccourci pour `apply(m, 1/2, sum)`

13.12 Listes

13.13 `dplyr` : Manipulation de data.frame

- RStudio : `Help > Cheatsheets > dplyr` : très bonne adresse
- `vignette(package="dplyr")`, par ex. `vignette("dplyr")`

- on préférera les tibbles qui apportent des améliorations cosmétiques aux data.frame (`as_tibble`)
- `tibble` et `tribble` pour le constructeur
- `slice` : filtre sur les lignes avec des indices
- `filter` : filtre sur les lignes avec des tests logiques
- `select` : sélectionner et réordonne les colonnes
- `starts_with` et autres helpers de `tidyselect`
- `rename` : renomme les colonnes
- `mutate` : crée de nouvelles colonnes
- `group_by`/`ungroup` : groupe, déproupe par colonnes
- `summarise` : résume en 1 ligne ou 1 ligne par groupe, des colonnes
- `group_split` : tranche une tibble par groupe
- `bind_rows/cols` : colle par lignes ou colonnes
- `*_join` : opérations de join sur deux tables partageant un index

13.14 `stringr` : manipulation de chaines de caractères

- `vignette(package="stringr")`, par ex. `vignette("stringr")`

13.15 `ggplot2` : un grammaire pour les graphes

- `vignette(package="ggplot2")`, par ex. `vignette("ggplot2")`
- RStudio : `Help > Cheatsheets > ggplot2` : très bonne adresse

13.16 `forcats` : manipulation de facteurs

- `vignette(package="forcats")`, par ex. `vignette("forcats")`

13.17 `purrr` : travailler avec des listes

- `vignette(package="purrr")`, par ex. `vignette("purrr")`
- RStudio : `Help > Cheatsheets > purrr` : très bonne adresse
- `map(liste, function)` : travaille sur une liste, retourne une liste
- `map(liste, ~.x %>% ...)` : accepte les fonctions anonymes
- `map_{dbl, chr, lgl, df, dfr, dfc}` : idem mais retourne des numeric, character, logical, data.frame, data.frame collés par lignes/colonnes
- `map2_*(liste1, liste2, ...)` : idem pour deux listes

- `pmap_*` : généralisation à N listes

13.18 Import/Export

- texte brut : `readLines`
- lire des tables : `read.table`, `readr::read_csv`, `xlsx::read.xlsx`.
- `Import dataset` dans RStudio est moins pénible de prime abord
- écrire des tables : `write.table`, `readr::write_csv`, `xlsx::write.xlsx`
- `save/load` : sauve et charge un objet R/`.rda`
- pour les plots de `graphics` : `pdf(...) {...} dev.off()`
- pour les ggplot de `ggplot2` : `ggsave`

13.19 Interactions audio-visuelles

- bips : package `beepr`
- barres de progression : package `progress`
- invite de commande : `readline`
- logs : `sink`
- `Sys.sleep`, `Sys.time`, `Sys.*` : pause, horodatage, autres interactions

14 Ressources

14.1 Must see

- [R] chez Stack Overflow

14.2 Moteur de recherche

- Rseek dédié à R

14.3 Journaux

- *R-Journal* : la gazette de R avec les nouveaux packages, des astuces, des articles techniques. *Open access*.
- *Journal of Statistical Software* : officiellement multilangages, en pratique, beaucoup de nouveaux packages R. *Open access*.

14.4 Manuels

- D'excellents tutoriels sont regroupés sur le site du CRAN, en anglais et d'autres langues.
- *Documentation officielle*, éditée par la *R Development Core Team* : introduction à R, définition du langage *per se*, package authoring, etc. : assez aride et souvent très technique (et pas uniquement pour les novices) mais reste la référence absolue.
- *R pour les débutants* [fr], [en] : la mère de tous les tutoriels courts sur R, très complete introduction à R, une excellent adresse par Emmanuel Paradis.
- *Une introduction à R* : une autre excellente ressource, par Julien Barnier.
- *Aide mémoire de statistique appliquée à la biologie* : un document très concis et très utile par Maxime Hervé. Malgré non nom, sa portée est au delà de la biologie et globalement une bon résumé pour l'analyse statistique depuis la préparation jusqu'à l'analyse.

14.5 Ouvrages

- *Discover statistics using R* (2012) par Andy Field, Jeremy Miles et Zoë Field (992pp.) chez *Sage*. Une excellente ressource très complète et vivante.
- *The R Book* (2012) Seconde édition par Michael J. Crawley. Une autre bonne ressource, complète, sur R.
- * R for Data Science* LA référence
- *Use R!* Série (50+ ouvrages) publiée chez *Springer*. Généralement excellente et focalisée sur une problématique
- *Advanced R* par Hadley Wickham. En ligne et en version imprimée. Plongée dans la tambouille interne de R.
- *R packages* par Hadley Wickham. En ligne et en version imprimée. Une excellente et concise introduction à l'écriture de packages.
- *Efficient R programming* par Colin Gillespie et Robin Lovelace. Un cran plus loin, et meilleur, pour les programmeurs avertis.
- *S poetry*: un livre sur les origines de R, sur l'élégance de S.
- *The R Inferno*: le contre-point du précédent, par le même auteur, sur les singularités de R.
- *R packages* par Hadley Wickham. En ligne et en version imprimée. Une excellente et concise introduction à l'écriture de packages.

14.6 Sites

- CRAN Views : pour bien commencer avec les packages. Organisé par tâches (par exemple les Omics, l'inférence bayésienne, etc.) et "curationné" avec amour.
- *R-bloggers* : aggrégation de blogs qui traitent de R, quelques posts par jour, éclectique, une excellente ressource pour être à la page. tl;dr: un best-of ici
- *crantastic* : un site qui récupère toute l'activité sur les dépôts de packages. Permet de reviewer des packages, de chercher par auteur, etc.
- RPubs : sérendipité, me voilà !
- *R graph gallery* : collection de graphiques, qui ne fera pas oublier la gallerie "addicted to R" qui est en maintenance ou abandonnée depuis un bail
- *R-Studio's blog* : par l'équipe de R Studio, dernières actualités du développement de R Studio et des packages par son équipe
- Portail Wikipédia des statistiques [en] : en attendant le bus, le métro, le train, le dodo, une excellente et quasi infinie ressource.
- *The Data and Story Library* : des jeux de données célèbres, accompagnés de leurs histoires.
- Karl Broman : tous ses tutoriels sont délicieux et en particulier : préparation de données dans un tableur et rééducation R pour les gens qui l'ont appris avec qu'il ne deviennent cool

- PBIL : Statistiques pour la biologie (mais pas que, loin de là) par l'université de Lyon.
- WikiStats : ressources statistiques par l'INSA et le département de maths de l'université de Toulouse.

14.7 Cheatsheets

- RStudio's, dplyr, shiny, rmarkdown, etc. cheatsheets : génial
- R Vocabulary : back to basics
- Magott et al.'s refcard : ma préférée
- Jonathan Baron's refcard
- Tom Short's refcard
- Colors cheatsheet

14.8 Style guides

- Google's R style guide : longtemps la référence
- Hadley Wickham's style guide : plus courte, meilleure à l'usage et dans l'usage

14.9 Miscellanées

- Débug et canard en plastique
- Calculer son alcoolémie avec R
- La recette des falafels avec R
- Une autre liste de ressources

14.10 Quotes

- Tout ici est délicieux mais on peut rajouter :

  Frustration is typical and temporary - Hadley Wickham

  If the statistics are boring, then you've got the wrong numbers. - Edward Tufte

  Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live. - John Wood

  At their best, graphics are tools for reasonning about quantitative information. - Edward Tufte

There are three types of lies - lies, damn lies, and statistics. Benjamin Disraeli

Les statistiques sont comme la dentelle des petites culottes : elles révèlent le superflu mais cachent l'essentiel - ?

L'analyse statistique n'est vraiment utile qu'à des personnes qui n'en n'ont pas la maîtrise, et n'est maîtrisée que par des personnes qui n'en n'ont pas vraiment l'usage - Daniel Chessel