

Initiation au logiciel R

Pour l'analyse de données et les représentations graphiques

Benoît Simon-Bouhet

La Rochelle Université

3, 4 et 5 avril 2023

Les logiciels



Les logiciels



Téléchargement, installation



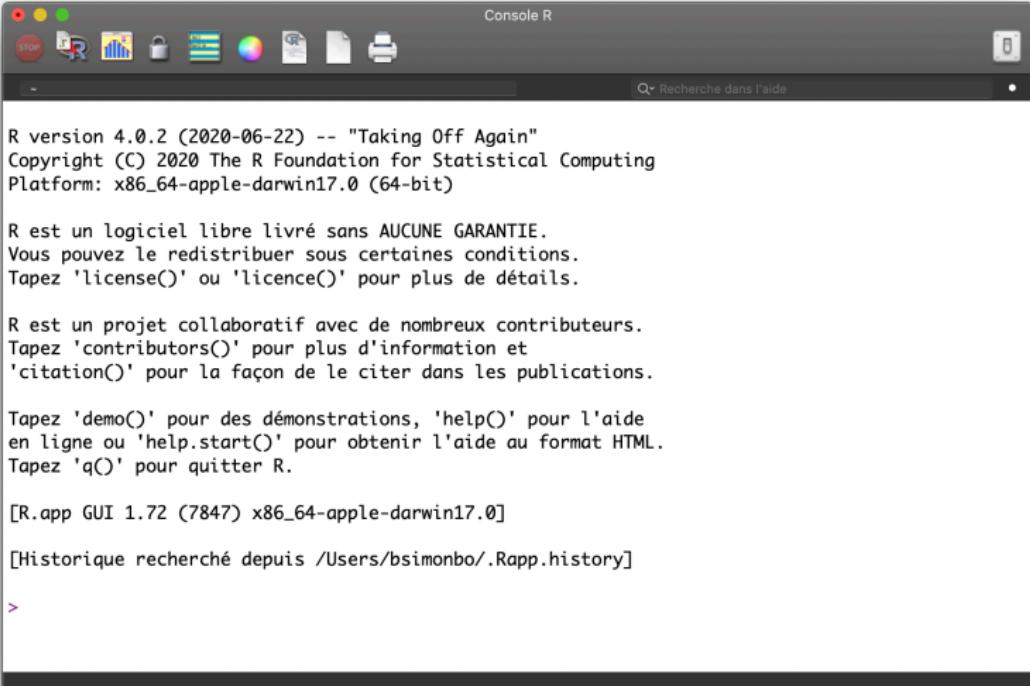
→ <https://cran.r-project.org>



→ <https://posit.co>

1^{er} contact

Interface : R



The screenshot shows the R console window running on a Mac OS X desktop. The window title is "Console R". The menu bar includes "File", "Edit", "View", "Help", and "Console". The toolbar contains icons for Stop, Run, Plot, Histogram, Lock, Bar Chart, Line Chart, Scatter Plot, Document, and Print. A search bar at the top right says "Recherche dans l'aide". The main text area displays the R startup message, license information, contributor details, help instructions, and the current session environment.

```
R version 4.0.2 (2020-06-22) -- "Taking Off Again"
Copyright (C) 2020 The R Foundation for Statistical Computing
Platform: x86_64-apple-darwin17.0 (64-bit)

R est un logiciel libre livré sans AUCUNE GARANTIE.
Vous pouvez le redistribuer sous certaines conditions.
Tapez 'license()' ou 'licence()' pour plus de détails.

R est un projet collaboratif avec de nombreux contributeurs.
Tapez 'contributors()' pour plus d'information et
'citation()' pour la façon de le citer dans les publications.

Tapez 'demo()' pour des démonstrations, 'help()' pour l'aide
en ligne ou 'help.start()' pour obtenir l'aide au format HTML.
Tapez 'q()' pour quitter R.

[R.app GUI 1.72 (7847) x86_64-apple-darwin17.0]

[Historique recherché depuis /Users/bsimonbo/.Rapp.history]

>
```

La console

On tape des commandes puis on presse Entrée :

```
3*4
```

```
[1] 12
```

```
4^2 # Puissance
```

```
[1] 16
```

```
sqrt(81) # Racine carrée
```

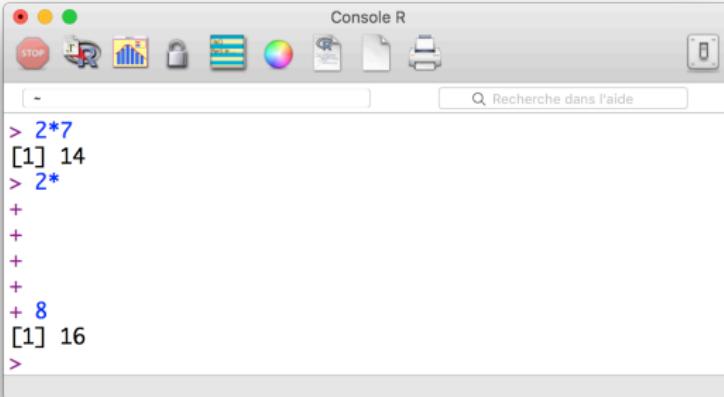
```
[1] 9
```

Le symbole “#” marque le début d'un commentaire.

La console

Par défaut, l'invite de commande est représentée par “>”

Si le symbole “+” apparaît à la place, c'est que la commande précédente était incomplète :



The screenshot shows the R Console window with the title "Console R". The window contains the following command history:

```
> 2*7
[1] 14
> 2*
+
+
+
+
+
+
+
[1] 16
>
```

La console

Si R comprend la commande, il affiche le résultat sur la ligne suivante.

Sinon, il dit haut et fort ce qui ne lui plaît pas¹ :

```
log("bonjour")
```

```
Error in log("bonjour"): argument non numérique pour une fonction mathématique
```

Si R n'affiche rien lorsque vous validez, c'est que la commande tapée a été comprise, exécutée, mais que l'affichage du résultat n'a pas été demandé :

```
a <- 10
```

1. Selon la version du logiciel et la langue du système, R s'exprime en Français ou en Anglais...

Trouver de l'aide : en ligne

1. Documents pdf : <https://www.r-project.org>

Documentation > Manuals > Contributed documentation

2. Forums, mailing lists, R-Help, Google : R CRAN + mots clés
3. Avant de poser des questions, fouillez² !

*“Getting **flamed** for asking dumb questions on a public mailing list is all part of growing up and being a man/woman.”*

— Michael Watson, 2004
(in a discussion on whether answers on R-help should be more polite)

2. RTFM...

Trouver de l'aide : dans R

1. Recherche par mot clé dans les fichiers d'aide installés

```
help.search("mot.clé")
```

```
help.search("wilcoxon")
```

2. Ouvrir un fichier d'aide : ?nom.de.la.fonction

```
?wilcox.test
```

Le répertoire de travail

Le répertoire de travail

Afficher le chemin du répertoire de travail (**get working directory**)

```
getwd()
```

```
[1] "/Users/bsimonbo/Formations/Initiation_R/Avril_2021"
```

Spécifier un nouveau répertoire de travail (**set working directory**)

```
setwd("~/Desktop/MonDossier")
```

ou *via* le menu Fichier > Changer le répertoire courant...³

Penser à changer le répertoire de travail (ou “répertoire courant”) **au début de chaque nouvelle session de travail.**

³. L'intitulé des menus change selon le système d'exploitation. Sous Mac, il faut aller dans Divers > Changer le Répertoire de Travail ou presser command+D

Le répertoire de travail : exercice

1. Sur votre clé USB, créez un dossier intitulé FormationR_J1.
2. Dans , changez le répertoire de travail en utilisant `setwd()` ou le menu approprié.
3. Vérifiez que la modification a bien été prise en compte en tapant `getwd()`.

les packages

Les packages additionnels

Ce sont des extensions, des groupes de fonctions qui ajoutent des possibilités spécifiques au logiciel.⁴

À ce jour, il en existe plus de 19 000⁵ !

On peut les trouver ici :

<https://cran.r-project.org/web/packages/>

L'installation se fait directement dans R grâce à la fonction :

```
install.packages("nom.du.package")
```

4. analyses multivariées, visualisation 3D, analyses génétiques, cartographie, réseaux neuronaux, ... La liste est très longue.

5. près de 100 000 si on compte les packages d'autres dépôts (*i.e.* Bioconductor, R-forge, GitHub, voir <https://rdrr.io>)

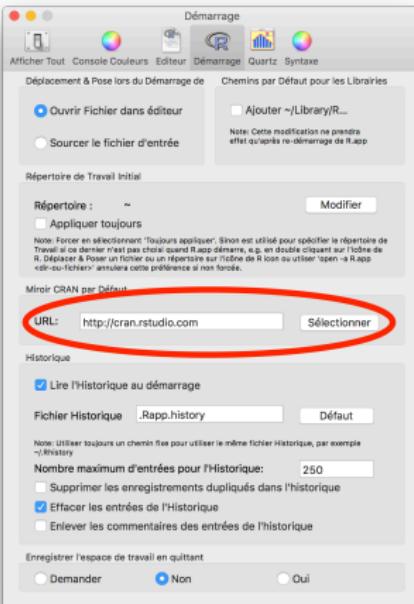
Les packages additionnels

La première fois que la fonction `install.packages()` est utilisée,  peut vous demander de sélectionner un site miroir :



Les packages additionnels

Choisir un site miroir français (Lyon 1 fonctionne très bien). Changer plus tard est toujours possible :



Les packages additionnels

`install.packages()` récupère les fichiers nécessaires sur internet et installe le package.

Cette opération est donc à réaliser **une seule fois** pour chaque package.

Pour utiliser le package, il faut ensuite charger ses fonctions en mémoire, **une fois par session de travail** :

```
library(nom.du.package)
```

Attention, la présence ou l'absence de **guillemets** est importante !

- ▶ `install.packages("nom.du.package")` : guillemets
- ▶ `library(nom.du.package)` : pas de guillemets

Les packages additionnels

La liste des fonctions d'un package est accessible en tapant :

```
library(help="nom.du.package")
```

Certains packages disposent d'une description de leur contenu accessible en tapant :

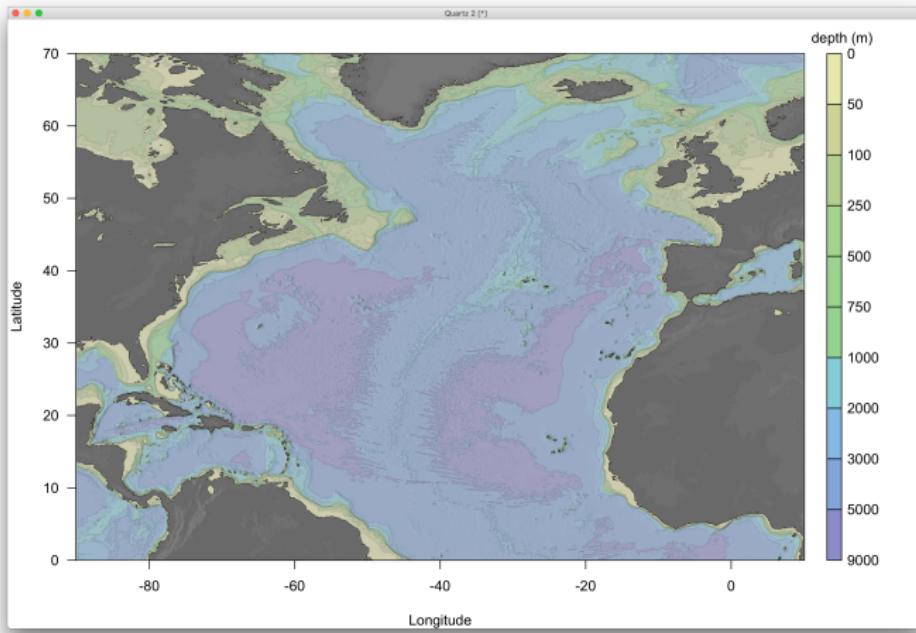
```
?nom.du.package
```

Enfin, certains packages disposent d'une ou plusieurs vignettes :

```
# Afficher la liste des vignettes d'un package  
vignette(package="nom.du.package")  
# Afficher le contenu d'une vignette  
vignette("nom.de.la.vignette")
```

Les packages additionnels : exercice

Le package `marmap` permet de télécharger des données bathymétriques, de les analyser et de construire des cartes :



Les packages additionnels : exercice

1. Installez le package marmap.
2. Chargez-le en mémoire afin de pouvoir accéder à ses fonctions, ses fichiers d'aide, etc.
3. Affichez la liste de ses fonctions.
4. Affichez la liste de ses vignettes. Combien y en a-t-il?
5. Ouvrez la vignette marmap-DataAnalysis pour avoir un aperçu des possibilités offertes par le package.
6. Exécutez les exemples du fichier d'aide de la fonction `create.buffer()` en tapant :

```
example(create.buffer)
```

Les scripts

Les scripts

Un script est un fichier **texte brut** portant l'extension “.R”.

Il contient :

1. des commandes qui peuvent être comprises par 
2. des commentaires, plein de commentaires⁶...

```
# Ceci est un exemple de script
# Création d'un objet contenant les entiers de 1 à 5
a <- 1:5

# affichage d'une table contenant ces entiers,
# leur carré et leur cube
data.frame(x = a, x2 = a^2, x3 = a^3)
```

Les commentaires sont **essentiels** !

6. Ne **jamais** sous-estimer à quel point “le futur soi” peut être **stupide**. Comprendre ce que vous tapez aujourd’hui ne signifie pas que vous le comprendrez toujours dans 6 mois...

Les scripts : un exemple

Sur la clé USB, dans le dossier Data, se trouve un fichier nommé alt.R.

Dans  R, cliquez sur Fichier > Ouvrir un script..., puis naviguez jusqu'au script.

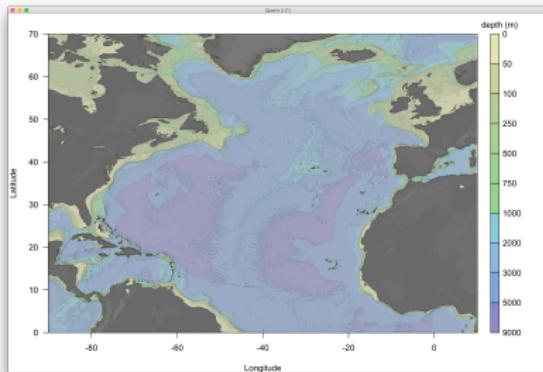
Il doit s'ouvrir dans une nouvelle fenêtre de  R. Vous pouvez envoyer chaque ligne du script dans la console en effectuant des “copier-coller”.

On peut aussi presser les touches **ctrl+R** pour envoyer la ligne active (ou plusieurs lignes sélectionnées) directement dans la console.

Les scripts : un exemple

1. modifiez le répertoire courant pour le faire pointer sur le dossier Data
2. sélectionnez la totalité du script (ctrl+A)
3. et envoyez toutes les lignes dans la console (ctrl+R)

La carte suivante doit être créée :



Les scripts : exercice

1. Placez-vous dans votre répertoire de travail.
2. Créez un nouveau script et donnez-lui un nom (sauvegardez-le dans votre répertoire de travail).
3. Tapez toutes les commandes permettant d'installer marmap, de le charger en mémoire, d'afficher la liste de ses fonctions, et toutes les autres étapes vues précédemment.
4. N'oubliez pas d'ajouter autant de commentaires que nécessaire.
5. Sauvegardez.

Les sauvegardes

Que faut-il sauvegarder?

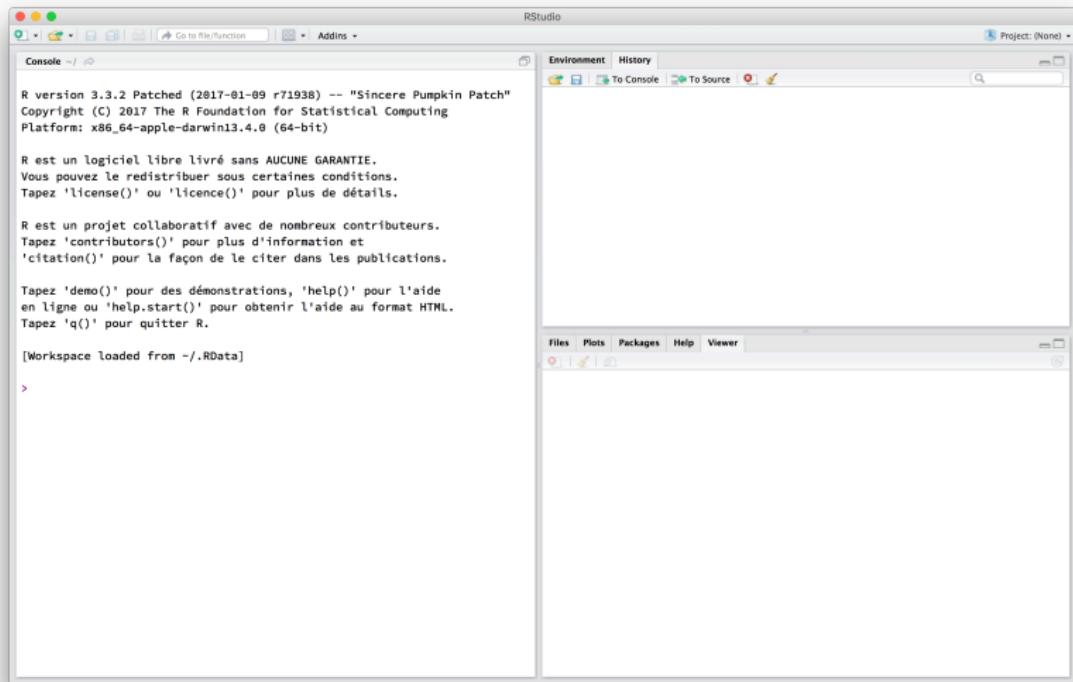
3 choses...

1. Indispensable : vos scripts!
2. Parfois : votre session de travail.
3. Rarement : l'historique des commandes.

Et avec R ?

R : prise en main

1. L'interface



: prise en main

1. L'interface

Les **avantages** :

1. plus de choses sont accessibles en cliquant (installation des packages, consultation de l'aide)
2. toutes les fenêtres sont regroupées
3. l'interface est plus facile à personnaliser (agencement des fenêtres, thèmes)
4. l'éditeur de scripts est **bien supérieur** : coloration syntaxique, auto-complétion, parenthèses, arguments des fonctions, R Notebook...

Les **inconvénients** :

1. plante (légèrement) plus souvent que R
2. gestion des graphiques parfois inconsistente
3. plus facile de prendre de mauvaises habitudes (trop de clics possibles!)
4. Consomme plus de batterie

R : prise en main

2. Les scripts

1. Ouvrez le script que vous avez créé dans R.
2. Envoyez les commandes dans la console de R.
3. Personalisez l'interface à votre goût (Tools > Global options... > Pane layout)
4. Personalisez l'interface à votre goût (Tools > Global options... > Appearance)
5. Tapez des commandes déjà vues dans le script et dans la console. Expérimitez.

R

: prise en main

3. Le répertoire de travail

Afficher le chemin du répertoire de travail (**get working directory**)

```
getwd()
```

```
[1] "/Users/bsimonbo/TAF/Formations/Initiation_R"
```

Spécifier un nouveau répertoire de travail (**set working directory**)

```
setwd("~/Desktop/MonDossier")
```

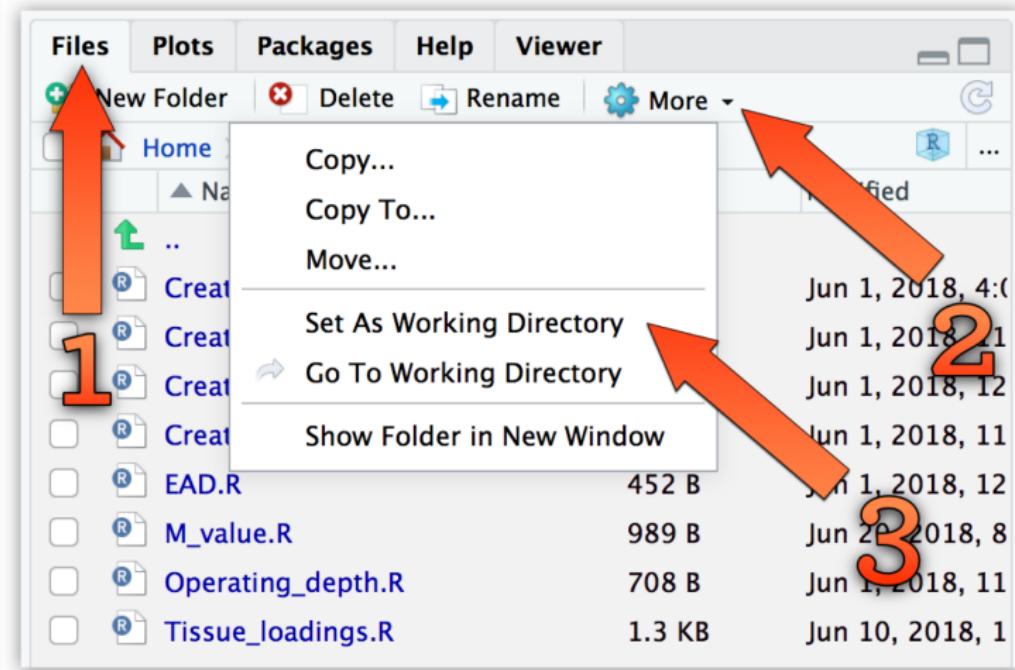
Ou, dans R, via le menu Session > Set Working Directory > Choose Directory...⁷

Penser à changer le répertoire de travail (ou “répertoire courant”) au début de chaque nouvelle session de travail.

7. L'intitulé des menus peut changer selon le système d'exploitation, la langue ou la version de R utilisée.

R : prise en main

3. Le répertoire de travail



R : prise en main

4. Les packages

The screenshot shows the RStudio interface with the 'Packages' tab selected. The 'Install' button in the top-left corner of the toolbar is highlighted with a red box. Below the toolbar, there are three tabs: 'Install' (highlighted), 'Update', and 'Packrat'. A search bar and a refresh icon are also present. The main area displays the 'System Library' with a table of packages. The columns are 'Name', 'Description', and 'Version'. Each package entry includes a checkbox and a remove icon (an 'X').

Name	Description	Version
abind	Combine Multidimensional Arrays	1.4-5
acepack	ACE and AVAS for Selecting Multiple Regression Transformations	1.4.1
ade4	Analysis of Ecological Data: Exploratory and Euclidean Methods in Environmental Sciences	1.7-11
adegenet	Exploratory Analysis of Genetic and Genomic Data	2.1.1
adehabitatMA	Tools to Deal with Raster Maps	0.3.12
AICmodavg	Model Selection and Multimodel Inference Based on (Q)AIC(c)	2.1-1
animation	A Gallery of Animations in Statistics and Utilities to Create Animations	2.5
ape	Analyses of Phylogenetics and Evolution	5.1
assertthat	Easy Pre and Post Assertions	0.2.0
backports	Reimplementations of Functions Introduced Since R-3.0.0	1.1.2
base64enc	Tools for base64 encoding	0.1-3
BH	Boost C++ Header Files	1.66.0-1

R : prise en main

5. Trouver de l'aide

Les méthodes vues dans R fonctionnent également dans R

The screenshot shows the R Help browser interface. The top menu bar includes 'Files', 'Plots', 'Packages', 'Help' (which is highlighted with a red box), and 'Viewer'. A search bar at the top right contains the text 'wilcoxon' (also highlighted with a red box). Below the menu, a search results page is displayed with the title 'Search Results' and a large blue R logo. The search string 'wilcoxon' is shown. Under 'Help pages:', there is a list of entries:

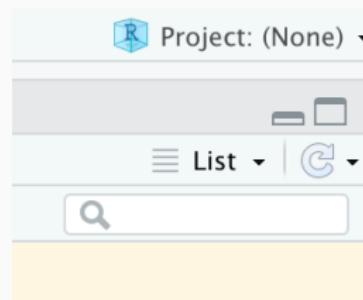
- [coin::oneway.test](#) Two- and K-Sample Location Tests
- [coin::sign_test](#) Symmetry Tests
- [rcompanion::wilcoxonOneSampleR](#) r effect size for Wilcoxon one-sample signed-rank test
- [rcompanion::wilcoxonPairedR](#) r effect size for Wilcoxon two-sample paired signed-rank test
- [rcompanion::wilcoxonR](#) r effect size for Wilcoxon two-sample rank-sum test
- [stats::SignRank](#) Distribution of the Wilcoxon Signed Rank Statistic
- [stats::Wilcoxon](#) Distribution of the Wilcoxon Rank Sum Statistic
- [stats::pairwise.wilcox.test](#) Pairwise Wilcoxon Rank Sum Tests
- [stats::wilcox.test](#) Wilcoxon Rank Sum and Signed Rank Tests

R

: prise en main

6. Les Rprojects

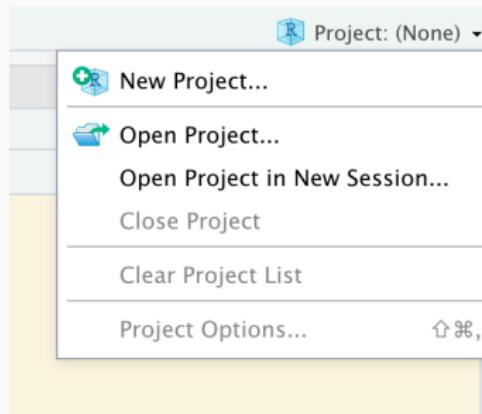
Une excellente façon d'**organiser** les fichiers d'un projet et d'oublier les complications liées aux répertoires de travail.



R : prise en main

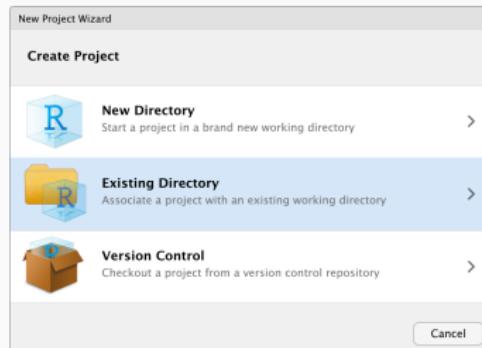
6. Les Rprojects

Une excellente façon d'**organiser** les fichiers d'un projet et d'oublier les complications liées aux répertoires de travail.



6. Les Rprojects

Une excellente façon d'**organiser** les fichiers d'un projet et d'oublier les complications liées aux répertoires de travail.



Les objets dans R

1. Les **vecteurs**

Les vecteurs : la fonction `c()`

`c()` : collection d'éléments qui sont tous du même type

Vecteur numérique (d'entiers) :

```
c(2, 4, 1, 6, 7, 8)  
[1] 2 4 1 6 7 8
```

Vecteur de caractères :

```
c("rouge", "vert", "bleu", "bleu")  
[1] "rouge" "vert"   "bleu"   "bleu"
```

Vecteur logique (vrais/faux) :

```
c(TRUE, TRUE, FALSE, FALSE, T, T, F)  
[1]  TRUE  TRUE FALSE FALSE  TRUE  TRUE FALSE
```

Les vecteurs : la fonction c()

c() : collection d'éléments qui sont tous du même type

Si les éléments sont incompatibles, R essaie de se débrouiller au mieux :

```
c(1, 2, 4, "rouge")  
[1] "1"      "2"      "4"      "rouge"
```

```
c(1, 0, 3, TRUE, FALSE, TRUE)  
[1] 1 0 3 1 0 1
```

```
c(1, 0, 3, TRUE, FALSE, TRUE, "vert")  
[1] "1"      "0"      "3"      "TRUE"   "FALSE"  "TRUE"   "vert"
```

Les vecteurs : la fonction `c()`

`c()` : collection d'éléments qui sont tous du même type

`c()` est une fonction.

Les fonctions possèdent (presque toutes) des arguments séparés par des virgules.

Pour ré-utiliser les résultats produits par une fonction, on leur donne un nom : on les assigne dans un objet grâce à la notation “`<-`” :

```
taille <- c(165, 162, 184, 191, 174, 174, 188, 157)
taille
[1] 165 162 184 191 174 174 188 157
mean(taille)
[1] 174.375
```

Les vecteurs : fonctions utiles

ls() et rm()

ls(), pour lister les objets créés au cours d'une session de travail :

```
ls()  
[1] "a"           "make_dfs"      "taille"  
[4] "theme_benoit" "thm"
```

rm(), pour supprimer (remove) des objets :

```
rm("a")  
a  
Error in eval(expr, envir, enclos): objet 'a' introuvable
```

```
ls()  
[1] "make_dfs"      "taille"       "theme_benoit"  
[4] "thm"
```

Les vecteurs : fonctions utiles

`class()`

Affiche la **classe** d'un objet :

```
x <- c(1, 4, 12, 8)
class(x)

[1] "numeric"
```

```
y <- c("Petit", "Moyen", "Grand", "Grand", "Petit")
class(y)

[1] "character"
```

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

[1] "logical"
```

Les vecteurs : fonctions utiles

Opérations classiques

Lorsqu'on dispose d'un vecteur numérique, les opérations arithmétiques classiques s'appliquent à chaque élément du vecteur :

```
x  
[1] 1 4 12 8  
  
2 * x  
[1] 2 8 24 16  
  
x^3  
[1] 1 64 1728 512  
  
log(x, base = 10)  
[1] 0.000000 0.602060 1.079181 0.903090
```

Les vecteurs : fonctions utiles

length() et dim()

Dans R, un vecteur n'a pas de dimension. Il a en revanche une longueur :

```
y  
[1] "Petit" "Moyen" "Grand" "Grand" "Petit"  
  
dim(y)  
NULL  
  
length(y)  
[1] 5
```

Comme TRUE et FALSE, NULL est un **mot réservé**.

Les vecteurs : fonctions utiles

Le recyclage

On peut additionner, soustraire, multiplier et diviser des vecteurs entre eux. Ces opérations se font **terme à terme**.

Les vecteurs doivent avoir la **même longueur**.

Sinon, R recycle le vecteur le plus court et affiche un message d'avertissement :

```
a <- c(1, 2, 3, 4)
b <- c(3, 6, 9)
a+b
```

Warning in a + b: la taille d'un objet plus long n'est pas multiple de la taille d'un objet plus court

```
[1] 4 8 12 7
```

Les vecteurs : fonctions utiles

L'opérateur ":"

Il sert essentiellement à générer des suites d'entiers, croissantes ou décroissantes :

```
croiss <- 1:10  
decroiss <- 30:17  
neg <- -4:3
```

```
croiss
```

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

```
decroiss
```

```
[1] 30 29 28 27 26 25 24 23 22 21 20 19 18 17
```

```
neg
```

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

Les vecteurs : fonctions utiles

L'opérateur ":"

Il est aussi possible d'utiliser des nombres à virgule, mais le pas est toujours de 1 :

```
1.3:9.3
```

```
[1] 1.3 2.3 3.3 4.3 5.3 6.3 7.3 8.3 9.3
```

Si ça ne tombe pas juste, le vecteur est tronqué :

```
1.3:9.2
```

```
[1] 1.3 2.3 3.3 4.3 5.3 6.3 7.3 8.3
```

Les vecteurs : fonctions utiles

La fonction seq()

Elle permet de créer des **séquences** régulières :

```
seq(from = 0, to = 25, by = 5)  
[1] 0 5 10 15 20 25
```

Comme précédemment, si ça ne tombe pas juste, le vecteur est tronqué :

```
seq(from = 0, to = 25, by = 4)  
[1] 0 4 8 12 16 20 24
```

Les vecteurs : fonctions utiles

La fonction seq()

Les séquences décroissantes sont possibles également, mais attention au signe du pas :

```
seq(from = 25, to = 0, by = 5)
```

```
Error in seq.default(from = 25, to = 0, by = 5): signe incorrect de l'argument 'by'
```

```
seq(from = 25, to = 0, by = -5)
```

```
[1] 25 20 15 10 5 0
```

Les pas non entiers sont possibles également :

```
seq(from = 1, to = 3, by = 0.2)
```

```
[1] 1.0 1.2 1.4 1.6 1.8 2.0 2.2 2.4 2.6 2.8 3.0
```

Les vecteurs : fonctions utiles

La fonction seq()

Autre argument utile de cette fonction : length.out

```
seq(from = 0, to = 100, length.out = 11)
[1] 0 10 20 30 40 50 60 70 80 90 100
```

```
seq(from = 0, to = 100, length.out = 6)
[1] 0 20 40 60 80 100
```

À quoi sert-il? Faites des essais, consultez l'aide de la fonction...

```
?seq
```

Les vecteurs : fonctions utiles

La fonction rep()

Permet de répliquer les valeurs fournies en guise de premier argument :

```
rep(c(1, 7, 6, 7), times = 4)
[1] 1 7 6 7 1 7 6 7 1 7 6 7 1 7 6 7
```

On peut répéter chaque élément un nombre de fois différent :

```
rep(c(1, 7, 6), times = c(4, 3, 2))
[1] 1 1 1 1 7 7 7 6 6
```

Enfin, il est possible de combiner les arguments times et each :

```
rep(c(1, 7, 6), each = 3, times = 2)
[1] 1 1 1 7 7 7 6 6 6 1 1 1 7 7 7 6 6 6
```

Les vecteurs

Exercices

À l'aide des fonctions `c()`, `rep()`, `seq()` et de l'opérateur “`:`”, créez les objets suivants :

```
vec1
```

```
[1] 0 6 6 12 12 12 18 18 18 18 24 24 24 24 24 24
```

```
vec2
```

```
[1] 1 1 2 2 8 8 10 10 1 1 2 2 8 8 10 10
```

```
vec3
```

```
[1] 8 8 8 8 7 7 6 6 6 5 5 5 4 4 4 4 3 3 3
```

Les vecteurs : solutions

Création du vecteur 1

```
vec1 <- rep(seq(from = 0, to = 24, by = 6), times = 1:5)
```

Création du vecteur 2

```
vec2 <- rep(c(1, 2, 8, 10), each = 2, times = 2)
```

Création du vecteur 3

```
vec3 <- rep(8:3, times = c(4, 3, 4, 3, 4, 3))
```

ou alors :

```
vec3 <- rep(8:3, times = rep(4:3, 3))
```

Les objets dans R

2. Les facteurs

Les facteurs

La fonction `factor()`

Un facteur est un vecteur qui possède un **attribut supplémentaire** :

```
vec <- c(1,4,7,5,6,4,1,7,7)
vec
[1] 1 4 7 5 6 4 1 7 7
```

```
fac <- factor(vec)
fac
[1] 1 4 7 5 6 4 1 7 7
Levels: 1 4 5 6 7
```

L'attribut “`Levels`” indique quelles sont les valeurs qui peuvent être observées dans le facteur.

Les facteurs sont utilisées pour coder des **variables catégorielles** dont le nombre de catégories est fini et connu.

Les facteurs

La fonction `levels()`

Elle permet d'afficher et de modifier les niveaux d'un vecteur :

```
fac  
[1] 1 4 7 5 6 4 1 7 7  
Levels: 1 4 5 6 7  
  
levels(fac)  
[1] "1" "4" "5" "6" "7"
```

```
levels(fac) <- c("a","b","c","d","e")  
fac  
[1] a b e c d b a e e  
Levels: a b c d e
```

Les facteurs : un exemple

La fonction `factor()`

200 juges ont goûté à l'aveugle un vin de Bordeaux grand cru classé. Ils ont attribué une note de 1 à 4, 1 étant la note minimale et 4 la note maximale. Le vecteur `wine` contient les notes des 200 juges.

```
wine
```

```
[1] 3 4 4 3 3 4 2 2 3 3 3 3 3 3 3 3 3 2 3 2 4 4 3 2 3 3 3  
[28] 3 3 3 3 3 3 1 3 4 3 3 3 4 3 2 4 3 2 3 3 3 3 2 3 3 4 4  
[55] 3 3 4 4 2 4 3 3 4 3 3 2 2 3 3 3 3 4 4 4 3 3 4 3 3 3 3 3  
[82] 3 2 2 3 3 2 3 3 2 3 3 3 3 3 3 4 3 4 4 3 3 3 3 3 3 4 1 2  
[109] 3 4 3 3 3 4 2 2 4 3 3 3 3 3 3 3 4 3 3 3 3 3 3 3 3 3 4  
[136] 3 3 3 3 3 4 4 3 3 1 4 4 3 2 3 3 4 3 3 3 3 3 3 2 4 2 3  
[163] 3 4 3 4 4 3 3 3 3 4 2 4 4 2 1 3 3 3 4 3 3 4 3 3 2 4 3  
[190] 3 3 3 4 4 3 1 4 3 4 3
```

Les facteurs : un exemple

La fonction `summary()`

La fonction `summary()` permet d'afficher un résumé des données :

```
summary(wine)
```

Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
1.000	3.000	3.000	3.055	3.000	4.000

```
class(wine)
```

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

`wine` est un vecteur numérique. La fonction `summary()` calcule donc la moyenne, la médiane, les valeurs extrêmes, etc.

Les facteurs : un exemple

La fonction `factor()`

Les notes attribuées correspondent en fait à des **catégories** plus qu'à une variable continue. Le résumé suivant est plus approprié :

```
wine.f <- factor(wine)
class(wine.f)
[1] "factor"
```

```
summary(wine.f)
 1   2   3   4
 5 24 126 45
```

`summary()` est une fonction **générique** : elle ne renvoie pas la même chose selon le type d'argument fourni. Pour un facteur, elle présente le nombre d'observations pour chaque valeur possible (**niveau**) du facteur.

Les facteurs : un exemple

```
levels()
```

Pour rendre les notes plus compréhensibles, on peut utiliser des niveaux non numériques :

```
levels(wine.f) <- c("Mauvais", "Moyen", "Bon", "Excellent")
head(wine.f, 20) # Affiche les 20 premières valeurs

[1] Bon      Excellent Excellent Bon      Bon
[6] Excellent Moyen     Moyen     Bon      Bon
[11] Bon      Bon       Bon       Bon      Bon
[16] Bon      Bon       Moyen     Bon      Moyen
Levels: Mauvais Moyen Bon Excellent
```

```
summary(wine.f)

Mauvais      Moyen      Bon  Excellent
      5          24        126        45
```

Les facteurs : un autre exemple

Ordonner un facteur

Soit le facteur suivant :

```
mois <- c("jan", "fev", "jan", "mar", "mai", "avr")
mois.f <- factor(mois)
mois.f

[1] jan fev jan mar mai avr
Levels: avr fev jan mai mar
```

Les niveaux ne sont pas dans le bon ordre.

C'est un problème pour les graphiques. Pour y remédier :

```
mois.f <- factor(mois, levels = c("jan", "fev", "mar",
                                    "avr", "mai"))
mois.f

[1] jan fev jan mar mai avr
Levels: jan fev mar avr mai
```

Les objets dans R

3. Les matrices

Les matrices

La fonction `matrix()`

Comme un vecteur, une matrice contient des éléments qui sont tous du même type.

Contrairement aux vecteurs, les matrices ont une dimension : un nombre de ligne et un nombre de colonnes.

```
mat <- matrix(1:12, ncol=4)
mat

      [,1]  [,2]  [,3]  [,4]
[1,]    1     4     7    10
[2,]    2     5     8    11
[3,]    3     6     9    12

dim(mat)
[1] 3 4
```

Les matrices

La fonction `matrix()`

La fonction `matrix()` peut prendre jusqu'à 5 arguments (consultez l'aide de la fonction).

On peut ainsi spécifier :

1. un **vecteur** contenant les coefficients de la matrice,
2. un nombre de lignes (`nrow`)...
3. ... et/ou de colonnes (`ncol`),
4. le sens de remplissage (par défaut, `byrow = FALSE`)
5. une **liste** contenant les noms de lignes et de colonnes de la matrice

Les matrices

La fonction `matrix()`

```
coef <- c(1, 3, 8, 12, 13, 14)
matrix(coef, ncol = 2)

[,1] [,2]
[1,]    1   12
[2,]    3   13
[3,]    8   14

matrix(coef, nrow = 2)

[,1] [,2] [,3]
[1,]    1    8   13
[2,]    3   12   14

matrix(coef, nrow = 2, byrow = TRUE)

[,1] [,2] [,3]
[1,]    1    3    8
[2,]   12   13   14
```

Les matrices

La fonction `matrix()`

Si les dimensions indiquées ne correspondent pas à la longueur du vecteur, R fait au mieux :

- ▶ si le vecteur est trop long, il le tronque
- ▶ si le vecteur est trop court, il recycle

```
matrix(coef, ncol = 2, nrow = 2)
```

```
Warning in matrix(coef, ncol = 2, nrow = 2): data length differs
from size of matrix: [6 != 2 x 2]
```

```
[,1] [,2]
[1,]    1    8
[2,]    3   12
```

```
matrix(coef, ncol = 3, nrow = 3)
```

```
Warning in matrix(coef, ncol = 3, nrow = 3): data length differs
from size of matrix: [6 != 3 x 3]
```

```
[,1] [,2] [,3]
[1,]    1   12    1
[2,]    3   13    3
[3,]    8   14    8
```

Les matrices

La fonction `matrix()`

Comme pour les vecteurs, on peut créer des matrices de vrais/faux et de caractères :

```
matrix(c("rouge", "bleu", "jaune", "violet"), ncol = 2)

[,1]      [,2]
[1,] "rouge"  "jaune"
[2,] "bleu"   "violet"

matrix(c(T,F,T,T,F,T), ncol = 2, byrow = TRUE)

[,1]      [,2]
[1,] TRUE FALSE
[2,] TRUE  TRUE
[3,] FALSE TRUE
```

Les objets dans R

4. Les tableaux

Les tableaux de données

La fonction `data.frame()`

Dans R, la plupart des tableaux que vous manipulerez seront des `data.frame()` et non des `matrix()`.

Les variables peuvent avoir des classes différentes et sont présentées en colonnes. Exemple : création de 2 facteurs et d'un vecteur :

```
day <- factor(c("lundi", "mardi", "mercredi", "jeudi",
                 "vendredi", "samedi", "dimanche"))
sun <- factor(c(TRUE, TRUE, FALSE, FALSE, FALSE, TRUE, TRUE))
wind <- c(0, 12, 32, 45, 21, 11, 14)
```

Les tableaux de données

La fonction `data.frame()`

On peut combiner ces 3 objets dans un `data.frame` :

```
meteo <- data.frame(jour = day, soleil = sun, vent = wind)
meteo

      jour soleil vent
1     lundi   TRUE    0
2     mardi   TRUE   12
3 mercredi FALSE   32
4     jeudi FALSE   45
5 vendredi FALSE   21
6     samedi  TRUE   11
7 dimanche  TRUE   14

class(meteo)

[1] "data.frame"
```

Les tableaux de données

La fonction `summary()`

Les tableaux de données que nous importerons bientôt dans  à partir de tableurs auront ce format.

La fonction `summary()` renvoie un résumé pour chaque colonne d'un `data.frame`. Le type de résumé dépend du contenu des colonnes.

```
summary(meteo)
```

	jour	soleil	vent
dimanche:1		FALSE:3	Min. : 0.00
jeudi :1		TRUE :4	1st Qu.:11.50
lundi :1			Median :14.00
mardi :1			Mean :19.29
mercredi:1			3rd Qu.:26.50
samedi :1			Max. :45.00
vendredi:1			

Les tableaux de données

Exercice

Sur la diapo précédente, les jours de la semaine sont classés par ordre alphabétique. Créez un data.frame nommé meteo2 contenant les mêmes données mais dans lequel les jours sont ordonnés chronologiquement.

La fonction `summary()` doit renvoyer ceci :

```
summary(meteo2)
```

	jour	soleil	vent
lundi	:1	FALSE:3	Min. : 0.00
mardi	:1	TRUE :4	1st Qu.:11.50
mercredi	:1		Median :14.00
jeudi	:1		Mean :19.29
vendredi	:1		3rd Qu.:26.50
samedi	:1		Max. :45.00
dimanche	:1		

Les objets dans R

5. Les listes

Les listes

La fonction list()

Les listes sont les objets les plus “souples” de R.

On peut regrouper dans une liste tous les autres types d'objets vus précédemment.

La plupart des fonctions statistiques du logiciel renvoient leurs résultats sous forme de liste.

```
my.list <- list(month = mois.f, size = taille,  
                 weather=meteo2)  
summary(my.list)
```

	Length	Class	Mode
month	6	factor	numeric
size	8	-none-	numeric
weather	3	data.frame	list

Les listes

La fonction list()

```
my.list  
  
$month  
[1] jan fev jan mar mai avr  
Levels: jan fev mar avr mai  
  
$size  
[1] 165 162 184 191 174 174 188 157  
  
$weather  
      jour soleil vent  
1     lundi   TRUE    0  
2     mardi   TRUE   12  
3 mercredi FALSE   32  
4     jeudi FALSE   45  
5 vendredi FALSE   21  
6     samedi  TRUE   11  
7 dimanche  TRUE   14
```

Exercice

Exercice

Listes et tirages aléatoires

Créez une liste contenant 3 éléments :

1. Un vecteur nommé uniforme qui contient 30 valeurs tirées au hasard entre 0 et 100 dans une loi de distribution uniforme.
2. Un vecteur nommé uniforme.crois qui contient ces mêmes 30 valeurs triées en ordre croissant.
3. Un vecteur nommé uniforme.decr qui contient ces mêmes 30 valeurs triées en ordre décroissant.

Vous aurez besoin de rechercher sur internet et dans l'aide de  comment tirer des valeurs dans une loi de distribution donnée (ici, la loi uniforme, chaque valeur entre 0 et 100 aura la même probabilité d'être tirée au hasard), et comment trier un vecteur en ordre croissant et décroissant.

Bilan

Bilan

Nous avons vu...

Les bonnes pratiques dans R (répertoire de travail, scripts...)

5 types d'objets : vecteurs, facteurs, matrices, tableaux de données et listes.

Les fonctions suivantes :

sqrt()	example()	factor()
log()	c()	levels()
help.search()	mean()	summary()
getwd()	ls()	head()
setwd()	rm()	matrix()
install.packages()	class()	data.frame()
library()	seq()	list()
vignette()	rep()	dim()

Les opérateurs suivants :

+ - * / ^ <- : ?

D'autres fonctions utiles

pour travailler avec les tableaux...

Nom	Rôle
colnames()	nom des colonnes d'un tableau
names()	nom des colonnes d'un tableau ou des éléments d'une liste
rownames()	nom des lignes d'un tableau
head()	affiche les premières lignes d'un tableau
tail()	affiche les dernières lignes d'un tableau
str()	affiche la structure d'un objet
subset()	pour récupérer des lignes spécifiques d'un tableau
attach()	“attache” un tableau
detach()	“détache” un tableau

Et beaucoup d'autres dans le tidyverse...

L'indexation

L'indexation

Par position

L'indexation permet d'accéder à des éléments spécifiques au sein d'objets en contenant plusieurs.

On utilise la notation “`objet[position]`”

Par exemple, pour accéder au 4^e élément du vecteur `mois` :

```
mois[4]
```

```
[1] "mar"
```

Pour mémoire :

```
mois
```

```
[1] "jan" "fev" "jan" "mar" "mai" "avr"
```

L'indexation

Par position

On peut utiliser des vecteurs en guise de position.

```
mois  
[1] "jan" "fev" "jan" "mar" "mai" "avr"  
mois[c(2,4)]  
[1] "fev" "mar"
```

```
mois[c(2,4,4,4)]  
[1] "fev" "mar" "mar" "mar"
```

```
mois[6:2]  
[1] "avr" "mai" "mar" "jan" "fev"
```

L'indexation

Par position

La notation -position permet de retirer un ou des éléments :

```
mois  
[1] "jan" "fev" "jan" "mar" "mai" "avr"  
mois[-1]  
[1] "fev" "jan" "mar" "mai" "avr"  
mois[-c(1,3)]  
[1] "fev" "mar" "mai" "avr"
```

Attention à ne pas donner une position qui n'existe pas :

```
mois[10]  
[1] NA
```

L'indexation

Par position

Pour les matrices et les tableaux, il faut 2 positions : le numéro de la ligne **puis** le numéro de la colonne :

```
mat  
      [,1] [,2] [,3] [,4]  
[1,]    1     4     7    10  
[2,]    2     5     8    11  
[3,]    3     6     9    12
```

```
mat[1,2]
```

```
[1] 4
```

```
mat[c(1,3), -2]
```

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

L'indexation

Par position

Pour les matrices et les tableaux, il faut 2 positions : le numéro de la ligne **puis** le numéro de la colonne :

```
meteo2
```

	jour	soleil	vent
1	lundi	TRUE	0
2	mardi	TRUE	12
3	mercredi	FALSE	32
4	jeudi	FALSE	45
5	vendredi	FALSE	21
6	samedi	TRUE	11
7	dimanche	TRUE	14

```
meteo2[6:7, -2]
```

	jour	vent
6	samedi	11
7	dimanche	14

L'indexation

Par position

Omettre le numéro de ligne (ou de colonne) signifie que l'on souhaite récupérer **toutes** les lignes (ou toutes les colonnes) :

```
meteo2[6:7,]
```

	jour	soleil	vent
6	samedi	TRUE	11
7	dimanche	TRUE	14

```
meteo2[,3]
```

```
[1] 0 12 32 45 21 11 14
```

```
meteo2[,2]
```

```
[1] TRUE TRUE FALSE FALSE FALSE TRUE TRUE  
Levels: FALSE TRUE
```

L'indexation

Par position

Pour les tableaux et matrices disposant de **noms de colonnes**, il est possible d'utiliser la notation nom.objet\$nom.colonne :

```
meteo2[,2]  
[1] TRUE TRUE FALSE FALSE FALSE TRUE TRUE  
Levels: FALSE TRUE  
  
meteo2$soleil  
[1] TRUE TRUE FALSE FALSE FALSE TRUE TRUE  
Levels: FALSE TRUE
```

```
meteo2$jour  
[1] lundi     mardi      mercredi  jeudi      vendredi samedi  
[7] dimanche  
7 Levels: lundi mardi mercredi jeudi vendredi ... dimanche
```

L'indexation

Par position

Pour les listes, on peut utiliser le nom des éléments ou la notation plus complexe “[[]]” :

```
my.list$size  
[1] 165 162 184 191 174 174 188 157  
  
my.list[[2]]  
[1] 165 162 184 191 174 174 188 157
```

Ici, pour accéder à certaines tailles :

```
my.list$size[c(1,3,4)]  
[1] 165 184 191  
  
my.list[[2]][-1]  
[1] 162 184 191 174 174 188 157
```

L'indexation

Par condition

Lorsqu'on travaille avec des objets de grande taille, on a fréquemment besoin de récupérer des éléments en fonction de **critères précis**, sans nécessairement savoir où ces éléments se trouvent dans l'objet.

Par exemple, dans le vecteur `wind`, quelles sont les valeurs supérieures à 30 ?

```
wind[wind > 30]
```

```
[1] 32 45
```

L'indexation

Par condition

Les opérateurs permettant de spécifier les conditions sont les suivants :

>	→	supérieur
\geq	→	supérieur ou égal
<	→	inférieur
\leq	→	inférieur ou égal
$=$	→	égal
\neq	→	différent

Par ailleurs, on peut enchaîner plusieurs conditions avec :

&	→	ET logique
	→	OU logique

L'indexation

Par condition

Dans meteo2, quelles sont les lignes pour lesquelles on avait du soleil?

```
meteo2[meteo2$soleil==TRUE, ]
```

	jour	soleil	vent
1	lundi	TRUE	0
2	mardi	TRUE	12
6	samedi	TRUE	11
7	dimanche	TRUE	14

Y a-t-il des lignes pour lesquelles il y avait du soleil et plus de 10 km/h de vent?

```
meteo2[meteo2$soleil==TRUE & meteo2$vent > 10, ]
```

	jour	soleil	vent
2	mardi	TRUE	12
6	samedi	TRUE	11
7	dimanche	TRUE	14

L'indexation

Exercice

Installez le package ade4 et chargez-le en mémoire.

Tapez ensuite les commandes suivantes :

```
data(deug)
notes <- deug$tab
```

Vous disposez maintenant d'un tableau contenant les notes de 104 étudiants dans 9 disciplines :

```
dim(notes)
[1] 104   9

names(notes)
[1] "Algebra"      "Analysis"      "Proba"        "Informatic"
[5] "Economy"       "Option1"       "Option2"       "English"
[9] "Sport"
```

L'indexation

Exercice

On cherche à savoir si, comme le veut la croyance populaire, les meilleurs élèves en math sont les plus mauvais en sport....

Pour cela :

1. calculez la moyenne des notes de sport de toute la promotion
2. identifiez à quoi correspond une “bonne note” en algèbre. On considérera comme “bonne”, une note obtenue par le meilleur quart des étudiants.
3. calculez la moyenne des notes de sport pour les étudiants ayant obtenu une note d’algèbre supérieure ou égale à la bonne note identifiée à l’étape précédente

Import / Export...

Importer des données dans R

Les étapes

Pour importer dans R des données issues de tableur, plusieurs étapes sont à respecter :

1. préparer les données dans Excel (ou Open Office Calc)
2. Exporter le fichier au format texte brut
3. Fermer le tableur
4. Importer les données dans R avec la fonction `read.table()`

Nous verrons très vite que dans R, l'étape 2 est inutile, et que plusieurs variantes de l'étape 4 existent.

Importer des données dans R

1. Préparation des données dans Excel

Avant toute chose, faire une **copie** du fichier brut. Ne **jamais** travailler directement sur l'**original**!

1. Placer les variables en colonnes et les observations en ligne
2. Pas de cases vides : remplir avec des NA
3. Placer les noms de colonnes sur la première ligne de la feuille Excel
4. Aucun caractère spécial tel que #, \$, %, &, ^, {}, [], accents, cédille, guillemets, ...
5. Pas d'espaces dans les noms de variables ou de catégories. Les remplacer par des points ou tirets bas
6. Les noms de lignes doivent être uniques
7. Privilégier les noms courts

Importer des données dans R

2. Exporter au format texte brut

Enregistrez dans le répertoire de travail de R le fichier au format texte brut en utilisant :

- ▶ soit Fichier > Enregistrer sous > texte csv
- ▶ soit Fichier > Enregistrer sous > texte séparateur tabulation

Si le format csv est choisi, il faut s'assurer (par exemple en ouvrant le fichier dans le bloc notes de Windows) que le séparateur de colonnes n'est pas le même symbole que celui utilisé pour les décimales.

Importer des données dans R

`read.table()`

R est capable d'importer des fichiers au format **texte brut** grâce à la fonction `read.table()`.

Il existe de nombreuses fonctions dérivées de `read.table()` :

- ▶ `read.delim()`
- ▶ `read.delim2()`
- ▶ `read.csv()`
- ▶ `read.csv2()`
- ▶ ...

Elles font toutes la même chose que `read.table()` mais leurs arguments utilisent des valeurs par défaut différentes.

Importer des données dans R

`read.table()`

La fonction `read.table()` prend plusieurs arguments :

- ▶ `file` : c'est le seul argument qui n'a pas de valeur par défaut. Il faut spécifier le nom (ou le chemin) du fichier à importer. Cela peut être un fichier issu du web.
- ▶ `header` : le fichier contient-il une ligne de titres de colonnes
- ▶ `dec` : quel est le symbole utilisé dans le fichier pour les décimales
- ▶ `sep` : quel est le symbole utilisé dans le fichier pour séparer les colonnes
- ▶ `row.names` : quel est le numéro de colonne contenant les titres des lignes (à supposer qu'il y en ait un)

Importer des données dans R

`read.table()`

Exemple fictif :

```
dat <- read.table("donnees.txt", header = TRUE,  
                  sep = "\t", dec = ",")
```

Dans la pratique, on utilise souvent l'assistant d'importation de R.

Importer des données dans R

Exercice

Prenez le temps de lire [le chapitre 4.2](#) du cours de L2 [en ligne](#).

En guise d'entraînement, essayez d'importer dans R :

1. Les données contenues dans le fichier "dauphin.xls" dans un objet que vous nommerez dauphin
2. Les données contenues dans le fichier "dauphin.csv" dans un objet que vous nommerez dauphin2

Exporter des données depuis R

`write.table()`

Cette fonction est très simple à utiliser :

```
write.table(objet, "NomFichier.txt")
```

Cette commande crée un fichier nommé “NomFichier.txt” dans le répertoire de travail. Ce fichier contiendra les données contenues dans objet.

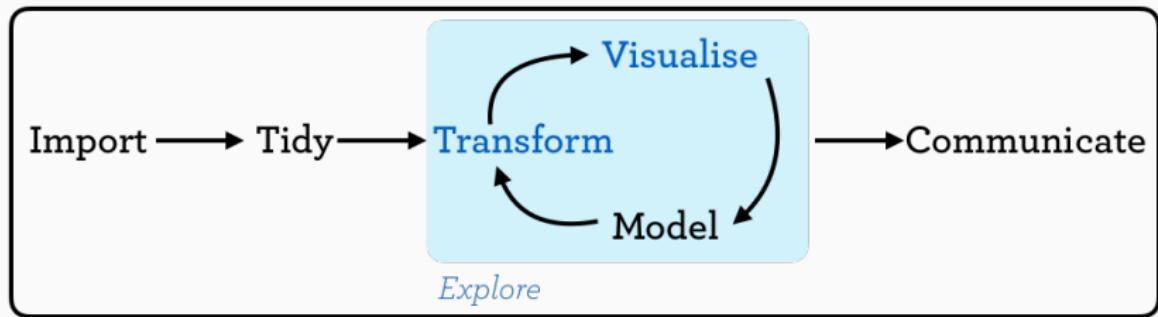
Comme pour `read.table()`, les arguments sont nombreux :

- ▶ `sep`
- ▶ `dec`
- ▶ `row.names`
- ▶ `col.names`
- ▶ `...`

Le **tidyverse** : vue d'ensemble

Les grands principes

C'est quoi les "Data Sciences"?



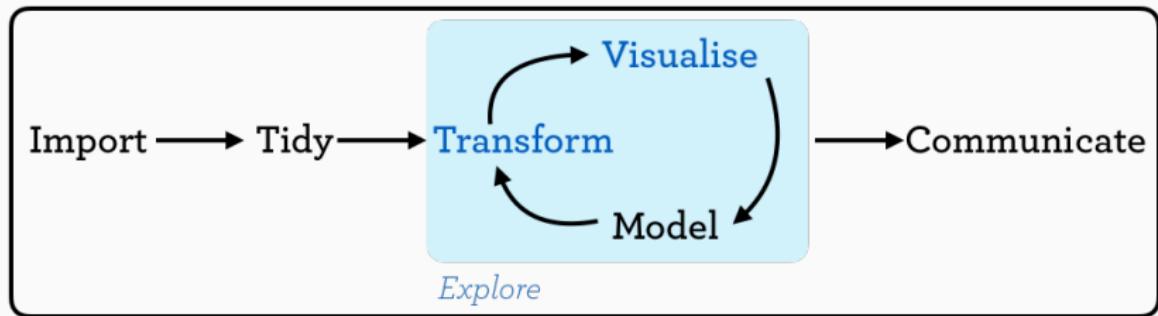
Program

Import :

- ▶ readr
- ▶ readxl
- ▶ haven
- ▶ httr
- ▶ rvest
- ▶ xml2

Les grands principes

C'est quoi les "Data Sciences"?



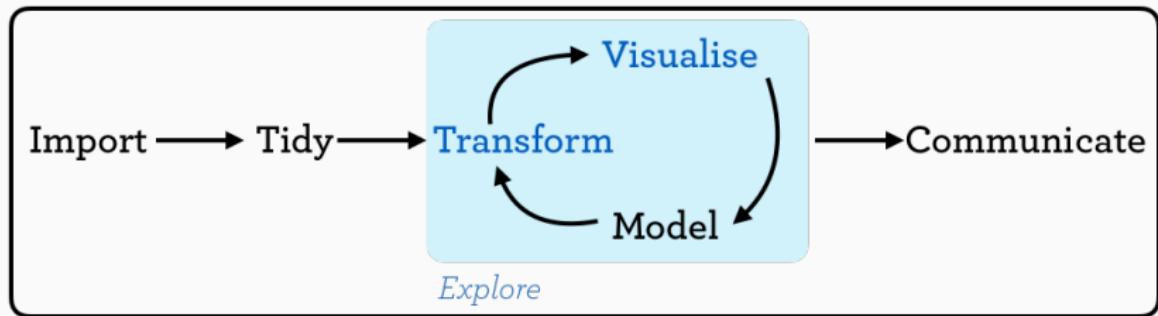
Program

Tidy :

- ▶ tibble
- ▶ tidyverse

Les grands principes

C'est quoi les "Data Sciences"?



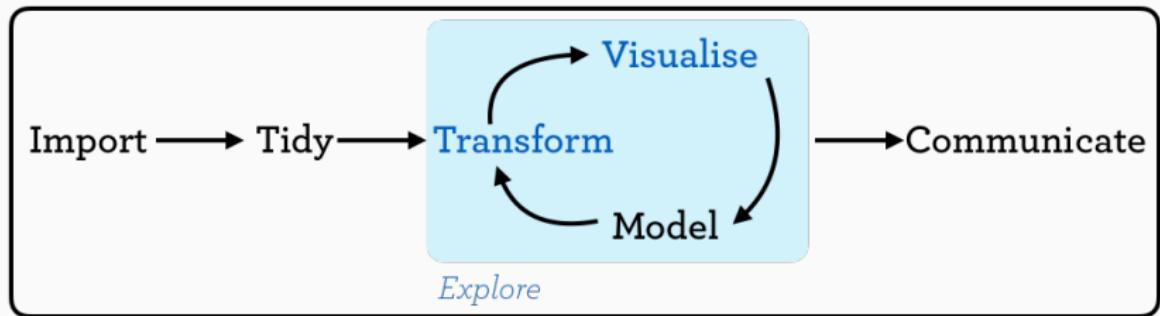
Program

Transform :

- ▶ dplyr
- ▶forcats
- ▶ hms
- ▶ lubridate
- ▶ stringr

Les grands principes

C'est quoi les "Data Sciences"?



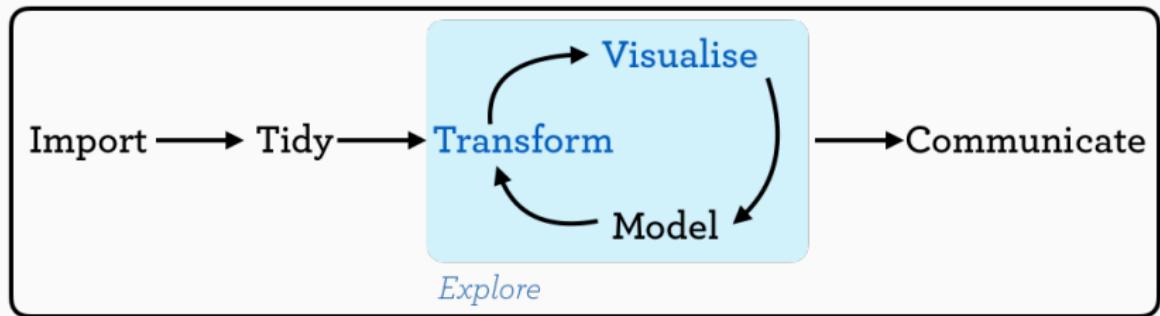
Program

Visualise :

- ▶ ggplot2

Les grands principes

C'est quoi les "Data Sciences"?



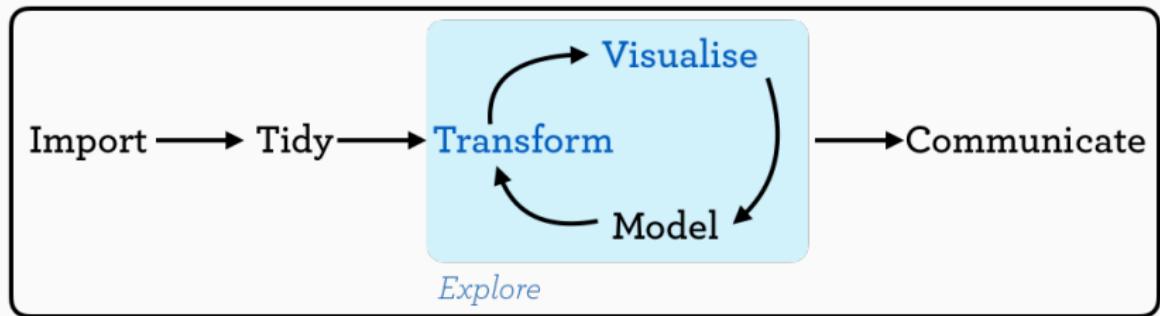
Program

Model :

- ▶ broom
- ▶ modelr

Les grands principes

C'est quoi les "Data Sciences"?



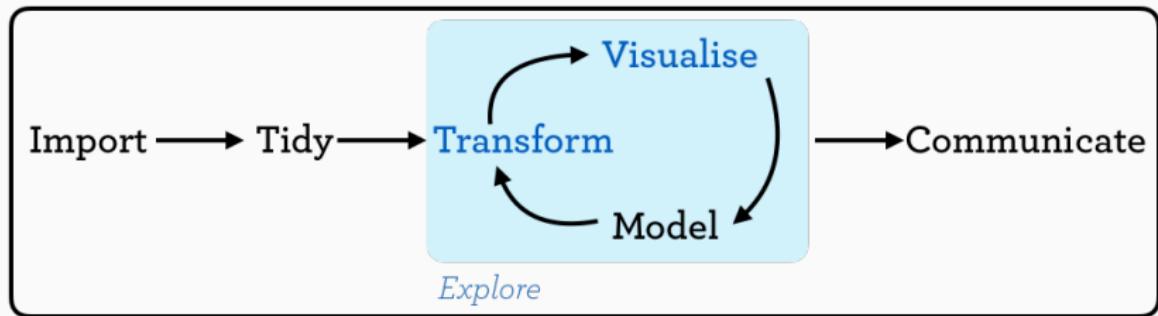
Program

Communicate :

- ▶ knitr
- ▶ rmarkdown
- ▶ blogdown
- ▶ bookdown
- ▶ shiny

Les grands principes

C'est quoi les "Data Sciences"?



Program

Program :

- ▶ purrr
- ▶ magritr

Visualisation :

1. La grammaire des graphiques

1. La grammaire des graphiques

Installation des packages nécessaires

```
## install.packages("tidyverse")
## install.packages("nycflights13")
library(tidyverse)
library(nycflights13)
```

1. La grammaire des graphiques

Principes du package `ggplot2`

Pour faire un graphique avec `ggplot2`, il nous faut au moins les 3 ingrédients suivants :

- ▶ `data` : le nom d'un tableau contenant les données
- ▶ `geom` : les objets géométriques que l'on souhaite voir apparaître sur le graphique (des points, des lignes, des boites à moustache...)
- ▶ `aes` : les attributs esthétiques des objets géométriques (position, taille, couleur, transparence...)

1. La grammaire des graphiques

Les données

Le tableau `flights` du package `nycflights13`:

```
flights

# A tibble: 336,776 x 19
  year month   day dep_time sched_dep_time dep_delay arr_time
  <int> <int> <int>    <int>          <int>     <dbl>     <int>
1 2013     1     1      517            515       2        830
2 2013     1     1      533            529       4        850
3 2013     1     1      542            540       2        923
4 2013     1     1      544            545      -1       1004
5 2013     1     1      554            600      -6        812
6 2013     1     1      554            558      -4        740
7 2013     1     1      555            600      -5        913
8 2013     1     1      557            600      -3        709
9 2013     1     1      557            600      -3        838
10 2013    1     1      558            600      -2        753
# i 336,766 more rows
# i 12 more variables: sched_arr_time <int>, arr_delay <dbl>,
#   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>,
#   dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
#   minute <dbl>, time_hour <dttm>
```

1. La grammaire des graphiques

Les données

Dans un premier temps nous travaillerons uniquement avec les vols de la compagnie **Alaska Airlines** :

```
# On filtre les données de la compagnie AS
alaska_flights <- flights %>%
  filter(carrier == "AS")
```

```
# On s'assure qu'on a un tableau beaucoup plus petit
dim(alaska_flights)

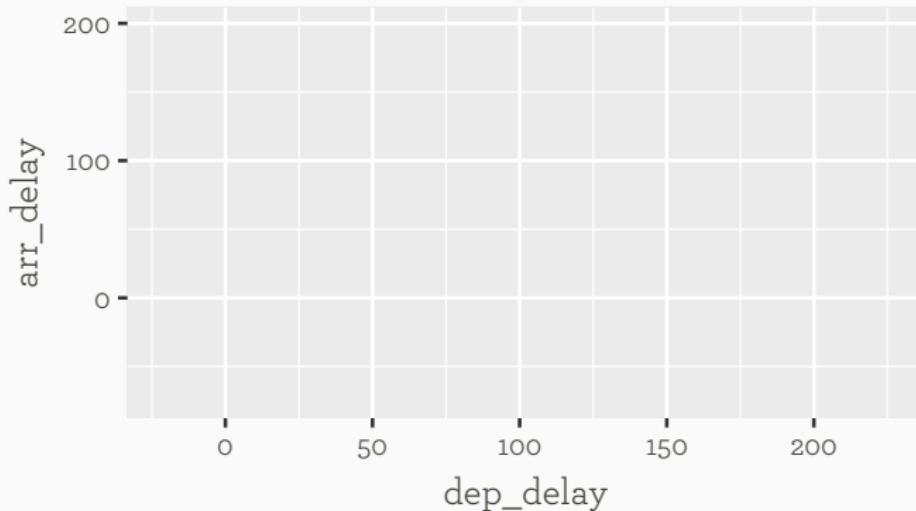
[1] 714 19
```

1. La grammaire des graphiques

Premier graphique

Les retards des vols au décollage et à l'atterrissage sont-ils liés ?

```
ggplot(data = alaska_flights,  
       mapping = aes(x = dep_delay, y = arr_delay))
```

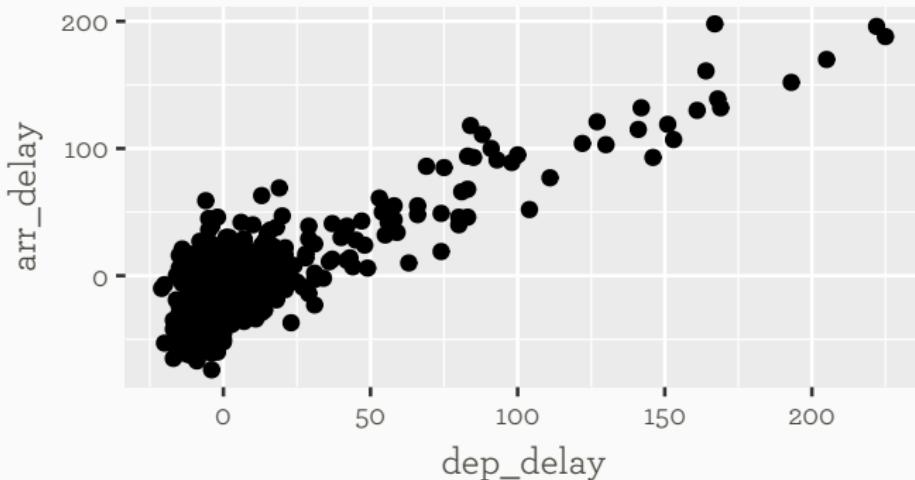


1. La grammaire des graphiques

Premier graphique

```
ggplot(data = alaska_flights,  
       mapping = aes(x = dep_delay, y = arr_delay)) +  
  geom_point()
```

Warning: Removed 5 rows containing missing values
(`geom_point()`).

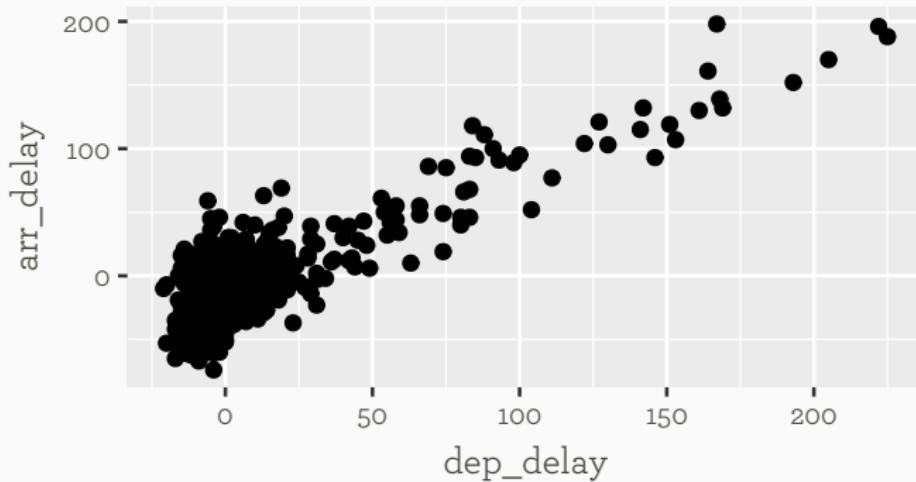


1. La grammaire des graphiques

Premier graphique

On peut se passer du nom des arguments `data` et `mapping` :

```
ggplot(alaska_flights, aes(x = dep_delay, y = arr_delay)) +  
  geom_point()
```



Visualisation :

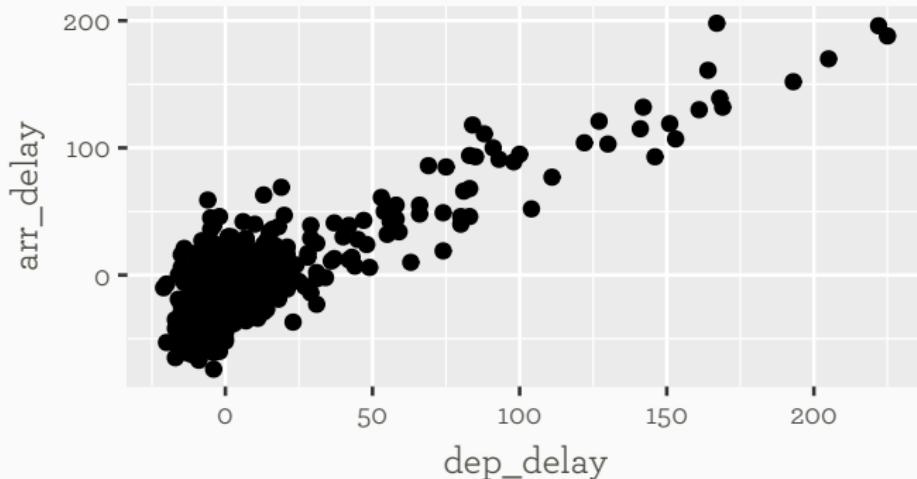
2. Nuages de points

2. Nuages de points

`geom_point()`

Les nuages de points sont créés grâce à la fonction `geom_point()` qui ajoute au graphique une couche contenant l'objet géométrique “points” :

```
ggplot(alaska_flights, aes(x = dep_delay, y = arr_delay)) +  
  geom_point()
```

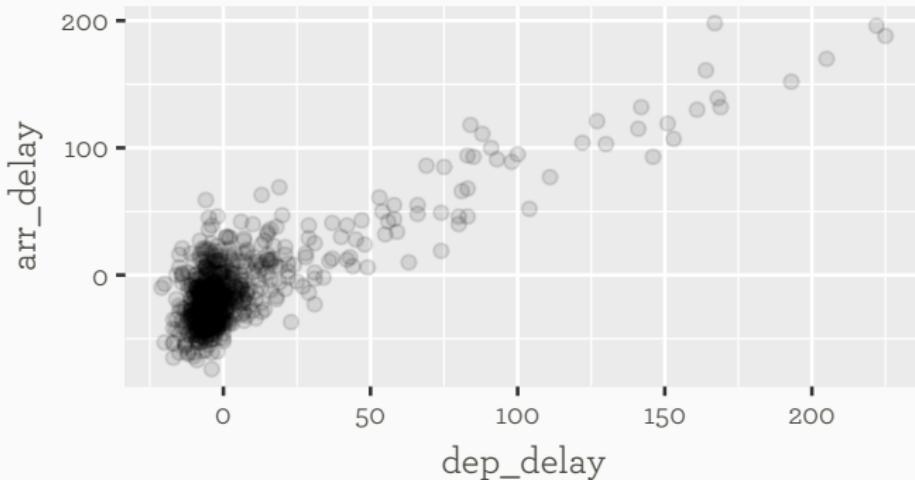


2. Nuages de points

Overplotting

Lorsque de nombreux points se superposent, on peut améliorer l'aspect du graphique en jouant sur la **transparence** des points :

```
ggplot(alaska_flights, aes(x = dep_delay, y = arr_delay)) +  
  geom_point(alpha = 0.1)
```

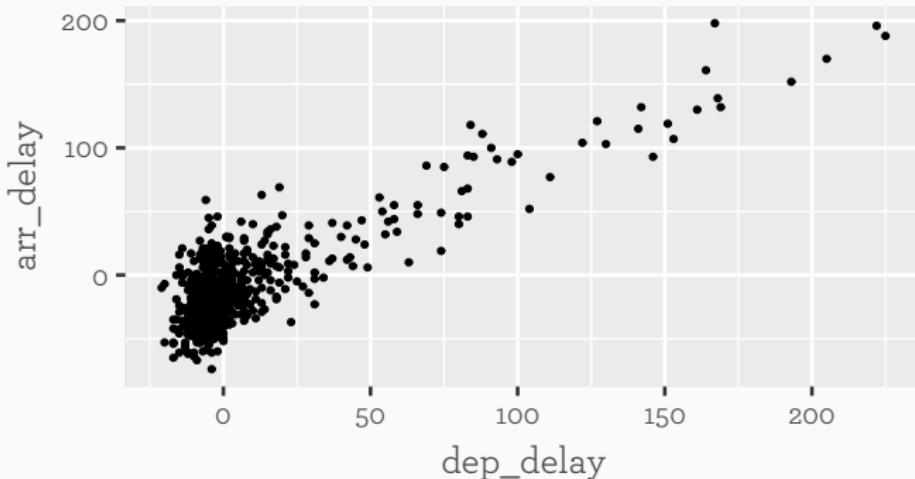


2. Nuages de points

Overplotting

Lorsque de nombreux points se superposent, on peut améliorer l'aspect du graphique en jouant sur la **taille** des points :

```
ggplot(alaska_flights, aes(x = dep_delay, y = arr_delay)) +  
  geom_point(size = 0.4)
```

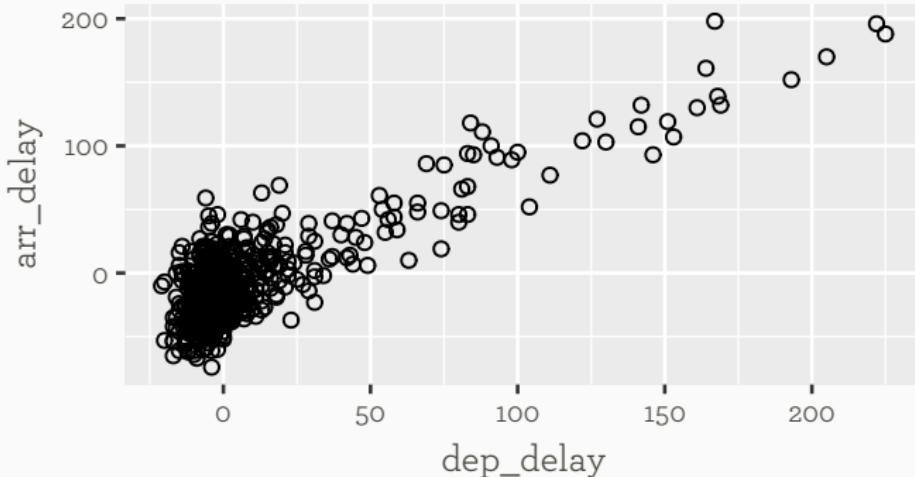


2. Nuages de points

Overplotting

Lorsque de nombreux points se superposent, on peut améliorer l'aspect du graphique en jouant sur le **symbole** utilisé :

```
ggplot(alaska_flights, aes(x = dep_delay, y = arr_delay)) +  
  geom_point(shape = 1)
```

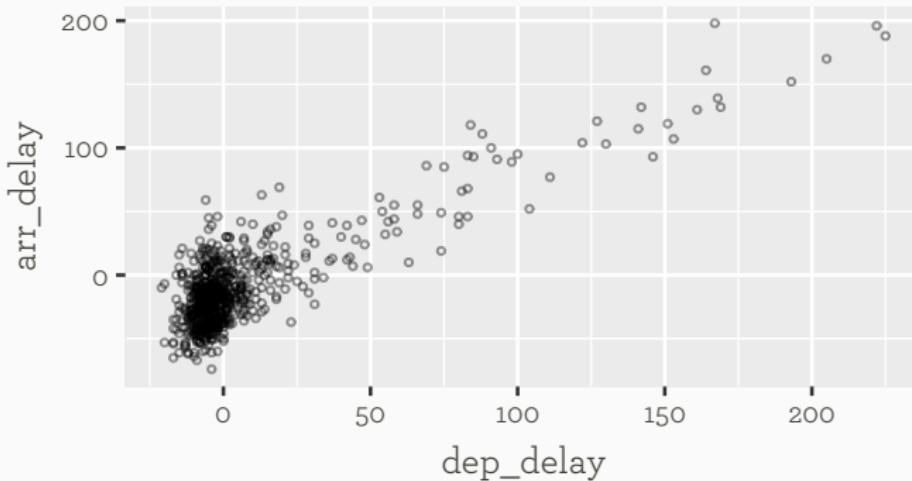


2. Nuages de points

Overplotting

Il est bien sûr possible de combiner plusieurs modifications :

```
ggplot(alaska_flights, aes(x = dep_delay, y = arr_delay)) +  
  geom_point(alpha = 0.4, size = 0.6, shape = 1)
```

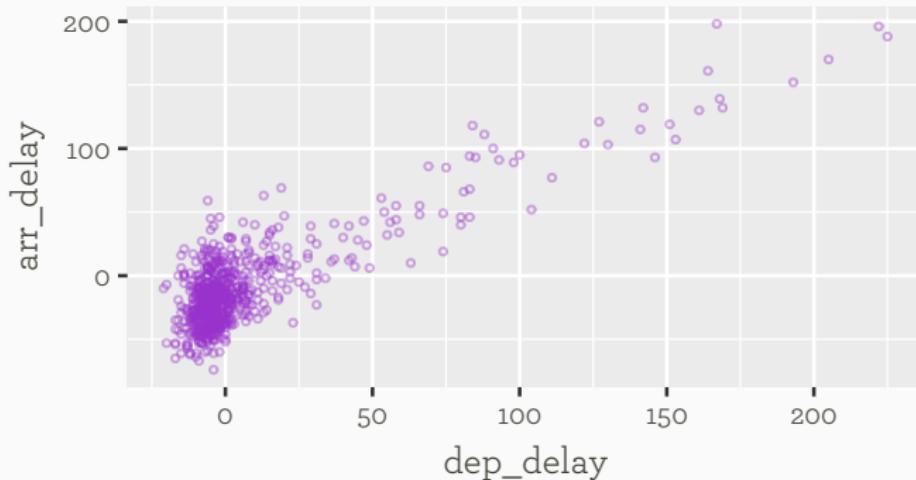


2. Nuages de points

Overplotting

Il est enfin possible de modifier la couleur des points :

```
ggplot(alaska_flights, aes(x = dep_delay, y = arr_delay)) +  
  geom_point(alpha = 0.4, size = 0.6,  
             shape = 1, color = "darkorchid")
```

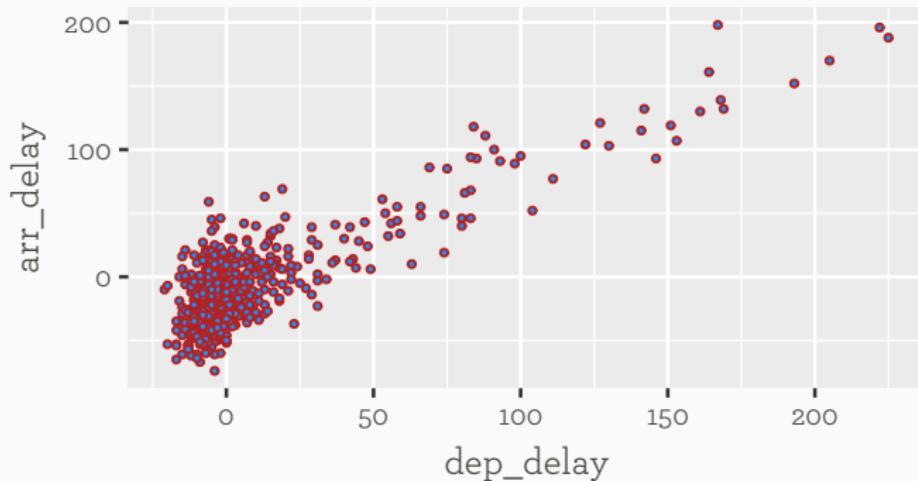


2. Nuages de points

Overplotting

Pour certains symboles, on fait la distinction entre couleur de **contour** et de **remplissage** :

```
ggplot(alaska_flights, aes(x = dep_delay, y = arr_delay)) +  
  geom_point(size = 0.6, shape = 21,  
             fill = "dodgerblue", color = "firebrick")
```

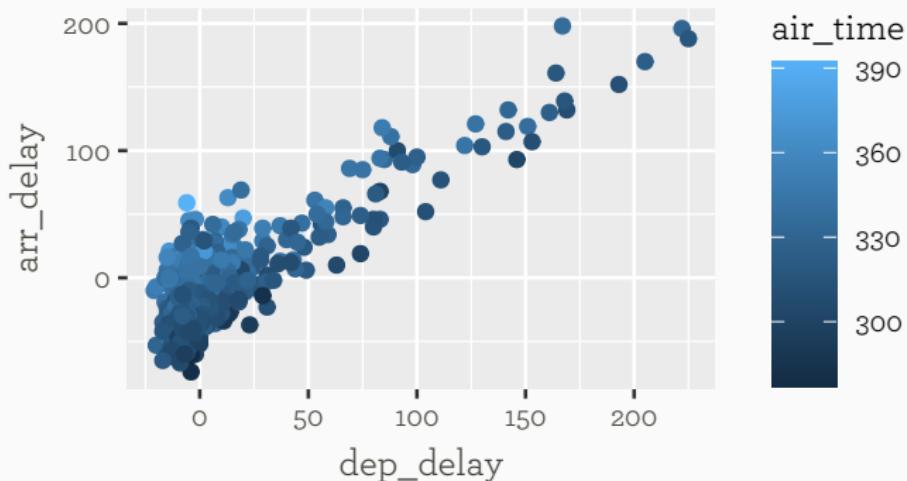


2. Nuages de points

Aesthetic mappings

Toutes ces options peuvent être associées à des variables :

```
ggplot(alaska_flights, aes(x = dep_delay, y = arr_delay,  
                           color = air_time)) +  
  geom_point()
```



2. Nuages de points

Aesthetic mappings

Est-ce que le jour de la semaine a une influence sur les retards des vols ?

```
library(lubridate)
alaska_small <- alaska_flights %>%
  mutate(jour = wday(time_hour,
                     label = TRUE,
                     abbr = TRUE,
                     week_start = 1)) %>%
  select(dep_delay, arr_delay, jour)
```

2. Nuages de points

Aesthetic mappings

Est-ce que le jour de la semaine a une influence sur les retards des vols ?

2. Nuages de points

Aesthetic mappings

Est-ce que le jour de la semaine a une influence sur les retards des vols ?

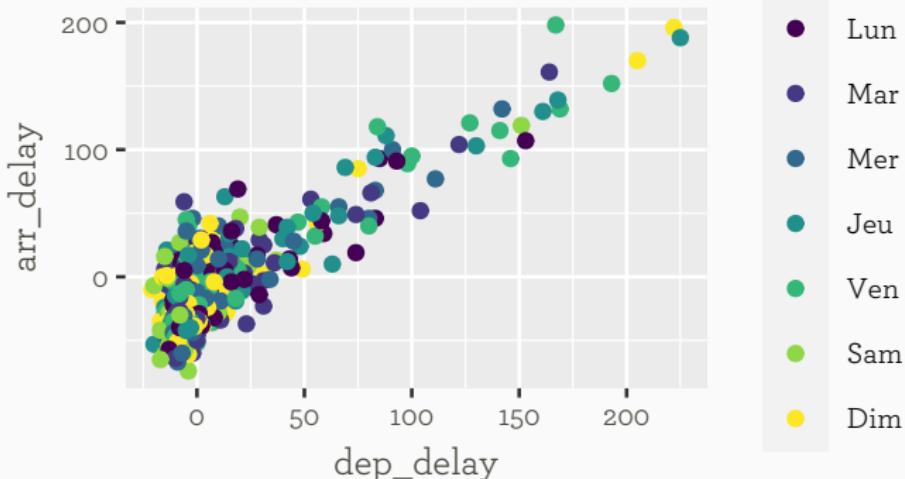
```
alaska_small
```

```
# A tibble: 714 x 3
  dep_delay arr_delay jour
  <dbl>      <dbl> <ord>
1       -1      -10 Mar
2       -7      -19 Mar
3       -3      -41 Mer
4        3       1 Mer
5       -1      -18 Jeu
6        2      -9 Jeu
7        0       1 Ven
8       -7      -29 Ven
9        0      -19 Sam
10      -12     -12 Sam
# i 704 more rows
```

2. Nuages de points

Aesthetic mappings

```
ggplot(alaska_small,  
       aes(x = dep_delay, y = arr_delay, color = jour)) +  
  geom_point()
```



2. Nuages de points

Aesthetic mappings

Les aesthetics possibles sont donc :

- ▶ x et y
- ▶ color (ou colour)
- ▶ fill
- ▶ shape
- ▶ size
- ▶ alpha

On peut placer ces arguments :

- ▶ **Dans** la fonction `aes()` pour associer une variable numérique ou catégorielle à ces caractéristiques esthétiques
- ▶ **En dehors** de la fonction `aes()` pour fixer manuellement la valeur voulue.

2. Nuages de points

Aesthetic mappings

Lorsque l'on fixe ces éléments manuellement, il faut fournir des valeur pertinentes :

color un nom de couleur, ou un numéro, ou un code hexadécimal, ou hsv, ou rgb, ou...

size un nombre de points en millimètres

alpha un degré d'opacité, compris entre 0 et 1

shape une valeur numérique comprise entre 0 et 24

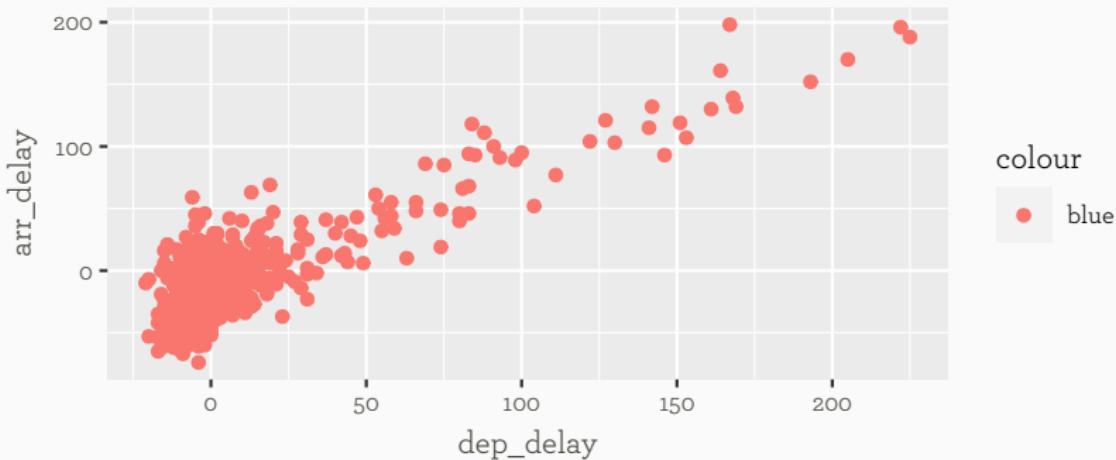
□ 0	× 4	⊕ 10	■ 15	■ 22
○ 1	▽ 6	★ 11	● 16	● 21
△ 2	⊗ 7	⊕ 12	▲ 17	▲ 24
◇ 5	* 8	⊗ 13	◆ 18	◆ 23
+ 3	◇ 9	□ 14	● 19	● 20

2. Nuages de points

Aesthetic mappings : exercices

Qu'est-ce qui ne va pas avec ce code? Pourquoi les points ne sont-ils pas bleus?

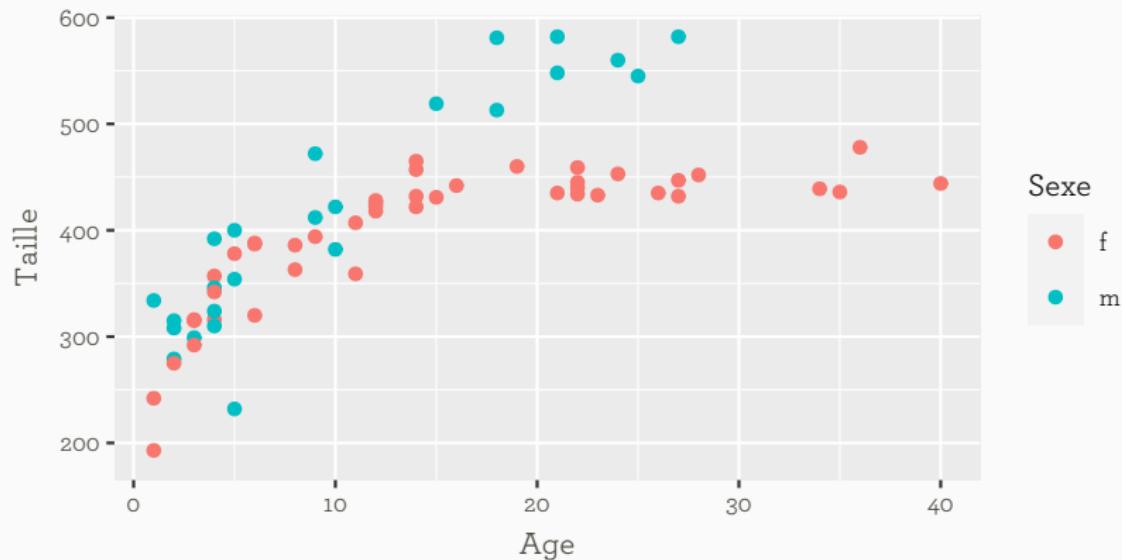
```
ggplot(alaska_flights,  
       aes(x = dep_delay, y = arr_delay, color = "blue")) +  
  geom_point()
```



2. Nuages de points

Aesthetic mappings : exercices

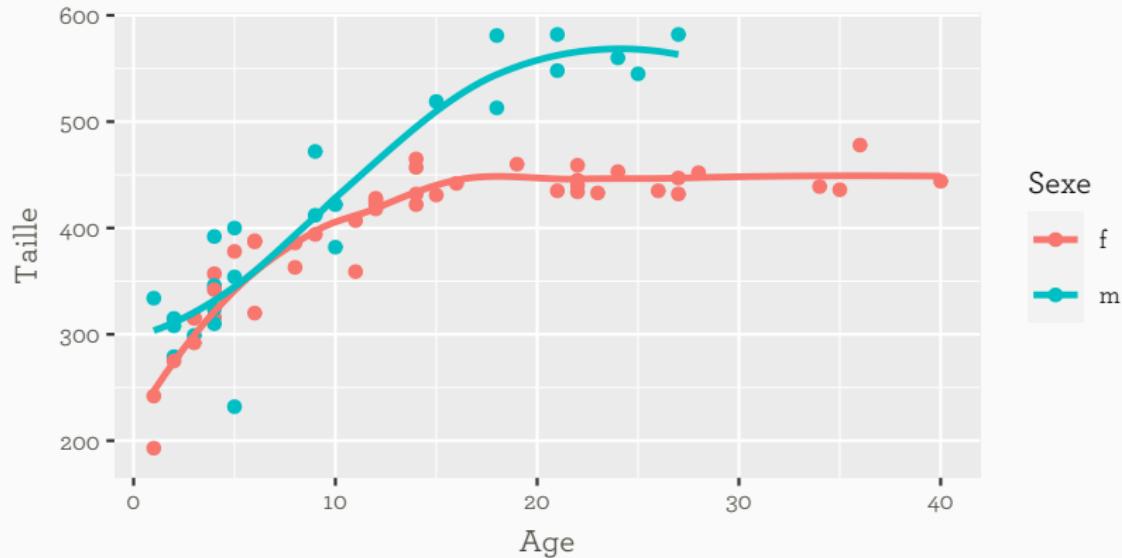
Avec les données contenues dans l'objet dauphin, faites le graphique suivant :



2. Nuages de points

Aesthetic mappings : exercices

En plus des points, ajoutez une couche supplémentaire à ce graphique en utilisant la fonction `geom_smooth()` :

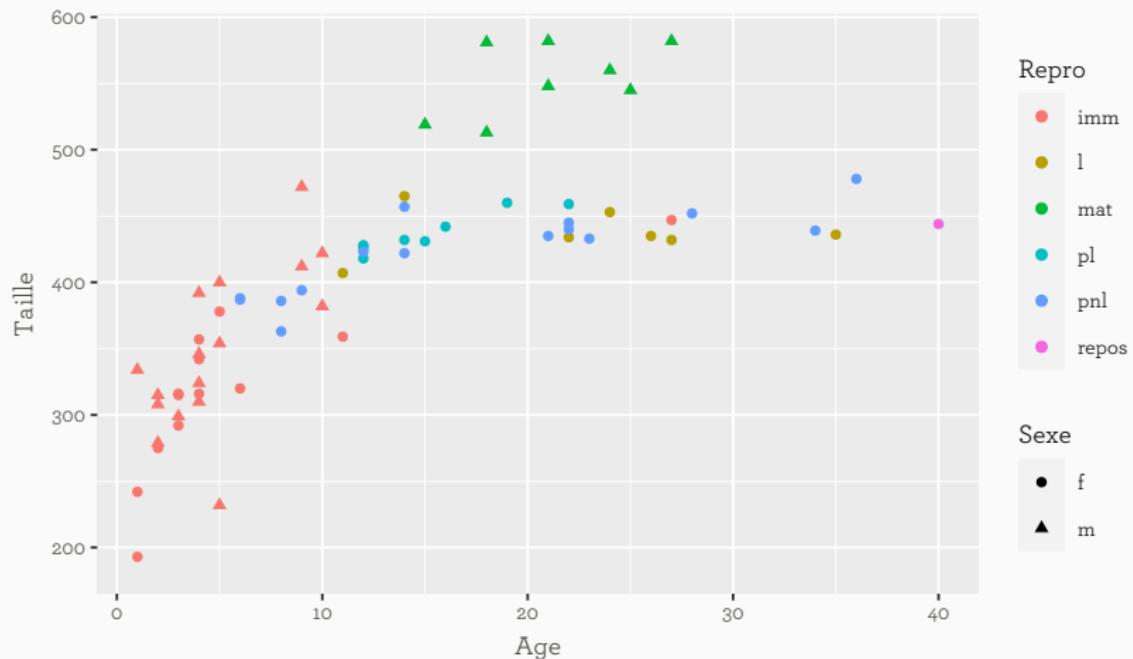


Que mettons-nous en évidence ici?

2. Nuages de points

Aesthetic mappings : exercices

Toujours avec dauphin, créez maintenant ce nouveau graphique :



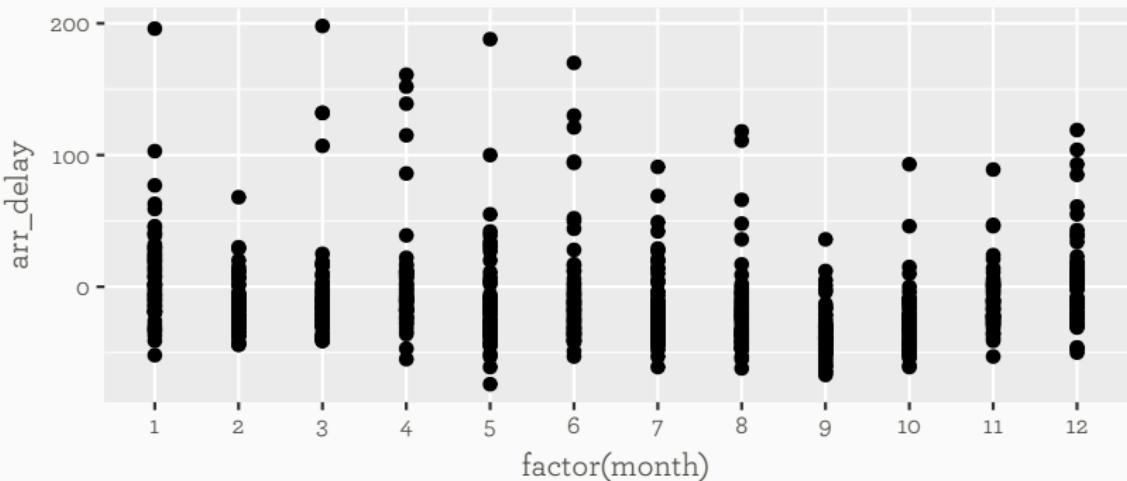
R nous dit qu'il a retiré 25 points. Pourquoi ?

2. Nuages de points

geom_jitter()

To jitter = trembler / être nerveux

```
ggplot(alaska_flights,  
       aes(x = factor(month), y = arr_delay)) +  
  geom_point()
```

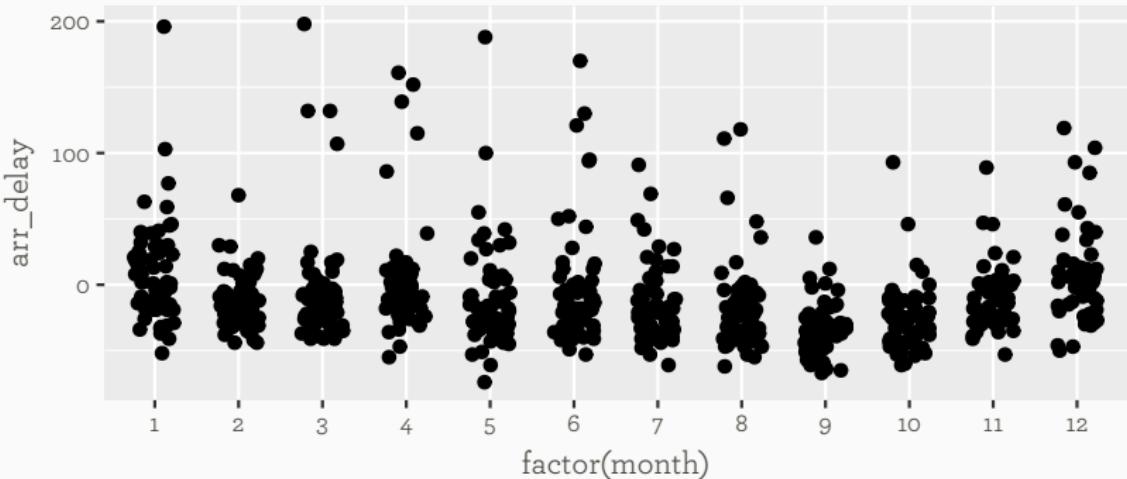


2. Nuages de points

geom_jitter()

To jitter = trembler / être nerveux

```
ggplot(alaska_flights,  
       aes(x = factor(month), y = arr_delay)) +  
  geom_jitter(height = 0, width = 0.25)
```



Visualisation :

3. Graphiques en lignes

3. Graphiques en lignes

Données météo

Nous allons maintenant travailler avec le jeu de données `weather` du package `nycflights13`.

- ▶ Examinez ce jeu de données
- ▶ Quelles sont les variables disponibles

Extraction des données météo de l'aéroport de Newark sur les 15 premiers jours de janvier :

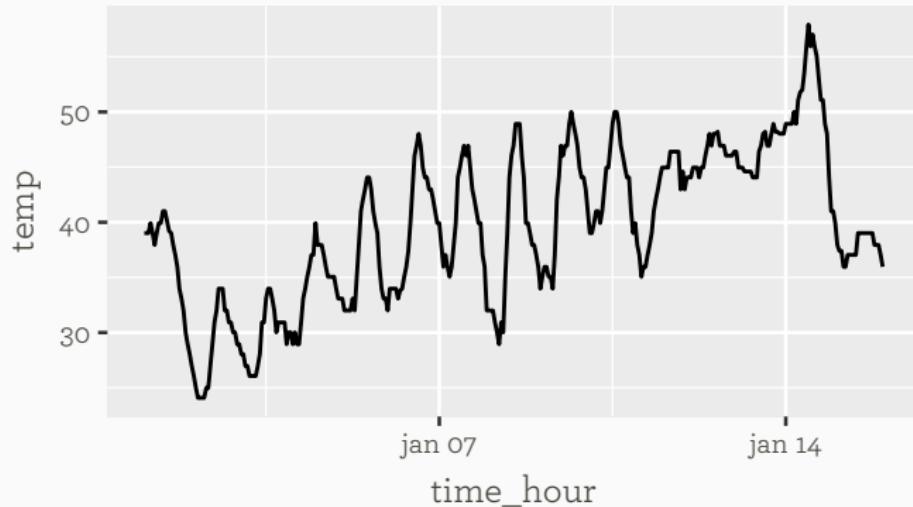
```
small_weather <- weather %>%
  filter(origin == "EWR",
        month == 1,
        day <= 15)
```

3. Graphiques en lignes

geom_line()

Températures horaires à l'aéroport de Newark :

```
ggplot(small_weather, aes(x = time_hour, y = temp)) +  
  geom_line()
```

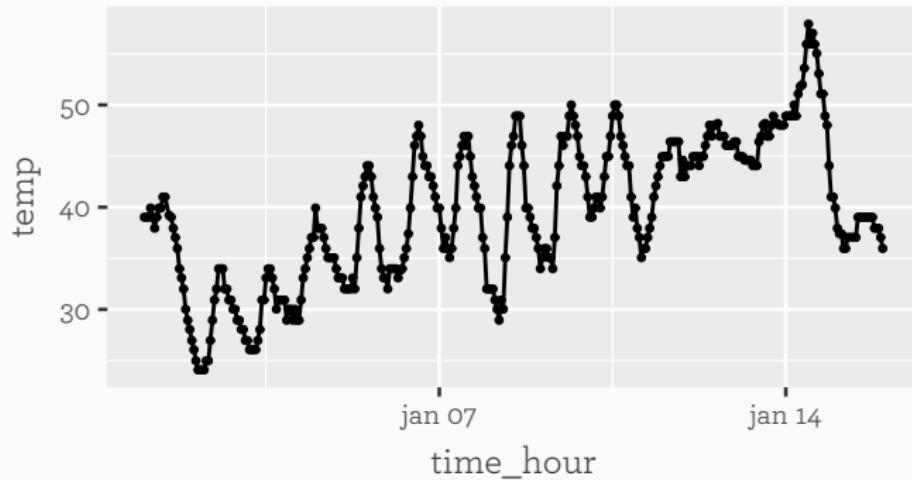


3. Graphiques en lignes

geom_line()

Il est tout à fait possible d'empiler les couches :

```
ggplot(small_weather, aes(x = time_hour, y = temp)) +  
  geom_line() +  
  geom_point(size = 0.5)
```

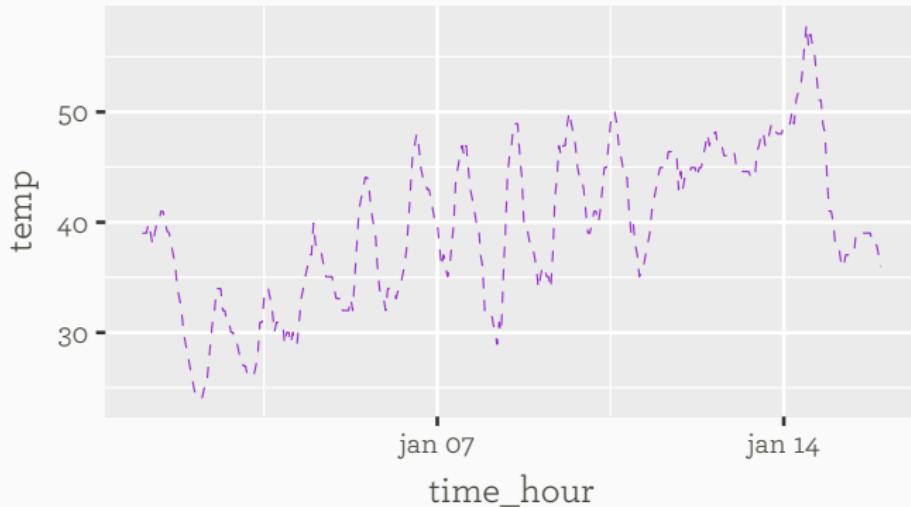


3. Graphiques en lignes

geom_line()

Comme pour geom_point(), nous pouvons modifier l'aspect de la ligne :

```
ggplot(small_weather, aes(x = time_hour, y = temp)) +  
  geom_line(size = 0.2, color = "darkorchid", linetype = 2)
```



3. Graphiques en lignes

geom_line()

Comme pour geom_point(), nous pouvons associer des variables aux caractéristiques esthétiques des lignes.

Pour l'illustrer, nous allons créer un nouveau jeu de données :

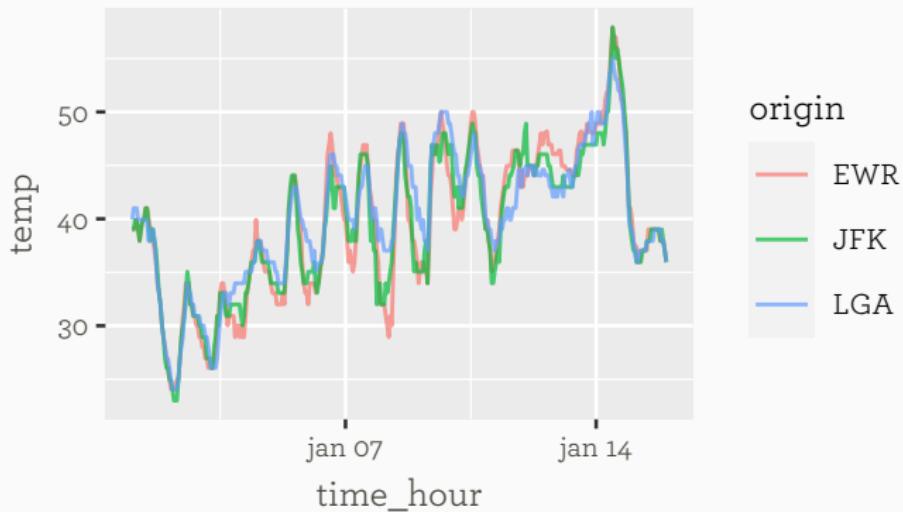
```
small_weather2 <- weather %>%
  filter(month == 1,
        day <= 15)
```

Selon vous, quelles est la différence entre small_weather et small_weather2?

3. Graphiques en lignes

geom_line()

```
ggplot(small_weather2,  
       aes(x = time_hour, y = temp, color = origin)) +  
  geom_line(alpha = 0.7)
```

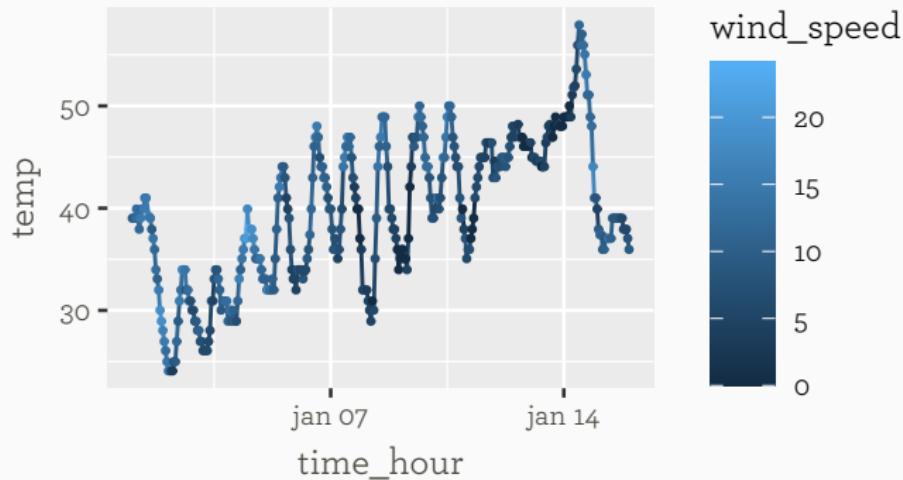


3. Graphiques en lignes

Où placer `aes()` ?

Comparer les 3 graphiques suivants et les commandes associées :

```
ggplot(small_weather,  
       aes(x = time_hour, y = temp, color = wind_speed)) +  
  geom_line() +  
  geom_point(size = 0.5)
```

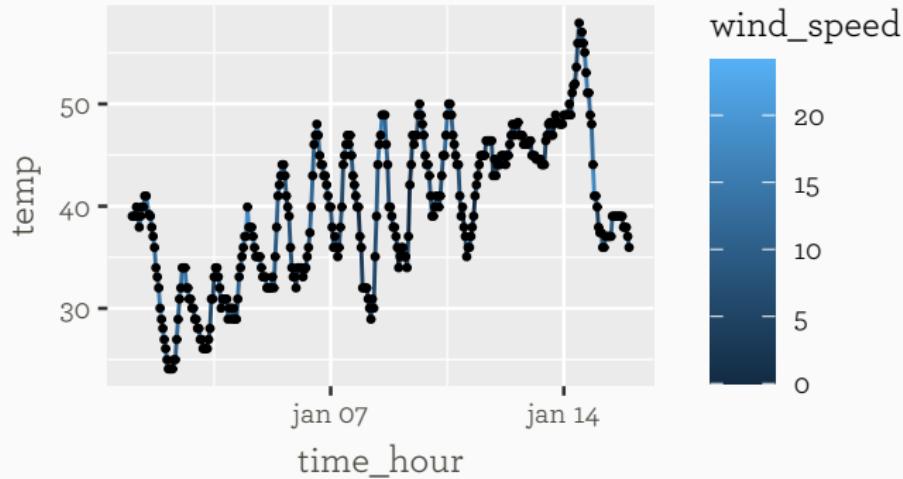


3. Graphiques en lignes

Où placer `aes()` ?

Comparer les 3 graphiques suivants et les commandes associées :

```
ggplot(small_weather,  
       aes(x = time_hour, y = temp)) +  
  geom_line(aes(color = wind_speed)) +  
  geom_point(size = 0.5)
```

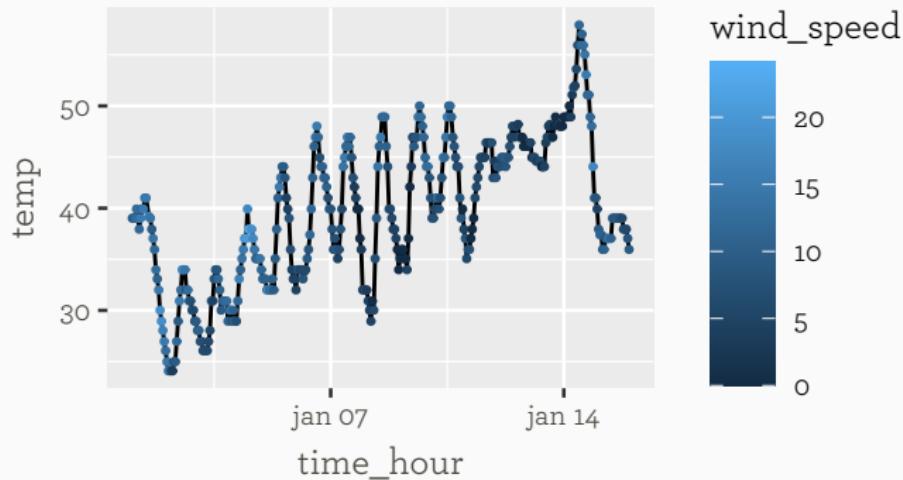


3. Graphiques en lignes

Où placer `aes()` ?

Comparer les 3 graphiques suivants et les commandes associées :

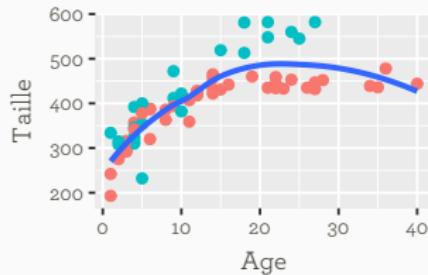
```
ggplot(small_weather,  
       aes(x = time_hour, y = temp)) +  
  geom_line() +  
  geom_point(aes(color = wind_speed), size = 0.5)
```



3. Graphiques en lignes

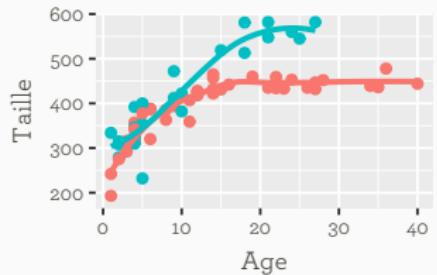
Où placer `aes()`? Exercice

Avec le jeu de données dauphin, créez les 4 graphiques ci-dessous⁸:



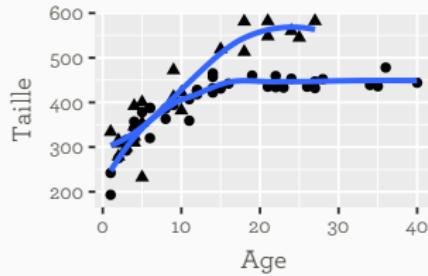
Sexe

- f
- m



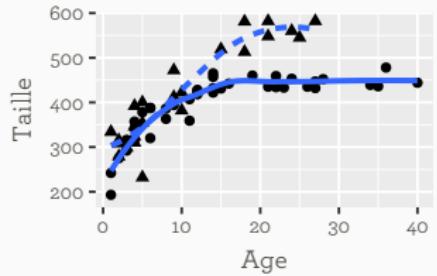
Sexe

- f
- m



Sexe

- f
- m



Sexe

- f
- m

8. Vous aurez besoin de la fonction `geom_smooth()` et de l'esthétique `group`.

Visualisation :

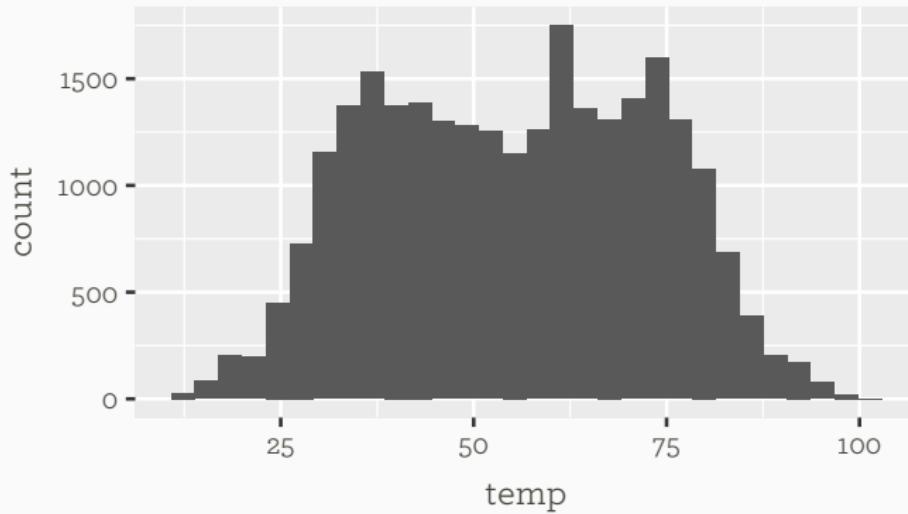
4. Les **histogrammes**

4. Les histogrammes

`geom_histogram()`

Distribution des températures horaires enregistrées en 2013 dans les 3 aéroports de New York :

```
ggplot(weather, aes(x = temp)) +  
  geom_histogram()
```

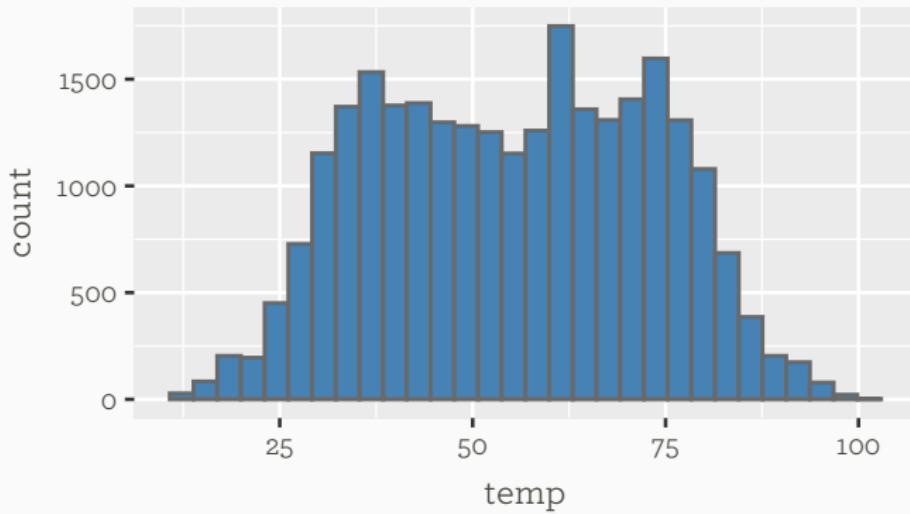


4. Les histogrammes

`geom_histogram()`

Comme d'habitude, on peut jouer sur un certain nombre de caractéristiques esthétiques :

```
ggplot(weather, aes(x = temp)) +  
  geom_histogram(fill = "steelblue", color = "dimgrey")
```



4. Les histogrammes

La taille des classes

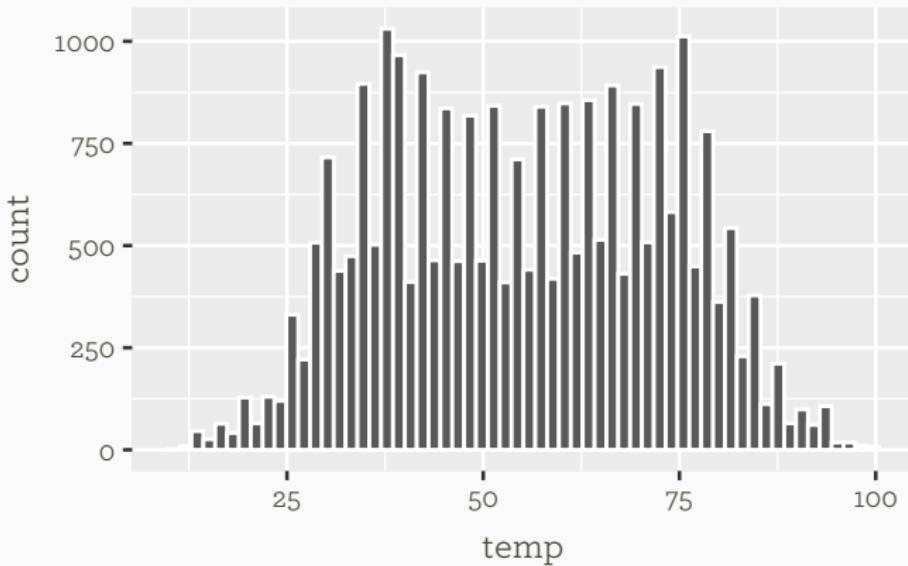
On peut spécifier manuellement la largeur des catégories de 3 façons différentes :

1. En ajustant le nombre de classes avec `bins`.
2. En précisant la largeur des classes avec `binwidth`.
3. En fournissant manuellement les limites des classes avec `breaks`.

4. Les histogrammes

La taille des classes

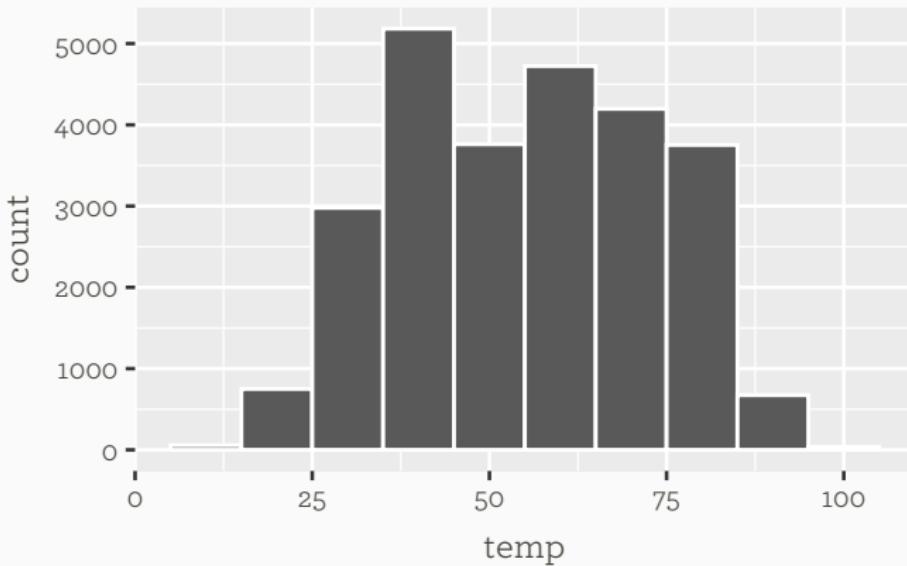
```
ggplot(weather, aes(x = temp)) +  
  geom_histogram(bins = 60, color = "white")
```



4. Les histogrammes

La taille des classes

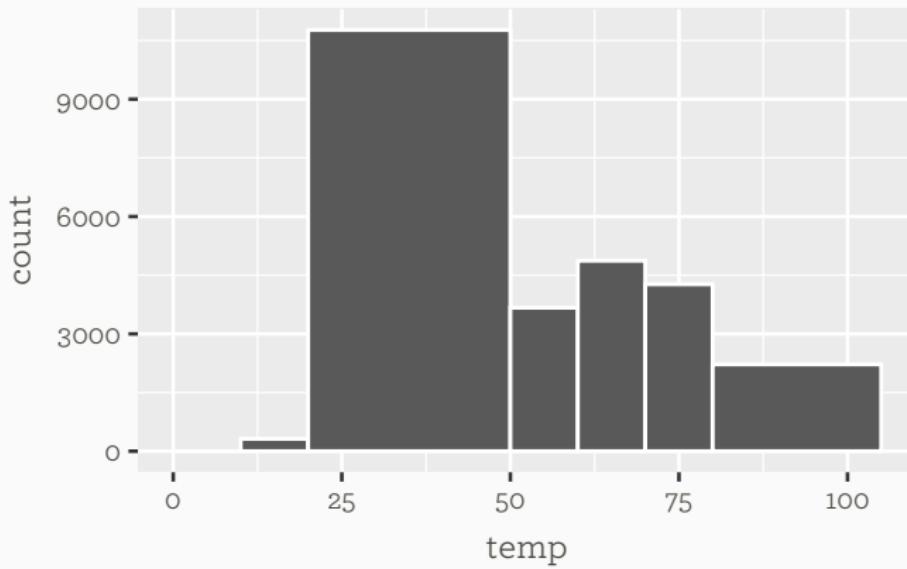
```
ggplot(weather, aes(x = temp)) +  
  geom_histogram(binwidth = 10, color = "white")
```



4. Les histogrammes

La taille des classes

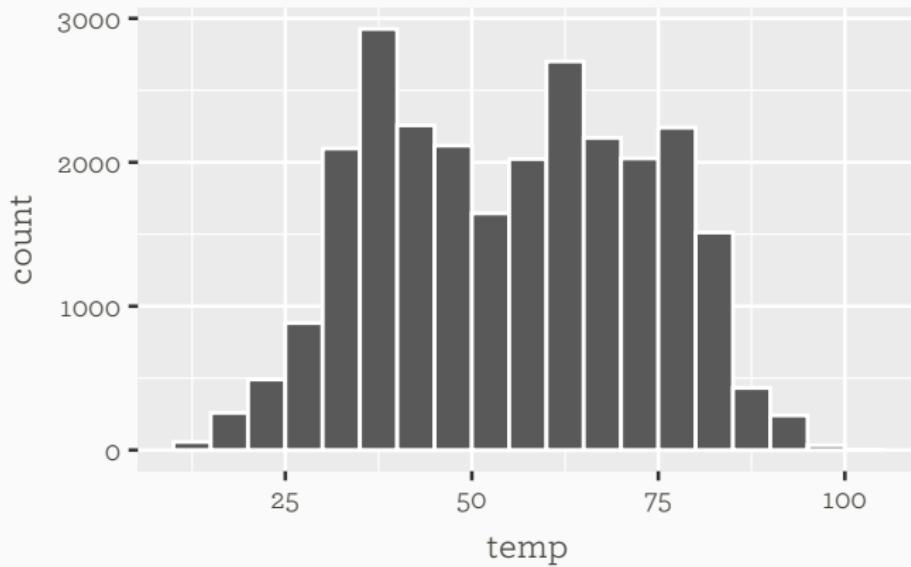
```
ggplot(weather, aes(x = temp)) +  
  geom_histogram(breaks = c(0, 10, 20, 50, 60, 70, 80, 105),  
                 color = "white")
```



4. Les histogrammes

La taille des classes

```
limits <- seq(from = 10, to = 105, by = 5)
ggplot(weather, aes(x = temp)) +
  geom_histogram(breaks = limits, color = "white")
```

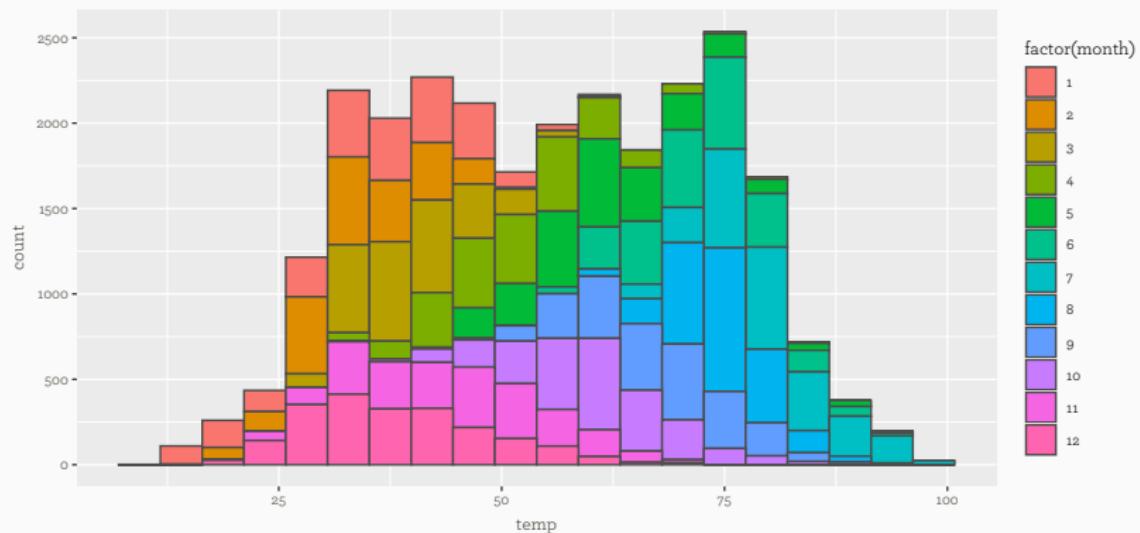


4. Les histogrammes

Variable supplémentaire

Comme pour les autres geom, il est possible d'associer une autre variable à une caractéristique esthétique de l'histogramme :

```
ggplot(weather, aes(x = temp, fill = factor(month))) +  
  geom_histogram(bins = 20, color = "grey30")
```



Visualisation :

5. Les facets

5. Les facets

Qu'est-ce que c'est ?

Une autre façon d'ajouter des variables supplémentaires est de séparer le graphiques en plusieurs facettes.

Definition

Un graphique composé de **facet**s est un graphique composé de plusieurs sous-graphiques, chacun d'entre eux présentant une partie des données.

Deux fonctions principales permettent de créer des facets :

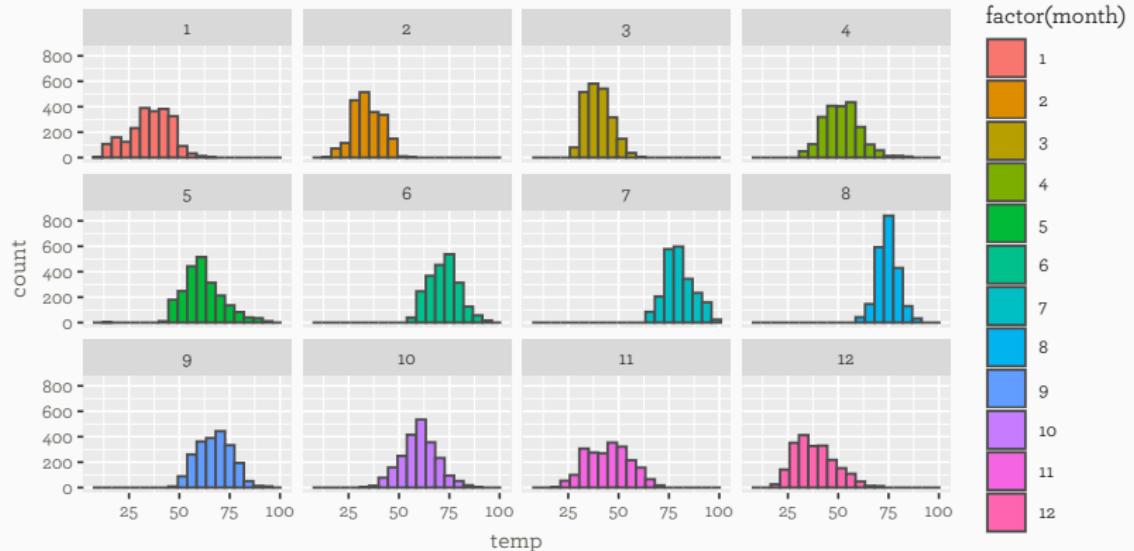
1. `facet_wrap()` pour séparer en fonction d'une seule variable
2. `facet_grid()` pour séparer en fonction de deux variables

Les variables utilisées pour “facetter” un graphique doivent être **catégorielles**.

5. Les facets

Exemple

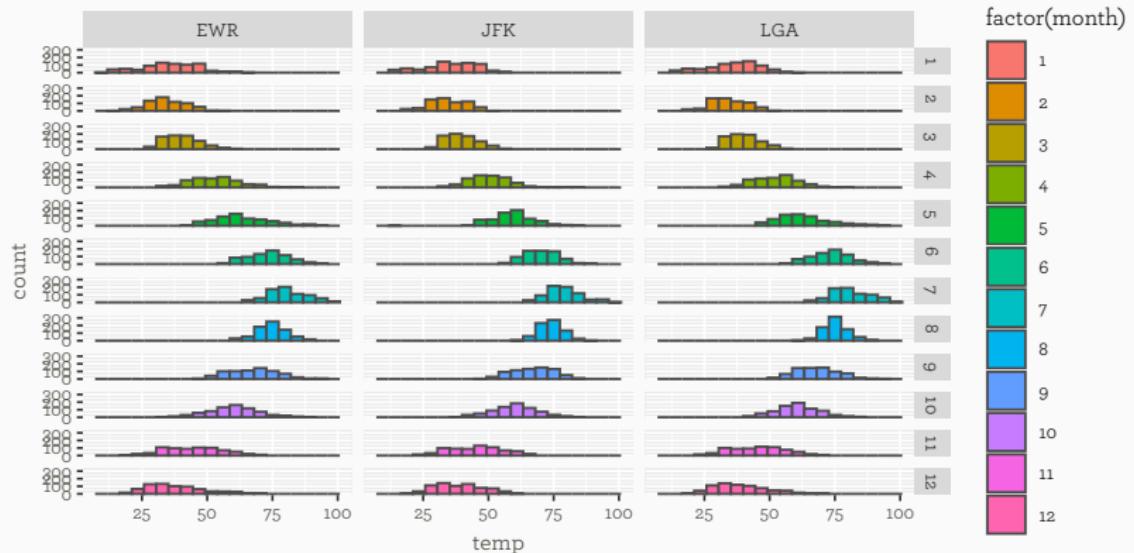
```
ggplot(weather, aes(x = temp, fill = factor(month))) +  
  geom_histogram(bins = 20, color = "grey30") +  
  facet_wrap(~factor(month), ncol = 4)
```



5. Les facets

Exemple

```
ggplot(weather, aes(x = temp, fill = factor(month))) +  
  geom_histogram(bins = 20, color = "grey30") +  
  facet_grid(factor(month) ~ origin)
```



Visualisation :

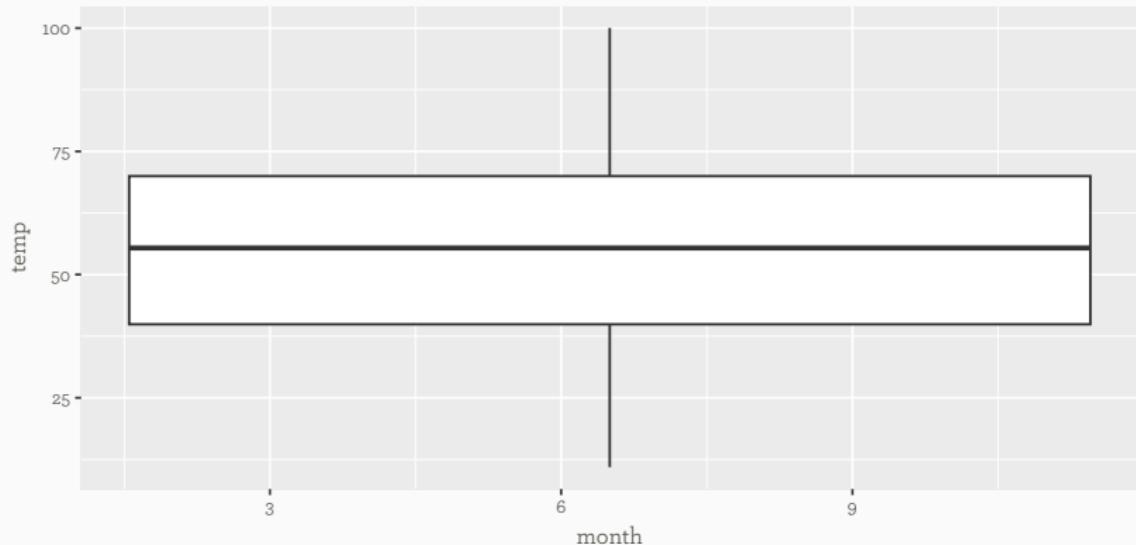
6. Les **boxplots**

6. Les boxplots

`geom_boxplot()`

Toujours avec weather, un boxplot fort peu utile...

```
ggplot(weather, aes(x = month, y = temp)) +  
  geom_boxplot()
```

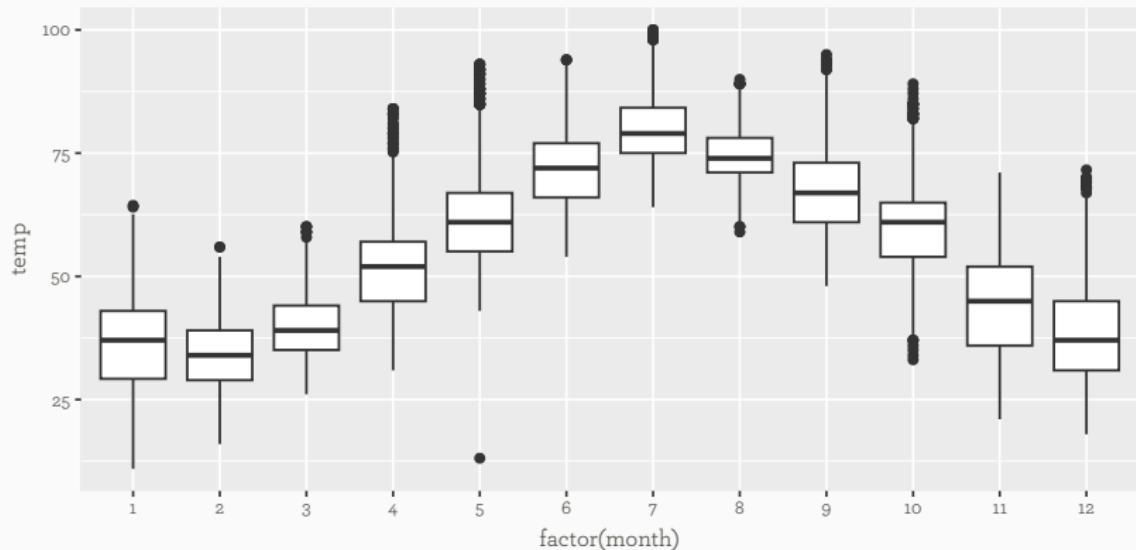


6. Les boxplots

`geom_boxplot()`

Les boxplots permettent de comparer la distribution d'une **variable numériques** pour plusieurs modalités d'une **variable catégorielle** :

```
ggplot(weather, aes(x = factor(month), y = temp)) +  
  geom_boxplot()
```

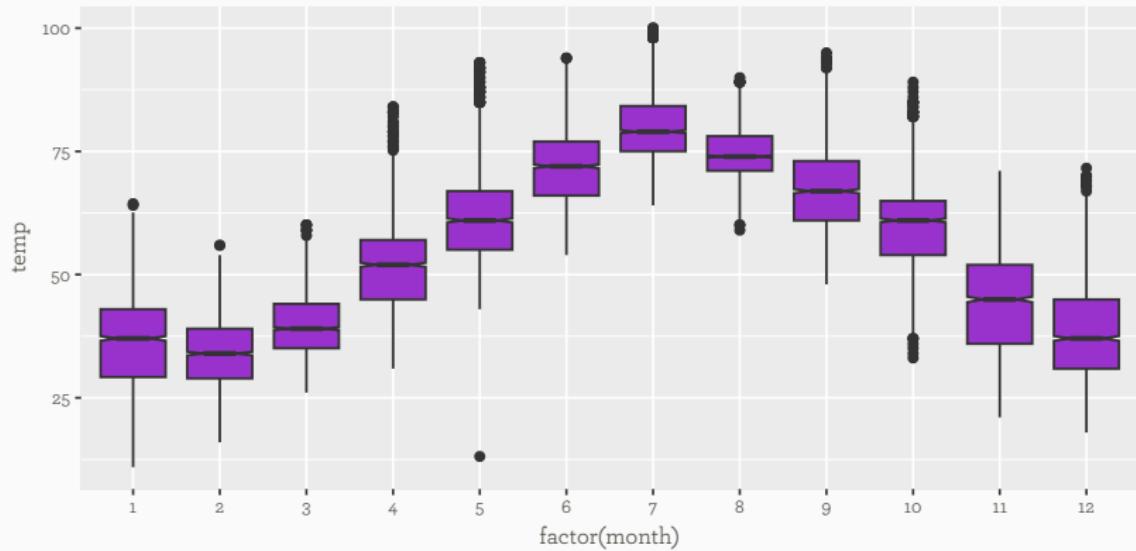


6. Les boxplots

`geom_boxplot()`

Ajout des intervalles de confiance à 95% des médianes :

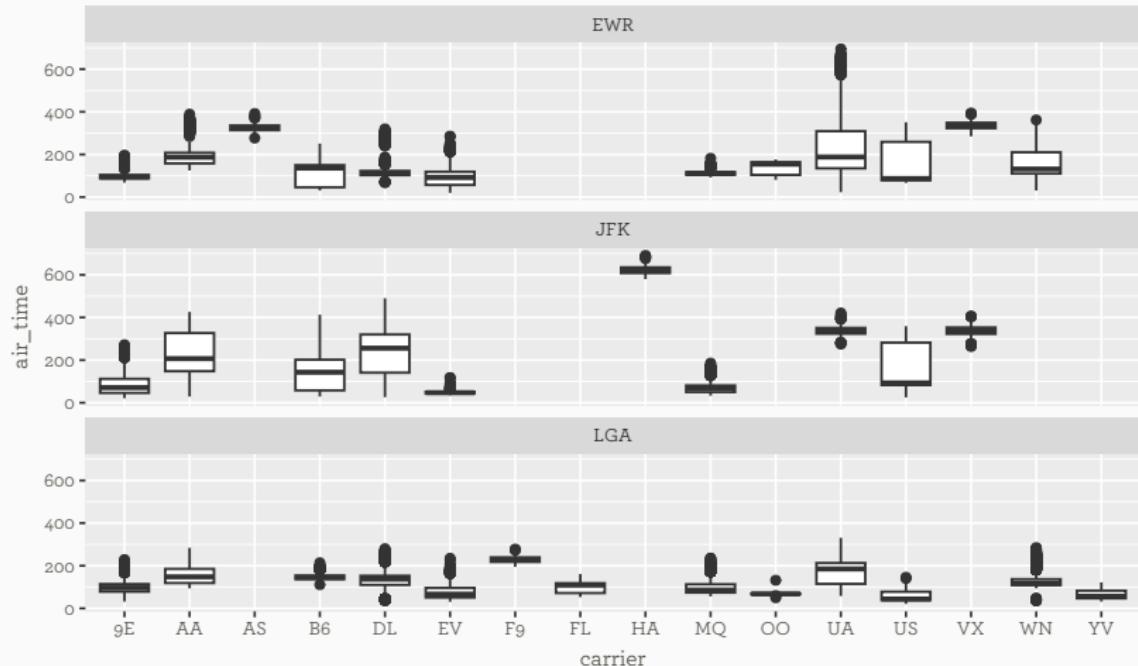
```
ggplot(weather, aes(x = factor(month), y = temp)) +  
  geom_boxplot(fill = "darkorchid", notch = TRUE)
```



6. Les boxplots

Exercice

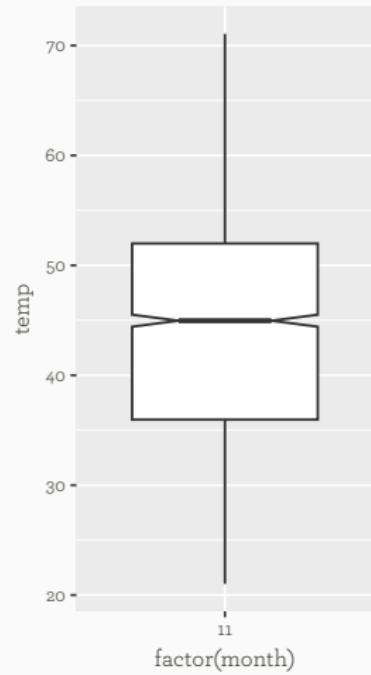
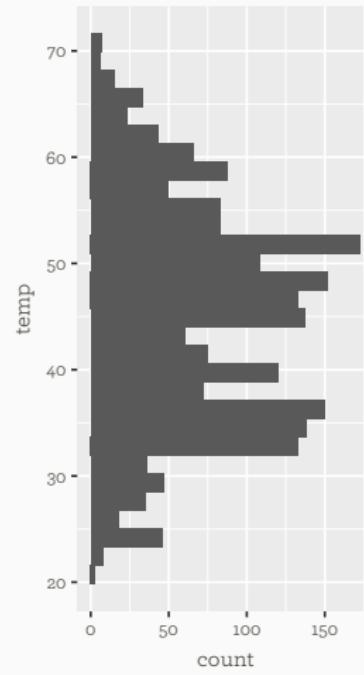
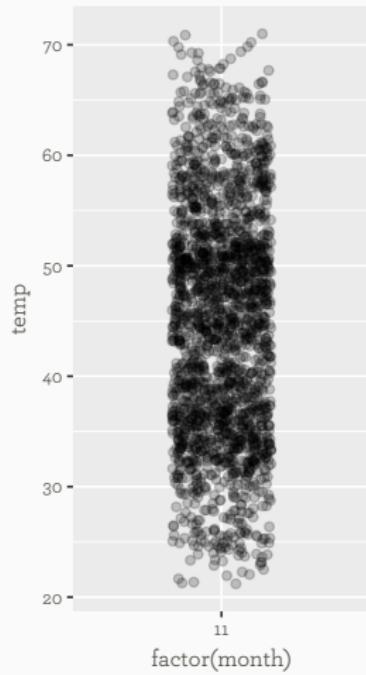
Avec le jeu de données flights, produisez le graphique suivant :



Que nous apprend-il?

6. Les boxplots

Une autre façon d'examiner des distributions



Visualisation :

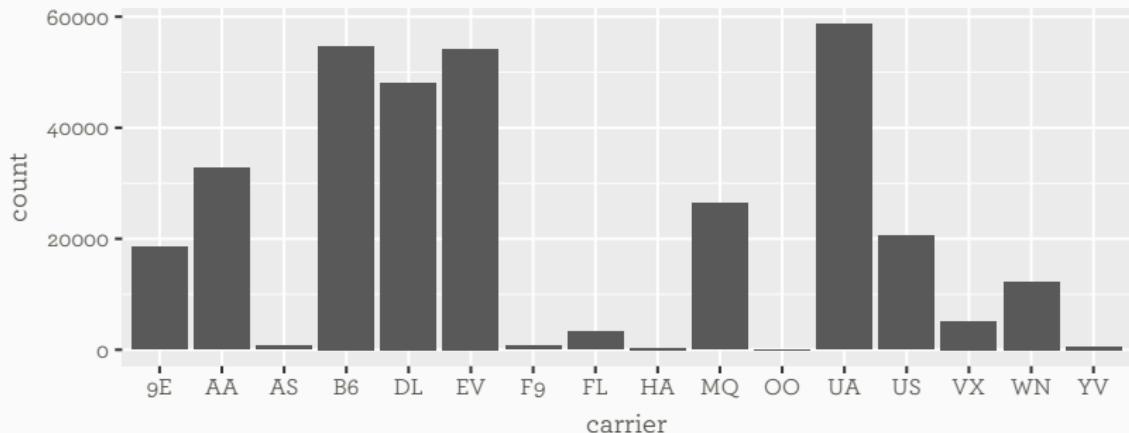
7. Les diagrammes bâtons

7. Les diagrammes bâtons

`geom_bar()`

Revenons au jeu de données `flights`. Combien de vols ont été affrétés en 2013 par chaque compagnie aérienne ?

```
ggplot(flights, aes(x = carrier)) +  
  geom_bar()
```



7. Les diagrammes bâtons

`geom_bar()`

Affichez le contenu de la variable `carrier`.

- ▶ Selon vous, comment le graphique précédent a-t'il été produit ?
- ▶ Quelle variable a été associée (“mappée”) à l’axe des ordonnées ?

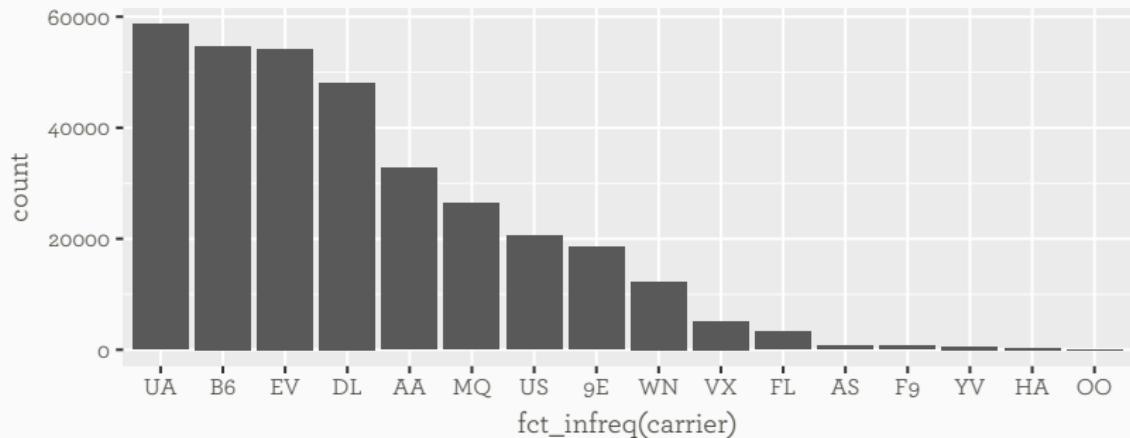
Si `carrier` est bien une variable de `flights`, `count` n’en est pas une.
D'où vient cette variable ?

7. Les diagrammes bâtons

`geom_bar()`

Les diagrammes bâtons sont souvent plus parlants quand les catégories sont triées :

```
ggplot(flights, aes(x = fct_infreq(carrier))) +  
  geom_bar()
```



7. Les diagrammes bâtons

geom_col()

Parfois, les données brutes ne sont pas disponibles, et nous disposons seulement de données déjà comptées :

```
# On prend flights, puis...
carrier_table <- flights %>%
  # On groupe les données par compagnie, puis...
  group_by(carrier) %>%
  # On calcule le nb de vols par Cie, puis ...
  summarize(nombre = n()) %>%
  # On trie le tableau par nb de vols décroissant
  arrange(desc(nombre))
```

7. Les diagrammes bâtons

geom_col()

Parfois, les données brutes ne sont pas disponibles, et nous disposons seulement de données déjà comptées :

```
# Enfin, on affiche la nouvelle table
carrier_table

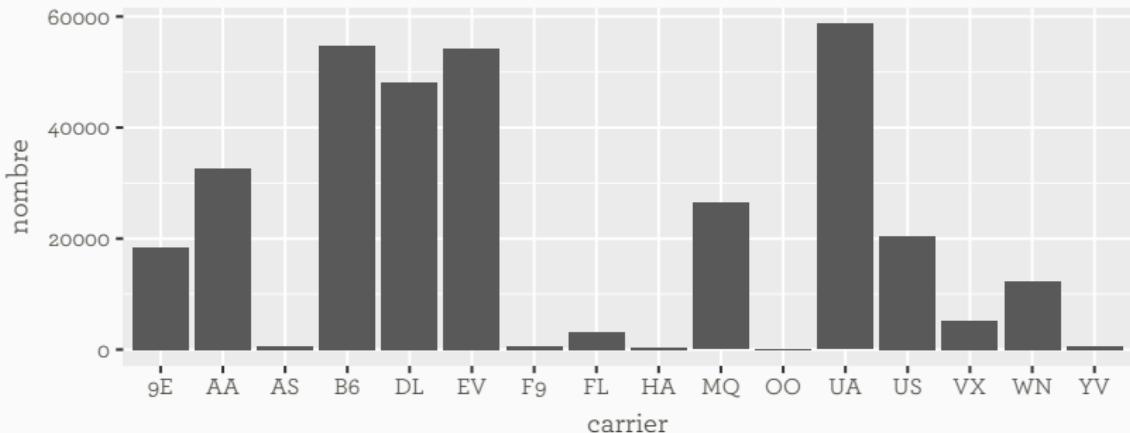
# A tibble: 16 x 2
  carrier   nombre
  <chr>     <int>
1 UA        58665
2 B6        54635
3 EV        54173
4 DL        48110
5 AA        32729
6 MQ        26397
7 US        20536
8 9E        18460
9 WN        12275
10 VX       5162
11 FL       3260
12 AS       714
13 F9       685
14 YV       601
15 HA       342
16 OO        32
```

7. Les diagrammes bâtons

`geom_col()`

Que faire si on ne dispose que de ces données “résumées”?

```
ggplot(carrier_table, aes(x = carrier, y = nombre)) +  
  geom_col()
```



Pourquoi les barres ne sont-elles pas ordonnées?

7. Les diagrammes bâtons

geom_col()

La table `carrier_table` est triée, mais pas les niveaux du facteur `carrier` :

```
factor(carrier_table$carrier)
[1] UA B6 EV DL AA MQ US 9E WN VX FL AS F9 YV HA OO
Levels: 9E AA AS B6 DL EV F9 FL HA MQ OO UA US VX WN YV
```

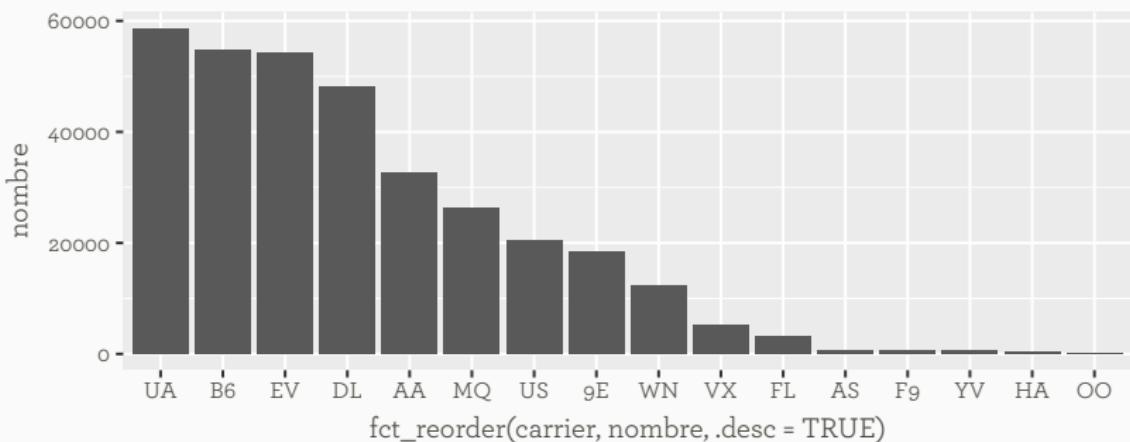
Les modalités sont toujours triées par ordre alphabétique.

Pour les modifier, il faut utiliser la fonction `fct_reorder()`.

7. Les diagrammes bâtons

geom_col()

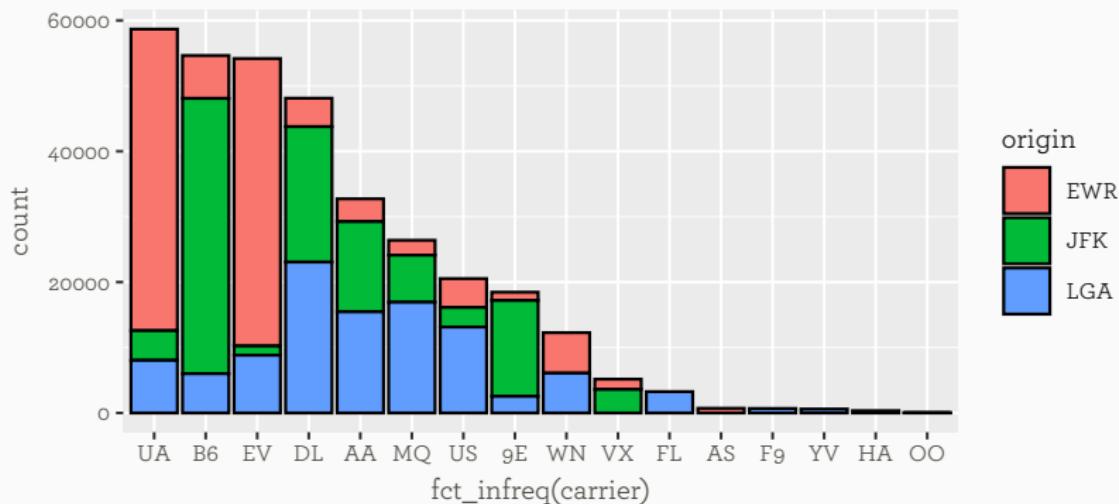
```
ggplot(carrier_table,  
       aes(x = fct_reorder(carrier, nombre, .desc = TRUE),  
            y = nombre)) +  
  geom_col()
```



7. Les diagrammes bâtons

Comparer 2 variables catégorielles

```
ggplot(flights, aes(x = fct_infreq(carrier), fill = origin)) +  
  geom_bar(color = "black")
```



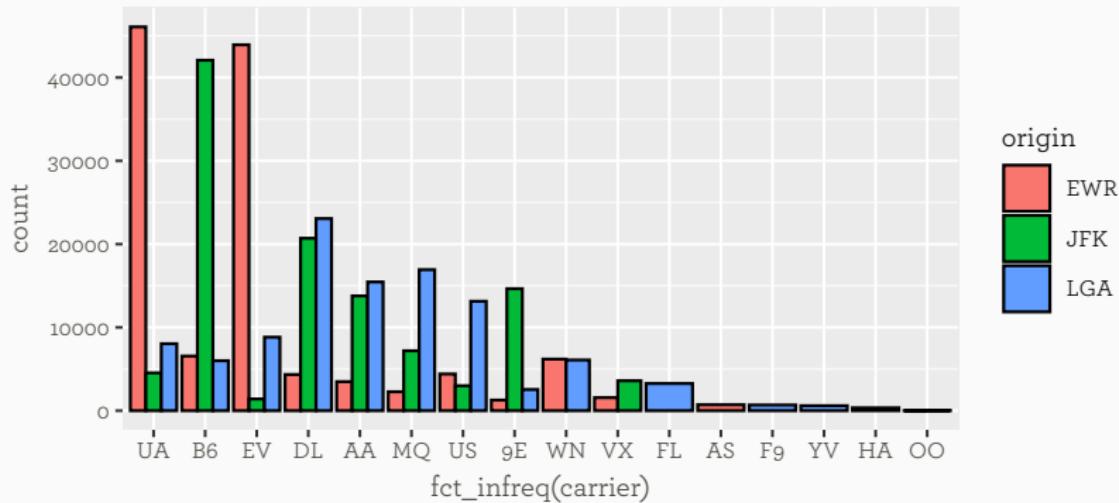
Il est rare que les barres empilées soient le choix le plus judicieux.

7. Les diagrammes bâtons

Comparer 2 variables catégorielles

Pour augmenter la lisibilité, on peut modifier la position des barres :

```
ggplot(flights, aes(x = fct_infreq(carrier), fill = origin)) +  
  geom_bar(color = "black", position = "dodge")
```

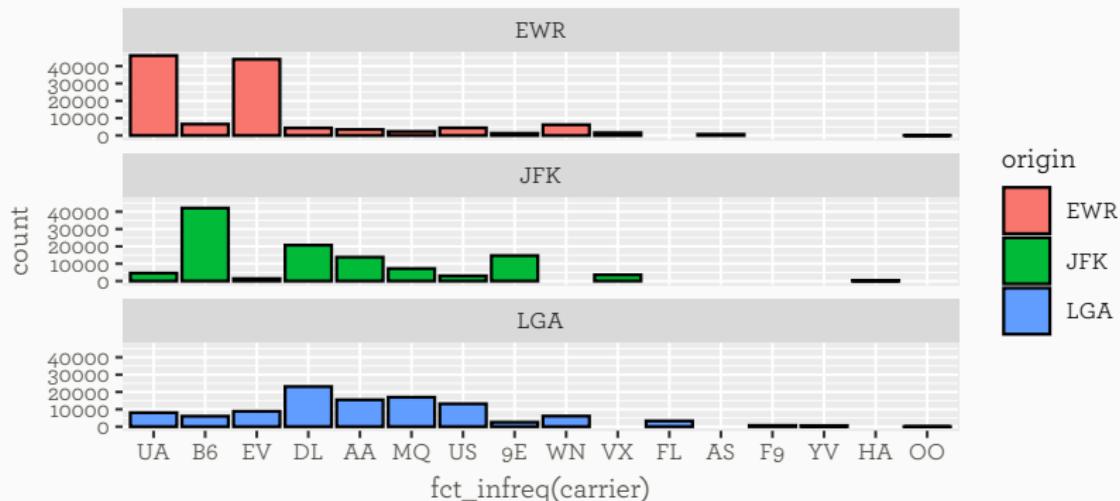


7. Les diagrammes bâtons

Comparer 2 variables catégorielles

Le plus simple reste souvent l'utilisation des facets :

```
ggplot(flights, aes(x = fct_infreq(carrier), fill = origin)) +  
  geom_bar(color = "black") +  
  facet_wrap(~origin, ncol = 1)
```



origin

- EWR
- JFK
- LGA

7. Les diagrammes bâtons

Comparer 2 variables catégorielles

Selon les types d'objets géométriques de vos graphiques, les choix possibles sont :

- ▶ “stacked”
- ▶ “identity”
- ▶ “dodge”
- ▶ “fill”
- ▶ “jitter”

Visualisation :

8. Systèmes de coordonnées

8. Système de coordonnées

Les systèmes de coordonnées sont probablement l'une des parties les plus compliquées de ggplot2.

Par défaut, le système de **coordonnées cartésiennes** est utilisé. Les axes x et y agissent indépendamment pour déterminer la position de chaque point.

D'autres systèmes existent. Examinons les plus utiles :

- ▶ `coord_cartesian()`
- ▶ `coord_flip()`
- ▶ `coord_map()`, `coord_quickmap()`
- ▶ `coord_polar()`

8. Système de coordonnées

coord_flip()

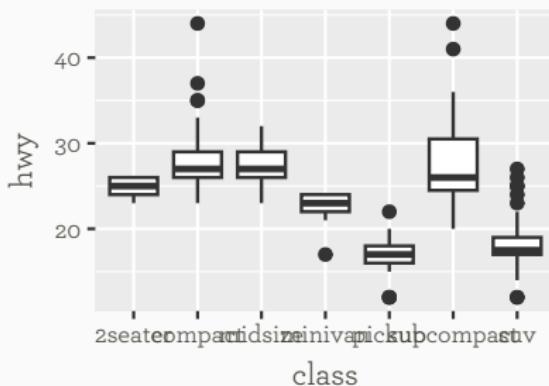
coord_flip() inverse la position des axes x et y :

```
# Gauche
```

```
ggplot(data = mpg, mapping = aes(x = class, y = hwy)) + geom_boxplot()
```

```
# Droite
```

```
ggplot(data = mpg, mapping = aes(x = class, y = hwy)) + geom_boxplot() +  
  coord_flip()
```



8. Système de coordonnées

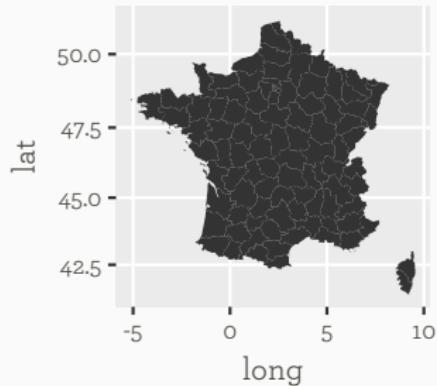
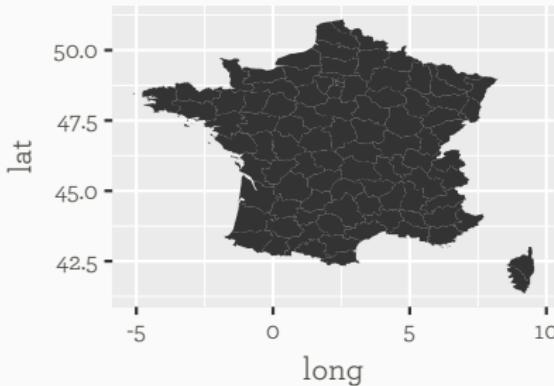
`coord_map()`, `coord_quickmap()`

`coord_map()` ou `coord_quickmap()` sont utiles pour... faire des cartes !

```
fr <- map_data("france")
```

```
ggplot(fr, aes(long, lat, group = group)) + geom_polygon()
```

```
ggplot(fr, aes(long, lat, group = group)) + geom_polygon() + coord_map()
```



8. Système de coordonnées

coord_polar()

coord_polar() permet d'utiliser des coordonnées polaires. Il existe un lien entre **barplot** et **graphique de Coxcomb**.

Tapez les commandes suivantes pour visualiser ce lien :

```
bar <- ggplot(data = diamonds) +
  geom_bar(
    mapping = aes(x = cut, fill = cut),
    show.legend = FALSE,
    width = 1
  ) +
  theme(aspect.ratio = 1) +
  labs(x = NULL, y = NULL)
```

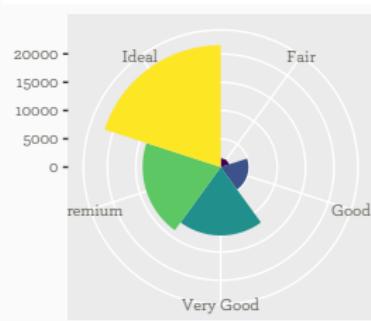
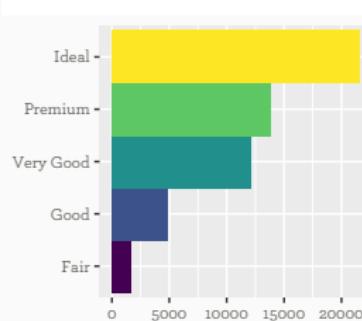
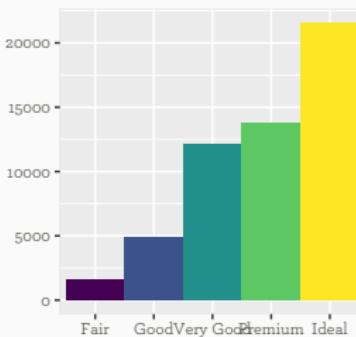
8. Système de coordonnées

coord_polar()

coord_polar() permet d'utiliser des coordonnées polaires. Il existe un lien entre **barplot** et **graphique de Coxcomb**.

Tapez les commandes suivantes pour visualiser ce lien :

```
bar                      # Gauche  
bar + coord_flip()      # Milieu  
bar + coord_polar()     # Droite
```



8. Système de coordonnées

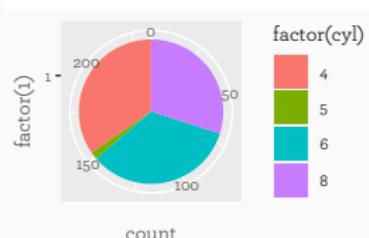
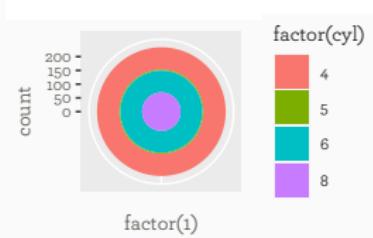
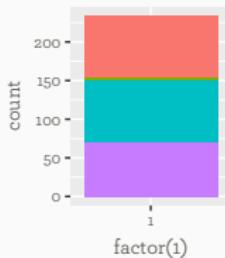
coord_polar()

coord_polar() permet d'utiliser des coordonnées polaires. Il existe aussi un lien entre barplot empilé et diagramme camembert.

Tapez les commandes suivantes pour visualiser ce lien :

```
pie <- ggplot(mpg, aes(x = factor(1), fill = factor(cyl))) +  
  geom_bar(width=1)
```

```
pie  
# Gauche  
pie + coord_polar()  
# Milieu  
pie + coord_polar(theta = "y")  
# Droite
```

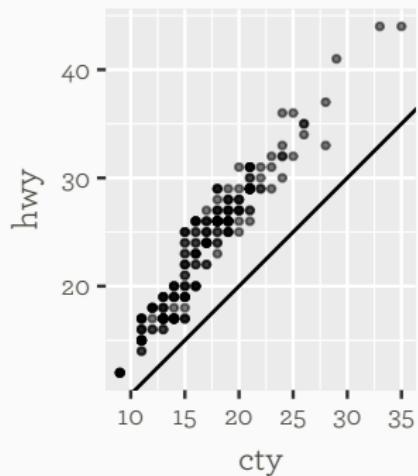


8. Système de coordonnées

Exercices

- ▶ Que nous apprend le graphique ci-dessous au sujet de la relation entre consommation en ville (cty) et consommation sur autoroute (hwy)⁹ ?
- ▶ Pourquoi est-il important d'utiliser `coord_fixed()` ?
- ▶ Que fait `geom_abline()` ?

```
ggplot(data = mpg,  
       aes(x = cty, y = hwy)) +  
  geom_point(size = 0.7,  
             alpha = 0.5) +  
  geom_abline() +  
  coord_fixed()
```

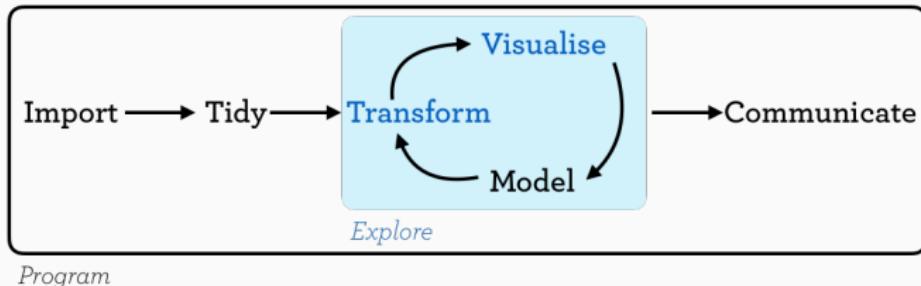


-
9. Consultez l'aide de `mpg` pour savoir de quoi nous parlons.

Visualisation :

9. Des graphiques présentables...

9. Des graphiques présentables



Jusqu'à maintenant, nous avons fait des graphiques pour **explorer** des jeux de données.

Il est tout aussi important d'être capable de réaliser des graphiques pour **communiquer** des résultats.

Pour cela, nous sommes souvent amenés à modifier l'**apparence** de nos graphiques. Nous aurons besoin des packages `ggrepel` et `viridis`. Installez-les dès maintenant.

9. Des graphiques présentables

Pour mettre en forme un graphique afin de communiquer clairement des résultats, il faut maîtriser au minimum les 3 éléments suivants :

1. Les labels avec la fonction `labs()`.
2. Les échelles avec notamment les fonctions `scale_XXX_YYY()`.
3. Les thèmes avec les fonctions `theme_XX()`.

Nous allons voir ensemble comment ça marche en reprenant des exemples de graphiques créés précédemment.

Manipuler des données

Manipuler des données

Préambule

Dans cette partie, nous allons découvrir comment **transformer** nos données à l'aide du package dplyr.

Comme dans la partie consacrée aux visualisations graphiques, nous utiliserons le jeu de données **flights** listant les caractéristiques de tous les vols au départ de New York City en 2013.

Nous découvrirons également ce qu'est le “**pipe**”.

Pour tout cela, nous aurons besoin d'installer 2 packages :

```
## install.packages("tidyverse")
## install.packages("nycflights13")
library(tidyverse)
library(nycflights13)
```

Manipuler des données

nycflights13

Jetons à nouveau un œil aux données contenues dans flights :

```
flights
```

```
# A tibble: 336,776 x 19
  year month   day dep_time sched_dep_time dep_delay arr_time
  <int> <int> <int>     <int>          <int>    <dbl>     <int>
1 2013     1     1      517            515       2     830
2 2013     1     1      533            529       4     850
3 2013     1     1      542            540       2     923
4 2013     1     1      544            545      -1    1004
5 2013     1     1      554            600      -6     812
6 2013     1     1      554            558      -4     740
7 2013     1     1      555            600      -5     913
8 2013     1     1      557            600      -3     709
9 2013     1     1      557            600      -3     838
10 2013    1     1      558            600      -2     753
# i 336,766 more rows
# i 12 more variables: sched_arr_time <int>, arr_delay <dbl>,
#   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>,
#   dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
#   minute <dbl>, time_hour <dttm>
```

Manipuler des données

dplyr : les bases

Manipuler des données avec dplyr revient en général à effectuer les 5 actions suivantes, qui correspondent chacune à une fonction portant le nom d'un verbe :

1. Choisir des observations en fonction de leur valeur : `filter()`
2. Réordonner les lignes : `arrange()`
3. Choisir des variables par leur nom : `select()`
4. Créer de nouvelles variables en combinant des variables existantes : `mutate()`
5. Regrouper de nombreuses valeurs sous la forme d'un résumé unique : `summarise()`

Toutes ces fonctions peuvent être utilisées en conjonction avec `group_by()`, qui permet de ne plus travailler sur l'ensemble du jeu de données, mais sur des sous-groupes spécifiques.

Manipuler des données

dplyr : les bases

Ces 6 fonctions constituent les **verbes** de la langue utilisée pour la manipulation de données

Ils travaillent tous de la même façons :

- ▶ Le premier argument est un `data.frame` (ou un `tibble`)
- ▶ Les arguments suivants décrivent ce qui doit être fait avec les données, en utilisant les noms de variables (sans les guillemets)
- ▶ Le résultat est un nouveau `data.frame` (ou un nouveau `tibble`)

Prises globalement, ces propriétés rendent aisé l'enchaînement de multiples **opérations simples** permettant d'obtenir des **résultats complexes**.

Manipuler des données :

1. Filtrer

1. Filtrer des données

La fonction filter()

filter() permet de récupérer des observations grâce à leur valeur.

Par exemple, pour sélectionner tous les vols du **premier janvier** :

```
filter(flights, month == 1, day == 1)

# A tibble: 842 x 19
  year month   day dep_time sched_dep_time dep_delay arr_time
  <int> <int> <int>     <int>           <int>     <dbl>    <int>
1 2013     1     1      517            515        2     830
2 2013     1     1      533            529        4     850
3 2013     1     1      542            540        2     923
4 2013     1     1      544            545       -1    1004
5 2013     1     1      554            600       -6     812
6 2013     1     1      554            558       -4     740
7 2013     1     1      555            600       -5     913
8 2013     1     1      557            600       -3     709
9 2013     1     1      557            600       -3     838
10 2013    1     1      558            600       -2     753
# i 832 more rows
# i 12 more variables: sched_arr_time <int>, arr_delay <dbl>,
#   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>,
#   dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
#   minute <dbl>, time_hour <dttm>
```

1. Filtrer des données

filter() : opérateurs logiques

Pour sélectionner des données, nous avons besoin de réaliser des **comparaisons**.

- > Supérieur à
- >= Supérieur ou égal à
- < Inférieur
- <= Inférieur ou égal à
- == Égal à
- != Différent de

Attention à l'arithmétique en “point flottant” :

```
sqrt(2) ^ 2 == 2
```

```
[1] FALSE
```

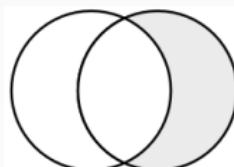
```
1 / 49 * 49 == 1
```

```
[1] FALSE
```

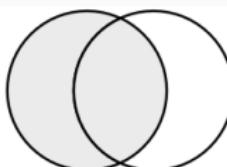
1. Filtrer des données

`filter()` : opérateurs logiques

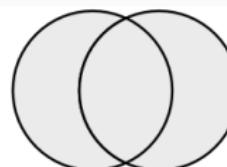
Pour enchaîner les comparaisons, nous avons besoin d'utiliser des opérateurs logiques.



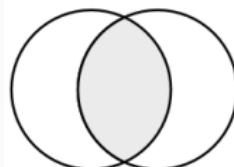
$y \& !x$



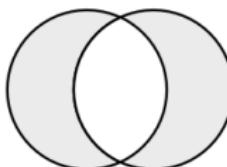
x



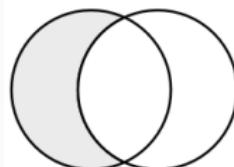
$x \mid y$



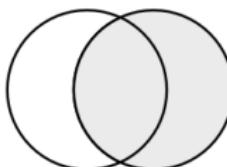
$x \& y$



$xor(x, y)$



$x \& !y$



y

1. Filtrer des données

filter() : opérateurs logiques

Par exemple, pour afficher tous les vols de Novembre ou Décembre :

```
filter(flights, month == 11 | month == 12)

# A tibble: 55,403 x 19
  year month   day dep_time sched_dep_time dep_delay arr_time
  <int> <int> <int>    <int>          <int>     <dbl>     <int>
1 2013     11     1       5        2359       6      352
2 2013     11     1      35        2250      105      123
3 2013     11     1     455        500       -5      641
4 2013     11     1     539        545       -6      856
5 2013     11     1     542        545       -3      831
6 2013     11     1     549        600      -11      912
7 2013     11     1     550        600      -10      705
8 2013     11     1     554        600       -6      659
9 2013     11     1     554        600       -6      826
10 2013    11     1     554        600       -6      749
# i 55,393 more rows
# i 12 more variables: sched_arr_time <int>, arr_delay <dbl>,
#   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>,
#   dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
#   minute <dbl>, time_hour <dttm>
```

1. Filtrer des données

filter() : opérateurs logiques

Un autre opérateur souvent utile : %in%

```
filter(flights, month %in% c(11, 12))

# A tibble: 55,403 x 19
  year month   day dep_time sched_dep_time dep_delay arr_time
  <int> <int> <int>    <int>          <int>     <dbl>     <int>
1 2013     11     1       5        2359       6      352
2 2013     11     1      35        2250      105      123
3 2013     11     1     455        500       -5      641
4 2013     11     1     539        545       -6      856
5 2013     11     1     542        545       -3      831
6 2013     11     1     549        600      -11      912
7 2013     11     1     550        600      -10      705
8 2013     11     1     554        600       -6      659
9 2013     11     1     554        600       -6      826
10 2013    11     1     554        600      -6      749
# i 55,393 more rows
# i 12 more variables: sched_arr_time <int>, arr_delay <dbl>,
#   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>,
#   dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
#   minute <dbl>, time_hour <dttm>
```

1. Filtrer des données

`filter()` : opérateurs logiques

Loi de Morgan :

- ▶ $!(x \ \& \ y)$ est équivalent à $!x \ \mid \ !y$
- ▶ $!(x \ \mid \ y)$ est équivalent à $!x \ \& \ !y$

Ainsi, pour trouver les vols qui n'ont pas eu plus de 2h de retard (ni au départ, ni à l'arrivée), les 2 commandes suivantes sont équivalentes¹⁰ :

```
filter(flights, !(arr_delay > 120 | dep_delay > 120))  
filter(flights, arr_delay <= 120, dep_delay <= 120)
```

10. Personnellement, je trouve la seconde plus facile à comprendre!

1. Filtrer des données

`filter()` : exercices

A. Trouver tous les vols qui :

1. ont eu 2 heures de retard ou plus à leur arrivée
2. ont volé vers houston (IAH ou HOU)
3. ont été affrétés par United Airlines, American Airlines ou Delta Airlines
4. ont décollé l'été (juillet, août et septembre)
5. sont arrivés avec plus de 2h de retard mais on décollé à l'heure
6. sont partis avec au moins une heure de retard, mais ont rattrappé au moins 30 minutes de retard en vol
7. ont décollé entre minuit et 6h du matin (inclus)

B. Une fonction utile de dplyr est `between()`. Que fait cette fonction ?

Utilisez là pour simplifier les réponses aux questions précédentes.

C. Combien de vols ont un `dep_time` manquant ? Pour ces vols, quelles autres variables sont manquantes ? Que peuvent représenter ces lignes ?

Manipuler des données : 2. Trier

2. Trier des données

La fonction `arrange()`

`arrange()` fonctionne comme `filter()`, mais au lieu de sélectionner des lignes, elle permet de modifier **l'ordre des lignes** d'un tableau.

```
arrange(flights, year, month, day)

# A tibble: 336,776 x 19
  year month   day dep_time sched_dep_time dep_delay arr_time
  <int> <int> <int>    <int>          <int>     <dbl>    <int>
1 2013     1     1      517            515        2     830
2 2013     1     1      533            529        4     850
3 2013     1     1      542            540        2     923
4 2013     1     1      544            545       -1    1004
5 2013     1     1      554            600       -6     812
6 2013     1     1      554            558       -4     740
7 2013     1     1      555            600       -5     913
8 2013     1     1      557            600       -3     709
9 2013     1     1      557            600       -3     838
10 2013    1     1      558            600       -2     753
# i 336,766 more rows
# i 12 more variables: sched_arr_time <int>, arr_delay <dbl>,
#   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>,
#   dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
#   minute <dbl>, time_hour <dttm>
```

2. Trier des données

La fonction `arrange()`

La fonction `desc()` permet de trier par ordre **décroissant**:

```
arrange(flights, desc(dep_delay))

# A tibble: 336,776 x 19
  year month   day dep_time sched_dep_time dep_delay arr_time
  <int> <int> <int>    <int>          <int>     <dbl>    <int>
1 2013     1     9       641            900      1301    1242
2 2013     6    15      1432           1935      1137    1607
3 2013     1    10      1121           1635      1126    1239
4 2013     9    20      1139           1845      1014    1457
5 2013     7    22       845           1600      1005    1044
6 2013     4    10      1100           1900      960     1342
7 2013     3    17      2321            810      911     135
8 2013     6    27       959           1900      899     1236
9 2013     7    22      2257            759      898     121
10 2013    12     5       756           1700      896     1058
# i 336,766 more rows
# i 12 more variables: sched_arr_time <int>, arr_delay <dbl>,
#   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>,
#   dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
#   minute <dbl>, time_hour <dttm>
```

2. Trier des données

La fonction `arrange()`

Les **valeurs manquantes** sont toujours placées **à la fin** :

```
df <- tibble(x = c(5, 2, NA))
arrange(df, x)
```

```
# A tibble: 3 x 1
      x
  <dbl>
1     2
2     5
3    NA
```

```
arrange(df, desc(x))
```

```
# A tibble: 3 x 1
      x
  <dbl>
1     5
2     2
3    NA
```

2. Trier des données

arrange() : exercices

1. Trouvez les vols les plus retardés (au départ).
2. Trouvez les vols qui ont décollé le plus tôt (*i.e.* avec le plus d'avance)
3. Trouvez les vols les plus rapides (*i.e.* les plus courts)
4. Trouvez les vols les plus longs et les plus courts (en distance)

Manipuler des données :

3. Sélectionner

3. Sélectionner des données

La fonction `select()`

Il n'est pas rare de travailler avec des jeux de données contenant des centaines ou même des milliers de variables. Le 1^{er} travail est alors souvent de **sélectionner les variables utiles**.

`select()` permet de zoomer rapidement sur un **sous-ensemble** de variables en utilisant des opérations sur **les noms des variables**.

`select()` n'est pas très utile pour `flights` car ce jeu de données ne compte que 19 variables. Néanmoins on peut malgré tout découvrir les **grands principes**.

3. Sélectionner des données

La fonction `select()`

Sélection de colonnes par leur nom :

```
select(flights, year, month, day)

# A tibble: 336,776 x 3
  year month   day
  <int> <int> <int>
1 2013     1     1
2 2013     1     1
3 2013     1     1
4 2013     1     1
5 2013     1     1
6 2013     1     1
7 2013     1     1
8 2013     1     1
9 2013     1     1
10 2013    1     1
# i 336,766 more rows
```

3. Sélectionner des données

La fonction `select()`

Sélection de toutes les colonnes entre `year` et `day` (inclus) :

```
select(flights, year:day)

# A tibble: 336,776 x 3
  year month   day
  <int> <int> <int>
1 2013     1     1
2 2013     1     1
3 2013     1     1
4 2013     1     1
5 2013     1     1
6 2013     1     1
7 2013     1     1
8 2013     1     1
9 2013     1     1
10 2013    1     1
# i 336,766 more rows
```

3. Sélectionner des données

La fonction `select()`

Sélection de toutes les colonnes sauf celles entre year et day (incluses) :

```
select(flights, -(year:day))

# A tibble: 336,776 x 16
  dep_time sched_dep_time dep_delay arr_time sched_arr_time arr_delay
  <int>        <int>     <dbl>    <int>        <int>     <dbl>
1      517          515       2      830         819      11
2      533          529       4      850         830      20
3      542          540       2      923         850      33
4      544          545      -1     1004        1022     -18
5      554          600      -6      812         837     -25
6      554          558      -4      740         728      12
7      555          600      -5      913         854      19
8      557          600      -3      709         723     -14
9      557          600      -3      838         846      -8
10     558          600     -2      753         745       8
# i 336,766 more rows
# i 10 more variables: carrier <chr>, flight <int>, tailnum <chr>,
#   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>,
#   hour <dbl>, minute <dbl>, time_hour <dttm>
```

3. Sélectionner des données

`select()` : fonctions d'aide

Il s'agit de fonctions qui permettent de sélectionner **plusieurs variables** à la fois avec plusieurs méthodes.

- ▶ `starts_with("abc")` : toutes les variables de le nom commence par "abc"
- ▶ `ends_with("abc")` : toutes les variables de le nom se termine par "xyz"
- ▶ `contains("ijk")` : toutes les variables de le nom contient "ijk"
- ▶ `num_range("x", 1:3)` : les variables dont le nom est x1, x2 et x3

3. Sélectionner des données

`select()` : renommer

```
select(flights, Depart = dep_time)

# A tibble: 336,776 x 1
  Depart
  <int>
1     517
2     533
3     542
4     544
5     554
6     554
7     555
8     557
9     557
10    558
# i 336,766 more rows
```

3. Sélectionner des données

`select()` : renommer

Il est donc souvent plus pertinent d'utiliser `rename()`

```
rename(flights, Depart = dep_time)

# A tibble: 336,776 x 19
  year month   day Depart sched_dep_time dep_delay arr_time
  <int> <int> <int> <int>           <int>     <dbl>     <int>
1 2013     1     1    517         515        2      830
2 2013     1     1    533         529        4      850
3 2013     1     1    542         540        2      923
4 2013     1     1    544         545       -1     1004
5 2013     1     1    554         600       -6      812
6 2013     1     1    554         558       -4      740
7 2013     1     1    555         600       -5      913
8 2013     1     1    557         600       -3      709
9 2013     1     1    557         600       -3      838
10 2013    1     1    558         600       -2      753
# i 336,766 more rows
# i 12 more variables: sched_arr_time <int>, arr_delay <dbl>,
#   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>,
#   dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
#   minute <dbl>, time_hour <dttm>
```

3. Sélectionner des données

`select()` : fonctions d'aide (*bis*)

`everything()` : sélectionne toutes les variables non listées :

```
select(flights, dep_delay, arr_delay, everything())

# A tibble: 336,776 x 19
  dep_delay arr_delay year month   day dep_time sched_dep_time
  <dbl>     <dbl> <int> <int> <int>    <int>          <int>
1       2        11  2013     1     1      517            515
2       4        20  2013     1     1      533            529
3       2        33  2013     1     1      542            540
4      -1       -18  2013     1     1      544            545
5      -6       -25  2013     1     1      554            600
6      -4        12  2013     1     1      554            558
7      -5        19  2013     1     1      555            600
8      -3       -14  2013     1     1      557            600
9      -3        -8  2013     1     1      557            600
10     -2         8  2013     1     1      558            600
# i 336,766 more rows
# i 12 more variables: arr_time <int>, sched_arr_time <int>,
#   carrier <chr>, flight <int>, tailnum <chr>, origin <chr>,
#   dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
#   minute <dbl>, time_hour <dttm>
```

3. Sélectionner des données

`select()` : exercices

1. Que se passe-t'il si on utilise plusieurs fois le même nom de variable avec `select()` ?
2. Que fait la fonction `one_of()` ? Pourquoi cette fonction peut-elle être utile avec un vecteur tel que :

```
vars <- c("year", "month", "day", "dep_delay", "arr_delay")
```

3. Le résultat de la commande suivante vous surprend-il ? Comment sont gérées, par défaut, les majuscules par les “helper functions” ? Comment peut-on changer ce comportement ?

```
select(flights, contains("TIME"))
```

Manipuler des données : 4. Modifier

4. Modifier des données

La fonction `mutate()` : principe

Mis à part la sélection de variables particulières avec `select()`, il est souvent utile de créer de nouvelles variables qui seront des **fonctions de variables existantes**.

C'est le rôle de `mutate()`

`mutate()` ajoute systématiquement les nouvelles variables à la fin du jeu de données fourni. Nous allons donc commencer par créer un jeu de données plus “étroit” afin de pouvoir visualiser les nouvelles variables que nous allons créer par la suite :

```
flights_sml <- select(flights, month:day,  
                      ends_with("delay"),  
                      distance,  
                      air_time)
```

4. Modifier des données

La fonction `mutate()` : principe

```
flights_sml

# A tibble: 336,776 x 6
  month   day dep_delay arr_delay distance air_time
  <int> <int>     <dbl>     <dbl>     <dbl>     <dbl>
1     1     1         2         4        11      1400      227
2     1     1         1         4        20      1416      227
3     1     1         2         2        33      1089      160
4     1     1        -1        -18       -18      1576      183
5     1     1        -6        -25       -25      762       116
6     1     1        -4         12        12      719       150
7     1     1        -5         19        19      1065      158
8     1     1        -3        -14       -14      229       53
9     1     1        -3         -8        -8      944       140
10    1     1        -2          8          8      733      138
# i 336,766 more rows
```

4. Modifier des données

`mutate()`: exemples

```
mutate(flights_sml,
       gain = dep_delay - arr_delay,
       speed = distance / air_time * 60)

# A tibble: 336,776 x 8
  month   day dep_delay arr_delay distance air_time    gain speed
  <int> <int>     <dbl>     <dbl>     <dbl>     <dbl> <dbl> <dbl>
1     1     1         2         11      1400      227    -9  370.
2     1     1         4         20      1416      227   -16  374.
3     1     1         2         33      1089      160   -31  408.
4     1     1        -1        -18      1576      183    17  517.
5     1     1        -6        -25      762       116    19  394.
6     1     1        -4         12      719       150   -16  288.
7     1     1        -5         19     1065      158   -24  404.
8     1     1        -3        -14      229       53    11  259.
9     1     1        -3         -8      944      140     5  405.
10    1     1        -2          8      733      138   -10  319.
# i 336,766 more rows
```

4. Modifier des données

mutate() : exemples

Il est également possible d'utiliser des noms de variables juste créées :

```
mutate(flights_sml, gain = dep_delay - arr_delay,
       hours = air_time / 60,
       gain_per_hour = gain / hours)

# A tibble: 336,776 x 9
  month   day dep_delay arr_delay distance air_time    gain hours
  <int> <int>     <dbl>     <dbl>     <dbl>     <dbl> <dbl> <dbl>
1     1     1        2        11      1400      227    -9  3.78
2     1     1        4        20      1416      227   -16  3.78
3     1     1        2        33      1089      160   -31  2.67
4     1     1       -1       -18      1576      183    17  3.05
5     1     1       -6       -25      762       116    19  1.93
6     1     1       -4        12      719       150   -16  2.5
7     1     1       -5        19     1065      158   -24  2.63
8     1     1       -3       -14      229       53    11  0.883
9     1     1       -3        -8      944      140     5  2.33
10    1     1       -2         8      733      138   -10  2.3
# i 336,766 more rows
# i 1 more variable: gain_per_hour <dbl>
```

4. Modifier des données

`mutate()` : alternative

Si l'on souhaite uniquement conserver les variables nouvellement créées, on utilise `transmute()` :

```
transmute(flights_sml, gain = dep_delay - arr_delay,
          hours = air_time / 60,
          gain_per_hour = gain / hours)

# A tibble: 336,776 x 3
  gain hours gain_per_hour
  <dbl> <dbl>      <dbl>
1   -9  3.78     -2.38
2  -16  3.78     -4.23
3  -31  2.67    -11.6 
4   17  3.05      5.57 
5   19  1.93      9.83 
6  -16  2.5       -6.4  
7  -24  2.63     -9.11 
8   11  0.883     12.5  
9    5  2.33      2.14  
10  -10  2.3     -4.35 
# i 336,766 more rows
```

4. Modifier des données

`mutate()` : fonctions utiles

Toute fonction prenant un vecteur en argument et retournant un vecteur de **même longueur**.

- ▶ Opérateurs arithmétiques : `+`, `-`, `*`, `/`, `^`
- ▶ Arithmétique modulaire : `%/%` et `%%`
- ▶ Logarithmes : `log()`, `log10()` et `log2()`
- ▶ Décalages : `lag()` et `lead()`
- ▶ Fonctions aggrégatives cumulées : `cumsum()`, `cummean()`,
`cumprod()`, `cummin()`, `cummax()`
- ▶ Comparaisons logiques : `<`, `>`, `<=`, `>=`, `==`, `!=`
- ▶ Fonctions de rang : `min_rank()`, `desc()`, `row_number()`,
`dense_rank()`, `percent_rank()`, `cume_dist()`, `ntile()`, ...

4. Modifier des données

`mutate()` : exercices

1. Actuellement, `dep_time` et `sched_dep_time` sont pratiques à examiner, mais il est difficile d'effectuer des calculs avec ces variables car ce ne sont pas réellement des nombres continus. Transformez-les en nombre de minutes depuis minuit, ce qui sera plus facile à manipuler
2. Comparez `air_time` à `arr_time - dep_time`. Que devriez-vous trouver? Qu'obtenez vous? Que faire pour régler le problème?
3. Comparez `dep_time`, `sched_dep_time` et `dep_delay`. Quelle relation vous attendez-vous à observer entre ces 3 variables?
4. Trouvez les 10 vols les plus retardés en utilisant une fonction de rang. Comment souhaitez-vous traiter les égalités? Lisez attentivement la documentation de `min_rank()`.
5. Quel est le résultat de `1:3 + 1:10`? Pourquoi?

Manipuler des données : 5. Résumer

5. Résumer des données

La fonction `summarise()` : principe

Le dernier verbe essentiel est `summarise()`. Il permet de réduire un jeu de données à une unique ligne :

```
summarise(flights, delay = mean(dep_delay, na.rm = TRUE))  
# A tibble: 1 x 1  
  delay  
  <dbl>  
1 12.6
```

Ici, nous apprenons que sur l'ensemble des vols de 2013, le retard moyen des vols au départ vaut 12.6 minutes.

`summarise()` n'est pas terriblement utile quand on l'utilise seule. Mais en combinaison avec `group_by()`, ça change tout...

5. Résumer des données

La fonction `summarise()` : principe

Ainsi, en appliquant la même fonction aux données groupées par mois, on obtient aisément la moyenne mensuelle des retards :

```
by_month <- group_by(flights, year, month)
summarise(by_month, delay = mean(dep_delay, na.rm = TRUE))

# A tibble: 12 x 3
# Groups:   year [1]
  year month delay
  <int> <int> <dbl>
1 2013     1 10.0
2 2013     2 10.8
3 2013     3 13.2
4 2013     4 13.9
5 2013     5 13.0
6 2013     6 20.8
7 2013     7 21.7
8 2013     8 12.6
9 2013     9  6.72
10 2013    10  6.24
11 2013    11  5.44
12 2013    12 16.6
```

Les résumés groupés sont très utiles lors de l'exploration des données.

5. Résumer des données

`summarise()` : digression, le pipe

Imaginons le scénario suivant : nous souhaitons étudier la relation entre la **distance parcourue** en vol d'une part et les **retards observés à l'arrivée** pour chaque destination d'autre part.

En utilisant ce nous avons vu jusqu'ici, nous pourrions taper le code suivant¹¹ :

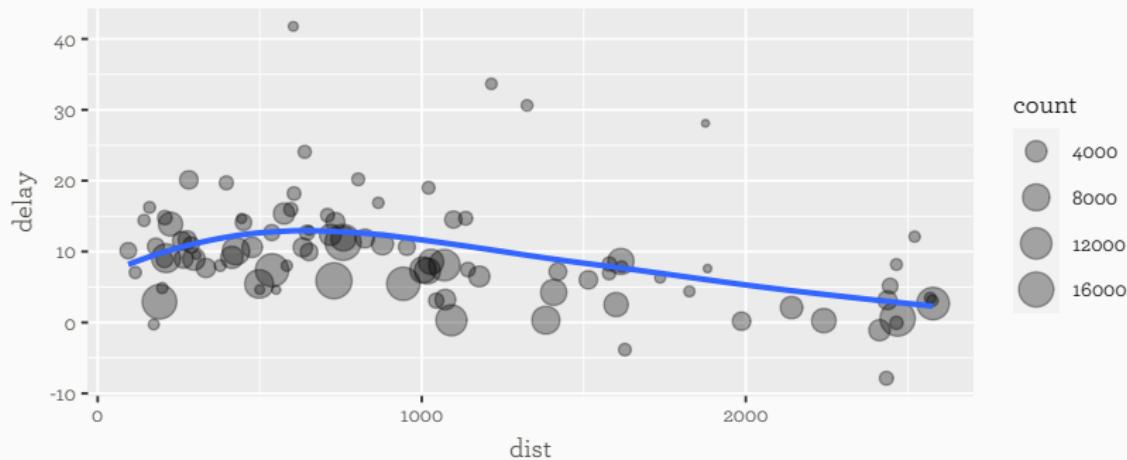
```
by_dest <- group_by(flights, dest)
delay <- summarise(by_dest,
  count = n(),
  dist = mean(distance, na.rm = TRUE),
  delay = mean(arr_delay, na.rm = TRUE)
)
delay_clean <- filter(delay, count > 20, dest != "HNL")
```

11. On commence par grouper les vols par destination. On calcule ensuite, pour chaque destination, le nombre de vols, la distance moyenne parcourue et le retard moyen à l'arrivée. Enfin, on élimine les destinations trop peu fréquentées ainsi qu'Honolulu (qui est 2 fois plus éloignée que toutes les autres destinations)

5. Résumer des données

summarise() : digression, le pipe

```
ggplot(data = delay_clean, mapping = aes(x = dist, y = delay)) +  
  geom_point(aes(size = count), alpha = 1/3) +  
  geom_smooth(se = FALSE)
```



Les retards augmentent avec la distance, jusqu'à 700 km environ, puis la relation est négative. C'est comme si les vols plus longs permettaient de rattraper une partie du retard accumulé au départ.

5. Résumer des données

`summarise()` : digression, le pipe

Nous avons donc ici utilisé 4 (groupes de) fonctions :

1. `group_by()`
2. `summarise()`
3. `filter()`
4. `ggplot()` et autres fonctions associées

L'un des problèmes de cette approche est que nous avons dû créer des objets intermédiaires dont le contenu ne nous intéresse pas vraiment :

1. `by_dest`
2. `delay`
3. `delay_clean`

Nommer est difficile et cette syntaxe met l'accent sur les objets au lieu de mettre l'accent sur les actions réalisées.

5. Résumer des données

`summarise()` : digression, le pipe

L'alternative ? Utiliser le pipe `%>%!`

```
delays <- flights %>%
  group_by(dest) %>%
  summarise(
    count = n(),
    dist = mean(distance, na.rm = TRUE),
    delay = mean(arr_delay, na.rm = TRUE)
  ) %>%
  filter(count > 20, dest != "HNL")
```

Concrètement, le pipe récupère l'objet qui est à gauche et le transmet à la fonction qui suit en tant que **premier argument**.

Ainsi,

- ▶ $x \%>\% f(y)$ est équivalent à $f(x, y)$
- ▶ $x \%>\% f(y) \%>\% g(z)$ est équivalent à $g(f(x, y), z)$

5. Résumer des données

`summarise()` : digression, les valeurs manquantes

Il est important de spécifier la façon dont les NAs doivent être traités :

```
flights %>%  
  group_by(year, month, day) %>%  
  summarise(mean = mean(dep_delay))  
  
# A tibble: 365 x 4  
# Groups:   year, month [12]  
  year month   day   mean  
  <int> <int> <int> <dbl>  
1 2013     1     1    NA  
2 2013     1     2    NA  
3 2013     1     3    NA  
4 2013     1     4    NA  
5 2013     1     5    NA  
6 2013     1     6    NA  
7 2013     1     7    NA  
8 2013     1     8    NA  
9 2013     1     9    NA  
10 2013    1    10    NA  
# i 355 more rows
```

5. Résumer des données

`summarise()` : digression, les valeurs manquantes

Ici, il y a 2 façons de faire :

1. On peut spécifier à la fonction `mean()` que faire des valeurs manquantes :

```
flights %>%
  group_by(year, month, day) %>%
  summarise(mean = mean(dep_delay, na.rm = TRUE))
```

2. On peut filtrer les valeurs manquantes en amont.¹² Donnons un nom à cet objet que nous ré-utiliserons plus tard.

```
not_cancelled <- flights %>%
  filter(!is.na(dep_delay), !is.na(arr_delay))

not_cancelled %>%
  group_by(year, month, day) %>%
  summarise(mean = mean(dep_delay))
```

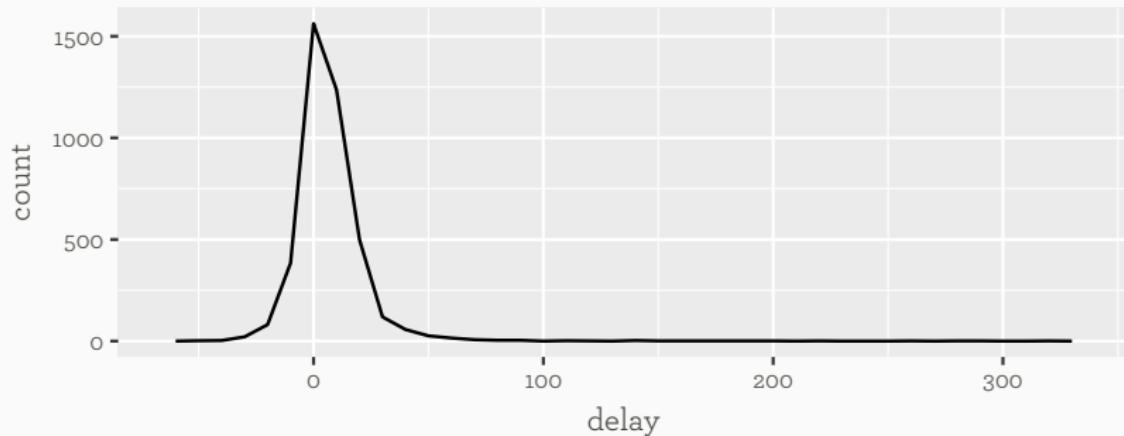
12. Ici, les valeurs manquantes représentent des vols annulés.

5. Résumer des données

`summarise()` : les comptages

Quand on agrège, il est toujours utile d'intégrer soit le nombre d'observations, soit le nombre d'observations hors valeurs manquantes.

```
delays <- not_cancelled %>%  
  group_by(tailnum) %>%  
  summarise(delay = mean(arr_delay))  
  
ggplot(data = delays, aes(x = delay)) + geom_freqpoly(binwidth = 10)
```

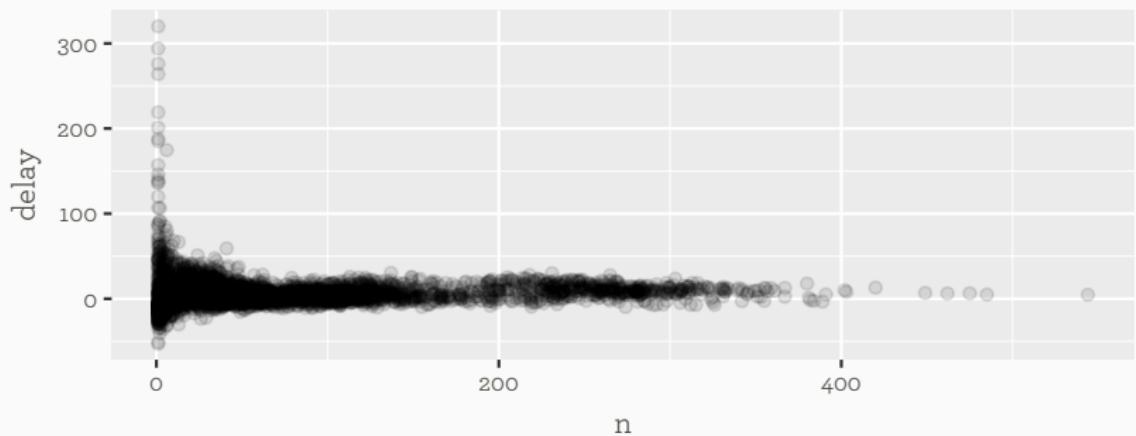


5. Résumer des données

summarise() : les comptages

```
delays <- not_cancelled %>%
  group_by(tailnum) %>%
  summarise(delay = mean(arr_delay),
            n = n())

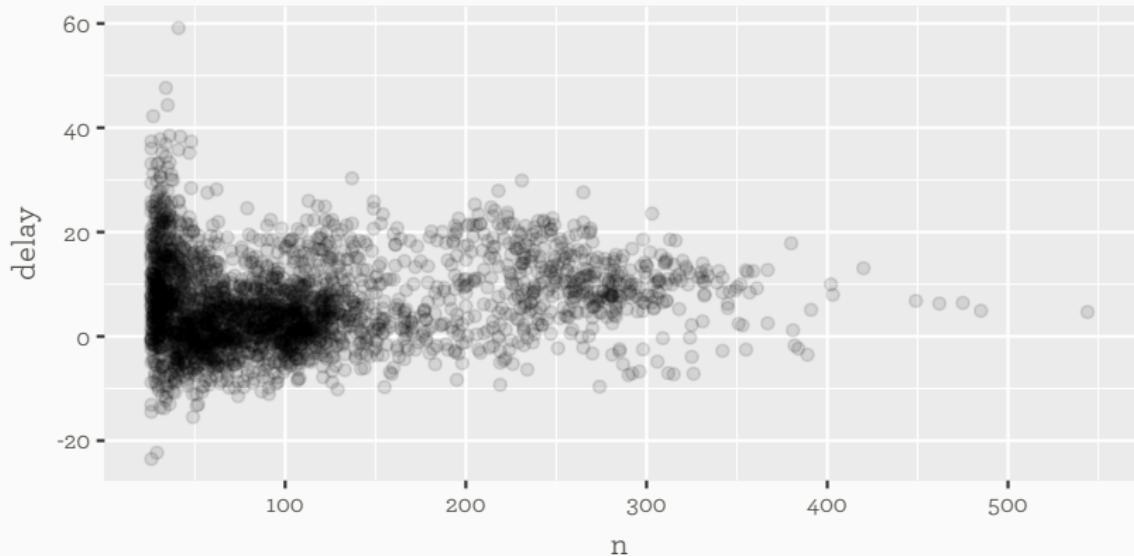
ggplot(data = delays, aes(x = n, y = delay)) +
  geom_point(alpha = 1/10)
```



5. Résumer des données

summarise() : les comptages

```
delays %>%
  filter(n > 25) %>%
  ggplot(aes(x = n, y = delay)) +
  geom_point(alpha = 1/10)
```



5. Résumer des données

summarise() : autres fonctions utiles

L'utilisation de "subsets" est souvent utile pour obtenir des résumés sur une partie seulement des données :

```
not_cancelled %>%
  group_by(year, month, day) %>%
  summarise(
    avg_delay1 = mean(arr_delay),
    avg_delay2 = mean(arr_delay[arr_delay > 0]))
  )

# A tibble: 365 x 5
# Groups:   year, month [12]
  year month   day avg_delay1 avg_delay2
  <int> <int> <int>      <dbl>      <dbl>
1  2013     1     1      12.7      32.5
2  2013     1     2      12.7      32.0
3  2013     1     3      5.73     27.7
4  2013     1     4     -1.93     28.3
5  2013     1     5     -1.53     22.6
6  2013     1     6      4.24     24.4
# i 359 more rows
```

5. Résumer des données

`summarise()` : autres fonctions utiles

Les mesures de **dispersion** : `var()`, `sd()`, `IQR()`, `mad()`...

```
# Pourquoi la distance vers certaines destinations
# est-elle plus variable que d'autres ?
not_cancelled %>%
  group_by(dest) %>%
  summarise(distance_sd = sd(distance)) %>%
  arrange(desc(distance_sd))

# A tibble: 104 x 2
  dest   distance_sd
  <chr>     <dbl>
1 EGE        10.5
2 SAN        10.4
3 SFO        10.2
4 HNL        10.0
5 SEA        9.98
6 LAS        9.91
# i 98 more rows
```

5. Résumer des données

`summarise()` : autres fonctions utiles

Les mesures de `rang` : `min()`, `max()`, `quantile(x, 0.25)`...

```
# À quelle heure décollent les premiers et derniers vols chaque jour ?
not_cancelled %>%
  group_by(year, month, day) %>%
  summarise(
    first = min(dep_time),
    last = max(dep_time)
  )

# A tibble: 365 x 5
# Groups:   year, month [12]
  year month   day first  last
  <int> <int> <int> <int> <int>
1 2013     1     1    517  2356
2 2013     1     2     42  2354
3 2013     1     3     32  2349
4 2013     1     4     25  2358
5 2013     1     5     14  2357
6 2013     1     6     16  2355
# i 359 more rows
```

5. Résumer des données

`summarise()` : autres fonctions utiles

Les mesures de `position` : `first(x)`, `nth(x, 2)`, `last(x)`...

```
# À quelle heure décollent les premiers et derniers vols chaque jour ?
not_cancelled %>%
  group_by(year, month, day) %>%
  summarise(
    first = first(dep_time),
    last = last(dep_time)
  )

# A tibble: 365 x 5
# Groups:   year, month [12]
  year month   day first  last
  <int> <int> <int> <int> <int>
1 2013     1     1    517  2356
2 2013     1     2     42  2354
3 2013     1     3     32  2349
4 2013     1     4     25  2358
5 2013     1     5     14  2357
6 2013     1     6     16  2355
# i 359 more rows
```

5. Résumer des données

`summarise()` : autres fonctions utiles

Les fonctions de `comptage` : `n()`, `sum(!is.na(x))`, `n_distinct()`...

```
# Quelles sont les destinations desservies
# par le plus grand nombre de compagnies ?
not_cancelled %>%
  group_by(dest) %>%
  summarise(n_carriers = n_distinct(carrier)) %>%
  arrange(desc(n_carriers))

# A tibble: 104 x 2
  dest   n_carriers
  <chr>     <int>
1 ATL          7
2 BOS          7
3 CLT          7
4 ORD          7
5 TPA          7
6 AUS          6
# i 98 more rows
```

5. Résumer des données

`summarise()` : autres fonctions utiles

Les fonctions de **comptage** :

L'opération qui consiste à grouper puis à résumer par un simple comptage est tellement courante que dplyr fournit un raccourci : `count()`.

```
# Combien de vols ont atteint chaque destination ?
not_cancelled %>%
  count(dest)

# A tibble: 104 x 2
  dest      n
  <chr> <int>
1 ABQ      254
2 ACK      264
3 ALB      418
4 ANC       8
5 ATL    16837
6 AUS     2411
# i 98 more rows
```

5. Résumer des données

`summarise()` : autres fonctions utiles

Les fonctions de **comptage** :

On peut même fournir une pondération avec l'argument `wt`, qui permet ainsi de calculer des sommes.

```
# Quelle distance chaque appareil a-t'il parcourue ?
not_cancelled %>%
  count(tailnum, wt = distance)

# A tibble: 4,037 x 2
  tailnum      n
  <chr>     <dbl>
1 D942DN    3418
2 NOEGMQ    239143
3 N10156    109664
4 N102UW    25722
5 N103US    24619
6 N104UW    24616
# i 4,031 more rows
```

5. Résumer des données

`summarise()` : autres fonctions utiles

Les fonctions de **comptage** : nombres et proportions de **valeurs logiques**.

Utilisés avec des fonctions telles que `sum()` et `mean()`, les TRUE sont transformés en 1 et les FALSE en 0. Les **sommes** donnent donc le **nombre** de vrais et les **moyennes** la **proportion** de vrais.

```
# Combien de vols ont décollé avant 5h00 ?
# (cela indique généralement des vols en retard de la veille)

not_cancelled %>%
  group_by(year, month, day) %>%
  summarise(n_early = sum(dep_time < 500))

# A tibble: 365 x 4
# Groups:   year, month [12]
  year month   day n_early
  <int> <int> <int>    <int>
1 2013     1     1      0
2 2013     1     2      3
3 2013     1     3      4
4 2013     1     4      3
5 2013     1     5      3
6 2013     1     6      2
# i 359 more rows
```

5. Résumer des données

`summarise()` : autres fonctions utiles

Les fonctions de **comptage** : nombres et proportions de **valeurs logiques**.

Utilisés avec des fonctions telles que `sum()` et `mean()`, les TRUE sont transformés en 1 et les FALSE en 0. Les **sommes** donnent donc le **nombre** de vrais et les **moyennes** la **proportion** de vrais.

```
# Quelle proportion de vols est arrivée avec plus d'une heure de retard ?  
  
not_cancelled %>%  
  group_by(year, month, day) %>%  
  summarise(hour_perc = mean(arr_delay > 60))  
  
# A tibble: 365 x 4  
# Groups:   year, month [12]  
  year month   day hour_perc  
  <int> <int> <int>     <dbl>  
1 2013     1     1  0.0722  
2 2013     1     2  0.0851  
3 2013     1     3  0.0567  
4 2013     1     4  0.0396  
5 2013     1     5  0.0349  
6 2013     1     6  0.0470  
# i 359 more rows
```

5. Résumer des données

`summarise()` : exercices

1. Proposez une autre façon d'obtenir les mêmes résultats que ceux produits par cette commande (sans utiliser `count()`) :

```
not_cancelled %>%  
  count(dest)
```

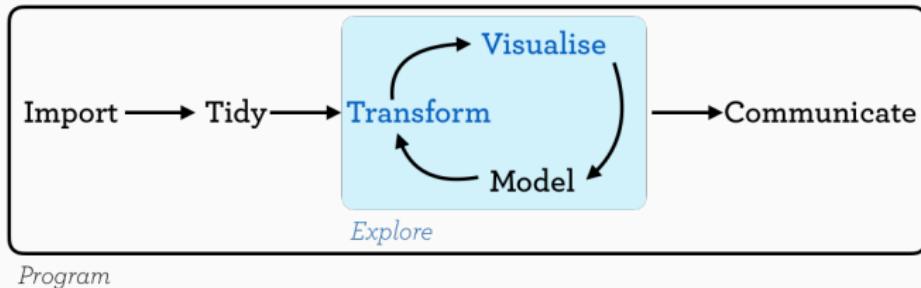
2. Examinez le nombre de vol annulés chaque jour. Y a-t'il une tendance? Est-ce que la proportion de vols annulés est liée au retard moyen?
3. Quelle compagnie aérienne a les retards les plus importants? Pouvez-vous démêler la cause la plus importante de retard entre une mauvaise compagnie et une mauvaise destination (un mauvais aéroport)? Pourquoi/comment? Indice : essayez de partir de ceci :

```
flights %>%  
  group_by(carrier, dest) %>%  
  summarise(n())
```

Mise en forme : Les données **rangées**

Les données rangées

Le package `tidyverse`



Jusqu'ici, nous avons utilisé des jeux de données dans un **format** idéal.

Dans la vraie vie, c'est loin d'être toujours le cas.

Bien souvent, nous aurons besoin de modifier les tableaux dont nous disposons à l'aide de 4 fonctions afin de mettre les données sous un format permettant les analyses et représentations graphiques.

Les données rangées

Le package `tidyverse`

Les 4 fonctions dont nous aurons besoin appartiennent toutes au package `tidyverse`.

Il s'agit des fonctions suivantes :

- ▶ `pivot_longer()`, pour regrouper plusieurs colonnes en 2 variables pertinentes. Cette fonction permet de passer de tableaux “larges” à des tableaux “longs”.
- ▶ `pivot_wider()`, pour faire l’opération inverse.
- ▶ `unite()`, pour unir 2 variables en une seule.
- ▶ `separate()`, pour séparer une variable en 2 variables ou plus.

Vous trouverez toutes les explications concernant le rôle de ces 4 fonctions dans le [chapitre 5.1 du cours en ligne](#).

Tests d'hypothèses :

1. Les grands **principes**

1. Principes des tests statistiques

Les hypothèses

Notion d'**inférence statistique** :

- ▶ On étudie une **population** à partir d'un **échantillon**
 - ▶ On cherche à déterminer la valeur d'un **paramètre** à partir d'un **estimateur**.
1. On formule une **hypothèse nulle** concernant la valeur d'un **paramètre d'intérêt** (e.g. la moyenne μ de la population vaut 32.4)
 2. On cherche à prendre une **décision** concernant cette hypothèse : sommes nous en mesure de la rejeter ou non, compte tenu des données dont nous disposons et de l'incertitude inhérente au processus d'inférence statistique ?

Définition : Hypothèse Nulle

On note H_0 . Elle porte sur la valeur d'un paramètre de la population d'intérêt. L'hypothèse nulle est l'hypothèse "**inintéressante**", celle qu'on aimerait bien rejeter car cela indiquerait que quelque chose d'intéressant, original, inattendu se passe dans la population étudiée.

1. Principes des tests statistiques

Les hypothèses

Important

1. Outre l'hypothèse nulle H_0 , on formule aussi une hypothèse opposée, notée H_A ou H_1 et appelée **hypothèse alternative**.
2. La décision du test sera toujours prise par rapport à H_0 .
3. Les hypothèses sont des **affirmations**, et non des questions !

1. Principes des tests statistiques

La statistique du test

Après les hypothèses, la **statistique du test** est le second ingrédient indispensable aux tests statistiques.

Statistique d'un test

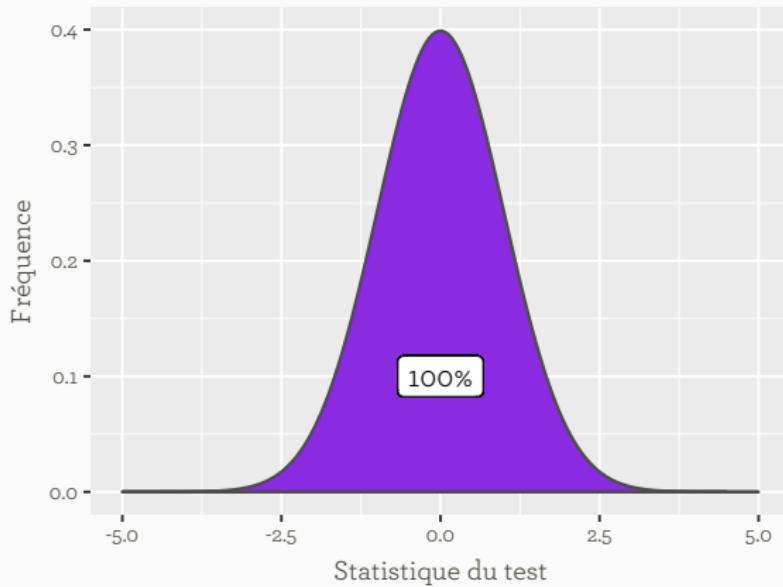
C'est un **nombre calculé** à partir des **données disponibles** et dont on connaît la **distribution** théorique sous H_0 . C'est ce nombre qui, indirectement, nous permettra de prendre une décision concernant le test statistique (et en particulier H_0).

Tout test statistique possède une statistique du test. En voici quelques exemples :

- ▶ Test de normalité de Shapiro-Whilk : W ou W_{calc}
- ▶ Test d'homoscédasticité de Levene : F ou F_{calc}
- ▶ Test d'homoscédasticité de Bartlett : χ^2 ou χ^2_{calc}
- ▶ Test de comparaison de moyennes de Student : t ou t_{calc}

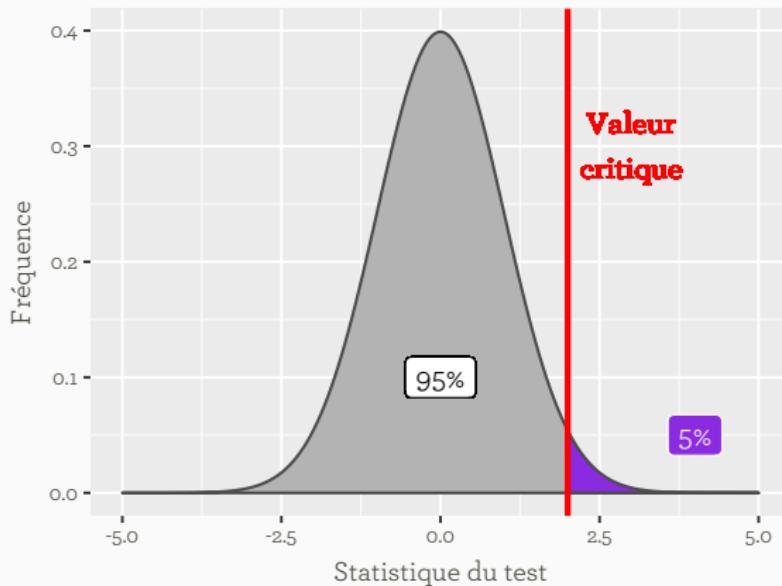
1. Principes des tests statistiques

La valeur critique de la statistique du test



1. Principes des tests statistiques

La valeur critique de la statistique du test



1. Principes des tests statistiques

La décision

On compare la statistique du test à la valeur critique :

- ▶ Si la statistique du test est **supérieure ou égale** à la valeur critique, on **rejette H_0** (et on valide donc H_A).
- ▶ Si la statistique du test est **inférieure** à la valeur critique, **on ne peut pas rejeter H_0** .

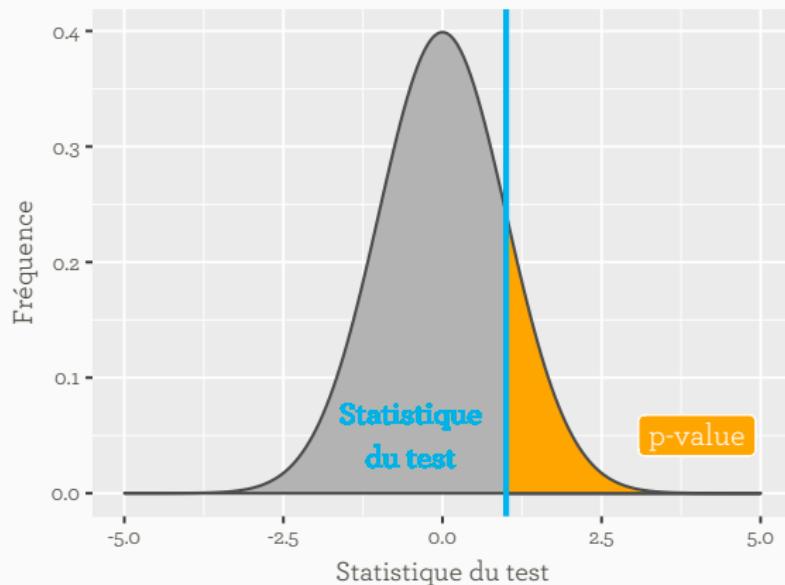
Important

- ▶ La décision du test se prend **toujours** par rapport à H_0 .
- ▶ On ne dit jamais que H_0 est vraie. Au mieux, on dit qu'**on ne peut pas rejeter H_0** .

1. Principes des tests statistiques

La *p*-value

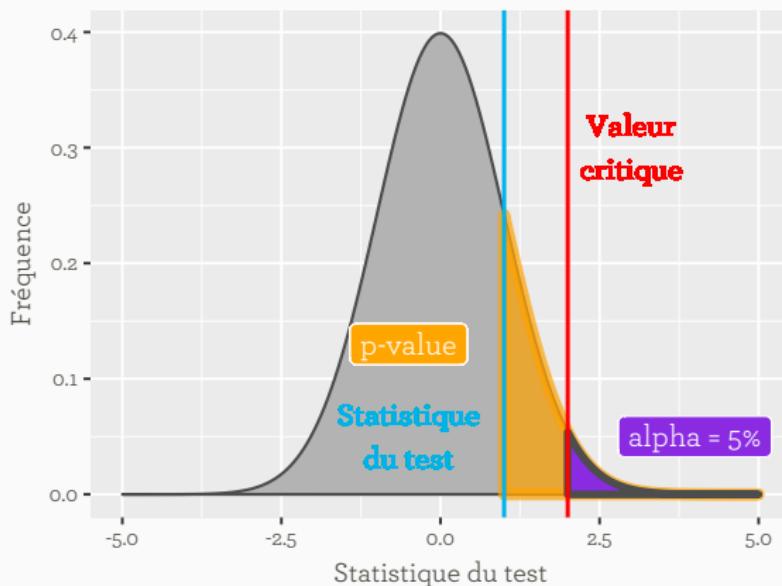
Aujourd'hui, on n'a plus besoin de valeurs critiques. On n'a plus besoin de tables statistiques : on utilise la *p*-value.



1. Principes des tests statistiques

La p -value

Mettons tout ça sur le même graphique :



1. Principes des tests statistiques

La *p*-value

Il y a donc équivalence entre les éléments suivants :

- ▶ Statistique du test < valeur critique $\Leftrightarrow p\text{-value} > \alpha$
- ▶ Statistique du test \geq valeur critique $\Leftrightarrow p\text{-value} \leq \alpha$

Important

Quand la *p*-value est inférieure ou égale au seuil α (généralement fixé à 5% ou 0.05 dans le domaine des Sciences du vivant), on rejette l'hypothèse nulle H_0 du test statistique. Sinon, on ne peut pas rejeter H_0 .

Dans R, toutes les fonctions permettant de réaliser des tests statistiques renvoient, au minimum, la valeur de la statistique du test, et la *p*-value associée.

À vous de choisir H_0 et H_A et de fixer α **avant** de réaliser le test

1. Principes des tests statistiques

La *p*-value

Definition : la p-value

- ▶ Ça n'est pas la probabilité que H_0 soit vraie ou fause
- ▶ C'est la probabilité, si H_0 est vraie, d'obtenir un effet au moins aussi grand que celui qu'on a observé, sous le seul effet du hasard.

Exemple

	Échantillon	Moyenne
A		$\bar{x}_A = 10$
B		$\bar{x}_B = 11$

H_0 : les moyennes des populations A et B sont égales, $\mu_A = \mu_B$

H_A : les moyennes des populations A et B sont différentes, $\mu_A \neq \mu_B$

Si p -value = 0.021, on rejette H_0

Si p -value = 0.49, on ne peut pas rejeter H_0 .

Tests d'hypothèses :

2. Les **notions** **importantes**

2. Notions importantes

Le seuil de significativité

C'est le seuil α . On le choisit une fois pour toutes avant de réaliser les tests.

À moins d'avoir une bonne raison de faire autrement, on fixe $\alpha = 0.05$.

α est également appelé l'**erreur de type I**

Definition : erreur de type I

C'est la probabilité de rejeter à tort H_0 .

Autrement dit, puisque que tout accusé est présumé innocent (l'innocence de l'accusé est l'hypothèse nulle), α est la probabilité de **condamner un innocent**.

2. Notions importantes

Les 2 types d'erreurs

Il existe un autre type d'erreur : l'erreur de type II, ou **erreur β** .

Definition : erreur de type II

C'est la probabilité d'accepter à tort H_0 .

Autrement dit, c'est la probabilité de relâcher un coupable.

L'erreur de type II **n'est pas sous notre contrôle**! Elle dépend notamment :

- ▶ De la variabilité des données
- ▶ De la taille de l'échantillon
- ▶ Du type de test statistique réalisé

2. Notions importantes

La puissance statistique

Il s'agit d'une autre **probabilité**, qui dépend directement de l'erreur de type II.

On la note $1 - \beta$

Definition : puissance statistique ($1 - \beta$)

C'est la probabilité de **déetecter un effet lorsqu'il y en a réellement un**. Autrement dit, c'est la probabilité de **condamner un coupable**.

On aimerait pouvoir maximiser la puissance. Augmenter la puissance revient à diminuer l'erreur de type II (erreur β).

Problème : diminuer β , c'est augmenter α .

Globalement, si on libère moins d'accusés, on libère moins de coupables (on baisse l'erreur de type II), mais on condamne aussi plus de monde, y compris des innocents (on augmente l'erreur de type I).

2. Notions importantes

La puissance statistique

Les leviers dont nous disposons pour **augmenter la puissance** sont les mêmes que ceux dont nous disposons pour diminuer l'erreur de type II :

- ▶ Augmenter la **taille** de l'échantillon
- ▶ Faire des tests **unilatéraux**
- ▶ Faire des tests **paramétriques**

2. Notions importantes

Tests unilatéraux et bilatéraux

Les tests unilatéraux sont plus puissants que les tests bilatéraux.

Toutefois, à moins d'avoir une bonne raison de faire le contraire, on choisit de préférence les **tests bilatéraux**.

Reprendons l'exemple examiné plus tôt :

Exemple

	Échantillon	Moyenne
A		$\bar{x}_A = 10$
B		$\bar{x}_B = 11$

L'hypothèse nulle est toujours H_0 : les moyennes sont égales, $\mu_A = \mu_B$
Pour l'hypothèse alternative, nous avons **3 choix possibles** :

1. Hypothèse bilatérale, $H_A : \mu_A \neq \mu_B$
2. Hypothèse unilatérale, $H_A : \mu_A > \mu_B$
3. Hypothèse unilatérale, $H_A : \mu_A < \mu_B$

2. Notions importantes

Tests paramétriques et non paramétriques

Définition : test paramétrique

Un test paramétrique est un test qui suppose que les données respectent un certain nombre de conditions qu'il conviendra de vérifier avant de réaliser le test (ou parfois après avoir réalisé le test, comme pour la régression linéaire et l'analyse de variance).

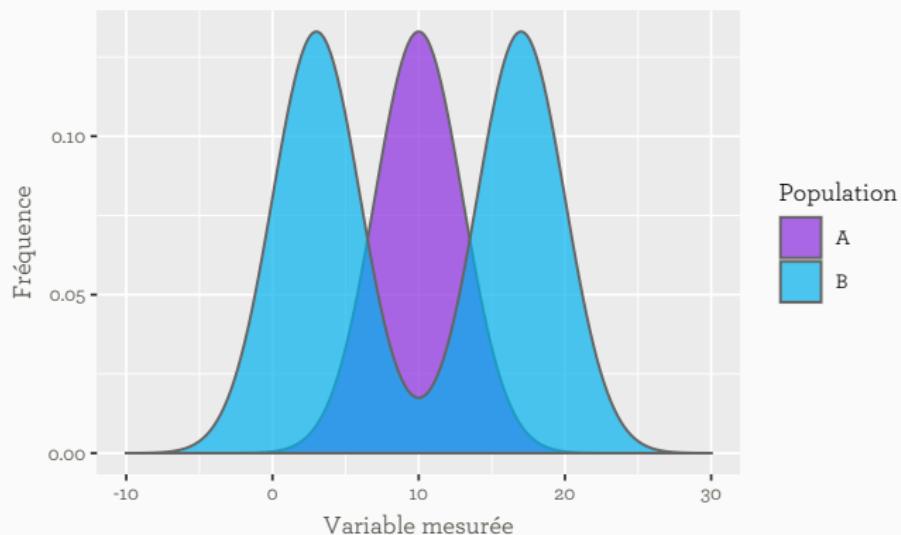
L'une de ces conditions est très souvent la **normalité des données**.

Les tests paramétriques sont plus puissants que les tests non paramétriques car les statistiques de ces tests sont calculées à partir des données observées **non modifiées**

2. Notions importantes

Tests paramétriques et non paramétriques

Faire un test paramétrique avec des données de ce type n'aurait pas de sens :



La moyenne de ces 2 populations vaut 10. Pour autant, peut-on dire que ces 2 populations ont les mêmes caractéristiques ?

Tests d'hypothèses :

3. La procédure paramétriques

Description des étapes à suivre

1. Faire un test de normalité

- H_0 : les données suivent la loi Normale
- H_A : les données ne suivent pas la loi Normale

2. Si les données suivent la loi Normale, faire un test

d'**homoscédasticité** (i.e. test d'homogénéité des variances)

- H_0 : les variances sont homogènes, $\frac{\sigma_A^2}{\sigma_B^2} = 1$
- H_A : les variances ne sont pas homogènes, $\frac{\sigma_A^2}{\sigma_B^2} \neq 1$

3. Si les variances sont homogènes, on peut faire un **test paramétrique de comparaison de moyennes**

- H_0 : les moyennes sont égales, $\mu_A - \mu_B = 0$
- H_A : les moyennes sont différentes, $\mu_A - \mu_B \neq 0$

Si l'un, l'autre ou les 2 échantillons ne suivent pas la loi Normale, inutile d'aller plus loin : on fera un **test de comparaison des moyennes non paramétrique**.

Si les 2 échantillons suivent la loi Normale mais qu'ils n'ont pas même variance : on fera un **test de comparaison des moyennes non paramétrique**.

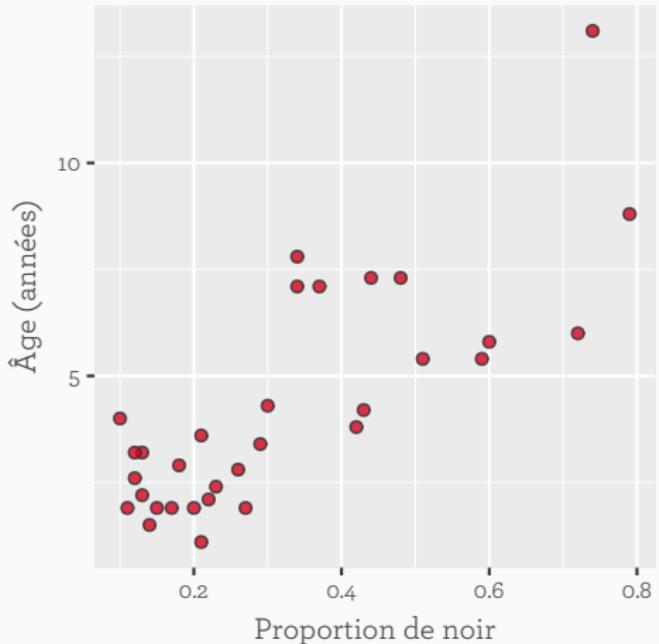
La **régression** linéaire

Régression linéaire simple

Le nez des lions

Données tirées de Whitman *et al.* (2004)

```
# A tibble: 32 x 2
  Prop.black    Age
  <dbl>     <dbl>
1      0.21    1.1
2      0.14    1.5
3      0.11    1.9
4      0.13    2.2
5      0.12    2.6
6      0.13    3.2
# i 26 more rows
```



Régression linéaire simple

Spécifier le modèle

```
lionReg <- lm(Age ~ Prop.black, data = lion)
lionReg

Call:
lm(formula = Age ~ Prop.black, data = lion)

Coefficients:
(Intercept)  Prop.black
            0.879        10.647

confint(lionReg)

                2.5 %    97.5 %
(Intercept) -0.2826733  2.040686
Prop.black   7.5643082 13.729931
```

Régression linéaire simple

Tester la pente et l'ordonnée à l'origine

Ici, nous avons deux groupes d'hypothèses :

- ▶ H_0 : L'**ordonnée à l'origine** (intercept) de la droite de régression est **égale à zéro**.
 - ▶ H_A : L'**ordonnée à l'origine** de la droite de régression est **différente de zéro**.
-
- ▶ H_0 : la **pente** de la droite de régression est **égale à zero**.
 - ▶ H_A : la **pente** de la droite de régression est **différente de zéro**.

Les intervalles de confiance nous donnent une première réponse, mais on peut tester ces hypothèses avec la fonction `lm()` de R.

Régression linéaire simple

Tester la pente et l'ordonnée à l'origine

```
summary(lionReg)

Call:
lm(formula = Age ~ Prop.black, data = lion)

Residuals:
    Min      1Q  Median      3Q     Max 
-2.5449 -1.1117 -0.5285  0.9635  4.3421 

Coefficients:
            Estimate Std. Error t value Pr(>|t|)    
(Intercept)  0.8790    0.5688   1.545   0.133    
Prop.black   10.6471   1.5095   7.053 7.68e-08 ***  
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 1.669 on 30 degrees of freedom
Multiple R-squared:  0.6238, Adjusted R-squared:  0.6113 
F-statistic: 49.75 on 1 and 30 DF,  p-value: 7.677e-08
```

Régression linéaire simple

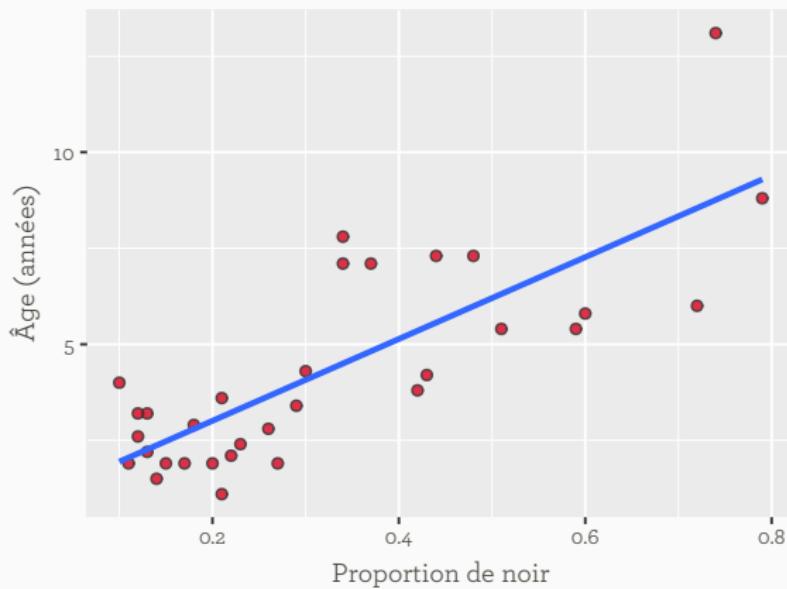
Visualisation graphique

```
# Create a scatter plot and add a regression line
pl <- lion %>%
  ggplot(aes(x = Prop.black, y = Age)) +
  geom_point(shape = 21,
             color = grey(0.2),
             fill = rgb(0.84, 0.01, 0.12, 0.8),
             alpha = 0.8) +
  geom_smooth(method = "lm", se = FALSE) +
  labs(x = "Proportion de noir",
       y = "Âge (années)")
```

Régression linéaire simple

Visualisation graphique

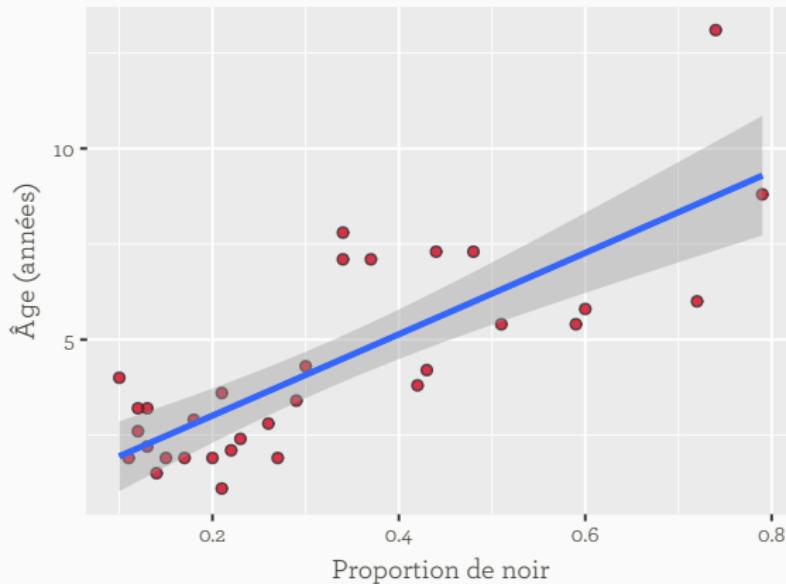
pl



Régression linéaire simple

Intervalle de confiance de la régression

```
pl + geom_smooth(method = "lm", se = TRUE)
```



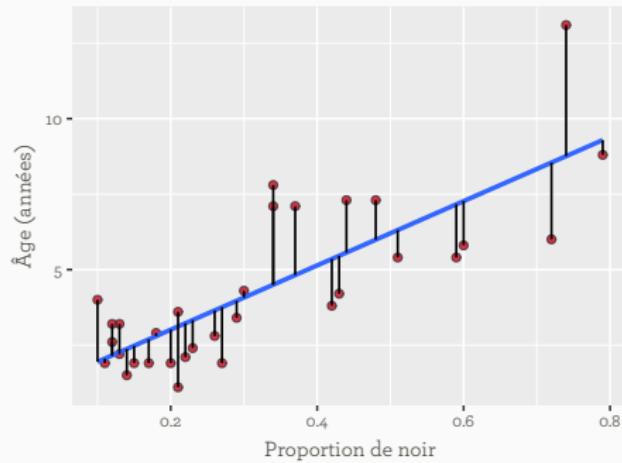
Régression linéaire simple

Vérification des conditions d'application

Important

Les conditions d'application de la régression linéaire se vérifient **APRÈS** la réalisation de la régression.

On vérifie les conditions d'application sur **les résidus** de la régression, et pas sur les **données brutes**.



Régression linéaire simple

Vérification des conditions d'application

On examine 4 graphiques pour vérifier que :

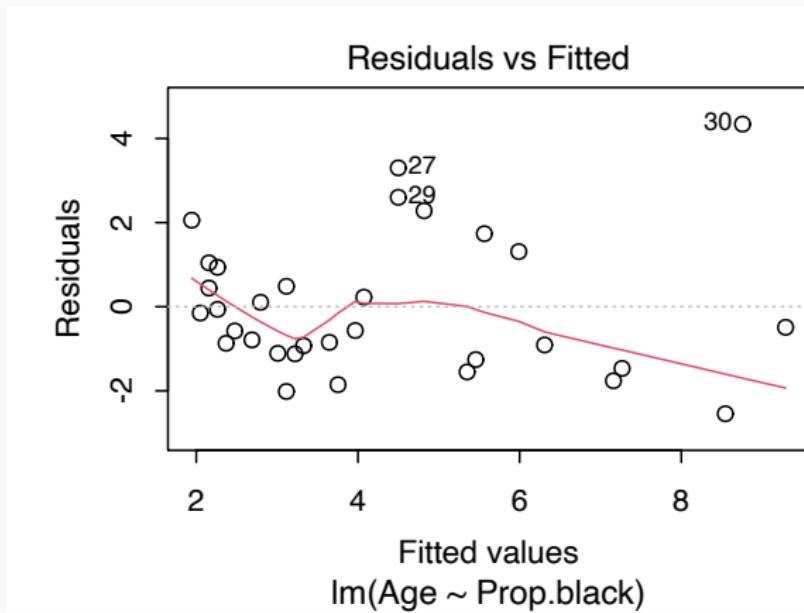
1. Les résidus sont homogènes
2. Les résidus ont une distribution Normale

Pour cela, il suffit de taper :

```
plot(lionReg)
```

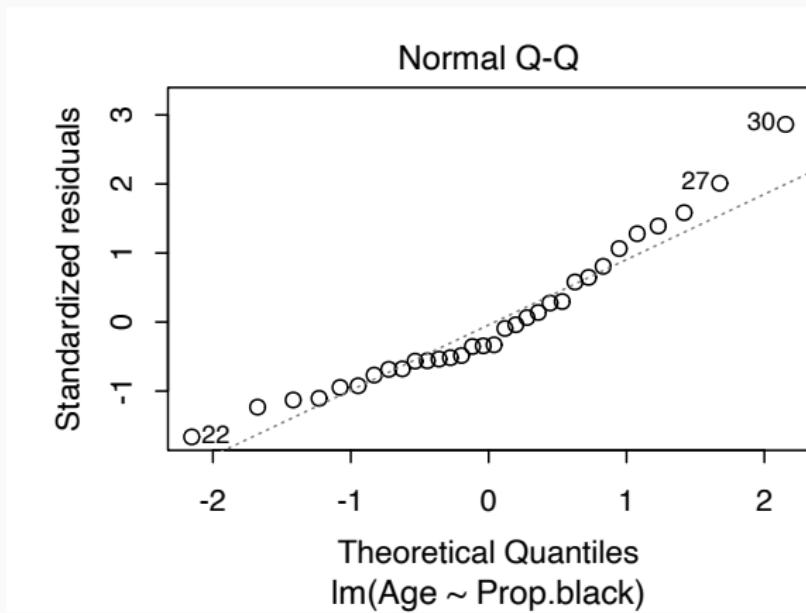
Régression linéaire simple

Vérification des conditions d'application



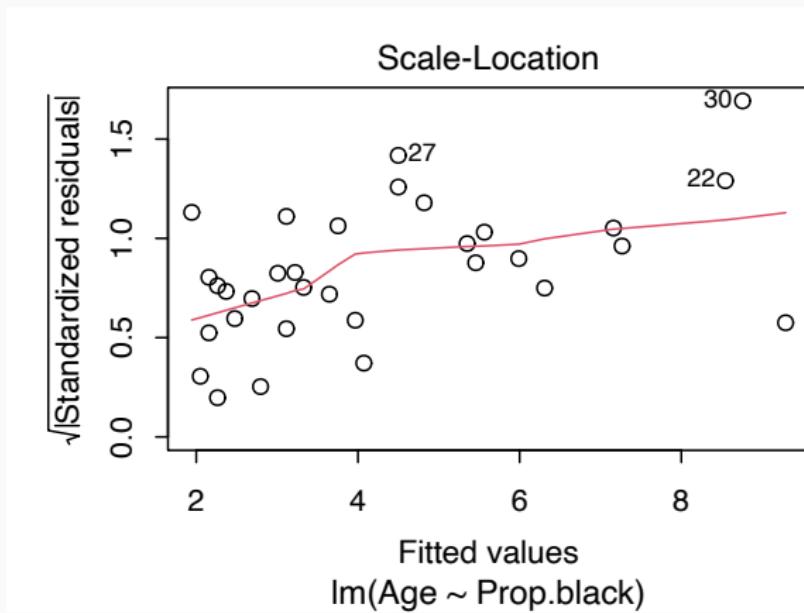
Régression linéaire simple

Vérification des conditions d'application



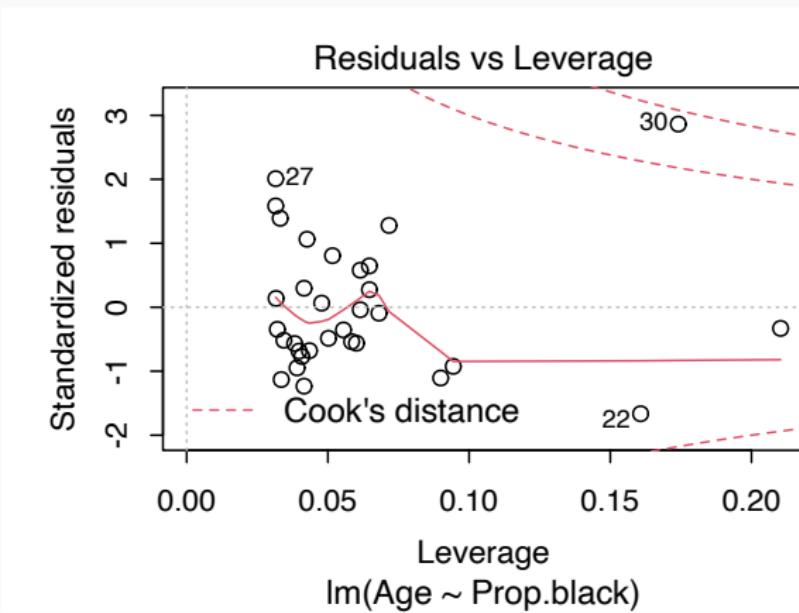
Régression linéaire simple

Vérification des conditions d'application



Régression linéaire simple

Vérification des conditions d'application



L'Analyse de variance

Analyse de variance 1 facteur

Même démarche que pour la régression

- ▶ Une ANOVA, c'est la même chose qu'une régression linéaire.
- ▶ Seule différence : **la variable explicative est un facteur**, c'est une variable catégorielle alors qu'elle était continue dans le cas de la régression linéaire.

Il s'agit d'un test paramétrique dont les conditions d'application se vérifient exactement comme pour la régression.