

Initiation aux logiciels R et RStudio

Pour l'analyse de données et les représentations graphiques

Benoît Simon-Bouhet

Conservatoire Botanique National

9, 10 et 11 décembre 2024

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 <dtm>
```

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 façon :

- ▶ 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 <dtm>
```


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
```

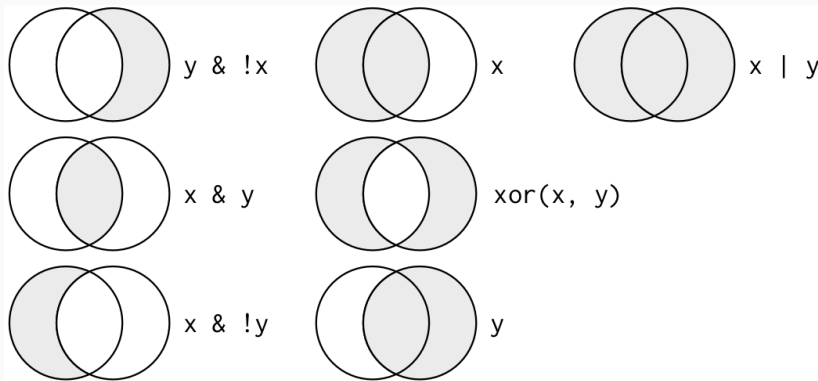
```
1/49 * 49 == 1
```

```
[1] FALSE
```

1. Filtrer des données

`filter()` : opérateurs logiques

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



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 <dtm>
```

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 <dtm>
```

1. Filtrer des données

`filter()` : opérateurs logiques

Loi de Morgan :

- ▶ $!(x \ \& \ y)$ est équivalent à $!x \ | \ !y$
- ▶ $!(x \ | \ y)$ 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 équivalentes¹ :

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

1. 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 l'eur 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 rattrapé 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 <dtm>
```


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 <dtm>
```

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` (incluses) :

```
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` (inclues) :

```
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 <dtm>
```


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 <dtm>
```

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 <dbl> | arr_delay <dbl> | year <int> | month <int> | day <int> | dep_time <int> | sched_dep_time <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 <dtm>
```

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       11    1400     227
2     1     1         4       20    1416     227
3     1     1         2       33    1089     160
4     1     1        -1      -18    1576     183
5     1     1        -6      -25     762     116
6     1     1        -4       12     719     150
7     1     1        -5       19    1065     158
8     1     1        -3      -14     229      53
9     1     1        -3       -8     944     140
10    1     1        -2        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 agrégatives 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 suivant² :

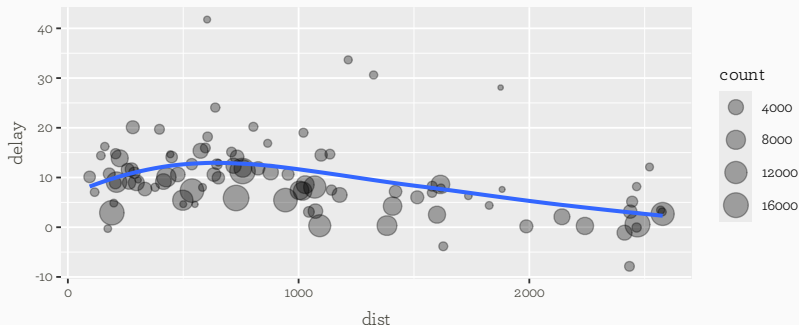
```
by_dest <- group_by(flights, dest)
delay <- summarise(by_dest,
  count = n(),
  dist = mean(distance, na.rm = TRUE),
  delay = mean(arr_delay, na.rm = TRUE)
)
delay_clean <- filter(delay, count > 20, dest != "HNL")
```

2. 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.³ 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))
```

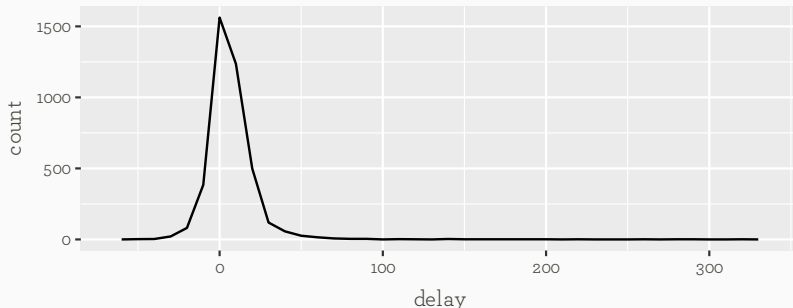
3. Ici, les valeurs manquantes représentent des vols annulés.

5. Résumer des données

`summarise()` : les comptages

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

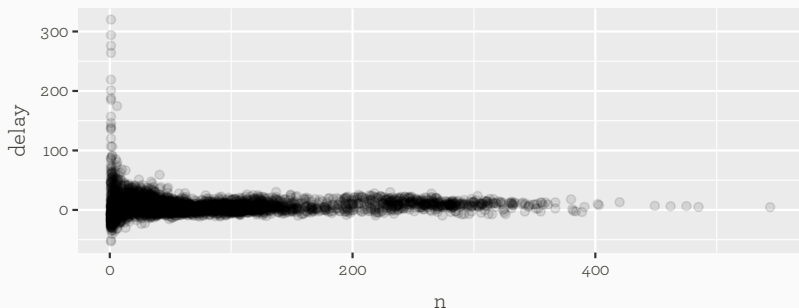
```
delays <- not_cancelled %>%  
  group_by(tailnum) %>%  
  summarise(delay = mean(arr_delay))  
  
ggplot(data = delays, aes(x = delay)) + geom_freqpoly(binwidth = 10)
```



5. Résumer des données

`summarise()` : les comptages

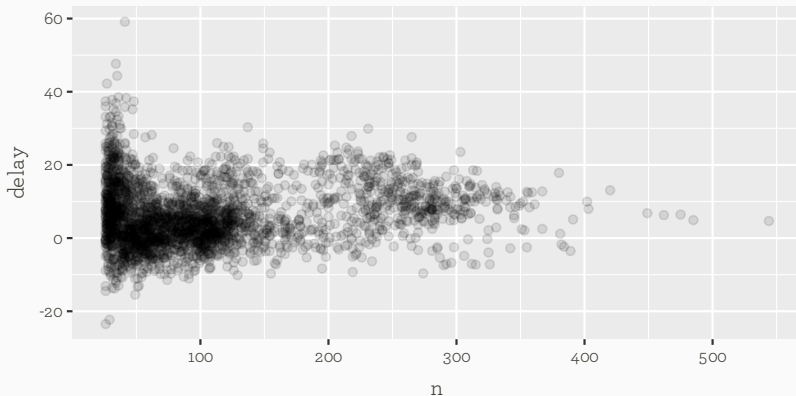
```
delays <- not_cancelled %>%  
  group_by(tailnum) %>%  
  summarise(delay = mean(arr_delay),  
            n = n())  
  
ggplot(data = delays, aes(x = n, y = delay)) +  
  geom_point(alpha = 1/10)
```



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)
```

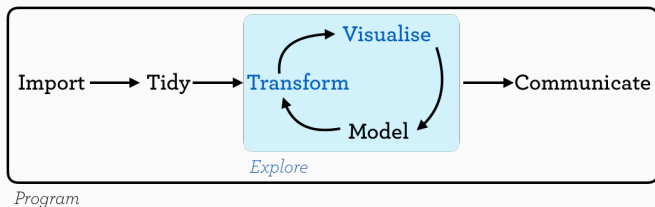
2. Examinez le nombre de vol annulés chaque jour. Y a-t'il une tendance? Est-ce que la proportion de vols annulés est liée au retard moyen?
3. Quelle compagnie aérienne a les retards les plus importants? Pouvez-vous démêler la cause la plus importante de retard entre une mauvaise compagnie et une mauvaise 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 tidyr



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 `tidyr`

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

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](#).