University of Glasgow

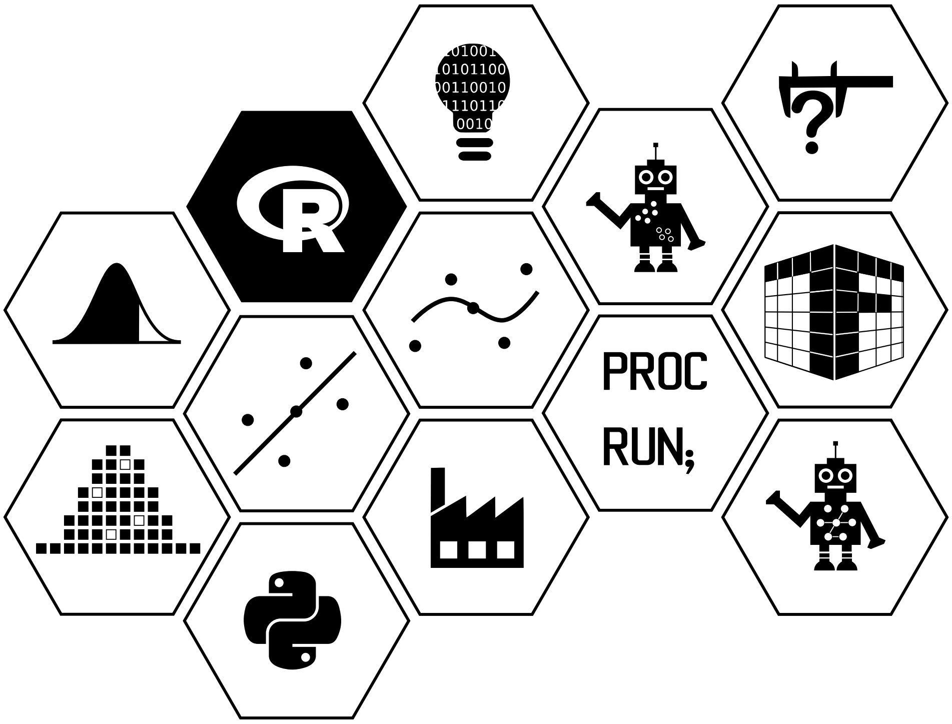
R Programming/ Statistical Computing

Craig Alexander

Academic Year 2023-24

Week 4:

Efficient Data Management using the tidyverse Packages



DATA ANALYTICS GLASGOW

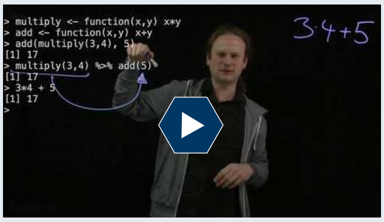
Tidyverse

Tidyverse is a collection of R packages designed to help data scientists to make more efficient use of R. It contains the following packages:

- tibble provides a "modern reimagining" of the good old `data.frame`, which you learned about last week. Tibbles (or `tbl_dfs`) are more flexible in terms of what they can store, but (purposefully) less flexible in terms of "sloppy code": tibbles are stricter about recycling and do not perform partial matching. We will look at `tibble` in more detail this week.
- readr provides alternative functions for reading in text data in tabular form. It provides faster and more consistent alternatives to `read.table` and `read.csv`.
- dplyr provides a powerful suite of functions for data manipulation with a focus on allowing for clean and simple code. We will look at `dplyr` in more detail this week.
- tidyr helps with reshaping data. Information can be organised in many different ways. `tidyr` is designed to make it easy to switch between these formats and has a focus on what its author (Hadley Wickham) believes is "tidy" data.
- ggplot2 is a very featureful and systematic set of plotting functions, which we will focus on in week 6.
- purr provides a more advanced interface for functional programming. We will come to this package in week 8.

Pipelines

Pipelines are at the centre of all the tidyverse packages. The R package magrittr provides a forward-pipe operator for R. If you are wondering about the package name: it is named after the Belgian surrealist artist René Magritte and his painting *La trahison des images*, which shows a pipe together with the text "Ceci n'est pas une pipe" (which is French for "This is not a pipe").



```
> multiply <- function(x,y) x*y
> add <- function(x,y) x+y
> add(multiply(3,4), 5)
[1] 17
> multiply(3,4) %>% add(5)
[1] 17
> 3*4 + 5
[1] 17
>
```

Pipelines

https://youtu.be/9UtN2mH52EM

Duration: 12m50s

Suppose we have a function `f` defined in R

```
f <- function(x)
  x^2
```

Then we can apply `f` to an argument `x` using

```
x <- 3
f(x)
```

```
## [1] 9
```

The forward-pipes from magrittr allow us to rewrite this function call as

```
library(magrittr)
x %>% f
```

```
## [1] 9
```

instead. The advantage of this alternative notation might not become immediately clear, but its advantage becomes more obvious when looking at nested function calls.

Consider the R data set `mtcars`, which contains data from the 1974 edition from the US magazine Motor Trend. Suppose we want to convert the fuel consumption to litres per 100 kilometres and then only retain the cars with a fuel economy better than 10 litres per 100 kilometres.

```
mtcars2 <- transform(mtcars, lp100k=235.21/mpg)
subset(mtcars2, lp100k<=10)
```

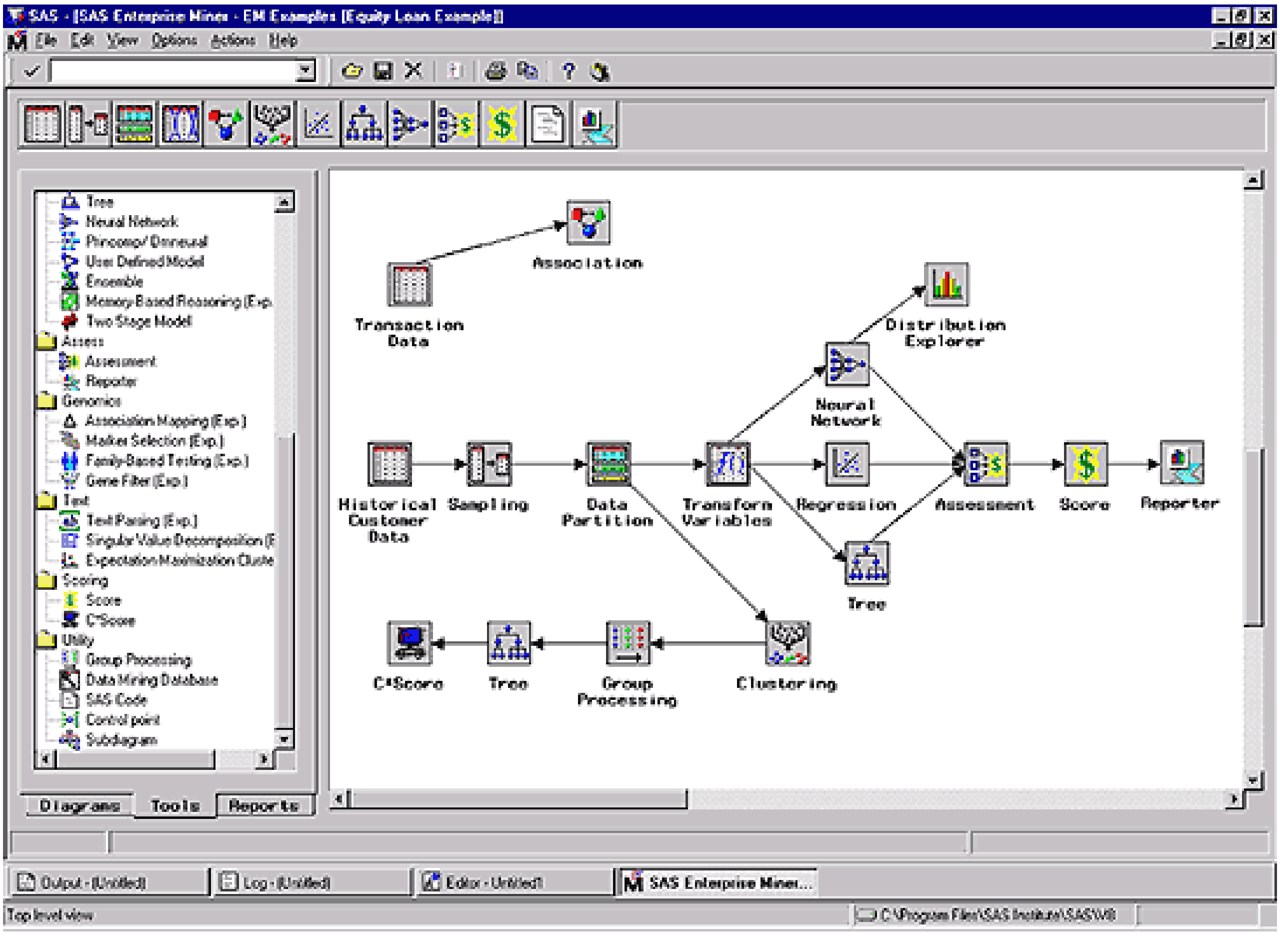


Figure 1: Screenshot from SAS Enterprise Miner

```
##                 mpg cyl  disp  hp drat    wt  qsec vs am gear carb   lp100k
## Merc 240D      24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2 9.639754
## Fiat 128       32.4   4  78.7  66 4.08 2.200 19.47  1  1    4    1 7.259568
## Honda Civic    30.4   4  75.7  52 4.93 1.615 18.52  1  1    4    2 7.737171
## Toyota Corolla 33.9   4  71.1  65 4.22 1.835 19.90  1  1    4    1 6.938348
## Fiat X1-9      27.3   4  79.0  66 4.08 1.935 18.90  1  1    4    1 8.615751
## Porsche 914-2  26.0   4 120.3  91 4.43 2.140 16.70  0  1    5    2 9.046538
## Lotus Europa   30.4   4  95.1 113 3.77 1.513 16.90  1  1    5    2 7.737171
```

(If you are wondering where the number of 235.21 comes from: A US gallon is roughly 3.785 litres and a mile is roughly 1.609 kilometres, and $\frac{100\times3.785}{1.609} \approx 235.21$)

If we want to perform both steps in one go, we can nest the two calls within one another and use

```
subset(transform(mtcars, lp100k=235.21/mpg), lp100k<=10)
```

This gives exactly the same results, but is not very easy to read and understand. It is not easy to see that the argument `lp100k<=10` belongs to `subset`. When nesting function calls, the arguments get increasingly far from the function call to which they belong.

The %>% operator however allows us to write this much more cleanly:

```
mtcars %>%
  transform(lp100k=235.21/mpg) %>%
  subset(lp100k<=10)
```

GUI-based software for data science like the SAS Enterprise Miner or Alteryx are based on visual representations of workflows like the one shown in Figure 1. Pipelines allow for arranging your code in a similar way.

Task 1.

The R code below uses pipelines. Convert it to nested function calls.

```
rnorm(1000) %>% sin() %>% max()
```

Task 2.

Convert the R code below to pipelines.

```
library(MASS)                       # Load package MASS, which contains the data
subset(transform(mammals, ratio=brain/body), ratio==max(ratio))
```

Additional pipeline operators

The package `magrittr` defines additional pipeline operators. We will look at two of them: %\$% and %<>%.

%\$% makes the variables in the data set on the left-hand side visible as variables in the expression on the right-hand side.

In the above example, suppose that after having subset the data we would like calculate correlation between `disp` (engine displacement) and `hp` (gross horsepower). We could store the result in a temporary data frame and then calculate the correlation.

```
mtcars2 <- mtcars %>%
               transform(lp100k=235.21/mpg) %>%
               subset(lp100k<=10)
cor(mtcars2$disp, mtcars2$hp)
```

```
## [1] 0.2003641
```

With %$% we can write this as

```
library(magrittr)
mtcars %>%
    transform(lp100k=235.21/mpg) %>%
    subset(lp100k<=10) %$%
    cor(disp, hp)
```

```
## [1] 0.2003641
```

Another useful (but potentially dangerous) operator in `magrittr` is %<>%. When we want to make changes to a data set, we sometimes do want to replace the data frame at hand, rather than storing the result in a new data frame. We would for example use code like

```
mtcars <- mtcars %>%
             transform(lp100k=235.21/mpg) %>%
             subset(lp100k<=10)
```

The %<>% operator from magrittr lets us write this more compactly as

```
mtcars %<>% transform(lp100k=235.21/mpg) %>%
               subset(lp100k<=10)
```

In other words, the %<>% operator "pipes" the left-hand side into the right-hand side, just like %>%, but it then also stores the result of the right-hand side in the variable given on the left-hand side.

Pipelines for Data Analysis in R

https://speakerdeck.com/hadley/pipelines-for-data-analysis-in-r

Hadley Wickham has produced a series of excellent slides about pipelines, which covers much of what we will look at this week.

Background reading: Chapter 18 of R for Data Science

http://r4ds.had.co.nz/pipes.html

Chapter 18 of *R for Data Science* gives a detailed overview of pipes and some of the underpinning technology (though the latter is rather advanced).

Tibbles

The package tibble provides `tbl_df`'s (or "tibbles", which is easier to pronounce). They are a modern take on the built-in class `data.frame`.

One key advantage of tibbles is that they can store anything. A `data.frame` can only store a single value per "cell", for example a number or a character string. However, in a tibble, you can store a list or even another tibble in a cell. An example of this is the tibble `starwars` from the package `dplyr`. The column `starships` contains for each row the list of starships flown by that character (which is a list of different length depending on the character.)

```
library(dplyr)                            # Load library dplyr which contains the data
starwars[,c("name", "starships")]         # Print columns name and starships
```

```
## # A tibble: 87 x 2
##    name               starships
##    <chr>              <list>
##  1 Luke Skywalker     <chr [2]>
##  2 C-3PO              <chr [0]>
##  3 R2-D2              <chr [0]>
##  4 Darth Vader        <chr [1]>
##  5 Leia Organa        <chr [0]>
##  6 Owen Lars          <chr [0]>
##  7 Beru Whitesun lars <chr [0]>
##  8 R5-D4              <chr [0]>
##  9 Biggs Darklighter  <chr [1]>
## 10 Obi-Wan Kenobi     <chr [5]>
## # ... with 77 more rows
```

```
starwars[10,"starships"][[1]]             # Starships flown by Obi-Wan
```

```
## [[1]]
## [1] "Jedi starfighter"         "Trade Federation cruiser" "Naboo star skiff"
## [4] "Jedi Interceptor"         "Belbullab-22 starfighter"
```

We could not have stored this information in a data frame. We would have had to either store the information across several data frames or stored the list of starships as a character string.

Creating tibbles

Coercion A data frame or matrix can be converted to a tibble using the function `as_tibble`. Let's start with the data frame kids from week 3:

```
kids
```

```
##       age weight height gender
## Sarah   4     15    101      f
## John   11     28    132      m
```

```
library(tibble)
kidstibble <- as_tibble(kids)
kidstibble
```

```
## # A tibble: 2 x 4
##     age weight height gender
##   <dbl>  <dbl>  <dbl> <chr>
## 1     4     15    101 f
## 2    11     28    132 m
```

As you can see from the output, tibbles do not use row names (though they store them so that the row names can be added back when the tibble is converted back to a data frame). Thus in this case it would be best to add a column called `name`: we will first add the column and then re-arrange the columns that the names come first.

```
kidstibble$name <- rownames(kids)      # Create a column with names
kidstibble <- kidstibble[,c("name", "age", "weight", "height", "gender")]
                                       # Re-arrange columns that names comes first
```

There also is a function `rownames_to_column` which we could have also used for this purpose. It adds the rownames as first column to the tibble.

```
kidstibble2 <- rownames_to_column(kids)
kidstibble2
```

```
##   rowname age weight height gender
## 1   Sarah   4     15    101      f
## 2    John  11     28    132      m
```

We can work with tibbles in pretty much the same way as with data frames, though not all R functions work with tibbles yet. In this case, you can convert a tibble into a data frame using `as.data.frame`.

```
kidsdf <- as.data.frame(kidstibble)
kidsdf
```

```
##    name age weight height gender
## 1 Sarah   4     15    101      f
## 2  John  11     28    132      m
```

Creation We can create tibbles using the function `tibble`. We can create the tibble from above using

```
kidstibble <- tibble(name=c("Sarah", "John"), age=c(4,11), weight=c(15,28),
                     height=c(101,132), gender=c("f", "m"))
```

In other words, the function `tibble` assembles a tibble on a column-by-column basis (akin to using `cbind`).

The function `tribble` ("transposed tibble") lets you create a tibble on a row-by-bow basis (akin to using `rbind`), which is typically more legible when creating a matrix in code.

```
kidstibble <- tribble(~name,   ~age, ~weight, ~height, ~gender,
                      "Sarah",    4,      15,     101,     "f",
                      "John",    11,      28,     132,     "m")
```

Task 3.

Create a tibble called `courses` containing the data shown below ...

- by converting it from a data frame using `as_tibble`,
- by creating it using `tibble`, and
- by creating it using `tribble`.

```
## # A tibble: 3 x 3
##   course taught_by          weeks
##   <chr>  <chr>              <dbl>
## 1 psm    Alexey                11
## 2 psf    Eilidh and Colette    11
## 3 rp     Craig                 11
```

By and large tibbles work like data frames

In most circumstances, you can work with the tibbles in the same way as you would work with data frames, though there are important differences:

- Variables/Columns can be accessed and added using `tibble$varname` (`varname` needs to be fully spelled out). You can also access a column using `tibble[,"varname"]` or `tibble[["varname"]]`.
- Rows can be selected using `tibble[rowindices,]` (note that you cannot use row names).
- Individual cells can be accessed using `tibble[rowindices, colindices]`.

Task 4.

Add a column called "coursework_perc" taking the values 30, `NA` and 100 to the tibble `courses` from task 3. Then print the first row and then print the first and second column.

Lazy and surly

Tibbles are stricter In some way, R's built-in data frames are designed to make interactive data analysis more convenient. Partial matching of column names and R's extensive recycling rules mean that (if you know what you are doing) you can get away with less typing. For example instead of typing `kids\$weight` you just need to type `kids\$w`.

However, there is a price to pay for this convenience: when used in complex programmes, these "convenience features" can sometimes conceal coding mistakes and make them much harder to track down.

For example, a programme using `kids\$w` instead of `kids\$weight` will stop working once you have added another column starting with the letter `w` (say `wakeup_time`). To make things worse, `kids\$w` will then simply return `NULL` (and not produce a warning or error message), so your code might not fail immediately, but simply return wrong results. You might spend hours (or even days or weeks) tracking down that using `kids\$w` was the culprit.

R's recycling rules are convenient, but can easily conceal semantic coding mistakes when say adding a column of the wrong length that happens to be a factor of the number of rows of the data frame (in which case R will recycle it and not complain, which is in most cases *not* what the user had intended).

For these reasons tibbles are a lot more restrictive, forcing you to write cleaner and more expressive (but also longer) code. Or, in the words of their designers, tibbles are "lazy and surly": they do less and complain more, which, at least in their eyes, are both good things, because it forces you to confront problems earlier. The main differences to data frames are …

- Column names of tibbles have to be fully spelled out, there is no partial matching of column names: `kidstibble\$w` for example will not work, you have to spell out `kidstibble\$weight` . Furthermore, tibbles produce a warning message when accessing a column that does not exist, where as data frames just return `NULL` without making any further noises.
- Tibbles do not recycle arguments, unless they are of length 1 (in which case it is pretty clear that the user wants the argument to be recycled).
- Tibbles also do not automatically convert strings to factors, which data frames in R do.

Subsetting tibbles always results in a tibble Tibbles are also more consistent. Subsetting tibbles always results in a tibble.

```
kidstibble[,1]                          # Result is a tibble
```

```
## # A tibble: 2 x 1
##   name
##   <chr>
## 1 Sarah
## 2 John
```

In contrast, subsetting a data frame or matrix is not guaranteed to result in a data frame or matrix (unless you use `drop=FALSE`). If the result is a single column or row, subsetting a data frame or matrix results in a vector.

```
kids[,1]                                # Result is "dropped" to a vector
```

```
## [1]  4 11
```

This "dropping" of the dimension can be very useful when using R interactively, but can be the source of many issues in more complex projects, when programmers incorrectly assume that subsetting a data frame or matrix will always result in another data frame or matrix, rather than possibly just a vector (it is thus a good idea to always use `drop=FALSE` when working with data frames or matrices in complex projects).

Data Import Cheat Sheet

https://github.com/rstudio/cheatsheets/blob/main/data-import.pdf

RStudio's cheat sheet for data import also covers tibbles.

Background reading: Chapter 10 of *R for Data Science*

http://r4ds.had.co.nz/tibbles.html

Chapter 10 of *R for Data Science* gives a detailed overview of tibbles.

Reading in data using readr

The package readr contains alternatives to the functions `read.table` and `read.csv`. The alternative functions from `readr` have four main advantages.

- They read in the data a lot faster and can show a progress bar (though this is only relevant for really big data sets).
- They store the data straight in a tibble, rather than a data frame.
- They allow specifying the intended data type for each column and thus make it easier to identify rows which cause problems.
- They are less intrusive: they don't automatically convert character strings to factors and do not change column names (`read.table` and `read.csv` will for example remove spaces from variable names and replace them by full stops). The functions from `readr` are also guaranteed to give the same result irrespective of the platform or operating system they are run under.

`readr` provides the following functions.

- `read_csv` reads in comma-separated files. `read_csv2` reads in files which are semicolon-separated (common in countries like France or Germany, where a comma is used as decimal separator).
- `read_tsv` reads in tab-separated files.
- `read_delim` is the most general function (like `read.table`). The delimiter has to be specified using the argument `delim`.
- `read_fwd` reads in fixed-width files.

All functions assume that the first row contains the column/variable names. If this is not the case, set the optional argument `col_names` to `FALSE` or to a character vector containing the intended column names.

The strings used to encode missing values can be specified using the optional argument `na`.

For example, we can read in the file chol.txt from week 3 using

```
library(readr)
read_delim("chol.txt", delim=" ", col_names=c("ldl", "hdl", "trig",
                                              "age", "gender", "smoke"))
```

```
## # A tibble: 13 x 6
##      ldl   hdl  trig   age gender smoke
##    <dbl> <dbl> <dbl> <dbl> <chr>  <chr>
##  1   175    25   148    39 female no
##  2   196    36    92    32 female no
##  3   139    65    NA    42 male   <NA>
##  4   162    37   139    30 female ex-smoker
##  5   140   117    59    42 female ex-smoker
##  6   147    51   126    65 female ex-smoker
##  7    82    81    NA    57 male   no
##  8   165    63   120    48 male   current
##  9   149    49    NA    32 female no
## 10    95    54   157    55 female ex-smoker
## 11   169    59    67    48 female no
## 12   174   117   168    41 female no
## 13    91    52   146    69 female current
```

Note that functions from `readr` show the data type it has used for each column. This makes it easier to spot mistakes like missing values not coded as expected, in which case a numeric column would show up as a character string.

For example, we can read in the file chol.csv from week 3 using

```
library(readr)
read_csv("chol.csv", na=".")
```

```
## Rows: 13 Columns: 6
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## chr (2): gender, smoke
## dbl (4): ldl, hdl, trig, age
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
## # A tibble: 13 x 6
##      ldl   hdl  trig   age gender smoke
##    <dbl> <dbl> <dbl> <dbl> <chr>  <chr>
##  1   175    25   148    39 female no
##  2   196    36    92    32 female no
##  3   139    65    NA    42 male   NA
##  4   162    37   139    30 female ex-smoker
##  5   140   117    59    42 female ex-smoker
##  6   147    51   126    65 female ex-smoker
##  7    82    81    NA    57 male   no
##  8   165    63   120    48 male   current
##  9   149    49    NA    32 female no
## 10    95    54   157    55 female ex-smoker
## 11   169    59    67    48 female no
## 12   174   117   168    41 female no
## 13    91    52   146    69 female current
```

Task 5.

Read the data files cars.csv and ships.txt from week 3 into R using the functions from `readr`.

Supplementary material:
Specifying column types

The functions from `readr` allow specifying the expected column types. This is especially important when writing which will then be run automatically. It provides an easy way of ensuring that the data provided is of the expected format.

The easiest way of specifying expected column types is to provide a character string with each letters standing for a column

Letter	Meaning
`c`	character
`i`	integer
`n`	number
`d`	double
`l`	logical
`D`	date
`T`	date time
`t`	time
`?`	guess the type
`_` or –	skip the column

So for the data file chol.csv we would expect the first four columns to be integers and the latter two to be character strings, so we would use

```
chol <- read_csv("chol.csv", na=".", col_types="iiiicc")
```

Specifying the expected column types can help pinpointing problems when reading in data. Suppose we had forgotten that missing values are coded using "`.`" in this data file. If we use …

```
chol <- read_csv("chol.csv")
```

```
## Rows: 13 Columns: 6
## -- Column specification --------------------------------------------------------------------
## Delimiter: ","
## chr (3): trig, gender, smoke
## dbl (3): ldl, hdl, age
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

… we can see from the output that `trig` was read in as a character string, but we do not know why.

However, if we use …

```
chol <- read_csv("chol.csv", col_types="iiiicc")
```

```
## Warning: One or more parsing issues, see `problems()` for details
```

… we obtain a warning and can print the problematic rows using

```
problems(chol)
```

```
## # A tibble: 3 x 5
##     row   col expected   actual file
##   <int> <int> <chr>      <chr>  <chr>
## 1     4     3 an integer .      /Users/Craig/Documents/GitHub/BOLDrprog/coursematerial/week4~
## 2     8     3 an integer .      /Users/Craig/Documents/GitHub/BOLDrprog/coursematerial/week4~
## 3    10     3 an integer .      /Users/Craig/Documents/GitHub/BOLDrprog/coursematerial/week4~
```

The output from `problems` shows us that for three rows (3, 7 and 9) the data in `chol.csv` was not of the expected format: a value of `.` is not compatible with the column being numeric. This makes it easy to identify the cause of the problem (NAs coded as "`.`") and rectify the issue.

Efficient data manipulation using dplyr

```
>
> bikes
# A tibble: 67,354 x 4
                         name available_bikes available_bike_stands  time
                        <chr>           <int>                 <int> <dbl>
 1        CHAMPEAUX (BAGNOLET)               9                    41    13
 2      POISSONNIÈRE - ENGHIEN              33                     0    13
 3                  METRO ROME               6                    38    13
 4          DE GAULLE (PANTIN)               2                    16    13
 5 PARC DE BELLEVILLE (20040)                4                    22    13
 6      SOLJENITSYNE (PUTEAUX)                                     4    13
 7                      SERRES                                    18    13
 8         PYRAMIDE ARTILLERIE                                    40    13
 9               SAINT GEORGES                                    10    13
10               MUSÉE D'ORSAY                                     0    13
# ... with 67,344 more rows
> bikes %>% group_by(time)
```

Efficient data manipulation using dplyr

https://youtu.be/uOo4s_s15aI

Duration: 14m26s

In this section we will work with data from Paris' Vélib' bicycle sharing system available through JCDecaux's API for open cycle data.

The data consists of the number of bikes available and the number of bike stands available at every Vélib' station, recorded every five minutes over six hours on a Tuesday afternoon in October 2017.

The data consists of two tibbles. The first, `bikes` contains data on the number of available bikes and stands at each station.

Variable	Description
`name`	Name of the station
`available_bikes`	Number of available at that time
`available_bike_stands`	Number of available bike stands
`time`	Decimal time for which the number have been recorded

The second, `stations` contains additional information about each station.

Variable	Description
`name`	Unique name of the station
`id`	Internal ID number of the station
`address`	Address of where the station is located
`lng`	GPS coordinate (longitude)
`lat`	GPS coordinate (latitude)
`departement`	Département in which the station is located

You can load the data into R using

```
library(tibble)
load(url("https://github.com/UofGAnalyticsData/R/raw/main/Week%204/velib"))
```

Overview: the key functions ("verbs") for `dplyr`

Function ("verb")	Description	R base equivalent(s)	SQL equivalent
`filter`	Select observations/rows	`subset`	`WHERE ...` (or `HAVING ...`)
`slice`	Select observations by row numbers	`[idx,]`	(vendor dependent)
`select`	Select variables/column	`$` or `[,sel]`	
`mutate`	Create new variables/column	`transform`	`SELECT ... AS ...`
`arrange`	Sort observations/rows	`order`	`ORDER BY ...`
`group_by`	Group observations by variable	`by` or `aggregate`	`GROUP BY ...`
`summarise`	Calculate summary statistics	`by` or `aggregate`	`SELECT ..` and `GROUP BY`

The functions in `dplyr` are designed to be used with tibbles, but they also work with data frames. When invoked with a data frame, they will return a data frame as long as this is possible.

Selecting observations (rows) using `filter` and `slice`

`filter` The function `filter` is used to select observations (or rows) in a similar way to the base R function `subset`.

Suppose we want to print all bike stations in Paris (rather than other départements from Île de France)

```
library(dplyr)
stations75 <- stations %>%
        filter(departement=="Paris")
stations75
```

```
## # A tibble: 743 x 6
##    name                                         id address                  lng   lat depar~1
##    <chr>                                     <dbl> <chr>                  <dbl> <dbl> <chr>
##  1 PORT SOLFERINO (STATION MOBILE)             901 BERGES DE SEINE, SOUS~  2.32  48.9 Paris
##  2 QUAI MAURIAC  / PONT DE BERCY               903 FETE DE L'OH (BERCY) ~  2.37  48.8 Paris
##  3 17/19 PLACE JOFFRE / ECOLE MILITAIRE        904 ECOLE MILITAIRE-AVENU~  2.30  48.9 Paris
##  4 CONCORDE/BERGES DE SEINE (STATION MOBILE)   905 BERGES DE SEINE, BAS ~  2.32  48.9 Paris
##  5 PORT DU GROS CAILLOU (STATION MOBILE)       908 BERGES DE SEINE, ESCA~  2.31  48.9 Paris
##  6 PONT D'ARCOLE (STATION MOBILE)              909 Voie Georges Pompidou~  2.35  48.9 Paris
##  7 ILE DE LA CITE PONT NEUF                   1001 41 QUAI DE L'HORLOGE ~  2.34  48.9 Paris
##  8 PLACE DU CHATELET                          1002 14 AVENUE VICTORIA - ~  2.35  48.9 Paris
##  9 RIVOLI SAINT DENIS                         1003 7 RUE SAINT DENIS - 7~  2.35  48.9 Paris
## 10 MARGUERITE DE NAVARRE                      1004 12 RUE DES HALLES - 7~  2.35  48.9 Paris
## # ... with 733 more rows, and abbreviated variable name 1: departement
```

Note the use of a double == to test whether the département is equal to "Paris".

We can create more complex expressions using the standard logical operators & ("and"), | ("or") and ! ("not"). Note that you *cannot* use && and || in this context, as they only work with scalar arguments.

For example, if we want to extract the stations which are in Paris or Hauts-de-Seine we can use

```
stations7592 <- stations %>%
         filter(departement=="Paris" | departement=="Hauts-de-Seine")
```

Rather than using a logical or we could have used `%in%`:

```
stations7592 <- stations %>%
         filter(departement %in% c("Paris" , "Hauts-de-Seine"))
```

Even though the functions from `dplyr` are designed to be used with pipelines, you can also provide the data set as first argument:

```
stations7592 <- filter(stations, departement %in% c("Paris" , "Hauts-de-Seine"))
```

`slice` You can use the function `slice` to select observations based on their row numbers.

```
stations %>%
 slice(5:7)
```

```
## # A tibble: 3 x 6
##   name                                     id address                       lng   lat depar~1
##   <chr>                                 <dbl> <chr>                       <dbl> <dbl> <chr>
## 1 PORT DU GROS CAILLOU (STATION MOBILE)   908 BERGES DE SEINE, ESCALIER ~  2.31  48.9 Paris
## 2 PONT D'ARCOLE (STATION MOBILE)          909 Voie Georges Pompidou - 75~  2.35  48.9 Paris
## 3 ILE DE LA CITE PONT NEUF               1001 41 QUAI DE L'HORLOGE - 750~  2.34  48.9 Paris
## # ... with abbreviated variable name 1: departement
```

selects the observations in rows 5 to 7 and is equivalent to

```
stations[5:7,]
```

```
## # A tibble: 3 x 6
##   name                                     id address                       lng   lat depar~1
##   <chr>                                 <dbl> <chr>                       <dbl> <dbl> <chr>
## 1 PORT DU GROS CAILLOU (STATION MOBILE)   908 BERGES DE SEINE, ESCALIER ~  2.31  48.9 Paris
## 2 PONT D'ARCOLE (STATION MOBILE)          909 Voie Georges Pompidou - 75~  2.35  48.9 Paris
## 3 ILE DE LA CITE PONT NEUF               1001 41 QUAI DE L'HORLOGE - 750~  2.34  48.9 Paris
## # ... with abbreviated variable name 1: departement
```

Task 6.

Identify the stations which had more than 60 bikes available at 3pm (i.e. `time` taking the value 15).

Selecting variables (columns) using `select`

The function `select` can be used to subset the variables (columns) of a data set.

You can either specify the columns to retain or (with a minus) those you do not want to retain.

We can only retain the name and département of each station using either

```
stations.small <- stations %>%
                     select(name, departement)
stations.small
```

```
## # A tibble: 928 x 2
##    name                                     departement
##    <chr>                                    <chr>
##  1 PORT SOLFERINO (STATION MOBILE)          Paris
##  2 QUAI MAURIAC  / PONT DE BERCY            Paris
##  3 17/19 PLACE JOFFRE / ECOLE MILITAIRE     Paris
##  4 CONCORDE/BERGES DE SEINE (STATION MOBILE) Paris
##  5 PORT DU GROS CAILLOU (STATION MOBILE)    Paris
##  6 PONT D'ARCOLE (STATION MOBILE)           Paris
##  7 ILE DE LA CITE PONT NEUF                 Paris
##  8 PLACE DU CHATELET                        Paris
##  9 RIVOLI SAINT DENIS                       Paris
## 10 MARGUERITE DE NAVARRE                    Paris
## # ... with 918 more rows
```

or

```
stations.small <- stations %>% select(-id, -address, -lng, -lat)
```

You can also use `select` to change the order of the columns of a data set.

Adding new variables using `mutate`

The function `mutate` can be used to create new variables (columns) in a data set. `mutate` is similar in functionality to the base R function `transform`.

We can add the total number of stands to the data set `bikes` using

```
bikes <- bikes %>%
           mutate(total_stands = available_bikes+available_bike_stands)
```

More than one new variable can be defined by adding further arguments to `mutate`.

`transmute` is a sibling of `mutate`. Just like `mutate` it creates new columns. It however also removes all existing columns so that only the new columns remain.

Task 7.

The time is currently encoded as decimal (e.g. `13.5` for 13:30). Create two columns `time_hours`, which contains the hour (13 in our example), and `time_minutes`, which contains the minutes, (30 in our example).

You can calculate `time_hours` as the floor of `time` (R function `floor`) and `time_minutes` as the remainder after integer division of 60 times `time` by 60 (R operator `%%`).

Sorting data sets using `arrange`

The function `arrange` can be used to sort a data set by one or more variables. We can sort the data set `bikes` by the number of available bikes suing

```
bikes %>%
  arrange(available_bikes)
```

```
## # A tibble: 67,354 x 5
##    name                          available_bikes available_bike_stands  time total_stands
##    <chr>                                   <int>                 <int> <dbl>        <int>
##  1 KARMAN (AUBERVILLIERS)                      0                     0    13            0
##  2 PIGALLE GERMAIN PILLON                      0                    20    13           20
##  3 ROND POINT DES CHAMPS ELYSEES               0                     0    13            0
##  4 MONTCALM                                    0                    47    13           47
##  5 PLACE HENOCQUE VERSION 2                    0                    34    13           34
##  6 PLACE DES FETES                             0                    19    13           19
##  7 MANIN SECRETAN                              0                    20    13           20
##  8 MARTINIE (VANVES)                           0                    24    13           24
##  9 HORTENSIAS (LES LILAS)                      0                    22    13           22
## 10 HAIES REUNION                               0                    22    13           22
## # ... with 67,344 more rows
```

You can use the function `desc` to sort in descending order

```
bikes %>%
  arrange(desc(available_bikes))
```

```
## # A tibble: 67,354 x 5
##    name    available_bikes available_bike_stands  time total_stands
##    <chr>             <int>                 <int> <dbl>        <int>
##  1 DUPLEIX              68                     0  16.2           68
##  2 DUPLEIX              68                     0  16.2           68
##  3 DUPLEIX              67                     1  15.4           68
##  4 DUPLEIX              67                     1  15.5           68
##  5 DUPLEIX              67                     1  15.8           68
##  6 DUPLEIX              67                     1  16.1           68
##  7 DUPLEIX              67                     1  16.3           68
##  8 SAHEL                67                     0  17.6           67
##  9 SAHEL                67                     0  18             67
## 10 SAHEL                67                     0  18.1           67
## # ... with 67,344 more rows
```

Task 8.

Identity the three bike stations that are furthest to the West (i.e. the ones with the smallest longitude `lng`).

Grouping data and calculating group-wise summary statistics: `group_by` and `summarise`

Suppose we want to identify the busiest stations in the system in the sense of having, on average, the most bikes taken out (and thus the highest number of available bike stands – this is assuming JCDecaux replenish all bike stations in the same way, which is not quite what is happening in reality; there are better, but more complex, ways of defining "busy").

To calculate the average number of available bike stands per station we need to first group the data by bike station and then compute the average number of bike stands available

```
bikes %>% group_by(name) %>%                             # Group by station name
  summarise(avg_stands=mean(available_bike_stands)) %>%  # Calculate averages
  arrange(desc(avg_stands))                              # Sort in descending order
```

```
## # A tibble: 928 x 2
##    name                               avg_stands
##    <chr>                                   <dbl>
##  1 PANTIN                                   70.3
##  2 BELLEVILLE (20041)                       65.1
##  3 PLACE ADOLPHE CHERIOUX                   60
##  4 HIPPODROME D AUTEUIL                     60.0
##  5 RUE DES BOULETS ( COMPLEMENTAIRE )       55
##  6 PLACE DE LA PORTE DE CHATILLON           54.9
##  7 PORTE DE LA CHAPELLE                     54.1
##  8 CHARMES (FONTENAY SOUS BOIS)             53.5
##  9 PORTE DE MONTROUGE                       53
## 10 ALLENDE (PANTIN)                         52.9
## # ... with 918 more rows
```

Task 9.

Can you think of another way of defining busy? Amend the commands accordingly.

Task 10.

Find the number of bike stations in each département.

You might find the function `n()` helpful, which returns the number of cases and is the `dplyr` equivalent of `COUNT(*)` in SQL (type `?n` to get help).

`group_by` can be also used to limit the scope of subsequent calls to other functions such as `filter`, `arrange` or `slice`. To make this more concrete, suppose we want to find for each time point the station which the most available bikes. We first have group the data by `time` and then find the station with the most available bikes.

```
bikes %>%
  group_by(time) %>%                           # Group by time
  arrange(desc(available_bikes)) %>%           # Sort by bikes within each group
  slice (1)                                    # Return only top one per group
```

```
## # A tibble: 73 x 5
## # Groups:   time [73]
##    name          available_bikes available_bike_stands  time total_stands
##    <chr>                   <int>                 <int> <dbl>        <int>
##  1 MUSÉE D'ORSAY              65                     0  13             65
##  2 MUSÉE D'ORSAY              65                     0  13.1           65
##  3 MUSÉE D'ORSAY              65                     0  13.2           65
##  4 MUSÉE D'ORSAY              62                     3  13.2           65
##  5 METZ                       64                     0  13.3           64
##  6 DUPLEIX                    64                     4  13.4           68
##  7 METZ                       64                     0  13.5           64
##  8 MUSÉE D'ORSAY              63                     2  13.6           65
##  9 SAINT EMILION              63                     3  13.7           66
## 10 MUSÉE D'ORSAY              65                     0  13.8           65
## # ... with 63 more rows
```

Alternatively, we can use `filter` and `min_rank`:

```
bikes %>%
  group_by(time) %>%                           # Group by time
  filter(min_rank(desc(available_bikes))==1)   # Find largest in each group
```

```
## # A tibble: 92 x 5
## # Groups:   time [73]
##    name                       available_bikes available_bike_stands  time total_stands
##    <chr>                                <int>                 <int> <dbl>        <int>
##  1 MUSÉE D'ORSAY                           65                     0  13             65
##  2 MUSÉE D'ORSAY                           65                     0  13.1           65
##  3 MUSÉE D'ORSAY                           65                     0  13.2           65
##  4 MUSÉE D'ORSAY                           62                     3  13.2           65
##  5 MOUFFETARD EPEE DE BOIS                 62                     1  13.2           63
##  6 SAINT PLACIDE CHERCHE MIDI              62                     0  13.2           62
##  7 METZ                                    64                     0  13.3           64
##  8 DUPLEIX                                 64                     4  13.4           68
##  9 METZ                                    64                     0  13.4           64
## 10 METZ                                    64                     0  13.5           64
## # ... with 82 more rows
```

You might have noticed that the answers differ a little. The reason for this are ties: for example, at 1.15pm the stations at Mussée d'Orsay, Mouffetard Epée de Bois and Sainte Placide Cherche-Midi all had 62 bikes available. The former commands extracts just one of them, whereas the bottom command extracts all three. (You would obtain the same results if you replaced `min_rank` by `row_number`, which breaks ties by using in doubt the order in the data set).

Merging (joining) data sets using the `join`-type functions

Suppose we want to extract the data from `bikes` relating to bike stations in Hauts-de-Seine only. The table `bikes` does not however contain any information about the département in which the stations are located. We need to merge the information from the `stations` and `bikes`. This can be done using one of the `join` functions of `dplyr`. We will use `inner_join`, which only retains cases if there are corresponding entries in both data sets: this corresponds to the default behaviour of the R function `merge`.

The `join` functions will be default use the columns with common names across the two data sets ("natural join").

```
bikes %>% inner_join(stations) %>%              # Merge data (using common variable: name)
  filter(departement=="Hauts-de-Seine")
```

```
## Joining, by = "name"
```

```
## # A tibble: 5,333 x 10
##    name                        avail~1 avail~2  time total~3    id address   lng   lat depar~4
##    <chr>                         <int>   <int> <dbl>   <int> <dbl> <chr>   <dbl> <dbl> <chr>
##  1 SOLJENITSYNE (PUTEAUX)           56       4    13      60 28002 BOULEV~  2.25  48.9 Hauts-~
##  2 DE GAULLE 3 (NEUILLY)             3      19    13      22 22005 195 AV~  2.26  48.9 Hauts-~
##  3 NATIONALE (BOULOGNE-BILLA~       20       3    13      23 21015 39 RUE~  2.24  48.8 Hauts-~
##  4 MONTROSIER (NEUILLY)             20       5    13      25 22011 7 RUE ~  2.28  48.9 Hauts-~
##  5 PETIT (CLICHY)                   22       0    13      22 21113 2 RUE ~  2.30  48.9 Hauts-~
##  6 GRENIER (BOULOGNE-BILLANC~        9      12    13      21 21013 4 AVEN~  2.25  48.8 Hauts-~
##  7 MARTINIE (VANVES)                 0      24    13      24 21703 5-7 AV~  2.29  48.8 Hauts-~
##  8 MORICE 2 (CLICHY)                22       3    13      25 21106 2-4 RU~  2.31  48.9 Hauts-~
##  9 SELLIER (SURESNES)               34      17    13      51 21501 RUE DE~  2.23  48.9 Hauts-~
## 10 VALITON (CLICHY)                 22       2    13      24 21101 4 RUE ~  2.30  48.9 Hauts-~
## # ... with 5,323 more rows, and abbreviated variable names 1: available_bikes,
## #   2: available_bike_stands, 3: total_stands, 4: departement
```

We could have specified the column to used to join the data sets manually by adding the argument `by="name"` (or `by=c("name"="name")`, which allows using columns with different names in the two data set).

As a side note, in this example, we could have avoided joining the two tables. We could have first extracted the names of the stations in Hauts-de-Seine and then used those to subset the data from `bikes` (essentially the equivalent of a subquery in SQL):

```
names92 <- stations %>% filter(departement=="Hauts-de-Seine") %>%
               select(name)
bikes %>% filter(name %in% names92[[1]])
```

```
## # A tibble: 5,333 x 5
##    name                             available_bikes available_bike_stands  time total_stands
##    <chr>                                      <int>                 <int> <dbl>        <int>
##  1 SOLJENITSYNE (PUTEAUX)                        56                     4    13           60
##  2 DE GAULLE 3 (NEUILLY)                          3                    19    13           22
##  3 NATIONALE (BOULOGNE-BILLANCOURT)              20                     3    13           23
##  4 MONTROSIER (NEUILLY)                          20                     5    13           25
##   5 PETIT (CLICHY)                          22               0    13           22
##   6 GRENIER (BOULOGNE-BILLANCOURT)           9              12    13           21
##   7 MARTINIE (VANVES)                        0              24    13           24
##   8 MORICE 2 (CLICHY)                       22               3    13           25
##   9 SELLIER (SURESNES)                      34              17    13           51
##  10 VALITON (CLICHY)                        22               2    13           24
## # ... with 5,323 more rows
```

We had to use `names92[[1]]` to extract the entries of the tibble `names92` as a character vector (we could have also used `unlist(names92)`).

You might notice a small difference in the results returned by the two approaches. The former retains the columns from `stations` which we have inserted, whereas the latter only contains the columns which `bikes` contained to start with.

Task 11.

Merge the data sets `patients` and `weights` from the tasks from week 3. You can load the data sets using

```
load(url(paste("https://github.com/UofGAnalyticsData/R/raw/",
                "main/Week%203/patients_weights.RData",sep="")))
```

Use the merged data set to calculate the average weight of male and female patients.

Translating dplyr statements into SQL commands

Supplementary material:
Translating dplyr statements into SQL commands

When working with large data sets stored in a relational database, it would be inefficient to transfer the data sets first into R and then manipulate them using `dplyr` in R. It will in almost all circumstances be faster to perform the data management in the database using SQL first and then importing the data into R. Also, R needs to store all data in memory, so combining large data sets might quickly exhaust R's memory.

However, you do not need to write the SQL statements yourself. The dbplyr package automatically translates `dplyr` commands into SQL statements, so that you can still use `dplyr` commands in R as if the data was in R.

We will look at a small example using an in-memory SQLite database.

```
library(dbplyr)                                 # Load required packages

##
## Attaching package: 'dbplyr'

## The following objects are masked from 'package:dplyr':
##
##     ident, sql

library(DBI)
library(RSQLite)
con <- dbConnect(RSQLite::SQLite(), ":memory:")  # Connect to temporary database
dbWriteTable(con, "stations", stations)         # Copy data to database
dbWriteTable(con, "bikes", bikes)               # (not needed in real world)

stations.db <- tbl(con, "stations")             # Define references to the tables
bikes.db <- tbl(con, "bikes")

query <- stations.db %>% filter(departement=="Seine-Saint-Denis" | departement=="Val-de-Marne")
                                                # Translate dplyr instruction to query

query %>% show_query()                          # Show the equivalent SQL statement

## <SQL>
## SELECT *
## FROM `stations`
## WHERE (`departement` = 'Seine-Saint-Denis' OR `departement` = 'Val-de-Marne')
```

```
query %>% collect()                              # Run the query and show results

## # A tibble: 110 x 6
##    name                        id address                                   lng   lat depar~1
##    <chr>                    <dbl> <chr>                                   <dbl> <dbl> <chr>
##  1 LAGNY (MONTREUIL)        31001 96 RUE DE LAGNY - 93100 MONTREUIL        2.42  48.8 Seine-~
##  2 REPUBLIQUE (MONTREUIL)   31002 38 RUE DE LA REPUBLIQUE - 93100 MONTRE~  2.42  48.9 Seine-~
##  3 PARIS (MONTREUIL)        31003 237-241 RUE DE PARIS - 93100 MONTREUIL   2.42  48.9 Seine-~
##  4 PARIS 2 (MONTREUIL)      31004 175/179 RUE DE PARIS - 93100 MONTREUIL   2.42  48.9 Seine-~
##  5 PARIS 2  (MONTREUIL)     31005 127/129 RUE DE PARIS - 93100 MONTREUIL   2.43  48.9 Seine-~
##  6 REPUBLIQUE 2 (MONTREUIL) 31006 2/4 PLACE DE LA REPUBLIQUE - 93100 MON~  2.42  48.9 Seine-~
##  7 VINCENNES (MONTREUIL)    31008 7 BIS RUE DE VINCENNES - 93100 MONTREU~  2.44  48.9 Seine-~
##  8 DE GAULLE (MONTREUIL)    31009 13/15 PLACE DU GENERAL DE GAULLE - 931~  2.43  48.9 Seine-~
##  9 STALINGRAD (MONTREUIL)   31010 67-69 RUE DE STALINGRAD - 93100 MONTRE~  2.44  48.9 Seine-~
## 10 STALINGRAD 2 (MONTREUIL) 31011 27 RUE DE STALINGRAD - 93100 MONTREUIL   2.44  48.9 Seine-~
## # ... with 100 more rows, and abbreviated variable name 1: departement

dbDisconnect(con)                                # Disconnect from the database
```

Data Transformation Cheat Sheet

https://github.com/rstudio/cheatsheets/blob/main/data-transformation.pdf

RStudio have put together a very handy and compact cheat sheet for dplyr.

Background reading: Chapter 13 of R for Data Science

http://r4ds.had.co.nz/relational-data.html

Chapter 13 of *R for Data Science* gives a detailed overview of the functions in `dplyr`.

Reshaping data using tidyr

There is more than one way of laying out data

In this section we will continue to work with the Vélib' data. However we will restrict ourselves to the data between 1pm and 1.15pm and also only four stations: Dupleix, Bourse, Jussieu and Montparnasse.

```
station.names <- c("DUPLEIX", "BOURSE", "JUSSIEU", "MONTPARNASSE")
bikes.sm <- bikes %>%
              filter(name %in% station.names & time<=13.25) %>%
                                            # Subset the stations
              select(name, time, available_bikes, available_bike_stands)
                                            # Reorder columns
bikes.sm
```

```
## # A tibble: 16 x 4
##    name          time available_bikes available_bike_stands
##    <chr>        <dbl>           <int>                 <int>
##  1 DUPLEIX       13                57                    11
##  2 JUSSIEU       13                33                     0
##  3 BOURSE        13                52                     5
##  4 MONTPARNASSE  13                 1                    46
##  5 DUPLEIX       13.1              58                    10
##  6 JUSSIEU       13.1              33                     0
##  7 BOURSE        13.1              49                     8
##  8 MONTPARNASSE  13.1               1                    46
##  9 DUPLEIX       13.2              61                     7
## 10 JUSSIEU       13.2              33                     0
## 11 BOURSE        13.2              49                     7
## 12 MONTPARNASSE  13.2               1                    46
## 13 DUPLEIX       13.2              61                     7
## 14 JUSSIEU       13.2              33                     0
## 15 BOURSE        13.2              50                     7
## 16 MONTPARNASSE  13.2               1                    46
```

In the data set `bikes` we have for every time point and for every bike station two variables: the number of bikes available and the number of stands available. There is one row for every combination of time and bike station.

The data is arranged in a way which Hadley Wickham would call "tidy". For a data set to be "tidy" it needs to be arranged such that:

- each column corresponds to exactly one variable (in the sense of a measurement of the *same* underlying attribute (like weights, price, or the number of available bikes) across units/cases.
- each row corresponds to exactly one observational unit or "case"

Tidy data

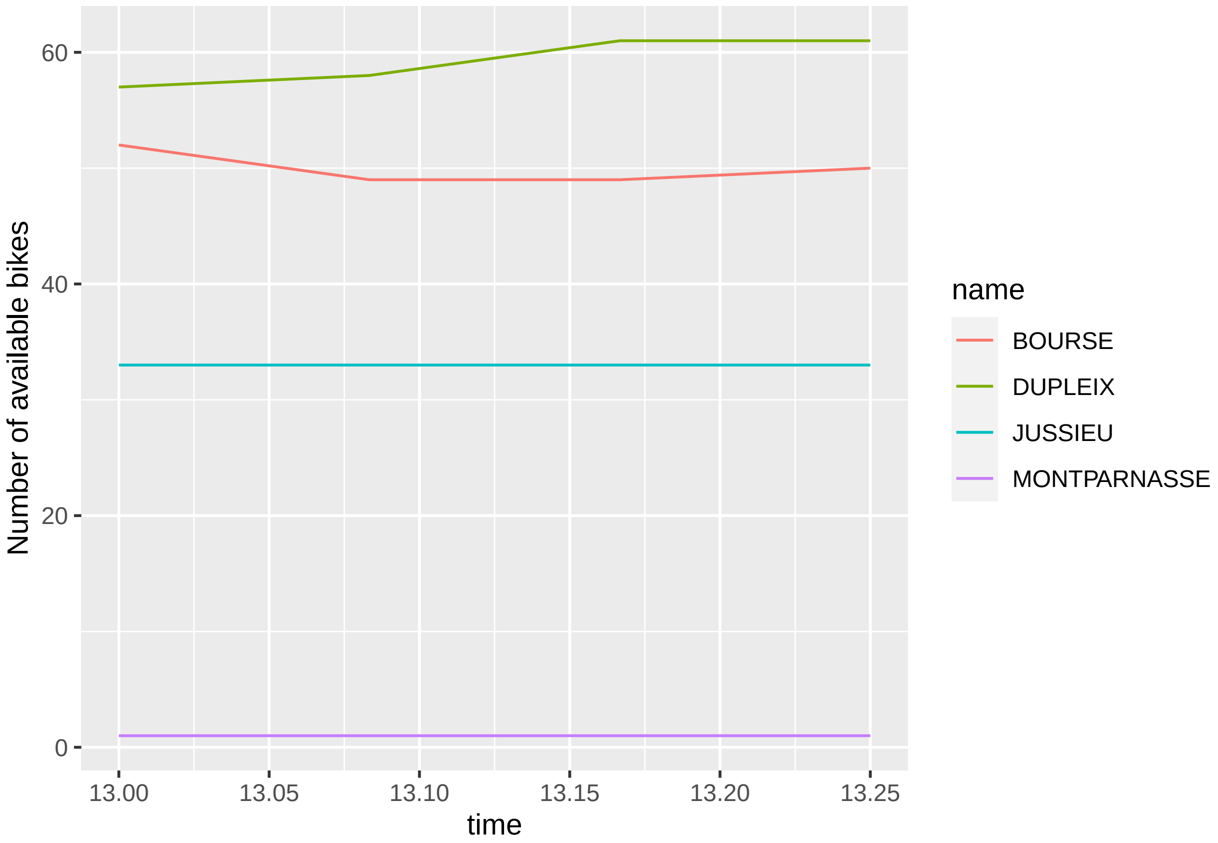
http://www.jstatsoft.org/v59/i10/paper

Wickham, H. (2014). Tidy Data. Journal of Statistical Software. Volume 59, Issue 10.

This paper explains some of the theory behind arranging data in a "tidy" way.

This "tidy" format is ideal when visualising the data set using ggplot2 (we will cover `ggplot2` in a fortnight)

```
library(ggplot2)
bikes.sm %>%
  ggplot(aes(time, available_bikes, colour=name)) +
    geom_line() + ylab("Number of available bikes")
```



There are many good reasons to store data in "tidy" format. However not all data we work with will be "tidy".

- When data is extracted from external sources we will not necessarily get hold of the data in "tidy" format, though data from relational databases is often already in a "tidy" format.
- Not every R function or modelling strategy expects data in this "tidy" format. Sometimes it is necessary to lay out the data in a different way. We will see that, for example, if we want to plot the data using the standard R plotting functions we need the data to be in a "non-tidy" format. (There is a nice (but slightly more protracted) blog post by Jeff Leek making the case that not all data are usefully tidy).

We can store the bicycle data in many other alternative formats, which however would all be "untidy".

> Happy families are all alike; every unhappy family is unhappy in its own way (Leo Tolstoy)
>
> Tidy data sets are all alike, but every messy data set is messy in its own way (Hadley Wickham)

Stored the "tidy" way, the number of available bikes is a single long column. This column is "indexed" by two other columns: the station name and the time: only together with these the number of available bikes makes sense. Given that the number of available bikes depends on two inputs (the station name and the time) we could arrange the data in a matrix, such that rows correspond to times and columns correspond to bike stations. This way of storing the data makes it "wider" and less "long". (We can create a second matrix which contains the numbers of available bike stands.)

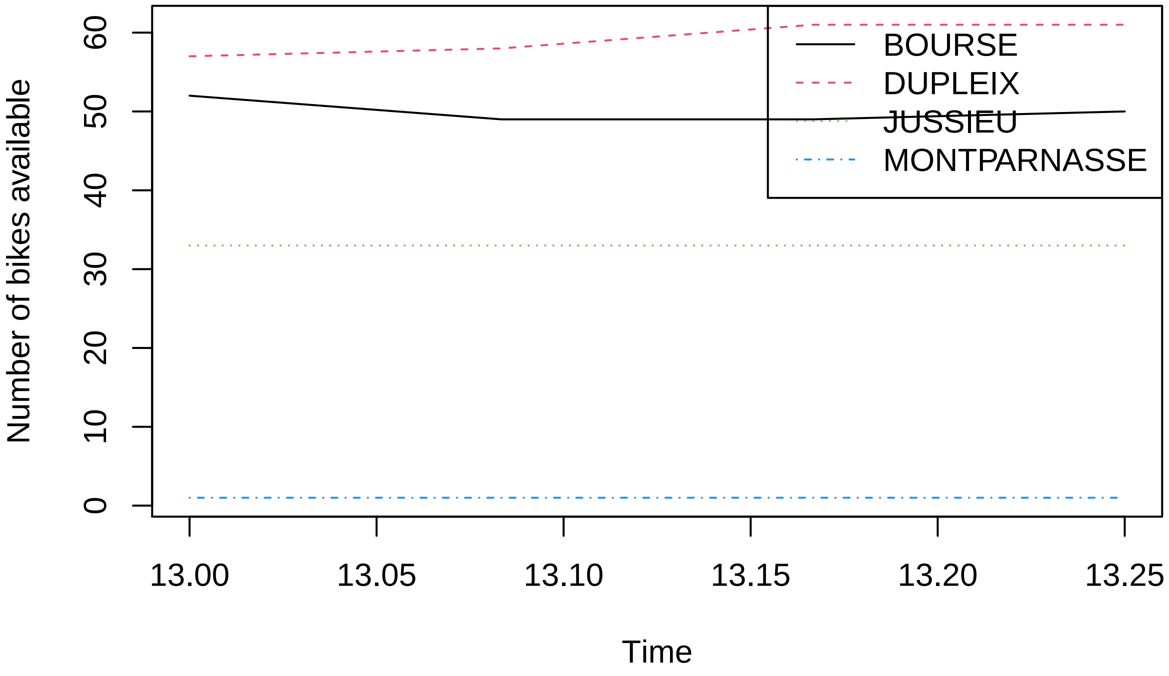
```
bikes.mat
```

```
## # A tibble: 4 x 5
##    time BOURSE DUPLEIX JUSSIEU MONTPARNASSE
##   <dbl>  <int>   <int>   <int>        <int>
## 1  13       52      57      33            1
## 2  13.1     49      58      33            1
## 3  13.2     49      61      33            1
## 4  13.2     50      61      33            1
```

This way of arranging the data is not "tidy": all the columns except for `time` contain measurements of the same type (the number of bikes available) and the column names are actually data. This way of arranging the data also means that the number of columns changes if new bike stations are added. This would however not be the case in our original layout, where it would just add additional rows.

However this matrix-style format is for example the format required when creating a plot of the number of available bikes using the standard R function `matplot` (we will look at this function in more detail next week).

```
matplot(bikes.mat%>%select(time), bikes.mat%>%select(-time),
        xlab="Time", ylab="Number of bikes available", type="l")
legend("topright", col=1:4, lty=1:4, unique(colnames(bikes.mat)[-1]))
```



Yet another alternative format of storing the data would be to store the number of bikes available and the number of bike stands available in alternate rows.

```
bikes.alt
```

```
## # A tibble: 32 x 4
##    name         time what                 bikes
##    <chr>       <dbl> <chr>                <int>
##  1 BOURSE       13   available_bikes         52
##  2 BOURSE       13   available_bike_stands    5
##  3 DUPLEIX      13   available_bikes         57
##  4 DUPLEIX      13   available_bike_stands   11
##  5 JUSSIEU      13   available_bikes         33
##  6 JUSSIEU      13   available_bike_stands    0
##  7 MONTPARNASSE 13   available_bikes          1
##  8 MONTPARNASSE 13   available_bike_stands   46
##  9 BOURSE       13.1 available_bikes         49
## 10 BOURSE       13.1 available_bike_stands    8
## # ... with 22 more rows
```

This way of storing the data is not “tidy” either. The third column does not contain data, but the description of the type of variable recorded and the fourth column contains measurements of different attributes (available bikes and available stands). This data set is “longer” than the original `bikes.sm` data.

Converting between different data layouts

It is often necessary to convert data between different (“tidy” and “non-tidy” layouts). The functions from `tidyr` help this these transformations.

Making data “longer” and less “wide” (“matrix to column”) The function `pivot_longer()` from `tidyr` reads the data from multiple columns and organises them as key-value pairs, making the data “longer” (more rows) and less “wide” (fewer columns). Consider the bicycle stored in wide matrix format:

```
bikes.mat
```

```
## # A tibble: 4 x 5
##    time BOURSE DUPLEIX JUSSIEU MONTPARNASSE
##   <dbl>  <int>   <int>   <int>        <int>
## 1  13        52       57       33              1
## 2  13.1      49       58       33              1
## 3  13.2      49       61       33              1
## 4  13.2      50       61       33              1
```

The data containing the number of available bikes is spread across the four columns, which are named after the stations.

If we want to reorganise the data and store the name of bike station in one column and the number of available bikes in another column we can use the function `pivot_longer()`:

```
bikes.mat %>% pivot_longer(cols=2:5,names_to = "station_name",values_to = "available_bikes")
```

```
## # A tibble: 16 x 3
##     time station_name available_bikes
##    <dbl> <chr>                  <int>
##  1  13   BOURSE                    52
##  2  13   DUPLEIX                   57
##  3  13   JUSSIEU                   33
##  4  13   MONTPARNASSE               1
##  5  13.1 BOURSE                    49
##  6  13.1 DUPLEIX                   58
##  7  13.1 JUSSIEU                   33
##  8  13.1 MONTPARNASSE               1
##  9  13.2 BOURSE                    49
## 10  13.2 DUPLEIX                   61
## 11  13.2 JUSSIEU                   33
## 12  13.2 MONTPARNASSE               1
## 13  13.2 BOURSE                    50
## 14  13.2 DUPLEIX                   61
## 15  13.2 JUSSIEU                   33
## 16  13.2 MONTPARNASSE               1
```

When used in a pipeline, the first argument to `pivot_longer` is the range of columns we wish to gather. The second argument is the name of the new column containing the names of the columns to be gathered ("keys" – in our case the name of the bike station). The third argument is the name of the new column containing the data ("values" – in our case the number of available bikes). `pivot_longer` will by default gather the data from all columns, unless we tell it not to. In our case, the column `time` is not a number of available bikes, so we have to tell R to leave it alone: we can do this by not selecting it within our list of specified columns.

Making data less "long" and "wider" ("column to matrix") We sometimes also have to go in the opposite direction and spread data stored in one column across several columns.

We can use the function `pivot_wider` for this. In our case (and in many other cases) it is easiest to first reduce the data to three columns: the future row identifiers (`time` in our case), the future column identifiers (`name` in our case) and the future content of the matrix (`available_bikes`)

```
bikes.sm %>%
  select(time, name, available_bikes) %>%
  pivot_wider(names_from = name,values_from = available_bikes)
```

```
## # A tibble: 4 x 5
##    time DUPLEIX JUSSIEU BOURSE MONTPARNASSE
##   <dbl>   <int>   <int>  <int>        <int>
## 1  13        57      33     52            1
## 2  13.1      58      33     49            1
## 3  13.2      61      33     49            1
## 4  13.2      61      33     50            1
```

When used in a pipeline the first argument to `pivot_wider` is the column that contains the future column names ("keys") and the second argument is the column that contains the data to be spread out in matrix form ("values").

Task 12.

The R example data set `Orange` contains records of the growth of five orange trees. Each row corresponds to one combination of age (in days since it was planted) and tree.

```
Orange <- as.data.frame(Orange)     # Convert to standard data frame
head(Orange, n=12)
```

```
##    Tree  age circumference
## 1     1  118            30
## 2     1  484            58
## 3     1  664            87
## 4     1 1004           115
## 5     1 1231           120
## 6     1 1372           142
## 7     1 1582           145
## 8     2  118            33
## 9     2  484            69
## 10    2  664           111
## 11    2 1004           156
## 12    2 1231           172
```

Use `pivot_wider` to arrange the data as matrix, such that rows correspond to ages and columns correspond to trees. Your data should be laid out similar to what is shown below (don't worry about the order of the columns).

```
## # A tibble: 7 x 6
##     age Tree_1 Tree_2 Tree_3 Tree_4 Tree_5
##   <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>
## 1   118     30     33     30     32     30
## 2   484     58     69     51     62     49
## 3   664     87    111     75    112     81
## 4  1004    115    156    108    167    125
## 5  1231    120    172    115    179    142
## 6  1372    142    203    139    209    174
## 7  1582    145    203    140    214    177
```

Data Import Cheat Sheet

https://github.com/rstudio/cheatsheets/blob/main/data-import.pdf

Rstudio's cheat sheet for data import also covers `tidyr`.

Background reading: Chapter 12 of R for Data Science

http://r4ds.had.co.nz/tidy-data.html

Chapter 12 of *R for Data Science* discusses the philosophy of "tidy" data (though in less detail than the paper linked above). It also covers functions in `tidyr` not discussed above.

Answers to tasks

Answer to Task 1. The code generates a random sample of size 1000 (from a standard normal distribution), computes the sine of each entry and then takes the maximum.

```
max(sin(rnorm(1000)))
```

```
## [1] 0.9999938
```

In this case the nested function call is easy to read because every function only takes one argument.

Answer to Task 2. You can use the following R code using pipelines.

```
library(MASS)
```

```
##
## Attaching package: 'MASS'
```

```
## The following object is masked from 'package:dplyr':
##
##     select
```

```
mammals %>%
  transform(ratio=brain/body) %>%
  subset(ratio==max(ratio))
```

```
##                  body brain    ratio
## Ground squirrel 0.101     4 39.60396
```

Oddly enough, ground squirrels have a higher brain-to-body weight ratio than humans.

Answer to Task 3. You can use the following R code.

```
courses_df <- data.frame(course=c("psm","psf", "rp"),
                         taught_by=c("Alexey","Eilidh and Colette", "Craig"),
                         weeks=c(11,11,11))
courses <- as_tibble(courses_df)

courses<- tibble(course=c("psm","psf","rp"),
          taught_by=c("Alexey","Eilidh and Colette", "Craig"),
          weeks=c(11,11,11))

courses <- tribble(~course,           ~taught_by, ~weeks,
                     "psm",             "Alexey", 11,
                     "psf", "Eilidh and Colette", 11,
                      "rp",             "Craig", 11)

courses
```

```
## # A tibble: 3 x 3
##   course taught_by          weeks
##   <chr>  <chr>              <dbl>
## 1 psm    Alexey                11
## 2 psf    Eilidh and Colette    11
## 3 rp     Craig                 11
```

You might notice a tiny difference: the first tibble stores the first two columns as factors, whereas the latter two store them as character strings. The reason for this different behaviour is that the first way first creates a data frame, and `data.frame` automatically converts character strings to factors.

Answer to Task 4. You can use the following R code:

```
courses$coursework_perc <- c(30,NA,100)
courses
```

```
## # A tibble: 3 x 4
##   course taught_by          weeks coursework_perc
##   <chr>  <chr>                <dbl>           <dbl>
## 1 psm    Alexey                  11              30
## 2 psf    Eilidh and Colette      11              NA
## 3 rp     Craig                   11             100
```

```
courses[1,]
```

```
## # A tibble: 1 x 4
##   course taught_by weeks coursework_perc
##   <chr>  <chr>     <dbl>           <dbl>
## 1 psm    Alexey       11              30
```

```
courses[,1:2]
```

```
## # A tibble: 3 x 2
##   course taught_by
##   <chr>  <chr>
## 1 psm    Alexey
## 2 psf    Eilidh and Colette
## 3 rp     Craig
```

Answer to Task 5. The first line of the file `cars.csv` contains the variable names and the fields are separated by commas. Missing values are encoded as asterisks.

```
cars <- read_csv("cars.csv", na="*")
```

```
## Rows: 20 Columns: 5
## -- Column specification --------------------------------------------------------------------
## Delimiter: ","
## chr (2): Manufacturer, Model
## dbl (3): MPG, Displacement, Horsepower
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
cars
```

```
## # A tibble: 20 x 5
##    Manufacturer Model     MPG Displacement Horsepower
##    <chr>        <chr>   <dbl>        <dbl>      <dbl>
##  1 Chevrolet    Camaro     19          3.4        160
##  2 Oldsmobile   Achieva    NA          2.3        155
##  3 Dodge        Spirit     22          2.5        100
##  4 Chevrolet    Astro      NA          4.3        165
##  5 Chevrolet    Corsica    25          2.2        110
##  6 Volkswagen   Corrado    18          2.8        178
##  7 Dodge        Stealth    18          3          300
##  8 Volkswagen   Fox        25          1.8         81
##  9 Cadillac     DeVille    16          4.9        200
## 10 Hyundai      Excel      29          1.5         81
## 11 Toyota       Tercel     32          1.5         82
## 12 Dodge        Colt       29          1.5         92
## 13 Volkswagen   Passat     21          2          134
## 14 Geo          Storm      30          1.6         90
## 15 Toyota       Previa     18          2.4        138
## 16 Nissan       Sentra     29          1.6        110
## 17 Toyota       Celica     25          2.2        135
## 18 Honda        Civic      42          1.5        102
## 19 Dodge        Caravan    17          3          142
## 20 Hyundai      Sonata     20          2          128
```

We could have also used the function `read_delim`.

```
read_delim("cars.csv", delim=",", na="*")
```

```
## Rows: 20 Columns: 5
## -- Column specification ---------------------------------------------------------------------
## Delimiter: ","
## chr (2): Manufacturer, Model
## dbl (3): MPG, Displacement, Horsepower
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
## # A tibble: 20 x 5
##    Manufacturer Model     MPG Displacement Horsepower
##    <chr>        <chr>   <dbl>        <dbl>      <dbl>
##  1 Chevrolet    Camaro     19          3.4        160
##  2 Oldsmobile   Achieva    NA          2.3        155
##  3 Dodge        Spirit     22          2.5        100
##  4 Chevrolet    Astro      NA          4.3        165
##  5 Chevrolet    Corsica    25          2.2        110
##  6 Volkswagen   Corrado    18          2.8        178
##  7 Dodge        Stealth    18          3          300
##  8 Volkswagen   Fox        25          1.8         81
##  9 Cadillac     DeVille    16          4.9        200
## 10 Hyundai      Excel      29          1.5         81
## 11 Toyota       Tercel     32          1.5         82
## 12 Dodge        Colt       29          1.5         92
## 13 Volkswagen   Passat     21          2          134
## 14 Geo          Storm      30          1.6         90
## 15 Toyota       Previa     18          2.4        138
## 16 Nissan       Sentra     29          1.6        110
## 17 Toyota       Celica     25          2.2        135
## 18 Honda        Civic      42          1.5        102
## 19 Dodge        Caravan    17          3          142
## 20 Hyundai      Sonata     20          2          128
```

The first line of the file `ships.txt` contains the variable names and the fields are separated by whitespace. Missing values are encoded as ".".

```
ships <- read_delim("ships.txt", delim=' ' , na=".")
```

```
## Rows: 40 Columns: 5
## -- Column specification ---------------------------------------------------------------------
## Delimiter: " "
## chr (1): type
## dbl (4): year, period, service, incidents
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
ships
```

```
## # A tibble: 40 x 5
##    type   year period service incidents
##    <chr> <dbl>  <dbl>   <dbl>     <dbl>
##  1 A        60     60     127         0
##  2 A        60     75      63         0
##  3 A        65     60      NA         3
##  4 A        65     75    1095         4
##  5 A        70     60    1512         6
##  6 A        70     75    3353        18
##  7 A        75     60       0         0
##  8 A        75     75    2244        11
##  9 B        60     60   44882        39
## 10 B        60     75   17176        29
## # ... with 30 more rows
```

Answer to Task 6. You can use the following R code:

```
bikes %>%
  filter(time==15 & available_bikes>60)
```

```
## # A tibble: 6 x 4
##   name                 available_bikes available_bike_stands  time
##   <chr>                          <int>                 <int> <dbl>
## 1 MUSÉE D'ORSAY                     63                     2    15
## 2 DUPLEIX                           65                     3    15
## 3 ASSEMBLEE NATIONALE               62                     0    15
## 4 SAINT EMILION                     65                     1    15
## 5 METZ                              63                     1    15
## 6 PRIMO LEVI                        61                     1    15
```

Answer to Task 7. We can create both columns in one call to `mutate`.

```
bikes %>%
  mutate(time_hour=floor(time), time_minutes=(60*time)%%60)
```

```
## # A tibble: 67,354 x 6
##    name                       available_bikes available_bike_stands  time time_hour time_mi~1
##    <chr>                                <int>                 <int> <dbl>     <dbl>     <dbl>
##  1 CHAMPEAUX (BAGNOLET)                     9                    41    13        13         0
##  2 POISSONNIÈRE - ENGHIEN                  33                     0    13        13         0
##  3 METRO ROME                               6                    38    13        13         0
##  4 DE GAULLE (PANTIN)                       2                    16    13        13         0
##  5 PARC DE BELLEVILLE (20040)               4                    22    13        13         0
##  6 SOLJENITSYNE (PUTEAUX)                  56                     4    13        13         0
##  7 SERRES                                   5                    18    13        13         0
##  8 PYRAMIDE ARTILLERIE                     14                    40    13        13         0
##  9 SAINT GEORGES                           12                    10    13        13         0
## 10 MUSÉE D'ORSAY                           65                     0    13        13         0
## # ... with 67,344 more rows, and abbreviated variable name 1: time_minutes
```

The output does not show the new columns (as they would take the output of a single row to more than one line). We can show them all, for example, if we remove the station name.

```
bikes %>%
  mutate(time_hour=floor(time), time_minutes=(60*time)%%60) %>%
  select(-name)
```

```
## # A tibble: 67,354 x 5
##    available_bikes available_bike_stands  time time_hour time_minutes
##              <int>                 <int> <dbl>     <dbl>        <dbl>
##  1               9                    41    13        13            0
##  2              33                     0    13        13            0
##  3               6                    38    13        13            0
##  4               2                    16    13        13            0
##  5               4                    22    13        13            0
##  6              56                     4    13        13            0
##  7               5                    18    13        13            0
##  8              14                    40    13        13            0
##  9              12                    10    13        13            0
## 10              65                     0    13        13            0
## # ... with 67,344 more rows
```

Alternatively, we can explicitly invoke the print method of the tibble and ask it to print everything.

```
bikes %>%
  mutate(time_hour=floor(time), time_minutes=(60*time)%%60) %>%
  print(width=Inf)
```

```
## # A tibble: 67,354 x 6
##    name                       available_bikes available_bike_stands  time time_hour
##    <chr>                                <int>                 <int> <dbl>     <dbl>
##  1 CHAMPEAUX (BAGNOLET)                     9                    41    13        13
##  2 POISSONNIÈRE - ENGHIEN                  33                     0    13        13
##  3 METRO ROME                               6                    38    13        13
##  4 DE GAULLE (PANTIN)                       2                    16    13        13
##  5 PARC DE BELLEVILLE (20040)               4                    22    13        13
##  6 SOLJENITSYNE (PUTEAUX)                  56                     4    13        13
##  7 SERRES                                   5                    18    13        13
##  8 PYRAMIDE ARTILLERIE                     14                    40    13        13
##  9 SAINT GEORGES                           12                    10    13        13
## 10 MUSÉE D'ORSAY                           65                     0    13        13
##    time_minutes
##           <dbl>
##  1            0
##  2            0
##  3            0
##  4            0
##  5            0
##  6            0
##  7            0
##  8            0
##  9            0
## 10            0
## # ... with 67,344 more rows
```

Answer to Task 8. We first sort the stations by the longitude and the select to top three observations.

```
stations %>%
  arrange(lng) %>%
  slice(1:3)
```

```
## # A tibble: 3 x 6
##   name                            id address                                lng   lat depar~1
##   <chr>                        <dbl> <chr>                                <dbl> <dbl> <chr>
## 1 GARE ROUTIERE ( SAINT CLOUD) 22101 GARE ROUTIERE - ARRET TRAM - 92210 ~  2.22  48.8 Hauts-~
## 2 SELLIER (SURESNES)           21501 RUE DE SAINT CLOUD / BOULEVARD HENR~  2.23  48.9 Hauts-~
## 3 VERDUN (SURESNES)            21502 18 BIS RUE DE VERDUN / COUR MADELAI~  2.23  48.9 Hauts-~
## # ... with abbreviated variable name 1: departement
```

We could have also used the function `filter` and the ranking function `min_rank`:

```
stations %>%
  filter(min_rank(lng)<=3)
```

```
## # A tibble: 3 x 6
##   name                            id address                                lng   lat depar~1
##   <chr>                        <dbl> <chr>                                <dbl> <dbl> <chr>
## 1 SELLIER (SURESNES)           21501 RUE DE SAINT CLOUD / BOULEVARD HENR~  2.23  48.9 Hauts-~
## 2 VERDUN (SURESNES)            21502 18 BIS RUE DE VERDUN / COUR MADELAI~  2.23  48.9 Hauts-~
## 3 GARE ROUTIERE ( SAINT CLOUD) 22101 GARE ROUTIERE - ARRET TRAM - 92210 ~  2.22  48.8 Hauts-~
## # ... with abbreviated variable name 1: departement
```

`min_rank` returns the rank of the observation when considering the variable given as argument (there are many different ways of computing ranks, see `?min_rank` for details.)

However, the latter answer does not show the stations in increasing order of longitude.

Answer to Task 9. There are many possible alternatives. One would be to consider the standard deviation of the number of available bikes, i.e. we measure how much the number of available bikes fluctuates over time.

```
bikes %>% group_by(name) %>%                 # Group by station name
  summarise(sd_bikes=sd(available_bikes)) %>% # Calculate standard deviations
  arrange(desc(sd_bikes))                     # Sort in descending order
```

```
## # A tibble: 928 x 2
##    name                             sd_bikes
##    <chr>                               <dbl>
##  1 DUPLEIX                              20.7
##  2 LAGROUA                              19.0
##  3 BOURSE                               18.9
##  4 PRIMO LEVI                           18.0
##  5 MUSÉE D'ORSAY                        18.0
##  6 PAU CASALS                           17.3
##  7 MOUFFETARD EPEE DE BOIS              16.9
##  8 BOULEVARD VOLTAIRE                   15.8
##  9 VICTOIR CHAUSSEE D ANTIN             15.2
## 10 CANAL SAINT DENIS - BD MACDONALD     15.2
## # ... with 918 more rows
```

The answer obtained this way is rather different from what we have obtained before.

Answer to Task 10. We can use the following R code:

```
stations %>% group_by(departement) %>%         # Group by department
  summarise(n_stations=n()) %>%                # Count cases
  arrange(desc(n_stations))                    # Sort in descending order
```

```
## # A tibble: 4 x 2
##   departement       n_stations
##   <chr>                  <int>
## 1 Paris                    743
## 2 Hauts-de-Seine            75
## 3 Seine-Saint-Denis         60
## 4 Val-de-Marne              50
```

Answer to Task 11. You can use the following R code:

```
weights %>%
  inner_join(patients) %>%             # Merge with patients data (using common variable: PatientID)
  group_by(Gender) %>%                 # Group observations by gender
  summarise(AvgWeight=mean(Weight))    # Calculate group-wise means
```

```
## Joining, by = "PatientID"
```

```
## # A tibble: 2 x 2
##   Gender AvgWeight
##   <fct>      <dbl>
## 1 female      53.2
## 2 male        83.3
```

Answer to Task 12. We can spread the content across columns using

```
OrangeSpread <- Orange %>% pivot_wider(names_from=Tree,values_from = circumference,names_prefix = "Tree
OrangeSpread
```

```
## # A tibble: 7 x 6
##     age Tree_1 Tree_2 Tree_3 Tree_4 Tree_5
##   <dbl>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>
## 1   118     30     33     30     32     30
## 2   484     58     69     51     62     49
## 3   664     87    111     75    112     81
## 4  1004    115    156    108    167    125
## 5  1231    120    172    115    179    142
## 6  1372    142    203    139    209    174
## 7  1582    145    203    140    214    177
```

Setting `names_prefix` adds the variable name ("Tree_") to the names of the new columns using `names_prefix` as a separator.