Part I: Introduction to R

This sections presents examples of R code whose syntax and results are chosen to illustrate who R works. All R commands are written as text that must respect a given syntax. Instead of describing details of the syntax, these exercises are meant to let you discover and understand R through commented examples.

In windows, R comes with a standard editor. Some prefer to use RStudio who provides a nicer user interface. This document does not focus on the interface, but on the code itself. Feel free to use any of the external tools that help you be productive in R coding.

Good working habits

R is an interpreted language: you may enter commands directly in the shell and they get run. Never develop your code directly in the console: you should rather use an editor as it will allow to keep track of your work, modify it, and run it over when needed. There are shortcuts for submitting code from the editor to the R console. The standard editor use the shortcuts `F5` or `CTRL-R`, RStudio uses `CTRL-RETURN`.

When coding, you should always add comments to keep track of what you are doing. When you need to use your code one year later, you may not remember what you were doing if you did not comment it properly, let alone if you share your code with others! On any given line, all text following the # symbol will be ignored by R. Do not be greedy on comments, they are useful!

Creating objects

The symbols `<-` and `=` allow to create or modify objects. Here are some examples where objects are created. After creating an object, type its name to see the result in the console. Try a few additional examples of your own.

```
x=2
x=c(1,4,5.6,9)
x=1:10
x=rep(1:3,c(2,5,3))
x=rep(c("A","B","C"),each=5)
```

The `c()` command is a function that creates a vector of objects or values. The `rep()` function creates a sequence of numbers. A semicolon `:` is a shortcut for consecutive integers.

The names of an R object must start with a letter and be followed by letters or numbers. The names are case sensitive. If you create an object of the same name as an existing R object, the latter may masked. Avoid using names such as `c`, `rep` or `pi` as they could interfere with the rest of the code.

You may see the list of existing R objects with the `ls()` function. To erase the object `x`, type `rm(x)`. You may not delete the R base functions, so if you mask an important function or value, you may recover it by removing the faulty object. For instance:

```
pi
pi=4
pi
```

```
rm(pi)
pi
```

Help on functions

If you forgot the exact syntax of an R function, you may type `help(name_of_the_command)`. The `help.start()` function allows to browse general help. You may also use Google to find the official help, as well as numerous comments and examples!

Operations on vectors

Mathematical operations are applied component-wise on vectors.

```
n=c(98,10,3,1)                     # number of employees per category
salary=c(36,42,54,81)*1000         # salary of employees per category
salary/12                          # monthly salary per category
n*salary                           # payroll per category
inc=c(.02,.02,.03,.05)             # salary increase per category
incsalary=salary*(1+inc)           # new salaries
```

Other functions are applied to the complete vector rather than its elements. Often, the vector may be considered as a data set, or a variable of a dataset.

```
sum(n*incsalary)                   # payroll after increase
allsal=rep(incsalary,n)            # list of the salary of each employee
mean(allsal)                       # mean salary
sd(allsal)                         # standard deviation of the salary
quantile(allsal,c(.1,.9))          # quantiles 10% and 90% of salaries
summary(allsal)
```

Types of variables

There exists different types of variables. The most common are:

- numeric (numbers)
- logical (TRUE or FALSE)
- factor (nominal)

Some functions behave differently depending on the type of variable.

```
x=rep(c("M","W"),c(4,5))
y=as.factor(x)
is.factor(y)
summary(x)
summary(y)
```

When a logical variable is converted to numeric, TRUE=1 and FALSE=0. For instance,

```
mean(x=="M")
```

Accessing parts of a vector

Brackets `[]` allow to access parts of a vector, or to remove some of its elements.

```
x=2001:2019
x[4]
x[c(3,4)]           # you may provide a vector of elements to retain,
x[-7]               # a vector of elements to remove
x[x>2006]           # or a vector of logical values.
```

In the latter example, using a vector of logical values of the same size as the vector of interest will retain only elements corresponding to a TRUE.

Matrices

The syntax to access parts of a vector may also be used to access elements of a matrix.

```
x=matrix(1:12,3,4)
x
x[2,2]
x[-2,]
x[x>5]
```

When using the comma, letting a space empty means that all corresponding rows or columns are kept. If the comma is not used, the matrix is treated as a vector where all columns of the matrix are stitched one after the other.

Mathematical operations are performed component-wise on matrices as they were on vectors. Matrix operators (applied on vectors in the examples below) use different symbols. The `t()` function transposes a matrix (or a vector that becomes a matrix through that step).

```
t(1:3) %*% 1:3
1:3 %*% t(1:3)
```

Loops and conditional statements

It is possible to run loops or conditional statements in R. For simple operations, when possible, inserting the conditions in the brackets of the vectors is shorter and more efficient.

```
for(i in 1:10){
  if(i>5){
    cat("Number",i,"is greater than 5\n")
  }
  else {
    cat("Number",i,"is less than or equal to 5\n")
  }
}
```

The `cat()` function displays a message in the console. It is possible to combine text and variables as illustrated. To create a character variable in a similar fashion, the `paste()` function may be used. This is useful, for instance, if you need to create of read a bunch of files and need to generate their names by pasting path, filename and a counter.

Graphics

R produces high quality graphics that can easily be inserted in Word, PowerPoint, or be exported to a file, including an EPS vectorial format that offers better quality and much flexibility.

```
x=rnorm(30)        # Randomly generated data
hist(x)
boxplot(x)
```

In descriptive statistics, side-by-side boxplots are very good to compare the values of a variable between different groups. The syntax `y~x` means that `y` is explained by `x`.

```
group=sample(1:2,30,replace=TRUE)    # random choice of a group for each
boxplot(x~group)
```

Similarly, a sequence of data may be plotted to see their evolution.

```
year=2001:2019
sales=rnorm(19,mean=10000+10*year,sd=100)   # randomly generated numbers
plot(year,sales,type='l')
title("Evolution of sales")
```

Different packages allow to plot graphics in other ways. The package `ggplot2` is very popular and produces graphics with a different look and feel. It is based on a grammar of graphics – the syntax is quite different. Packages such as `rgl` can be used for multi-dimensional plots. Other packages may be used for maps. This document restrain from going into much details about any of the package, focusing on the basics first, but feel free to explore them There are plenty of examples online.

Reading a data file

R objects may be saved through the `save` and `load` functions, but typically, the data will be available under a structured file. Many functions such as `scan`, `read.table`, `read.csv`, `read.fwf` allow to read such data.

Download the data file `employees.txt`. It contains information about sales (number of transactions and total value) for each employee of a company for each month of a given year.

```
x=read.table("employees.txt")  # Write complete path, or change
                                # File → set working directory
x
names(x)                       # Name of variables in x
unique(x[,1])                  # Removes repeated values
boxplot(x[,3]~x[,1])
title("Monthly number of sales per employee")
```

The `x` object is a `data.frame`, a special kind of list. We will discuss list and data frames further later, but for now, note that you may access the columns by their name.

```
boxplot(x$TotSales~x$Month,col="gray",lty=1,pch=20,xlab="Month")
title("Total sales of each employee per month")
```

Optional arguments of the graphical functions can make them look nicer! Help on graphical parameters are found under the `par` function that can modify characteristics of many graphical functions. Try `help(par)` for further information.

Other functions may be used to show multiple graphics simultaneously:

```
emps=unique(x$EmpID)
par(mfrow=c(2,3))       # The par function controls some graphics properties
for(i in 1:6){
  plot(1:12,x$TotSales[x$EmpID==emps[i]],type='l',xlab="Month",
    ylab="Total Sales",ylim=range(x$TotSales))
  title(paste("Montthly sales of employee",emps[i]))
}
```

Note the `ylim` option that sets the height of the graphics, giving them a common scale so they can be compared. If we omit fixing the scale with this data, all graphics will look alike, but they will have different scales.

Structure of statistical analyses

The results from fitting a model are typically a list (details to come) containing information about the model and its fit. . To extract useful information, we may use functions such as `print`, `summary`, `plot`, `predict`, `residuals`, anova, etc.

Let us consider an example where 500 clients answered a questionnaire that we used to evaluate the five dimensions of their personality (the *Big 5*). We use a linear regression to link the amount they spent to their personality, but we first consider a simpler analysis that considers only extraversion. You probably came across linear regression at some point at school, so you should be able to make sense of the output.

```
x=read.table("Big5.txt")
reg=lm(Amount ~ Extraversion, data=x)
print(reg)         # Summary of the fit
summary(reg)       # More details on the fit
plot(reg)          # Plots of residuals
predict(reg)       # Predictions for the dataset
names(reg)         # To see the name of the elements of the list
```

You may also fit a multiple linear regression and proceed to model selection. By default, the stepwise method uses AIC rather than p-values to dismiss variables.

```
mreg=lm(Amount ~ Neuroticism + Extraversion + Openness + Agreeableness +
                  Conscientiousness, data=x)
summary(mreg)
step(mreg)         # Stepwise method
```

The `print` and `summary` methods are almost always available. Other functions are usually available as well when they are relevant.

Exercises

Using the same `employees.txt` file, answer the following questions.

1. What is the total number of sales transactions for this company during the year (for all employees)? What is the total value of those sales?
2. Which employee holds the record for the most sales in a given month this year? What is that record?
3. Compute the total amount of the sales made by each employee during the year. Who sold the most?
4. Create figures that look at the monthly average amount of a sale: one figure that compares months, and one that can compare employees to each other.
5. Compute the average amount of a sale for each employee for the entire year.
6. Trace one plot per employee showing the evolution of their monthly average sale during the year.
7. Fit a linear regression where we look for the time trend (with the variable month) on total sales with a fixed effect for each employee (their ID is a factor variable) to account for the difference in their ability.

Index of functions mentioned in this document (in order of appearance)

`c`	`quantile`	`{}`	`read.table`	`read.csv`
`rep`	`summary`	`cat`	`names`	`read.fwf`
`ls`	`as.factor`	`paste`	`unique`	`par`
`rm`	`is.factor`	`rnorm`	`title`	`lm`
`help`	`mean`	`hist`	`save`	`summary`
`help.start`	`matrix`	`boxplot`	`load`	`predict`
`mean`	`for`	`sample`	`scan`	`step`
`sd`	`if {} else`	`plot`	`read.table`	

Part II: lists, arrays, functions and graphics

In this section, we continue the self-paced exploration of R through examples. Make sure you understand each section of code presented and try the exercises on your own. Ask questions as needed.

Missing values

Datasets typically contain missing values. In R, missing values are represented as NA (*not available*), a specific symbol which is different from the chain of characters "NA". Missing values may come from the file (because the value of the variable is unknown), or be generated by a calculation whose solution does not exist. In the latter case, they are indicated as NaN (*not a number*).

```
x=c(1,NA,5.6,-1,0)
is.na(x)
y=log(x)
y
is.na(y)
is.nan(y)
is.finite(y)
```

The NaN missing value is considered as a particular case of NA, but a value NA is not NaN. To know if a value is missing, we must use the function `is.na()`. Note also that some operations such as a division by 0 or log(0) return `Inf` or `–Inf`. These symbols are used to represent infinity and are not considered as missing.

Many functions have options that allow a special treatment of missing values. The details of these options are found in the help documentation. For instance,

```
mean(x)
mean(x,na.rm=TRUE)
```

To treat the missing value manually, you may use the `is.na()` function.

```
sum(is.na(x))            # Number of missing values
x[!is.na(x)]             # Vectors of non missing values
```

Did you note the exclamation mark? It is the “not” operator. In front of a logical expression, the exclamation mark will change all TRUE as FALSE, and inversely.

Empty object (NULL)

There exists an empty object in R (NULL) which is very useful to initiate loop of functions. If you attempt to use an object that does not exist in R, you get an error that stops the execution.

```
rm(x)
c(1:5,x)
```

With the NULL object, this error does not occur. Using NULL may be used with a loop to create an object that will be constructed incrementally.

```
x=NULL
for(i in 1:10){
   x=c(x,i)
}
x
```

Logical operators

Evaluating logical expressions allows to create conditional statements. Mathematical comparisons are made with operators <, >, <=, >=, == and !=. Note the double "equal". If you use a simple equal, R will understand that you are creating a new object rather than evaluating a logical expression. When vectors are compared, the logical statement is evaluated component-wise.

```
x=2
x==2
x>0
y=1:10
y!=5
y[y!=5]
```

Logical operatos may be used to combine conditions. Important symbols are: & (and), | (or), ! (not).

```
x=1:10
(x<7) & (x>3)
x[(x<7) & (x>3)]
```

Lists

In the first series, we worked with vectors and matrices. These objects are made of elements of a single type. Lists are more flexible since they may contain objects and values of different types.

```
w=read.table("employees.txt")
d=list(year=2012,
   succ="Montreal Centre",
   ID=1000:1007,
   nb=t(matrix(w$nSales,nrow=12)),
   amount=t(matrix(w$TotSales,nrow=12))
)
names(d)
d$nb
d[[4]]
d[[4]][1,]
```

To access the elements of a list, you may use $ followed by the name of the element, or a double bracket with the number of the element in that list. Note that in a same object, we stores a number, a chain of characters, a vector of numbers and two 8 by 12 matrices.

Given the flexibility of a list, the output of some functions are under that form. This is the case for linear models, including regression.

```
x=1:20
y=4+2*x+rnorm(20)  # Creates a dataset
a=lm(y~x)          # Fits a linear regression
```

```
summary(a)
names(a)
a$residuals
a$coeff
```

When it is not ambiguous, R will complete the name of the elements of the list. This is why the element `coefficients` is shown even though we wrote only the beginning of this expression.

data.frame

A `data.frame` is a special type of list whose elements are vectors of the same length, but possibly different types. While a matrix must contain only one type of data, a `data.frame` may contain multiple types, like a real dataset.

```
sex=as.factor(rep(c("M","W"),each=48))
w=data.frame(w,sex)      # Add the sex variable to an existing data.frame
summary(w)
w$EmpID=as.character(w$EmpID)
summary(w)
```

Even if a `data.frame` is made of different types of data, it is possible to access its elements with brackets as if it were a matrix.

```
w[1,]
w[w$EmpID=="1000",]
```

The quotes allow to specifiy a chain of characters, made necessary here because we converted `EmpID` above.

To check if an object is a list or a `data.frame`, you may use the functions `is.list()` and `is.data.frame()`. The function `names()`works on lists as well as `data.frame`.

If required, it is possible to modify the type of a colun of a `data.frame` by applying the appropriate function. For instance, the following code converts `EmpID` into a factor.

```
w$EmpID=as.factor(w$EmpID)
summary(w)
summary(w$EmpID)
```

Such conversions are often very useful after reading a dataset.

Functions

We have used numerous functions so far, but we may also create some to automatize some tasks.

```
nb.na=function(x,p=.1){
# x : a vector
# p : proportion of missing values that is acceptable
# output : number of missing values
   if(mean(is.na(x))>=p) warning("There are too many missing values.")
   sum(is.na(x))
}
```

```
nb.na(c(1,2,3,4,NA))
nb.na(c(1,2,3,4,NA),p=.25)
```

In the definition of the function, we state what variables should be used. To establish a default value, we may determine it using an "=" sign. The last line is returned as an output, except if the `return()` function is used to do this explicitly. Statement `warning()` and `stop()`transmit a warning of an error message in the console.

```
sales=function(x,emp){
# x   : dataset
# emp : employee ID whose data we want to analyze
# output : graphics for employee emp

   if(sum(x$EmpID==emp)==0) stop("This employee does not exist.")
   par(mfrow=c(1,2))
   plot(1:12,x$nSales[x$EmpID==emp],type='l',xlab="Month",
      ylab="Number of transactions")
   title(paste("Number of transactions of employee",emp,"by month"))
   plot(1:12,( x$TotSales/x$nSales)[x$EmpID==emp],type='l',xlab="Month",
             ylab="Average sale")
   title("Average amount of a sale per month")
}

sales(w,1001)
sales(w,1006)
sales(w,1)
```

In this example, plots are produced, but no output is sent back to the console.

Recycling

Some operators use vectors. If a vector is too short, its values are recycled: R starts over from the first value of the vector until all required spaces are filled.

```
cbind(1:10,1:3)    # Sticks vectors side by side into a matrix
matrix(1:4,4,5)    # Creates a 4 x 5 matrix from a single vector
```

A warning will tell if recycling ends in the middle of a vector. Sometimes, recycling is very useful. For instance,

```
sum(w$nSales*c(1,1,1,0,0,0,0,0,0,0,0,0))   # Nb sales Jan to Mar
```

Arrays

A matrix is a table of numbers in two dimensions. Inside a computer, we may handle more dimensions.For instance,

```
w=read.table("employees.txt")
a=array(as.matrix(w),c(12,8,4))
a[1,1,]
```

The `as.matrix()` function forces all variables into the same type (numerical here). In comparison, the `data.frame` could contain different types, including factors or characters. There is now three indices to access the appropriate data. The first index is for the month, the second for the employee, and the third will access the different variables that we have. Using an `array`, it becomes easy to access a subset of the data. For instance,

```
a[1,,c(1,3)]       # Number of sales in January for all employees
a[,3,4]/a[,3,3]    # Average monthly sale for the 3rd employee
```

Working with an `array` requires a higher level of abstraction, but once you get used to it, this approach is very lean and efficient. Personally, I use them heavily in connection with the implicit `apply` loops that we discuss next.

Implicit "apply" loops

The `apply()` function and its variants `sapply()`, `lapply()` and `tapply()` allow to create implicit loops where a function is applied to a "slice" of a matrix or an `array`, or to the elements of a list.

```
nbv=a[,,3]              # Number of sales : lines = month, col = employees
apply(nbv,1,mean)       # Average per month
apply(nbv,2,mean)       # Average per employee
apply(nbv,1,range)      # Min and max values per month
apply(nbv,2,range)      # Min et max values per employee
```

In an apply statement, we must first specify the R object (a matrix, an array, or a data.frame). We must then specify on which indices the loop should run. In the first example above, the value 1 indicates to loop on the first dimension, hence each line will be treated in turn. The third argument is the function that will be applied to each element (i.e. each line). The `apply()` statements apply to an `array` as well, allowing to replace multiple loops.

```
                              # Average for number and total values of sales
apply(a[,,3:4],c(1,3),mean)   # Per month
apply(a[,,3:4],c(2,3),mean)   # Per employee
```

The variants `sapply()` and `lapply()`are used for vectors and lists. Note that you may apply your own function with an `apply` statement.

```
fn=function(i){sum((1:10)^i)}
sapply(0:5,fn)
```

Finally, `tapply()` is different as it is used with vectors, not with an `array`. It applies a function to a subset of a vector that correspond to each category defined by a `factor` variable. For instance, the following code provides the total number of transactions for each employee during the year.

```
tapply(w$nSales,as.factor(w$EmpID),sum)
```

The `apply` statements may appear strange at first, and their coding may feel less intuitive initially, especially for those who are used to explicit loops. However, their use provide a leaner more compact code and eliminates errors related to the index in a loop. When I code in R, I rarely use an explicit loop.

Parallel computing

One additional advantage of `apply` statements is the easiness with which we can parallelize them. Modern computers typically have multiple cores. When a program is run, it will use only one core unless specifically designed to do otherwise. Parallel computing consists in splitting the task in pieces that can be performed simultaneously by multiple cores. There exists functions in R to help parallelizing your code.

The following example illustrates parallel computing, but it uses a "package". We will talk about packages again later. The `system.time()` function is used to measure the computing time, so we can compare both approaches.

```
fn=function(i){ sum(log(sqrt((1:10000)*i))) }
system.time(sapply(1:10000,fn))          # Using one core
library(parallel)
cl=makeCluster(4)
system.time(parSapply(cl,1:10000,fn))    # Using 4 cores
```

More graphics

Here are additional examples of graphics. Try modifying their parameters, explore.

```
pie(apply(nbv,2,mean),labels=unique(w$EmpID))
title("Proportion of sales by each employee")

barplot(nbv[,1],names.arg=month.abb[1:12])
title("Number of sales per month for employee 1000")
```

Once a figure is plotted, other functions such as `lines`, `points`, `text`, `abline` and `legend` may add to it. The file `Rapplets.zip` contains numerous examples that I use for demos in class. You are welcome to explore them, especially `Bayes.R`, `CLT-truedist.R` and `regression.R` to see how we can create tailored outputs.

Using a "package"

As soon as you start R, a large number of functions are available. However, there is a much larger ecosystem of available functions that are not available by default. R was designed to easily share code that is ready to use, and such functions are available as a "package".

To install a package, you may use the function `install.packages` that has the ability to connect directly to the Comprehensive R Archive Network (CRAN). Once the package is installed on your system, use `library` to load it.

```
install.packages("depth")      # Run just the first time
library(depth)                 # Run once if you need the package in a session
```

The name in quote corresponds to the name of the package to install. In this example, the "depth" package allows to compute depth functions which are generalisations of ranks in multiple dimensions, which in turn may be used to define generalized medians. They are not very common measures.

```
x=read.table("salesmen.txt")
jan=x[x[,2]==1,4]/x[x[,2]==1,3]
feb=x[x[,2]==2,4]/x[x[,2]==2,3]
ID=unique(x$EmpID)
data=cbind(jan,feb)[ID<1100,]
data=data[complete.cases(data),]
isodepth(data)                 # Plot average amount of a sale Jan vs Feb
isodepth(data,twodim=FALSE)
med(data)
```

You may explore available packages on `r-project.org`. As you search examples online, many will also use different packages.

Exercises

Data for the illustrations were only about one branch of this company. The file `salesmen.txt` presents the results for each branch. Each employee has 12 consecutive entries, one per month, and the data are in increasing number of employee number. Note that the first two digits of employee numbers are the branch number.

1. How many missing values are found in this data set for each of the variables?
2. What can you notice regarding employees of branch 25 (employee numbers 25xx)? Can you find a plausible reason for their missing values?
3. Determine how many FTE employees (full-time equivalent) are in each branch. When these salesmen work, they are full time. If a salesman has worked for 3 months, then he corresponds to 3/12 FTE.
4. Create a list where each element is the matrix of total sales (employee × month) for a branch.
5. Using an `_apply` statement, compute total sales for each branch. Create a plot that shows the relative contribution of each branch to the sales of the company.
6. Compute a measure of performance: the sales of each branch per FTE. Create a plot that allows to compare the efficiency of the branches.
7. Write a function that takes the branch number as input and returns a summary of the evolution of sales (number and total amount) through time.

Index of functions mentioned in this document (in order of appearance)

`is.na`	`t`	`if`	`apply`	`barplot`
`log`	`names`	`warning`	`sapply`	`lines`
`is.nan`	`rnorm`	`return`	`lapply`	`points`
`is.finite`	`lm`	`stop`	`tapply`	`text`
`mean`	`summary`	`ventes*`	`range`	`abline`
`sum`	`as.factor`	`plot`	`fn*`	`legend`
`rm`	`data.frame`	`title`	`system.time`	`install.packages`
`c`	`as.character`	`cbind`	`library`	`complete.cases`
`for`	`is.list`	`matrix`	`makeCluster`	`isodepth`
`read.table`	`is.data.frame`	`array`	`parSapply`	`med`
`list`	`nb.na*`	`as.matrix`	`pie`	

*: Functions that were created in the examples.

Part III: Additional topics

Split-apply-combine – Concept

Data analysis requires frequent use of some tasks:

- Split data in parts
- Apply a function to each part
- Combine results

Or in short, the split-apply-combine concept. Some libraries such as `dplyr` have been developed to make such steps with simplified easier to read code.

Piping – Concept

The piping operator %>%, often called "pipe" allow to replacer parentheses in the call of some compatible functions. As an illustration, the two following sets of command are equivalent:

```
install.packages("dplyr")
library("dplyr")
employees=read.table("employees.txt")
nrow(employees)  #traditionnal way of using nrow
employees %>% nrow   #using the pipe
```

The pipe operator takes everything in the left, and inserts it as the first argument of the function on the right. Piping hence replaces hard to read imbedded series of parentheses into a more natural sequence of operations. The code is read from left to right, and changes are much easy to perform. See for instance:

```
nrow(filter(employees, EmpID == 1000))
employees %>%
  filter(EmpID == 1000) %>%
  nrow()
```

Lines may be removed, added or changed easily. It is recommended to change line after each pipe to improve the code's readability, and to indent all lines with respect to the first line of a sequence.

The `dplyr` library

The `tidyverse` library is a collection of tools for data science. Among those tools, the `dplyr` library provides keywords (functions) that are frequently used when preparing data, including:

- select : select columns
- filter : filter lines
- arrange : sort or arrange rows
- mutate : create new columns
- summarise : Aggregate values of multiple observations
- group_by : apply a function to subgroups based on the split-apply-combine concept

These functions all accept a table as input and return data tables as output. The `dplyr` library is optimized for working with data frames. The basic syntax involves parentheses as most functions in R, but they may also be used with the pipe operator and a particular syntax.

Select: The first argument is the data table, the rest are the columns to keep, or to remove

```
empl_mo = select(employees, EmpID, Month)
empl_mo2 = select(employees, -nSales, -TotSales)  # Equivalent in this case
```

In addition, a colon may be used to refer to a group of consecutive columns in the table

```
empl_mo_ns = select(employees, EmpID:nSales)
```

Additional arguments offer efficient shortcuts:

- starts_with : select columns whose name start with a given chain of characters
- ends_with : select columns whose name end with a given chain of characters
- contains : select columns whose name contains a given chain of characters
- matches : select columns whose name fits a regular expression
- one_of : select columns whose name belongs to a given group

```
empl_sales = select(employees, contains("Sales"))
empl_id = select(employees, starts_with("Emp"))
```

Filter: The first argument is the data, the following conditions for the rows to retain and may be expressed using any of (>, <, >=, <=, !=, %in%).

```
empl_fil_emp = filter(employees, EmpID > 1005)
empl_fil_emp_sale = filter(employees, EmpID > 1005, nSales <= 10)
empl_fil_emp_sale = filter(employees, EmpID > 1005 & nSales <= 10) #commas = &
empl_fil_emp_or = filter(employees, EmpID == 1000 | EmpID == 1005)
empl_fil_ord = filter(employees, EmpID %in% c(1003, 1005))
```

Combining data with `dplyr`

The `cbind` and `rbind` functions may be used to add columns or rows to a dataset, but `dplyr` offers alternatives that work even if the combined elements do not have the same number of columns or rows.

- `inner_join(x, y, by=)`: returns observations for which x and y have a common key (by).
- `left_join` : returns all observations with a key in x whether of not it appears in y.
- `right_join` : returns all observations with a key in y whether of not it appears in x.
- `anti_join` : returns all observations with a key in x but no key in y.
- `full_join` : returns all pairs of observations whether there is a match or not between x and y.

Consider these illustrations:

```
emp_part1 = read.table("sales_part1.txt")
emp_part2 = read.table("sales_part2.txt")
emp_part3 = read.table("sales_part3.txt")
sales_comb=bind_rows(emp_part1, emp_part2)
```

```
sales_full=full_join(x=sales_comb, y=emp_part3, by=c("EmpID", "Month"))
sales_left=left_join(x=emp_part1, y=emp_part3, by=c("EmpID", "Month"))
sales_right=right_join(x=emp_part1, y=emp_part3, by=c("EmpID", "Month"))
```

Piping with `dplyr`

Consider now an example of piping with `dplyr`

```
empl_transform = employees %>%
  select(EmpID, TotSales, nSales) %>%
  filter(EmpID <= 1005) %>%
  group_by(EmpID) %>%
  summarise(TotSales = sum(TotSales), nSales = sum(nSales)) %>%
  mutate(Ratio = TotSales / nSales) %>%
  arrange(desc(Ratio))
```

The following operations are performed; the object created is a data frame.

- Consider the `employes` table,
- Retain columns EmpID, TotVentes et nVentes,
- Keep observations (rows) for employees 1000 to 1005,
- Aggregate data by employee number,
- Compute total sales and number of sles for each employee,
- Create a new column with a ratio,
- Sort the results uin descending ratios.

Other tools for data manipulation

Concatenation is the operation of adding two datasets one after the other. In R, both data need to have the same column in the same order.

```
emp_comb=rbind(emp_part1, data.frame(emp_part2, Team="B"))
```

There are options to merge tables as well (add new columns to existing rows). The `cbind` is able to glue together two tables with the same number of lines (the observations then need to be in the same order in both tables). In most case, a key may be used to identify which lines correspond to each other in multiple tables.

```
# No common key, data are sorted, each line correspond to the same individual
emp_comb = emp_comb[order(emp_comb$EmpID, emp_comb$Month),]
emp_part3 = emp_part3[order(emp_part3$EmpID, emp_part3$Month),]
sales_merge1 = cbind(emp_comb, emp_part3)
sales_merge1 = cbind(emp_comb, emp_part3$TotSales)  # Avoids repeating columns
```

The merge function is a safer and more flexible alternative:

```
sales_merge2 = merge(x=emp_comb, y=emp_part3, by=c("EmpID", "Month"), all=TRUE)
```

The argument `all`=TRUE allows an full join, which means that all observations are found in the output even if they do not appear in one of the datasets. In this case, it does not make a difference in the output since all individuals appear in both sets. The `merge` function allow multiple types of join:

```
sales_leftjoin=
    merge(x=emp_part1, y=emp_part3, by=c("EmpID", "Month"), all.x=TRUE)
sales_rightjoin=
    merge(x=emp_part1, y=emp_part3, by=c("EmpID", "Month"), all.y=TRUE)
# Cross join
tm = data.frame(Team=(c("A","B","C","D")))
employees_tm = merge(employees, tm, by=NULL)
```

Aggregation

A common task when analyzing data consists in calculating a statistic (mean, min, max, etc.) on groups of data based on a factor variable. The `aggregate` function makes such calculations easy to code.

```
aggregate(employees$nSales~employees$EmpID, FUN=sum, na.rm = TRUE)
aggregate(cbind(employees$nSales, employees$TotSales)~employees$EmpID,
    FUN=sum, na.rm = TRUE)
```

Using `cbind` allowed to compute the statistics on more than one variable at a time. If more than one aggregating variable is requires, a "+" sign may be used.

```
aggregate(employees$nSales~employees$EmpID+employees$Month, FUN=sum, na.rm = TRUE)
```

Subsetting

Previous examples chow how appropriate arguments in brackets may be used to select a subset of data. The subset function may also be used. It is even possible to select only some variables

```
subset(employees,EmpID==1000&Month<=5)
subset(employees,EmpID==1000&Month<=5, select=c(Month, nSales, TotSales))
```

RMarkdown / knitr / Sweave

R offers a number of options to produce high quality documents efficiently, including reports, dashboards and presentations. Many of those solutions generate LaTeX or html code automatically to produce documents that may contain images, text, R code as well as results generated with R code.

While *RMarkdown* is specifically developed for R Studio (it will not work from Rbase), other packages that it is built on, namely `knitr` and Sweave will work on other platforms.

For more details on RMarkdown, visit :

https://www.rstudio.com/wp-content/uploads/2016/03/rmarkdown-cheatsheet-2.0.pdf
https://www.rstudio.com/wp-content/uploads/2015/03/rmarkdown-reference.pdf

You may also find templates and try it on your own. Some templates should be available on the course website.

Exercices

1. Consider the questions in Part I and II and use the piping operator to simplify the solutions
2. Create an RMarkdown document from RStudio. Attempt to modify the default template that it creates.

Attribution: Parts of this document are adapted from course notes written by Sarah Legendre-Bilodeau.