Data Analytics for audit using R

Anil Goyal

2023-04-03

Contents

Preface

In today's digital age, data is abundant and ubiquitous, and its importance cannot be overstated. As a result, auditors must be equipped with the necessary skills to leverage this data and draw insights from it. The use of programming languages such as R in auditing has become increasingly popular due to their ability to analyze large volumes of data and produce accurate results.

This book aims to provide a comprehensive guide on data analytics in audit using R. It is designed to be a practical resource for auditors who wish to enhance their data analysis skills and learn how to apply them in real-world scenarios.

The book covers a wide range of topics, including data collection, data cleaning, data visualization, and statistical analysis. It also includes examples of how R can be used to analyze financial and operational data, and how it can be used to identify risks and anomalies.

The book is structured in a way that allows readers to gradually build their skills from basic concepts to advanced techniques. Each chapter includes practical examples, exercises, and case studies to reinforce the concepts covered. The book also includes tips and tricks to help readers optimize their workflow and avoid common pitfalls.

It is worth noting that this book assumes some basic knowledge of R programming. However, readers who are new to R can still benefit from this book by following the step-by-step examples and exercises.

Finally, I would like to express my gratitude to all the individuals who contributed to the creation of this book. Without their support and encouragement, this project would not have been possible.

I hope this book serves as a valuable resource for auditors looking to enhance their data analytics skills and ultimately improve the quality of their work.

Why read this book

Structure of the book

Software information and conventions

Packages **knitr** (Xie, 2015) and the **bookdown** (Xie, 2023) have been used to compile this book. My R session information is shown below:

```
xfun::session_info()
```

```
## R version 4.2.2 (2022-10-31 ucrt)
## Platform: x86_64-w64-mingw32/x64 (64-bit)
## Running under: Windows 10 x64 (build 22621)
##
## Locale:
##   LC_COLLATE=English_India.utf8
##   LC_CTYPE=English_India.utf8
##   LC_MONETARY=English_India.utf8
##   LC_NUMERIC=C
##   LC_TIME=English_India.utf8
##
## Package version:
##   base64enc_0.1.3   bookdown_0.33
##   bslib_0.4.2       cachem_1.0.7
##   cli_3.6.1         compiler_4.2.2
##   digest_0.6.31     ellipsis_0.3.2
##   evaluate_0.20     fastmap_1.1.1
##   fontawesome_0.5.0 fs_1.6.1
##   glue_1.6.2        graphics_4.2.2
##   grDevices_4.2.2   highr_0.10
##   htmltools_0.5.5   jquerylib_0.1.4
##   jsonlite_1.8.4    knitr_1.42
##   lifecycle_1.0.3   magrittr_2.0.3
##   memoise_2.0.1     methods_4.2.2
##   mime_0.12         R6_2.5.1
##   rappdirs_0.3.3    rlang_1.1.0
##   rmarkdown_2.21    rstudioapi_0.14
##   sass_0.4.5        stats_4.2.2
##   stringi_1.7.12    stringr_1.5.0
##   tinytex_0.44      tools_4.2.2
##   utils_4.2.2       vctrs_0.6.1
##   xfun_0.38         yaml_2.3.7
```

Package names are in bold text (e.g., **rmarkdown**), and inline code and filenames are formatted in a typewriter font (e.g., `knitr::knit('foo.Rmd')`). Function names are followed by parentheses (e.g., `bookdown::render_book()`).

Acknowledgments

A lot of people helped me when I was writing the book.

Anil Goyal

About the Author

Gearing up

0.1 Overview

R programming language is the extended version of the S programming language. John Chambers, the creator of the S programming language in 1976 at Bell laboratories. In 1988, the official version of the S language came into existence with the name S-PLUS. The R language is almost the unchanged version of S-PLUS.

In 1991, R was created by **Ross Ihaka** and **Robert Gentleman** in the Department of Statistics at the University of Auckland. Ross's and Robert's experience developing R is documented in a 1996 paper in the Journal of Computational and Graphical Statistics (**?**). In 1997 the R Core Group was formed, containing some people associated with S and S-PLUS. Currently, the core group controls the source code for R and is solely able to check in changes to the main R source tree. Finally, in 2000 R version 1.0.0 was released to the public.

0.2 Why R?

R programming language is an open-source programming language for statistical computation. It supports n number of statistical analysis techniques, machine learning models, and graphical visualization for data analysis. It serves the purpose of the free software environment for statistical computation and graphics. R is easy to understand and implement. The packages are available to create an effective R program, data models, and graphical charts. For research and analytics purposes, it is a popular language among statisticians and data scientists.

0.3 Download and installation

The R programming language for your local computer can be downloaded from web portal of **The Comprehensive R Archive Network**, in short mostly

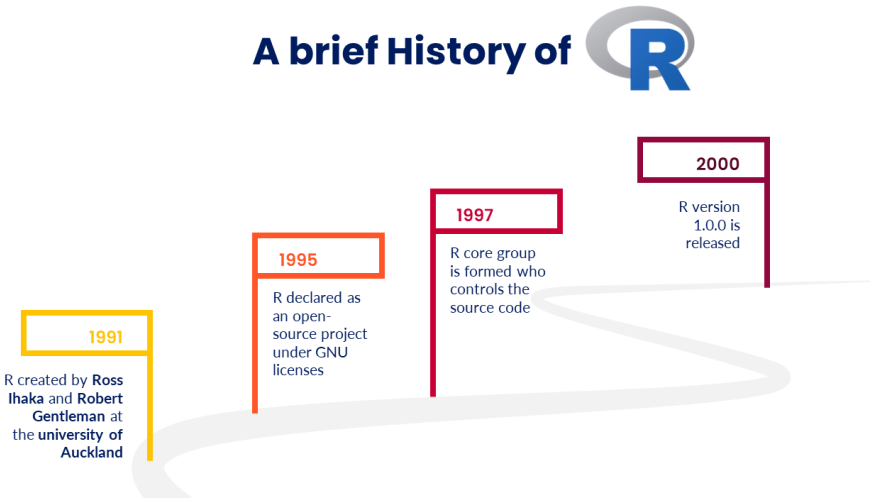


Figure 1: A Brief History of R

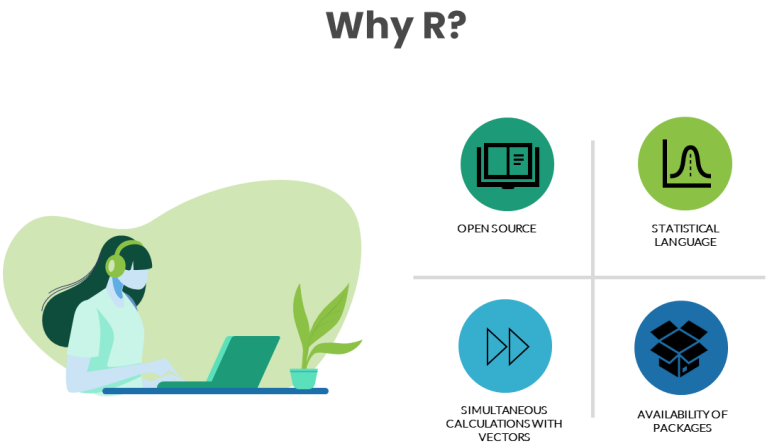


Figure 2: Why R

referred to as **CRAN**, which is a network of ftp and web servers around the world that store identical, up-to-date, versions of code and documentation for R. The portal address is https://cran.r-project.org/ -

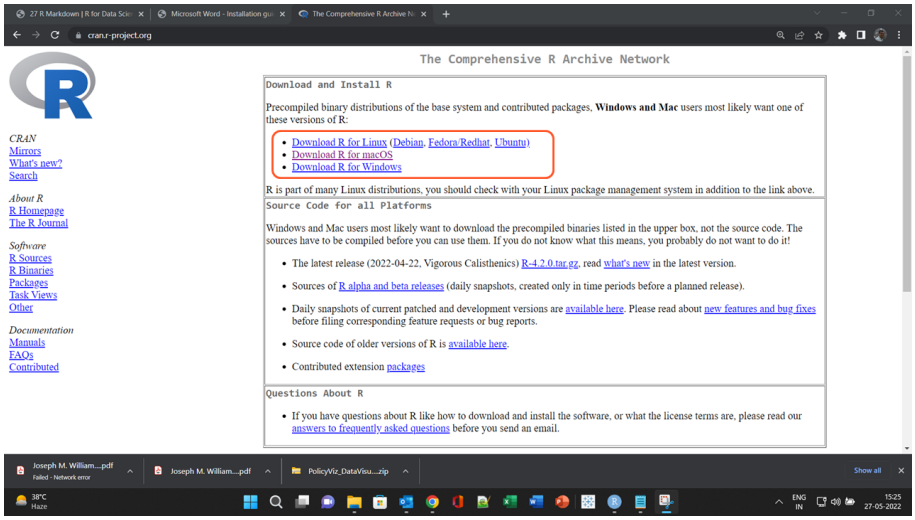


Figure 3: CRAN Portal

Download the specific (as per the operating system) file from the port and install it following the instructions. The R programming interface looks like-

0.4 Writing your first code

Writing code in R is pretty easy. Just type the command in front of `>`, as shown in figure 4 prompt and press enter(Return) key. R will display the results in next line.

Remember -

1. R is case sensitive. This will have to be remebered while writing/storing/calling functions or other objects. So `Anil`, `ANIL`, `anil` all are different objects in R.
2. White spaces between different pieces of codes don't matter. See figure-5 above. Both `3+4` and `3 + 4` will evaluate same.
3. Parenthesis `()` are generally used to change the natural order of precedence. Moreover, these are also used in passing arguments to functions, which will be discussed in detail in chapter-3 and onwards.
4. Multi-line code(s) aren't required to be indented in R. In R, indents have no meaning. However, following best practices to write a code that is

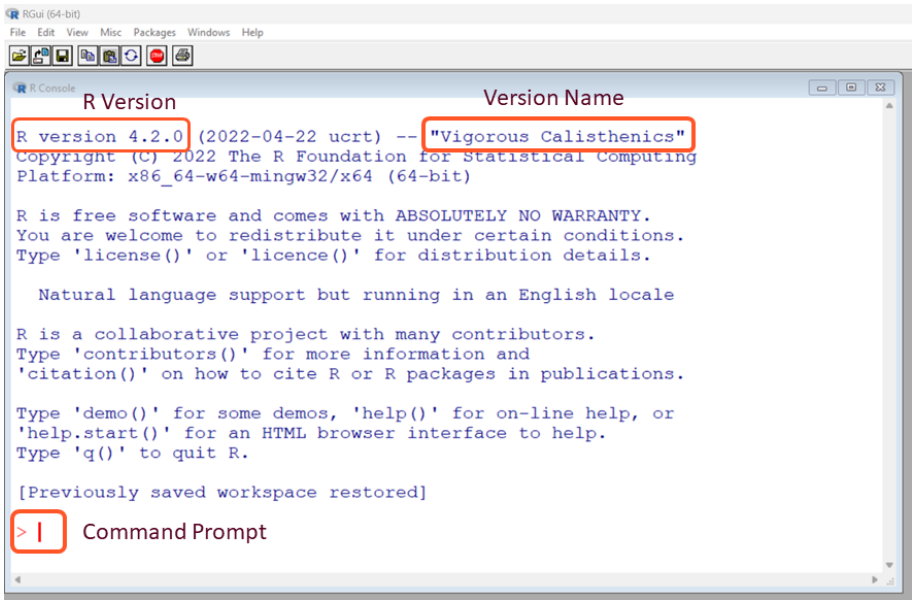


Figure 4: R Workspace

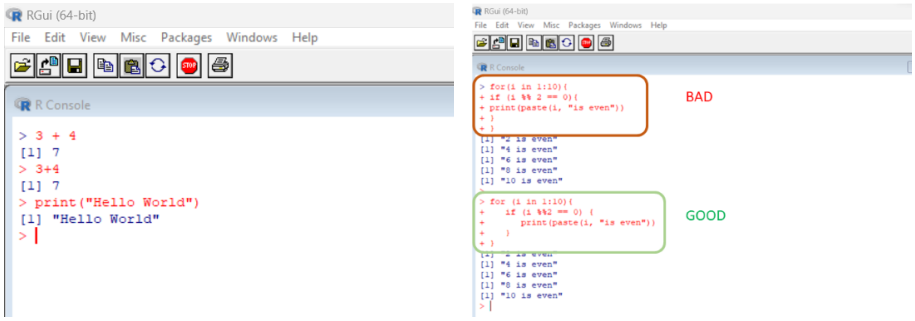


Figure 5: Left - Writing first Code in R; Right - Indenting code not necessary but recommended

understandable by readers, proper indentation is suggested. See figure-5 (right) above.

5. If an incomplete code is written in the first line of the code (useful when a single line is not sufficient to write complete code), R will automatically prompt as displaying `+` at the beginning of line, instead of a `>`. See figure-5 (right) above.
6. Indices in R always start from 1 (and not from 0). This has been discussed in detail in chapter-2.
7. Code that start with hash symbol `#` do not execute. Even in a line if `#` appears in between the line, the code from that place does not get executed. See the following example. Comments may be used in codes for either of the purposes -
   - Code Readability
   - Explanation of code
   - Inclusion of metadata, other references, etc.
   - Prevent execution of certain line of code

```
# 1 + 3 (this won't be executed)
1 + 3 # +5
```

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

Tip: to clear the workspace, just click `ctrl` + `l` together.

Normally R code files have an extension `.R` but other R files may have other extensions, such as project files `.Rproj`, markdown files `.Rmd`, and many more.

All of the programming/code writing may be done in R. But you may have noticed that code once executed cannot be edited. The code has to written again (Tip: To get previous executed command just use scroll up key on keyboard). Thus, in order to use many other smart features, we will write our code as R scripts i.e. in `.R` files using most popular IDE for R which is `R Studio`.

> Using R studio IDE is so much popular that many persons using R, do not distinguish between R and its IDE. Even Stack Overflow which is a popular forum to seek online help explicitly asks users not to tag 'R studio' in general R code problems[1].

0.5 R studio IDE

RStudio is free and open source IDE (Integrated Development Environment) for R, which is available for Windows, Mac OS and LINUX. It can be downloaded

[1] https://stackoverflow.com/tags/rstudio/info

from its portal https://www.rstudio.com. For our most of the data analytics needs, we require Rstudio desktop version, which is available for free to download and installation.

It includes a console, syntax-highlighting editor that supports direct code execution, and a variety of robust tools for plotting, viewing history, debugging and managing your work-space. After downloading and installing it the local machine, a work-space/UI similar to that shown in following figure, is opened.

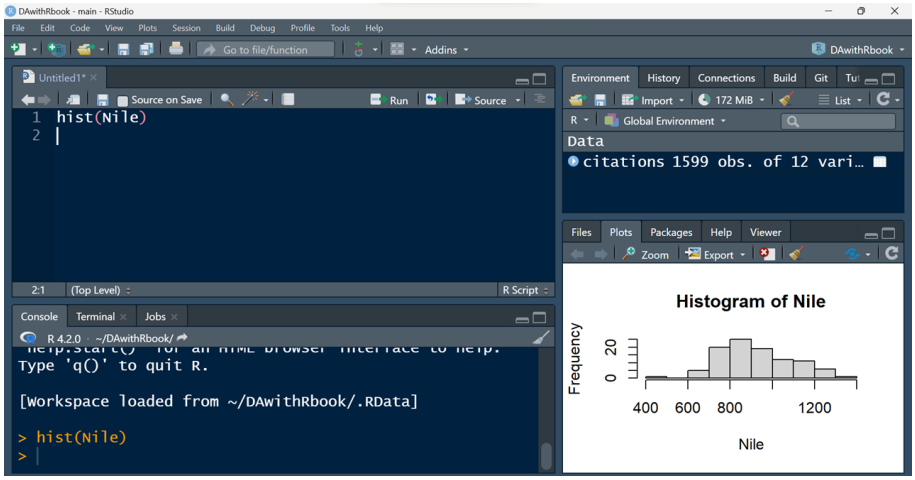
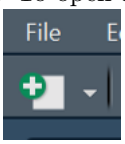


Figure 6: R Studio interface

There are four panels

- Top-left:
  - **scripts and files:** The script files which we will be working on, will be opened and displayed here. To open a new script, you just need to click the new script button 

which is just below the *file menu.*
- Bottom-left:
  - **R console:** is where the R commands can be written and see the output. Even the commands run on script will show the output in this panel.
  - **Terminal:**
- Top-right:
  - **Environment:** To see the objects saved in current environment

  - **history** To view the history of commands run, in the current session
  - **Connections:** Used to connect/import with external database/data

- Bottom-right:
  - **tree of folders**, to see the file structure of current working directory
  - **graph window**, if the out of r command is a plot/graph, it will be generated here
  - **packages**, to download and load the external packages using mouse click
  - **help window**, to get help on desired functions. Even the help sought through r command will be displayed in this window.
  - **viewer:** can be used to view local web content.

0.6 Packages and libraries

One of the strength of R is that numerous user-written packages (or *libraries*) are available on **Comprehensive R Archive Network** i.e. CRAN. Package installation is perhaps easiest of the jobs in R.

The command is fairly simple -

```
install.packages("library_name")
```

which downloads the given package name (to be given in quotes and is case-sensitive), compiles it and then load it into the specified/default directory. This will however, not load into the memory. The library once downloaded need not be downloaded every time but need to be loaded every time using the command-

```
library(library_name)
```

Quotes here are optional but package name is still case sensitive. So to install and load `tidyverse` we need to run first command once (which will download the package into your local computer) but second command (to load it in the current R session) at every new session.

```
install.packages('tidyverse')
library(tidyverse)
```

0.6.1 Double Colon operator ::

In R, we can use double colon operator i.e. `::` to access functions that are defined as part of the internal functions that a package uses. These may be used in at least two cases-

1. To call a function say `filter` from package `dplyr` we may use `dplyr::filter()` without actually loading it.

2. In cases of conflicts (e.g. when two or more packages have same function names) if we want to use the function specifically from a package. E.g. While loading `dplyr` the function `filter` masks the function with same available in `stats` package (a part of base R). So, if the requirement is to use function `filter` from `stats` we can use `stats::filter()`.

0.7 Getting Help within R

Once R is installed, there is a comprehensive built-in help system. We can use any of the following commands-

```
help.start()   # general help
help(foo)      # help about function `foo`
?foo           # same as above
apropos("foo") # show all functions containing word `foo`
example(foo)   # show an example of function `foo`
```

Alternatively, features under the Help menu or help pane, can also be used.

0.8 TIDYVERSE

The tidyverse is a *package of packages* that work in harmony because they share common data representations and 'API' design. This package is designed to make all these easy to install and load multiple 'tidyverse' packages in a single step.

Though `tidyverse` is a collection 20+ packages (in fact 80+ packages will be installed including depended packages) which are all installed by `install.packages("tidyverse")` command, yet `library(tidyverse)` load eight of them. Others (like `lubridate`) will have to loaded explicitly.

1. **ggplot2** is a system for declaratively creating graphics, based on *The Grammar of Graphics*.
2. **dplyr** provides a grammar of data manipulation, providing a consistent set of verbs that solve the most common data manipulation challenges.
3. **tidyr** provides a set of functions useful for data transformation.
4. **readr** is used to read and write rectangular/tabular data formats.
5. **purrr** is functional programming (FP) toolkit for working with functions and vectors.

6. **tibble** provides functionalities related to displaying data frames.
7. **stringr** provides set of functions designed to work with strings. It is built on top of another package stringi.
8. **forcats** provides a suite of useful tools that solve common problems with factors.

There are several other `tidyverse` packages which we will be working with-

- `lubridate`
- `hms`
- `readxl`

Part-I: Basic R Programming Concepts

Chapter 1

R Programming Language

1.1 Use R as a calculator

To start learning R, just start entering equations directly at the command prompt `>` and press enter. So, `3+4` will give you result `7`. Common mathematical operators are listed in table 1.1.

Table 1.1: Common Mathematical Operators in R

Operator/ function	Meaning	Example
`+`	Addition	`4+5` is `9`
`-`	Substraction	`4-5` is `-1`
`*`	Multiplication	`4*5` is `20`
`/`	Division	`4/5` is `0.8`
`^`	Exponent	`2^4` is `16`
`%%`	Modulus (Remainder from division)	`15 %% 12` is `3`
`%/%`	Integer Division	`15 %/% 12` is `1`

Strings or Characters have to be enclosed in single or double quotes (more on strings in section 1.3.4). So a few examples of calculations that can be performed in R could be-

```
4 + 3 ^ 2
```

```
## [1] 13
```

```
8 * (9 + 4)
```

```
## [1] 104
```

> Note that R follows common mathematical order of precedence while evalauting expressions. That may be changed using simple parenthesis i.e. `()`. Also note that other brackets/braces i.e. curly braces `{}` and `[]` have been assigned different meaning, so to change nested order of operations only `()` may be used.

1.2 Basic Concepts

1.2.1 Object Assignment

R is an object-oriented language.(R Core Team, 2022) This means that *objects* are created and stored in R environment so that they can be used later.

So what is an object? An object can be something as simple as a number (value) that can be assigned to a variable. Think of it like this; Suppose we have greet each user by his/her name prefixing *hello* to his/her name. Now user's name may be saved in our work environment for later use. Thus, once the user name is saved in a variable then can be retrieved later on, by calling the variable name instead of asking the user name again and again. An object can be also be a data-set or complex model output or some function. Thus, an object created in R can hold multiple values.

The other important thing about objects is that objects are created in R, using the assignment operator `<-`. Use of equals sign `=` to set something as an object is not recommended thought it will work properly in some cases. For now we will stick with the assignment operator, and interpret it as the left side is the object name that is storing the object information specified on the right side. *If `->` right hand side assignment is used, needless to say things mentioned above will interchange.*

```
# user name
user_name <- 'Anil Goyal'

# when the above variable is called
user_name
```

```
## [1] "Anil Goyal"
```

> **Case sensitive nature:** Names of variables even all objects in R are case sensitive, and thus `user`, `USER` and `useR`; all are different variables.

1.3 Atomic data types in R

We have seen that objects in R can be created to store some values/data. Even these objects can contain other objects as well. So a question arises, what is the most atomic/basic data type in R. By atomic we mean that the object cannot be split any further. Thus, the atomic objects created in R can be thought of variables holding one single value. E.g. user's name, user's age, etc. Now atomic objects created in R can be of six types -

- logical (or Boolean i.e. TRUE FALSE etc.)
- integer (having non-decimal numeric values like 0, 1, etc.)
- double (or floating decimal type i.e. having numeric values in decimal i.e. 1.0 or 5.25, etc.)
- character (or string data type having some alphanumeric value)
- complex (numbers having both real and imaginary parts e.g. 1+1i)
- raw (not discussed here)

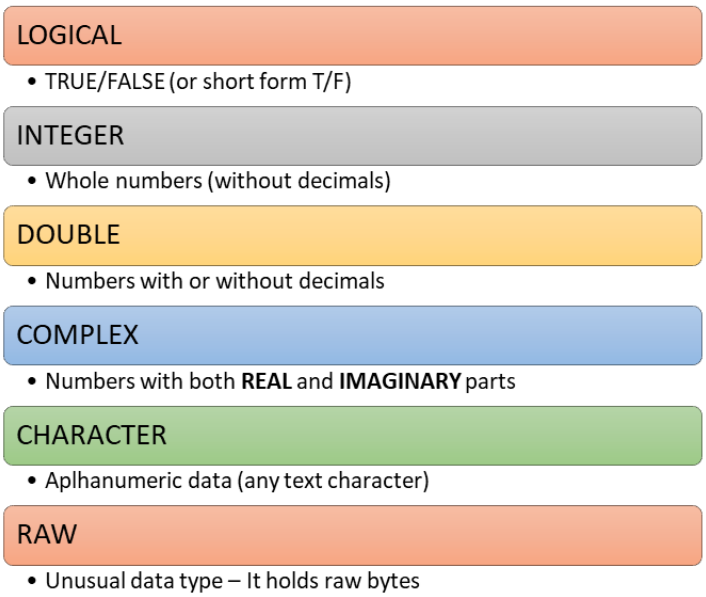


Figure 1.1: Data types in R

Let us discuss all of these.

Note: We will use a pre-built function `typeof()` to check the type of given value/variable. However, functions as such will be discussed later-on.

1.3.1 Logical

In R logical values are stored as either `TRUE` or `FALSE` (all in caps)

```
TRUE
```

```
## [1] TRUE
```

```
typeof(TRUE)
```

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

```
my_val <- TRUE
typeof(my_val)
```

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

```
NA
```

There is one special type of logical value i.e. `NA` (short for *Not Available*). This is used for missing data. Remember missing data is not empty string. The difference between two is explained in section 1.3.4.

1.3.2 Integer

Numeric values can either be integer (i.e. without a floating point decimal) or with a floating decimal value (called `double` in r). Now integers in R are differentiated by a suffix `L`. E.g.

```
my_val1 <- 2L
typeof(my_val1)
```

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

```
typeof(2)
```

```
## [1] "double"
```

1.3.3 Double

Numeric values with decimals are stored in objects of type `double`. It should be kept in mind that if storing an integer value directly to a variable, suffix `L` must be used otherwise the object will be stored as `double` type as shown in above example.

In double type, exponential formats or hexadecimal formats to store these numerals may also be used.

```
my_val2 <- 2.5
my_val3 <- 1.23e4
my_val4 <- 0xcafe # hexadecimal format (prefixed by 0x)

typeof(my_val2)
```

```
## [1] "double"
```

```
typeof(my_val3)
```

```
## [1] "double"
```

```
typeof(my_val4)
```

```
## [1] "double"
```

> Note: Suffix `L` may also be used with numerals in hexadecimal (e.g. `0xcafeL`) or exponential formats (e.g. `1.23e4L`), which will coerce these numerals in `integer` format.

```
typeof(0xcafeL)
```

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

Thus, both `integer` and `double` data types may be understood in R as having sub-types of `numeric` data. There are three other types of special numerals (specifically doubles) `Inf`, `-Inf` and `NaN`. First two are infinity (positive and negative) and last one denotes indefinite number (`NaN` short for *Not a Number*).

```
1/0
```

```
## [1] Inf
```

```
-45/0
```

```
## [1] -Inf
```

```
0/0
```

```
## [1] NaN
```

1.3.4 Character

Strings are stored in R as character type. Strings should either be surrounded by single quotes `''` or double quotes `""`[1].

```
my_val5 <- 'Anil Goyal'
my_val6 <- "Anil Goyal"
my_val7 <- "" # empty string
my_missing_val <- NA # missing value

typeof(my_val5)
```

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

```
typeof(my_val6)
```

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

```
typeof(my_val7)
```

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

```
typeof(my_missing_val)
```

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

[1] Single and double quotes can be used interchangeably and won't have any difference in the objects created using any of these. Only thing that should be kept in mind is that the quote used to start the string must be used to close the string, otherwise an error may be thrown. However, this may be used to store objects with either type of string in the data itself.

Notes:
1. Though `NA` is basically of type logical yet it will be used to store missing values in any other data type also as shown in subsequent chapter(s). 2. Special characters are escaped with `\`; Type `?Quotes` in console and check documentation for full details. 3. A simple use of `\` escape character may be to use `"` or `'` within these quotes. Check Example-3 below.

Example-1: Usage of double and single quote interchangeably.

```
my_val8 <- "R's book"
my_val8
```

```
## [1] "R's book"
```

Example-2: Usage of escape character.

```
cat("This is first line.\nThis is new line")
```

```
## This is first line.
## This is new line
```

Example-3: Usage of escape character to store single/double quotes as string themselves.

```
cat("\' is single quote and \" is double quote")
```

```
## ' is single quote and " is double quote
```

Note: If absence of indices has been noticed in above code output, learn more about `cat` function here.

1.3.5 NULL

`NULL` (note: all caps) is a specific data type used to create an empty vector. Even this `NULL` can be used as a vector in itself.

```
typeof(NULL)
```

```
## [1] "NULL"
```

```
vec <- 1:5
vec
```

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

```
vec <- NULL
vec
```

```
## NULL
```

1.3.6 Complex

Complex numbers are made up of real and imaginary parts. As these will not be used in the data analysis tasks, it is not discussed in detail here.

```
my_complex_no <- 1+1i
typeof(my_complex_no)
```

```
## [1] "complex"
```

1.4 Data structures/Object Types in R

Objects in R can be either homogeneous or heterogeneous.

Figure 1.2: Objects/Data structures in R, can either be homogeneous (left) or heterogeneous (right)

Homogeneous objects

1.4.1 Vectors

What is a vector? A vector is simply a collection of values/data of same type.

Figure 1.3: Vectors are homegeneous data structures in R

1.4.1.1 Simple vectors (Unnamed vectors)

Though, `Vector` is the most atomic data type used in R, yet it can hold multiple values (of same type) simultaneously. In fact vector is a collection of multiple values of same type. So why vector is atomic when it can hold multiple values? You may have noticed a `[1]` printed at the start of line of output whenever a variable was called/printed. This `[1]` actually is the index of that element. Thus, in R instead of having *scalar(s)* as most atomic type, we have *vector(s)* containing only one element. Whenever a vector is called all the values stored in it are displayed with its index at the start of each new line only.

Even processing of multiple values simultaneously, stored in a vector, to produce a desired output, is one of the most powerful strengths of R. The three variables shown in the figure below, all are vectors.

How to create a vector? Vectors in R are created using either -

- `c()` function which is shortest and most commonly used function in r. The elements are concatenated (and hence the shortcut `c` for this function) using a comma `,` ; *OR*
- `vector()` where one argument namely `length` specifies the number of elements therein.

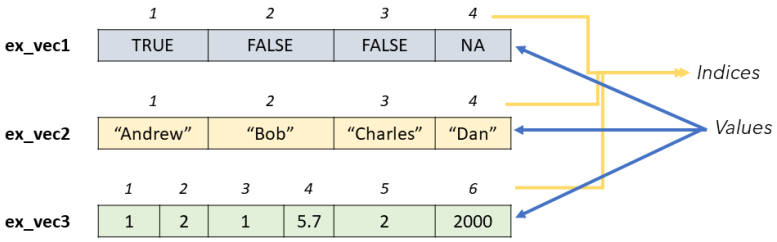


Figure 1.4: Examples of Vectors

```
my_vector <- c(1, 2, 3)
my_vector
```

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

```
my_vector2 <- vector(mode = 'integer', length = 15)
my_vector2
```

```
##  [1] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

Function `c()` can also be used to **join two or more vectors**.

```
vec1 <- c(1, 2)
vec2 <- c(11, 12)
vec3 <- c(vec1, vec2)
vec3
```

```
## [1]  1  2 11 12
```

Useful Functions to create new vectors

There are some more useful functions to create new vectors in R, which we should discuss here as we will be using these vectors in subsequent chapters.

Generate integer sequences with Colon Operator :

This function generates a sequence from the number preceding `:` to next specified number, in arithmetical difference of `1` or `-1` as the case may be. Notice that output vector type is of `integer`.

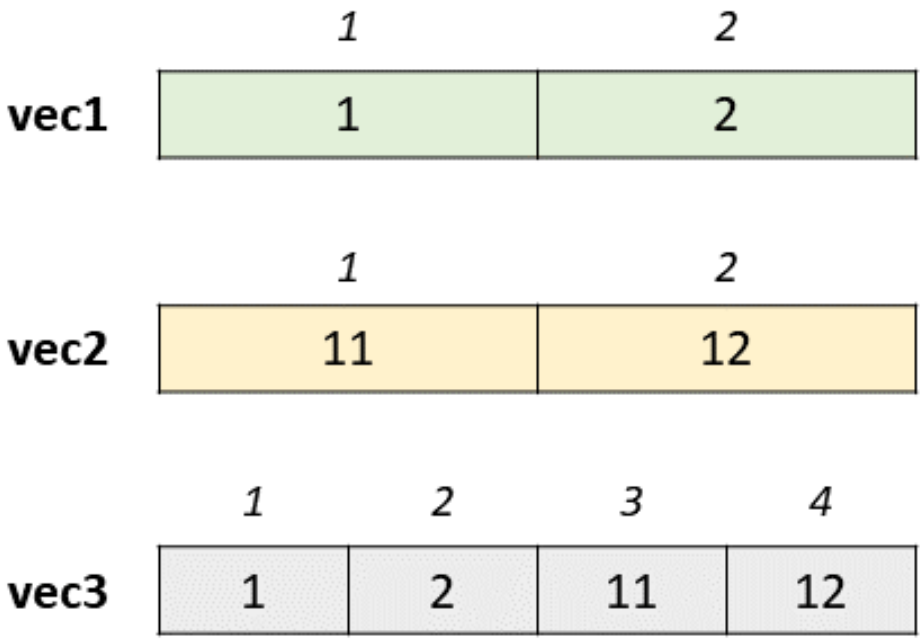


Figure 1.5: Vector Concatenation

```
1:25
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17
## [18] 18 19 20 21 22 23 24 25
```

```
25:30
```

```
## [1] 25 26 27 28 29 30
```

```
10:1
```

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

```
typeof(2:250)
```

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

> Note: One of the common mistakes with colon operator is assuming its **operator precedence**. In R, colon operator has calculation precedence over any mathematical operator. Think of outputs you may get with these-

```
n <- 5
1:n+1
1:n*2
```

Generate specific sequences with function `seq`

This function generates a sequence from a given number to another number, similar to `:`, but it gives us more control over the output desired. We can provide the difference specifically (`double` type also) in the `by` argument. Otherwise if `length.out` argument is provided it calculates the difference automatically.

```
seq(1, 5, by = 0.3)
```

```
##  [1] 1.0 1.3 1.6 1.9 2.2 2.5 2.8 3.1 3.4 3.7 4.0 4.3
## [13] 4.6 4.9
```

```
seq(1, 2, length.out = 11)
```

```
##  [1] 1.0 1.1 1.2 1.3 1.4 1.5 1.6 1.7 1.8 1.9 2.0
```

Repeat a pattern/vector with function `rep`

As the name suggests `rep` is short for *repeat* and thus it repeat a given element, a given number of times.

```
rep('repeat this', 5)
```

```
## [1] "repeat this" "repeat this" "repeat this"
## [4] "repeat this" "repeat this"
```

```
# We can even repeat already created vectors
vec <- c(1, 10)
rep(vec, 5)
```

```
##  [1]  1 10  1 10  1 10  1 10  1 10
```

```
rep(vec, each = 5) # notice the difference in results
```

```
##  [1]  1  1  1  1  1 10 10 10 10 10
```

Generate english alphabet with `LETTERS` / `letters`

These are two inbuilt vectors in R having all 26 alphabets in upper and lower cases respectively.

```
LETTERS
```

```
##  [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L"
## [13] "M" "N" "O" "P" "Q" "R" "S" "T" "U" "V" "W" "X"
## [25] "Y" "Z"
```

```
letters
```

```
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l"
## [13] "m" "n" "o" "p" "q" "r" "s" "t" "u" "v" "w" "x"
## [25] "y" "z"
```

Generate gregorian calendar month names with `month.name` / `month.abb`

```
month.name
```

```
##  [1] "January"   "February"  "March"     "April"
##  [5] "May"       "June"      "July"      "August"
##  [9] "September" "October"   "November"  "December"
```

```
month.abb
```

```
##  [1] "Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul" "Aug"
##  [9] "Sep" "Oct" "Nov" "Dec"
```

1.4.1.2 Named Vectors

Vectors in R, can be named also, i.e. where each of the element has a name. E.g.

```
ages <- c(A = 10, B = 20, C = 15)
ages
```

```
##  A  B  C
## 10 20 15
```

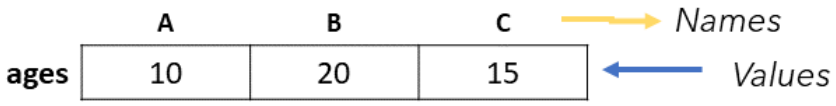


Figure 1.6: Vector elements can have names

Note here that while assigning names to each element, the names are not enclosed in quotes similar to variable assignment. Also notice that this time R has not printed the numeric indices/index of first element (on each new line). There are other ways to assign names to an existing vector. We can use `names()` function, which displays the names of all elements in that vector (*and this time in quotes as these are displayed in a vector*).

```
names(ages)
```

```
## [1] "A" "B" "C"
```

Using this function we can assign names to existing vector. See

```
vec1
```

```
## [1] 1 2
```

```
names(vec1) <- c('first_element', 'second_element')
vec1
```

```
##  first_element second_element
##              1              2
```

Names may also be assigned using `setNames()` while creating the vector simultaneously.

```
new_vec <- setNames(1:26, LETTERS)
new_vec
```

```
##  A  B  C  D  E  F  G  H  I  J  K  L  M  N  O  P  Q  R
##  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18
##  S  T  U  V  W  X  Y  Z
## 19 20 21 22 23 24 25 26
```

Function `unname()` may be used to remove all names. Even all the names can be removed by assigning `NULL` to `names` of that vector. Also remember that `unname` does not modify vector in place. To have this change we will have to assigned unnamed vector to that vector again. Check this,

```
unname(new_vec)
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17
## [18] 18 19 20 21 22 23 24 25 26
```

```
new_vec
```

```
##  A  B  C  D  E  F  G  H  I  J  K  L  M  N  O  P  Q  R
##  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18
##  S  T  U  V  W  X  Y  Z
## 19 20 21 22 23 24 25 26
```

```
new_vec <- unname(new_vec)
new_vec
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17
## [18] 18 19 20 21 22 23 24 25 26
```

Type coercion

There are occasions when different classes of R objects get mixed together. Sometimes this happens by accident but it can also happen on purpose. Let us deal with each of these.

But prior to this let us learn how to check the type of a vector. Of course we can check the type of any vector using function `typeof()` but what if we want to check whether any vector is of a specific type. So there are `is.*()` functions to check this, and all these functions return either `TRUE` or `FALSE`.

- `is.logical()`
- `is.integer()`
- `is.double()`
- `is.character()`
- `is.complex()`

```
is.integer(1:10)
```

```
## [1] TRUE
```

```
is.logical(LETTERS)
```

```
## [1] FALSE
```

Implicit Coercion

As already stated, vector is the most atomic data object in R. Even all the elements of a vector (having multiple elements) are vectors in themselves. We have also discussed that vectors are homogeneous in types. So what happens when we try to mix elements of different types in a vector.

In fact when we try to mix elements of different types in a vector, the resultant vector is coerced to the type which is most feasible. Since a numeral say `56` can easily be converted into a complex number (`56+0i`) or character (`"56"`), but alphabet say `A`, cannot be converted into a numeral, the atomic data types normally follow the order of precedence, tabulated in table 1.2.

Table 1.2: Order of Precedence for Atomic Data Types

Rank	Type
1	Character
2	Complex
3	Double
4	Integer
5	Logical

For e.g. in the following diagram, notice all individual elements in first vector. Out of the types of all elements therein, character type is having highest rank and thus resultant vector will be silently coerced to a character vector. Similarly, second and third vectors are coerced to `double` (second element) and `integer` (first element) respectively.

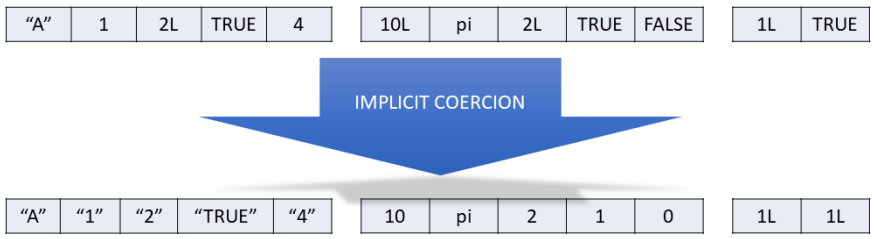


Figure 1.7: Implicit Coercion of Vectors

It is also important to note here that this implicit coercion is without any warning and is silently performed. This implicit coercion is also carried out when two (or more) vectors having different data types are concatenated together.

Example- `vec` is an existing vector of type `integer`. When we try to add an extra element say of `character` type, `vec` type is coerced to `character`.

```
vec <- 1:5
typeof(vec)
```

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

```
vec <- append(vec, 'ABCD')
typeof(vec)
```

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

R also implicitly coerces vectors to appropriate type when we try to perform calculations on vectors of other types. Example

```
(TRUE == FALSE) + 1
```

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

```
typeof(TRUE + 1:100)
```

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

```
typeof(FALSE + 56)
```

```
## [1] "double"
```

Explicit Coercion

We can explicitly coerce by using an `as.*()` function, like `as.logical()`, `as.integer()`, `as.double()`, or `as.character()`. **Failed coercion of strings generates a warning and a missing value:**

```
as.double(c(TRUE, FALSE))
```

```
## [1] 1 0
```

```
as.integer(c(1, 'one', 1L))
```

```
## Warning: NAs introduced by coercion
```

```
## [1]  1 NA  1
```

Checking dimensions

Now a vector can have `n` number of vectors (recall that each element is a vector in itself) and at times we may need to check how many elements a given vector contains. Using function `length()`, we can check the number of elements.

```
length(1:100)
```

```
## [1] 100
```

```
length(LETTERS)
```

```
## [1] 26
```

```
length('LENGTH') # If you thought its output should have been 6, check again.
```

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

1.4.2 Matrix (Matrices)

Matrix (or plural matrices) is a two dimensional arrangement (similar to a matrix in linear algebra and hence its name) of elements of again same type as in vectors. E.g.

$$\begin{matrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \end{matrix}$$

Thus, matrices are vectors with an attribute named *dimension*.

> The dimension attribute is itself an integer vector of length 2 (number of rows, number of columns).

Create a new matrix

A new matrix can be created using function `matrix()` where a vector is given which is to be converted into a matrix and either number of rows `nrow` or number of columns `ncol` may be given.

```
matrix(1:12, nrow = 3)
```

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

```
matrix(1:12, ncol=3)
```

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

Another useful argument is `byrow` which by default is `FALSE`. So if it is explicitly changed, we get

```
matrix(1:12, ncol=3, byrow = TRUE)
```

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

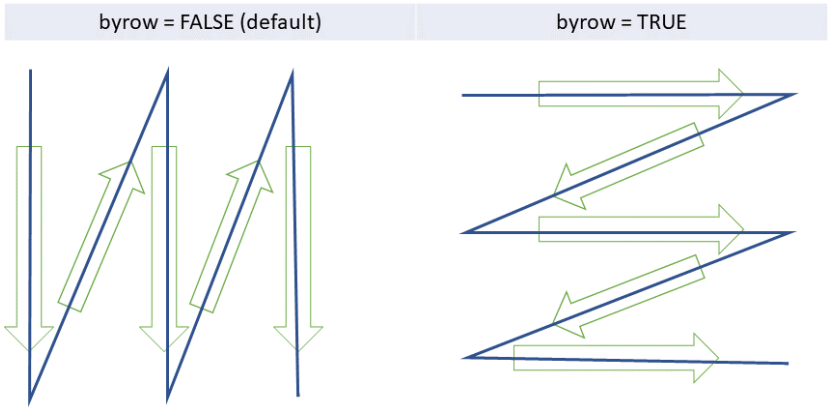


Figure 1.8: Arrangement of Matrix, if byrow argument is used

Matrix can be of any type. But rules of explicit and implicit coercion (as explained in vectors) also apply here.

```
matrix(LETTERS, nrow = 2)
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]
## [1,] "A"  "C"  "E"  "G"  "I"  "K"  "M"  "O"  "Q"
## [2,] "B"  "D"  "F"  "H"  "J"  "L"  "N"  "P"  "R"
##      [,10] [,11] [,12] [,13]
## [1,] "S"   "U"   "W"   "Y"
## [2,] "T"   "V"   "X"   "Z"
```

```
matrix(c(LETTERS, 1:4), nrow=5)
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,] "A"  "F"  "K"  "P"  "U"  "Z"
## [2,] "B"  "G"  "L"  "Q"  "V"  "1"
## [3,] "C"  "H"  "M"  "R"  "W"  "2"
## [4,] "D"  "I"  "N"  "S"  "X"  "3"
## [5,] "E"  "J"  "O"  "T"  "Y"  "4"
```

Names in matrices

Similar to vectors, rows or columns or both in matrices may have names. Check `?matrix()` for complete documentation.

Dimension

To check dimension of a matrix we can use `dim()` (short for dimension) (similar to `length` in case of vectors) which will return a vector with two numbers (rows first, followed by columns).

```
my_mat <- matrix(c(LETTERS, 1:4), nrow=5)
dim(my_mat)
```

```
## [1] 5 6
```

This gives us another method to create matrix from a vector. See

```
my_mat2 <- 1:10
dim(my_mat2) <- c(2,5)
my_mat2
```

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

Have a check on replication

What happens when product of given dimensions is less than or greater than given vector to be converted. It replicates but it is advised to check these properly as resultant vector may not be as desired. Check these cases, and notice when R gives result silently and when with a warning.

```
matrix(1:10, nrow=5, ncol=5)
```

```
## Warning in matrix(1:10, nrow = 5, ncol = 5): data
## length differs from size of matrix: [10 != 5 x 5]
```

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

```
matrix(1:1000, nrow=2, ncol=3)
```

```
## Warning in matrix(1:1000, nrow = 2, ncol = 3): data
## length [1000] is not a sub-multiple or multiple of the
## number of columns [3]
```

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

Combining matrices

Using `cbind()` or `rbind()` we can combine two matrices column-wise or row-wise respectively.

See these two examples.

```
mat1 <- matrix(1:4, nrow = 2)
mat2 <- matrix(5:8, nrow = 2)
cbind(mat1, mat2)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    3    5    7
## [2,]    2    4    6    8
```

Example-2

```
rbind(mat1, mat2)
```

```
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
## [3,]    5    7
## [4,]    6    8
```

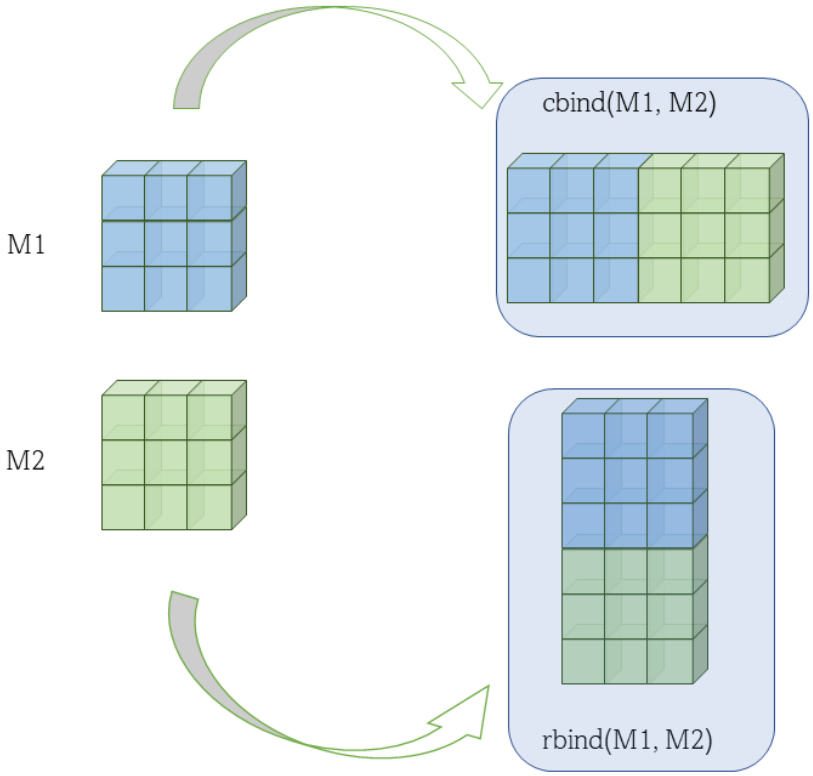


Figure 1.9: Binding of Two or more matrices together

1.4.3 Arrays

Till now we have seen that elements in one dimension are represented as vectors and in two dimension as matrices. So a question arises here, how many dimensions we can have. Actually we can have n number of dimensions in r, in object type `array`, but they'll become increasingly difficult to comprehend and are not thus discussed here. Check these however for your understanding,

```
array(1:24, dim = c(3,2,4)) # a three dimensional array
```

```
## , , 1
##
##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6
##
## , , 2
##
##      [,1] [,2]
## [1,]    7   10
## [2,]    8   11
## [3,]    9   12
##
## , , 3
##
##      [,1] [,2]
## [1,]   13   16
## [2,]   14   17
## [3,]   15   18
##
## , , 4
##
##      [,1] [,2]
## [1,]   19   22
## [2,]   20   23
## [3,]   21   24
```

Try creating 4 or 5 dimensional arrays in your console and see the results.

Further properties of vectors, matrices will be discussed in next chapter on subsetting and indexing where we will learn how to retrieve specific elements of vector/matrices/etc. But till now we have created objects which have elements of same type. What if we want to have different types of elements/data retaining their types, together in a single variable? Answer is in next section, where we will discuss hetergeneous objects.

Heterogeneous objects

1.4.4 Lists

So lists are used when we want to combine elements of different types together. Function used to create a list is `list()`. Check this

```
list(1, 2, 3, 'My string', TRUE)
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] 2
##
## [[3]]
## [1] 3
##
## [[4]]
## [1] "My string"
##
## [[5]]
## [1] TRUE
```

Pictorially this list can be depicted as

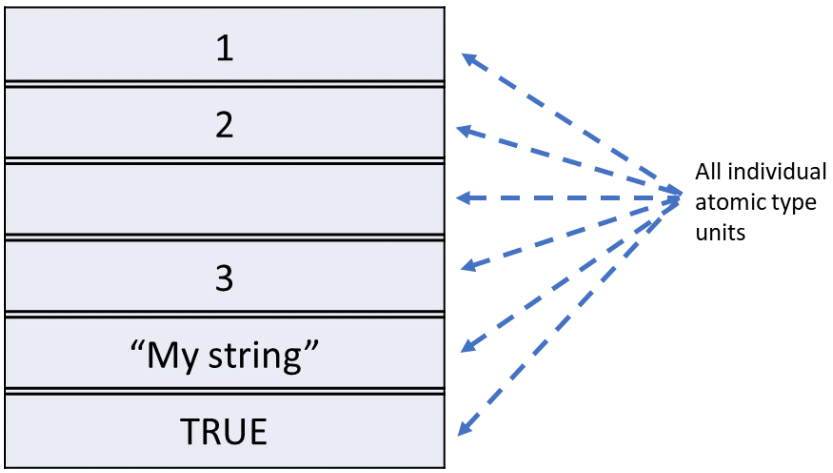


Figure 1.10: A list in R is a heterogeneous object

Interestingly list can contain vectors, matrices, arrays as individual elements. See

```
list(1:3, LETTERS, TRUE, my_mat2)
```

```
## [[1]]
## [1] 1 2 3
##
## [[2]]
##  [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L"
## [13] "M" "N" "O" "P" "Q" "R" "S" "T" "U" "V" "W" "X"
## [25] "Y" "Z"
##
## [[3]]
## [1] TRUE
##
## [[4]]
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    3    5    7    9
## [2,]    2    4    6    8   10
```

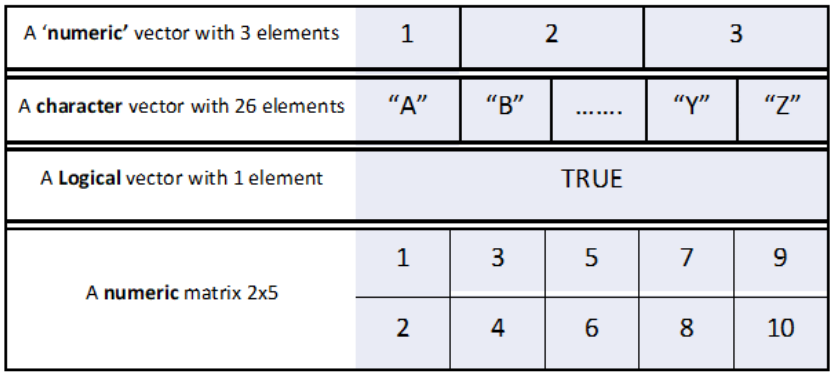


Figure 1.11: A list in R, can contain vector, matrices, array or even lists

Similar to vectors these elements can be named also.

```
list(first_item = 1:5, second_item = my_mat2)
```

```
## $first_item
## [1] 1 2 3 4 5
```

```
##
## $second_item
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    3    5    7    9
## [2,]    2    4    6    8   10
```

OR

```
my_list <- list(first=c(A=1, B=2, C=3),second=my_mat2)
my_list
```

```
## $first
## A B C
## 1 2 3
##
## $second
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    3    5    7    9
## [2,]    2    4    6    8   10
```

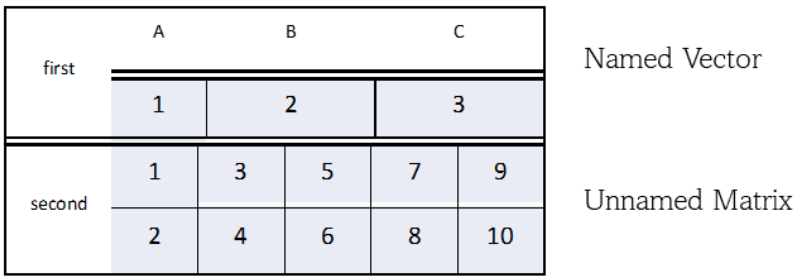


Figure 1.12: Similar to vector elements, the elements in list can be named also

OR more interestingly, lists can even contain another lists.

```
my_list2 <- list(my_list, new_item = LETTERS)
my_list2
```

```
## [[1]]
## [[1]]$first
## A B C
## 1 2 3
```

```
##
## [[1]]$second
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    3    5    7    9
## [2,]    2    4    6    8   10
##
##
## $new_item
##  [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L"
## [13] "M" "N" "O" "P" "Q" "R" "S" "T" "U" "V" "W" "X"
## [25] "Y" "Z"
```

Number of items at first level can be checked using `length` as in vectors. Checking number of items in second level onward will be covered in subsequent chapter(s).

```
length(my_list)
```

```
## [1] 2
```

```
length(my_list2) # If you thought its output should have been 3, think again.
```

```
## [1] 2
```

1.4.5 Data Frame

Data frames are used to store tabular data (or rectangular) in R. They are an important type of object in R.

Data frames are represented as a special type of list where every element of the list has to have the same length. Each element of the list can be thought of as a column and the length of each element of the list is the number of rows.

Unlike matrices, data frames can store different classes of objects in each column. (Remember that matrices must have every element be the same class).

To create a data frame from scratch we will use function `data.frame()`. See

```
my_df <- data.frame(emp_name = c('Thomas', 'Andrew', 'Jonathan', 'Bob', 'Charles'),
                    department = c('HR', 'Accounts', 'Accounts', 'Execution', 'Tech'),
                    age = c(40, 43, 39, 42, 25),
                    salary = c(20000, 22000, 21000, 25000, NA),
                    whether_permanent = c(TRUE, TRUE, FALSE, NA, NA))
my_df
```

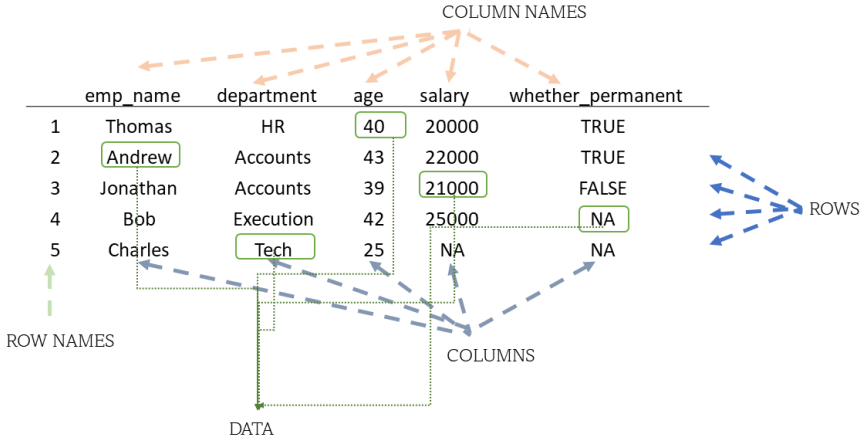


Figure 1.13: An example data frame

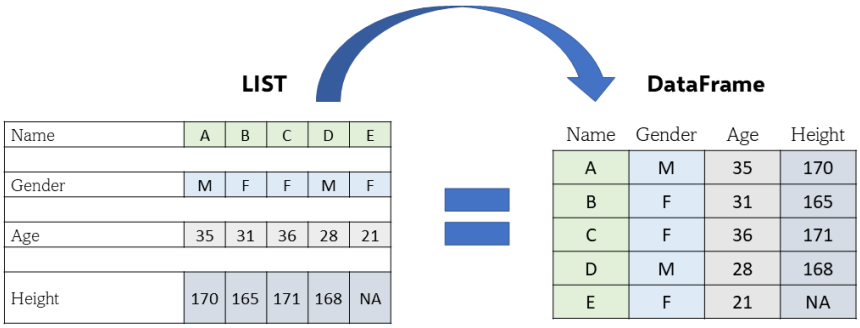


Figure 1.14: A data frame in R, is just a special kind of list

```
##   emp_name department age salary whether_permanent
## 1   Thomas         HR  40  20000              TRUE
## 2   Andrew   Accounts  43  22000              TRUE
## 3 Jonathan   Accounts  39  21000             FALSE
## 4      Bob  Execution  42  25000                NA
## 5  Charles       Tech  25     NA                NA
```

Note that R, on its own, has allocated row names that are numbers to each of the row on its own.

Of course at most of the times we will have data frames ready for us to analyse and thus we will learn to import/read external data in r, in subsequent chapters. To check dimensions of a data frame use `dim` as in matrix.

```
dim(my_df)
```

```
## [1] 5 5
```

Thus, the object types in R, can be depicted as -

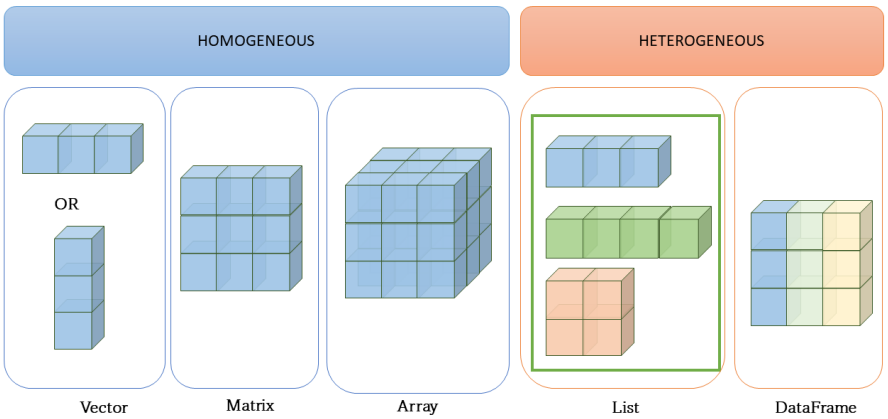


Figure 1.15: Most important Data structures, in R

1.5 Other Data types

Of course, there are other data types in R of which three are particularly useful `factor`, `date` and `date-time`. These types are actually built over the base

atomic types, `integer`, `double` and `double` respectively and that's why are being discussed separately. These types are built as `S3 objects` in r, and users may also define their own data types in _object oriented programming‘. OOP being concept of core programming concepts and therefore are out of the scope here.

However, to understand better the S3 objects we have to understand that atomic objects, let us for the sake of simplicity consider only vectors, have attributes. One of the attributes that each vector has is `names`, which we have seen already. Attributes of any object can be viewed/called from function `attributes()`.

```
vec <- setNames(1:26, LETTERS)
attributes(vec)
```

```
## $names
##  [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L"
## [13] "M" "N" "O" "P" "Q" "R" "S" "T" "U" "V" "W" "X"
## [25] "Y" "Z"
```

It should have been clear by now that apart from `names`, other `attributes` may be assigned to a vector.

1.5.1 Factors

A factor is a vector that can contain only predefined values. It is used to store categorical data. Factors are built on top of an integer vector with two attributes: a *class*, 'factor', which makes it behave differently from regular integer vectors, and *levels*, which defines the set of allowed values. To create factors we will use function `factor`.

```
fac <- factor(c('a', 'b', 'c', 'a'))
fac
```

```
## [1] a b c a
## Levels: a b c
```

```
typeof(fac) # notice its output
```

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

```
attributes(fac)
```

```
## $levels
## [1] "a" "b" "c"
##
## $class
## [1] "factor"
```

So if `typeof` of a factor is returning integer, how will we check its type? We may use `class` or `is.factor` in this case.

```
class(fac)
```

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

```
is.factor(fac)
```

```
## [1] TRUE
```

Now a factor can be ordered also. We may use its argument `ordered = TRUE` along with another argument `levels`.

```
my_degrees <- c("PG", "PG", "Doctorate", "UG", "PG")
my_factor <- factor(my_degrees, levels = c('UG', 'PG', 'Doctorate'), ordered = TRUE)
my_factor # notice output here
```

```
## [1] PG        PG        Doctorate UG        PG
## Levels: UG < PG < Doctorate
```

```
is.ordered(my_factor)
```

```
## [1] TRUE
```

Another argument `labels` can also be used to display labels, which may be different from levels. See

```
my_factor <- factor(my_degrees, levels = c('UG', 'PG', 'Doctorate'),
                    labels = c("Under-Graduate", "Post Graduate", "Ph.D"),
                    ordered = TRUE)
my_factor # notice output here
```

```
## [1] Post Graduate  Post Graduate  Ph.D
## [4] Under-Graduate Post Graduate
## Levels: Under-Graduate < Post Graduate < Ph.D
```

```
is.factor(c(my_factor, "UG"))
```

```
## [1] FALSE
```

Attribute `levels` can be used as a function to retrieve/modify these.

```
levels(my_factor)
```

```
## [1] "Under-Graduate" "Post Graduate"  "Ph.D"
```

```
levels(my_factor) <- c("Grad", "Masters", "Doctorate")
my_factor
```

```
## [1] Masters   Masters   Doctorate Grad      Masters
## Levels: Grad < Masters < Doctorate
```

Remember that while factors look like (and often behave like) character vectors, they are built on top of integers. Try to think of output of this `is.factor(c(my_factor, "UG"))` before running it in your console.

1.5.2 Date

Date vectors are built on top of double vectors. They have class "Date" and no other attributes. A common way to create `date` vectors in R, is converting a character string to date using `as.Date()` (see case carefully),

```
my_date <- as.Date("1970-01-31")
my_date
```

```
## [1] "1970-01-31"
```

```
attributes(my_date)
```

```
## $class
## [1] "Date"
```

Do check other arguments of as.Date by running `?as.Date()` in your console. To check whether a given variable is of type Date in r, there is no function like `is.Date` in base r, so we may use `inherits()` in this case.

```
inherits(my_date, 'Date')
```

```
## [1] TRUE
```

1.5.3 Date-time (`POSIXct`)

Times are represented by the `POSIXct` or the `POSIXlt` class.

- POSIXct is just a very large integer under the hood. It use a useful class when you want to store times in something like a data frame.
- POSIXlt is a list underneath and it stores a bunch of other useful information like the day of the week, day of the year, month, day of the month. See

```
my_time <- Sys.time()
my_time
```

```
## [1] "2023-04-03 15:56:30 IST"
```

```
class(my_time)
```

```
## [1] "POSIXct" "POSIXt"
```

```
my_time2 <- as.POSIXlt(my_time)
class(my_time2)
```

```
## [1] "POSIXlt" "POSIXt"
```

```
names(unclass(my_time2))
```

```
##  [1] "sec"    "min"    "hour"   "mday"   "mon"
##  [6] "year"   "wday"   "yday"   "isdst"  "zone"
## [11] "gmtoff"
```

1.5.4 Duration (`difftime`)

Duration, which represent the amount of time between pairs of dates or datetimes, are stored in `difftimes`. `Difftimes` are built on top of doubles, and have a units attribute that determines how the integer should be interpreted.

```
two_days <- as.difftime(2, units = 'days')
two_days
```

```
## Time difference of 2 days
```

These over the top, data types will be discussed in more detail in subsequent chapters.

Chapter 2

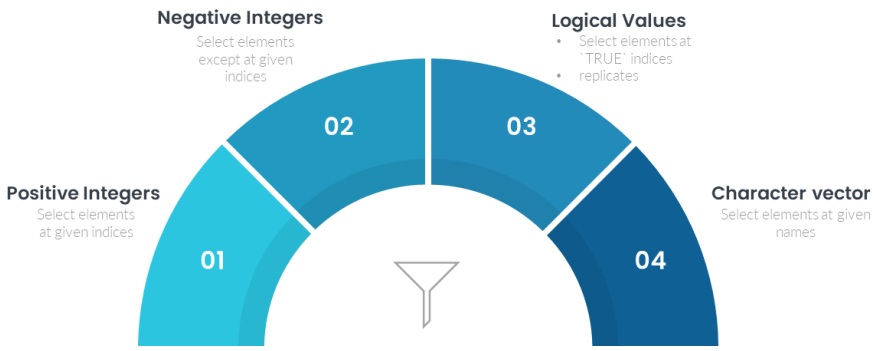
Subsetting R objects

There are multiple methods for sub-setting R objects (vectors, matrices, data frames, lists, etc.) and each have its own uses and benefits. We will discuss each one of them. Three operators `[`, `[[` & `$` will be used.

2.1 Subsetting vectors

Let us first start sub-setting vectors, which is as we have learned, atomic object in R. To subset the vectors we will use `[`. For this we will use following `x` vector, which has 6 elements (names) each starting with alphabets A to F.

```
x <- c('Andrew', 'Bob', 'Chris', 'Danny', 'Edmund', 'Freddie')
```



2.1.1 Subsetting through a vector of positive integers

Sub-setting through positive integers will give us elements at those given position (indices). See this

```
# fourth element
x[4]
```

```
## [1] "Danny"
```

```
# third to fifth element
x[3:5]
```

```
## [1] "Chris"  "Danny"  "Edmund"
```

```
# first and fifth element
x[c(1,3)]
```

```
## [1] "Andrew" "Chris"
```

Note: Check what happens when the integer vector has repeated integers.

2.1.2 Subsetting through a vector of negative integers

Sub-setting through negative integers will give us all elements **except** those at given indices. See

```
# all elements except that at fourth
x[-4]
```

```
## [1] "Andrew"  "Bob"     "Chris"   "Edmund"  "Freddie"
```

```
# all elements except third to fifth
x[-(3:5)]
```

```
## [1] "Andrew"  "Bob"     "Freddie"
```

```
# all elements except first and fifth
x[-c(1,5)]
```

```
## [1] "Bob"     "Chris"   "Danny"   "Freddie"
```

Note: Try mixing sub-setting with a vector having both positive and negative integers in your console and check what happens.

2.1.3 Subsetting through a logical vector

We can also subset a given vector through another vector having `logical` values i.e. `TRUE` and `FALSE`. As you can understand output/result will have elements at places having `TRUE` only.

```
# First, third and fifth element only
x[c(TRUE, FALSE, TRUE, TRUE, FALSE, FALSE)]
```

```
## [1] "Andrew" "Chris"  "Danny"
```

Recycling is an important concept while sub-setting though a logical vector. It recycles the given logical vector up to the length of vector to be subset. Thus, `x[TRUE]` will give us original `x` only.

```
x[TRUE]
```

```
## [1] "Andrew"  "Bob"     "Chris"   "Danny"   "Edmund"
## [6] "Freddie"
```

```
x[c(TRUE, FALSE)] # will give elements at odd indices
```

```
## [1] "Andrew" "Chris"  "Edmund"
```

Note: Try to subset a vector through a logical vector having missing values i.e. `NA` along with `TRUE` and/or `FALSE` in your console and check what happens.

Sub-setting through logical vector is most important and used sub-setting method as we will see it subsequent chapter/sections when we will filter a vector on the basis of some conditions.

2.1.4 Subsetting through a character vector

This method is used when the given vector is named. We can pass desired names inside `[]` to get/filter those desired elements. See this example.

```
# let us create a named vector `y`
y <- setNames(1:6, LETTERS[1:6])
# display `y`
y
```

```
## A B C D E F
## 1 2 3 4 5 6
```

```
# subset elements named `A` and `C`
y[c('A', 'C')]
```

```
## A C
## 1 3
```

Note that we have used quotes in above method of sub-setting. We can use this method when we have names saved in another variable. See this

```
var <- c('A', 'C', 'E')
# subset those elements from `y` which are named as per `var`
y[var] # notice that since `var` is a variable, we have not used quotes.
```

```
## A C E
## 1 3 5
```

Note: Similar to positive integer indexing we will get repeated values if character vector has repeated names.

```
y[c("A", "A", "C", "A")]
```

```
## A A C A
## 1 1 3 1
```

Other two methods of indexing will not be used frequently but are important to know for debugging the code as sometimes your subset vector may be `NULL` or `zero`

2.1.5 Subsetting through nothing

Indexing through nothing i.e. simply with `[]` will give us original vector.

```
x[]
```

```
## [1] "Andrew"  "Bob"     "Chris"   "Danny"   "Edmund"
## [6] "Freddie"
```

2.1.6 Subsetting through Zero

Sub-setting through `NULL` or `0` will give us a zero length vector.

```
x[NULL]
```

```
## character(0)
```

```
y[0]
```

```
## named integer(0)
```

```
is.null(x[NULL])
```

```
## [1] FALSE
```

2.2 Subsetting Matrices and arrays

We can subset higher dimensional structures (Matrix - 2 dimensional and arrays - dimension greater than 2) using (i) multiple vectors, (ii) single vector and (iii) matrix.

Let us first create a 5x5 matrix say `mat` with elements named A_{mn} where `m` will denote row number and `n` will denote column number.

```
##      [,1]  [,2]  [,3]  [,4]  [,5]
## [1,] "A11" "A12" "A13" "A14" "A15"
## [2,] "A21" "A22" "A23" "A24" "A25"
## [3,] "A31" "A32" "A33" "A34" "A35"
## [4,] "A41" "A42" "A43" "A44" "A45"
## [5,] "A51" "A52" "A53" "A54" "A55"
```

2.2.1 Multiple vectors

This is extension of all sub-setting methods explained for a vector. In objects with higher dimensionality we will have to provide one vector for each dimension. Blank values, as you may understood (ref - sub-setting through nothing explained above) will do nothing and return that dimension complete.

```
# first and second row with third and fifth column
mat[1:2, c(3,5)]
```

```
##      [,1]  [,2]
## [1,] "A13" "A15"
## [2,] "A23" "A25"
```

```
# third to fifth column, all rows
mat[,3:5]
```

```
##      [,1]  [,2]  [,3]
## [1,] "A13" "A14" "A15"
## [2,] "A23" "A24" "A25"
## [3,] "A33" "A34" "A35"
## [4,] "A43" "A44" "A45"
## [5,] "A53" "A54" "A55"
```

```
# all columns except third
mat[, -3]
```

```
##      [,1]  [,2]  [,3]  [,4]
## [1,] "A11" "A12" "A14" "A15"
## [2,] "A21" "A22" "A24" "A25"
## [3,] "A31" "A32" "A34" "A35"
## [4,] "A41" "A42" "A44" "A45"
## [5,] "A51" "A52" "A54" "A55"
```

```
# Odd rows, all columns
mat[c(TRUE, FALSE),]
```

```
##      [,1]  [,2]  [,3]  [,4]  [,5]
## [1,] "A11" "A12" "A13" "A14" "A15"
## [2,] "A31" "A32" "A33" "A34" "A35"
## [3,] "A51" "A52" "A53" "A54" "A55"
```

The idea can be extended to a named matrix also.

```
# First create a named matrix
rownames(mat) <- paste0("Row", 1:5)
colnames(mat) <- paste0("Col", 1:5)
mat
```

```
##      Col1  Col2  Col3  Col4  Col5
## Row1 "A11" "A12" "A13" "A14" "A15"
## Row2 "A21" "A22" "A23" "A24" "A25"
## Row3 "A31" "A32" "A33" "A34" "A35"
## Row4 "A41" "A42" "A43" "A44" "A45"
## Row5 "A51" "A52" "A53" "A54" "A55"
```

```
# filter desired rows/columns
mat[c("Row1"), c("Col2", "Col3")]
```

```
##  Col2  Col3
## "A12" "A13"
```

In the above example you must have noticed that indexing objects with higher dimensionality may return the objects with lower dimensionality. E.g. sub-setting a matrix may return a vector. **We can control the dimensionality reduction through the argument `drop=FALSE` which is by default TRUE and may thus introduce bugs in the code.**

```
mat[c("Row1"), c("Col2", "Col3"), drop=FALSE]
```

```
##      Col2  Col3
## Row1 "A12" "A13"
```

```
#check this
dim(mat[c("Row1"), c("Col2", "Col3"), drop=FALSE])
```

```
## [1] 1 2
```

```
#versus this
dim(mat[c("Row1"), c("Col2", "Col3")])
```

```
## NULL
```

2.2.2 Subsetting through one vector

By now it should be clear that objects with higher dimensionality like matrices, array are actually vectors at the core of r, displayed and acting like objects having more than one dimension. So sub-setting with single vector on these objects coerce the behavior of these objects as vectors only and give output exactly as shown in previous section.

```
mat[c(1, 10, 15, 25)]
```

```
## [1] "A11" "A52" "A53" "A55"
```

```
# OR
mat[c(TRUE, FALSE)]
```

```
##  [1] "A11" "A31" "A51" "A22" "A42" "A13" "A33" "A53"
##  [9] "A24" "A44" "A15" "A35" "A55"
```

2.2.3 Subsetting through a matrix

We can also subset objects with higher dimensionality with integer matrix (having number of columns equal to dimensions). In other words, to subset a matrix (2D) with the help of other matrix we will need a 2 column matrix where first column will indicate row number and second column will indicate column number. See

```
selection_matrix <- matrix(c(1,1,
                             2,2,
                             3,3), ncol = 2, byrow = TRUE)
mat[selection_matrix]
```

```
## [1] "A11" "A22" "A33"
```

2.3 Subsetting lists

List sub-setting can be done using either `[]`, `[[]]` or `$`. To understand the difference between these, let us consider these one by one. As done earlier let us consider a list of 4 elements - one vector, one matrix, one list and one data frame. For now let us consider that list is unnamed.

```
my_list <- list(
  11:20,                                                  # first element
  outer(1:4, 1:4, FUN = function(x, y) paste0('B', x, y)),  # second element
  list(LETTERS[1:8], TRUE),                               # third element
  data.frame(col1 = letters[1:4], col2 = 5:8)             # fourth element
)
# display the list
my_list
```

```
## [[1]]
##  [1] 11 12 13 14 15 16 17 18 19 20
##
## [[2]]
```

```
##      [,1]  [,2]  [,3]  [,4]
## [1,] "B11" "B12" "B13" "B14"
## [2,] "B21" "B22" "B23" "B24"
## [3,] "B31" "B32" "B33" "B34"
## [4,] "B41" "B42" "B43" "B44"
##
## [[3]]
## [[3]][[1]]
## [1] "A" "B" "C" "D" "E" "F" "G" "H"
##
## [[3]][[2]]
## [1] TRUE
##
##
## [[4]]
##   col1 col2
## 1    a    5
## 2    b    6
## 3    c    7
## 4    d    8
```

2.3.1 Subsetting lists with []

Sub-setting lists with [] will always result a list containing desired element(s).

```
my_list[2]
```

```
## [[1]]
##      [,1]  [,2]  [,3]  [,4]
## [1,] "B11" "B12" "B13" "B14"
## [2,] "B21" "B22" "B23" "B24"
## [3,] "B31" "B32" "B33" "B34"
## [4,] "B41" "B42" "B43" "B44"
```

```
class(my_list[1])
```

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

We can apply other ideas of vector sub-setting as explained earlier with this list sub-setting. The output will also be list containing one or more items.

2.3.2 3.2 Subsetting lists with [[]]

Sub-setting list with `[[]]` will return that specific item (as per index given) but the output will be of type of that specific item.

```
my_list[[2]]
```

```
##      [,1]  [,2]  [,3]  [,4]
## [1,] "B11" "B12" "B13" "B14"
## [2,] "B21" "B22" "B23" "B24"
## [3,] "B31" "B32" "B33" "B34"
## [4,] "B41" "B42" "B43" "B44"
```

```
class(my_list[[4]])
```

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

Notice the difference in outputs created with `my_list[2]` and `my_list[[2]]` in above 2 code blocks.

Needless to say, one cannot index/subset lists using multiple indices. Check `my_list[[1:2]]` in your console as the results may not be as what you think. Now **chaining** may also be applied here.

```
# third element of second element
my_list[[2]][3] # recall that by default matrix is by column
```

```
## [1] "B31"
```

```
# or
my_list[[2]][1:3,2:4]
```

```
##      [,1]  [,2]  [,3]
## [1,] "B12" "B13" "B14"
## [2,] "B22" "B23" "B24"
## [3,] "B32" "B33" "B34"
```

2.3.3 Subsetting with $

`$` is a shorthand operator: `x$y` is roughly equivalent to `x[["y"]]`. To check this let us assign our list some names.

```
names(my_list) <- c("first", "second", "third", "fourth")
# Now see
my_list$first
```

```
##  [1] 11 12 13 14 15 16 17 18 19 20
```

```
my_list$fourth$col2
```

```
## [1] 5 6 7 8
```

Notice that rules for dimensionality reduction also applies with $.

Another difference between [[sub-setting versus $ sub-setting is partial matching (*left to right only*), which is possible with $ only and not with [[. See

```
my_list$fir
```

```
##  [1] 11 12 13 14 15 16 17 18 19 20
```

```
my_list[['fir']]
```

```
## NULL
```

2.4 Data frames

As already explained data frames are basically lists with each element having equal length, rules for sub-setting lists all apply with data frames. One addition is that data frames can also be subset using rules for matrix sub-setting.

```
mtcars # it is a default data frame in r
```

```
##                    mpg cyl  disp  hp drat    wt
## Mazda RX4         21.0   6 160.0 110 3.90 2.620
## Mazda RX4 Wag     21.0   6 160.0 110 3.90 2.875
## Datsun 710        22.8   4 108.0  93 3.85 2.320
## Hornet 4 Drive    21.4   6 258.0 110 3.08 3.215
## Hornet Sportabout 18.7   8 360.0 175 3.15 3.440
## Valiant           18.1   6 225.0 105 2.76 3.460
## Duster 360        14.3   8 360.0 245 3.21 3.570
## Merc 240D         24.4   4 146.7  62 3.69 3.190
```

```
## Merc 230            22.8   4 140.8  95 3.92 3.150
## Merc 280            19.2   6 167.6 123 3.92 3.440
## Merc 280C           17.8   6 167.6 123 3.92 3.440
## Merc 450SE          16.4   8 275.8 180 3.07 4.070
## Merc 450SL          17.3   8 275.8 180 3.07 3.730
## Merc 450SLC         15.2   8 275.8 180 3.07 3.780
## Cadillac Fleetwood  10.4   8 472.0 205 2.93 5.250
## Lincoln Continental 10.4   8 460.0 215 3.00 5.424
## Chrysler Imperial   14.7   8 440.0 230 3.23 5.345
## Fiat 128            32.4   4  78.7  66 4.08 2.200
## Honda Civic         30.4   4  75.7  52 4.93 1.615
## Toyota Corolla      33.9   4  71.1  65 4.22 1.835
## Toyota Corona       21.5   4 120.1  97 3.70 2.465
## Dodge Challenger    15.5   8 318.0 150 2.76 3.520
## AMC Javelin         15.2   8 304.0 150 3.15 3.435
## Camaro Z28          13.3   8 350.0 245 3.73 3.840
## Pontiac Firebird    19.2   8 400.0 175 3.08 3.845
## Fiat X1-9           27.3   4  79.0  66 4.08 1.935
## Porsche 914-2       26.0   4 120.3  91 4.43 2.140
## Lotus Europa        30.4   4  95.1 113 3.77 1.513
## Ford Pantera L      15.8   8 351.0 264 4.22 3.170
## Ferrari Dino        19.7   6 145.0 175 3.62 2.770
## Maserati Bora       15.0   8 301.0 335 3.54 3.570
## Volvo 142E          21.4   4 121.0 109 4.11 2.780
##                      qsec vs am gear carb
## Mazda RX4           16.46  0  1    4    4
## Mazda RX4 Wag       17.02  0  1    4    4
## Datsun 710          18.61  1  1    4    1
## Hornet 4 Drive      19.44  1  0    3    1
## Hornet Sportabout   17.02  0  0    3    2
## Valiant             20.22  1  0    3    1
## Duster 360          15.84  0  0    3    4
## Merc 240D           20.00  1  0    4    2
## Merc 230            22.90  1  0    4    2
## Merc 280            18.30  1  0    4    4
## Merc 280C           18.90  1  0    4    4
## Merc 450SE          17.40  0  0    3    3
## Merc 450SL          17.60  0  0    3    3
## Merc 450SLC         18.00  0  0    3    3
## Cadillac Fleetwood  17.98  0  0    3    4
## Lincoln Continental 17.82  0  0    3    4
## Chrysler Imperial   17.42  0  0    3    4
## Fiat 128            19.47  1  1    4    1
## Honda Civic         18.52  1  1    4    2
## Toyota Corolla      19.90  1  1    4    1
## Toyota Corona       20.01  1  0    3    1
```

```
## Dodge Challenger    16.87  0  0    3    2
## AMC Javelin         17.30  0  0    3    2
## Camaro Z28          15.41  0  0    3    4
## Pontiac Firebird    17.05  0  0    3    2
## Fiat X1-9           18.90  1  1    4    1
## Porsche 914-2       16.70  0  1    5    2
## Lotus Europa        16.90  1  1    5    2
## Ford Pantera L      14.50  0  1    5    4
## Ferrari Dino        15.50  0  1    5    6
## Maserati Bora       14.60  0  1    5    8
## Volvo 142E          18.60  1  1    4    2
```

```
# list type sub-setting
mtcars[[2]]  # second column
```

```
##  [1] 6 6 4 6 8 6 8 4 4 6 6 8 8 8 8 8 8 4 4 4 4 8 8 8 8
## [26] 4 4 4 8 6 8 4
```

```
# matrix type
mtcars[1:4, 2:3] # first four rows with second & third columns
```

```
##                cyl disp
## Mazda RX4        6  160
## Mazda RX4 Wag    6  160
## Datsun 710       4  108
## Hornet 4 Drive   6  258
```

Remember

1. If sub-setting data frames with single vector, data frame behave like lists,
2. If however, sub-setting data frame through two vectors, these behave like matrices.

2.5 Subsetting and assignment

All the sub-setting that we have seen can be used for assignment as well.

```
my_list$first <- mtcars[1:4, 2:3]
my_list
```

```
## $first
##                cyl disp
```

```
## Mazda RX4         6  160
## Mazda RX4 Wag     6  160
## Datsun 710        4  108
## Hornet 4 Drive    6  258
##
## $second
##      [,1]  [,2]  [,3]  [,4]
## [1,] "B11" "B12" "B13" "B14"
## [2,] "B21" "B22" "B23" "B24"
## [3,] "B31" "B32" "B33" "B34"
## [4,] "B41" "B42" "B43" "B44"
##
## $third
## $third[[1]]
## [1] "A" "B" "C" "D" "E" "F" "G" "H"
##
## $third[[2]]
## [1] TRUE
##
##
## $fourth
##   col1 col2
## 1    a    5
## 2    b    6
## 3    c    7
## 4    d    8
```

Chapter 3

Useful functions and operations in R

What is a function? A function f is a relationship which map an input x to an specific output, which is denoted as $f(x)$. There are only two conditions i.e. every input should have an output, and same input if passed into same function multiple times, it should produce same output each time. So if $x = y$ we should have $f(x) = f(y)$.

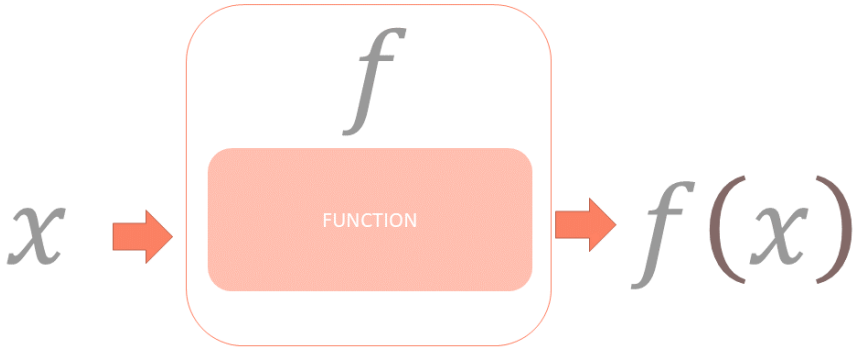


Figure 3.1: Author's illustration of a function

For example `squaring` if considered on numbers is a function. We denote this as $f(x) = x^2$. Or, `square-root` on positive numbers is also a function.

Now there may be more than one input, let us assume three inputs `x`, `y` and `z` and our function's job is to add three times `x`, two times `z` and one time `y` together. We will write this function as $f(x, y, z) = 3x + y + 2z$. Each

programming language has some pre-defined functions. Here inputs are usually termed as `arguments`. Normally values to `arguments` should be passed by users, but many times there's a default value. R's engine then calculates the output as per definition of that function and gives us the output. If that output is assigned to some variable R does not displays/prints anything but if function is performed only the output is displayed usually, with the exception that many times function is carried out silently and nothing is returned.

In this chapter we will learn about some of the pre-defined functions which shall be used in our data analysis operations. We can also define our own custom functions which we will learn in chapter 4.2.

As an example, `sum()` is a predefined function available in R, which produces sum of one or more vectors passed in the function as arguments.

```
sum(1:10, 15:45)
```

```
## [1] 985
```

To check the arguments avaialble for any pre-defined function, we can use another function `args()` which take a *function name* as an argument and returns all the available arguments to that function.

```
args(sum)
```

```
## function (..., na.rm = FALSE)
## NULL
```

> To get the definition of any existing function, we may just type its name without parenthesis on console, and the definition will be returned as an output.

```
sum
```

```
## function (..., na.rm = FALSE)  .Primitive("sum")
```

To get further help about any existing function, refer section 0.7.

3.1 Conditions and logical operators/operands

Table 3.1: Conditions and logical operators/operands

Operator/ function	Meaning	Example
`==`	Is RHS equal to LHS?	`5 == 2` will return FALSE `'Anil' == 'anil'` is FALSE
`!=`	Is RHS not equal to LHS?	`'ABCD' != 'abcd'` is TRUE
`>=`	Is LHS greater than or equal to RHS?	`5 >= 2` will return TRUE
`<=`	Is LHS less than or equal to RHS?	`15 <= 2` will return FALSE
`>`	Is LHS strictly greater than RHS?	`2 > 2` will return FALSE
`<`	Is LHS strictly less than RHS?	`12 < 12` will return FALSE
`is.na()`	Whether the argument passed is NA	`is.na(NA)` is TRUE
`is.null()`	Whether the argument passed is null	`is.null(NA)` is FALSE
`|`	Logical OR	`TRUE | FALSE` will return `TRUE`
`&`	Logical AND	`TRUE & FALSE` will return `FALSE`
`!`	Logical NOT	`!TRUE` will return `FALSE`
`||`	Element wise Logical OR	Examines only the first element of the operands resulting into a single length logical vector
`&&`	Element wise Logical AND	Examines only the first element of the operands resulting into a single length logical vector
`%in%`	LHS **IN** RHS	Checks whether LHS elements are present in RHS vector

Vectorisation of operations and functions

All the above mentioned operators are **vectorised**. Except `||` and `&&` will return vector of same length as we are comparing. Check

```
LETTERS[1:4] == letters[1:4]
```

```
## [1] FALSE FALSE FALSE FALSE
```

```
10:1 >= 1:10
```

```
##  [1]  TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE
##  [9] FALSE FALSE
```

```
# TRUE will act as 1 and FALSE as 0
x <- c(TRUE, FALSE, FALSE, TRUE)
y <- c(1, 0, 1, 10)
x == y
```

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

```
# Examples of element wise operations
x && y
```

```
## Warning in x && y: 'length(x) = 4 > 1' in coercion to
## 'logical(1)'
```

```
## Warning in x && y: 'length(x) = 4 > 1' in coercion to
## 'logical(1)'
```

```
## [1] TRUE
```

```
x || y
```

```
## Warning in x || y: 'length(x) = 4 > 1' in coercion to
## 'logical(1)'
```

```
## [1] TRUE
```

```
# character strings may be checked for alphabetic order
'ABCD' >= 'AACD'
```

```
## [1] TRUE
```

Recycling

Recycling rules apply when two vectors are not of equal length. See these examples.

```
# Notice that results are displayed silently
LETTERS[1:4] == 'A'
```

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

```
#Notice that results are displayed with a warning
LETTERS[1:5] == LETTERS[1:3]
```

```
## Warning in LETTERS[1:5] == LETTERS[1:3]: longer object
## length is not a multiple of shorter object length
```

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

The **operator %in%** behaves slightly different from above. Each searches each element of LHS in RHS and gives result in a logical vector equal to length of LHS vector. See these examples carefully.

```
'A' %in% LETTERS
```

```
## [1] TRUE
```

```
LETTERS %in% LETTERS[1:4]
```

```
##  [1]  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE
##  [9] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [17] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [25] FALSE FALSE
```

Handling Missing values NA in these operations

While checking for any condition to be TRUE or FALSE missing values NA and/or NaN should be handled carefully or a bug may be introduced. See these examples-

```
FALSE != NA
```

```
## [1] NA
```

```
TRUE != NA
```

```
## [1] NA
```

Thus, if any of the condition is evaluated on a vector, we can have NA in our output along with TRUE and FALSE. See this example

```
x <- c(1, 5, 15, NA, 2, 3)
x <= 5
```

```
## [1]  TRUE  TRUE FALSE    NA  TRUE  TRUE
```

These missing values however behaves slightly different with logical operators `&` `|`. See these examples.

```
TRUE | NA
```

```
## [1] TRUE
```

```
FALSE & NA
```

```
## [1] FALSE
```

Use of above logical operators for subsetting

Since the logical operations on vectors gives a `logical` vector as output, these can be used for sub-setting as well. See these examples.

```
my_ages <- c(40, 45, 31, 51, 25, 27, 59, 45)
# filter ages greater than or equal to 30
my_ages[my_ages >= 30]
```

```
## [1] 40 45 31 51 59 45
```

```
my_names <- c("Andrew", "Bob", "Carl", "Daven", "Earl")
# filter names which start with alphabet either A, B or C
my_names[my_names <= "D"]
```

```
## [1] "Andrew" "Bob"    "Carl"
```

Conditions with `ifelse`

Syntax `ifelse(test, yes, no)` will be used to return value (of same shape as `test`) which is filled with elements selected from either `yes` or `no` depending on whether the elements of `test` are `TRUE` or `FALSE`. See this example

```
x <- c(1:5, NA, 16:20)
ifelse(x>5, 'Greater than 5', 'Upto 5')
```

```
##  [1] "Upto 5"         "Upto 5"         "Upto 5"
##  [4] "Upto 5"         "Upto 5"         NA
##  [7] "Greater than 5" "Greater than 5" "Greater than 5"
## [10] "Greater than 5" "Greater than 5"
```

Functions `all()` and `any()`

These are shortcut functions to tell us whether `all` or `any` of the elements of given object are `TRUE`. See This example

```
x <- 11:20
all(x > 5)
```

```
## [1] TRUE
```

```
any(x > 20)
```

```
## [1] FALSE
```

All of the above mentioned operators (along with those listed in section 1.1) are **vectorised**. Check these examples.

```
x <- 1:5
y <- 6:10

x + y
```

```
## [1]  7  9 11 13 15
```

```
x - y
```

```
## [1] -5 -5 -5 -5 -5
```

```
x * y
```

```
## [1]  6 14 24 36 50
```

```
x / y
```

```
## [1] 0.1667 0.2857 0.3750 0.4444 0.5000
```

```
x ^ y
```

```
## [1]       1     128    6561  262144 9765625
```

```
# Caution: here RHS is not a vector
y %% 3
```

```
## [1] 0 1 2 0 1
```

```
y %/% 3
```

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

Recycling also applies on mathematical operators. See these examples and notice when R gives results silently and when with a warning.

```
10:15 + 4
```

```
## [1] 14 15 16 17 18 19
```

```
100:110 - 50
```

```
##  [1] 50 51 52 53 54 55 56 57 58 59 60
```

```
# when length of one vector is multiple of length of smaller vector
x <- c(5, 2, 7, 9)
y <- c(7, 8)
x + y
```

```
## [1] 12 10 14 17
```

```
# when length of one vector is not multiple of length of smaller vector
x + c(1, 2, 3)
```

```
## Warning in x + c(1, 2, 3): longer object length is not
## a multiple of shorter object length
```

```
## [1]  6  4 10 10
```

All the above-mentioned operators/functions may also be used on matrices, arrays of larger dimension, since we have already seen that matrices/arrays are actually vectors at the core.

```
mat1 <- matrix()
```

3.2 Common arithmetical Functions

Table 3.2: Common Arithmetical Functions

Function	Meaning	Input	Output
`sum()`	Adds all elements	One or more Vector, matrix, array	Vector having 1 element only
`prod()`	Returns product of all elements	One or more Vector, matrix, array	Vector having 1 element only
`mean()`	Returns the arithmetic mean	One Vector, matrix, array	Vector having 1 element only
`max()`	Returns maximum value	One or more Vector, matrix, array	Vector having 1 element only
`min()`	Returns minimum value	One or more Vector, matrix, array	Vector having 1 element only
`ceiling()`	Returns integer(s) not less than given values	One Vector, matrix, array	Vector, matrix, array having same `dim`
`floor()`	Returns largest integers not greater than given values	One Vector, matrix, array	Vector, matrix, array having same `dim`
`trunc()`	returns integers formed by truncating the values towards 0	One Vector, matrix, array	Vector, matrix, array having same `dim`

Function	Meaning	Input	Output
`round(x, digits = 0)`	Rounds the given value(s) to number of decimal places provided	One Vector, matrix, array	Vector, matrix, array having same `dim`
`signif(x, digits = 6)`	Round to `significant` digits	One Vector, matrix, array	Vector, matrix, array having same `dim`
`factorial()`	Returns factorial	One Vector, matrix, array of `integer` type	Vector having 1 element
`sqrt()`	Returns square root	One Vector, matrix, array	Vector, matrix, array having same `dim`
`log10()` or `log2()`	Logrithm with base 10 or 2 respectively	One Vector, matrix, array	Vector, matrix, array having same `dim`
`exp(x)`	returns exponential	One Vector, matrix, array	Vector, matrix, array having same `dim`

See these examples.

```
sum(1:100, 1:10)
```

```
## [1] 5105
```

```
Mat1 <- matrix(1:10, nrow = 2)
Mat2 <- matrix(1:4, nrow = 2)

prod(Mat1, Mat2)
```

```
## [1] 87091200
```

```
sqrt(Mat2)
```

```
##       [,1]  [,2]
## [1,] 1.000 1.732
## [2,] 1.414 2.000
```

```
log10(Mat1)
```

```
##       [,1]   [,2]   [,3]   [,4]   [,5]
## [1,] 0.000 0.4771 0.6990 0.8451 0.9542
## [2,] 0.301 0.6021 0.7782 0.9031 1.0000
```

```
factorial(10:1)
```

```
##  [1] 3628800  362880   40320    5040     720     120
##  [7]      24       6       2       1
```

Missing values

If the vector on which we are calculating `sum` etc., has missing values, we will have to use argument `na.rm = TRUE` in these functions (Check documentation of these functions individually once). See these examples -

```
x <- c(1:50, NA)
sum(x)
```

```
## [1] NA
```

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

```
## [1] 1275
```

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

```
## [1] 25.5
```

3.3 Some Statistical functions

Table 3.3: Some commonly used Statistical Functions

Function	Meaning	Input	Output
`sd()`	Returns standard deviation	One Vector, matrix, array	Vector having 1 element only
`var()`	Returns variance	One or more Vector, matrix, array	Vector having 1 element only

Function	Meaning	Input	Output
`median()`	Returns median value	One Vector, matrix, array	Vector having 1 element only
`range()`	Returns range	One Vector, matrix, array	Vector having 2 elements
`IQR()`	Computes interquartile range of the x values	One Vector, matrix, array	Vector having 1 element only
`quantile()`	Computes percentile of given values for the given probabilities in `probs` argument	One Vector, matrix, array	Named Vector having 5 elements by default, OR equal to the length of `probs` vector given

Examples-

```
median(1:100)
```

```
## [1] 50.5
```

```
range(1:100, 45, 789)
```

```
## [1]   1 789
```

```
quantile(1:100)
```

```
##     0%    25%    50%    75%   100%
##   1.00  25.75  50.50  75.25 100.00
```

```
quantile(0:100, probs = 1:10 / 10)
```

```
##  10%  20%  30%  40%  50%  60%  70%  80%  90% 100%
##   10   20   30   40   50   60   70   80   90  100
```

3.4 Functions related to sampling and probability distributions

Set the random seed with `set.seed()`

It is a way to specify the random seed which is an integer vector, containing the random number generator (RNG) state for random number generation in R. It

does not given any output but makes your code reproducible for further use.

Generate random numbers with `rnorm()` / `runif()` / `rpois()` etc.

Used to generate random numbers from normal, uniform and poisson distributions respectively. Of course there are numerous other functions not only to calculate random numbers but to calculate probability, density of these and other probability distributions (such as binomial, t), but those are beyond the scope of this book. E.g.

```
rnorm(n=10) #default mean is 0 and SD is 1
```

```
##  [1] -0.4289 -1.8543  0.9333  0.1046  2.5222  0.8488
##  [7] -0.4944 -0.2863 -1.5073  1.4500
```

```
rnorm(n=10) # notice these will produce different results each time.
```

```
##  [1]  0.3539 -0.6832  0.7927  0.6322  0.7212  1.6827
##  [7]  0.6983  0.4955 -0.4322  1.1345
```

```
# If however seed is fixed as above, these will be reproducible.
set.seed(123)
runif(10) # default min and max are 0 and 1 respectively
```

```
##  [1] 0.28758 0.78831 0.40898 0.88302 0.94047 0.04556
##  [7] 0.52811 0.89242 0.55144 0.45661
```

```
set.seed(123)
runif(10)
```

```
##  [1] 0.28758 0.78831 0.40898 0.88302 0.94047 0.04556
##  [7] 0.52811 0.89242 0.55144 0.45661
```

Random Sample with `sample()`

Used to take a sample of the specified `size` from the elements of x using either with or without replacement. E.g.

```
set.seed(123)
sample(LETTERS, 5, replace = FALSE)
```

```
## [1] "O" "S" "N" "C" "J"
```

```
set.seed(111)
sample(LETTERS, 15, replace = TRUE)
```

```
##  [1] "N" "T" "S" "O" "Y" "E" "C" "H" "Z" "Q" "M" "J"
## [13] "D" "O" "H"
```

If the sampling is proportionate to given `probabilities` the same can be provided in `prob` argument.

```
set.seed(12)
sample(LETTERS, 5, replace = FALSE, prob = 1:26)
```

```
## [1] "Z" "K" "F" "V" "X"
```

3.5 Other Mathematical functions

Progressive calculations with `cumsum()` /`cumprod()`

Used to calculate running total or product. Output vector length will be equal to that of input vector.

```
cumsum(1:10)
```

```
##  [1]  1  3  6 10 15 21 28 36 45 55
```

```
cumprod(-5:5)
```

```
##  [1]   -5   20  -60  120 -120    0    0    0    0    0
## [11]    0
```

Other similar functions like `cummax()` (cumulative maximum) and `cummin()` may also be useful.

```
set.seed(1)
x <- sample(1:100, 10)
cummin(x)
```

```
##  [1] 68 39  1  1  1  1  1  1  1  1
```

```
cummax(x)
```

```
##  [1] 68 68 68 68 87 87 87 87 87 87
```

Progressive difference `diff()`

Used to calculate running difference (difference between two consecutive elements) in the given numeric vector. Output will be shorter by one element. E.g.

```
set.seed(123)
x <- rnorm(10)
x
```

```
##  [1] -0.56048 -0.23018  1.55871  0.07051  0.12929
##  [6]  1.71506  0.46092 -1.26506 -0.68685 -0.44566
```

```
diff(x)
```

```
## [1]  0.33030  1.78889 -1.48820  0.05878  1.58578
## [6] -1.25415 -1.72598  0.57821  0.24119
```

```
length(diff(x))
```

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

3.6 String Manipulation functions

Concatenate strings with `paste()` and `paste0()`

R's inbuilt function `paste()` concatenates each element of one or more vectors given as argument. Argument `sep` is used to provide separator is any, which by default is a space i.e. `" "`. On the other `sep` argument is not available in `paste0` which thus concatenates elements without any separator.

```
paste(LETTERS, letters)
```

```
##  [1] "A a" "B b" "C c" "D d" "E e" "F f" "G g" "H h"
##  [9] "I i" "J j" "K k" "L l" "M m" "N n" "O o" "P p"
## [17] "Q q" "R r" "S s" "T t" "U u" "V v" "W w" "X x"
## [25] "Y y" "Z z"
```

```
paste0(letters, '_', 1:26) # check replication here
```

```
##  [1] "a_1"  "b_2"  "c_3"  "d_4"  "e_5"  "f_6"  "g_7"
##  [8] "h_8"  "i_9"  "j_10" "k_11" "l_12" "m_13" "n_14"
## [15] "o_15" "p_16" "q_17" "r_18" "s_19" "t_20" "u_21"
## [22] "v_22" "w_23" "x_24" "y_25" "z_26"
```

Note: that both `paste` and `paste0` returns vector with length equal to length of larger vector. **Thus if the requirement is to concatenate each of the element in the given vector(s), use another argument `collapse`. See this example.**

```
paste0(letters, 1:26, collapse = '+')
```

```
## [1] "a1+b2+c3+d4+e5+f6+g7+h8+i9+j10+k11+l12+m13+n14+o15+p16+q17+r18+s19+t20+u21+v22+
```

Functions `startsWith()` / `endsWith()`

To check whether the given string vector say `x` start or end with string (entries of) `prefix` or `suffix` we can use `startsWith(x, prefix)` or `endsWith(x, suffix)` respectively. E.g.

```
x <- c('apples', 'oranges', 'apples and oranges', 'oranges and apples', 'apricots')
startsWith(x, 'apples')
```

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

```
startsWith(x, 'ap')
```

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

```
endsWith(x, 'oranges')
```

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

Note that both these functions return logical vectors having same length as `x`.

Check number of characters in string vector using `nchar()`

To count the number of characters in each of the element in string vector, say `x`, we can use `nchar(x)` which will return a vector of integer types. E.g.

```
nchar(x)
```

```
## [1]  6  7 18 18  8
```

```
y <- c('', ' ', '   ', NA)
nchar(y)
```

```
## [1]  0  1  3 NA
```

Change case using `toupper()` / `tolower()`

Changes the case of given vector to all UPPER or lower case respectively. Example-

```
x <- c('Andrew', 'Bob')
tolower(x)
```

```
## [1] "andrew" "bob"
```

```
toupper(x)
```

```
## [1] "ANDREW" "BOB"
```

Extract a portion of string using `substr()`

To extract the characters from a given vector say `x` from a given `start` position to `stop` position (both being integers) we will use `substr(x, start, stop)`. E.g.

```
substr(x, 2, 8)
```

```
## [1] "ndrew" "ob"
```

Split a character vector using `strsplit()`

To split the elements of a character vector `x` into sub-strings according to the matches to sub-string `split` within them. E.g.

```
strsplit(x, split = ' ')
```

```
## [[1]]
## [1] "Andrew"
##
## [[2]]
## [1] "Bob"
```

Notice that output will be of `list` type.

Replace portions of string vectors `sub()` / `gsub()`

These two functions are used to perform replacement of the first and all matches respectively. E.g.

```
sub(pattern = 'an', replacement = '12', x)
```

```
## [1] "Andrew" "Bob"
```

```
gsub(pattern = 'an', replacement = '12', x)
```

```
## [1] "Andrew" "Bob"
```

Match patterns using `grep()` / `grepl()` / `regexpr()` / `gregexpr()`

These functions are used to match string passed as argument `pattern` under a string vector. These four however, differ in output/results. E.g.

```
grep(pattern = 'an', x) # will give indices.  Output will be integer vector and length
```

```
## integer(0)
```

```
grepl(pattern = 'an', x) # will give a logical vector of same length as `x`
```

```
## [1] FALSE FALSE
```

```
regexpr(pattern = 'an', x) # output will have multiple attributes
```

```
## [1] -1 -1
## attr(,"match.length")
## [1] -1 -1
## attr(,"index.type")
## [1] "chars"
## attr(,"useBytes")
## [1] TRUE
```

Note that `regexpr()` outputs the character position of first instance of pattern match within the elements of given vector. `gregexpr()` is same as `regexpr()` but finds all instances of pattern. Output will be in `list` format. E.g.

```
gregexpr(pattern = 'an', x)
```

```
## [[1]]
## [1] -1
## attr(,"match.length")
## [1] -1
## attr(,"index.type")
## [1] "chars"
## attr(,"useBytes")
## [1] TRUE
##
## [[2]]
## [1] -1
## attr(,"match.length")
## [1] -1
## attr(,"index.type")
## [1] "chars"
## attr(,"useBytes")
## [1] TRUE
```

3.7 Other functions

Transpose a matrix using `t()`

Used to return transpose of given matrix. E.g.

```
mat <- outer(1:5, 1:5, FUN = \(x, y) paste0('A', x, y))
mat
```

```
##      [,1]  [,2]  [,3]  [,4]  [,5]
## [1,] "A11" "A12" "A13" "A14" "A15"
```

```
## [2,] "A21" "A22" "A23" "A24" "A25"
## [3,] "A31" "A32" "A33" "A34" "A35"
## [4,] "A41" "A42" "A43" "A44" "A45"
## [5,] "A51" "A52" "A53" "A54" "A55"
```

```
t(mat)
```

```
##      [,1]  [,2]  [,3]  [,4]  [,5]
## [1,] "A11" "A21" "A31" "A41" "A51"
## [2,] "A12" "A22" "A32" "A42" "A52"
## [3,] "A13" "A23" "A33" "A43" "A53"
## [4,] "A14" "A24" "A34" "A44" "A54"
## [5,] "A15" "A25" "A35" "A45" "A55"
```

Generate a frequency table using `table()`

Returns a frequency/contingency table of the counts at each combination of factor levels. E.g.

```
set.seed(123)
x <- sample(LETTERS[1:5], 100, replace = TRUE)
table(x)
```

```
## x
##  A  B  C  D  E
## 21 20 23 17 19
```

If more than one argument is passed-

```
set.seed(1234)
df <- data.frame(State_code = x,
                 Code2 = sample(LETTERS[11:15], 100, replace = TRUE))
my_table <- table(df$State_code, df$Code2)
my_table
```

```
##
##    K L M N O
##   A 5 5 4 4 3
##   B 4 3 6 2 5
##   C 6 3 3 6 5
##   D 2 2 4 6 3
##   E 2 6 4 4 3
```

Generate proportion of frequencies using `prop.table()`

This function takes a table object as input and calculate the proportion of frequencies.

```
prop.table(my_table)
```

```
##
##        K    L    M    N    O
##   A 0.05 0.05 0.04 0.04 0.03
##   B 0.04 0.03 0.06 0.02 0.05
##   C 0.06 0.03 0.03 0.06 0.05
##   D 0.02 0.02 0.04 0.06 0.03
##   E 0.02 0.06 0.04 0.04 0.03
```

Column-wise or Row-wise sums using `colSums()` / `rowSums()`

Used to sum rows/columns in a matrix/data.frame. E.g.

```
# Row sums
rowSums(my_table)
```

```
##  A  B  C  D  E
## 21 20 23 17 19
```

```
# Col sums
colSums(my_table)
```

```
##  K  L  M  N  O
## 19 19 21 22 19
```

Note Similar to `colSums()`/ `rowSums()` we also have `colMeans()` and `rowMeans()`.

```
rowMeans(my_table)
```

```
##   A   B   C   D   E
## 4.2 4.0 4.6 3.4 3.8
```

Extract unique values using `unique()`

Used to extract only unique values/elements from the given vector. E.g.

```
unique(x) # note the output
```

```
## [1] "C" "B" "E" "D" "A"
```

Check if two vectors are identical using `identical()`

Used to check whether two given vectors/objects are identical.

```
identical(unique(x), LETTERS)
```

```
## [1] FALSE
```

Retreive duplicate items in a vector using `duplicated()`

Used to check which elements have already appeared in the vector and are thus duplicate.

```
set.seed(123)
x <- sample(LETTERS[1:5], 8, replace = TRUE)
x
```

```
## [1] "C" "C" "B" "B" "C" "E" "D" "A"
```

```
duplicated(x)
```

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

Generate sequences using other objects with `seq_len()` / `seq_along()`

Used to generate sequence of given integer length starting with 1, or with length equal to given vector, respectively. E.g.

```
seq_len(5)
```

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

```
x <- c('Andrew', 'Bob')
seq_along(x)
```

```
## [1] 1 2
```

Divide a vector into categories (factor) using `cut()`

The function divides the range of `x` into intervals and codes the values in `x` according to which interval they fall. The leftmost interval corresponds to level one, the next leftmost to level two and so on. The output vector will be of type `factor`.

Example-1:

```
x <- c(1,2,3,4,5,2,3,4,5,6,7)
cut(x, 3)
```

```
##  [1] (0.994,3] (0.994,3] (0.994,3] (3,5]     (3,5]
##  [6] (0.994,3] (0.994,3] (3,5]     (3,5]     (5,7.01]
## [11] (5,7.01]
## Levels: (0.994,3] (3,5] (5,7.01]
```

Example-2:

```
cut(x, 3, dig.lab = 1, ordered_result = TRUE)
```

```
##  [1] (1,3] (1,3] (1,3] (3,5] (3,5] (1,3] (1,3] (3,5]
##  [9] (3,5] (5,7] (5,7]
## Levels: (1,3] < (3,5] < (5,7]
```

Note: that the output `factor` above is ordered.

Scale the columns of a matrix using `scale()`

Used to scale the columns of a numeric matrix.

```
x <- matrix(1:10, ncol = 2)
x
```

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

```
scale(x)
```

```
##         [,1]    [,2]
## [1,] -1.2649 -1.2649
## [2,] -0.6325 -0.6325
## [3,]  0.0000  0.0000
## [4,]  0.6325  0.6325
## [5,]  1.2649  1.2649
## attr(,"scaled:center")
## [1] 3 8
## attr(,"scaled:scale")
## [1] 1.581 1.581
```

Note: The output will always be of a matrix type with two more attributes. See this example

```
scale(1:5)
```

```
##         [,1]
## [1,] -1.2649
## [2,] -0.6325
## [3,]  0.0000
## [4,]  0.6325
## [5,]  1.2649
## attr(,"scaled:center")
## [1] 3
## attr(,"scaled:scale")
## [1] 1.581
```

Output the results using `cat()`

Outputs the objects, concatenating the representations. `cat` performs much less conversion than `print`.

```
cat('ABCD')
```

```
## ABCD
```

Note: that indices are now *not printed.* `cat` may print objects also. Example-2:

```
cat(month.name)
```

```
## January February March April May June July August September October November Decembe
```

`cat` is useful to print *special characters.* Example-3:

```
cat('Budget Allocation is \u20b91.5 crore')
```

```
## Budget Allocation is ₹1.5 crore
```

Sort a vector using `sort()`

Used to **sort** the given vector. Example-1:

```
vec <- c(5, 8, 4, 1, 6)
sort(vec)
```

```
## [1] 1 4 5 6 8
```

Argumemt `decreasing = TRUE` is used to sort the vector in descending order instead of default ascending order. Example-2:

```
sort(vec, decreasing =  TRUE)
```

```
## [1] 8 6 5 4 1
```

Arrange the elements of a vector using `order()`

In contrast to `sort()` explained above, `order()` returns the indices of given vector in ascending order. Example

```
order(vec)
```

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

Thus, `sort(vec)` will essentially perform the same operations as `vec[order(vec)]`. We may check-

```
identical(vec[order(vec)], sort(vec))
```

```
## [1] TRUE
```

Check structure using `str()`

The short `str` is not to be confused with strings as it instead is short for `structure`. Thus, `str` returns structure of given object. Example

```
str(vec)
```

```
##  num [1:5] 5 8 4 1 6
```

Extremely useful when we need to inspect data frames.

```
str(iris)
```

```
## 'data.frame':    150 obs. of  5 variables:
##  $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
##  $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
##  $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
##  $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
##  $ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1 1 .
```

Generate a summary using `summary()`

In addition to `str` explained above, `summary()` is also useful is getting result summaries of given objects. Example-1: When given object is vector

```
summary(vec)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##     1.0     4.0     5.0     4.8     6.0     8.0
```

We observe that when numeric vector is passed, it produces quantile summary. Example-2: When input object is data frame.

```
summary(iris)
```

```
##   Sepal.Length   Sepal.Width    Petal.Length
##  Min.   :4.30   Min.   :2.00   Min.   :1.00
##  1st Qu.:5.10   1st Qu.:2.80   1st Qu.:1.60
##  Median :5.80   Median :3.00   Median :4.35
##  Mean   :5.84   Mean   :3.06   Mean   :3.76
##  3rd Qu.:6.40   3rd Qu.:3.30   3rd Qu.:5.10
##  Max.   :7.90   Max.   :4.40   Max.   :6.90
##   Petal.Width        Species
##  Min.   :0.1   setosa    :50
##  1st Qu.:0.3   versicolor:50
##  Median :1.3   virginica :50
##  Mean   :1.2
##  3rd Qu.:1.8
##  Max.   :2.5
```

Chapter 4

Control statements and Custom Functions

In the previous chapter we learnt about many of the useful pre-built functions in R. In this chapter we will learn how to create customized functions suited to our needs.

Though these are core concepts of a programming language, yet a reading to this chapter is advised for better understanding and better application while using r for data analytics.

4.1 Control flow/Loops

`if else`

The basic form(s) of `if else` statement in R, are-

```
if (test) do_this_if_true
if (test) do_this_if_true else else_do_this
```

So, if `test` is true, `do_this_if_true` will be performed and optionally if `test` is not true `else_do_this` will be performed. See this example-

```
x <- 50
if(x < 10){
  'Smaller than 10'
} else {
  '10 or more'
}
```

```
## [1] "10 or more"
```

Note that the `if`/`if else` are evaluated for a single `TRUE` or `FALSE` i.e. this control flow is **not vectorised** as we have in the case of `ifelse()` function which was vectorised.

`for` loop

The `for` loops in r are used to iterate over *given* items. So, the basic structure of these loops are -

```
for(item in vector) perform_some_action

# OR

for(item in vector) {
  perform_some_action
}
```

Thus, for each item in `vector`, `perform_some_action` is called once; updating the value of item each time. This can be understood by the following simple example-

```
for(i in 1:3){
  print(i)
}
```

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

Conventionally `i` has been used in above example to iterate over given vector `1:3`, however any other symbol may also be used.

```
for(item in 1:3){
  print(item)
}
```

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

If we use the name of any existing variable as `item` to iterate over the given object, `for loop` assigns the `item` to the current environment, overwriting any existing variable with the same name. See this example -

```
x <- 500
for(x in 1:3){
  # do nothing
}
x
```

```
## [1] 3
```

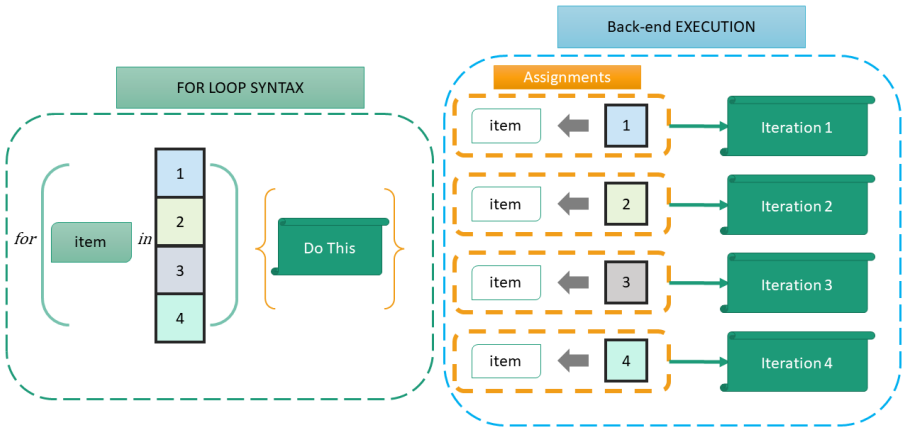


Figure 4.1: A Diagrammatic representation of For Loop

The idea can also used to iterate over any object any *number of times* as we want. See these two examples.

Example-1

```
my_names <- c('Andrew', 'Bob', 'Charles', 'Dylan', 'Edward')
# If we want first 4 elements
for(i in 1:4){
  print(my_names[i])
}
```

```
## [1] "Andrew"
## [1] "Bob"
## [1] "Charles"
## [1] "Dylan"
```

Example-2

```
# if we want all elements
for(i in seq_along(my_names)){
  print(my_names[i])
}
```

```
## [1] "Andrew"
## [1] "Bob"
## [1] "Charles"
## [1] "Dylan"
## [1] "Edward"
```

There are 2 ways to terminate any `for loop` early-

- `next` which exits the current iteration only
- `break` which breaks the entire loop.

See these examples.

Example-1

```
for(i in 1:5){
  if (i == 4){
    next
  }
  print(i)
}
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 5
```

Example-2

```
for(i in 1:5){
  if (i == 4){
    break
  }
  print(i)
}
```

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

`while` loop

We have seen that `for` loop is used to iterate over a set of `known values` or at least known number of times. If however, we want to perform some iterative action unknown number of times, we may use `while` loop which iterates till a given `condition` is `TRUE`. Another option is to have `repeat` loop which can be used to iterate any number of times till it encounters a `break`.

The basic syntax of `while` loop is-

```
while (condition) action
```

See these examples-

Example-1

```
i <- 1
while(i <=4){
  print(LETTERS[i])
  i <- i+1
}
```

```
## [1] "A"
## [1] "B"
## [1] "C"
## [1] "D"
```

We may check the value of `i` here after executing the loop

```
i
```

```
## [1] 5
```

Example-2:

```
i <- 4
while(i >=0){
  print(LETTERS[i])
  i <- i-1
}
```

```
## [1] "D"
## [1] "C"
## [1] "B"
## [1] "A"
## character(0)
```

Note: We have to make sure that the statements inside the brackets modify the `while condition` so that sooner or later the given condition is no longer `TRUE` otherwise the loop will never end and will go on forever.

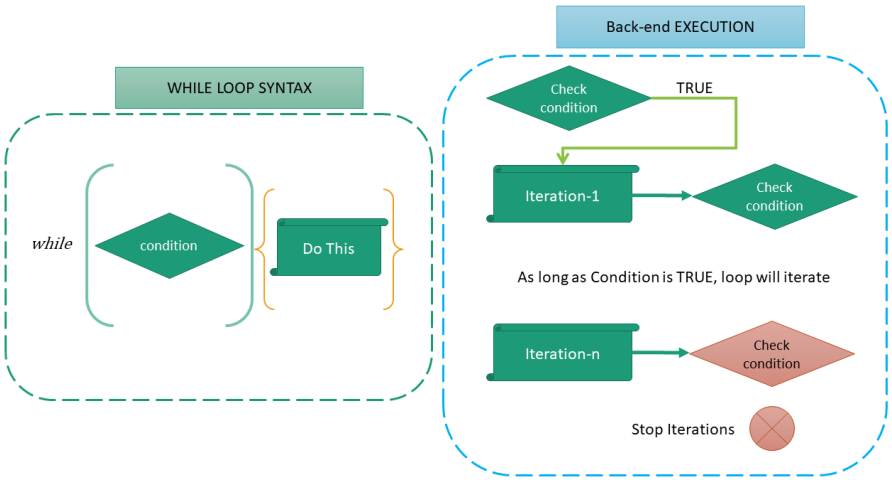


Figure 4.2: Author's illustration of While Loop

Looping in R can be inefficient and time consuming when you're processing the rows or columns of large data-sets. Even one of greatest feature in R is its parallel processing of vector objects. Thus, whenever possible, it's better to use R's built-in numerical and character functions in conjunction with the `apply` family of functions.(We will discuss these in detail in the chapter related to functional programming)

4.2 Custom Functions

One of R's greatest strengths is the user's ability to add functions. In fact, many of the functions in R are functions of existing functions. The structure of a function looks like this:

```
myfunctionname <- function(arg1, arg2, ... ){
  statements
  return(object)
}
```

Note: Objects in the function are local to the function. The object returned can be any data type, from scalar to list.

Let's take a look at an example. We will create a function which will take 3 numbers, will give an output by adding thrice of first, second and twice of third.

```
my_fun1 <- function(first,second,third){
  first*3+second+third*2
}
# let's check whether it is working as desired
my_fun1(3,1,10)
```

```
## [1] 30
```

- If the arguments provided are not named, it will take all arguments in the order these are defined.
- However, we can provide named arguments in any order. See this

```
my_fun1(second=3, first=1, third=10)
```

```
## [1] 26
```

- Partial matching of names are also allowed. Example

```
my_fun1(sec=3,fir=1,thi=10)
```

```
## [1] 26
```

- We can also provide default values to any argument. These default values are however, overridden when specific values are given. See this example.

```
# let's create a new function which adds twice the second argument to first argument, which in tu
my_fun2 <- function(first=10, second){
  first+second*2
}
my_fun2(second = 10)
```

```
## [1] 30
```

```
my_fun2(1, 10)
```

```
## [1] 21
```

- There may be functions which do not require any argument. See this example

```
my_fun3 <- function(){
  print('Hi')
}
my_fun3()
```

```
## [1] "Hi"
```

Special argument ellipsis ...

While searching for help of a function in r, you may have came across something like this `sum(..., na.rm = FALSE)`. The three dots `...` here are referred to as ellipsis. Basically it means that the function is designed to take any number of named or unnamed arguments.

Thus it means we can provide any number of arguments in place of `....` Now the point to be noted here is that values to all agruments occurring after `...` must only be named. See this example-

```
sum(1:100, NA, TRUE)
```

```
## [1] NA
```

```
sum(1:100, NA, na.rm = TRUE)
```

```
## [1] 5050
```

Now we can even use these three dots in our own custom functions. Just unpack these before writing the actual statement for that function. See this simple example-

```
my_ellipsis_func <- function(...){
  l <- list(...) # unpack ellipsis
  length(l) # return length of l
}
my_ellipsis_func(1:10, 11:20, 'a string') # we are passing three arguments
```

```
## [1] 3
```

Environment issues

- Any of the argument values are not saved/updated in global environment. See this example

```
x <- 10
my_fun4 <- function(x){
  x*2
}
my_fun4(2)
```

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

```
x
```

```
## [1] 10
```

- Even if we create another variable inside the function, that variable is not available outside that function's environment.

```
y <- 5
my_fun5 <- function(){
  y <- 1
  return(y)
}
my_fun5()
```

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

```
y
```

```
## [1] 5
```

- If however, we want to create a variable (or update existing variable) inside the function intentionally, we may use `forced assignment` denoted as `<<-`. See this example

```
y <- 5
my_fun5 <- function(){
  y <<- 1
  return(y)
}
my_fun5()
```

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

```
y
```

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

- As already stated, we can create object of any type using a custom function.

```
my_list_fun <- function(x){
  list(sum=sum(x),
       mean = mean(x),
       sd = sd(x))
}
my_list_fun(1:10)
```

```
## $sum
## [1] 55
##
## $mean
## [1] 5.5
##
## $sd
## [1] 3.028
```

4.3 Pipes

Now here I would like to introduce you with the concept of pipes in R. There are two types of pipes used-

- `|>` is native pipe of R. It was introduced in R version 4.1
- `%>%` pipe introduced in `magrittr` package(Bache and Wickham, 2022), now part of `tidyverse` which we will use extensively in our data analysis tasks.

Actually `%>%` is predecessor to native R's pipe `|>`. The pipes are powerful tools for clearly expressing a sequence of operations that transform an object, without the need of actually creating that object in each step. Let us understand this concept with the following example. Suppose, we have to three functions say `FIRST` , `SECOND` and `THIRD` to an object `OBJ` in sequence. So the order of operations would either be like-

```
THIRD(SECOND(FIRST(OBJ)))
```

or with creating intermediate objects, when instead we actually do not need those intermediate objects.

```
OBJ1 <- FIRST(OBJ)
OBJ2 <- SECOND(OBJ1)
OBJ3 <- THIRD(OBJ2)
```

Here actually we do not require `OBJ1` and `OBJ2`. So in these cases we either have to compromise with the readability of code i.e. inside out or have to create unwanted objects. Pipes actually mitigate both these issues simultaneously. With pipes we can write above operations as either of these -

```
OBJ1 |> FIRST() |> SECOND() |> THIRD()
OBJ1 %>% FIRST() %>% SECOND() %>% THIRD()
```

A diagrammatic representation is given in figure 4.3.

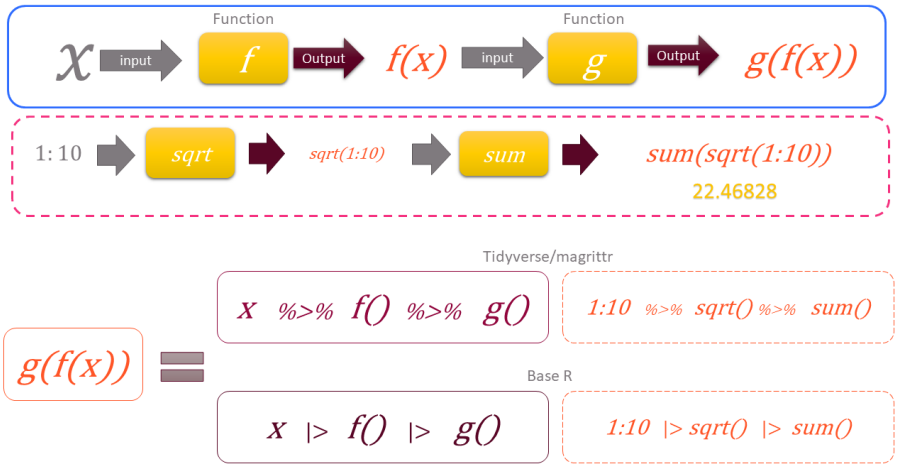


Figure 4.3: A diagrammtic illustation of Pipe concept in base R and tidyverse

Now two questions may arise here-

1. What if there are multiple arguments to be passed in any of the operations?
2. Is there any difference between the two pipes? If yes, which is better OR what are the pros and cons of each?

To answer these questions, we will discuss both pipes separately.

4.3.1 Magrittr/Dplyr pipe %>%

Pipes usually pass result of previous operation silently into first argument of next/right expression. So `data %>% filter(col == 'A')` means `filter(data, col=='A')`. But there may be cases when result of previous (LHS) expression is required to be passed on second or other argument in RHS expression. A simple example may be of function `lm`, where `data` argument is second argument. In such cases we can make use special placeholder `.` as result of LHS specifically. In other words aforesaid filter example can be written with placeholder as `data %>% filter(. , col == 'A')`. Now using this placeholder we can use result of LHS wherever we want. See this example

```
iris %>% lm(Sepal.Length ~ Sepal.Width, data = .)
```

```
##
## Call:
## lm(formula = Sepal.Length ~ Sepal.Width, data = .)
##
## Coefficients:
## (Intercept)  Sepal.Width
##       6.526       -0.223
```

Thus `x %>% f(y)` is equivalent to `f(x, y)` but `x %>% f(y, .)` is equivalent to `f(y, x)`.

4.3.2 Base R pipe |> (Version 4.2.0 +)

R version 4.2.0 introduced concept of placeholder `_` similar to dplyr/magrittr, but with a few differences.

- The argument where `_` is to be used, must be named. So `f(y, z = x)` can be written as `x |> f(y, z= _)`.

```
iris |> lm(Sepal.Length ~ Sepal.Width, data = _)
```

```
##
## Call:
## lm(formula = Sepal.Length ~ Sepal.Width, data = iris)
##
## Coefficients:
## (Intercept)  Sepal.Width
##       6.526       -0.223
```

The requirement of named argument is not there in dplyr pipe. So essentially, `iris %>% lm(Sepal.Length ~ Sepal.Width, .)` will also work. But in base R `iris |> lm(Sepal.Length ~ Sepal.Width, _)` would not work and throw an error. Thus, in cases where the argument of placeholder is not named, we have to use anonymous function. Thus we have write like this-

```
iris |> {\(.x) lm(Sepal.Length ~ Sepal.Width, .x)}()
```

```
##
## Call:
## lm(formula = Sepal.Length ~ Sepal.Width, data = .x)
##
## Coefficients:
## (Intercept)  Sepal.Width
##       6.526       -0.223
```

Type ?|> in console and see help page for more details.

Chapter 5

Concepts of Functional Programming

What is functional programming?

Conceptually functional programming philosophy is based on lambda calculus. Lambda calculus is a framework developed by Alonzo Church[1] to study computations with functions.

Functional programming is a programming paradigm in which we try to bind everything in pure mathematical functions style. It is a declarative type of programming style. Its main focus is on **what to solve** in contrast to an imperative style where the main focus is **how to solve**. For a more elaborated definition readers may see this wikipedia link. Simply putting functional programming is like doing something repeatedly but in declarative style. Here, functions are the primary method with which we carry out tasks. All actions are just implementations of functions we are using.

Functional programming use high order functions. A high order function is actually a function that accepts a function as an argument, or returns a function; in short, function that operates upon a function. We have already seen one such example may be without noticing it, `args()` function take a function as an argument and in turn return its arguments.

Let us learn a bit more here.

[1] Alan Turing, who created Turing machine which in turn laid the foundation of imperative programming style, was a student of **Alonzo Church**.

Usage of functional programming in R

Strictly speaking R is not a functional programming language. But we have already seen that one of the greatest strengths of R is parallel operations on vectors. In fact we need functional programming where concurrency or parallelism is required. Till now we have seen that most of the functions work on all atomic objects (vectors, matrices, arrays, etc.), but what about working of these functions on recursive objects i.e. lists? Check this example (in your console)-

```
list1 <- list(50000, 5000, 56)
sum(list1)
```

Of course, we can solve the above problem by using for loops. See

```
list1 <- list(50000, 5000, 56)
# for loop strategy
x <- c()
for(i in seq_along(list1)){
  x[i] <- list1[[i]]
}
sum(x)
```

```
## [1] 55056
```

Consider another list, where we want to calculate mean of each element of that list.

```
list2 <- list(
  1:10,
  11:20,
  21:30
)
```

Of course, we may use a for loop again, but in R these operations can be done easily with *apply* group of functions, which are one of the most famous and most used features in R.

5.1 `apply` family of functions

First of these functions is `apply()` which works on matrices/data frames.

5.1.1 Function `apply()`

The basic syntax of `apply` is

```
apply(m, MARGIN, FUN, f_args)
```

where

- `m` is the matrix
- `MARGIN` is the dimension. If we want to *apply* function to each row then use `1` or else if it is to be column-wise use `2`
- `FUN` is the desired function which we want to apply
- `f_args` are the optional set of arguments, if needed to be supplied to `fun`.

An illustrative construction of `apply` function can be seen in 5.1.

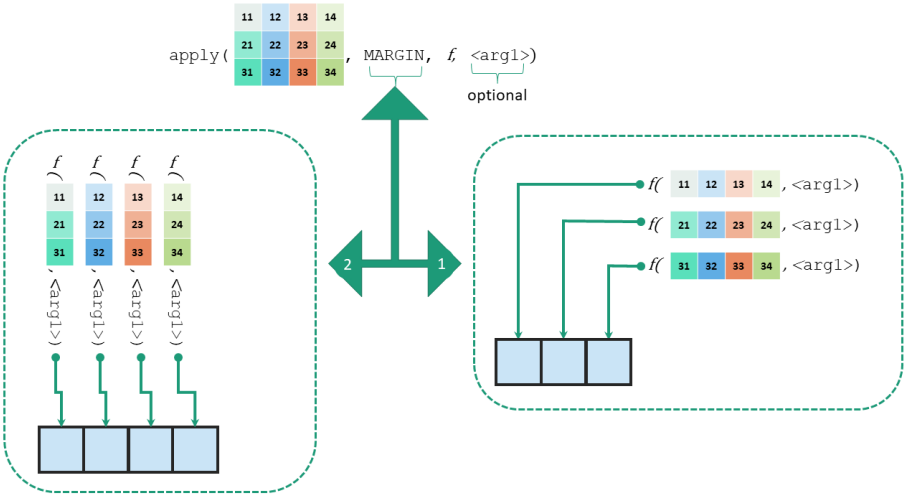


Figure 5.1: Illustration of function apply

Check this example

```
(mat <- matrix(1:10, nrow = 5))
```

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

```
apply(mat, 1, mean)
```

```
## [1] 3.5 4.5 5.5 6.5 7.5
```

Note: `rowMeans(mat)` in above example would have given similar results, but for sake of simplicity we have provided a simplest example.

Further note that we may also write our own customised function in the argument. See this another example, where we will take sum of squares of each row. We may define our own custom function for the purpose and then `apply` it.

```
my_fun <- function(x){
  sum(x^2)
}
apply(mat, 1, my_fun)
```

```
## [1]  37  53  73  97 125
```

The need to writing a custom function before hand may be eliminated if the function so defined is not be used further. We may write anonymous function directly in the `apply` syntax -

```
apply(mat, 1, FUN = function(x) sum(x^2))
```

```
## [1]  37  53  73  97 125
```

In R version 4.1 and onwards R has devised shorthand style of defining inline custom functions, where we can write backslash i.e. \ instead of writing `function`. We could have written above expression as-

```
apply(mat, 1, FUN = \(x) sum(x^2))
## [1]  37  53  73  97 125
```

`apply()` need not necessarily output vectors only

If `FUN` applied on rows/columns of matrix outputs vector of length more than 1, the output will be in matrix format. But the thing to note here is that matrix will be displayed columnwise always irrespective of fact whether `MARGIN` is `1` or `2`. As an easy example we could have shown this using function like `sqrt`, but `apply(matrix, MARGIN, sqrt)` will work like `sqrt(matrix)` only. So let's take a different example. Suppose we want to calculate `column-wise` cumulative sum in a given matrix.

```
apply(mat, 2, cumsum)
```

```
##      [,1] [,2]
## [1,]    1    6
## [2,]    3   13
## [3,]    6   21
## [4,]   10   30
## [5,]   15   40
```

The output here is eaxctly what was desired. But what if, our requirement was to take `row-wise` cumulative sum?

```
apply(mat, 1, cumsum)
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    2    3    4    5
## [2,]    7    9   11   13   15
```

It may now be noticed that the output is actually *transpose* of what we were expecting. Actually the output of each iteration of `apply` function is displayed in one column always. Let us check our understanding with one more example taking function which may give output that is not dependent on input vector length.

```
set.seed(1)
apply(mat, 1, sample, 4, TRUE)
```

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

Thus we may conclude that-

- If `FUN` outputs a scalar, the output of `apply` will be a vector of **length** equal to
  - number of `rows` in input matrix given that `MARGIN` selected in 1,
  - number of `columns` in input matrix given that `MARGIN` selected in 2.
- if `FUN` outputs a vector(of length >1) then output of `apply` will be a matrix having **number of columns** equal to -

- number of `rows` in input matrix given that `MARGIN` selected in 1,
- number of `columns` in input matrix given that `MARGIN` selected in 2.

These have been tabulated in table 5.1.

Table 5.1: Relation between input and output data structure in `apply`

Input matrix $m * n$	`MARGIN = 1`	`MARGIN = 2`
FUN gives scalar	Vector size m	Vector size n
FUN gives vector size p	Matrix $p * m$	Matrix $p * n$

We may thus have to be careful while getting the output from `apply` function as it may lead to introduction of bug in our code.

`apply()` on data frames

Now we know that data frames despite being special type of lists also behave like matrices, we may use `apply` on data frames too. See this example.

```
(my_df <- as.data.frame(mat))
```

```
##   V1 V2
## 1  1  6
## 2  2  7
## 3  3  8
## 4  4  9
## 5  5 10
```

```
apply(my_df, 2, sum)
```

```
## V1 V2
## 15 40
```

5.1.2 Function `lapply()`

Another cousin of apply is `lapply` which can thought of `apply` to `lists`. So as the name suggests it is applied on lists instead of matrices. Now since `data frame` is also a list `lapply` can be applied on these. The basic syntax of `lapply()` is -

```
lapply(l, FUN, f_args)
```

where

- `l` is the list
- `FUN` is the desired function which we want to apply
- `f_args` are the optional set of arguments, if needed to be supplied to `fun`.

It may be noted that `MRAGIN` argument is not available here. See these examples.

```
lapply(my_df, sum)
```

```
## $V1
## [1] 15
##
## $V2
## [1] 40
```

Now you may have noticed two things here -

1. The output is of `list` type.
2. Unlike `apply` as `MARGIN` is not passed/available here, it applies `FUN` to every element of list. When we consider any `data.frame` as a list its each column is a separate element of that list. So `FUN` cannot be applied to `rows` in a `data.frame`.

Thus `lapply()` -

- loops over a list, iterating over each element in that list
- then *applies* the function `FUN` to each element
- and then returns a list.

Example-2: Let's try to find type of each column in a given data frame.

```
lapply(iris, typeof)
```

```
## $Sepal.Length
## [1] "double"
##
## $Sepal.Width
## [1] "double"
##
## $Petal.Length
## [1] "double"
##
```

```
## $Petal.Width
## [1] "double"
##
## $Species
## [1] "integer"
```

Similar to `apply` we can define `FUN` inline here (anonymously) also. Example-3:

```
lapply(my_df, \(a) a^2)
```

```
## $V1
## [1]  1  4  9 16 25
##
## $V2
## [1]  36  49  64  81 100
```

Example-4:

```
set.seed(1)
lapply(1:4, runif, min=0, max=10)
```

```
## [[1]]
## [1] 2.655
##
## [[2]]
## [1] 3.721 5.729
##
## [[3]]
## [1] 9.082 2.017 8.984
##
## [[4]]
## [1] 9.4468 6.6080 6.2911 0.6179
```

Note that even if `lapply` is applied over a vector, it returns a list only.

5.1.3 Function `sapply()`

There is not much of the difference between `lapply()` and `sapply()`, as `sapply` is actually simplified `lapply`. It simplifies the argument as much as possible.

Example:

```
sapply(my_df, sum)
```

```
## V1 V2
## 15 40
```

5.2 Other loop functions

5.2.1 Function `replicate()`

Function `replicate()`replicate) is used for repeated evaluation of an expression. Syntax is

```
replicate(n, expr, simplify = "array")
```

where -

- `n` is integer denoting the number of replications
- `expr` is the expression to evaluate repeatedly
- `simplify` takes either 'character' or 'logical' to value to indicate whether the results should be simplified.

Example:

```
set.seed(123)
# Default value of simplify will simplify the results as much possible
replicate(5, runif(3))
```

```
##        [,1]    [,2]   [,3]   [,4]   [,5]
## [1,] 0.2876 0.88302 0.5281 0.4566 0.6776
## [2,] 0.7883 0.94047 0.8924 0.9568 0.5726
## [3,] 0.4090 0.04556 0.5514 0.4533 0.1029
```

```
# Notice the difference with simplify=FALSE
replicate(3, runif(5), simplify = FALSE)
```

```
## [[1]]
## [1] 0.89982 0.24609 0.04206 0.32792 0.95450
##
## [[2]]
## [1] 0.8895 0.6928 0.6405 0.9943 0.6557
##
## [[3]]
## [1] 0.7085 0.5441 0.5941 0.2892 0.1471
```

5.2.2 Function `split()`

The `split()`split() function) function takes object (vector or other) and splits it into groups determined by a given factor. The basic syntax is-

```
split(x, f, drop=FALSE, ...)
```

where

- `x` is input object - `vector` or `list` or `data.frame`
- `f` is a factor or a list of factors. If a factor is not provided, it will be coerced to factor.
- `drop` argument indicates whether empty factors should be dropped.

Example: (To divide the given list by alternate elements)-

```
split(LETTERS, rep(1:2, 13))
```

```
## $`1`
##  [1] "A" "C" "E" "G" "I" "K" "M" "O" "Q" "S" "U" "W"
## [13] "Y"
##
## $`2`
##  [1] "B" "D" "F" "H" "J" "L" "N" "P" "R" "T" "V" "X"
## [13] "Z"
```

Example-2: Find out sum of every odd and even number from 1:100-

```
split(1:100, (1:100) %% 2) |> lapply(sum)
```

```
## $`0`
## [1] 2550
##
## $`1`
## [1] 2500
```

Example-3: Find out mean of `mpg` column splitting the `mtcars` data by `cyl`

```
split(mtcars$mpg, mtcars$cyl) |> sapply(mean)
```

```
##     4     6     8
## 26.66 19.74 15.10
```

5.2.3 `tapply()`

The `tapply()` functiontapply() function) can be thought of combination of `split` and `sapply` for vectors, exactly as used in above example. It actually applies the function over subsets of a given vector. The basic syntax is-

```
tapply(X, INDEX, FUN = NULL, ..., default = NA, simplify = TRUE)
```

Where -

- `X` is a vector
- `INDEX` is factor or list of factors
- `FUN` is function to be applied
- `...` are other arguments, if any, of `FUN` to be passed
- `simplify` if TRUE simplifies the result.

See this example

```
tapply(mtcars$mpg, mtcars$cyl, mean)
```

```
##     4     6     8
## 26.66 19.74 15.10
```

Needless to say if `simplify` is `FALSE` the results will not be simplified. See this example-

```
# month-wise mean of temperatures from `airquality` data
tapply(airquality$Temp, airquality$Month, mean, simplify = FALSE)
```

```
## $`5`
## [1] 65.55
##
## $`6`
## [1] 79.1
##
## $`7`
## [1] 83.9
##
## $`8`
## [1] 83.97
##
## $`9`
## [1] 76.9
```

5.2.4 by() function

Thisby() function) works something like `tapply` but with the difference that input object here is `data.frame`. See this example

```
# Split the data by `cyl` column and subset first six rows only
by(mtcars, mtcars$cyl, head)
```

```
## mtcars$cyl: 4
##                 mpg cyl  disp hp drat    wt  qsec vs
## Datsun 710     22.8   4 108.0 93 3.85 2.320 18.61  1
## Merc 240D      24.4   4 146.7 62 3.69 3.190 20.00  1
## Merc 230       22.8   4 140.8 95 3.92 3.150 22.90  1
## Fiat 128       32.4   4  78.7 66 4.08 2.200 19.47  1
## Honda Civic    30.4   4  75.7 52 4.93 1.615 18.52  1
## Toyota Corolla 33.9   4  71.1 65 4.22 1.835 19.90  1
##                am gear carb
## Datsun 710      1    4    1
## Merc 240D       0    4    2
## Merc 230        0    4    2
## Fiat 128        1    4    1
## Honda Civic     1    4    2
## Toyota Corolla  1    4    1
## ------------------------------------------
## mtcars$cyl: 6
##                 mpg cyl  disp  hp drat    wt  qsec vs
## Mazda RX4      21.0   6 160.0 110 3.90 2.620 16.46  0
## Mazda RX4 Wag  21.0   6 160.0 110 3.90 2.875 17.02  0
## Hornet 4 Drive 21.4   6 258.0 110 3.08 3.215 19.44  1
## Valiant        18.1   6 225.0 105 2.76 3.460 20.22  1
## Merc 280       19.2   6 167.6 123 3.92 3.440 18.30  1
## Merc 280C      17.8   6 167.6 123 3.92 3.440 18.90  1
##                am gear carb
## Mazda RX4       1    4    4
## Mazda RX4 Wag   1    4    4
## Hornet 4 Drive  0    3    1
## Valiant         0    3    1
## Merc 280        0    4    4
## Merc 280C       0    4    4
## ------------------------------------------
## mtcars$cyl: 8
##                   mpg cyl  disp  hp drat   wt  qsec
## Hornet Sportabout  18.7   8 360.0 175 3.15 3.44 17.02
## Duster 360         14.3   8 360.0 245 3.21 3.57 15.84
## Merc 450SE         16.4   8 275.8 180 3.07 4.07 17.40
```

```
## Merc 450SL         17.3   8 275.8 180 3.07 3.73 17.60
## Merc 450SLC        15.2   8 275.8 180 3.07 3.78 18.00
## Cadillac Fleetwood 10.4   8 472.0 205 2.93 5.25 17.98
##                    vs am gear carb
## Hornet Sportabout   0  0    3    2
## Duster 360          0  0    3    4
## Merc 450SE          0  0    3    3
## Merc 450SL          0  0    3    3
## Merc 450SLC         0  0    3    3
## Cadillac Fleetwood  0  0    3    4
```

5.2.5 Specifying the output type with `vapply()`

Function `vapply()`vapply() function) works exactly like `sapply()` described above, with only difference that type of return value (output) has to be specifically provided through `FUN.VALUE` argument. Its syntax is -

```
vapply(X, FUN, FUN.VALUE, ..., USE.NAMES = TRUE)
```

- In the argument `FUN.VALUE` we have to provide the format type of output. See this example.

```
vapply(mtcars, max, FUN.VALUE = double(1))
```

```
##     mpg     cyl    disp      hp    drat      wt
##  33.900   8.000 472.000 335.000   4.930   5.424
##    qsec      vs      am    gear    carb
##  22.900   1.000   1.000   5.000   8.000
```

Through `FUN.VALUE = double(1)` we have specifically provided that our output should be of `double` type with length `1`. So in case we have to find out `range` of each column-

```
vapply(mtcars, range, FUN.VALUE = double(2))
```

```
##       mpg cyl  disp  hp drat    wt qsec vs am gear
## [1,] 10.4   4  71.1  52 2.76 1.513 14.5  0  0    3
## [2,] 33.9   8 472.0 335 4.93 5.424 22.9  1  1    5
##      carb
## [1,]    1
## [2,]    8
```

If we will try this function on a dataset having mixed type columns like `iris` dataset, `vapply` will throw an error.

```
vapply(iris, range, FUN.VALUE = double())
```

5.3 Functional Programming in purrr

Package purrr[2], which is part of core tidyverse, enhances R's functional programming (FP) toolkit by providing a complete and consistent set of tools for working with functions and vectors.

```
library(purrr)
```

```
##
## Attaching package: 'purrr'
```

```
## The following object is masked from 'package:magrittr':
##
##     set_names
```

5.3.1 Iterate over single list/vector with map_*() family of functions

This package has many families of functions; and most primary family is map family of functions. map_*() works nearly similar to vapply where we can control the type of output. The syntax style of each of these functions is nearly same, where these accept one object (list or vector) as .x argument, one function (or alternatively a formula) as .f argument; and outputs an object of specified type.

Example

```
map(mtcars, .f = sum)
```

```
## $mpg
## [1] 642.9
##
## $cyl
## [1] 198
##
## $disp
## [1] 7383
##
## $hp
## [1] 4694
##
## $drat
```

[2] https://purrr.tidyverse.org/

```
## [1] 115.1
##
## $wt
## [1] 103
##
## $qsec
## [1] 571.2
##
## $vs
## [1] 14
##
## $am
## [1] 13
##
## $gear
## [1] 118
##
## $carb
## [1] 90
```

Note that output type is list. If the output can be simplified to an atomic vector we can use either of these functions depending upon the output type of that vector.

- `map_lgl` for `logical` format
- `map_int` for `integer` format
- `map_dbl` for `double` format
- `map_chr` for `character` format

`map` always return a list. See these further examples.

Example-1:

```
map_dbl(mtcars, max)
```

```
##     mpg     cyl    disp      hp    drat      wt
##  33.900   8.000 472.000 335.000   4.930   5.424
##    qsec      vs      am    gear    carb
##  22.900   1.000   1.000   5.000   8.000
```

5.3.2 Iterate over two or more lists/vectors using `map2_*()`/ `pmap_*()` family

So far we have seen that `map_*()` family of functions are used to iterate over elements of a list. Even if extra lists/vectors are provided as extra arguments,

these are used as it is, in each iteration, as can be seen in first illustration in figure[3] 5.2.

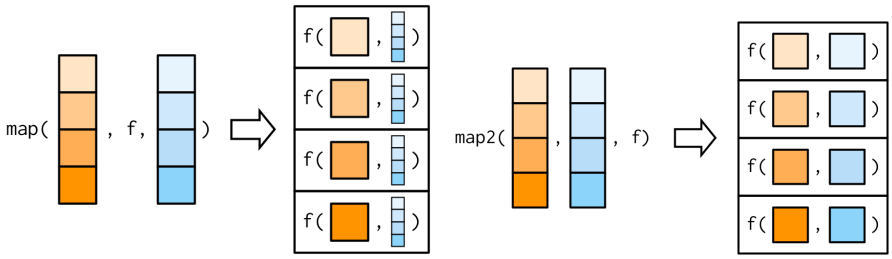


Figure 5.2: Working of map vs map2 family of functions Source Advanced R by Hadley Wickham

In order to iterate over two vectors/lists, we will however, need `map2_*()` family of functions (Refer second illustration in figure 5.2).

See the following example

```
x <- list(1, 2, 3)
y <- list(11, 12, 13)
map2(x, y, `*`)
```

```
## [[1]]
## [1] 11
##
## [[2]]
## [1] 24
##
## [[3]]
## [1] 39
```

Similarly, to iterate over multiple lists we will use `pmap_*()`, with the only difference being that here all the vectors/list should be collectively passed on to `pmap` in a list. This can be better understood with the illustration used by Hadley Wickham in his book. For reference see figure 5.3.

[3]Source: Advanced R by Hadley Wickham

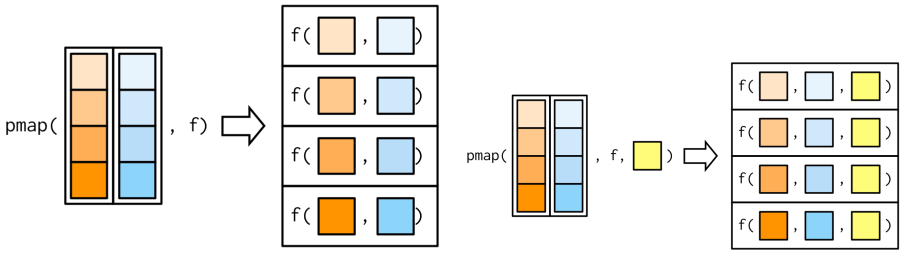


Figure 5.3: Working of pmap family of functions
Source Advanced R by Hadley Wickham

Chapter 6

Visualisation with ggplot2

6.1 Core concepts of grammar of graphics

GGPLOT2[1] (Wickham et al., 2023a) is the package developed by Hadley Wickham, which is based on concepts laid (2005) down by Leland Wilkinson in his *The Grammar of Graphics.*[2] Basically, a grammar of graphics is a framework which follows a layered approach to describe and construct visualizations or graphics in a structured manner. Even the letters gg in ggplot2 stand for grammar of graphics.

Hadley Wilkinson, in his paper titled **A Layered Grammar of Graphics**[3](2010) (**?**) proposed his idea of layered grammar of graphics in detail and simultaneously put forward his idea of *ggplot2* as an open source implementation framework for building graphics. Readers/Users are advised to check the paper as it describes the concept of grammar of graphics in detail. By the end of the decade the package progressed[4] to one of the most used and popular packages in R.

The relationship between the components explained in both the grammars can be illustrated with the following figure[5]. The components on the left have been put forward by Wilkinson whereas those on right were proposed by Wickham. It may be seen that `TRANS` has no relation in `ggplot2` as its role is played by in-built features of R.

Thus, to build a graphic having one or more dimensions, from a given data, we use *seven* major components -

[1] https://ggplot2.tidyverse.org/
[2] https://link.springer.com/book/10.1007/0-387-28695-0
[3] http://vita.had.co.nz/papers/layered-grammar.pdf
[4] Version 3.3.0 was released in March 2020
[5] Source: Hadley Wickham's paper on *the layered grammar of graphics*

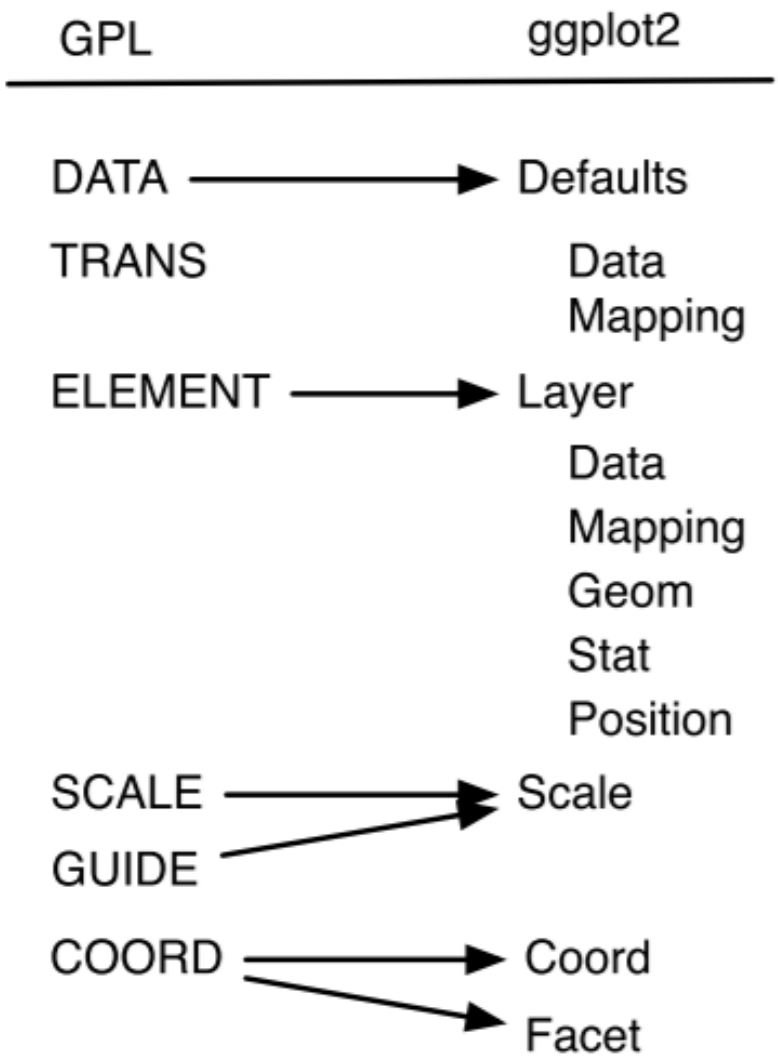


Figure 6.1: Layers in Grammar of Graphics mapped in GGPLOT2

1. **Data:** Unarguably, a graphic/visualisation should start with a data. It is also the first argument in most important function in the package i.e. `ggplot(data =)`.
2. **Aesthetics:** or `aes()` in short, provide a mapping of various data dimensions to axes so as to provide positions to various data points in the output plot/graphic.
3. **Geometries:** or `geoms` for short, are used to provide the *geometries* so that data points may take a concrete shape on the visualisation. For e.g. the data points should be depicted as bars or scatter points or else are decided by the provided `geoms.`
4. **Statistics:** or `stat` for short, provides the statistics to show in the visualisation like measures of central tendency, etc.
5. **Scale:** This component is used to decide whether any dimension needs some scaling like logrithmic transformation, etc.
6. **Coordinate System:** Though most of the time *cartesian coordinate system* is used, yet there are times when *polar coordinate system* (e.g. pie chart) or *spherical coordinate system* (e.g. geographical maps) are used.
7. **Facets:** Used when based on certain dimension, the plot is divided into further sub-plots.

6.2 Prerequisites:

```
library(ggplot2)
```

6.3 GGPLOT2 in action

Out of the afore-mentioned components, three are to be explicitly provided and thus can be understood as mandatoty components. These three componenets are `data`, `aesthetics` and `geometries`. Whilst these three compoenents are mandatorily provided, it is not that others are not mandatory. basically other componenets have their defaults (e.g. default coordinate system is cartesian coordinate system). Let us dive into these three essential components and build a plot using these.

6.3.1 Building a basic plot

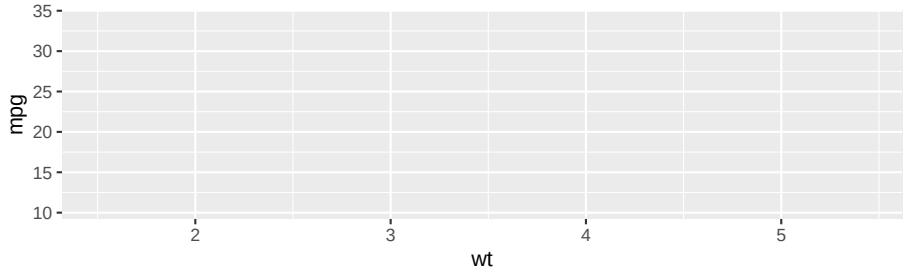
We will use `mtcars` datasets, a default dataset to learn the concepts.

See what happens when `data` is provided to `ggplot` function-

```
ggplot(data=mtcars)
```

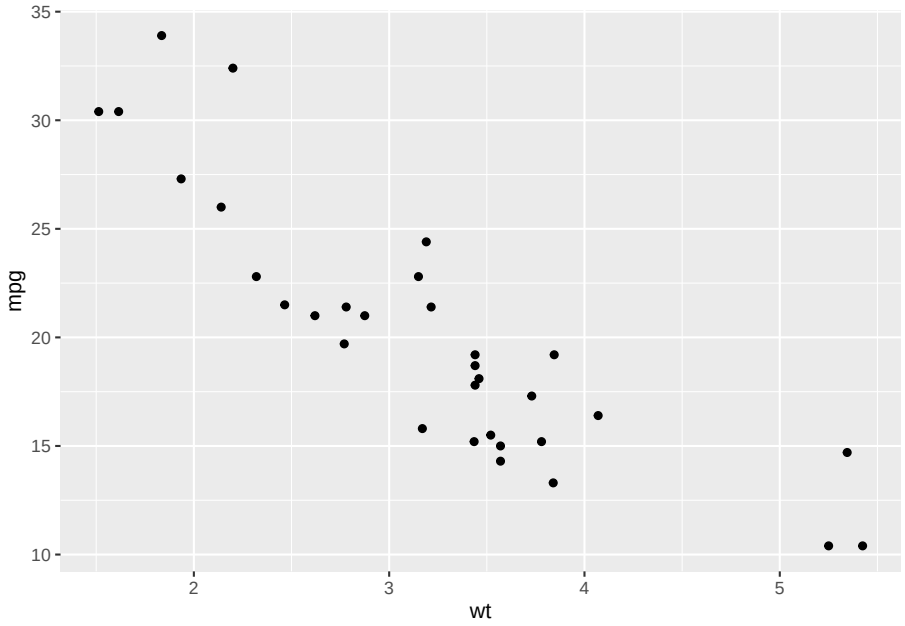

We can see that a blank chart/plot space has been created as our data `mtcars` has now mapped with ggplot2. Now let us provide aesthetic mappings to this using function `aes()`

```
ggplot(data = mtcars, mapping = aes(x=wt, y=mpg))
```



You may now notice, apart from creating a blank space for plot, the two dimensions provided, i.e. `wt` and `mpg` have been *mapped* with `x` and `y` axes respectively. Since no geometry has been provided, the plot area is still blank. Now we will provide geometry to our dimension say *point*. To do this we will use another layer of function `geom_*` (`geom_point()` in this case specifically).

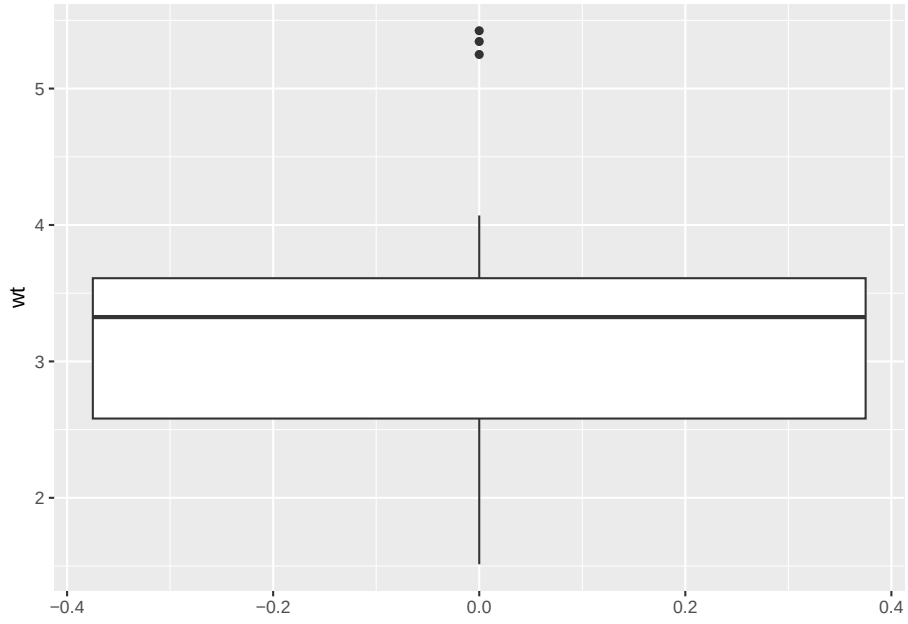
```
ggplot(mtcars, aes(wt, mpg)) +
  geom_point()
```



Notice that another layer has been added to function `ggplot()` using a `+` sign here.

We could have used another geometry say boxplot here.

```
ggplot(mtcars, aes(y = wt)) +
  geom_boxplot()
```

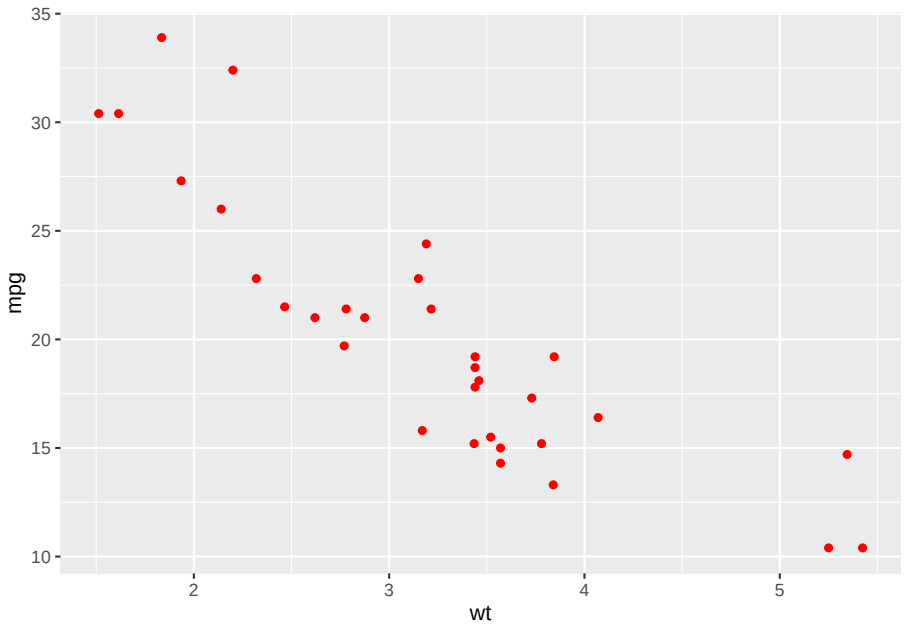


That's basic architecture of this package. Now lets discuss more on `aesthetics` and `geometries` before moving on to another compoenents.

6.3.2 More on Aesthetics

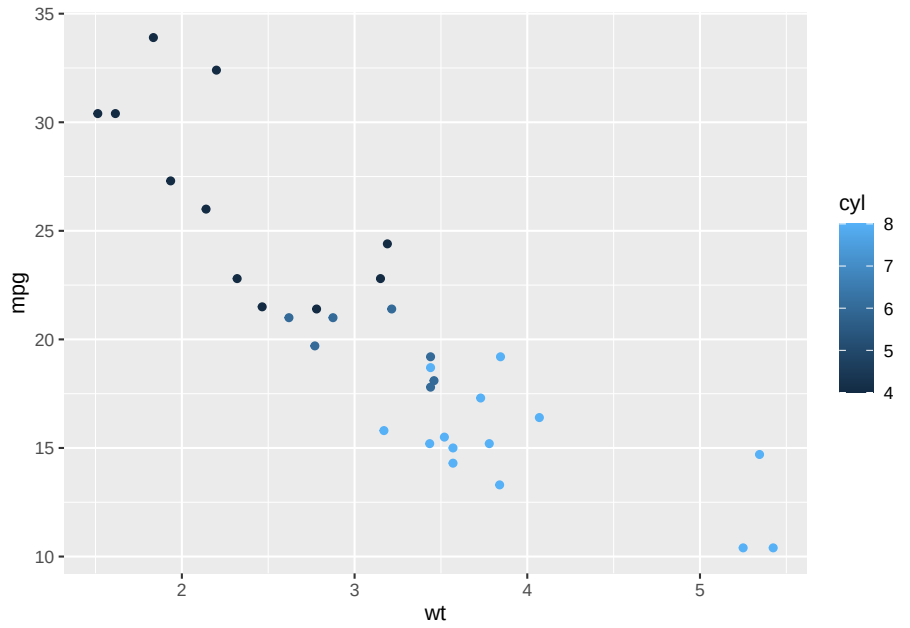
Now what if color is provided inside `geom_*` function.

```
ggplot(mtcars, aes(wt, mpg)) +
  geom_point(color='red')
```



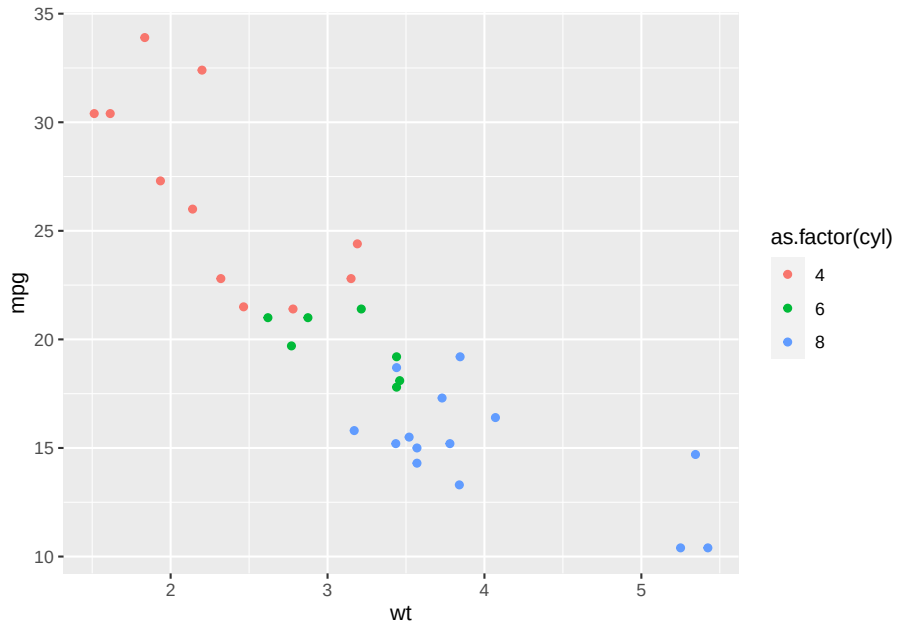
As the argument `color='red'` was mentioned inside the `geom_point()` function, it turned every point to red. But if we have to pass a vector/column based on which the points should be colored, it should be wrapped within aesthetics function `aes()` -

```
ggplot(mtcars, aes(wt, mpg)) +
  geom_point(aes(color=cyl))
```



Since the `cyl` column was a numeric column, ggplot2 thought it to be a `continuos` column and thus produced a color scale instead of a legend. We however, know that this is a categorical column here, and thus if we want to produce a color legend we will have to convert it to a factor first. See now the changes-

```
ggplot(mtcars, aes(wt, mpg)) +
  geom_point(aes(color=as.factor(cyl)))
```



One more thing - `aes()` function wrapped in `geom_point()` function could have been wrapped in `ggplot()` also. So basically the following code will also produce exactly the same chart-

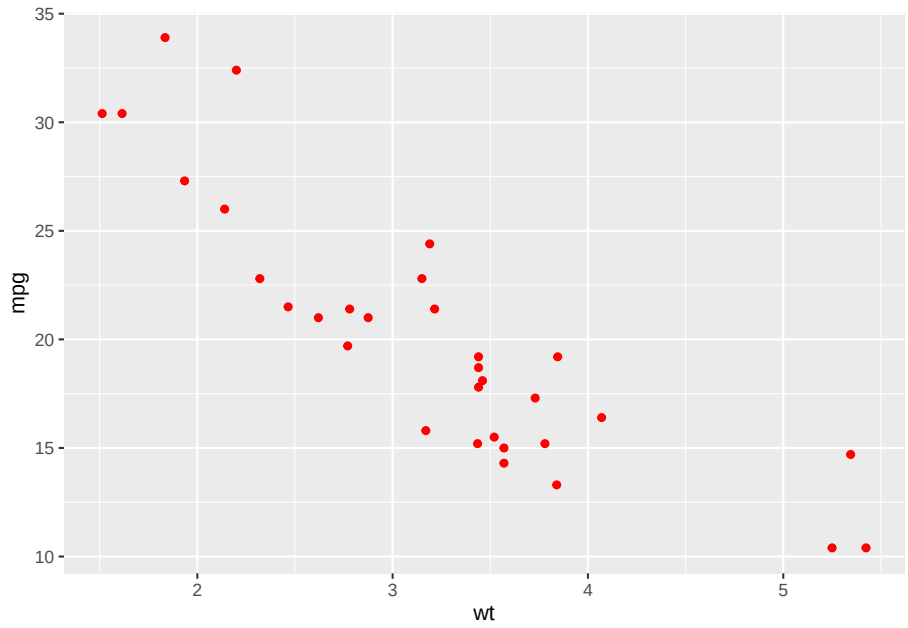
```
ggplot(mtcars, aes(wt, mpg, color = as.factor(cyl))) +
  geom_point()
```

Two questions arise here -

1. Is there any difference between the two?

Ans: Yes, basically aesthetics if provided under the `geoms`, will override those aesthetics which are already provided under `ggplot` function. See the result of following command in your console-

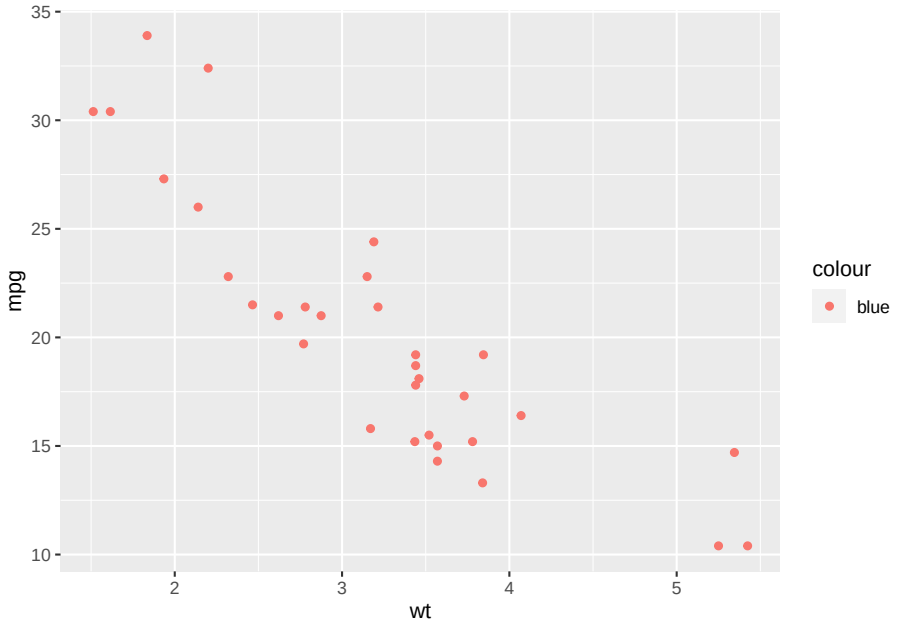
```
ggplot(mtcars, aes(wt, mpg, color = as.factor(cyl))) +
  geom_point(color='red')
```



2. What if `color='red'` (or blue) is passed inside `aes()`?

Ans: In this case ggplot will try to map it some aesthetics called 'blue'. Let's see

```
ggplot(mtcars, aes(wt, mpg)) +
  geom_point(aes(color='blue'))
```



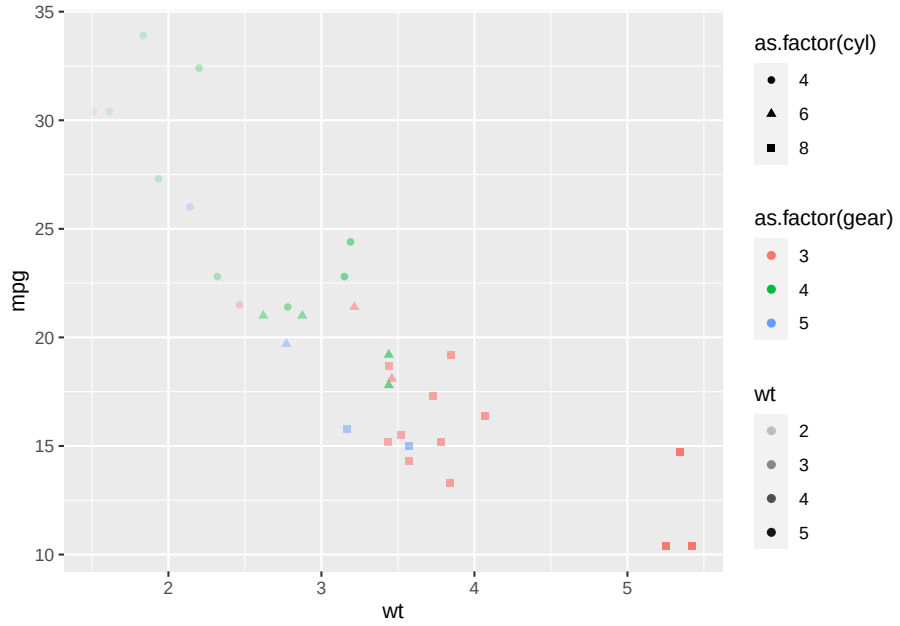
Interesting! GGPLOT2 has not only mapped a dummy variable called `'blue'` with color of points, but also created a legend. More interestingly the color is not what we wanted.

Different types of aesthetic attributes work better with different types of variables. For example, `color` and `shape` work well with discreet variables, while `size` or `alpha` (transparency) works well for continuous variables. In your console run the following command and check results

```
ggplot(mtcars, aes(wt, mpg, shape=as.factor(cyl))) +
  geom_point()
# OR
ggplot(mtcars, aes(wt, mpg, size=cyl)) +
  geom_point()
# OR
ggplot(mtcars, aes(wt, mpg, alpha=cyl)) +
  geom_point()
```

Multiple aesthetics can be mapped simultaneously, as per requirement. See this example-

```
ggplot(mtcars, aes(wt, mpg, shape=as.factor(cyl), color=as.factor(gear), alpha=wt)) +
  geom_point()
```



6.3.3 More on Geoms

In previous section we have seen that as soon as we passed a `geom_*` function/layer to `data` & `aesthetics` layers, the chart/graph was constructed. Actually, `geom_point()` function, in the background added three more layers i.e. `stat`, `geom` and `position`. Why? The answer is simple, `geom_*` are generally shortcuts, which add these three layers. So in our example, `ggplot(mtcars, aes(wt, mpg)) + geom_point()` is actually equivalent to -

```
ggplot(mpg, aes(displ, hwy)) +
  layer(
  mapping = NULL,
  data = NULL,
  geom = "point",
  stat = "identity",
  position = "identity"
)
```

A complete list of geoms available in ggplot2 is given in Annex-

6.3.4 Faceting

6.3.5 More to read

Book (Wickham, 2016)

The amount of data also makes a difference: if there is a lot of data it can be hard to distinguish different groups. An alternative solution is to use faceting, as described next.

Note that in ggplot2 `color` aesthetic represent border color of geometry and `fill` aesthetic represent color used to be fill that geometry.

6.3.6 Shapes

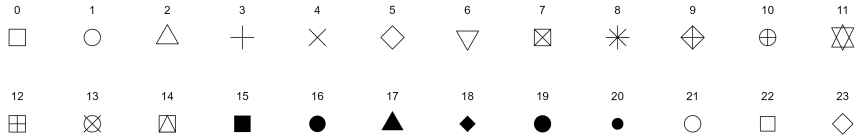


Figure 6.2: Shapes available in GGplot

Part-II: Dealing with tabular data

Chapter 7

Getting data in and out of R

Till now, we have created our own datasets or we have used sample datasets available in R. In practical usage, there will hardly be any case when we get the data imported in R. So we have to import and load our data sets in R, for our analytics tasks. Even after completing the analytics, the need for summarised data, other reports, charts, etc. need to be exported. This chapter is intended to do all these tasks.

First section deals about functions related to reading external data i.e. importing objects into R. This is followed by another section dealing with writing data to external files i.e. exporting data out of R.

7.1 Importing external data in R - Base R methods

Base R has many functions which can fulfill nearly any of our jobs to import external data into R. However, there are packages which are customised to do certain tasks in easier way and thus, we will also learn two of the packages from `tidyverse` also.

There are some important functions in base R, which are used frequently to import external data into R environment. Let us discuss these one by one.

Reading tables through `read.table()` and/or `read.csv()`

Basically these two functions are most commonly used functions in R, to get tabular data out of flat files. The two functions namely `read.table()` and

`read.csv()` are used respectively to read tabular data out of flat files having simple text format (.txt) and comma separated values (.csv) formats respectively. There are three more cousins to these functions.

- `read.csv2()` to read csv files where `;` is used as delimiter and `,` is used for decimals instead.
- `read.delim()` to read delimited files where `tab character` has been used as delimiter and `.` as decimal.
- `read.delim2()`, similarly to read delimited files where `tab character` has been used as delimiter and `,` as decimal.

Most important arguments to these functions are -

- `file` the name of the file along with complete path as string[1].
- `header` a logical value indicating if first line of the file has to be read as header or not.
- `sep` a string indicating a separator value to separate columns.
- `colClasses` a character vector indicating type of the columns if these are to be read explicitly in these types/formats only.
- `skip` an integer, indicating how many rows (from beginning) are to be skipped.

Readers may check results of `?read.table()` to get a complete list of arguments of these functions.

Example: Let us try to download data related to *World Happiness Report 201=21* which is available on data.world portal.

```
wh2021 <- read.csv("https://query.data.world/s/qbsbmxlfj54sl4mq3y6uxsr3pkhhmo")
# Check dimensions
dim(wh2021)
# Check column names
colnames(wh2021)
```

We can see that dataset named `wh2021` having `20` columns and `149` rows is now available in our environment.

> **Tip: Use `"clipboard"` in file argument for pasting copied data into R. E.g. Copy a few rows and cells from excel spreadsheet and run this command `read.delim("clipboard", header = FALSE)`.**

[1] If backward slash `\` is used in file paths, these must be `escaped` as R recognises `\` as escape character itself. So, `"my/location/here/file.txt"` and `"my\\file\\here\\file.txt"` are the correct way of giving file name

Read data into a vector through `scan()` or `readline()`

As afore-mentioned functions `read.table()` *et al* are used in reading data into tables, we may also require reading data into a vector or simple list. The two functions `scan()` and `readline()` are used for these purposes.

The basic syntax of `scan()` is -

```
scan(file = "",
     what = double(),
     nmax = -1,
     ...
)
```

Where -

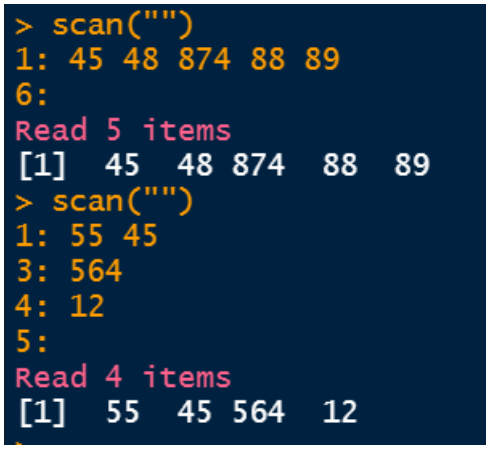
- `file` is name of the file, or link
- `what` is format/data type to be read. Default type is double
- `nmax` is mux number of data values to be read, default is `-1` which means all.

There are many other usefule arguments, for which please check results of `?scan` in your console.

Example -

```
scan("http://mattmahoney.net/iq/digits.txt", nmax = 10)
```

This function is sometimes useful to read data from keyboard into a vector. Just use a blank string `""` in file name. See this example



Function `readline()` on the other hand does similar job, but with a prompt. See this example

```
> user_name <- readline("Please input your name:")
Please input your name:Anil Goyal
> user_name
[1] "Anil Goyal"
> |
```

Reading text files through `readLines()`

Function `readLines()` is used to read text lines from a file (or connection). To see this in action, prepare a text file (say `"txt.txt"`) and try reading it using `readLines("txt.txt")`.

7.2 Exporting data out of R - Base R methods

Since the nature of most of the data anaytic jobs carried out in R, will be followed after reading the external data which will be followed by wrangling, transformation, modelling, etc., all in R. Exporting files will not be used as much as reading external data. Still, there will be times, when wrangled data tables need to be exported out of R. For each of the different use cases, the following functions will almost complete our export requirements.

Let's learn these.

Writing tabular data through `write.table()` and/or `write.csv()`

Exporting data frames, whether after cleaning, wrangling, or transformation, etc., can be exported using these functions. Latter will be used to write data frames in csv formats specifically. The syntax is -

```
write.table(x, file = "", sep = " ", ...)
write.csv(x, file = "", ...)
write.csv2(x, file = "", ...)
```

where -

- `x` is the data frame object to be exported
- `file` is used to give file name (along with path)

- `sep` is separator
- `...` - there are many more arguments which are used to cutomised export needs. See `?write.table()` for full details.

E.g. - The following command will export `iris` data frame as `iris.csv` file in the current working directory.

```
write.csv(iris, 'iris.csv')
```

Writing character data line by line to a file through `writeLines()`

Similar to `readLines`, function `writeLines()` will write the text data into a file with given file name. Type the following code in your console and check that a new file with name `my_new_file.txt` has been created with the given contents in your current working directory.

```
writeLines("Andrew 25
           Bob 45
           Charles 56", "my_new_file.txt")
```

Using `dput()` to get a code representation of R object

This function will output the code representation of the given R object. This function is particularly useful, when you are searching for help online and you need to give some sample data to reproduce the problem. E.g. on *Stack Overflow* when asking for a solution to a specific problem, reproducible data needs to be furnished. Please also refer to section 7.3.3 for more.

Example-1:

```
my_data <- data.frame(Name = c("Andrew", "Bob", "Charles"),
                      Age = c(25, 45, 56))
dput(my_data)
```

```
## structure(list(Name = c("Andrew", "Bob", "Charles"), Age = c(25,
## 45, 56)), class = "data.frame", row.names = c(NA, -3L))
```

And while reproducing someone else's dput-

```
now_mydata <- structure(list(Name = c("Andrew", "Bob", "Charles"), Age = c(25,
45, 56)), class = "data.frame", row.names = c(NA, -3L))

now_mydata
```

```
##      Name Age
## 1  Andrew  25
## 2     Bob  45
## 3 Charles  56
```

7.3 Using external packages for reading/writing data

7.3.1 Package `readr`

The `readr` package (Wickham et al., 2023c) is part of core `tidyverse` and is loaded directly when we load it through `library(tidyverse)`. It provides a range of analogous functions for each of the reading functions in base R.

Base R	readr	Uses
read.table	read_table	Reading table
read.csv	read_csv	Reading CSV file with comma as sep
read.csv2	read_csv2	Reading CSV file with semi-colon as sep
read.delim	read_delim	Reading files with any delimiter
read.fwf	read_fwf	Reading fixed width files
read.tsv	read_tsv	Reading tab delimited file
--	write_delim	Writing files with any delimiter
write.csv	write_csv	Writing files with comma delimiter
write.csv2	write_csv2	Writing files with semi-colon delimiter
--	write_tsv	writing a tab delimited file

So a question may be asked here that what's the difference between these two sets of functions. Firstly, `readr` alternatives are much faster than their base R counterparts. Secondly, it provides an informative problem report when parsing leads to unexpected results.

For these, check results of these examples-

```
read_csv(readr_example("chickens.csv"))
```

```
## Rows: 5 Columns: 4
## -- Column specification --------------------------------
## Delimiter: ","
## chr (3): chicken, sex, motto
## dbl (1): eggs_laid
##
## 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: 5 x 4
##   chicken                 sex     eggs_laid motto
##   <chr>                   <chr>       <dbl> <chr>
## 1 Foghorn Leghorn         rooster         0 That's a j~
## 2 Chicken Little          hen             3 The sky is~
## 3 Ginger                  hen            12 Listen. We~
## 4 Camilla the Chicken     hen             7 Bawk, buck~
## 5 Ernie The Giant Chicken rooster         0 Put Captai~
```

Note that the column types, while parsing the data frame, have now been printed. These column types have been guessed by `readr` actually. If the column types are not what were actually required to be parsed, then argument `col_types` may be used. We can also use `spec()` function to retrieve the data types guessed by `readr` later-on, so these can be modified and used again in `col_types` argument. See this example

```
write.csv(iris, "iris.csv") #write a dummy data
spec(read_csv("iris.csv"))
```

```
## New names:
## Rows: 150 Columns: 6
## -- Column specification
## ------------------------------- Delimiter: "," chr
## (1): Species dbl (5): ...1, Sepal.Length, Sepal.Width,
## Petal.Len...
## 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.
## * `` -> `...1`
```

```
## cols(
##   ...1 = col_double(),
##   Sepal.Length = col_double(),
##   Sepal.Width = col_double(),
##   Petal.Length = col_double(),
##   Petal.Width = col_double(),
##   Species = col_character()
## )
```

```
read_csv("iris.csv",
         col_select = 2:6,
         col_types = cols(
           Sepal.Length = col_double(),
```

```
          Sepal.Width = col_double(),
          Petal.Length = col_double(),
          Petal.Width = col_double(),
          Species = col_factor(levels = c('setosa', 'versicolor', 'virginica'))
        )
) %>% head(2)
```

```
## New names:
## * `` -> `...1`
```

```
## # A tibble: 2 x 5
##   Sepal.Length Sepal.Width Petal.Length Petal.Width
##          <dbl>       <dbl>        <dbl>       <dbl>
## 1          5.1         3.5          1.4         0.2
## 2          4.9         3            1.4         0.2
## # i 1 more variable: Species <fct>
```

7.3.2 Package `readxl`

This package is also part of `tidyverse` but this one has to be loaded specifically using `library(readxl)`. As the name suggests, it has functions which are useful to read/write data to/from excel files. Excel files (having extension .xls or .xlsx) are slightly different in a way that these may contain several *sheets* of data at once. The function `read_excel()` has been designed for reading sheets from excel files. The syntax is

```
read_excel(path, sheet = NULL, range = NULL)
```

7.3.3 Package `reprex`

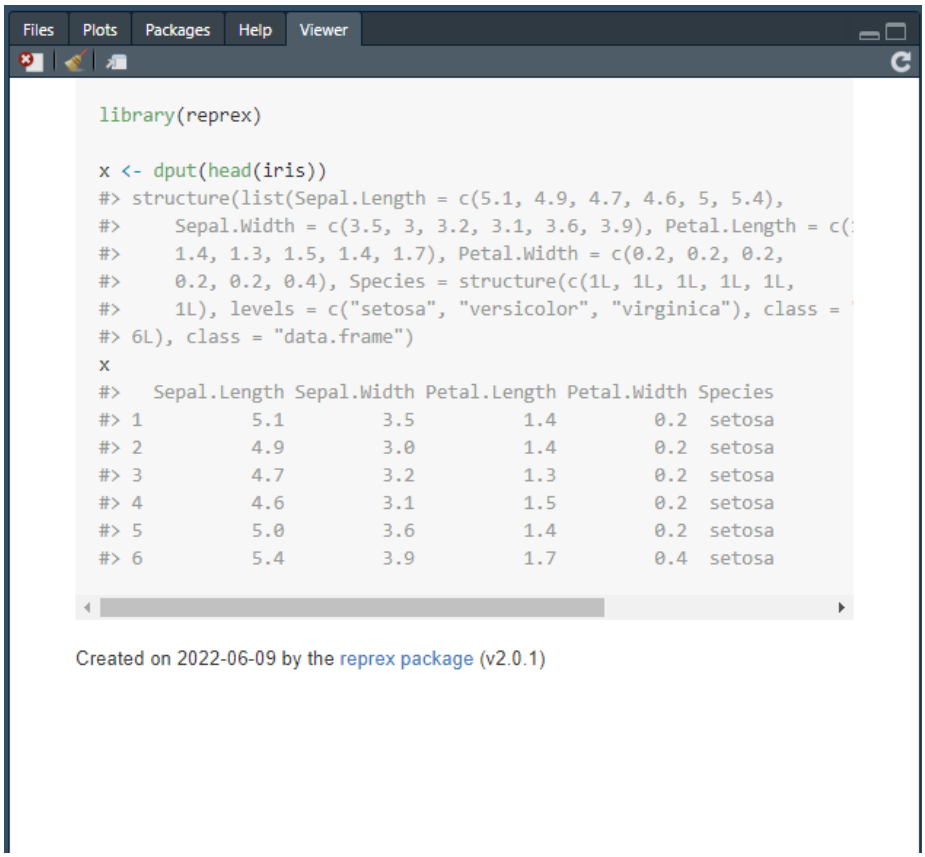
This is again part of tidyverse, but has to be loaded specifically by calling `library(reprex)`. The name `reprex` is actually short for reproducible example. This is useful particularly when we are stuck in some problem and seek for online help on some forum such as Stack Overflow, R Studio Community, etc.

As an example do this in your console

```
library(reprex)

x <- dput(head(iris))
x
```

Thereafter run `reprex()`, a small window in the `Viewer` tab will be opened like this. Moreover, the code has been copied on the clipboard.



For more reading please refer this page.[2]

[2]https://www.tidyverse.org/help/

Chapter 8

Data Transformation in `dplyr`

8.1 Prerequisites

Obviously `dplyr` (Wickham et al., 2023b) will be needed. This package also comes with matrittr pipe i.e. `%>%` and therefore in dplyr syntax we will be using these pipes. `library(magrittr)` is not needed.

```
library(dplyr)
library(knitr)
```

The package `dplyr` (8.1) calls its functions as 'verbs' because these are actually doing some action. So `dplyr verbs` can be divided in three classifications depending upon where they operate -

- 'Row' verbs that operate on Rows
- 'Column' verbs
- 'group' verbs that operate on table split into different groups.

Let's learn each of these -

8.2 Column verbs

8.2.1 `select()`

In real world data sets we will often come across with data frames having numerous columns. However for many of the data analysis tasks, most of these

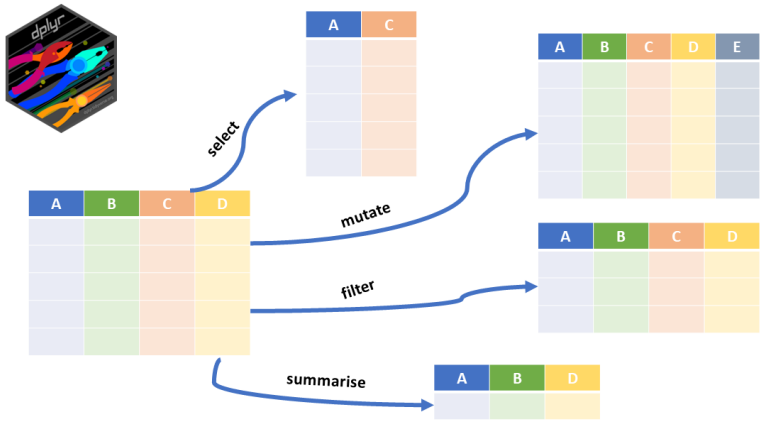


Figure 8.1: Package Dplyr

columns are not needed. As already stated `select` (figure 8.2) operates on columns. Like `SELECT` in `SQL`, it just *select* the column(s) from the given data frame. The basic syntax is - `select(data_frame, column_names, ...)`. So with pipes the same syntax goes like this

```
data_frame %>%
  select(column_name)
```

For example, let's try our hands on `mtcars` dataset.
Example-1

```
mtcars %>%
  select(mpg)
```

```
##                      mpg
## Mazda RX4           21.0
## Mazda RX4 Wag       21.0
## Datsun 710          22.8
## Hornet 4 Drive      21.4
## Hornet Sportabout   18.7
## Valiant             18.1
## Duster 360          14.3
## Merc 240D           24.4
## Merc 230            22.8
## Merc 280            19.2
```

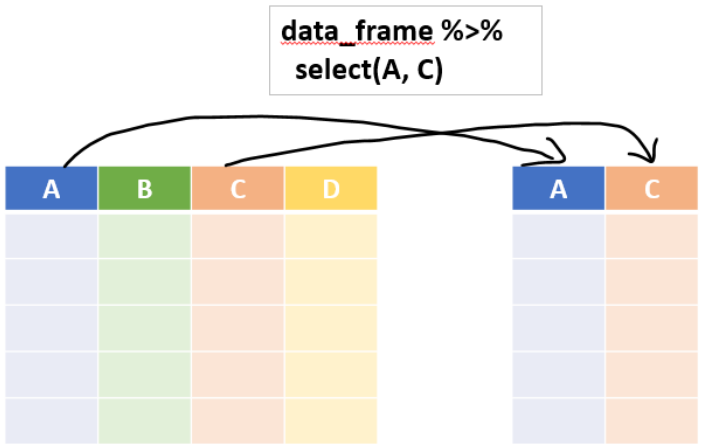


Figure 8.2: Illustration of dplyr::select()

```
## Merc 280C           17.8
## Merc 450SE          16.4
## Merc 450SL          17.3
## Merc 450SLC         15.2
## Cadillac Fleetwood  10.4
## Lincoln Continental 10.4
## Chrysler Imperial   14.7
## Fiat 128            32.4
## Honda Civic         30.4
## Toyota Corolla      33.9
## Toyota Corona       21.5
## Dodge Challenger    15.5
## AMC Javelin         15.2
## Camaro Z28          13.3
## Pontiac Firebird    19.2
## Fiat X1-9           27.3
## Porsche 914-2       26.0
## Lotus Europa        30.4
## Ford Pantera L      15.8
## Ferrari Dino        19.7
## Maserati Bora       15.0
## Volvo 142E          21.4
```

Note that output is still a data frame unlike the `mtcars[['mpg']]` which re-

turns a vector. We can subset multiple columns here. Example-2

```
mtcars %>%
  select(mpg, qsec) %>%
  head()
```

```
##                   mpg  qsec
## Mazda RX4         21.0 16.46
## Mazda RX4 Wag     21.0 17.02
## Datsun 710        22.8 18.61
## Hornet 4 Drive    21.4 19.44
## Hornet Sportabout 18.7 17.02
## Valiant           18.1 20.22
```

We can also provide column numbers instead of names. Example-3

```
mtcars %>%
  select(4, 6) %>%
  tail()
```

```
##                 hp    wt
## Porsche 914-2   91 2.140
## Lotus Europa   113 1.513
## Ford Pantera L 264 3.170
## Ferrari Dino   175 2.770
## Maserati Bora  335 3.570
## Volvo 142E     109 2.780
```

We can also use `select` to reorder the columns in output, by using a `dplyr` *helping verb* `everything()` which is basically *everything else*. See this example-

```
mtcars %>%
  select(qsec, mpg, everything()) %>%
  names()
```

```
##  [1] "qsec" "mpg"  "cyl"  "disp" "hp"   "drat" "wt"
##  [8] "vs"   "am"   "gear" "carb"
```

We may also use mix and match of *column names* and *column numbers*. See

```
mtcars %>%
  select(5, 7, mpg, everything()) %>%
  names()
```

```
##  [1] "drat" "qsec" "mpg"  "cyl"  "disp" "hp"   "wt"
##  [8] "vs"   "am"   "gear" "carb"
```

Operator `:` can also be used with column names. Ex-

```
mtcars %>%
  select(mpg:drat) %>%
  head(n=4)
```

```
##                  mpg cyl disp  hp drat
## Mazda RX4       21.0   6  160 110 3.90
## Mazda RX4 Wag   21.0   6  160 110 3.90
## Datsun 710      22.8   4  108  93 3.85
## Hornet 4 Drive 21.4   6  258 110 3.08
```

Other helping verbs There are other helping verbs, apart from `everything()` that can be used within `select()` just to eliminate need to type the column names and select columns based on some conditions. These verbs are self explanatory-

- `starts_with('ABC')` will select all columns the names of which **starts with** string `ABC`
- `ends_with('ABC')` will select all columns the names of which **ends with** string `ABC`
- `contains('ABC')` will select all columns the names of which **contains** string `ABC`
- `num_range('A', 1:3)` will select all columns named `A1`, `A2` and `A3`

Some Examples-

```
starwars %>%
  select(ends_with('color'))
```

```
## # A tibble: 87 x 3
##    hair_color    skin_color  eye_color
##    <chr>         <chr>       <chr>
##  1 blond         fair        blue
##  2 <NA>          gold        yellow
##  3 <NA>          white, blue red
##  4 none          white       yellow
##  5 brown         light       brown
##  6 brown, grey   light       blue
##  7 brown         light       blue
```

```
##  8 <NA>          white, red  red
##  9 black         light       brown
## 10 auburn, white fair        blue-gray
## # i 77 more rows
```

Example-2

```
starwars %>%
  select(contains('or'))
```

```
## # A tibble: 87 x 4
##    hair_color    skin_color  eye_color homeworld
##    <chr>         <chr>       <chr>     <chr>
##  1 blond         fair        blue      Tatooine
##  2 <NA>          gold        yellow    Tatooine
##  3 <NA>          white, blue red       Naboo
##  4 none          white       yellow    Tatooine
##  5 brown         light       brown     Alderaan
##  6 brown, grey   light       blue      Tatooine
##  7 brown         light       blue      Tatooine
##  8 <NA>          white, red  red       Tatooine
##  9 black         light       brown     Tatooine
## 10 auburn, white fair        blue-gray Stewjon
## # i 77 more rows
```

8.2.2 mutate()

This perhaps is one of the most important functions in `dplyr` kitty. It enables us to create new column(s) that are functions of one or more existing columns. Refer figure 8.3

More than one column can be added simultaneously. Newly created column may also be used for creation of another new column. See example.

```
starwars %>%
  select(name:mass) %>%
  mutate(name_upper = toupper(name),
         BMI = mass/(height/100)^2)
```

```
## # A tibble: 87 x 5
##    name               height  mass name_upper       BMI
##    <chr>               <int> <dbl> <chr>          <dbl>
##  1 Luke Skywalker        172    77 LUKE SKYWALKER  26.0
##  2 C-3PO                 167    75 C-3PO           26.9
```

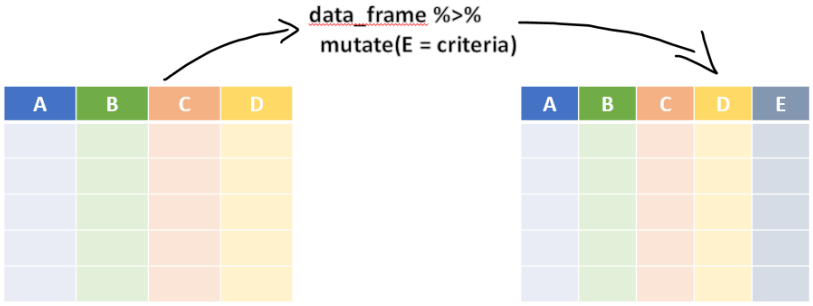


Figure 8.3: Illustration of dplyr::mutate()

```
##  3 R2-D2                 96    32 R2-D2           34.7
##  4 Darth Vader          202   136 DARTH VADER     33.3
##  5 Leia Organa          150    49 LEIA ORGANA     21.8
##  6 Owen Lars            178   120 OWEN LARS       37.9
##  7 Beru Whitesun lars   165    75 BERU WHITESUN~  27.5
##  8 R5-D4                 97    32 R5-D4           34.0
##  9 Biggs Darklighter    183    84 BIGGS DARKLIG~  25.1
## 10 Obi-Wan Kenobi       182    77 OBI-WAN KENOBI  23.2
## # i 77 more rows
```

By default the new column will be added to the last of data frame. As shown in above example, more operations can be combined in one using `%>%`. There is a cousin `transmute()` of *mutate* which drops all the old columns and keeps only newly created columns. Example

```
starwars %>%
  transmute(name_upper = toupper(name))
```

```
## # A tibble: 87 x 1
##    name_upper
##    <chr>
##  1 LUKE SKYWALKER
##  2 C-3PO
##  3 R2-D2
##  4 DARTH VADER
##  5 LEIA ORGANA
##  6 OWEN LARS
##  7 BERU WHITESUN LARS
```

```
##  8 R5-D4
##  9 BIGGS DARKLIGHTER
## 10 OBI-WAN KENOBI
## # i 77 more rows
```

Other useful dplyr functions Another good use of `mutate` is to generate summarised result and display it corresponding to each row in data. For example if the requirement is to calculate proportion of say `wt` column in `mtcars` data.

```
mtcars %>%
  head() %>%
  select(wt) %>%
  mutate(total_wt = sum(wt),
         wt_proportion = wt*100/total_wt)
```

```
##                    wt total_wt wt_proportion
## Mazda RX4         2.620    17.93         14.61
## Mazda RX4 Wag     2.875    17.93         16.03
## Datsun 710        2.320    17.93         12.94
## Hornet 4 Drive    3.215    17.93         17.93
## Hornet Sportabout 3.440    17.93         19.19
## Valiant           3.460    17.93         19.30
```

1. `n()` is used to count number of rows
2. `n_distinct()` is used to count number of distinct values for the given variable

```
mtcars %>%
  select(1:5) %>%
  mutate(total_cars = n()) %>%
  head()
```

```
##                   mpg cyl disp  hp drat total_cars
## Mazda RX4         21.0   6  160 110 3.90         32
## Mazda RX4 Wag     21.0   6  160 110 3.90         32
## Datsun 710        22.8   4  108  93 3.85         32
## Hornet 4 Drive    21.4   6  258 110 3.08         32
## Hornet Sportabout 18.7   8  360 175 3.15         32
## Valiant           18.1   6  225 105 2.76         32
```

8.2.3 `rename()`

It is used to *rename* the column names. Refer figure 8.4 for illustration.

See this example

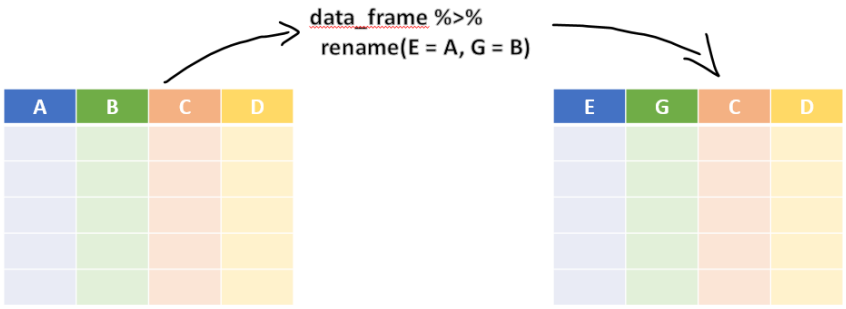


Figure 8.4: Illustration of dplyr::rename()

```
mtcars %>%
  rename(miles_per_gallon = mpg) %>%
  head(3)
```

```
##                miles_per_gallon cyl disp  hp drat    wt
## Mazda RX4                  21.0   6  160 110 3.90 2.620
## Mazda RX4 Wag              21.0   6  160 110 3.90 2.875
## Datsun 710                 22.8   4  108  93 3.85 2.320
##                 qsec vs am gear carb
## Mazda RX4      16.46  0  1    4    4
## Mazda RX4 Wag 17.02  0  1    4    4
## Datsun 710     18.61  1  1    4    1
```

Note that `select` can also rename the columns but will drop all unselected columns. Check this

```
mtcars %>%
  select(miles_per_gallon = mpg) %>%
  head(3)
```

```
##                miles_per_gallon
## Mazda RX4                  21.0
## Mazda RX4 Wag              21.0
## Datsun 710                 22.8
```

8.2.4 relocate()

It *relocates* column or block of columns simultaneosly either before the column mentioned in argument `.before` or after mentioned in `.after`. See the example-

```
starwars %>%
  relocate(ends_with('color'), .after = name) %>%
  head(5)
```

```
## # A tibble: 5 x 14
##   name     hair_color skin_color eye_color height  mass
##   <chr>    <chr>      <chr>      <chr>      <int> <dbl>
## 1 Luke Sk~ blond      fair       blue         172    77
## 2 C-3PO    <NA>       gold       yellow       167    75
## 3 R2-D2    <NA>       white, bl~ red           96    32
## 4 Darth V~ none       white      yellow       202   136
## 5 Leia Or~ brown      light      brown        150    49
## # i 8 more variables: birth_year <dbl>, sex <chr>,
## #   gender <chr>, homeworld <chr>, species <chr>,
## #   films <list>, vehicles <list>, starships <list>
```

8.3 Row verbs

8.3.1 filter

This verb/function is used to subset the data, or in other words filter rows of data frame based on certain condition. Refer figure 8.5 for illustration.

See this example-

```
starwars %>%
  filter(eye_color %in% c('red', 'yellow'))
```

```
## # A tibble: 16 x 14
##    name    height  mass hair_color skin_color eye_color
##    <chr>    <int> <dbl> <chr>      <chr>      <chr>
##  1 C-3PO      167    75 <NA>       gold       yellow
##  2 R2-D2       96    32 <NA>       white, bl~ red
##  3 Darth ~    202   136 none       white      yellow
##  4 R5-D4       97    32 <NA>       white, red red
##  5 Palpat~    170    75 grey       pale       yellow
##  6 IG-88      200   140 none       metal      red
##  7 Bossk      190   113 none       green      red
```

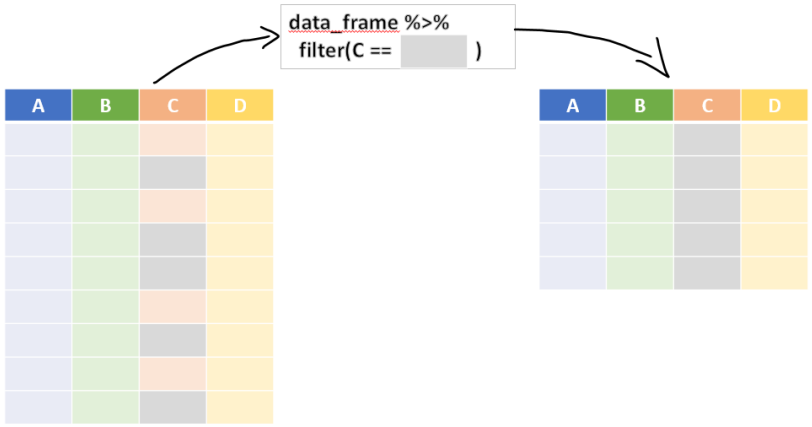


Figure 8.5: Illustration of dplyr::filter()

```
##  8 Nute G~    191    90 none       mottled g~ red
##  9 Watto      137    NA black      blue, grey yellow
## 10 Darth ~    175    80 none       red        yellow
## 11 Dud Bo~     94    45 none       blue, grey yellow
## 12 Ki-Adi~    198    82 white      pale       yellow
## 13 Yarael~    264    NA none       white      yellow
## 14 Poggle~    183    80 none       green      yellow
## 15 Zam We~    168    55 blonde     fair, gre~ yellow
## 16 Dexter~    198   102 none       brown      yellow
## # i 8 more variables: birth_year <dbl>, sex <chr>,
## #   gender <chr>, homeworld <chr>, species <chr>,
## #   films <list>, vehicles <list>, starships <list>
```

Multiple conditions can be passed simultaneously. Example

```
starwars %>%
  filter(skin_color == 'white',
         height >= 150)
```

```
## # A tibble: 2 x 14
##   name     height  mass hair_color skin_color eye_color
##   <chr>     <int> <dbl> <chr>      <chr>      <chr>
## 1 Darth V~    202   136 none       white      yellow
## 2 Yarael ~    264    NA none       white      yellow
## # i 8 more variables: birth_year <dbl>, sex <chr>,
```

```
## #   gender <chr>, homeworld <chr>, species <chr>,
## #   films <list>, vehicles <list>, starships <list>
```

Note that these conditions act simultaneously as in operator `AND` is used. So if `OR` is to be used, use `|` explicitly

```
starwars %>%
  filter(skin_color == 'white' | height >= 150) %>%
  nrow()
```

```
## [1] 71
```

8.3.2 `slice()` / `slice_*()`

`slice()` and its cousins also filters rows but based on rows placement. So, `data_fr %>% slice(1:5)` will filter out first five rows of the `data_fr`. See example

```
starwars %>%
  slice(4:10) # filter 4 to 10th row
```

```
## # A tibble: 7 x 14
##   name     height  mass hair_color skin_color eye_color
##   <chr>     <int> <dbl> <chr>      <chr>      <chr>
## 1 Darth V~    202   136 none       white      yellow
## 2 Leia Or~    150    49 brown      light      brown
## 3 Owen La~    178   120 brown, gr~ light      blue
## 4 Beru Wh~    165    75 brown      light      blue
## 5 R5-D4        97    32 <NA>       white, red red
## 6 Biggs D~    183    84 black      light      brown
## 7 Obi-Wan~    182    77 auburn, w~ fair       blue-gray
## # i 8 more variables: birth_year <dbl>, sex <chr>,
## #   gender <chr>, homeworld <chr>, species <chr>,
## #   films <list>, vehicles <list>, starships <list>
```

Other `slice()` cousins -

- `slice_head(5)` will slice out first 5 rows
- `slice_tail(10)` will slice out last 10 rows
- `slice_min()` or `slice_max()` will slice rows with highest or lowest values of given variable. The full syntax is `slice_max(.data, order_by, ..., n, prop, with_ties = TRUE)` or equivalent

- `slice_sample()` will randomly select the rows. Its syntax is `slice_sample(.data, ..., n, prop, weight_by = NULL, replace = FALSE)`

Example-1:

```
starwars %>%
  slice_min(height, n=3)
```

```
## # A tibble: 3 x 14
##   name     height  mass hair_color skin_color eye_color
##   <chr>     <int> <dbl> <chr>      <chr>      <chr>
## 1 Yoda         66    17 white      green      brown
## 2 Ratts T~     79    15 none       grey, blue unknown
## 3 Wicket ~     88    20 brown      brown      brown
## # i 8 more variables: birth_year <dbl>, sex <chr>,
## #   gender <chr>, homeworld <chr>, species <chr>,
## #   films <list>, vehicles <list>, starships <list>
```

Example-2:

```
set.seed(2022)
starwars %>%
  slice_sample(prop = 0.1) #sample 10% rows
```

```
## # A tibble: 8 x 14
##   name     height  mass hair_color skin_color eye_color
##   <chr>     <int> <dbl> <chr>      <chr>      <chr>
## 1 Eeth Ko~    171    NA black      brown      brown
## 2 Tarfful     234   136 brown      brown      blue
## 3 Plo Koon    188    80 none       orange     black
## 4 San Hill    191    NA none       grey       gold
## 5 Dexter ~    198   102 none       brown      yellow
## 6 Owen La~    178   120 brown, gr~ light      blue
## 7 Dormé       165    NA brown      light      brown
## 8 Captain~     NA    NA unknown    unknown    unknown
## # i 8 more variables: birth_year <dbl>, sex <chr>,
## #   gender <chr>, homeworld <chr>, species <chr>,
## #   films <list>, vehicles <list>, starships <list>
```

8.3.3 `arrange()`

This verb also act upon rows and it actually *rearranges* them on the basis of some condition. Refer figure 8.6 for illustration.

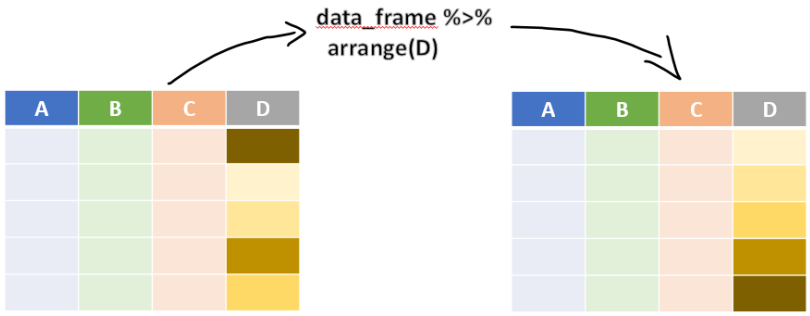


Figure 8.6: Illustration of dplyr::arrange()

Example-

```
starwars %>%
  arrange(height) %>%
  slice(1:5)
```

```
## # A tibble: 5 x 14
##   name     height  mass hair_color skin_color eye_color
##   <chr>     <int> <dbl> <chr>      <chr>      <chr>
## 1 Yoda         66    17 white      green      brown
## 2 Ratts T~     79    15 none       grey, blue unknown
## 3 Wicket ~     88    20 brown      brown      brown
## 4 Dud Bolt     94    45 none       blue, grey yellow
## 5 R2-D2        96    32 <NA>       white, bl~ red
## # i 8 more variables: birth_year <dbl>, sex <chr>,
## #   gender <chr>, homeworld <chr>, species <chr>,
## #   films <list>, vehicles <list>, starships <list>
```

8.4 Group verbs

8.4.1 `group_by()`

A data analyst will be hard to find who is not using `group_by`. It basically groups the rows on the basis of values of a given variable or block of variables. The returned result is still a data frame (and one too) but now the rows are grouped. Refer figure 8.7 for illustration. So any of the above functions we learnt above will give a different result after group by.

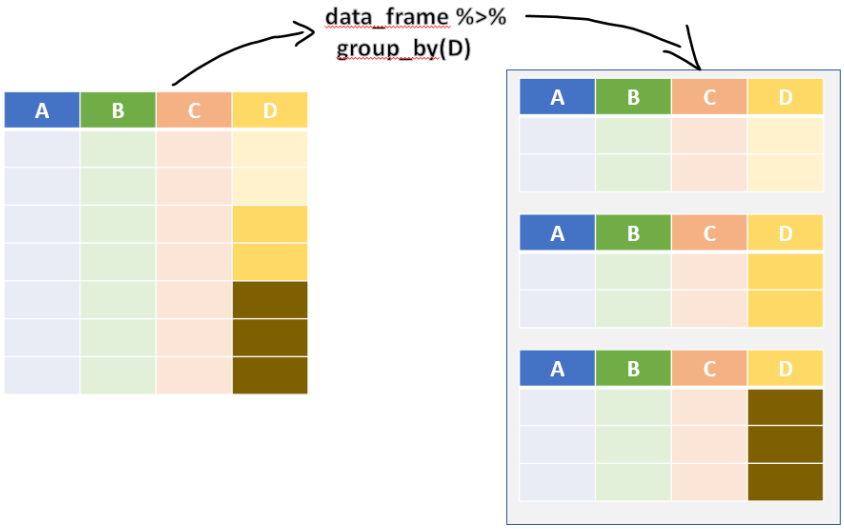


Figure 8.7: Illustration of Grouped Operations in dplyr

Note the output of this simple example

```
starwars %>%
  group_by(sex)
```

```
## # A tibble: 87 x 14
## # Groups:   sex [5]
##    name    height  mass hair_color skin_color eye_color
##    <chr>    <int> <dbl> <chr>      <chr>      <chr>
##  1 Luke S~    172    77 blond      fair       blue
##  2 C-3PO      167    75 <NA>       gold       yellow
##  3 R2-D2       96    32 <NA>       white, bl~ red
##  4 Darth ~    202   136 none       white      yellow
##  5 Leia O~    150    49 brown      light      brown
##  6 Owen L~    178   120 brown, gr~ light      blue
##  7 Beru W~    165    75 brown      light      blue
##  8 R5-D4       97    32 <NA>       white, red red
##  9 Biggs ~    183    84 black      light      brown
## 10 Obi-Wa~    182    77 auburn, w~ fair       blue-gray
## # i 77 more rows
## # i 8 more variables: birth_year <dbl>, sex <chr>,
## #   gender <chr>, homeworld <chr>, species <chr>,
## #   films <list>, vehicles <list>, starships <list>
```

Note that output now has 5 groups, though nothing different is seen in the displayed data.

This operation/verb is thus more useful if used in combination with other verbs.

Example-1: How many total characters are with same skin_color?

```
starwars %>%
  select(name, skin_color) %>%
  group_by(skin_color) %>%
  mutate(total_with_s_c = n())
```

```
## # A tibble: 87 x 3
## # Groups:   skin_color [31]
##    name               skin_color  total_with_s_c
##    <chr>              <chr>                <int>
##  1 Luke Skywalker     fair                    17
##  2 C-3PO              gold                     1
##  3 R2-D2              white, blue              2
##  4 Darth Vader        white                    2
##  5 Leia Organa        light                   11
##  6 Owen Lars          light                   11
##  7 Beru Whitesun lars light                   11
##  8 R5-D4              white, red               1
##  9 Biggs Darklighter  light                   11
## 10 Obi-Wan Kenobi     fair                    17
## # i 77 more rows
```

Example- 2: Sample 2 rows of each `cyl` size from `mtcars`?

```
set.seed(123)
mtcars %>%
  group_by(cyl) %>%
  slice_sample(n=2)
```

```
## # A tibble: 6 x 11
## # Groups:   cyl [3]
##     mpg   cyl  disp    hp  drat    wt  qsec    vs    am
##   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1  22.8     4  141.    95  3.92  3.15  22.9     1     0
## 2  21.4     4  121    109  4.11  2.78  18.6     1     1
## 3  21       6  160    110  3.9   2.88  17.0     0     1
## 4  19.7     6  145    175  3.62  2.77  15.5     0     1
## 5  10.4     8  472    205  2.93  5.25  18.0     0     0
## 6  13.3     8  350    245  3.73  3.84  15.4     0     0
## # i 2 more variables: gear <dbl>, carb <dbl>
```

Also note that `grouped` varaible(s) will always be available in the output.

```
mtcars %>%
  group_by(cyl) %>%
  select(drat) %>% # despite not selecting cyl
  head() # it is available in output
```

```
## Adding missing grouping variables: `cyl`
```

```
## # A tibble: 6 x 2
## # Groups:   cyl [3]
##     cyl  drat
##   <dbl> <dbl>
## 1     6  3.9
## 2     6  3.9
## 3     4  3.85
## 4     6  3.08
## 5     8  3.15
## 6     6  2.76
```

8.4.2 summarise()

This verb creates a summary row for each group if grouped data frame is in input, otherwise one single for complete operation.

Example-1:

```
mtcars %>%
  summarise(total_wt = sum(wt))
```

```
##   total_wt
## 1      103
```

Example-2:

```
mtcars %>%
  group_by(cyl) %>%
  summarise(total_wt = sum(wt))
```

```
## # A tibble: 3 x 2
##     cyl total_wt
##   <dbl>    <dbl>
## 1     4     25.1
## 2     6     21.8
## 3     8     56.0
```

8.5 Other Useful functions in dplyr

if_else()

This function operates nearly as similar to base R's `ifelse()` with two exceptions-

- There is an extra argument to provide values when missing values are encountered. (See example-1)
- `NA` will have to be provided specifically. (See Example-2)

See these examples. Example-1:

```
x <- c(-2:2, NA)
if_else(x>=0, "positive", "negative", "missing")
```

```
## [1] "negative" "negative" "positive" "positive"
## [5] "positive" "missing"
```

Example-2:

```
x <- c(-2:2, NA)
if_else(x>=0, "positive", "negative", NA_character_)
```

```
## [1] "negative" "negative" "positive" "positive"
## [5] "positive" NA
```

Due to the additional restrictions, this function is sometimes faster than its base R alternative and may also be useful in prevention of bugs in code as the output will be known beforehand.

case_when()

Though both `ifelse` and `if_else` variants provide for nesting multiple conditions, yet `case_when` provides a simpler alternative in these conditions as here multiple conditions can be provided simultaneously. Syntax follows this style-

```
case_when(
  condition1 ~ value_if_true,
  condition2 ~ value_if_true,
  ...,
  TRUE ~ value_if_all_above_are_false
)
```

See this example.

```
set.seed(123)
income <- runif(7, 1, 9)*100000
income
```

```
## [1] 330062 730644 427182 806414 852374 136445 522484
```

```
# tax brackets say 0% upto 2 lakh, then 10% upto 5 Lakh
# then 20% upto 7.5 lakh otherwise 30%
tax_slab <- case_when(
  income <= 200000 ~ 0,
  income <= 500000 ~ 10,
  income <= 750000 ~ 20,
  TRUE ~ 30
)

# check tax_slab
data.frame(
  income=income,
  tax_slab = tax_slab
)
```

```
##   income tax_slab
## 1 330062       10
## 2 730644       20
## 3 427182       10
## 4 806414       30
## 5 852374       30
## 6 136445        0
## 7 522484       20
```

8.6 Window functions/operations

We learnt that by using `group_by` function we can create windows in data and we can make our calculations in each separate window specifically.

Dplyr provides us with some useful window functions which will operate on these windows.

1. `row_number()` can be used to generate row number
2. `dense_rank` / `min_rank` / `percent_rank()` / `ntile()` / `cume_dist()` are other windowed functions in dplyr. Check `?dplyr::ranking` for complete reference.

3. `lead()` and `lag()` will give leading/lagging value in that window.

These functions can be very helpful while analysing time series data.

Example-1:

```
# example data
df <- data.frame(
  val = c(10, 2, 3, 2, NA)
)

df %>%
  mutate(
    row = row_number(),
    min_rank = min_rank(val),
    dense_rank = dense_rank(val),
    perc_rank = percent_rank(val),
    cume_dist = cume_dist(val)
  )
```

```
##   val row min_rank dense_rank perc_rank cume_dist
## 1  10   1        4          3    1.0000      1.00
## 2   2   2        1          1    0.0000      0.50
## 3   3   3        3          2    0.6667      0.75
## 4   2   4        1          1    0.0000      0.50
## 5  NA   5       NA         NA        NA        NA
```

Example-2:

```
Orange %>%
  group_by(Tree) %>%
  mutate(prev_circ = lag(circumference))
```

```
## # A tibble: 35 x 4
## # Groups:   Tree [5]
##    Tree    age circumference prev_circ
##    <ord> <dbl>         <dbl>     <dbl>
##  1 1       118            30        NA
##  2 1       484            58        30
##  3 1       664            87        58
##  4 1      1004           115        87
##  5 1      1231           120       115
##  6 1      1372           142       120
##  7 1      1582           145       142
##  8 2       118            33        NA
```

```
##  9 2       484            69        33
## 10 2       664           111        69
## # i 25 more rows
```

Chapter 9

Combining Tables/tabular data

Figure 9.1: Most of times, joining two or more tables will be required to perform analytics

In real world scenarios, there may hardly be a case when we have to analyse one single table. There may be cases when we have to either join tables split into multiple smaller tables (e.g. we can have smaller tables split States-wise), or the tables may be divided into various smaller master and transaction tables (relational databases).

We may thus divide the data tables joining requirements into three broad categories-

- Simple joins or concatenation
- Relational Joins
- Filtering Joins

Let us discuss each of these with examples.

9.1 Simple joins/concatenation

Many times tables split into smaller tables have to be joined back before proceeding further for data analytics. We may have to join two or more tables either columnwise (e.g. some of the features for all rows have been split into a separate table) or row wise (e.g. all the fields/columns are split into smaller tables like a separate table for each State). Diagramatically these joins may be depicted as shown in figure 9.2.

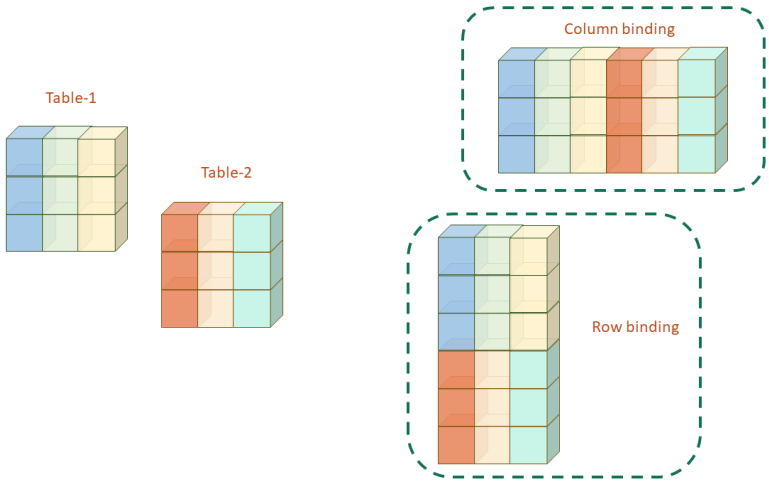


Figure 9.2: Illustration of Simple joins/concatenation

9.1.1 Column binding

As we have already seen that data frames act like matrices in many ways except to that fact that these support heterogeneous data unlike matrices. We have also discussed the ways two matrices can be joined. Base R has two dedicated functions i.e. `cbind()` and `rbind()` for these operations.

In *tidyverse* (dplyr specifically) we have two similar functions `bind_cols()` and `bind_rows` respectively which provide us better functionality for these use cases.

The syntax for finction `bind_cols()` used for concatenating two or more tables *column wise* is -

```
bind_cols(
  ...,
  .name_repair = c("unique", "universal", "check_unique")
)
```

Where -

- `...` represent data frames to be combined
- `.name_repair` argument chooses method to rename duplicate column names, if any.

Example-1:

```
df1 <- iris[1:3, c(1,2,5)]
df2 <- iris[1:3, 3:5]

bind_cols(df1, df2, .name_repair = 'universal')
```

```
## New names:
## * `Species` -> `Species...3`
## * `Species` -> `Species...6`
```

```
##   Sepal.Length Sepal.Width Species...3 Petal.Length
## 1          5.1         3.5      setosa          1.4
## 2          4.9         3.0      setosa          1.4
## 3          4.7         3.2      setosa          1.3
##   Petal.Width Species...6
## 1         0.2      setosa
## 2         0.2      setosa
## 3         0.2      setosa
```

> Note: The data frames to be merged should be row-consistent for column binding. Try this `bind_cols(iris[1:3, 1:2], iris[1:4, 3:4])` and see the results.

9.1.2 Row binding

The syntax used for appending rows of one or more tables together in one table is -

```
bind_rows(..., .id = NULL)
```

where -

- `...` represent data frames to be combined
- `.id` argument creates a new column of identifiers.

To understand it better, let us see this example

```
setosa <- iris[1:3, 1:4]
versicolor <- iris[51:53, 1:4]
virginica <- iris[101:103, 1:4]

bind_rows(setosa, versicolor, virginica, .id = 'groups')
```

```
##   groups Sepal.Length Sepal.Width Petal.Length
## 1      1          5.1         3.5          1.4
## 2      1          4.9         3.0          1.4
## 3      1          4.7         3.2          1.3
## 4      2          7.0         3.2          4.7
## 5      2          6.4         3.2          4.5
## 6      2          6.9         3.1          4.9
## 7      3          6.3         3.3          6.0
## 8      3          5.8         2.7          5.1
## 9      3          7.1         3.0          5.9
##   Petal.Width
## 1         0.2
## 2         0.2
## 3         0.2
## 4         1.4
## 5         1.5
## 6         1.5
## 7         2.5
## 8         1.9
## 9         2.1
```

> Note: In the above example if the requirement is to store data tables names into the new *identifier* column just list convert the databases into a list and `.id` will take element_names as the values in identifiers. Try this `bind_rows(list(setosa=setosa, versicolor=versicolor, virginica=virginica), .id = 'Species')`

9.2 Relational joins

Relational Joins are usually needed to join multiple tables based on *primary key* and *secondary keys*. The joins may either be *one-to-one* key join or *one-to-many* key joins. Broadly these can either be *inner joins* or *outer joins*. Diagrammatically these may be represented as shown in figure 9.3.

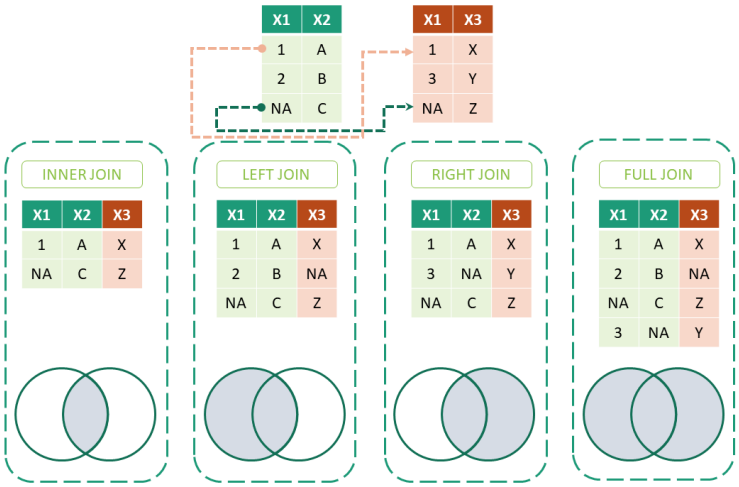


Figure 9.3: Illustration of Mutating Joins in dplyr

The syntax of all these joins is nearly same-

```
*_join(x, y, by = NULL, copy = FALSE, suffix = c('.x', '.y'), ... , keep = FALSE)
```

where -

- `x` and `y` are data frames to be joined
- `by` is a character vector of column names to be joined by
- `suffix` argument provides suffixes to be added to column names, if any of those are duplicate
- `keep` argument decides whether the join keys from both x and y be preserved in the output?

Let's discuss each of these joins individually.

9.2.1 Inner Joins

Inner Joins keeps only those rows where matching keys are present in both the data frames.

Example-1:

```
band_members
```

```
## # A tibble: 3 x 2
##   name  band
##   <chr> <chr>
## 1 Mick  Stones
## 2 John  Beatles
## 3 Paul  Beatles
```

```
band_instruments
```

```
## # A tibble: 3 x 2
##   name  plays
##   <chr> <chr>
## 1 John  guitar
## 2 Paul  bass
## 3 Keith guitar
```

```
inner_join(band_members, band_instruments)
```

```
## Joining with `by = join_by(name)`
```

```
## # A tibble: 2 x 3
##   name  band    plays
##   <chr> <chr>   <chr>
## 1 John  Beatles guitar
## 2 Paul  Beatles bass
```

9.2.2 Left Joins

Left Joins on the other hand preserves all rows of data frame passed as `x` i.e. first argument irrespective of the fact that matching key record is available in second data table or not.

Example-1:

```
left_join(band_members, band_instruments)
```

```
## Joining with `by = join_by(name)`
```

```
## # A tibble: 3 x 3
##   name  band    plays
##   <chr> <chr>   <chr>
## 1 Mick  Stones  <NA>
## 2 John  Beatles guitar
## 3 Paul  Beatles bass
```

9.2.3 Right Joins

Right join, is similar to left join and preserves all rows of data frame passed as `y` i.e. second argument irrespective of the fact that matching key record is available in first data table or not.

Example-1:

```
right_join(band_members, band_instruments)
```

```
## Joining with `by = join_by(name)`
```

```
## # A tibble: 3 x 3
##   name  band    plays
##   <chr> <chr>   <chr>
## 1 John  Beatles guitar
## 2 Paul  Beatles bass
## 3 Keith <NA>    guitar
```

9.2.4 Full Joins

Full join returns all the rows of both the data tables despite non-availability of matching key in either of the tables.

Example-1:

```
full_join(band_members, band_instruments)
```

```
## Joining with `by = join_by(name)`
```

```
## # A tibble: 4 x 3
##   name  band    plays
##   <chr> <chr>   <chr>
## 1 Mick  Stones  <NA>
## 2 John  Beatles guitar
## 3 Paul  Beatles bass
## 4 Keith <NA>    guitar
```

You must have noticed that each of the examples shown above has thrown a warning that join has been performed on variable `name`. We may override this warning by specifically providing the joining *key* column name(s) in `by` argument i.e. `by = "name"`.

There may be cases when the joining *key* column(s) in the two data frames are of different names. These cases can also be handled by using `by` argument.

Example-

```
band_instruments2
```

```
## # A tibble: 3 x 2
##   artist plays
##   <chr>  <chr>
## 1 John   guitar
## 2 Paul   bass
## 3 Keith  guitar
```

```
left_join(band_members, band_instruments2, by = c('name' = 'artist'))
```

```
## # A tibble: 3 x 3
##   name  band    plays
##   <chr> <chr>   <chr>
## 1 Mick  Stones  <NA>
## 2 John  Beatles guitar
## 3 Paul  Beatles bass
```

> Note: Each of the `*_join()` can be joined on multiple keys/columns (i.e. more than one) using `by` argument as explained above.

Many-to-many Joins

Example of four of the above-mentioned joins are many to many joins and thus need to be used carefully. See the following example

```
df1 <- data.frame(
  x = c(1, 1, NA),
  y = c(11, 12, 21)
)
df1
```

```
##    x  y
## 1  1 11
## 2  1 12
## 3 NA 21
```

```
df2 <- data.frame(
  x = c(1, 1, NA, NA),
  z = c(101, 102, 201, 202)
)
df2
```

```
##    x   z
## 1  1 101
## 2  1 102
## 3 NA 201
## 4 NA 202
```

```
df1 %>% left_join(df2, by = 'x')
```

```
## Warning in left_join(., df2, by = "x"): Detected an unexpected many-to-many relationship
## between `x` and `y`.
## i Row 1 of `x` matches multiple rows in `y`.
## i Row 1 of `y` matches multiple rows in `x`.
## i If a many-to-many relationship is expected, set
##   `relationship = "many-to-many"` to silence this
##   warning.
```

```
##    x  y   z
## 1  1 11 101
## 2  1 11 102
## 3  1 12 101
## 4  1 12 102
## 5 NA 21 201
## 6 NA 21 202
```

9.3 Filtering Joins

The other joins available in `dplyr` are basically filtering joins.

9.3.1 Semi Joins

First of these is `semi_join` which essentially filters those rows from a data frame, which are based on another data frame. See this example-

```
df1 <- data.frame(
  x = c(1, 2, NA),
  y = c(11, 21, 100)
)
df1
```

```
##    x   y
## 1  1  11
## 2  2  21
## 3 NA 100
```

```
df2 <- data.frame(
  x = c(1, 3, 4, NA),
  z = c(101, 301, 401, 501)
)
df2
```

```
##    x   z
## 1  1 101
## 2  3 301
## 3  4 401
## 4 NA 501
```

```
df1 %>% semi_join(df2, by = 'x')
```

```
##    x   y
## 1  1  11
## 2 NA 100
```

9.3.2 Anti Joins

The `anti_join` is basically opposite to that of `semi_join`. It keeps only those records from left data-frame which are not available in right data-frame. Example

```
df1 %>% anti_join(df2, by = 'x')
```

```
##   x  y
## 1 2 21
```

Chapter 10

Data Wrangling in `tidyr`

In this chapter we will learn about reshaping data to the format most suitable for our data analysis work. To reshape the data, we will use `tidyr` package (Wickham et al., 2023d) which is part of core `tidyverse` and can be loaded either by calling `library(tidyr)` or `library(tidyverse)`.

10.1 Prerequisites

```
library(tidyverse)
```

10.2 Concepts of tidy data

Hadley Wickham, the chief scientist behind development of RStudio, tidyverse, and much more, introduced the concept of tidy data in a paper[1] published in the Journal of Statistical Software (**?**). Tidy data is a framework to structure data sets so they can be easily analyzed and visualized. It can be thought of as a goal one should aim for when cleaning data. Once we understand what tidy data is, that knowledge will make our data analysis, visualization, and collection much easier. (**?**)

A tidy data-set has the following properties:

1. Each variable forms a column.
2. Each observation forms a row.
3. Each type of observational unit forms a table.

[1] https://www.jstatsoft.org/article/view/v059i10

Diagrammatically[2] this can be represented as in figure 10.1.

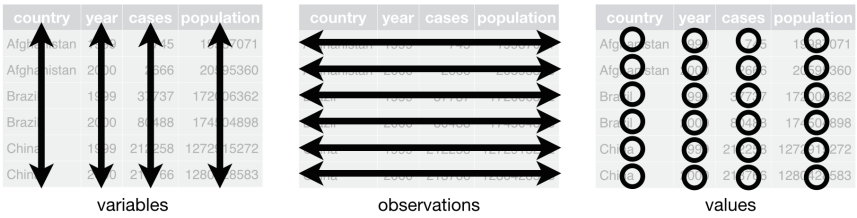


Figure 10.1: Diagrammatic representation of tidy data

Once a dataset is tidy, it can be used as input into a variety of other functions that may transform, model, or visualize the data.

Consider these five examples[3]. All these examples[4] represent same data but shown in different formats-

```
#Example-1
tidyr::table1
```

```
## # A tibble: 6 x 4
##   country      year  cases population
##   <chr>       <dbl>  <dbl>      <dbl>
## 1 Afghanistan  1999    745   19987071
## 2 Afghanistan  2000   2666   20595360
## 3 Brazil       1999  37737  172006362
## 4 Brazil       2000  80488  174504898
## 5 China        1999 212258 1272915272
## 6 China        2000 213766 1280428583
```

```
# Example-2
table2
```

```
## # A tibble: 12 x 4
##    country      year type            count
##    <chr>       <dbl> <chr>           <dbl>
##  1 Afghanistan  1999 cases             745
```

[2]Image taken from Hadley Wickham's book R for data science

[3]all taken from package `tidyr`

[4]These data-tables display the number of TB cases documented by the World Health Organization in Afghanistan, Brazil, and China between 1999 and 2000. The data contains values associated with four variables (country, year, cases, and population), but each table organizes the values in a different layout.

```
##  2 Afghanistan  1999 population   19987071
##  3 Afghanistan  2000 cases            2666
##  4 Afghanistan  2000 population   20595360
##  5 Brazil       1999 cases           37737
##  6 Brazil       1999 population  172006362
##  7 Brazil       2000 cases           80488
##  8 Brazil       2000 population  174504898
##  9 China        1999 cases          212258
## 10 China        1999 population 1272915272
## 11 China        2000 cases          213766
## 12 China        2000 population 1280428583
```

```
# Example-3
table3
```

```
## # A tibble: 6 x 3
##   country      year rate
##   <chr>       <dbl> <chr>
## 1 Afghanistan  1999 745/19987071
## 2 Afghanistan  2000 2666/20595360
## 3 Brazil       1999 37737/172006362
## 4 Brazil       2000 80488/174504898
## 5 China        1999 212258/1272915272
## 6 China        2000 213766/1280428583
```

```
# Example-4 (Same data in 2 data tables now)
table4a
```

```
## # A tibble: 3 x 3
##   country     `1999` `2000`
##   <chr>        <dbl>  <dbl>
## 1 Afghanistan    745   2666
## 2 Brazil       37737  80488
## 3 China       212258 213766
```

```
table4b
```

```
## # A tibble: 3 x 3
##   country         `1999`     `2000`
##   <chr>            <dbl>      <dbl>
## 1 Afghanistan   19987071   20595360
## 2 Brazil       172006362  174504898
## 3 China       1272915272 1280428583
```

```
# Example-5
table5
```

```
## # A tibble: 6 x 4
##   country     century year  rate
##   <chr>       <chr>   <chr> <chr>
## 1 Afghanistan 19      99    745/19987071
## 2 Afghanistan 20      00    2666/20595360
## 3 Brazil      19      99    37737/172006362
## 4 Brazil      20      00    80488/174504898
## 5 China       19      99    212258/1272915272
## 6 China       20      00    213766/1280428583
```

Let us discuss these example one by one -

- `table1` fulfills all three rules stated above and is thus, in tidy format. Notice that every observation has its own row, and each variable is stored in a separate column.
- `table2` stores one observation in two columns (separately for cases and population) and is thus not tidy.
- `table3` stores two variables in one column (cases and population together) and is thus not tidy.
- `table4a` and `table4b` clearly stores one observation in two different tables and is thus not tidy. We may further notice that both these tables use values as column headers, which also violate rule no.3 stated above.
- `table5` again stores one variable i.e. year in two separate columns, and thus does not follow rule no.2 stated above.

10.3 Reshaping data

In real world problems we will mostly come across data-sets that are not in tidy formats and for data analysis, visualisation we will need `tidying` the datasets. To do this we will first need to understand what actually is a value, a variable/field and an observation. As a second step of *tidying* we will require to reshape the data by either -

- re-organising the observation originally spread into multiple rows (e.g. `table2`), in one row; OR
- re-organising the variable spread into multiple columns (e.g. `table3`, etc.), in one single column.

To perform this *tidying* exercise, we will need two most important functions from `tidyr` i.e. `pivot_longer` and `pivot_wider`. So let us understand the functioning of these.

10.3.1 LONGER format through function `pivot_longer()`

Often we will come across data sets that will have values (instead of actual variable name) as column headers. Let us take the example of `table4a` or `table4b` shown above. Both these tables have values of variable `year` as column names. So we need to re-structure these tables into a longer format where these values form part of columns instead of column names. We will use `pivot_longer()` function for this. Th basic syntax is-

```
pivot_longer(
  data,
  cols,
  names_to = "name",
  values_to = "value",
  ...
)
```

Note that there many more useful arguments to this function, but first let us consider on these only.

- `cols` indicate names of columns (as a character vector) to be converted into longer format
- `names_to = "name"` argument will actually convert values used as column headers back to a column with given "name"
- `values_to = "value"` argument will convert values of all those columns back into one column with given name "value" (e.g. population in table4b)

Basic functionality can be understood using the following example-

```
iris_summary <- iris %>%
  group_by(Species) %>%
  summarise(mean = mean(Sepal.Width),
            st_dev = sd(Sepal.Width))
iris_summary
```

```
## # A tibble: 3 x 3
##   Species     mean st_dev
##   <fct>      <dbl>  <dbl>
## 1 setosa      3.43  0.379
## 2 versicolor  2.77  0.314
## 3 virginica   2.97  0.322
```

Using `pivot_wider` we can convert column headers into values. Check the diagram in figure 10.2.

Now we are ready to convert `table4a` into a tidier format.

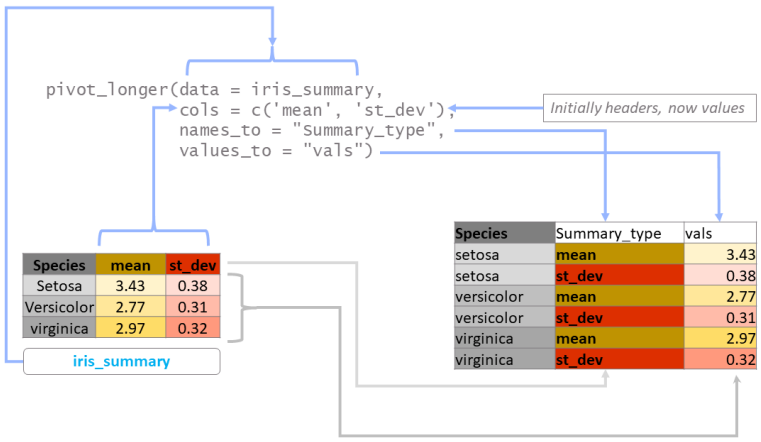


Figure 10.2: Diagrammatic representation of pivot_longer

Case-I when values are in column headers

```
pivot_longer(
  table4a,
  cols = c('1999', '2000'),
  names_to = 'year',
  values_to = 'cases')
```

```
## # A tibble: 6 x 3
##   country     year   cases
##   <chr>       <chr>  <dbl>
## 1 Afghanistan 1999     745
## 2 Afghanistan 2000    2666
## 3 Brazil      1999   37737
## 4 Brazil      2000   80488
## 5 China       1999  212258
## 6 China       2000  213766
```

With *pipes* and a little tweaking, the above syntax could have been written as-

```
table4a %>%                          # first argument passed through pipe
  pivot_longer(cols = -country,      # all columns except country
               names_to = "year",
               values_to = "cases")
```

Case-II when both variables and variable names are combined together as column names

We have seen a simple case to tidy the table when the values (e.g. years) were depicted as column names instead of variables i.e. actual data. There may be cases when column names are a combination of both.

Example - Say we have a table `table6` as

```
## # A tibble: 3 x 5
##   country     cases_1999 cases_2000   pop_1999 pop_2000
##   <chr>            <dbl>      <dbl>      <dbl>    <dbl>
## 1 Afghanistan        745       2666   19987071   2.06e7
## 2 Brazil           37737      80488  172006362   1.75e8
## 3 China           212258     213766 1272915272   1.28e9
```

We may use `names_sep` argument in this case, which will separate the combined variables from the column names -

```
table6 %>%
  pivot_longer(cols = !country,
               names_sep = "_",
               names_to = c("count_type", "year"),
               values_to = "count")
```

```
## # A tibble: 12 x 4
##    country     count_type year       count
##    <chr>       <chr>      <chr>      <dbl>
##  1 Afghanistan cases      1999         745
##  2 Afghanistan cases      2000        2666
##  3 Afghanistan pop        1999    19987071
##  4 Afghanistan pop        2000    20595360
##  5 Brazil      cases      1999       37737
##  6 Brazil      cases      2000       80488
##  7 Brazil      pop        1999   172006362
##  8 Brazil      pop        2000   174504898
##  9 China       cases      1999      212258
## 10 China       cases      2000      213766
## 11 China       pop        1999  1272915272
## 12 China       pop        2000  1280428583
```

Note that we have two column names in argument `names_to`.

Though the above table is still not in tidy format, yet the example was taken to show the functioning of other arguments of the `pivot_longer`. We provided

two static column names to the related argument and the variables were created after splitting the headers with sep `_`. We actually require one dynamic value to be retained as column name (`cases` and `pop` here) but need to convert `year` to variables.

To do so we will use special value `".value"` in the related argument. See

```
table6 %>%
  pivot_longer(cols = !country,
               names_sep = "_",
               names_to = c(".value", "year"),
               values_to = "count")
```

```
## # A tibble: 6 x 4
##   country     year   cases        pop
##   <chr>       <chr>  <dbl>      <dbl>
## 1 Afghanistan 1999     745   19987071
## 2 Afghanistan 2000    2666   20595360
## 3 Brazil      1999   37737  172006362
## 4 Brazil      2000   80488  174504898
## 5 China       1999  212258 1272915272
## 6 China       2000  213766 1280428583
```

> Note that by using `.value` the argument `values_to` becomes meaning less.

10.3.2 WIDER format through `pivot_wider()`

As the name suggests, `pivot_wider()` does exactly opposite to what a `pivot_longer` does. Additionally, this function is used to create summary reports, as pivot functionality in MS Excel, through `values_fn` argument. Diagrammatically this can be represented as in figure 10.3. The basic syntax (with commonly used arguments) is-

```
pivot_wider(
  data,
  id_cols = NULL,
  names_from = name,
  values_from = value,
  values_fill = NULL,
  values_fn = NULL,
  ...
)
```

where-

- `id_cols` is a vector of columns that uniquely identifies each observation
- `names_from` is a vector of columns to get the name of the output column
- `values_from` similarly provides columns to get the cell values from
- `values_fill` provides what each value should be filled in with when missing
- `values_fn` is a named list - to apply different aggregations to different `values_from` columns

  Also note that there are many other arguments for this function, which may be used to deal with complicated tables.

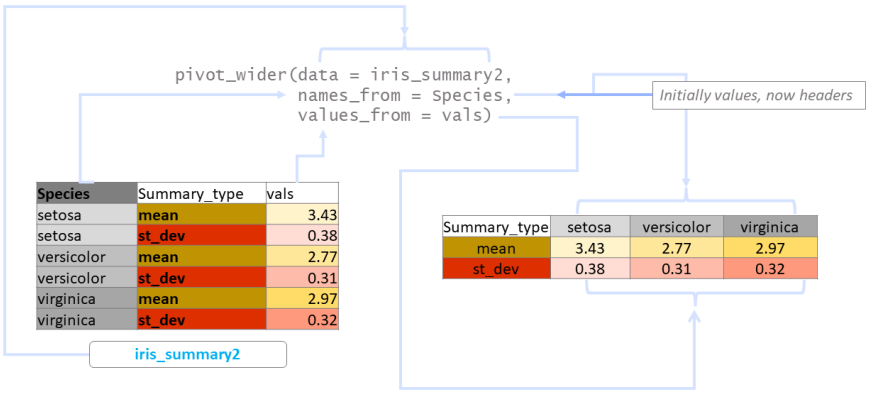


Figure 10.3: Diagrammatic representation of pivot_wider

Example-1:

```
table2 %>%
  pivot_wider(names_from = "type",
              values_from = "count")
```

```
## # A tibble: 6 x 4
##   country      year  cases population
##   <chr>       <dbl>  <dbl>      <dbl>
## 1 Afghanistan  1999    745   19987071
## 2 Afghanistan  2000   2666   20595360
## 3 Brazil       1999  37737  172006362
## 4 Brazil       2000  80488  174504898
## 5 China        1999 212258 1272915272
## 6 China        2000 213766 1280428583
```

Example-2:

```
table2 %>%
  pivot_wider(names_from = year,
              values_from = count)
```

```
## # A tibble: 6 x 4
##   country     type           `1999`     `2000`
##   <chr>       <chr>           <dbl>      <dbl>
## 1 Afghanistan cases             745       2666
## 2 Afghanistan population   19987071   20595360
## 3 Brazil      cases           37737      80488
## 4 Brazil      population  172006362  174504898
## 5 China       cases          212258     213766
## 6 China       population 1272915272 1280428583
```

Example-3: Summarisation

```
table2 %>%
  pivot_wider(id_cols = country,
              names_from = type,
              values_from = count,
              values_fn = mean)
```

```
## # A tibble: 3 x 3
##   country       cases  population
##   <chr>         <dbl>       <dbl>
## 1 Afghanistan   1706.   20291216.
## 2 Brazil       59112.  173255630
## 3 China       213012  1276671928.
```

Example-4: Summarisation with different id_cols

```
table2 %>%
  pivot_wider(id_cols = year,
              names_from = type,
              values_from = count,
              values_fn = sum)
```

```
## # A tibble: 2 x 3
##    year  cases population
##   <dbl>  <dbl>      <dbl>
## 1  1999 250740 1464908705
## 2  2000 296920 1475528841
```

Example-5: Use of multiple columns in `names_from` argument

```
table2 %>%
  pivot_wider(names_from = c(year, type),
              values_from = count)
```

```
## # A tibble: 3 x 5
##   country   `1999_cases` `1999_population` `2000_cases`
##   <chr>            <dbl>             <dbl>        <dbl>
## 1 Afghanis~          745          19987071         2666
## 2 Brazil           37737         172006362        80488
## 3 China           212258        1272915272       213766
## # i 1 more variable: `2000_population` <dbl>
```

What if order is reversed in `names_from` arg. See Example-6:

```
table2 %>%
  pivot_wider(names_from = c(type, year),
              values_from = count)
```

```
## # A tibble: 3 x 5
##   country     cases_1999 population_1999 cases_2000
##   <chr>            <dbl>           <dbl>      <dbl>
## 1 Afghanistan        745        19987071       2666
## 2 Brazil           37737       172006362      80488
## 3 China           212258      1272915272     213766
## # i 1 more variable: population_2000 <dbl>
```

Example-7: Multiple columns in `values_from`

```
table1 %>%
  pivot_wider(names_from = year,
              values_from = c(cases, population))
```

```
## # A tibble: 3 x 5
##   country     cases_1999 cases_2000 population_1999
##   <chr>            <dbl>      <dbl>           <dbl>
## 1 Afghanistan        745       2666        19987071
## 2 Brazil           37737      80488       172006362
## 3 China           212258     213766      1272915272
## # i 1 more variable: population_2000 <dbl>
```

Example-8: Use of names_vary argument to control the order of output columns

```
table1 %>%
  pivot_wider(names_from = year,
              values_from = c(cases, population),
              names_vary = "slowest")
```

```
## # A tibble: 3 x 5
##   country     cases_1999 population_1999 cases_2000
##   <chr>            <dbl>           <dbl>      <dbl>
## 1 Afghanistan        745        19987071       2666
## 2 Brazil           37737       172006362      80488
## 3 China           212258      1272915272     213766
## # i 1 more variable: population_2000 <dbl>
```

For more details please refer to package vignette or Chapter-12 of R for Data Science book.

10.3.3 Separate Column(s) into multiple columns/ Join columns into one column

Separate a character column into multiple with `separate()`

As the name suggests, `separate()` function is used to separate a given character column into multiple columns either using a regular expression or a vector of character positions. The syntax is -

```
separate(
  data,
  col,
  into,
  sep = "[^[:alnum:]]+",
  remove = TRUE,
  convert = FALSE,
  extra = "warn",
  fill = "warn",
  ...
)
```

Explanation of purpose of different arguments in above syntax -

1. `data` is as usual name of the data frame
2. `col` is the name of the column which is required to be separated.

3. `into` should be a character vector, which usually should be equal length of maximum number of new columns which will be created out of such separation. (Refer examples nos. 1)
4. `sep` provides a separator value. (Refer Example -3 below).
5. `remove` if `FALSE`, the original column is not removed from the output. (Refer Example -3 below).
6. `convert` if `TRUE`, the component columns are converted to double/integer/logical/NA, if possible. This is useful if the component columns are integer, numeric or logical. (Refer Example-1 below).
7. `extra` argument is used to control when number of desired component columns are less than the maximum possible count. (Refer Example-4 below).
8. `fill` argument is on the other hand, useful when the number of components are different for each row. (Refer Example-2 below)

Example-1:

```
table3 %>%
  separate(rate, into = c("cases", "population"),
           convert = TRUE) # optional - will convert the values
```

```
## # A tibble: 6 x 4
##   country      year  cases population
##   <chr>       <dbl>  <int>      <int>
## 1 Afghanistan  1999    745   19987071
## 2 Afghanistan  2000   2666   20595360
## 3 Brazil       1999  37737  172006362
## 4 Brazil       2000  80488  174504898
## 5 China        1999 212258 1272915272
## 6 China        2000 213766 1280428583
```

Example-2:

```
data.frame(
  x = c("a", "a+b", "c+d+e")
) %>%
  separate(x,
           into=c('X1', 'X2', 'X3'),
           fill = "left")
```

```
##     X1   X2 X3
## 1 <NA> <NA>  a
## 2 <NA>    a  b
## 3    c    d  e
```

Example-3:

```
data.frame(
  x = c("A$B", "C+D", "E-F")
) %>%
  separate(x,
           sep = "\\-|\\$",
           into = c('X1', 'X2'),
           remove = FALSE)
```

```
## Warning: Expected 2 pieces. Missing pieces filled with `NA` in
## 1 rows [2].
```

```
##     x  X1   X2
## 1 A$B   A    B
## 2 C+D C+D <NA>
## 3 E-F   E    F
```

Example-4:

```
data.frame(
  x = c("a", "a+b", "c+d+e")
) %>%
  separate(x,
           into=c('X1', 'X2'),
           extra = "merge")
```

```
## Warning: Expected 2 pieces. Missing pieces filled with `NA` in
## 1 rows [1].
```

```
##   X1   X2
## 1  a <NA>
## 2  a    b
## 3  c  d+e
```

Unite multiple character columns into one using `unite()`

It complements `separate` by uniting the columns into one. Its syntax is

```
unite(data,
      col,
      ...,
      sep = "_",
      remove = TRUE,
      na.rm = FALSE)
```

Explanation of arguments in the above syntax-

1. `data` is as usual name of the data frame.
2. `col` should be the name of new column to be formed (should be a string),
3. `...` the names of columns to be united should be provided
4. `sep` is separator to be used for uniting
5. `remove` if `FALSE`, will not remove original component columns
6. `na.rm` if `TRUE`, the missing values will be removed beforehand.

Example-1a:

```
table5 %>%
  unite("Year",
        c("century", "year"),
        sep = "",
        remove = FALSE)
```

```
## # A tibble: 6 x 5
##   country     Year  century year  rate
##   <chr>       <chr> <chr>   <chr> <chr>
## 1 Afghanistan 1999  19      99    745/19987071
## 2 Afghanistan 2000  20      00    2666/20595360
## 3 Brazil      1999  19      99    37737/172006362
## 4 Brazil      2000  20      00    80488/174504898
## 5 China       1999  19      99    212258/1272915272
## 6 China       2000  20      00    213766/1280428583
```

Example-1b: We may complete the tidying process in the next step

```
table5 %>%
  unite("Year",
        c("century", "year"),
        sep = "") %>%
  separate(rate,
           into = c("cases", "population"),
           convert = TRUE)
```

```
## # A tibble: 6 x 4
##   country     Year   cases population
##   <chr>       <chr>  <int>      <int>
## 1 Afghanistan 1999     745   19987071
## 2 Afghanistan 2000    2666   20595360
## 3 Brazil      1999   37737  172006362
## 4 Brazil      2000   80488  174504898
## 5 China       1999  212258 1272915272
## 6 China       2000  213766 1280428583
```

Example-

10.3.4 Separate row(s) into multiple rows

Split data into multiple rows with `separate_rows()`

This function is used to separate delimited values placed in one single cell/column into multiple rows (as against in rows using `separate`). See Example-

```
table3 %>%
  separate_rows(
    rate,
    sep = "/",
    convert = TRUE
  )
```

```
## # A tibble: 12 x 3
##    country      year       rate
##    <chr>       <dbl>      <int>
##  1 Afghanistan  1999        745
##  2 Afghanistan  1999   19987071
##  3 Afghanistan  2000       2666
##  4 Afghanistan  2000   20595360
##  5 Brazil       1999      37737
##  6 Brazil       1999  172006362
##  7 Brazil       2000      80488
##  8 Brazil       2000  174504898
##  9 China        1999     212258
## 10 China        1999 1272915272
## 11 China        2000     213766
## 12 China        2000 1280428583
```

To create `type` we have to however, add an extra step to the above example-

```
table3 %>%
  separate_rows(
    rate,
    sep = "/",
    convert = TRUE
  ) %>%
  group_by(country, year) %>%
  mutate(type = c("cases", "pop"))
```

```
## # A tibble: 12 x 4
## # Groups:   country, year [6]
##    country      year       rate type
##    <chr>       <dbl>      <int> <chr>
##  1 Afghanistan  1999        745 cases
##  2 Afghanistan  1999   19987071 pop
##  3 Afghanistan  2000       2666 cases
##  4 Afghanistan  2000   20595360 pop
##  5 Brazil       1999      37737 cases
##  6 Brazil       1999  172006362 pop
##  7 Brazil       2000      80488 cases
##  8 Brazil       2000  174504898 pop
##  9 China        1999     212258 cases
## 10 China        1999 1272915272 pop
## 11 China        2000     213766 cases
## 12 China        2000 1280428583 pop
```

10.3.5 Expand table to handle missing rows/values

Sometimes, when we deal with missing data, we require to handle implicit missing values as well. Either we have to turn these values into explicit missing values or we have to fill appropriate values. In such cases, two functions namely `complete` and `fill` both from same package are extremely useful. Let's learn these as well.

Turn implicit missing values to explicit using `complete()`

As th name suggests, this function is used to turn implicit missing values (invisible rows) to explicit missing values (visible rows with `NA`).

As an example, let's suppose fuel prices are revised randomly. Say, after revision on `1 January 2020` prices revise on `20 January 2020`. So we will have only 2 rows in data for say `Janaury 2020`. Thus, there are 29 implicit missing values in the data.

The syntax is

```
complete(data,
         ...,
         fill = list(),
         explicit = TRUE)
```

Where -

- `data` is as usual argument to provide data frame

- `...` are meant to provide column names to be completed
- `fill` provides a list to supply a single value which can be provided instead of `NA` for missing combinations.

In the example mentioned above we can proceed as

```
# First create a sample data
df <- data.frame(
  date = c(as.Date("2020-01-01"), as.Date("2020-01-20")),
  price = c(75.12, 78.32)
)
df
```

```
##         date price
## 1 2020-01-01 75.12
## 2 2020-01-20 78.32
```

```
# Use tidyr::complete to see explicit missing values
df %>%
  complete(date = seq.Date(as.Date("2020-01-01"), as.Date("2020-01-31"), by = "day"))
```

```
## # A tibble: 31 x 2
##    date       price
##    <date>     <dbl>
##  1 2020-01-01  75.1
##  2 2020-01-02  NA
##  3 2020-01-03  NA
##  4 2020-01-04  NA
##  5 2020-01-05  NA
##  6 2020-01-06  NA
##  7 2020-01-07  NA
##  8 2020-01-08  NA
##  9 2020-01-09  NA
## 10 2020-01-10  NA
## # i 21 more rows
```

Though the above example created dates as per the criteria given, `complete` function can find all unique combinations in the set of columns provided and return complete set of observations. See this example

```
#Let's create a sample data
set.seed(123)
df2 <- data.frame(
  year = c(2020, 2020, 2020, 2021, 2021),
```

```
  qtr = c(1,3,4,2,3),
  sales = runif(5, 100, 200)
)
df2
```

```
##   year qtr sales
## 1 2020   1 128.8
## 2 2020   3 178.8
## 3 2020   4 140.9
## 4 2021   2 188.3
## 5 2021   3 194.0
```

```
# use complete to find all combination
df2 %>%
  complete(year, qtr, # cols provided
           fill = list(sales = 0))
```

```
## # A tibble: 8 x 3
##    year   qtr sales
##   <dbl> <dbl> <dbl>
## 1  2020     1  129.
## 2  2020     2    0
## 3  2020     3  179.
## 4  2020     4  141.
## 5  2021     1    0
## 6  2021     2  188.
## 7  2021     3  194.
## 8  2021     4    0
```

Note: If `fill` argument would not have been used, the value of sales in missing columns would have been `NA` instead of provided value.

Fill missing values based on criteria `fill()`

This function helps in filling the missing values using previous or next entry. Think of a paper sheet where many cells in a table have been filled as `"-do-"`. Syntax is -

```
fill(data, # used to provide data
     ..., # provide columns to be filled
     .direction = c("down", "up", "downup", "updown"))
```

The arguments are pretty simple. Most of the time `"down"` method is used. As an example we could see the example of fuel prices mentioned above.

```
fill_df <- df %>%
  complete(date = seq.Date(as.Date("2020-01-01"), as.Date("2020-01-31"), by = "day")) %
  fill(price, .direction = "down")

head(fill_df)
```

```
## # A tibble: 6 x 2
##   date       price
##   <date>     <dbl>
## 1 2020-01-01  75.1
## 2 2020-01-02  75.1
## 3 2020-01-03  75.1
## 4 2020-01-04  75.1
## 5 2020-01-05  75.1
## 6 2020-01-06  75.1
```

```
tail(fill_df)
```

```
## # A tibble: 6 x 2
##   date       price
##   <date>     <dbl>
## 1 2020-01-26  78.3
## 2 2020-01-27  78.3
## 3 2020-01-28  78.3
## 4 2020-01-29  78.3
## 5 2020-01-30  78.3
## 6 2020-01-31  78.3
```

Part III: Case Studies

Chapter 11

Data Cleaning in R

Data cleansing is one of the important steps in data analysis. Multiple packages are available in r to clean the data sets. One of such packages is `janitor` which we will be using in this chapter along with few other packages.

Let's load it

```
library(janitor)
```

```
##
## Attaching package: 'janitor'
```

```
## The following objects are masked from 'package:stats':
##
##     chisq.test, fisher.test
```

11.1 Cleaning Column names.

We know that names of objects in R follow certain conventions like we may not have certain special characters in names. If a space has been used that is to be quoted under a pair of backticks. But generally when we read data from files in excel, we can have some 'dirty' names, which we should clean before proceeding. In such `clean_names()` come handy. E.g.

```
# Create a data.frame with dirty names
test_df <- as.data.frame(matrix(ncol = 6))

names(test_df) <- c("firstName", "ábc@!*", "% successful (2009)",
```

```
                    "REPEAT VALUE", "REPEAT VALUE", "")
# View this data
test_df
```

```
##   firstName ábc@!* % successful (2009) REPEAT VALUE
## 1        NA     NA                   NA           NA
##   REPEAT VALUE
## 1           NA NA
```

Using `clean_names()` which is also pipe friendly, we can clean names in one step. (Results will be in snake case)

```
test_df %>%
  clean_names()
```

```
##   first_name abc percent_successful_2009 repeat_value
## 1         NA  NA                      NA           NA
##   repeat_value_2  x
## 1             NA NA
```

It -

- Parses letter cases and separators to a consistent format.
- Default is to `snake_case`, but other cases like `camelCase` are available
- Handles special characters and spaces, including transliterating characters like œ to `oe`.
- Appends numbers to duplicated names
- Converts `"%"` to "percent" and `"#"` to `"number"` to retain meaning
- Spacing (or lack thereof) around numbers is preserved

11.2 Handling duplicate columns

11.3 Handling duplicate records

11.4 Remove Constant (Redundant) columns

11.5 Remove empty rows and/or columns

11.6 Fix excel dates stored as serial numbers

11.7 Convert a mix of date and datetime formats to date

Chapter 12

Random sampling in R

International Standard On Auditing - 530 defines[1] audit sampling as *the application of audit procedures to less than 100% of items within a population of audit relevance such that all sampling units have a chance of selection in order to provide the auditor with a reasonable basis on which to draw conclusions about the entire population.* Statistical sampling is further defines as *an approach to sampling having two characteristics - random selection of samples, and the use of probability theory to evaluate sample results, including measurement of sampling risk.*

Appendix 4 of ISA 53 further prescribes different statistical methods of sample selection. We will discuss here each type of sampling methodology used to sample records for audit.

Prerequisites

Load `tidyverse`

```
library(tidyverse)
```

12.1 Simple Random Sampling (With and without replacement)

In this method, records are selected completely at random, by generating random numbers e.g. using random number tables, etc. Refer figure 12.1 for illustration. We can replicate the method of random number generation in R. Even the

[1] https://www.ifac.org/system/files/downloads/a027-2010-iaasb-handbook-isa-530.pdf

method of random number generation can be reproducible, by fixing the random number seed. Mainly two functions will be used here `sample()` and `set.seed()` already discussed in section 3.4. Since `sample()` function takes a vector as input and gives vector as output again, we can make use of `dplyr::slice_sample()` function, discussed in section 3.4, which operates on data frames instead.

Figure 12.1: Illustration of Simple Random Sampling

Let's see this sampling on `iris` data. Suppose we have to select a sample of `n=12` records, without replacement-

```
dat <- iris # input data
# set the seed
set.seed(123)
# sample n records
dat %>%
  slice_sample(n = 12, replace = FALSE)
```

Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
4.3	3.0	1.1	0.1	setosa
5.0	3.3	1.4	0.2	setosa
7.7	3.8	6.7	2.2	virginica
4.4	3.2	1.3	0.2	setosa
5.9	3.0	5.1	1.8	virginica
6.5	3.0	5.2	2.0	virginica
5.5	2.5	4.0	1.3	versicolor
5.5	2.6	4.4	1.2	versicolor
5.8	2.7	5.1	1.9	virginica
6.1	3.0	4.6	1.4	versicolor
6.3	3.4	5.6	2.4	virginica
5.1	2.5	3.0	1.1	versicolor

The syntax is simple. In the first step we have fixed the random number seed for reproducibility. Using `slice_sample()` we have selected `n=12` records without replacement (`replace = FALSE`).

If sample size is based on some proportion, we have to use `prop = .10` (say 10%) instead of `n` argument. Moreover, if sampling is with replacement, we have to use `replace = TRUE`.

12.2 Systematic random sampling

ISA 530 defines this sampling approach as *'Systematic selection, in which the number of sampling units in the population is divided by the sample size to give a sampling interval, for example 50, and having determined a starting point within the first 50, each 50th sampling unit thereafter is selected. Although the starting point may be determined haphazardly, the sample is more likely to be truly random if it is determined by use of a computerized random number generator or random number tables. When using systematic selection, the auditor would need to determine that sampling units within the population are not structured in such a way that the sampling interval corresponds with a particular pattern in the population.'* Refer figure 12.2 for illustration.

Figure 12.2: Illustration of Systematic Random Sampling

We can replicate this approach again following two steps-

Step-1: Select `n` as the sample size. Then generate a maximum starting point say `s` by dividing number of rows in the data by `n`. Thereafter we have to choose a starting point from `1:s`. We can use sample function here. Let's say this starting number is `s1`. Then we have to generate an arithmetic sequence, say `rand_seq` starting from `s1` and increasing every `s` steps thereafter with total `n` terms.

Step-2: In the next step we will shuffle the data by using `slice_sample` and select a sample using function `filter`.

The methodology is replicated as

```
set.seed(123)
n <- 15 # sample size
s <- floor(nrow(dat)/n)
```

```
s1 <- sample(1:s, 1, replace = FALSE)
rand_seq <- seq(s1, by = s, length.out = n)
dat %>%
  slice_sample(prop = 1) %>%
  filter(row_number() %in% rand_seq)
```

Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
7.7	3.8	6.7	2.2	virginica
6.1	2.8	4.0	1.3	versicolor
7.6	3.0	6.6	2.1	virginica
7.2	3.2	6.0	1.8	virginica
7.2	3.0	5.8	1.6	virginica
6.3	2.7	4.9	1.8	virginica
4.8	3.1	1.6	0.2	setosa
6.4	2.8	5.6	2.1	virginica
5.6	3.0	4.5	1.5	versicolor
4.9	2.5	4.5	1.7	virginica
5.9	3.2	4.8	1.8	versicolor
6.3	3.3	6.0	2.5	virginica
5.4	3.7	1.5	0.2	setosa
6.7	3.1	4.4	1.4	versicolor
5.2	3.5	1.5	0.2	setosa

12.3 Probability Proportionate to size (with or without replacement) a.k.a monetary unit sampling

This sampling approach is defined in ISA-530 as *"a type of value-weighted selection in which sample size, selection and evaluation results in a conclusion in monetary amounts."*

Our methodology is not much difference from methodology adopted in section 12.2 except that we will make use of `weight_by =` argument now.

Let's use `state.x77` data that comes with base R. Since the data is in matrix format, let's first convert it data frame using `as.data.frame()` first.

```
dat <- as.data.frame(state.x77)
```

Other steps are simple.

```
set.seed(123)
```

```
dat %>%
  slice_sample(n=12, weight_by = Population)
```

	Population	Income	Illiteracy	Life Exp	Murder	HS Grad	Frost	Area
Pennsylvania	11860	4449	1.0	70.43	6.1	50.2	126	44966
Kentucky	3387	3712	1.6	70.10	10.6	38.5	95	39650
Michigan	9111	4751	0.9	70.63	11.1	52.8	125	56817
Oregon	2284	4660	0.6	72.13	4.2	60.0	44	96184
Utah	1203	4022	0.6	72.90	4.5	67.3	137	82096
California	21198	5114	1.1	71.71	10.3	62.6	20	156361
Virginia	4981	4701	1.4	70.08	9.5	47.8	85	39780
Arizona	2212	4530	1.8	70.55	7.8	58.1	15	113417
Georgia	4931	4091	2.0	68.54	13.9	40.6	60	58073
Massachusetts	5814	4755	1.1	71.83	3.3	58.5	103	7826
Hawaii	868	4963	1.9	73.60	6.2	61.9	0	6425
New Jersey	7333	5237	1.1	70.93	5.2	52.5	115	7521

12.4 Stratified random sampling

Stratification is defined in ISA-530 as *the process of dividing a population into sub-populations, each of which is a group of sampling units which have similar characteristics (often monetary value).* Thus, stratified random sampling may imply any of the afore-mentioned sampling techniques applied to individual strata instead of whole population. Refer figure 12.3 for illustration.

Figure 12.3: Illustration of Stratified Random Sampling

The function `dplyr::group_by()` will be used here for stratification. Thereafter we can proceed for sampling described as above.

Example Data: - Let's include region in `state.x77` data using `dplyr::bind_cols`.

```
dat <- bind_cols(
  as.data.frame(state.x77),
  as.data.frame(state.region)
)
```

Let's see first 6 rows of this data

	Population	Income	Illiteracy	Life Exp	Murder	HS Grad	Frost	Area	sta
Alabama	3615	3624	2.1	69.05	15.1	41.3	20	50708	Sou
Alaska	365	6315	1.5	69.31	11.3	66.7	152	566432	We
Arizona	2212	4530	1.8	70.55	7.8	58.1	15	113417	We
Arkansas	2110	3378	1.9	70.66	10.1	39.9	65	51945	Sou
California	21198	5114	1.1	71.71	10.3	62.6	20	156361	We
Colorado	2541	4884	0.7	72.06	6.8	63.9	166	103766	We

We can check a summary of number of States per region

```
dat %>%
  tibble::rownames_to_column('State') %>% # this step will not be
                                          # used in databases without row names
  group_by(state.region) %>%
  summarise(states = n())
```

```
## # A tibble: 4 x 2
##   state.region  states
##   <fct>          <int>
## 1 Northeast          9
## 2 South             16
## 3 North Central     12
## 4 West              13
```

Case-1: When the sample size is constant for all strata. Say 2 records per region.

```
set.seed(123)
n <- 2
dat %>%
  tibble::rownames_to_column('State') %>% # this step will not be used in databases wi
  group_by(state.region) %>%
  slice_sample(n=n) %>%
  ungroup()
```

State	Population	Income	Illiteracy	Life Exp	Murder	HS Grad	Frost	Area	state.region
Massachusetts	5814	4755	1.1	71.83	3.3	58.5	103	7826	Northeast
New York	18076	4903	1.4	70.55	10.9	52.7	82	47831	Northeast
Delaware	579	4809	0.9	70.06	6.2	54.6	103	1982	South
North Carolina	5441	3875	1.8	69.21	11.1	38.5	80	48798	South
Indiana	5313	4458	0.7	70.88	7.1	52.9	122	36097	North Cent
Minnesota	3921	4675	0.6	72.96	2.3	57.6	160	79289	North Cent
Utah	1203	4022	0.6	72.90	4.5	67.3	137	82096	West
Hawaii	868	4963	1.9	73.60	6.2	61.9	0	6425	West

Case-2: When the sample size or proportion is different among strata. This time let us assume that column for *stratum* is not directly available in the data. - Say, 20% of States having Population upto `1000`; - 30% of States having population greater than `1000` but upto `5000` and finally; - 50% of states having population more than `5000` have to be sampled.

In this scenario, our strategy would be use `purrr::map2_dfr()` function after splitting the data with `group_split()` function.

Syntax would be

```
# define proportions
props <- c(0.2, 0.3, 0.5)

# set seed
set.seed(123)

# take data
dat %>%
  # reduntant step where data has no column names
  tibble::rownames_to_column('State') %>%
  # create column according to stratums
  mutate(stratum = cut(Population, c(0, 1000, 5000, max(Population)),
                       labels = c("Low", "Mid", "High"))) %>%
  # split data into groups
  group_split(stratum) %>%
  # sample in each group
  map2_dfr(props,
           .f = function(d, w) slice_sample(d, prop = w))
```

We may check the sample selected across each stratum

stratum	Total	Selected
Low	12	2
Mid	26	7
High	12	6

12.5 Cluster sampling

ISA 530 does not explicitly define cluster sampling. Actually this sampling is sampling of strata and we can apply above mentioned techniques easily to sample clusters. E.g. in the sample data above, we can sample say, 2 clusters (or regions).

Thus, our strategy would be first to sample groups from unique available values and thereafter filter all the records.

```
# set the seed
set.seed(123)
# sample clusters
clusters <- sample(
  unique(dat$state.region),
  size = 2
)
# filter all records in above clusters
clust_samp <- dat %>%
  filter(state.region %in% clusters)
# check number of records
clust_samp$state.region %>% table()
```

```
## .
##     Northeast         South North Central
##             9             0            12
##          West
##             0
```

Chapter 13

Benford Law

13.1 History

Benford's Law is named after physicist **Frank Benford**, who worked on the theory in 1938 and as a result a paper titled **The Law of Anomalous Numbers** was published.(**?**). However, its discovery is associated more than five decades earlier when astronomer **Simon Newcomb**, observed that initial pages of log tables booklet were more worn out than later pages and publisged a two page article titled **Note on the Frequency of Use of the Different Digits in Natural Numbers** in 1881 (**?**).

13.2 What the law states

If someone asks us, what is probability of any digit occurring in first place (from left side), by natural intuition we may answer this question is *one out of nine*. In other words, probability of any digit that be in first place should follow uniform distribution. However when the Canadian-American astronomer **Simon Newcomb** was flipping, in 1881, through logarithmic tables booklet, he noticed that initial pages were more worn out than those in the end. Later on, **Frank Benford** by analysing 20 different data-sets, ranging from river sizes, chemical compound weights, population, etc., showed that probability diminishes successively from digit 1 to 9, which means that probability of digit 1 occurring at initial place should be highest and that of 9 should be lowest.

Mathematically, *Benford's Law* or *Law of first digits* states that the probability of any digit in first place should follow -

$P(d = d_i) = log(1 + 1/d_i)$ where $d_i = [1...9]$

The probabilities may be plotted as -

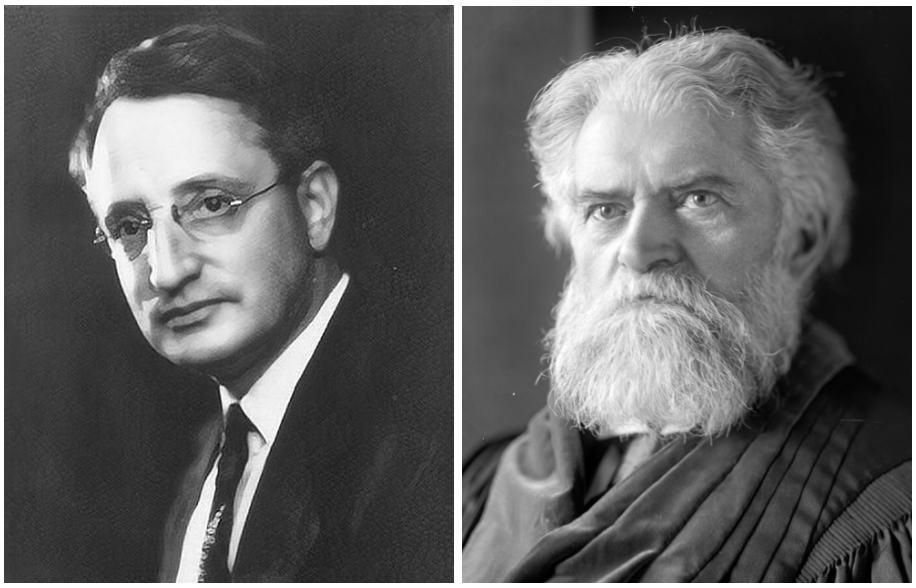


Figure 13.1: Frank Benford and Simon Newcomb

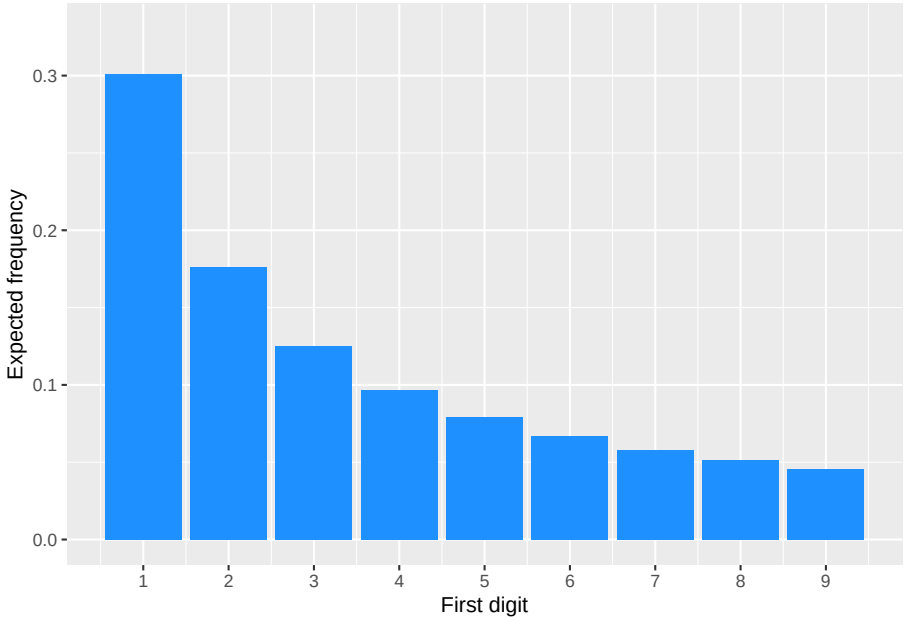


Figure 13.2: Diminishing Probabilities of First Digits - Benford Law

Benford showed it on 20 different data-sets. Later in 1961, Roger Pinkham later showed that law is invariant to scaling (**?**). By *Scale Invariance*, he actually showed that the law is invariant to measurements units i.e. law still holds if we measure price in USD or in INR, measure length in KMs or Miles, etc.

To test the proposed law, Benford showed it on many different datasets. In R, there is a package named `benford.analysis` (Cinelli, 2018) to perform analytics based on this law. Moreover, the library also has many of those datasets on which Benford tested his proposed law.

Let us see the results on dataset named `census.2009` available with the aforementioned package. This data-set contains the figures of population of towns and cities of the United States, as of July of 2009.

```
# Load package benford.analysis
library(benford.analysis)
data(census.2009)
# Check conformity
bfd.cen <- benford(census.2009$pop.2009,
                   number.of.digits = 1)
plot(bfd.cen, except = c("second order",
                         "summation",
                         "mantissa",
                         "chi squared",
                         "abs diff",
                         "ex summation",
                         "Legend"),
     multiple = F)
```

Figure 13.3 shows that the law holds for the data. Let us also test the Pinkham's corollary on the aforesaid data. For this let's multiply all the figures of population by say 3.

```
# Multiply the data by 3 and check conformity again
data <- census.2009$pop.2009 * 3
bfd.cen3 <- benford(data, number.of.digits=1)
plot(bfd.cen3, except = c("second order",
                          "summation",
                          "mantissa",
                          "chi squared",
                          "abs diff",
                          "ex summation",
                          "Legend"),
     multiple = F)
```

Through figure 13.4 it is clear that law still holds after scaling. Theodore P. Hill

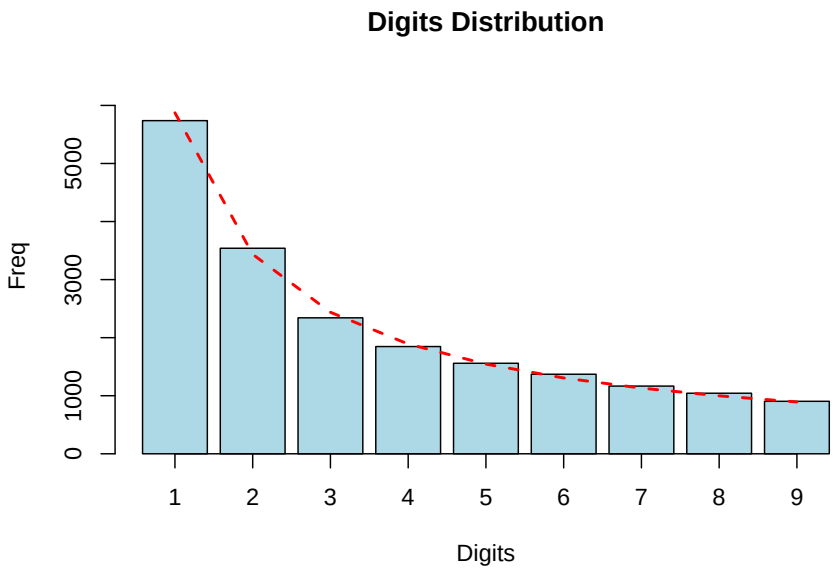


Figure 13.3: First Digit Analysis on US Census 2009 data

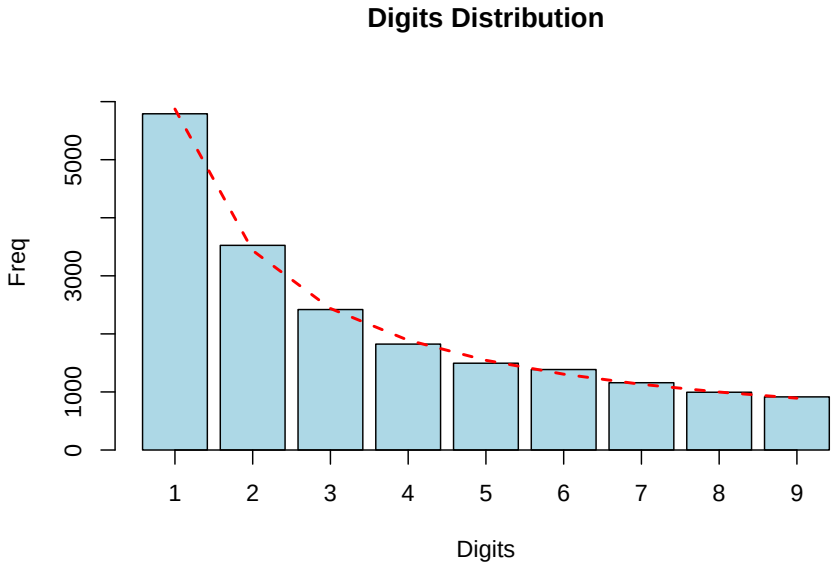


Figure 13.4: Law still holds after scaling the numbers

later-on showed that law extends for further digits as well. (**?**) We can check it for first two digits, in the above example (Refer figure 13.5).

```
bfd.cen2 <- benford(census.2009$pop.2009, number.of.digits = 2)
plot(bfd.cen2, except = c("second order",
                          "summation",
                          "mantissa",
                          "chi squared",
                          "abs diff",
                          "ex summation",
                          "Legend"),
     multiple = F)
```

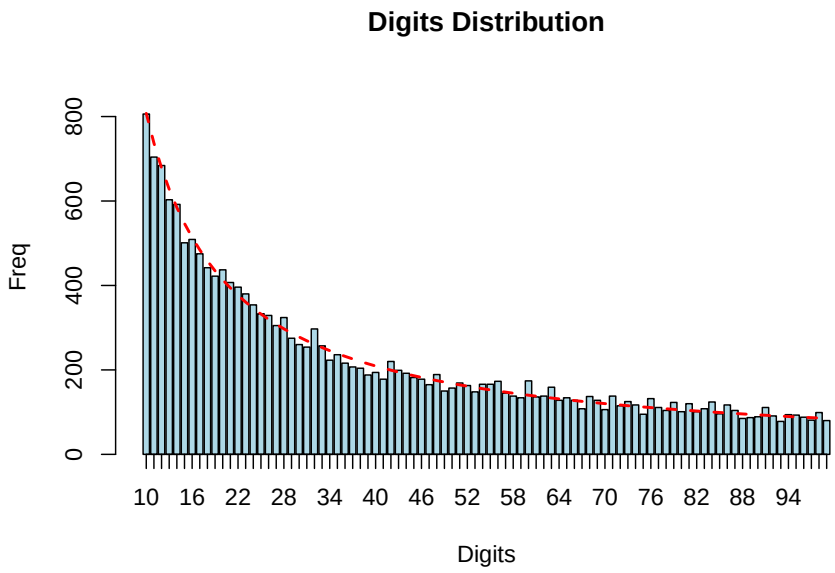


Figure 13.5: Law holds for first two digits as well

13.3 Limitations

Benford's Law may not hold in the following circumstances-

1. When the data-set is comprised of assigned numbers. Like cheque numbers, invoices numbers, telephone numbers, pincodes, etc.
2. Numbers that may be influenced viz. ATM withdrawals, etc.

3. Where amounts have either lower bound, or upper bounds or both. E.g. passengers onboard airplane, hourly wage rate, etc.
4. Count of transactions less than 500.

13.4 Fraud detection using Benford's Law

Dr Mark Nigrini, an accountancy professor from Dallas, has made use of this law for fraud detection. His theory is that - *If somebody tries to falsify, say, their tax return then invariably they will have to invent some data. When trying to do this, the tendency is for people to use too many numbers starting with digits in the mid range, 5,6,7 and thus not enough numbers starting with 1.*

13.5 Second Order Tests

13.5.1 Digits distribution second order Test (?)

Mark J Nigrini, in a reasearch paper published in 2009, proposed this new test which diagnoses the relationships and patterns found in transactional data and is based on the *digits of the differences between the amounts that have been sorted from smallest to largest.* Nigrini showed that these digits are expected to closely follow the frequencies of Benford law. Using four different datasets he showed that this test can detect -

- anamolies occuring in data downloads
- rounded data
- use of 'regression output' OR 'statistically generated data' in place of actual (transactional) data.

13.5.2 Summation Test (?)

The **summation test**, another second order test, looks for excessively large numbers in a dataset. It identifies numbers that are large compared to the norm for that data. The test was proposed by Nigrini (**?**) and it is based on the fact that the sums of all numbers in a Benford distribution with first-two digits (10, 11, 12, ...99) should be the same. Therefore, for each of the 90 first-two digits groups sum proportions should be equal, i.e. 1/90 or 0.011. The spikes, if any indicate that there are some large single numbers or set of numbers.

13.6 Examining significance using statistical tests

13.6.1 Chi-square statistic

The critical value for Chi-Square Test, also known as goodness of fit test, comes from a chi-square distribution. Here we have (9-1) = 8 degrees of freedom. For a significance level of 0.01, we get a critical value of 20.09. If the value of χ^2 is greater than this value, then we can reject a fit to Benford's law with 99% certainty. For significance level of 0.05, we may take 15.507 and 112.022 as critical values for first and first-two digits tests, respectively.

13.6.2 Z-score

Mark Nigrini, in his book, proposed that if the values of Z-statistic exceed the critical value 1.96, the null hypothesis H_{0A} is rejected at 5% of significance level.

13.6.3 Mean absolute deviation

These values prescribed by Mark J Nigrini are -

First Digits		First-Two Digits	
0.000 to 0.006	Close conformity	0.000 to 0.012	Close conformity
0.006 to 0.012	Acceptable conformity	0.012 to 0.018	Acceptable conformity
0.012 to 0.015	Marginally acceptable conformity	0.018 to 0.022	Marginally acceptable conformity
above 0.015	Nonconformity	above 0.022	Nonconformity

13.7 Using external package for Benford Analytics

There is a package named `benford.analysis`, which provides tools that make it easier to validate data using Benford's Law. This package has been developed by **Carlos Cinelli**. As the package author himself states that the main purpose of the package is to identify suspicious data that need further verification, it should always be kept in mind that these analytics only provide us red-flagged transactions that should be validated further.

Apart from useful functions in the package, this also loads some default datasets specially those which were used by Frank Benford while proposing his law. Loading the library is pretty simple-

```
library(benford.analysis)
```

13.7.1 Creating a benford object

The main function `benford()` taken a data as input and creates an output of class benford. The syntax is

```
benford(data, number.of.digits=2)
```

where-

- `data` is numeric vector on which analysis has to be performed.
- `number.of.digits` is number of digits on which analysis has to be performed. Default value is `2`.

Let us apply this on `lakes.perimeter`[1] data which is available with the package.

```
# load sample data
data(lakes.perimeter)
lake_ben <- benford(lakes.perimeter$perimeter.km, number.of.digits = 2)
```

To view it, we can either -

- print it
- plot it

13.7.2 Printing Benford object

The print method first shows the general information of the analysis, like the name of the data used, the number of observations used and how many significant digits were analyzed. After that you have the main statistics of the log mantissa of the data. If the data follows Benford's Law, the numbers should be close to those shown in table following.

[1] A dataset of the perimeter of the lakes arround the water from the global lakes and wetlands database (GLWD) http://www.worldwildlife.org/pages/global-lakes-and-wetlands-database

Table 13.2: Ideal Statistics for data that follows Benford's Law

Statistic	Value
Mean	0.5
Variance	1/12 (0.08333...)
Ex. Kurtosis	-1.2
Skewness	0

The basic syntax is -

```
print(x, how.many=5, ...)
```

where -

- `x` is a benford object
- `how.many` is a number that defines how many of the biggest absolute differences to show.

```
print(lake_ben)
```

```
##
## Benford object:
##
## Data: lakes.perimeter$perimeter.km
## Number of observations used = 248607
## Number of obs. for second order = 3086
## First digits analysed = 2
##
## Mantissa:
##
##    Statistic  Value
##         Mean  0.595
##          Var  0.062
##  Ex.Kurtosis -0.147
##     Skewness -0.722
##
##
## The 5 largest deviations:
##
##   digits absolute.diff
## 1     15          4132
## 2     16          4025
## 3     14          4000
```

```
## 4     13          3999
## 5     17          3902
##
## Stats:
##
##  Pearson's Chi-squared test
##
## data:  lakes.perimeter$perimeter.km
## X-squared = 88111, df = 89, p-value <2e-16
##
##
##  Mantissa Arc Test
##
## data:  lakes.perimeter$perimeter.km
## L2 = 0.14, df = 2, p-value <2e-16
##
## Mean Absolute Deviation (MAD): 0.006013
## MAD Conformity - Nigrini (2012): Nonconformity
## Distortion Factor: -43.32
##
## Remember: Real data will never conform perfectly to Benford's Law. You should not fo
```

If we plot the object, it gives us 5 plots, as can be seen in following figure.

```
plot(lake_ben)
```

Further Reading-

1. ISACA JOURNAL ARCHIVES - Understanding and Applying Benford's Law - 1 May 2011

2. Newcomb, Simon. "Note on the Frequency of Use of the Different Digits in Natural Numbers." American Journal of Mathematics, vol. 4, no. 1, 1881, pp. 39–40. JSTOR, https://doi.org/10.2307/2369148. Accessed 15 Jun. 2022.

3. Durtschi, Cindy & Hillison, William & Pacini, Carl. (2004). The Effective Use of Benford's Law to Assist in Detecting Fraud in Accounting Data. J. Forensic Account.

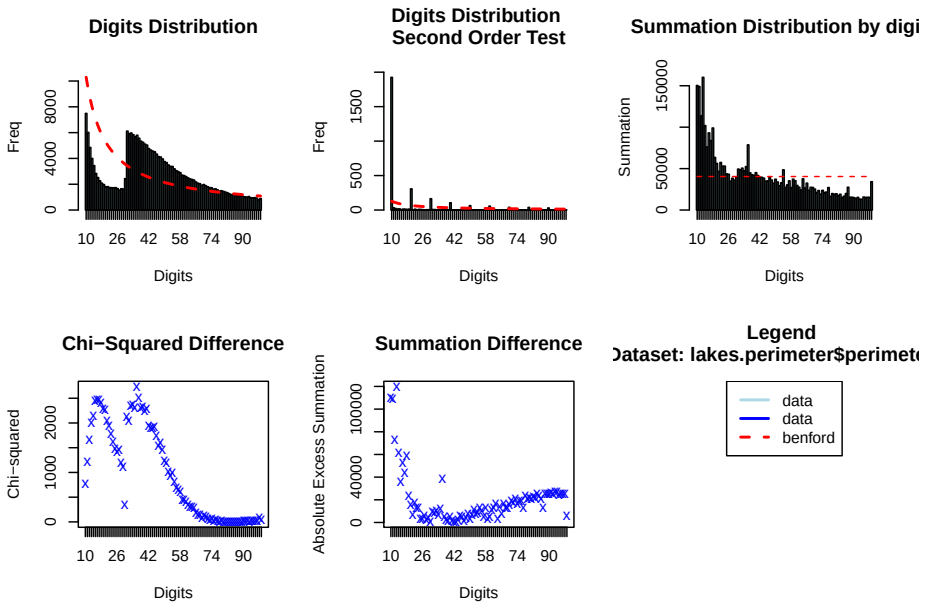


Figure 13.6: Plotting the diagnostic charts of lake perimeter data

Chapter 14

Finding duplicates

Prerequisites

Let us load `tidyverse` first.

```
library(tidyverse)
```

14.1 Simply duplicates!

Detection of duplicates is a frequent requirement in audit analytics, be it case of detecting risk of duplicate invoices or simply detection of duplicates in social sector audits[1].

Let us create an sample data of invoices.

```
set.seed(123)
# let's create 10 invoices
invoices <- data.frame(invoice_no = 1:10,
                       invoice_date = seq.Date(as.Date("2010-01-01"),
                                               as.Date("2010-01-10"),
                                               by = "day"),
                       amount = sample(1000:2000, 10))
invoices <- slice_sample(invoices,
                         n = 12,
                         replace = TRUE)
# let us also assume each has different description
```

[1] Identifying Fraud Using Abnormal Duplications Within Subsets. 2012. John Wiley & Sons, Ltd. https://doi.org/10.1002/9781118386798.ch12.

```
invoices$description <- LETTERS[1:12]
# view the data
invoices
```

```
##    invoice_no invoice_date amount description
## 1           4   2010-01-04   1525           A
## 2           6   2010-01-06   1937           B
## 3           9   2010-01-09   1298           C
## 4          10   2010-01-10   1228           D
## 5           5   2010-01-05   1194           E
## 6           3   2010-01-03   1178           F
## 7           9   2010-01-09   1298           G
## 8           9   2010-01-09   1298           H
## 9           9   2010-01-09   1298           I
## 10          3   2010-01-03   1178           J
## 11          8   2010-01-08   1117           K
## 12         10   2010-01-10   1228           L
```

To find duplicate records on the basis of multiple atrributes say `invoice_no`, `invoice_date` and `amount`, we can use function `duplicated` as shown below-

```
attrib_cols <- c("invoice_no", "invoice_date", "amount")
invoices %>%
  filter(if_all(all_of(attrib_cols), duplicated))
```

```
##   invoice_no invoice_date amount description
## 1          9   2010-01-09   1298           G
## 2          9   2010-01-09   1298           H
## 3          9   2010-01-09   1298           I
## 4          3   2010-01-03   1178           J
## 5         10   2010-01-10   1228           L
```

We may see that duplicate records have been thrown in result, which excludes first record considered as original.

> **Explanation:** Using `if_all` or `if_any` with `dplyr::filter` provides us a way to filter simultaneously based on many columns. The second argument to be provided in these functions is mostly a function which returns logical values. So use of `duplicate` in second argument filters all (because of `if_all`) columns provided in first argument based on values returned by applying function mentioned on second argument, on those columns.

If however, the requirement is to find all the records with duplicate attributes, we may change our strategy slightly.

```
invoices %>%
  select(all_of(attrib_cols)) %>%
  filter(if_all(everything(), duplicated)) %>%
  distinct() %>%
  semi_join(invoices, ., by = all_of(attrib_cols))
```

```
## Warning: Using `all_of()` outside of a selecting function was
## deprecated in tidyselect 1.2.0.
## i See details at
##   <https://tidyselect.r-lib.org/reference/faq-selection-context.html>
## This warning is displayed once every 8 hours.
## Call `lifecycle::last_lifecycle_warnings()` to see
## where this warning was generated.
```

```
##   invoice_no invoice_date amount description
## 1          9   2010-01-09   1298           C
## 2         10   2010-01-10   1228           D
## 3          3   2010-01-03   1178           F
## 4          9   2010-01-09   1298           G
## 5          9   2010-01-09   1298           H
## 6          9   2010-01-09   1298           I
## 7          3   2010-01-03   1178           J
## 8         10   2010-01-10   1228           L
```

> **Explanation:** Only columns of intrest (`attrib_cols`) have been selected in first step. Thereafter, `dplyr::filter` is applied on all of these columns. We will have all duplicate values as a result. In the last step we will filter invoices data using `semi_join`. One thing to note, we are filtering `invoices` on the basis of output of result of previous step and therefore `.` placeholder is specifically used after `invoices` i.e. first argument instead of default.

14.2 Finding network of duplicates - network analysis

Imagine a scenario when users may have multiple IDs such as mobile numbers, email ids, and say some other ID issued by a Government Department say Income Tax Department (e.g. PAN number in Indian Scenario). Using techniques mentioned in section 14.1, we may easily find out duplicate users, i.e. duplicates

on the basis of one ID. Sometimes need arise where we have to find out network of all the duplicate users where they have changed one or two IDs but retained another. E.g. There may be a social sector scheme where any beneficiary is expected to be registered only once for getting that scheme benefits. Scheme audit(s) may require auditors to check duplicate beneficiaries using multiple IDs.

Understand this with the figure 14.1

System Generated User ID	1	2	3	4
Name	A	B	C	B
Phone – ID1	11	12	13	13
Email – ID2	1a	1b	1b	2a
ID issued by Govt – ID3	AB	AB	BC	CD

Figure 14.1: An example case of availability of duplicates on network of multiple IDs

It may be seen that out of nine beneficiaries, two beneficiaries are using duplicate Phone numbers (ID1), thus there are seven unique users. Same is case with ID2 (Email) and ID3. However, if all the three IDs are collectively seen we may see that there are only three distinct beneficiaries and we have actually six duplicates. *Note that we are not considering names while finding out duplicates.*

We may find these duplicates using a branch of mathematics called *Graph Theory*.[2] We won't be discussing any core concepts of graph theory here. There are a few packages to work with graph theory concepts in R, and we will be using `igraph` (file., 2023) for our analysis here. Let's load the library.

```
library(igraph)
```

```
dat <- data.frame(
  MainID = 1:9,
  Name = c("A", "B", "C", "B", "E", "A", "F", "G", "H"),
  ID1 = c(11,12,13,13,14,15,16,17,17),
  ID2 = c("1a", "1b","1b", "2a", "2b", "2c", "2c", "2e", "3a"),
```

[2] https://en.wikipedia.org/wiki/Graph_theory

```
  ID3 = c("AB", "AB", "BC", "CD", "EF", "GH", "HI", "HI", "JK")
)
# A preview of our sample data
dat
```

```
##   MainID Name ID1 ID2 ID3
## 1      1    A  11  1a  AB
## 2      2    B  12  1b  AB
## 3      3    C  13  1b  BC
## 4      4    B  13  2a  CD
## 5      5    E  14  2b  EF
## 6      6    A  15  2c  GH
## 7      7    F  16  2c  HI
## 8      8    G  17  2e  HI
## 9      9    H  17  3a  JK
```

Now the complete algorithm is as under-

```
id_cols <- c("ID1", "ID2", "ID3")
dat %>%
  mutate(across(.cols = all_of(id_cols), as.character)) %>%
  pivot_longer(cols = all_of(id_cols),
               values_drop_na = TRUE) %>%
  select(MainID, value) %>%
  graph_from_data_frame() %>%
  components() %>%
  pluck(membership) %>%
  stack() %>%
  set_names(c('UNIQUE_ID', 'MainID')) %>%
  right_join(dat %>%
               mutate(MainID = as.factor(MainID)),
             by = c('MainID'))
```

```
##   UNIQUE_ID MainID Name ID1 ID2 ID3
## 1         1      1    A  11  1a  AB
## 2         1      2    B  12  1b  AB
## 3         1      3    C  13  1b  BC
## 4         1      4    B  13  2a  CD
## 5         2      5    E  14  2b  EF
## 6         3      6    A  15  2c  GH
## 7         3      7    F  16  2c  HI
## 8         3      8    G  17  2e  HI
## 9         3      9    H  17  3a  JK
```

We may see that we have got unique ID of users based on all three IDs. Let us understand the algorithm used step by step.

Step-1: First we have to ensure that all the ID columns (Store names of these columns in one vector say `id_cols`) must be of same type. Since we had a mix of character (Alphanumeric) and numeric IDs, using `dplyr::across` with `dplyr::mutate` we can convert all the three ID columns to character type. Readers may refer to section 1.4.1 for type change, and section **??** for changing data type of multiple columns simultaneously using `dplyr::across`.

Thus, first two lines of code above correspond to this step only.

```
id_cols <- c("ID1", "ID2", "ID3")
dat %>%
  mutate(across(.cols = id_cols, as.character))
```

Step-2: Pivot all id columns to longer format so that all Ids are linked with one main ID. Now two things should be kept in mind. One that there should be a main_Id column in the data frame. If not create one using `dplyr::row_number()` before pivoting. Secondly, if there are `NA`s in any of the IDs these have to be removed while pivoting. Use argument `values_drop_na = TRUE` inside the `tidyr::pivot_longer`. Thus, this step will correspond to this line-

```
pivot_longer(cols = all_of(id_cols), values_drop_na = TRUE)
```

where - first argument data is invisibly passed through dplyr pipe i.e. `%>%`. Upto this step, our data frame will look like -

```
## # A tibble: 27 x 4
##    MainID Name  name  value
##     <int> <chr> <chr> <chr>
##  1      1 A     ID1   11
##  2      1 A     ID2   1a
##  3      1 A     ID3   AB
##  4      2 B     ID1   12
##  5      2 B     ID2   1b
##  6      2 B     ID3   AB
##  7      3 C     ID1   13
##  8      3 C     ID2   1b
##  9      3 C     ID3   BC
## 10      4 B     ID1   13
## # i 17 more rows
```

Step-3: Now we need only two columns, one is `mainID` and another is `value` which is created by pivoting all ID columns. We will use `select(MainID, value)` for that.

Step-4: Thereafter we will create a graph object from this data (output after step-3), using `igraph` package. Interested readers may see how the graph object will look like, using `plot()` function. The output is shown in figure 14.2. **However, this step is entirely optional and it may also be kept in mind that graph output of large data will be highly cluttered and may not be comprehensible at all.**

```
dat %>%
  mutate(across(.cols = all_of(id_cols), as.character)) %>%
  pivot_longer(cols = all_of(id_cols),
               values_drop_na = TRUE) %>%
  select(MainID, value) %>%
  graph_from_data_frame() %>%
  plot()
```

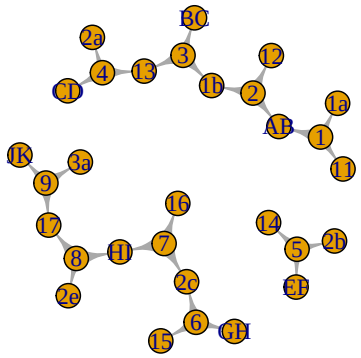


Figure 14.2: Plot of graph object

Step-5: This step will be a combination of three lines of codes which will number each ID based on connectivity of all components in the graph objects. Actually `components` will give us an object where `$membership` will give us `unique_ids` for each component in the graph.

```
## $membership
##  1  2  3  4  5  6  7  8  9 11 1a AB 12 1b 13 BC 2a CD
```

```
##  1  1  1  1  2  3  3  3  3  1  1  1  1  1  1  1  1  1
## 14 2b EF 15 2c GH 16 HI 17 2e 3a JK
##  2  2  2  3  3  3  3  3  3  3  3  3
##
## $csize
## [1] 13  4 13
##
## $no
## [1] 3
```

Next we have to `purrr::pluck`, `$membership` only from this object, which will return a named vector.

```
##  1  2  3  4  5  6  7  8  9 11 1a AB 12 1b 13 BC 2a CD
##  1  1  1  1  2  3  3  3  3  1  1  1  1  1  1  1  1  1
## 14 2b EF 15 2c GH 16 HI 17 2e 3a JK
##  2  2  2  3  3  3  3  3  3  3  3  3
```

We can then `stack` this named vector into a data frame using `stack` and `set_names`

```
##    UNIQUE_ID MainID
## 1          1      1
## 2          1      2
## 3          1      3
## 4          1      4
## 5          2      5
## 6          3      6
## 7          3      7
## 8          3      8
## 9          3      9
## 10         1     11
## 11         1     1a
## 12         1     AB
## 13         1     12
## 14         1     1b
## 15         1     13
## 16         1     BC
## 17         1     2a
## 18         1     CD
## 19         2     14
## 20         2     2b
## 21         2     EF
## 22         3     15
## 23         3     2c
```

```
## 24          3     GH
## 25          3     16
## 26          3     HI
## 27          3     17
## 28          3     2e
## 29          3     3a
## 30          3     JK
```

I suggest to purposefully name second column in the output data as `MainID` so that it can be joined with original data frame in the last step. `UNIQUE_ID` in this data will give us the new column which will allocate same ID to all possible duplicates in network of three IDs.

Step-6: In the last step we have to join the data frame back to original data frame. Since the type of `MainID` is now factor type, we can convert type of this column in original data frame before `right_join` the same. Hence the final step, `right_join(dat %>% mutate(MainID = as.factor(MainID)), by = c('MainID'))`.

Chapter 15

Detecting gaps in sequences

Audit analytics often requires us to check for gaps in sequences of numbers. Gaps in Sequentially numbered objects such as purchase orders, invoice numbers, cheque numbers, etc should be accounted for. Thus, auditors may require to exercise this audit check as a part of audit analytics.

15.1 When sequence numbers are available as `numeric` column

We will have starting and ending numbers in such sequence. Let us allocate these in two variables.

```
start_num <- 500301 # say
end_num <- 503500 # say
```

It means we have 3200 terms (say cheques) issued in the series. Further suppose, the cheque numbers issued are stored in some column say `cheque_no` in a given data frame, have a total count say 3177. To simulate

```
set.seed(123)
cheque_no <- sample(start_num:end_num, 3177, replace = FALSE)
```

To find out the gaps we may simply use function `setdiff` on these two.

```
setdiff(start_num:end_num, cheque_no)
```

```
##  [1] 500398 500445 500457 500716 500747 500862 501017
```

```
##  [8] 501018 501109 501333 501459 501609 501823 501908
## [15] 502160 502191 502609 502937 502974 503002 503284
## [22] 503385 503422
```

15.2 When sequence numbers are available as `character()` column

We may easily replicate above procedure for gap detection, even if the sequence column is of character type. E.g. If the cheque numbers have a prefix say, 'A', then the cheque numbers may look like-

```
##  [1] "A502763" "A502811" "A502527" "A500826" "A500495"
##  [6] "A503286" "A502142" "A501442" "A501553" "A501568"
```

In these case, we may first substitute prefix with nothing and then proceed as above.

```
modified_cheques <- sub("A", "", cheque_no) |> as.integer()
missing_cheques <- setdiff(start_num:end_num, modified_cheques)
missing_cheques
```

```
##  [1] 500398 500445 500457 500716 500747 500862 501017
##  [8] 501018 501109 501333 501459 501609 501823 501908
## [15] 502160 502191 502609 502937 502974 503002 503284
## [22] 503385 503422
```

Chapter 16

Merging large number of similar datasets into one

Data preparation for performing analytics is an important task and may require more time than actual analytics because we rarely have data in ideal format. Importing csv or flat files is rather an easy job. However, considering large popularity of MS Excel, we at times have our data saved in excel files.

Sometimes, one single data frame/table is divided into multiple sheets in one excel file whereas sometimes these tables are divided in multiple files. Here we are discussing few of these cases, where we can reduce our data preparation time by effectively writing the code for import of such data into our environment.

16.1 Case-1: Merging multiple excel sheets into one data frame

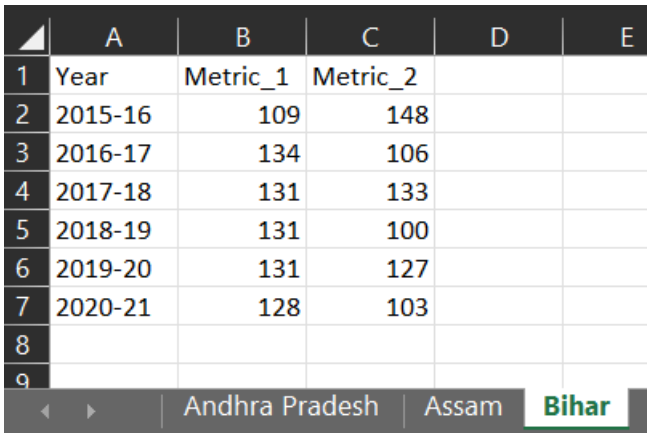
As an example, let's say we have multiple States' data saved on a different sheet in one excel file say `Daur.xlsx`. See preview in fig 16.1.

We will use library `readxl` to read excel files. This library is bundled with `tidyverse` but is not part of core tidyverse, so it has to be loaded explicitly, though explicit download is not required if `tidyverse` is installed in the system.

```
library(tidyverse)
library(readxl)
```

The following steps are used-

Step-1: Read path



	A	B	C	D	E
1	Year	Metric_1	Metric_2		
2	2015-16	109	148		
3	2016-17	134	106		
4	2017-18	131	133		
5	2018-19	131	100		
6	2019-20	131	127		
7	2020-21	128	103		
8					
9					

Andhra Pradesh | Assam | Bihar

Figure 16.1: Preview of example excel file

Step-2: Collect Names of all sheets

Step-3: Set names of elements of above vector onto itself

Step-4: Read and combine all tables into one. We will use `purrr::map_dfr` for this.

```
# Step-1
path <- "data/daur1.xlsx"
# Step-2
states_data <- excel_sheets(path) %>%
# step-3
    set_names() %>%
# step-4
    map_dfr(read_excel, path=path, .id = 'State_name')
# print file
states_data
```

```
## # A tibble: 18 x 4
##    State_name     Year    Metric_1 Metric_2
##    <chr>          <chr>      <dbl>    <dbl>
##  1 Andhra Pradesh 2015-16      119      121
##  2 Andhra Pradesh 2016-17      114      134
##  3 Andhra Pradesh 2017-18      115      122
##  4 Andhra Pradesh 2018-19      129      137
##  5 Andhra Pradesh 2019-20      149      129
##  6 Andhra Pradesh 2020-21      104      124
##  7 Assam          2015-16      104      145
##  8 Assam          2016-17      114      144
```

```
##  9 Assam          2017-18      116      116
## 10 Assam          2018-19      129      130
## 11 Assam          2019-20      144      134
## 12 Assam          2020-21      124      116
## 13 Bihar          2015-16      109      148
## 14 Bihar          2016-17      134      106
## 15 Bihar          2017-18      131      133
## 16 Bihar          2018-19      131      100
## 17 Bihar          2019-20      131      127
## 18 Bihar          2020-21      128      103
```

We can see that data from all the sheets have been merged into table and one extra column has been created using sheet name. The name of that column has been provided through `.id` argument. If the new column is not required, simply don't use this argument.

16.2 Case-2: Merging multiple files into one data frame

Often we have our source data split into multiple files, which we will have to compile in one single data frame before proceeding In this case, we may collect all such files in one directory and follow these steps-

Step-1: Store all file names using `list.files()`

Step-2: We may read all files in one list using either `lapply` or `purrr::map`.

Step-3: If data structures in all the files are same, we can directly use `purrr::map_dfr` which will read all files and give us a data frame. If however, the structure of data in all files are not same, we may convert all columns into character type before merging these files. We can thereafter proceed for merging all data using either `purrr::map_dfr` or `lapply` in combination with `do.call`.

16.3 Case-3: Split and save one data frame into multiple excel/csv files simultaneously.

As an example will use `states_data` created in case-1. We can use the following algorithm

Step-1: Create a vector of file names using `paste0` Step-2: Split data frame into a list with separate dataframe for each state Step-3: Write to a separate file using `purrr::walk2()`

The complete algorithm is

```
# step-1 : create a vector of file names (output)
file_names <- paste0("data/", unique(states_data$State_name), ".csv")

states_data %>%
  group_split(State_name) %>%
  purrr::walk2(file_names, write.csv)
```

We can check that 3 new files with state_names as filenames have been created in the `data` folder/directory as desired.

16.4 Case-4: Splitting one data into muliple files having multiple sheets

Sometimes, we may require to split a file not only into multiple files, but simultaeously require to split each file into multiple excel sheets. E.g. A data having States and districts is to be split into State-wise files having a separate sheet for each district.

This can be achieved using `writexl` library. In this case, we may write a custom function which can do our job easily.

```
library(tidyverse)
library(writexl)

book_and_sheets <- function(df, x, y){
  df_by_x <- df %>%
    split(.[[x]])

  save_to_excel <- function(a, b){
    a %>%
      split(.[[y]]) %>%
      writexl::write_xlsx(
        path = paste0("data/data_by_", b,"_",x, ".xlsx")
      )

  }

  imap(df_by_x, save_to_excel)
}
```

The function `book_and_sheets` designed in above code helps us to write a data say `df` into separate files based on column `x` and each of these files is further

divided into sheets based on column `y`. Only thing to be remembered is that we have to pass, `x` and `y` arguments as character strings; and `df` as variable.

Example - splitting `mtcars` into files based on `cyl` and sheets based on `gear`

```
book_and_sheets(mtcars, 'cyl', 'gear')
```

```
## $`4`
## [1] "G:\\OneDrive - A c GeM CAG of INDIA (1)\\Documents\\R-DA\\Data\\data_by_4_cyl.xlsx"
##
## $`6`
## [1] "G:\\OneDrive - A c GeM CAG of INDIA (1)\\Documents\\R-DA\\Data\\data_by_6_cyl.xlsx"
##
## $`8`
## [1] "G:\\OneDrive - A c GeM CAG of INDIA (1)\\Documents\\R-DA\\Data\\data_by_8_cyl.xlsx"
```

We can check that three excel files have been created in directory `data/`.

Chapter 17

Sentiment Analysis through Word-Cloud

17.1 Step-1:Prepare data and load libraries

As an example we will create a word cloud with Budget Speech made by Finance Minister during her Budget speech[1] 2022-23. All of the budget speech is available in file called `budget.txt`.

Load Libraries

```
library(tidyverse)
library(tidytext) #install.packages("tidytext")
library(wordcloud) #install.packages("wordcloud")
library(ggtext)
library(ggalt)
library(ggthemes)
library(ggpubr)
```

Load data

```
dat <- read.table('data/budget.txt', header = FALSE, fill = TRUE)
```

[1]Data Source: Indian Budget Portal

17.2 Step-2: Reshape the .txt data frame into one column

Above steps will create one row per line. Let's create a tidy data frame out of this data.

```
tidy_dat <- dat %>%
  pivot_longer(everything(), values_to = 'word', names_to = NULL)
```

17.3 Step-3: Tokenize the data/words

To tokenize the words we will use function `unnest_tokens()` from `tidytext` library. As a further step we will have a count of each word, using `dplyr::count` which will create a column `n` against each word.

```
tokens <- tidy_dat %>%
  unnest_tokens(word, word) %>%
  count(word, sort = TRUE)
```

17.4 Step-4: Clean stop words

The library `tidytext` has a default database which can eliminate stop words from above data. Let's load this default stop words data.

```
data("stop_words")
```

We may then remove stop words using `dplyr::anti_join`.

```
tokens_clean <- tokens %>%
  anti_join(stop_words, by='word') %>%
  # remove numbers
  filter(!str_detect(word, "^[0-9]"))
```

We may remove additional stop words those specific to this data/input. To have an idea of these stop words, we may at firt, skip this step altogether and proceed to generate word cloud in next step directly. After having a first look, we can identify and then remove these additional stop words seen in first round(s).

```
uni_sw <- data.frame(word = c("cent", "pm", "crore",
                              "lakh", "set",
                              "level", "sir"))

tokens_clean <- tokens_clean %>%
  anti_join(uni_sw, by = "word")
```

17.5 Step-5: Plot/generate word cloud

Output/Word cloud of following code can be seen in figure 17.1.

```
pal <- RColorBrewer::brewer.pal(8,"Dark2")

# plot the 40 most common words
tokens_clean %>%
  with(wordcloud(word,
                 n,
                 random.order = FALSE,
                 max.words = 40,
                 colors=pal,
                 scale=c(2.5, .5)))
```

Figure 17.1: Word Cloud of FM's Budget Speech 2022

Chapter 18

Finding string similarity

Comparison of two (or more) numeric fields is an easy job in the sense that we can use multiple statistical methods available to measure comparison between these. On the other hand, comparing strings in any way, shape or form is not a trivial task. Despite this complexity, comparing text strings is a common and fundamental task in many text-processing algorithms. Basic objective of all string similarity algorithms are to quantify the similarity between two text strings in terms of string metrics.

The fuzzy matching problems are to input two strings and return a score quantifying the likelihood that they are expressions of the same entity. So (`Geeta` and `Gita`) should get a high score but not (`Apple` and `Microsoft`). Over several decades, various algorithms for fuzzy string matching have emerged. They have varying strengths and weaknesses. These fall into two broad categories: `lexical matching` and `phonetic matching`.

18.1 Lexical matching

Lexical matching algorithms match two strings based on some model of errors. Typically they are meant to match strings that differ due to spelling or typing errors. Consider `Atharv` and `ahtarv`. A lexical matching algorithm would pick up that `ht` is a transposition of `th`. Such transposition errors are common. Given this, and that the rest of the two strings match exactly and are long enough, we should score this match as high.

Normally, algorithms to find lexical matching, can be classified into 'edit distance based' or 'token based'.

18.1.1 Levenshtein algorithm

It is named after *Vladimir Levenshtein*, who considered this distance in 1965. The `Levenshtein distance` between two words is the minimum number of single-character edits (i.e. insertions, deletions or substitutions) required to change one word into the other. Levenshtein distance may also be referred to as *edit distance*, although it may also denote a larger family of distance metrics. It is closely related to pairwise string alignments.

For the two words `helo` and `hello`, it is obvious that there is a missing character `"l"`. Thus to transform the word `helo` to `hello` all we need to do is insert that character. The distance, in this case, is `1` because there is only one edit needed.

18.1.2 Hamming distance

This distance is computed by overlaying one string over another and finding the places where the strings vary. Note, classical implementation was meant to handle strings of same length. Some implementations may bypass this by adding a padding at prefix or suffix. Nevertheless, the logic is to find the total number of places one string is different from the other.

18.1.3 Jaro-Winkler

This algorithms gives high scores to two strings if,

1. they contain same characters, but within a certain distance from one another, and
2. the order of the matching characters is same.

To be exact, the distance of finding similar character is 1 less than half of length of longest string. So if longest strings has length of 5, a character at the start of the string 1 must be found before or on ((5/2)–1) ~ 2nd position in the string 2 to be considered valid match. Because of this, the algorithm is directional and gives high score if matching is from the beginning of the strings.

18.1.4 Q-Gram

Q-Grams is based on the difference between occurences of `Q` consecutive characters in two strings. To illustrate take a case of Q=3 (this special case is also called trigrams). For `atharv` and its possible typo `ahtarv` the trigrams will be

- For atharv `{ath tha har arv}`
- for ahtarv `{aht hta tar arv}`

We can see that a total of `7` unique trigrams have been formed and out of these only `1` is similar. Thus, 3-gram similarility would be `1/7=14%`. We can see that this algorithm is not very effective for transpositions.

18.2 Phonetic matching

Phonetic matching algorithms match strings based on how similar they sound. Consider `Geeta` and `Gita.` They sound similar enough that one person might spell as `Geetha` or `Geeta`, another as `Gita.` As in this case, one is not necessarily a misspelling of the other. just sounds similar.

18.2.1 Soundex

Created by *Robert Russel* and *Margaret King Odell* in 1918, this algorithm intended to match names and surnames based on the basic rules of English pronunciation, hence, similar names get the same value.

18.2.2 Metaphone

Developed by *Lawrence Philips* in 1990, the Metaphone is also more accurate compared with the `Soundex` method as it takes into consideration the groups of letters. The disadvantage shows up when we apply it to reconcile the strings that are not in English, as it is based on the rules of English pronunciation.

18.2.3 Double Metaphone

Following `Metaphone`, *Philips* also designed the *Double Metaphone*. As its name suggests, it returns two codes, so you have more chances to match the items, however, at the same time, it means a higher probability of an error. According to the algorithm, there are three matching levels:

- `primary key to the primary key = strongest match`,
- `secondary key to the primary key = normal match`,
- `secondary key against the secondary key = weakest match`.

18.2.4 Metaphone 3

Philips further refined the double metaphone algorithm to produce better results. The algorithm (Metaphone 3) is however, proprietary and is not open-source.

18.3 Examples

In R, we can use `stringdist` package to calculate many of the above mentioned distances. The function is vectorised. The synatx is

```
stringdist(
  a,
  b,
  method = c("osa", "lv", "dl", "hamming", "lcs", "qgram", "cosine", "jaccard", "jw",
    "soundex"),
  useBytes = FALSE,
  weight = c(d = 1, i = 1, s = 1, t = 1),
  q = 1,
  p = 0,
  bt = 0,
  nthread = getOption("sd_num_thread")
)
```

where -

- `a` and `b` are two strings/vectors for which similarity/distance is to be measured.
- `method` to be used. Default is
  - `osa` for *Optimal String Alignment*. Other methods are-
  - `lv` for *Levenstein distance*,
  - `dl` for *Damerau-Levenshtein*
  - `hamming` for *Hamming distance*
  - `lcs` for *Longest Common Substring*
  - `qgram` for Q-Grams
  - `cosine` for cosine
  - `jaccard` for Jaccard's algorithm
  - `jw` for Jaro-Winkler
  - `soundex` for Soundex
- Other arguments are needed on the basis of algorithm chosen.

To calculate 'metaphone' index we can use `phonics` package and for 'Double Metaphone' we can use `PGRdup` package in R.

Example - Suppose we have a set of two names.

```
nameset1 <- c('Geeta', 'Susheel', 'Ram', 'Dr. Suchitra')
nameset2 <- c('Gita', 'Sushil', 'Rama', 'Suchitra')
```

Note most of these distances/similarity indices are cases sensitive, and therefore we have to use these methods with a bit cleaning first. We can convert cases of all strings to lower-case to eliminate these (if) unwanted errors.

```
library(stringdist)
```

```
##
## Attaching package: 'stringdist'
```

```
## The following object is masked from 'package:tidyr':
##
##     extract
```

```
## The following object is masked from 'package:magrittr':
##
##     extract
```

```
suppressPackageStartupMessages(library(dplyr))

data.frame(
  nameset1 = tolower(nameset1),
  nameset2 = tolower(nameset2)
) %>%
  mutate(lv_dist = stringdist(nameset1, nameset2, method = 'lv'),
         jw_dist = stringdist(nameset1, nameset2, method = 'jw'),
         qgram_3 = stringdist(nameset1, nameset2, method = 'qgram', q=3))
```

```
##       nameset1 nameset2 lv_dist jw_dist qgram_3
## 1        geeta     gita       2 0.21667       5
## 2      susheel   sushil       2 0.15079       5
## 3          ram     rama       1 0.08333       1
## 4 dr. suchitra suchitra       4 0.25694       4
```

Creating Metaphone and Double Metaphone

```
library(phonics)

data.frame(
  nameset1 = tolower(nameset1),
  nameset2 = tolower(nameset2)
) %>%
  mutate(metaphone_1 = metaphone(nameset1),
         metaphone_2 = metaphone(nameset2))
```

```
## Warning: There was 1 warning in `mutate()`.
## i In argument: `metaphone_1 = metaphone(nameset1)`.
## Caused by warning in `metaphone()`:
## ! unknown characters found, results may not be consistent
```

```
##       nameset1 nameset2 metaphone_1 metaphone_2
## 1        geeta     gita          JT          JT
## 2      susheel   sushil         SXL         SXL
## 3          ram     rama          RM          RM
## 4 dr. suchitra suchitra        <NA>        SXTR
```

Note that we cannot calculate `metaphone` for special characters even for spaces.

Double metaphone is not vectorised. So we have to use `apply` family of functions here.

```
suppressPackageStartupMessages(library(PGRdup))
library(purrr)

data.frame(
  nameset1 = tolower(nameset1),
  nameset2 = tolower(nameset2)
) %>%
  mutate(DMP_1_1 = map_chr(nameset1, ~DoubleMetaphone(.x)[[1]]),
         DMP_1_2 = map_chr(nameset1, ~DoubleMetaphone(.x)[[2]]),
         DMP_2_1 = map_chr(nameset2, ~DoubleMetaphone(.x)[[1]]),
         DMP_2_2 = map_chr(nameset2, ~DoubleMetaphone(.x)[[2]]))
```

```
##       nameset1 nameset2 DMP_1_1 DMP_1_2 DMP_2_1
## 1        geeta     gita      JT      KT      JT
## 2      susheel   sushil     SXL     SXL     SXL
## 3          ram     rama      RM      RM      RM
## 4 dr. suchitra suchitra    TRSX    TRSK    SXTR
##   DMP_2_2
## 1      KT
## 2     SXL
## 3      RM
## 4    SKTR
```

Appendix A

File handling operations in R

In chapter 7 we have already learned about reading and writing data from/to files. In this section, we will learn about some other functions that are useful while reading and writing data, such as - changing directory, creating a file, renaming a file, check the existence of the file, listing all files in the working directory, copying files and creating directories.

A.1 Handling files

A.1.1 Creating a file within R, using `file.create()`

Using `file.create()` function, we can create a new file from console. If the file already exists it truncates. The function returns a `TRUE` logical value if file is created otherwise, returns `FALSE`.

Example - The following command will create a blank text file in the current working directory.

```
file.create("my_new_text_file.txt")
```

```
## [1] TRUE
```

A.1.2 Checking whether a file exists, using `file.exists()`

Similar to above, we can check whether a file with given name exists, using function `file.exists()`. Example-

```
file.exists("my_new_text_file.txt")
```

```
## [1] TRUE
```

A.1.3 Renaming file with `file.rename()`

The file name can be changed within the R console using, function `file.rename()`. Basic syntax is `file.rename(from = "old_name", to = "new_name")`. The function will return `TRUE` or `FALSE` depending upon the successful execution. See example

```
file.rename(from = "my_new_text_file.txt", to = "my_renamed_file.csv")
```

```
## [1] TRUE
```

```
# Check whether old file exists
file.exists("my_new_text_file.txt")
```

```
## [1] FALSE
```

A.1.4 Copying file with `file.copy()` function

Using `file.copy(from = "old_path", to = "new_path")` syntax files can be copied from one directory to another.

A.1.5 Deleting file with `file.remove()`

The syntax for function, that removes a file with given name, is also very simple. Example-

```
file.remove("my_renamed_file.csv")
```

```
## [1] TRUE
```

Check whether the file has been really deleted.

```
file.exists("my_renamed_file.csv")
```

```
## [1] FALSE
```

A.2 Handling directories

A.2.1 Get/Set path of current working directory using `getwd()`/ `setwd()`

We can check/get the path of current working directory (wd in short) as a character vector, using `getwd()` function.

```
getwd()
```

```
## [1] "G:/OneDrive - A c GeM CAG of INDIA (1)/Documents/R-DA"
```

Similarly, using `setwd("given\\path\\here")` we can change the current working directory.

> Two things to be noted here - Either the path is to be given using forward slash `/` or if backslash `\` is used these need to be escaped, using an extra `\` as `\` is itself an escape character in R.

A.2.2 Create new directory using `dir.create()` and other operations

A new directory can be created using function `dir.create()`. Example- the command below will create a new directory named 'new_dir' in the current working directory. If `TRUE` is returned, directory with given name is created.

```
dir.create("new_dir")
```

We can check whether any directory named 'new_dir' exists in current working directory, using function `dir.exists()` function. Function will return either `TRUE` or `FALSE`.

```
dir.exists("new_dir")
```

```
## [1] TRUE
```

We can also check all files that exists in current working directory/any other directory using `list.files()` function.

```
any(list.files() == 'new_dir')
```

```
## [1] TRUE
```

A given directory can be removed using `unlink()` function be specifically setting argument `recursive` to `TRUE`. Example

```
unlink("new_dir", recursive = TRUE)
dir.exists("new_dir")
```

```
## [1] FALSE
```

A.3 An important function for opening a dialog box for selecting files and folder

We may use either of `choose.dir()` or `file.choose()`, to let the user select directory or file of her/his choice respectively.

Try these in your console

```
list.files(choose.dir())
file.copy(from = file.choose(), to = "new_name")
```

A.4 Other useful functions for listing/removing variables

We can list all the variables available in current environment using function `ls()`. Another function `rm()` will remove the given variables. So a command like `rm(lm())` will remove all the available variables from the environment(Use this will caution as it will erase all the saved variables/data).

A.5 Using `save()` to save objects/collection of objects

We can save objects using function `save()` which saves the objects on disk for later usage. The saved objects can be retrieved using `load()` function. See this example-

```
h <- hist(Nile)
```

```
save(h, file="nile_hist")
rm(h)
any(ls() == 'h')
```

```
## [1] FALSE
```

```
load("nile_hist")
any(ls() == 'h')
```

```
## [1] TRUE
```

A.6 Projects

While working on a project, often a requirement is to keep all scripts, data, results, charts, figures, etc. at a single place. R studio has thus, a concept of working in projects, which associates a specific directory with a specific project and creates a specific file with extension `.Rproj`, which can reopen the complete scripts/ data/ etc. associated with that project.

To open a new project in Rstudio, click `file` menu then `New Project`. Check the following screenshots-

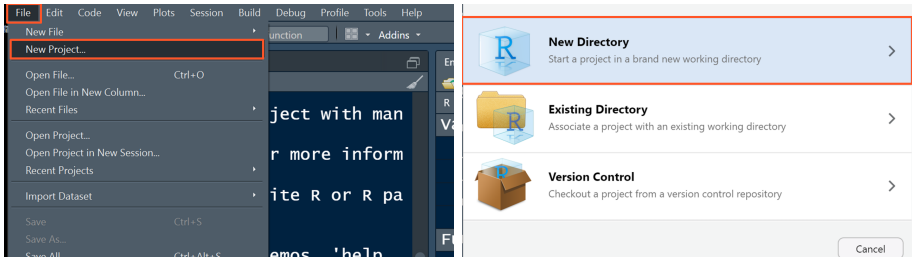


Figure A.1: Creating Projects in R Studio

After Creating the projects you will notice that a file with an extension `.Rproj` has been created by R Studio in the selected directory/location of project.

To resume working in the same project directory, either double click the file, or open the project using file menu i.e. `file` $->$ `Open Project`.

Appendix B

Regex - A quick introduction

A **Regular Expression**, or **regex** for short, is a powerful tool, which helps us writing code for pattern matching in texts. Regex, is a pattern that describes a set of strings. It is a sequence of characters that define a search pattern. It is used to search for and manipulate text. Regex can be used in many programming languages, including R.

Regex patterns are made up of a combination of regular characters and special characters. Regular characters include letters, digits, and punctuation marks. Special characters have a specific meaning in regex and are used to represent patterns of characters.

Regex patterns can be used for a variety of purposes, including:

- Searching for specific strings in text
- Extracting specific parts of a string
- Replacing parts of a string with other text
- Validating input from users

In R, we can use the `grep` and `gsub` functions to search for and manipulate text using regex.

B.1 Basic Regex- Literal Characters

Every *literal character*, in itself is a regex that matches itself. Thus, `a` matches third character in text `Charles`. These literal characters are case sensitive. Example-1

```
ex_text <- "This is an example text"
# Match literal `x`
grepl(pattern = "x", ex_text)
```

```
## [1] TRUE
```

```
# Match Upper case literal "X"
grepl("X", ex_text)
```

```
## [1] FALSE
```

B.2 Metacharacters

B.2.1 Character sets

It is always not feasible to put every literal characters. We may also match literal characters from a given set of options. To **match a group of characters** we have to put all these in square brackets. So, `[abc]` matches either of `a`, `b`, or `c`. Example-

```
ex_vec <- c("Apple", "Orange", "Myrrh")
# matches a vowel
grepl("[aeiou]", ex_vec)
```

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

To **match a range of characters/numbers** we can separate these by hyphen in square brackets. So, `[a-n]` will match a character from range `[abcdefghijklmn]`. Example-

```
ex_text <- "The quick brown fox jumps over the lazy dog"
grepl("[a-z]", ex_text)
```

```
## [1] TRUE
```

```
grepl("[X-Z]", ex_text)
```

```
## [1] FALSE
```

Example-2

```
ex_colors <- c("grey", "black", "gray")
grepl("gr[ae]y", ex_colors)
```

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

We can also use **pre-built character classes** listed below.

- `[:punct:]` punctuation.
- `[:alpha:]` letters.
- `[:lower:]` lowercase letters.
- `[:upper:]` uppercase letters.
- `[:digit:]` digits.
- `[:xdigit:]` hex digits.
- `[:alnum:]` letters and numbers.
- `[:cntrl:]` control characters.
- `[:graph:]` letters, numbers, and punctuation.
- `[:print:]` letters, numbers, punctuation, and white-space.
- `[:space:]` space characters (basically equivalent to `\\s`).
- `[:blank:]` space and tab.

Example-

```
ex_vec2 <- c("One apple", "2 Oranges")
grepl("[:digit:]", ex_vec2)
```

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

B.2.2 Non-printable characters/ Meta characters (shorthand character classes)

We can use special character sequences to put non-printable characters in our regular expression(s). E.g. `\t` matches a tab character. **But since \ is an escape character in R, we need to escape it too.** So to match a tab character we have to put `\\t` in our regex sequence. Regex for that matches new line (line feed) is `\\n`. `Regex` for other meta characters is listed below-

- `\\s` matches a white-space character. Moreover, its complement `\\S` matches any character except a white-space.
- `\\w` matches any alphanumeric character. Similarly, its complement is `\\W` which matches any character except alphanumeric charachters.
- `\\d` matches any digit. Similarly, its complement is `\\D` which matches any character except digits.

- `\\b` matches any word boundary. Thus, `\\B` matches any character except a word boundary.
- `.` matches any character. To match a literal dot `.` we have to escape that; and thus `\\.` matches a dot charcter.

See these examples-

```
ex_vec3 <- c("One apple", "2 oranges & 3 bananas.")
# match word boundary
grepl("\\w", ex_vec3)
```

```
## [1] TRUE TRUE
```

```
# match any character followed by a dot character
grepl(".\\.", ex_vec3)
```

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

B.3 Quantifiers

What if we want to match more than one literal/character through `regex`? Let's say if we want to check whether the given string or string vector contain two consecutive vowels. One method may be to use character classes two times i.e. using `[aeiou][aeiou]`. But this method is against the principles of **DRY**[1] which is one of the common principle of programming. To solve these issues, we have quantifiers.

- `+` **1 or more** occurrences
- `*` **0 or more**
- `?` **0 or 1**
- `{}` specified numbers
  - `{n}` exactly n
  - `{n,}` n or more
  - `{n,m}` between n and m

Thus, we may match two consecutive vowels using `[aeiou]{2}`. See this example

[1] Dont repeat yourself

```
ex_vec <- c("Apple", "Banana", "pineapple")
grepl("[aeiou]{2}", ex_vec)
```

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

B.4 Alternation

Alternation in regular expressions allows you to match one pattern or another, depending on which one appears first in the input string. The pipe symbol `|` is used to separate the alternative patterns.

B.4.0.0.1 Basic Alternation Let's start with a basic example to illustrate how alternation works:

```
library(stringr)

string <- "I have an apple and a banana"
pattern <- "apple|banana"

str_extract_all(string, pattern)
```

```
## [[1]]
## [1] "apple"  "banana"
```

B.4.0.0.2 Order of Precedence When using alternation, it's important to keep in mind the order of precedence rules. In general, the first pattern that matches the input string will be selected, and subsequent patterns will not be considered. Here's an example to illustrate this:

```
string <- "I have a pineapple and an apple"
pattern <- "apple|pineapple"

str_extract(string, pattern)
```

```
## [1] "pineapple"
```

In this example, we have a string `string` that contains the words "apple" and "pineapple". We want to extract the first occurrence of either "apple" or "pineapple" from this text using a regular expression pattern that utilizes alternation. The pattern `apple|pineapple` means "match 'apple' OR 'pineapple' ". However, since the input string contains "pineapple" before "apple", the `str_extract()` function selects the first matching string "pineapple".

B.4.0.0.3 Grouping Alternatives We can also use parentheses to group alternative patterns together. This can be useful for specifying more complex patterns. Example:

```
string <- "Apple and pineapples are good for health"
pattern <- "(apple|banana|cherry) (and|or) (pineapple|kiwi|mango)"

str_extract(string, regex(pattern, ignore_case = TRUE))
```

```
## [1] "Apple and pineapple"
```

```
# Returns "apple and pineapple"
```

In above example, we have used `stringr::regex()` to modify regex flag to ignore cases while matching.

B.5 Anchors

Anchors in regular expressions allow you to match patterns at specific positions within the input string. In R, you can use various anchors in your regular expressions to match the beginning, end, or specific positions within the input text.

B.5.1 Beginning and End Anchors

The beginning anchor `^` and end anchor `$` are used to match patterns at the beginning or end of the input string, respectively. Example

```
string <- "The quick brown fox jumps over the lazy dog. The fox is brown."
pattern <- "^the"
str_extract_all(string, regex(pattern, ignore_case = TRUE))
```

```
## [[1]]
## [1] "The"
```

In the above example, if we want to extract word `the` which is at the beginning of a sentence only, we can use this regex.

B.5.2 Word Boundary Anchors

The word boundary anchor `\\b` is used to match patterns at the beginning or end of a word within the input string. Example

```
string <- 'Apple and pineapple, both are good for health'
pattern <- '\\bapple\\b'
str_extract_all(string, regex(pattern, ignore_case = TRUE))
```

```
## [[1]]
## [1] "Apple"
```

In the above example, though `apple` string is contained in another word `pineapple` we are limiting our search for whole words only.

B.6 Capture Groups

A capture group is a way to group a part of a regular expression and capture it as a separate substring. This can be useful when you want to extract or replace a specific part of a string. In R, capture groups are denoted by parentheses `()`. Anything inside the parentheses is captured and can be referenced later in the regular expression or in the replacement string.

One use of capturing group is to refer back to it within a match with back reference: `\1` refers to the match contained in the first parenthesis, `\2` in the second parenthesis, and so on.

Example-1

```
my_fruits <- c('apple', 'banana', 'coconut', 'berry', 'cucumber', 'date')
# search for repeated alphabet
pattern <- '(.)\\1'
grep(pattern, my_fruits, perl = TRUE, value = TRUE)
```

```
## [1] "apple" "berry"
```

Example-2

```
# search for repeated pair of alphabets
pattern <- '(..)\\1'
grep(pattern, my_fruits, perl = TRUE, value = TRUE)
```

```
## [1] "banana"   "coconut"  "cucumber"
```

Another way to use capturing group is, when we want to replace the pattern with something else. It is better to understand this with the following example-

```
# We have names in last_name, first_name format
names <- c('Hanks, Tom', 'Affleck, Ben', 'Damon, Matt')
# Using this regex, we can convert these to first_name last_name format

gsub('(\\w+),\\s+(\\w+)', '\\2 \\1', names, perl = TRUE)
```

```
## [1] "Tom Hanks"   "Ben Affleck" "Matt Damon"
```

B.7 Lookarounds

Lookahead and **lookbehinds** are zero-width assertions in regex. They are used to match a pattern only if it is followed or preceded by another pattern, respectively. The pattern in the lookahead or lookbehind is not included in the match.

B.7.1 Lookahead

A lookahead is used to match a pattern only if it is followed by another pattern. *Positive Lookaheads* are written as `(?=...)`, where `...` is the pattern that must follow the match.

For example, the regex pattern `hello(?= world)` matches "hello" only if it is followed by " world". It matches "hello world" but not "hello there world" or "hello".

Example

```
string <- c("hello world", "hello there world", "hello")
grep("hello(?= world)", string, value = TRUE, perl = TRUE)
```

```
## [1] "hello world"
```

B.7.2 Lookbehind

A lookbehind is used to match a pattern only if it is preceded by another pattern. Lookbehinds are written as `(?<=...)`, where `...` is the pattern that must precede the match.

For example, the regex pattern `(?<=hello )world` matches "world" only if it is preceded by "hello". It matches "hello world" but not "world hello" or "hello there world".

Example

```
string <- c("hello world", "world hello", "hello there world")
grep("(?<=hello )world", string, value = TRUE, perl = TRUE)
```

```
## [1] "hello world"
```

B.7.3 Negative Lookahead and Lookbehinds

Negative lookahead and negative lookbehinds are used to match a pattern only if it is not followed or preceded by another pattern, respectively. Negative lookahead and lookbehinds are written as `(?!...)` and `(?<!...)`, respectively.

For example, the regex pattern `hello(?! world)` matches "hello" only if it is not followed by " world". It matches "hello there" but not "hello world" or "hello world there".

Example-

```
string <- c("hello there", "hello world", "hello world there")
grep("hello(?! world)", string, value = TRUE, perl = TRUE)
```

```
## [1] "hello there"
```

And the regex pattern `(?<!hello )world` matches "world" only if it is not preceded by "hello". It matches "world hello" and "hello there world" but not "hello world".

```
string <- c("hello world", "world hello", "hello there world")
grep("(?<!hello )world", string, value = TRUE, perl = TRUE)
```

```
## [1] "world hello"       "hello there world"
```

While the difference between the lookahead and lookbehind may be subtle, yet these become clear when string/pattern replacment or extraction is required.
Example-

```
library(stringr)

string <- "I have 10 apples and 5 bananas"

pattern1 <- "(?<=\\d\\s)apples"  # lookbehind to match "apples" preceded by a digit and a space
pattern2 <- "\\d+(?=\\sbanana)"  # lookahead to match count of bananas

str_extract(string = string, pattern = pattern1)
```

```
## [1] "apples"
```

```
str_extract(string = string, pattern = pattern2)
```

```
## [1] "5"
```

B.8 Comments

B.8.1 Comments within regex

We can use the # character to add comments within a regex pattern. Any text following the `#` symbol on a line is ignored by the regex engine and treated as a comment. This can be useful for documenting your regex patterns or temporarily disabling parts of a pattern for testing or debugging. Example-

```
grep( "x(?#this is a comment)", c("xyz","abc"), perl = TRUE, value = TRUE)
```

```
## [1] "xyz"
```

B.8.2 Verbose Mode (multi-line comments)

In regular expressions, verbose mode is a feature that allows you to write more readable and maintainable regex patterns by adding comments and whitespace without affecting their behavior. To enable verbose mode, we can use the `(?x)` or `(?verbose)` modifier at the beginning of your regex pattern.

Example - Using this regex we can extract words that contain a vowel at third place.

```
string <- "The quick brown fox jumps over the lazy dog"
pattern <- "(?x)      # Enable verbose mode
            \\b       # Match word boundary
            \\w{2}    # matches first two alphabets
            [aeiou]   # Match a vowel
            \\w*      # Match optional word characters
            \\b       # Match word boundary"
str_extract_all(string, pattern)
```

```
## [[1]]
## [1] "The"   "quick" "brown" "over"  "the"
```

Appendix C

List of geoms available in `ggplot2`

Table: List of GEOMS available in `ggplot2` package

Topic	Title
geom_abline	Reference lines: horizontal, vertical, and diagonal
geom_bar	Bar charts
geom_bin_2d	Heatmap of 2d bin counts
geom_blank	Draw nothing
geom_boxplot	A box and whiskers plot (in the style of Tukey)
geom_contour	2D contours of a 3D surface
geom_count	Count overlapping points
geom_density	Smoothed density estimates
geom_density_2d	Contours of a 2D density estimate
geom_dotplot	Dot plot
geom_errorbarh	Horizontal error bars
geom_function	Draw a function as a continuous curve
geom_hex	Hexagonal heatmap of 2d bin counts
geom_freqpoly	Histograms and frequency polygons
geom_jitter	Jittered points
geom_crossbar	Vertical intervals: lines, crossbars & errorbars
geom_map	Polygons from a reference map
geom_path	Connect observations
geom_point	Points
geom_polygon	Polygons
geom_qq_line	A quantile-quantile plot
geom_quantile	Quantile regression
geom_ribbon	Ribbons and area plots
geom_rug	Rug plots in the margins
geom_segment	Line segments and curves
geom_smooth	Smoothed conditional means
geom_spoke	Line segments parameterised by location, direction and distance
geom_label	Text
geom_raster	Rectangles
geom_violin	Violin plot
CoordSf	Visualise sf objects
update_geom_defaults	Modify geom/stat aesthetic defaults for future plots

Bibliography

Bache, S. M. and Wickham, H. (2022). *magrittr: A Forward-Pipe Operator for R*. R package version 2.0.3.

Cinelli, C. (2018). *benford.analysis: Benford Analysis for Data Validation and Forensic Analytics*. R package version 0.1.5.

file., S. A. (2023). *igraph: Network Analysis and Visualization*. R package version 1.4.1.

R Core Team (2022). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria.

Wickham, H. (2016). *ggplot2: Elegant Graphics for Data Analysis*. Springer-Verlag New York.

Wickham, H., Chang, W., Henry, L., Pedersen, T. L., Takahashi, K., Wilke, C., Woo, K., Yutani, H., and Dunnington, D. (2023a). *ggplot2: Create Elegant Data Visualisations Using the Grammar of Graphics*. R package version 3.4.1.

Wickham, H., François, R., Henry, L., Müller, K., and Vaughan, D. (2023b). *dplyr: A Grammar of Data Manipulation*. R package version 1.1.1.

Wickham, H., Hester, J., and Bryan, J. (2023c). *readr: Read Rectangular Text Data*. R package version 2.1.4.

Wickham, H., Vaughan, D., and Girlich, M. (2023d). *tidyr: Tidy Messy Data*. R package version 1.3.0.

Xie, Y. (2015). *Dynamic Documents with R and knitr*. Chapman and Hall/CRC, Boca Raton, Florida, 2nd edition. ISBN 978-1498716963.

Xie, Y. (2023). *bookdown: Authoring Books and Technical Documents with R Markdown*. R package version 0.33.