R Best Practice

Chad Goymer

2018-08-15

Contents

Chapter 1

Introduction

This document sets out the proposals for best practice when using R. The aim is to provide a consistent and comprehensive approach to using R for analysis and applications. It forms part of the wider guidance on end-user computing. The document is split into the following areas:

1. **Software**: Details the recommended software to install, both the core R applications and supporting software.
2. **Writing**: Gives guidance on on how to write R scripts and applications.
3. **Packages**: Lists the recommended packages to use for achieving common tasks.
4. **Development**: Describes the recommended approaches to developing in R.

This document is not supposed to be a comprehensive guide to using R. It gives guidance and sign-posts users to resources with more details.

1.1 End-User Computing

The Bank of England's 2016 report *Solvency II: internal model approval process data review findings* identifies end-user computing as a Solvency II risk. In particular, it states:

> Spreadsheets and other user-developed applications are a form of information technology, and all information technology needs to be appropriately controlled.

End-user computing requires a user to think about the following areas when developing in R:

- Risk analysis
- Documentation
- Testing
- Reproducibility
- Change control
- Access control

The result of the risk analysis determines the level at which the subsequent points need to be addressed.

1.2 Risk Analysis

The level of documentation, testing and change control depends on two factors:

1. Complexity
2. Criticality

Complexity: If a piece of work is very simple it may only require a short description, a manual test and saving to a secure location. However, if it is highly complex it is important for others, and your future self, to document it thoroughly, provide repeatable tests and save the scripts in version control.

Criticality: If a piece of work is a simple analysis for curiosity it may not need any controls applied to it. However, if it provides functionality which other systems rely on or informs major business decisions then it is important to ensure it is thoroughly tested and reviewed and changes are not made without careful consideration of how it affects dependents.

In order to assess the level of controls we perform a simple risk analysis by estimating the level of complexity and criticality of the work, using a scale of **high**, **medium** or **low**. If both are low then no further consideration is required. However, If any of the above factors are medium or high then the work is considered material and should be subject to the following controls.

1.3 Documentation

As a minimum the following should be documented with the code. For simple scripts, a commented header in the file is sufficient. For multi-file applications it is recommended that a README file is added.

- **Name:** The name of the project, analysis or application.
- **Description:** A brief description of of what the code does (not how).
- **Purpose:** The reason for writing the code.
- **Instructions:** How to use the code - for the benefit of other users.

For **high complexity** work the following should be considered

- Write documentation for each section/function of code separately .
- Provide examples of usage.

For **high criticality** work consider creating an R package then:

- Create a DESCRIPTION file to document the package.
- Use the package `Roxygen2` to document functions.
- Write a vignette explaining its use.

1.4 Testing

As a minimum, manual tests should be performed and the results recorded alongside the code. This may take the following forms:

- Assertions in the code to ensure reasonableness, e.g. result is within a range.
- Comparisons with known results.

For **high complexity** work consider the following:

- Separate tests for each section/function of code.
- Comparisons with simple inputs and expected results

For **high criticality** work consider creating an R package then:

- Use the package `testthat` to write executable tests.
- Execute regression tests whenever changes are made.
- Perform integration tests with dependent systems.

1.5 Reproducibility

Important work should reproducible by others, and your future self. This means keeping track of inputs, parameters and code used to produce results. The simplest approach is to keep a copy of input data and parameters alongside the the code used to produce the saved results. Then it is self-contained.

Also consider using R markdown which captures description, code and results in a single document.

For **high complexity** work consider creating an RStudio project it has the following benefits:

- Sets the current working directory so relative file paths can be used.
- Saves open files so you can start where you left off.

Some other recommendations worth considering are:

- Avoid referencing external files if possible.
- Make dependencies clear.
- Cache versions of input data into a subfolder, if not large.

For **high criticality** work also consider

- Outputting audit logs with time stamp, username and computing environment
- Holding inputs in version control

1.6 Change control

Change control is about managing updates and bug fixes to the code in a way that is tracked, reviewed, and approved. It ensures change, and its implications, is understood and can also be reversed, if necessary. The simplest approach is to produce new files for each version, keeping older versions elsewhere. Each new version should document the reason for the change.

A much better solution is to use version control software to track changes. This makes the process much more robust. It is recommended that Git is used for version control and shared using a hosting application such as GitHub (or TFS if not available).

For **high complexity** and/or **criticality** consider the following:

- Track changes in the hosting application (GitHub or TFS).
- Record description and reason for change and prioritise.
- Create a new branch for each change.
- Merge back into the main branch only after a review.
- Require approval from an appropriate person or group before releasing to production.

1.7 Access control

There should be appropriate controls on who has access to the source code and/or results. The developers should be aware of who has access the work they are producing. The simplest approach is to save files into a location with adequate access controlled by the IT dept. It may also be appropriate to limit who has the ability to change the source code separately.

For **high complexity** work it is important only those with the appropriate level of knowledge can change the code.

For **high criticality** work production code should not be changable directly. A release process should be set up so appropriate approval must be given before code is promoted.

In both cases:

- The list of people with access to the source code and results should be reviewed on a regular basis.
- Appropriate back up and recovery strategies should be implemented to ensure rollback and re-installation is possible.

1.8 Appendix: Solvency II: internal model approval process data review findings

1.8.1 Sub-risk 5: IT environment, technology and tools

> Spreadsheets and other user-developed applications are a form of information technology, and all information technology needs to be appropriately controlled.

Finding 9: end-user computing (EUC)

4.36 Spreadsheets and other end-user applications (2012.4.39) remained common in capital and balance sheet modelling. The PRA does not have a view on whether end-user computing (EUC) is appropriate, as it is a form of IT, and all IT needs to be appropriately controlled. Where EUC is material to the internal model data flow, the PRA will be looking for appropriate controls for data quality such as reasonableness checks, input validations, peer reviews, systems environment configuration, logical access management, ongoing change controls (development, build , systems and user acceptance testing) and release management (including implementation and operational testing), disaster recovery, and documentation.

4.37 Automation of spreadsheets reduces the risk of manual error (2012.4.42), but can introduce different problems such as reduced oversight, inadequate transparency about the extent of linking and the proliferation of nested spreadsheets and the attendant issue of 'broken links'.

4.38 The 2012 report did not engage comprehensively with cyber risk. This is likely to be an area of increasing focus, following alerts and increasing concerns about security as firms move away from localised application and onto networked platforms. As noted in the Bank of England Financial Stability Report,1 cyber attacks can threaten financial stability by disrupting the provision of critical functions from the financial system to the real economy. The Financial Policy Committee has recommended that resilience testing be a regular part of core firms' cyber resilience assessment. Insurers providing cover for cyber or business interruption are also indirectly exposed to cyber risk.

Finding 10: IT infrastructure

4.39 Complex IT implementations (2012.4.44) can be challenging to manage without a clear definition of user requirements, design, testing and appropriate controls for effective operation in business as usual. This continued to be an area of risk. One firm took seven years to implement a tactical system, and still has no strategic system for its upstream administration processes.

Chapter 2

Software

When developing R code which may be shared or executed by another person it is important to ensure everyone is using the same versions of software and packages.

2.1 Core R Applications

The minimum software required to write R code is the R application itself. However, it is better to use a separate editor for creating R scripts. The recommended software installtion is:

- Microsoft R Open, which is 100% compatible with the standard version of R and provides better performance in some cases. Like the standard version of R, Microsoft's version is free.
- RStudio Desktop, to edit R script and build projects. RStudio can also be used for free.
- RTools, which is required to build R packages. RTools is free to download.

RStudio provides a cheatsheet detailing commonly used functionality of its editor:

2.2 Supporting Software

In addition to the above, when writing R code of high complexity or criticality it is recommended that version control is used:

- Git is the market leader and integrates well with RStudio. Git is free
- GitKraken provides a very good graphical interface to Git and access to more features than RStudio. GitKraken is not free, but it does not cost very much.

2.3 Package Versions

A common problem with R is ensuring everyone is using the same version of packages. The are a number of ways to deal with this problem:

1. Another approach is to create a site library, a location on a network drive where everyone has access to. The default library location can then be set to this location so everyone uses the same packages.

   The disadvantage of this approach is that if someone updates a packages it is updated for everyone.

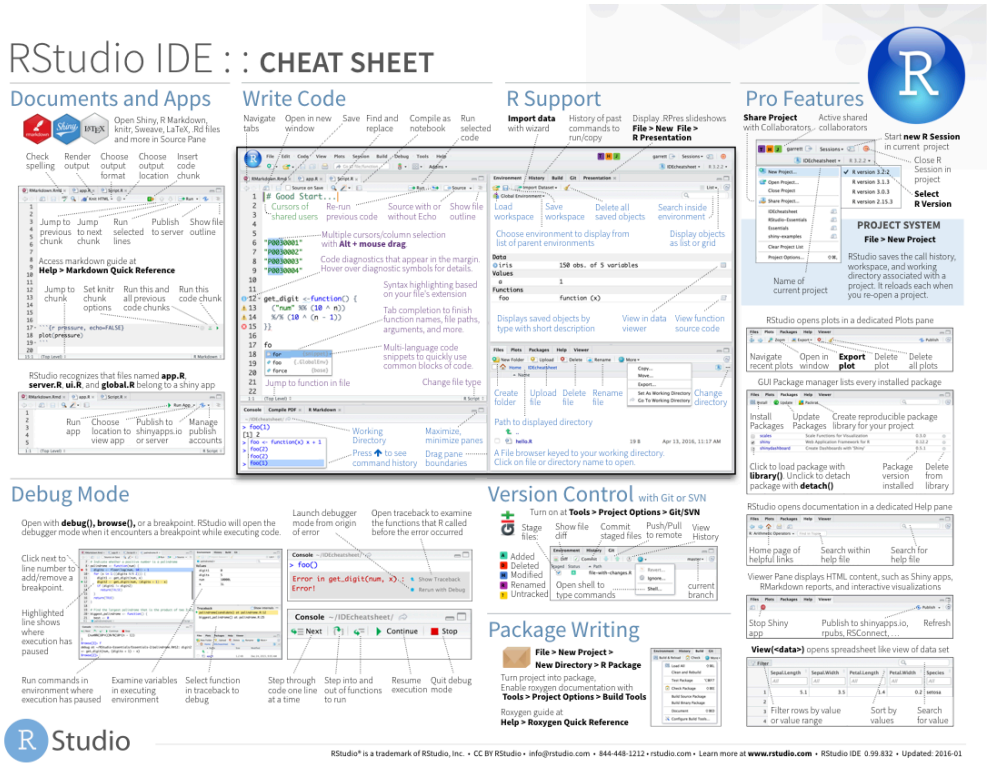

Figure 2.1: RStudio Cheat Sheet

2. The simplest is to use the checkpoint in Microsoft's R. When installing Microsoft R Open a date is set and when packages are installed they are taken from a snapshot of the CRAN repository on that date. That way everyone using the same version of Microsoft R Open is guaranteed to be using the same version of packages.

   The disadvantage of this approach is you cannot use the latest features of a package unless the date is brought forward.

   1 and 2 can be combined to avoid installing packages for every user.

3. Packrat

4. Use Microsoft's `checkpoint` package to set the date for a piece of work. If a newer version of a package is required it can be set in the script and it will be downloaded for that bit of work only.

   The disadvantage of this approach is the required packages are downloaded separately for each project, so should only be used for exceptional cases.

Chapter 3

Writing R Code

3.1 Writing Style

When writing code it is important to follow a common style so it is readable and other people, and your future self, can easily understand and extend it.

3.1.1 Tidyverse

It is recommended, when using R, that we use the "Tidyverse" approach and packages wherever possible. The Tidyverse is a collection of packages designed for data science, as well as a philosophy and style for formatting data and writing functions. The tidyverse.org website details the packages involved and contains articles on how to use them. Some of it will be summarised below, but the website contains the definitive information.

The most in-depth treatment on the Tidyverse can be found in the book "R for Data Science", which is available online for free at r4ds.had.co.nz.

The tidyverse has an extensive style guide, which we summarise here:

3.1.2 Files

- File names should be meaningful and end in `.R`, or `.Rmd` for R markdown files. Only use letters, numbers and `-` or `_`.
- If your script required packages load them all at once at the very beginning of the file.
- Use comments to explain the "why" not the "what" or "how".
- Break up files into named sections using commented lines (`# ----`, example below). In RStudio you can collapse and expand sections commented this way.

```
# Load data ------------------------------------------------------------------
```

3.1.3 Syntax

- Variable and function names should use only lowercase letters, numbers, and `_`.
- Always indent the code inside curly braces `{}` by two spaces.

3.1.4 Functions

- Use verbs for function names where possible.
- If a function has numerous arguments put each one on a new line.
- A function should do one thing well. If it is doing too much the break it up.
- A function should be easily understandable in isolation. It should not refer to any variables outside the function scope.

3.1.5 Pipes

- Use `%>%` when you find yourself composing three or more functions together, instead of a nested call.
- `%>%` should always have a space before it and a new line after it. After the first step, each line should be indented by two spaces.

3.1.6 Documentation

- Created functions should be documentated so others, including the future you, can understand what the function does and how to use it.
- Documentation should be written before the function definition in an roxygen2 style. roxygen uses special comments, starting with `#'`. The first line is the title, and anything else, not prefixed with a keyword forms the description. Keywords start with `@` and the most important ones are `@param` to describe a function parameter and `@return` to describe what the function returns.

Here is a very simple example:

```
#' The length of a string.
#'
#' This function returns the number of characters in the supplied string.
#'
#' @param string input character vector
#'
#' @return integer vector giving number of characters in each element of the
#'   character vector.
#'
#' @export
#'
str_length <- function(string) {
  nchar(string)
}
```

3.2 Structure

Organising files for a particular piece of work becomes more important as the scale and complexity increases. Standard approaches exists to simplify the workflow.

3.2.1 Projects

Use RStudio projects to organise files. This has a number of advantages:

1. Sets the working directory to the project location
2. Reopens the same files when returning to the projects

3.2.2 Folders

When an analysis becomes complex it should be split up into logical parts and stored in subfolders. Store the original data in a folder, unchanged. It is better to "cleanse" input data with an R script as it can repeated when data changes, and/or the approach changed itself.

R code may also be stored in a separate folder. You may have an R script for cleansing the data and another for performing an analysis. Include an R script at the top level which executes the code in the subfolder in the appropriate order. Use relative paths to the files. If an RStudio project has been created the working directory will be set to the project directory automatically.

Output the results, plots, data, etc, in another folder so it is clear whether data files are results rather than inputs.

An example of a project structure:

- data
  - interesting_data.xlsx
  - reference_data.csv
- R
  - clean_data.R
  - analyse.R
- tests
  - test_cleansed_data.R
  - test_analysis.R
- results
  - cool_plot.png
  - table_of_results.csv
- run_code.R
- README.md

3.2.3 README

Adding a README file is a good way to explain to other, and you future self, what the analysis does and how to use it. The documentation section of the best practice has more detail on what should be included.

It is recommended that markdown is used to write the README. It is a very simple way to specify text formatting in a plain text file and can be converted to many other formats (HTML, docx, PDF) if required. In addition, if the package is stored in GitHub a markdown README is automatically rendered on the repository's page.

3.2.4 R Packages

When R code has high criticality consider turning it into a package. A package is a way of collecting together related code in a robust way. It has the following advantages:

- Easier to share with others (as a zip file)
- Documentation is compiled into help pages
- All tests can be executed with a single command
- Can implement a development and release process

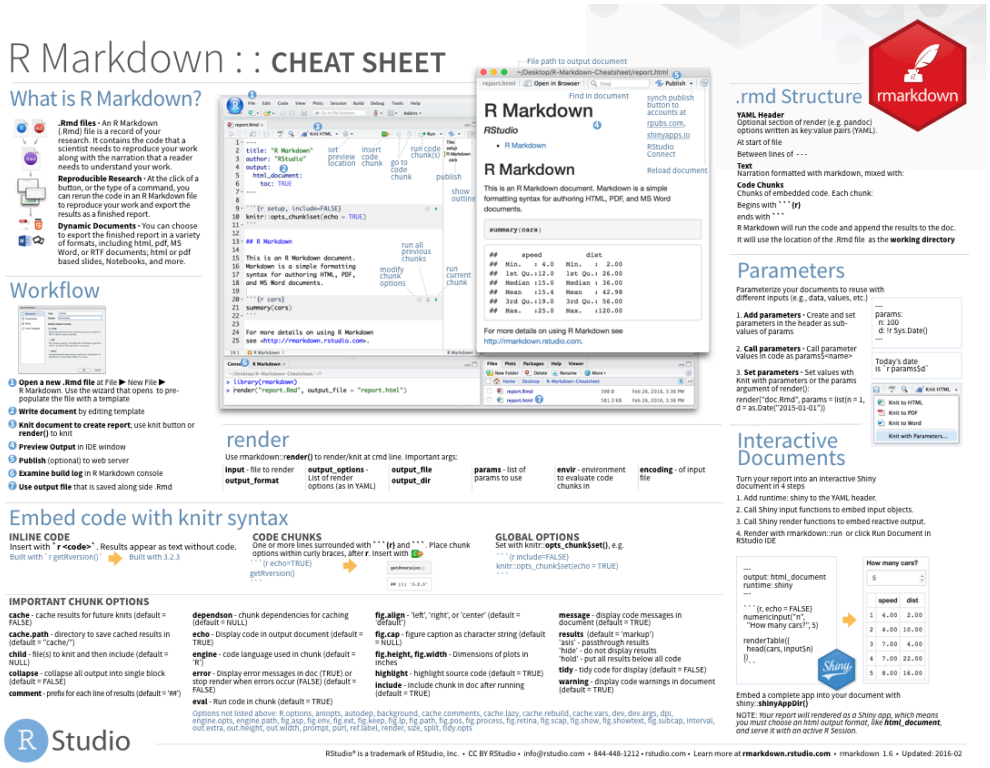

Figure 3.1: R Markdown Cheat Sheet

- Code is broken up into useful functions

Writing a package is very straight forward with the helper packages available today. More information can be found in Package Development.

3.3 R Markdown

R markdown is a way of capturing documentation, code and results and in a single file. The document is written in plain text using a style called markdown. This has a simple syntax for specifying text formatting. R code is added in "chunks" and when the document is rendered the R code is executed and replaced with the results.

R markdown can be used to produce web pages, Word and PDF documents. The provide a robust way of capturing an analysis and the results and can be re-run when the data changes.

RStudio provides a cheatsheet detailing R markdown functionality:

3.3.1 Markdown

Markdown is a lightweight markup language with plain text formatting syntax. It is designed so that it can be converted to HTML and many other formats.

3.3.1.1 Paragraphs

Leave at least one empty line between text to start a new paragraph.

```
This is the first paragraph.

This is the second paragraph.
```

This is the first paragraph.

This is the second paragraph.

3.3.1.2 Headers

```
# Header 1

## Header 2

### Header 3
```

3.3.1.3 Emphasis

```
*italic*   **bold**

_italic_   __bold__
```

italic **bold**

italic **bold**

3.3.1.4 Lists

Unordered List:

```
* Item 1
* Item 2
    + Item 2a
    + Item 2b
```

- Item 1
- Item 2
    - Item 2a
    - Item 2b

Ordered List:

```
1. Item 1
2. Item 2
3. Item 3
    a. Item 3a
    b. Item 3b
```

1. Item 1
2. Item 2
3. Item 3
    a. Item 3a
    b. Item 3b

3.3.1.5 Links

Use a plain http address or add a link to a phrase:

```
http://example.com

[linked phrase](http://example.com)
```

http://example.com
linked phrase

3.3.1.6 Images

Images on the web or local files in the same directory:

```
![](https://upload.wikimedia.org/wikipedia/commons/1/1b/R_logo.svg)

![optional caption text](images/octocat.png)
```

3.3.1.7 Reference Style Links and Images

Links

```
A [linked phrase][id].

At the bottom of the document:

[id]: http://example.com/ "Title"
```

A linked phrase.
At the bottom of the document:
Images

```
![alt text][id]

At the bottom of the document:

[id]: images/octocat.png "Octocat"
```

At the bottom of the document:

3.3.1.8 Blockquotes

```
A friend once said:

> It's always better to give
> than to receive.
```

A friend once said:

> It's always better to give than to receive.

Figure 3.2:

Figure 3.3: optional caption text

Figure 3.4: alt text

3.3.1.9 Plain Code Blocks

Plain code blocks are displayed in a fixed-width font but not evaulated

```
```
This text is displayed verbatim / preformatted
```
```

```
This text is displayed verbatim / preformatted
```

3.3.1.10 Inline Code

```
We defined the `add` function to compute the sum of two numbers.
```

We defined the `add` function to compute the sum of two numbers.

3.3.1.11 LaTeX Equations

Inline equation:

```
Einstein's famous equation $E = mc^2$
```

Einstein's famous equation $E = mc^2$

Display equation:

```
$$
E = mc^2
$$
```

$$E = mc^2$$

3.3.1.12 Horizontal Rule / Page Break

Three or more asterisks or dashes:

```
******
```

```
------
```

3.3.1.13 Tables

```
First Header  | Second Header
------------- | -------------
Content Cell  | Content Cell
Content Cell  | Content Cell
```

First Header	Second Header
Content Cell	Content Cell
Content Cell	Content Cell

3.3.1.14 Manual Line Breaks

End a line with a backslash:

```
Roses are red,\
Violets are blue.
```

Roses are red,
Violets are blue.

3.3.1.15 Miscellaneous

```
superscript^2^
```

```
~~strikethrough~~
```

superscript^2

~~strikethrough~~

3.3.2 R Markdown

3.3.2.1 R Code Chunks

R code will be evaluated and printed

```
summary(cars$dist)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    2.00   26.00   36.00   42.98   56.00  120.00
```

```
summary(cars$speed)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##     4.0    12.0    15.0    15.4    19.0    25.0
```

Inline R Code

There were 50 cars studied

3.3.3 R Notebooks

Chapter 4

Recommended Packages

4.1 Data Wrangling

> Data wrangling is the process of transforming and mapping data from one "raw" data form into another format with the intent of making it more appropriate and valuable for a variety of downstream purposes such as analytics.
>
> – *Wikipedia*

4.1.1 Tidy Data

The Tidyverse of packages are built around the concept of tidy data, first introduced by Jeff Leek in his book *The Elements of Data Analytic Style*. Hadley Wickham summarises the characteristics of tidy data with the following points:

1. Each variable forms a column.
2. Each observation forms a row.
3. Each type of observational unit forms a table.

4.1.2 Cheatsheets

RStudio provide a selection of cheatsheets containing quick reference of R functions useful for common task:

4.1.3 Importing Data

The first step in wrangling is importing the data into R. The methods to do so depend on the source of data:

- Reading files
- Connecting to databases
- Web APIs or pages

4.1.3.1 Reading Files

Files may come in many formats and R has packages to read many of them. The most common for data science are tabular text files, such as CSVs, and Excel spreadsheets. The recommended packages for reading these files are:.

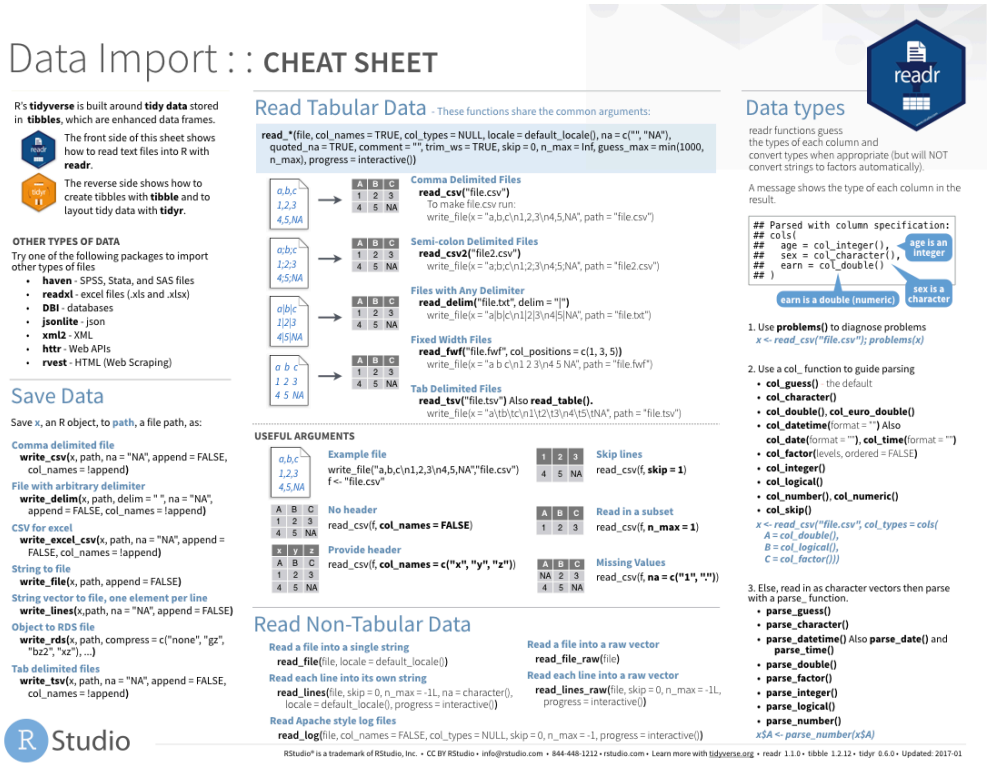

Figure 4.1: Data Import

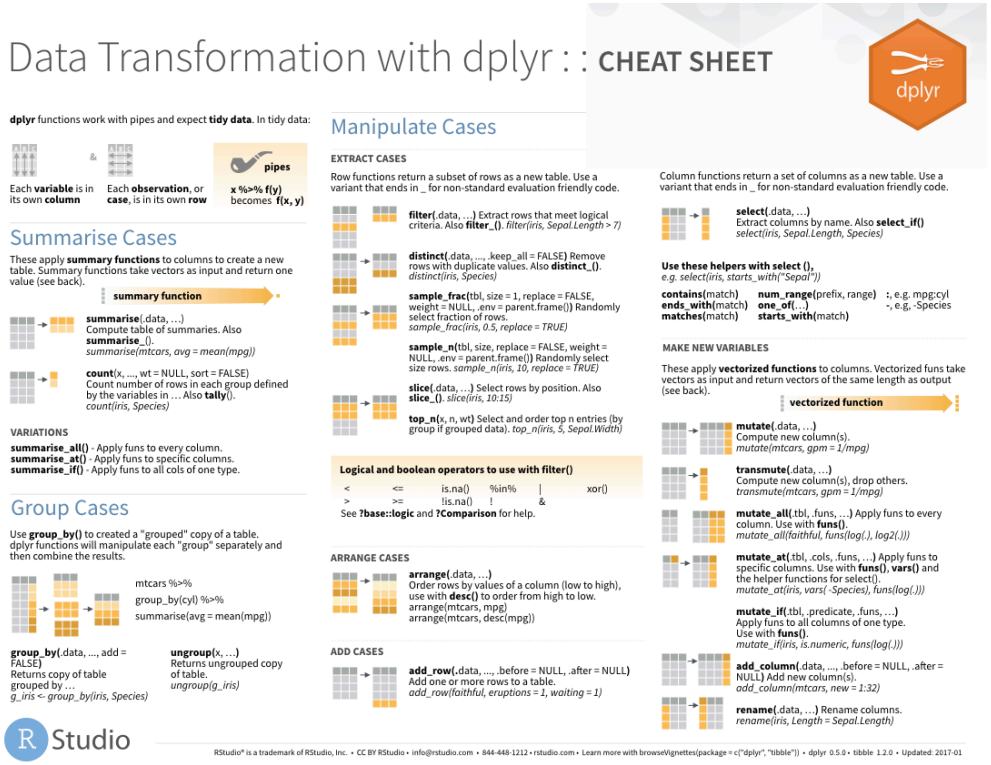

Figure 4.2: Data Transformation

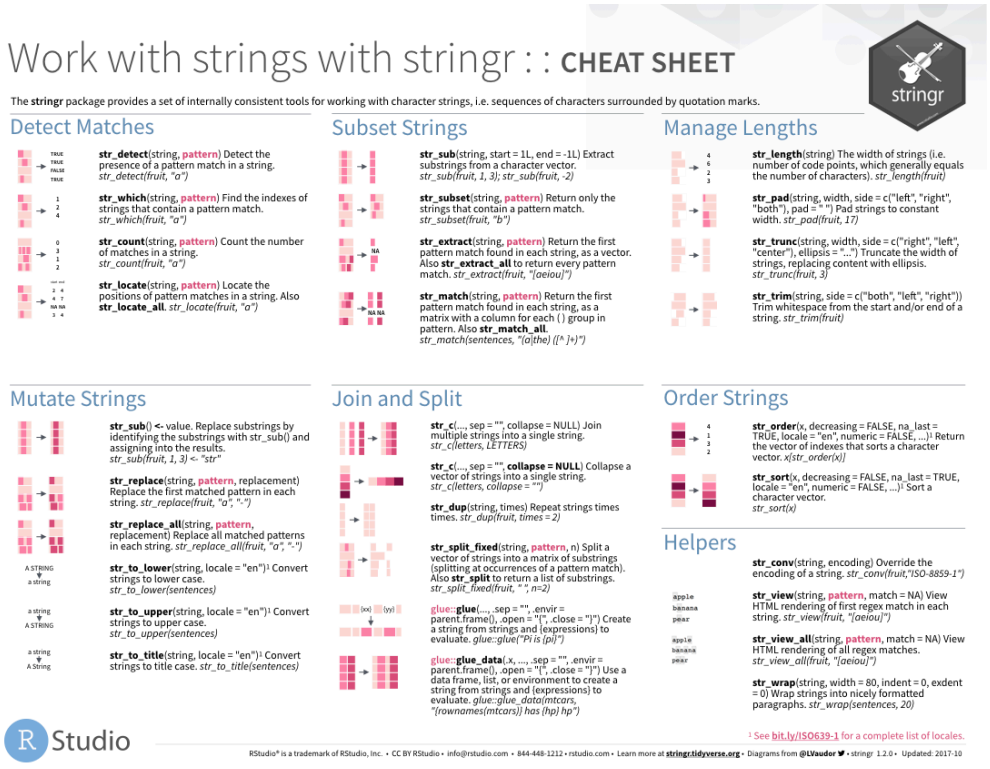

Figure 4.3: Strings

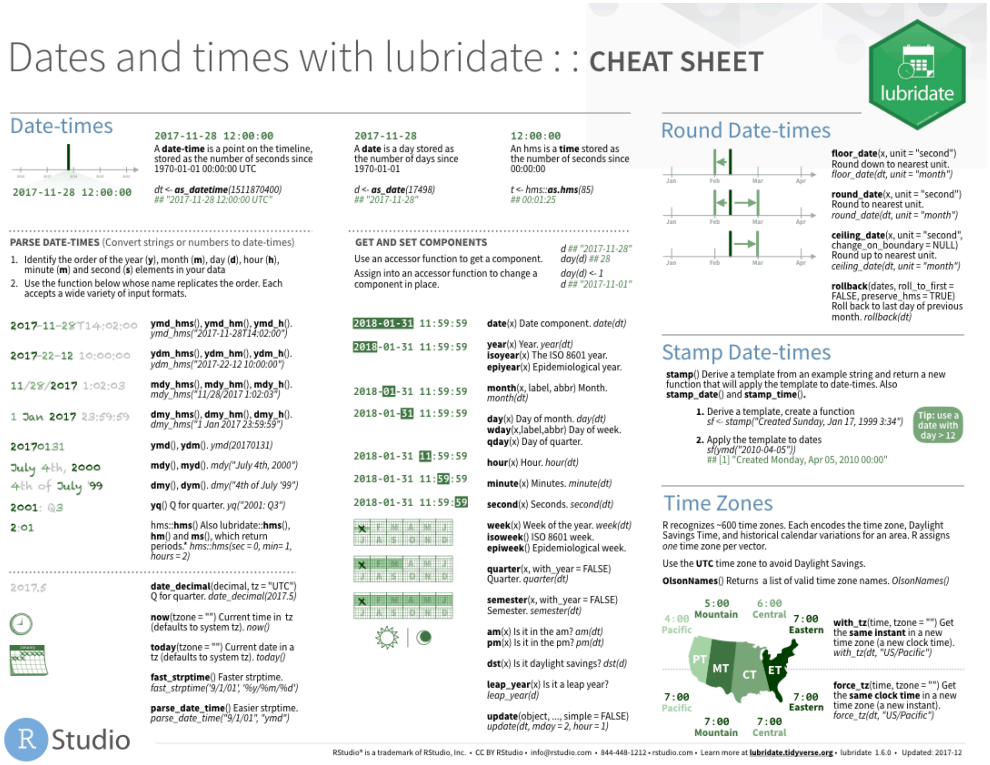

Figure 4.4: Dates and Times

- `readr` for reading text files
- `readxl` for reading Excel files
- `openxlsx` for writing to Excel files

4.1.3.2 Connecting to Databases

The new RStudio Connections Pane makes it possible to easily connect to a variety of data sources, and explore the objects and data inside the connection.

The recommended packages for connecting to databases (also used by RStudio) are:

- `DBI` provides a standard interface to any database
- `odbc` for connecting to databases using ODBC
- `dplyr` for transforming tables in a database

4.1.3.3 Web APIs and Pages

Obtaining data from the internet has two main approaches. If the website provides an application programming interface (API) then you can send and receive data through it. The data from web APIs is usually returned in JSON or XML format. Alternatively, you can scrape the website itself, extracting data from the pages. The recommended packages for these approaches are:

- `httr` for communicating with web APIs
- `jsonlite` for reading JSON formatted text
- `xml2` for reading XML formatted text
- `rvest` for scraping web pages

4.1.4 Tidying Data

Now you have the data you want, it probably requires some processing in order to get it into a structure that is useful for analysis. There are two main packages for this `tidyr` and `dplyr`:

4.1.4.1 `tidyr`

The focus of `tidyr` is to get the data into a tidy format. The main functions it provides to do this are:

- `gather()` takes multiple columns, and gathers them into key-value pairs: it makes "wide" data longer.
- `spread()` takes two columns (key & value) and spreads in to multiple columns, it makes "long" data wider.
- `separate()` and `extract()` pulls apart a column that represents multiple variables.
- `unite()` combines columns, useful if one is redundant.
- `separate_rows()` for separating a row that contains multiple observations.

In addition, `tidyr` provides a set of functions for dealing with implicit and explicit missing values:

- `drop_na()`: Drop rows containing missing values
- `replace_na()`: Replace missing values
- `fill()`: Fill in missing values.
- `complete()`: Complete a data frame with missing combinations of data.
- `expand()`, `crossing()`: Expand data frame to include all combinations of values

4.1.4.2 `dplyr`

`dplyr` is a grammar of data manipulation, providing a consistent set of verbs that help you solve the most common data manipulation challenges:

- `mutate()` adds new variables that are functions of existing variables.
- `select()` picks variables based on their names.
- `filter()` picks cases based on their values.
- `summarise()` reduces multiple values down to a single summary.
- `arrange()` changes the ordering of the rows.

These all combine naturally with `group_by()` which allows you to perform any operation "by group".

`dplyr` also provides a set of functions for combining tables. Mutating joins combine tables based on matching a subset of common variables; Set operations expect the tables to have the same variables and combine observations like sets.

Mutating joins

- `inner_join(x, y)` only includes observations that match in both `x` and `y`.
- `left_join(x, y)` includes all observations in `x`, regardless of whether they match or not.
- `right_join(x, y)` includes all observations in `y`. It's equivalent to `left_join(y, x)`, but the columns will be ordered differently.
- `full_join()` includes all observations from `x` and `y`.

Set operations

- `intersect(x, y)`: returns only observations in both `x` and `y`.
- `union(x, y)`: returns unique observations in `x` and `y`.
- `setdiff(x, y)`: return observations in `x`, but not in `y`.

4.1.4.3 Variable Types

There are also packages which simplify working with specific types of data:

- `stringr` for strings and regular expressions
- `forcats` for factors, used to handle categorical data
- `lubridate` for dates and date-times.
- `hms` for time-of-day values.

4.1.4.4 `tibble`

When using these packages you may notice they return a `tibble` rather than a `data.frame`. They are basically the same thing with three key differences:

1. Printing: Tibbles only show the first ten rows and all the columns that fit on one screen.
2. Subsetting: `[` always returns another tibble. `$` never uses partial matching.
3. Recycling: Only values of length 1 are recycled. Trying to combine columns of different length throws an error.

The package `tibble` has some useful functions for constructing tibbles. In particaular, you can construct a tibble row-wise using the `tribble` function (transposed tibble).