R Best Practice

Chad Goymer

2019-03-06

Contents

Chapter 1

Introduction

This document sets out the recommendations 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. **Recommended Software**: Details the recommended software to install, both the core R applications and supporting software.
2. **Writing R Code**: Gives guidance on on how to write R scripts and applications.
3. **Recommended Packages**: Lists the recommended packages to use for achieving common tasks.
4. **Development Practices**: 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:

- Documentation
- Testing
- Reproducibility
- Change control
- Access control

The degree to which the above points need to be addressed depend on the result of a simple risk analysis. For example, if a piece of work is a one-off analysis and the results are not distributed further, then some simple documentation is all that is required. If the work is a complex set of functions, which important systems depend on, then each function must be clearly documented and user guides specified.

More detail is given on each of the areas below.

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 our end-user computing standards require 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 the following recommendations should be considered.

1.3 Documentation

Documentation ensures everyone using the code understands what it does and how to use it effectively. For simple scripts, a commented header in the file is sufficient. For multi-file applications it is recommended that a separate README text file is added.

1.3.0.1 Low Complexity and Criticality

As a minimum, it is recommended that the following are documented with the code:

- Record the **name** of the project, analysis or application.
- Write a **description** of of what the code does (not how).
- Record the **purpose** of the code.
- Write **instructions** for how to use the code.

1.3.0.2 Medium Complexity

As the complexity increases it becomes important to ensure users know how to use the code correctly. In addition to the list above, the following is also recommended:

- Provide **examples** of usage.

1.3.0.3 High Complexity

For high complexity work it is important that the code is broken up into sections or functions. Therefore the following are recommended:

- Write **separate descriptions** for each section or function.
- Provide **separate examples** for each section or function.
- Write a **user guide**

1.3.0.4 Medium Criticality

As criticality increases it becomes important to ensure reproducibility is possible. Therefore, in addition to the minimum requirements above, following is recommended:

- Clearly define **dependencies** (packages used, input data used,…etc).

1.3.0.5 High Criticality

For high criticality work consider creating an R package then:

- **Document the package** by creating a DESCRIPTION.
- **Document each function** using `Roxygen2`.
- Write a **user guide** as a vignette explaining its use.

1.4 Testing

Testing is required to ensure the code actually does what is intended. It is also necesary to record the tests, so they can be repeated, and the results, as evidence of correctness.

1.4.0.1 Low Complexity and Criticality

As a minimum, manual tests should be performed to ensure the user is comfortable the code is working as intended. In order to reproduce the tests it is recommended the following are recorded:

- Describe the **test process**.
- Record the **test results**.

1.4.0.2 Medium Complexity

As the complexity increases it becomes more important to ensure the code is tested, consider:

- **Comparisons** with known results.
- **Regression tests** to compare new results to previous ones.

1.4.0.3 High Complexity

For high complexity work it is important that the code is broken up into sections or functions. Therefore, following are recommended:

- **Separate tests** for each section or function.
- **Unit tests** to compare with expected results when given simple inputs.

1.4.0.4 Medium Criticality

As the criticality increases checks should be added within the code. This ensures the code and the results are tested each time it is run. The following are also recommended:

- **Assertions on the inputs** to ensure they are valid for the code (e.g. a particular value is positive).
- Check **appropriate errors or warnings** are returned when inputs are invalid.

1.4.0.5 High Criticality

For high criticality work consider creating an R package then:

- **Test the package** using `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.

1.5.0.1 Low Complexity and Criticality

As a minimum, it should be clear where the inputs came from. The following are recommended:

- Clearly define how to **import input data** so it can be re-executed.
- Consider keeping a **copy of input data and parameters** alongside the code.

1.5.0.2 Medium Complexity

As the complexity increases consider using a structured project, for example.

- Use **RStudio projects**, which allows you to reference files using relative paths and save open files and workspaces.
- Consider using **R markdown** which captures description, code and results in a single document.

1.5.0.3 High Complexity

For high complexity work it the following are recommended:

- It is **peer reviewed**. The best way to ensure others can reproduce your work is to get a peer review.
- **Avoid referencing external files**, unless you can be positive they will always be available in the specified location. Even then, it is worth **caching external files** in a subfolder, if they are not too large.

1.5.0.4 Medium Criticality

As the criticality increases following is recommended:

- Output an **audit log** with a time stamp, username of the person executing the code and a summary of computing environment (e.g. package versions used).

1.5.0.5 High Criticality

For high criticality work also consider:

- Holding **inputs in version control** or other system that allows you to retrieve old versions.
- Using packrat to **manage the versions of packages** used.

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.

1.6.0.1 Low Complexity and Criticality

The simplest approach is to:

- Produce **new files for each version**.
- **Archive older versions** so you can rollback is necessary.
- **Document changes**, concentrating on why rather than what.

1.6.0.2 Medium Complexity

As complexity increases it becomes more important to track changes in a more robust manner. Instead of the approach above:

- Use **version control software**, such as Git, to track changes.

1.6.0.3 High Complexity

- **Break changes down into tasks** and implement each one separately.
- Create a **branch for each task** and merge back into the main branch once complete.

1.6.0.4 Medium Criticality

As criticality increases it becomes more important to confirm changes are correct and appropriate.

- Use **version control software**, such as Git, to track changes.
- Ensure **changes are peer reviewed** and documented.

1.6.0.5 High Criticality

- Use **version control hosting application**, such as GitHub or Microsoft's Azure DevOps, to share changes.
- **Formalise reviews** using pull requests.
- **Require approval** from an appropriate person or group before deploying 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.

1.7.0.1 Low Complexity and Criticality

The simplest approach is:

- Save files to a network drive, with **access controlled** by the IT dept.

1.7.0.2 Medium Complexity

As complexity increases it is important to ensure that no unintentional changes are propagated into production.

- **Separate source code** from results.
- **Maintain a protected version**, which can only be updated by appropriate people (Note: GitHub provides this functionality out-of-the-box).

1.7.0.3 High Complexity

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

- Use **version control hosting application**, such as GitHub or Microsoft's Azure DevOps, to share changes.
- Deploy to a **production location** for users.

1.7.0.4 Medium Criticality

As criticality increases is important to ensure work is protected and backed up.

- Ensure an appropriate **back-up and recovery strategy** is in place.
- Ensure that the code has gone through an **appropriate review** before deploying to production.

1.7.0.5 High Criticality

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.

- Maintain and **regularly review the list of developers** and users.
- Ensure that the code has gone through an **appropriate release process** before deploying to production.

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. Differences in packages can produced unexpected results and all versions of packages are not compatible with all versions of R.

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 installation is:

- R: Use the latest version whenever possible. If older versions are required they can be installed side-by-side. Within RStudio, the version of R to use can be selected in the Global Options. R can be used for Free.
- RStudio Desktop: To edit R scripts and manage projects. As with R use the latest version possible. RStudio can also be used for free.
- RTools: A set of tools required to build R packages. You must make sure you have the version compatible with the most recent version of R installed. RTools can also be used for free.

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 medium (or high) complexity or criticality it is recommended that version control is used:

- Git is the market leader version control software and integrates well with RStudio. Git is free.
- GitKraken provides a very good graphical interface to Git and access to more features than RStudio. If you are working on complex projects with other team members then consider using GitKraken. GitKraken is not free, but it does not cost very much.
- GitHub is a hosting platform for Git repositories. It also allows you to control access, formalise code reviews and plan projects.

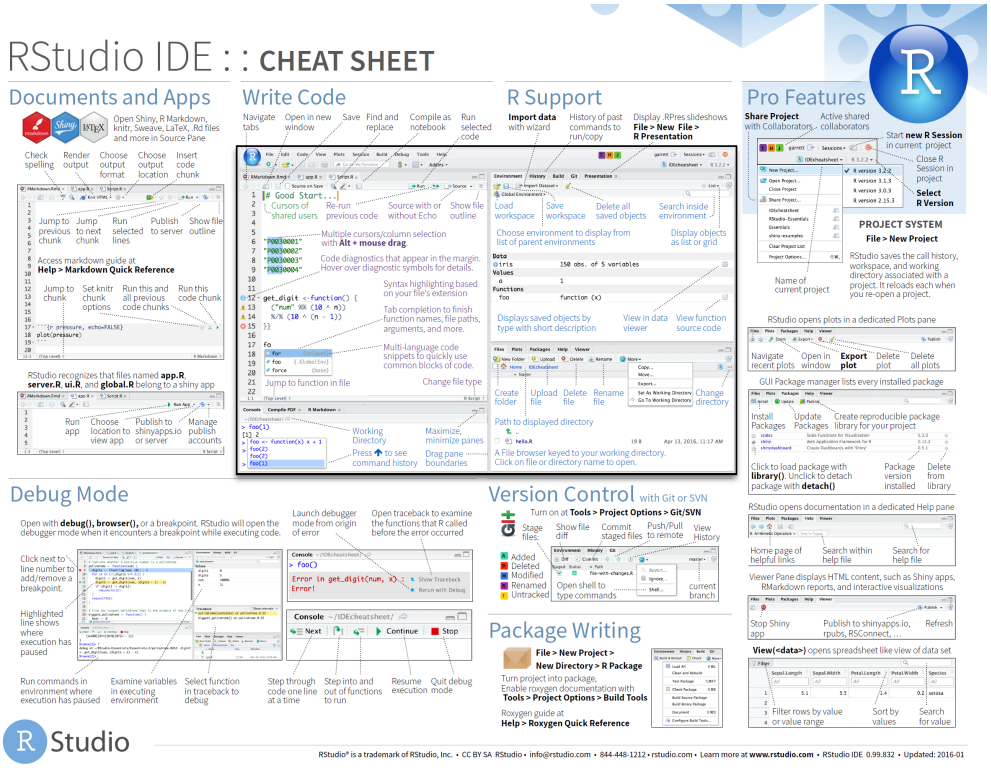


Figure 2.1: RStudio Cheat Sheet

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. The simplest 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.

2. Another option, which can be used with the first option, is to point everyone's R installation to a snapshot of CRAN. This ensures that everyone installs packages released at a specified date. This can be achieved using either:

   - Microsoft's Time Machine allows you select the version of CRAN to used based on a date.
   - RStudio's Package Manager allows you to select a version of CRAN based on an ID. RStudio Package Manager is installed on premise and also allows the company to control the packages that can be installed.

3. The most flexible approach is to use packrat when developing projects or packages. Packrat allows you to specify which versions of dependent packages you use for the project, and ensures that everyone using it also uses the same versions of the packages.

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. Saves the workspace so data and code is loaded when you re-open the project

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
- Code is broken up into useful functions

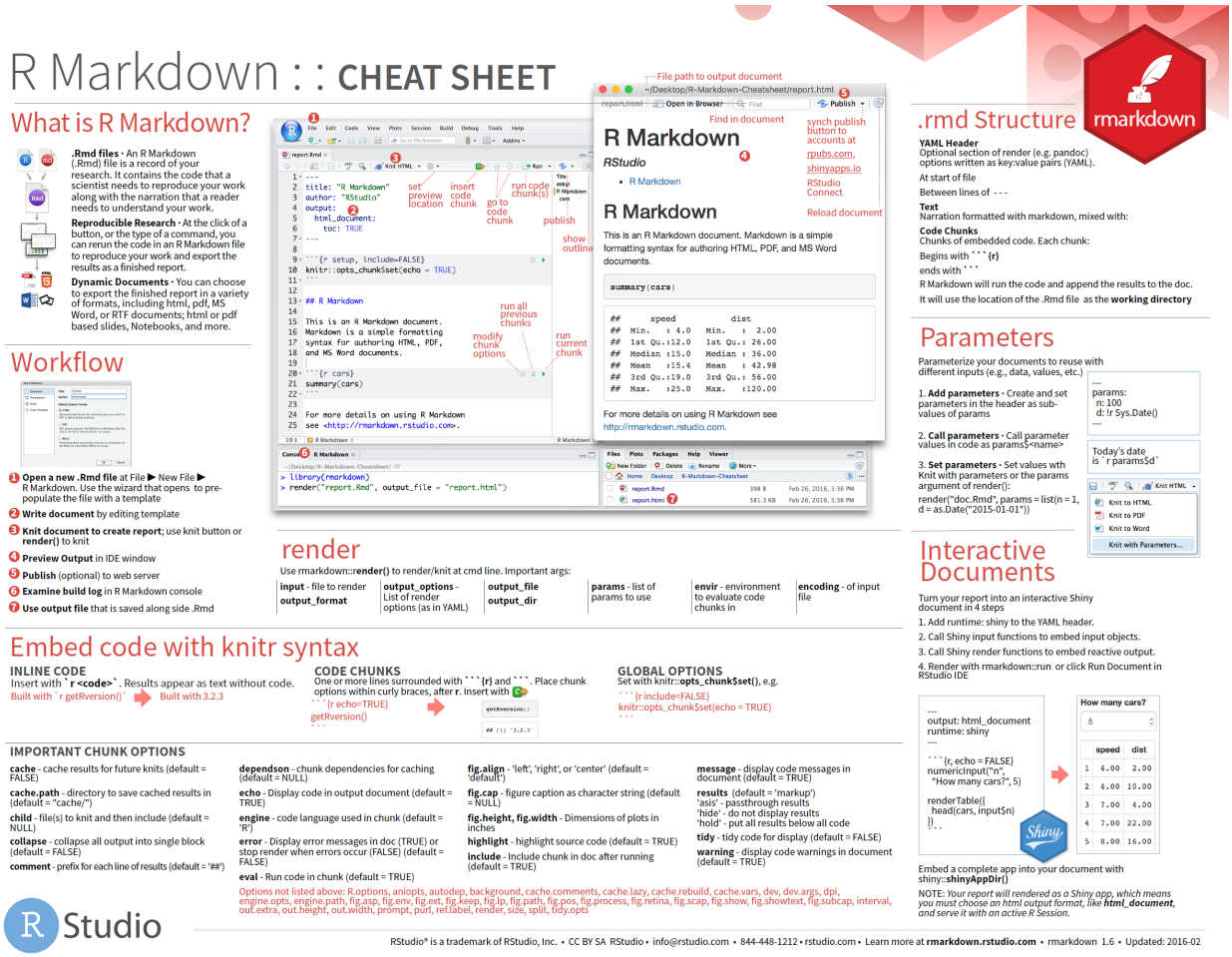
Writing a package is very straight forward with the helper packages available today. More information can be found in Package Development.

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:



For a detailed guide the book R Markdown is available for free online.

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:

Figure 3.1: optional caption text

Figure 3.2: alt text

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.

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 Code Chunks

R code surrounded with three ticks and designated `{R}` (see below) 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
```

```
```

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

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

```r
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
```

There were 50 cars studied

There are many options available when executing R code chunks. For more information read the R code chunks chapter in the R Markdown book.

3.3.3 R Notebooks

An R Notebook is an R Markdown document with chunks that can be executed independently and interactively, with output visible immediately beneath the input. They direct interaction with R while producing a reproducible document with publication-quality output.

Any R Markdown document can be used as a notebook, and all R Notebooks can be rendered to other R Markdown document types. A notebook can therefore be thought of as a special execution mode for R Markdown documents. The immediacy of notebook mode makes it a good choice while authoring the R Markdown document and iterating on code. When you are ready to publish the document, you can share the notebook directly, or render it to a publication format with the Knit button.

3.3.3.1 Creating a Notebook

You can create a new notebook in RStudio with the menu command `File -> New File -> R Notebook`, or by using the `html_notebook` output type in your document's YAML metadata.

```
---
title: "My Notebook"
output: html_notebook
---
```

3.3.3.2 Inserting chunks

Notebook chunks can be inserted quickly using the keyboard shortcut `Ctrl + Alt + I`, or via the `Insert` menu in the editor toolbar.

Because all of a chunk's output appears beneath the chunk (not alongside the statement which emitted the output, as it does in the rendered R Markdown output), it is often helpful to split chunks that produce multiple outputs into two or more chunks which each produce only one output.

3.3.4 Executing chunks

To execute a chunk of code use the green triangle button on the toolbar of a code chunk that has the tooltip "Run Current Chunk", or `Ctrl + Shift + Enter` to run the current chunk. The result is then displayed underneath the chunk.

3.3.5 Saving and sharing

When a notebook `*.Rmd` file is saved, a `*.nb.html` file is created alongside it. This file is a self-contained HTML file which contains both a rendered copy of the notebook with all current chunk outputs (suitable for display on a website) and a copy of the `*.Rmd` file itself.

You can view the `*.nb.html` file in any ordinary web browser. It can also be opened in RStudio; when you open there (e.g., using `File -> Open File`), RStudio will do the following:

1. Extract the bundled `*.Rmd` file, and place it alongside the `*.nb.html` file.
2. Open the `*.Rmd` file in a new RStudio editor tab.
3. Extract the chunk outputs from the `*.nb.html` file, and place them appropriately in the editor.

3.3.5.1 More information

For more a more detailed guide on R Notebooks read the Notebook chapter in the R Markdown book.

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 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

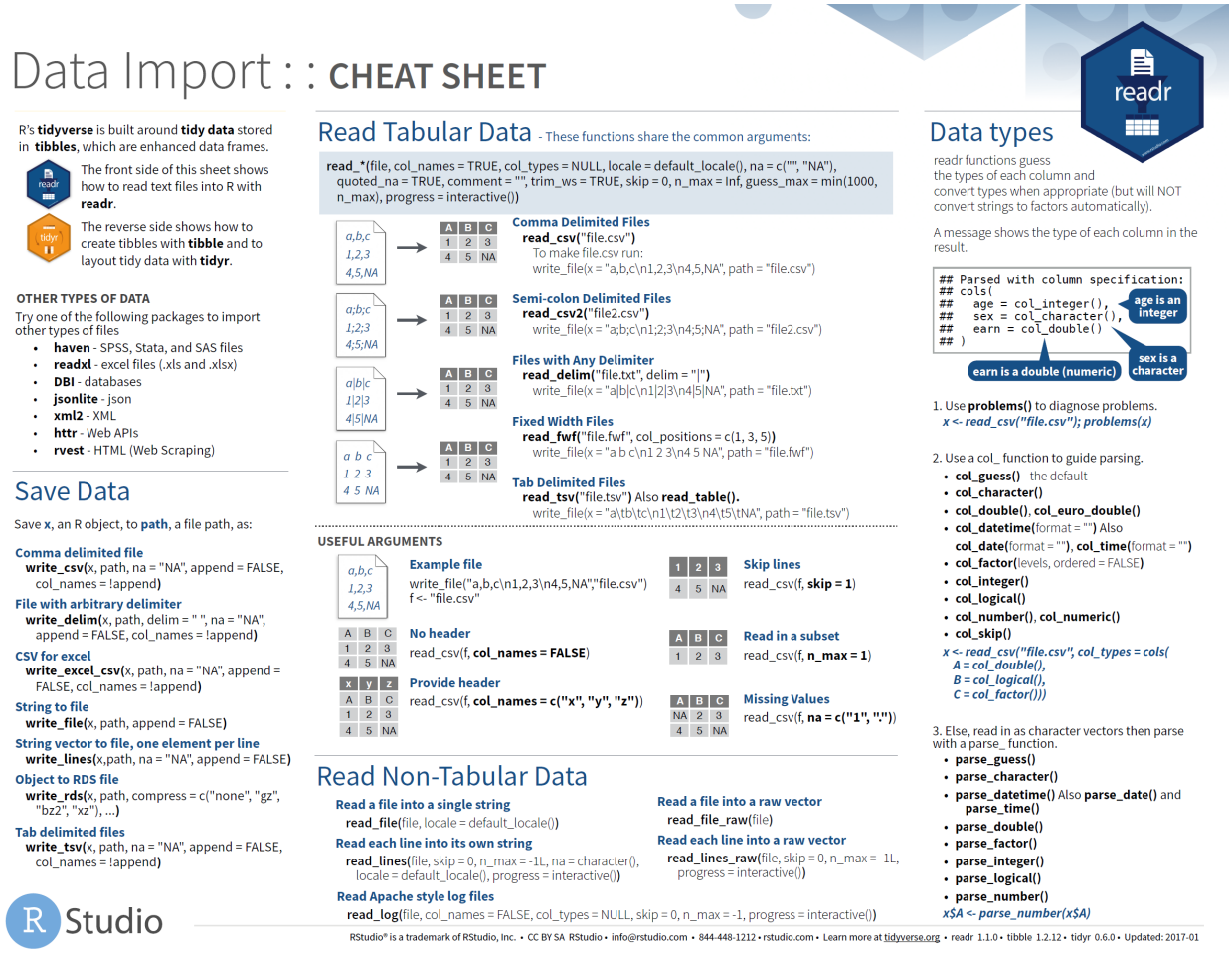
4.1.2.1 Reading Files

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:.

- `readr` for reading text files
- `readxl` for reading Excel files

- `openxlsx` for writing to Excel files

For more detail, read the Data Import chapter in the *R for Data Science* book or use the cheatsheet below for reference.



4.1.2.2 Tibbles

When using the Tidyverse packages you may notice they return a `tibble` rather than a `data.frame` They are basically the same thing except construction and subsetting are more consistent. They also display nicely when printed to the console. The package `tibble` has some useful functions for constructing tibbles.

For more detail, read the Tibbles chapter in the *R for Data Science* book or use the *Data Import* cheatsheet above for reference.

4.1.2.3 Connecting to Databases

The 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

For more detail, read the Databases using R website by RStudio.

4.1.2.4 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

4.1.3 Transforming 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`

4.1.3.1 Tidying Data

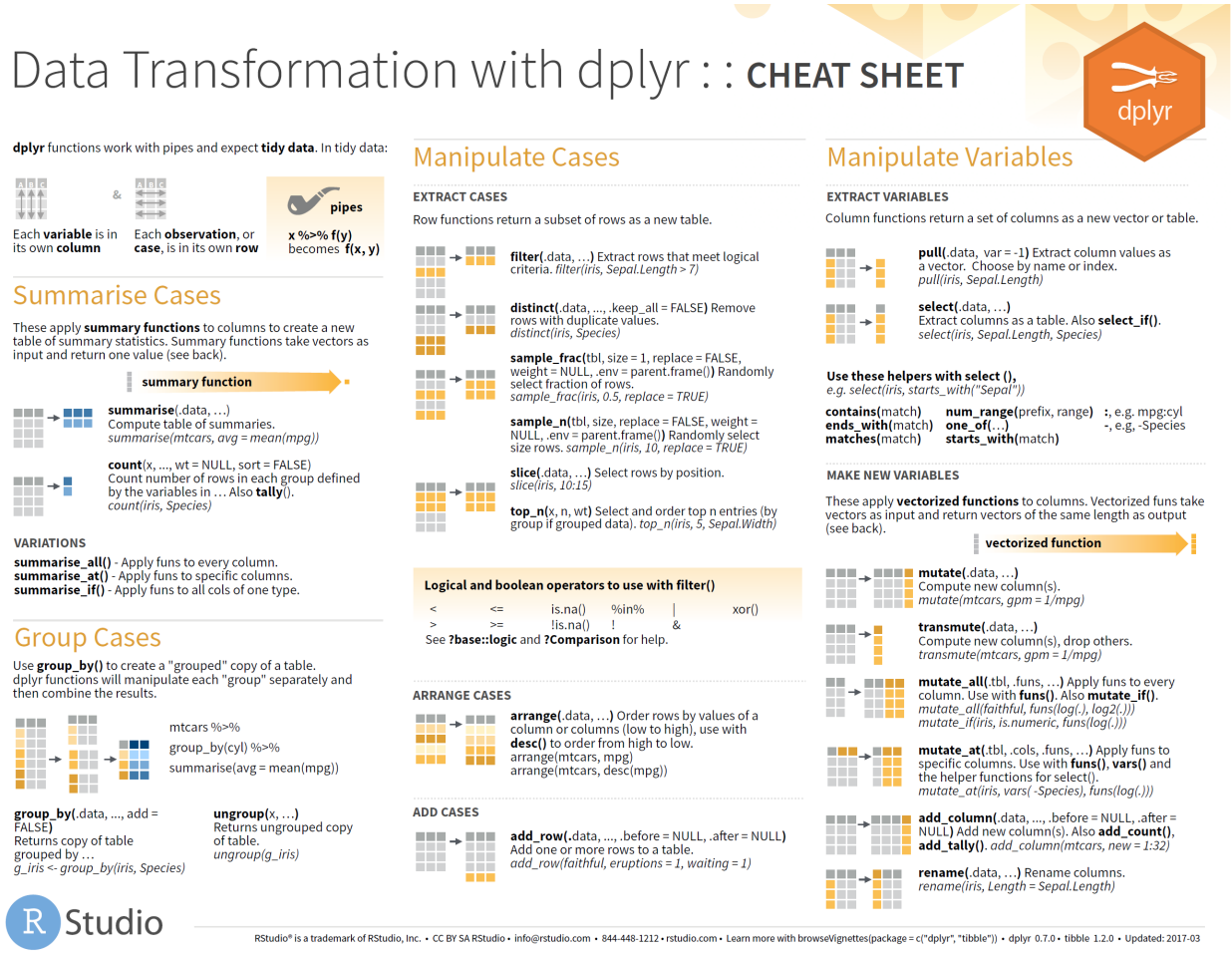
The focus of `tidyr` is to get the data into a tidy format. It provides functions for reshaping the data, as well as dealing with implicit and explicit missing values.

For more detail, read the Tidy Data chapter in the *R for Data Science* book or use the *Data Import* cheatsheet above for reference.

4.1.3.2 Manipulating Data

`dplyr` is a grammar of data manipulation, providing a consistent set of verbs that help you solve the most common data manipulation challenges. For example; Calculating a new variable (or column) based on existing ones, selecting columns, filtering tables and summarising them by groups. `dplyr` also provides a set of functions for combining tables.

For more detail, read the Data Transformation chapter in the *R for Data Science* book or use the cheatsheet below for reference.



4.1.3.3 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.

For more detail, read the Strings, Factors, or Dates and Times chapters in the *R for Data Science* book or use the cheatsheets below for reference.

String manipulation with stringr : : CHEAT SHEET

The **stringr** package provides a set of internally consistent tools for working with character strings, i.e. sequences of characters surrounded by quotation marks.

Detect Matches

str_detect(string, **pattern**) Detect the presence of a pattern match in a string. *str_detect(fruit, "a")*

str_which(string, **pattern**) Find the indexes of strings that contain a pattern match. *str_which(fruit, "a")*

str_count(string, **pattern**) Count the number of matches in a string. *str_count(fruit, "a")*

str_locate(string, **pattern**) Locate the positions of pattern matches in a string. Also **str_locate_all**. *str_locate(fruit, "a")*

Subset Strings

str_sub(string, start = 1L, end = -1L) Extract substrings from a character vector. *str_sub(fruit, 1, 3); str_sub(fruit, -2)*

str_subset(string, **pattern**) Return only the strings that contain a pattern match. *str_subset(fruit, "b")*

str_extract(string, **pattern**) Return the first pattern match found in each string, as a vector. Also **str_extract_all** to return every pattern match. *str_extract(fruit, "[aeiou]")*

str_match(string, **pattern**) Return the first pattern match found in each string, as a matrix with a column for each ( ) group in pattern. Also **str_match_all**. *str_match(sentences, "(a|the) ([^]+)")*

Manage Lengths

str_length(string) The width of strings (i.e. number of code points, which generally equals the number of characters). *str_length(fruit)*

str_pad(string, width, side = c("left", "right", "both"), pad = " ") Pad strings to constant width. *str_pad(fruit, 17)*

str_trunc(string, width, side = c("right", "left", "center"), ellipsis = "...") Truncate the width of strings, replacing content with ellipsis. *str_trunc(fruit, 3)*

str_trim(string, side = c("both", "left", "right")) Trim whitespace from the start and/or end of a string. *str_trim(fruit)*

Mutate Strings

str_sub() **<-** value. Replace substrings by identifying the substrings with str_sub() and assigning into the results. *str_sub(fruit, 1, 3) <- "str"*

str_replace(string, **pattern**, replacement) Replace the first matched pattern in each string. *str_replace(fruit, "a", "-")*

str_replace_all(string, **pattern**, replacement) Replace all matched patterns in each string. *str_replace_all(fruit, "a", "-")*

str_to_lower(string, locale = "en")[1] Convert strings to lower case. *str_to_lower(sentences)*

str_to_upper(string, locale = "en")[1] Convert strings to upper case. *str_to_upper(sentences)*

str_to_title(string, locale = "en")[1] Convert strings to title case. *str_to_title(sentences)*

Join and Split

str_c(..., sep = "", collapse = NULL) Join multiple strings into a single string. *str_c(letters, LETTERS)*

str_c(..., sep = "", **collapse = NULL**) Collapse a vector of strings into a single string. *str_c(letters, collapse = "")*

str_dup(string, times) Repeat strings times times. *str_dup(fruit, times = 2)*

str_split_fixed(string, **pattern**, n) Split a vector of strings into a matrix of substrings (splitting at occurrences of a pattern match). Also **str_split** to return a list of substrings. *str_split_fixed(fruit, " ", n=2)*

str_glue(..., .sep = "", .envir = parent.frame()) Create a string from strings and {expressions} to evaluate. *str_glue("Pi is {pi}")*

str_glue_data(.x, ..., .sep = "", .envir = parent.frame(), .na = "NA") Use a data frame, list, or environment to create a string from strings and {expressions} to evaluate. *str_glue_data(mtcars, "{rownames(mtcars)} has {hp} hp")*

Order Strings

str_order(x, decreasing = FALSE, na_last = TRUE, locale = "en", numeric = FALSE, ...)[1] Return the vector of indexes that sorts a character vector. *x[str_order(x)]*

str_sort(x, decreasing = FALSE, na_last = TRUE, locale = "en", numeric = FALSE, ...)[1] Sort a character vector. *str_sort(x)*

Helpers

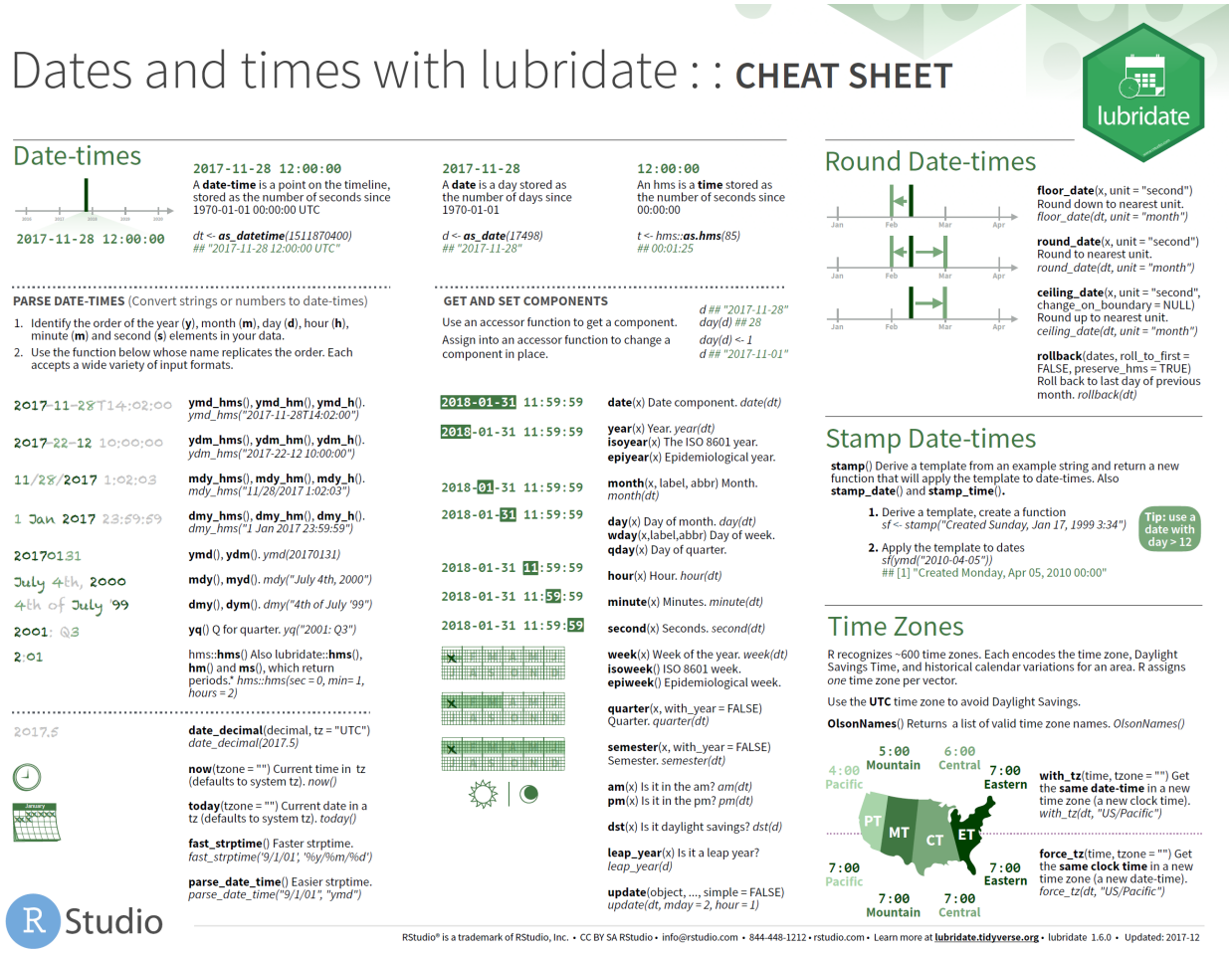
str_conv(string, encoding) Override the encoding of a string. *str_conv(fruit,"ISO-8859-1")*

str_view(string, **pattern**, match = NA) View HTML rendering of first regex match in each string. *str_view(fruit, "[aeiou]")*

str_view_all(string, **pattern**, match = NA) View HTML rendering of all regex matches. *str_view_all(fruit, "[aeiou]")*

str_wrap(string, width = 80, indent = 0, exdent = 0) Wrap strings into nicely formatted paragraphs. *str_wrap(sentences, 20)*

[1] See bit.ly/ISO639-1 for a complete list of locales.



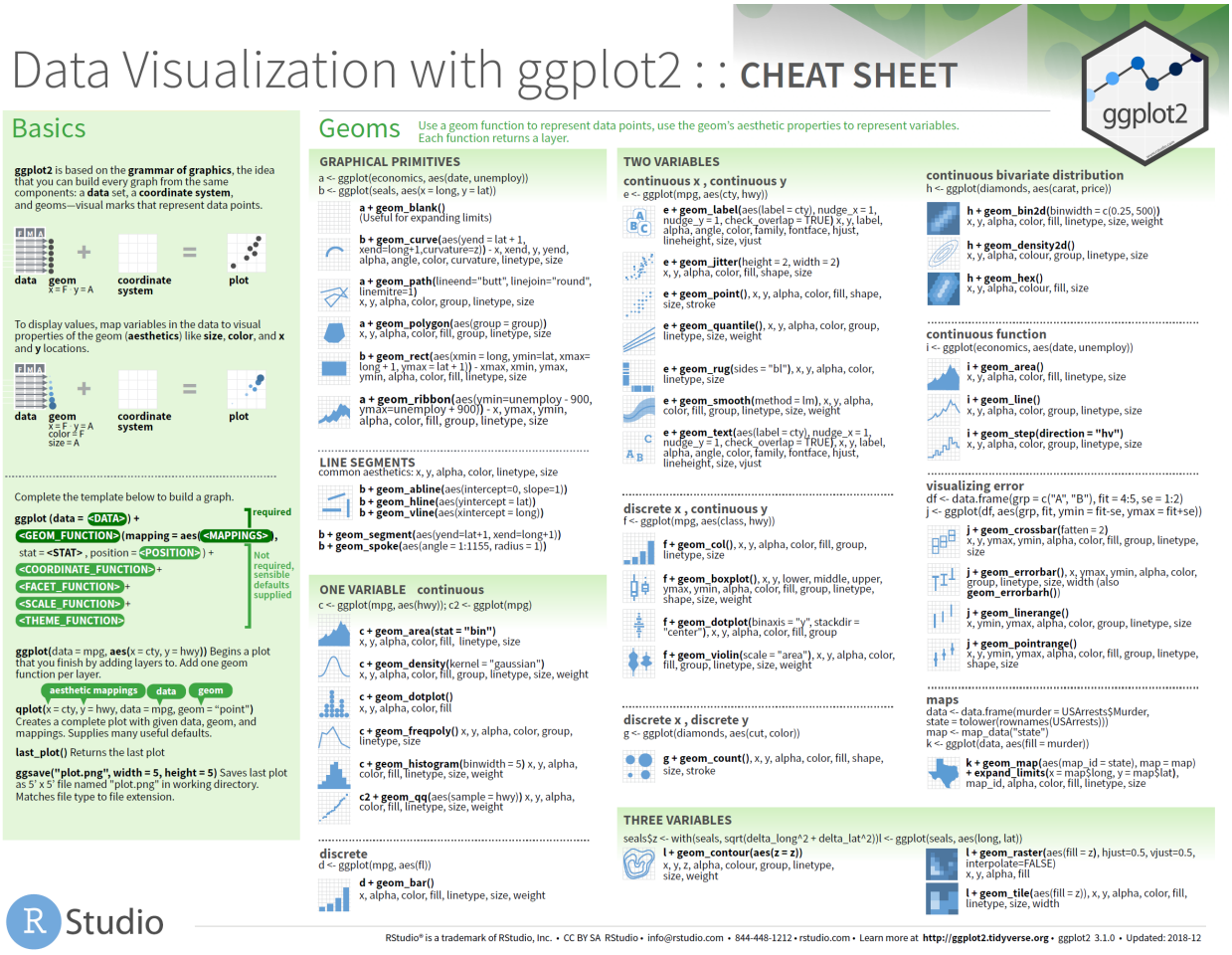
4.2 Visualising Data

When it comes to visualising data the most commonly used package is `ggplot2`. `ggplot2` only produces static plots, which is great if you want to insert it into a report, but if you want to create interactive graphs in a web page use the `plotly` package.

4.2.1 Grammer of Graphics

The `ggplot2` package is based on the grammer of graphics which allows you to create plots by adding layers of geometric objects, e.g. point, lines, etc, with aesthetics mapped to variables, e.g. coordinates, size, etc. This framework allows for highly customisable plots.

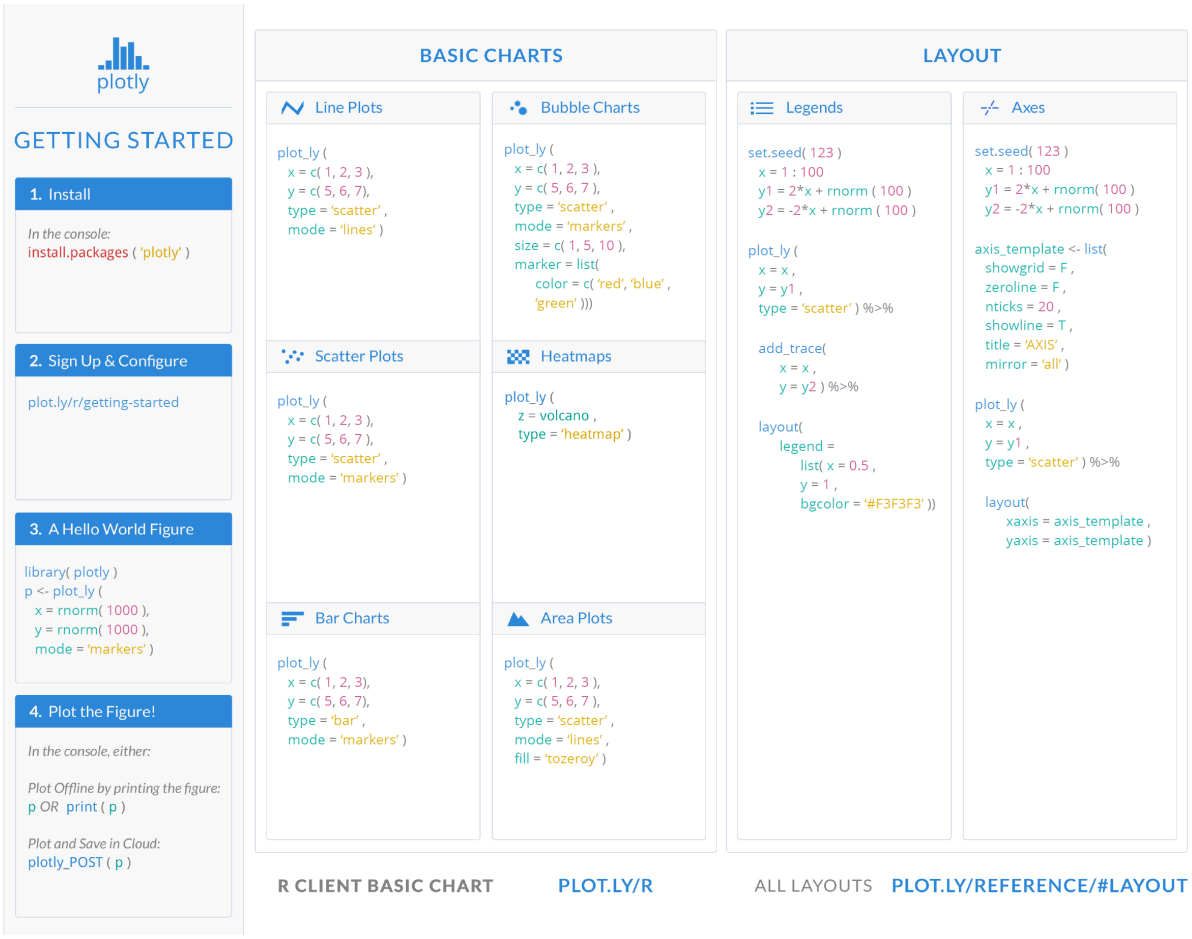
For more detail, read the Data Visualisation chapter in the *R for Data Science* book or use the cheatsheet below for reference.



4.2.2 Interactive Graphics

If you want to add a plot to a web page then you'll probably want something more interactive. `plotly` produces high-quality graphics with the ability to zoom, pan and add tool-tips, etc.

For more detail, read the Plotly for R book or use the cheatsheet below for reference.

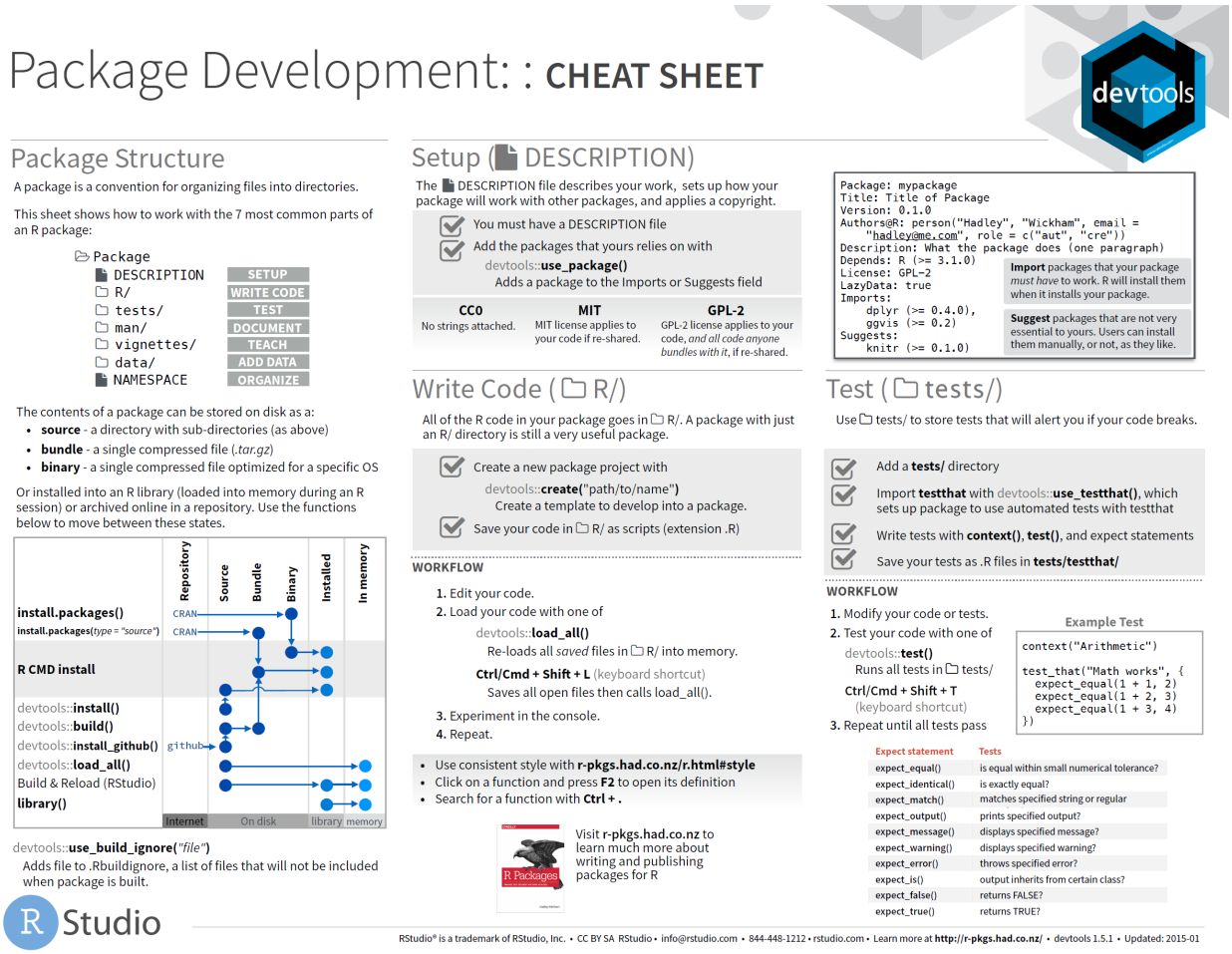

plotly
GETTING STARTED
1. Install
In the console:
install.packages ('plotly')
2. Sign Up & Configure
plot.ly/r/getting-started
3. A Hello World Figure
library(plotly)
p <- plot_ly (
x = rnorm(1000),
y = rnorm(1000),
mode = 'markers')
4. Plot the Figure!
In the console, either:
Plot Offline by printing the figure:
p OR print (p)
Plot and Save in Cloud:
plotly_POST (p)
BASIC CHARTS
Line Plots
plot_ly (
x = c(1, 2, 3),
y = c(5, 6, 7),
type = 'scatter' ,
mode = 'lines')
Bubble Charts
plot_ly (
x = c(1, 2, 3),
y = c(5, 6, 7),
type = 'scatter' ,
mode = 'markers' ,
size = c(1, 5, 10),
marker = list(
color = c('red', 'blue' ,
'green')))
Scatter Plots
plot_ly (
x = c(1, 2, 3),
y = c(5, 6, 7),
type = 'scatter' ,
mode = 'markers')
Heatmaps
plot_ly (
z = volcano ,
type = 'heatmap')
Bar Charts
plot_ly (
x = c(1, 2, 3),
y = c(5, 6, 7),
type = 'bar' ,
mode = 'markers')
Area Plots
plot_ly (
x = c(1, 2, 3),
y = c(5, 6, 7),
type = 'scatter' ,
mode = 'lines' ,
fill = 'tozeroy')
LAYOUT
Legends
set.seed(123)
x = 1 : 100
y1 = 2*x + rnorm (100)
y2 = -2*x + rnorm (100)
plot_ly (
x = x ,
y = y1 ,
type = 'scatter') %>%
add_trace(
x = x ,
y = y2) %>%
layout(
legend =
list(x = 0.5 ,
y = 1 ,
bgcolor = '#F3F3F3'))
Axes
set.seed(123)
x = 1 : 100
y1 = 2*x + rnorm(100)
y2 = -2*x + rnorm(100)
axis_template <- list(
showgrid = F ,
zeroline = F ,
nticks = 20 ,
showline = T ,
title = 'AXIS' ,
mirror = 'all')
plot_ly (
x = x ,
y = y1 ,
type = 'scatter') %>%
layout(
xaxis = axis_template ,
yaxis = axis_template)
R CLIENT BASIC CHART
PLOT.LY/R
ALL LAYOUTS
PLOT.LY/REFERENCE/#LAYOUT

Chapter 5

Development Practices

5.1 Package Development

If you have functions or code that is used across projects then the best practice is to build a package. A package combines a set of related functionality which can then be shared easily between projects and other users. The package `devtools` makes package development straight forward. For more detailed information about creating packages, read the R Packages book or use the cheatsheet below for reference.



NOTE: `devtools` is currently being split up into smaller packages and the above book and cheatsheet are slightly out of date.

Package development has a formal structure, which addresses many of the areas detailed in the End-User Computing standards. The package `usethis` provides functions to ease the setup of the package structure. To set up an empty package use:

```
usethis::create_package("pkgname")
```

Functions you wish to add to the package are saved in scripts in the `R` subfolder that has been created.

5.1.1 Documentation

Documentation comes in three flavours in an R package. The DESCRIPTION file details the package itself including author and dependencies; help files contains documentation for each function in the package; and vignettes are small articles describing how to use the functions in practice.

5.1.1.1 DESCRIPTION

The function `usethis::create_package()` creates the basic structure of the package and a placeholder DESCRIPTION file. You can then edit it manually or use other functions in the `usethis` package to update it.

For more detail, read the Package metadata chapter in the *R Packages* book.

5.1.1.2 Help Files

When writing a function in a package it is good practice to document it, so a user can see how it works with `help("function_name")`. Documenting a package is as straight forward as including a special commented section before the function definition. The package `roxygen2` is then used to convert those comments into help files.

For more detail, read the Object documentation chapter in the *R Packages* book.

5.1.1.3 Vignettes

When writing a package, it is also good practice to write a short article explaining how to use it. If the package is complex it may require a number articles describing different aspects. These short articles are called vignettes. Vignettes are written in markdown format and stored in a `vignettes` subfolder.

For more detail, read the Vignettes chapter in the *R Packages* book.

5.1.2 Testing

R packages provide a structure for specifying tests that allows you easily re-run all tests and determine whether something has broken. The package `testthat` provides functions for specifying tests and comparing the results with expectations.

For more detail, read the Testing chapter in the *R Packages* book.

5.2 Version Control

5.2.1 Git

5.2.2 GitHub Flow

5.3 Managing Package Dependencies

Chapter 6

Advanced Topics

6.1 Non-Standard Evaluation

Non-standard evaluation is a catch-all term that means they don't follow the usual R rules of evaluation. Instead, they capture the expression that you typed and evaluate it in a custom way.

6.1.1 Strings and quoting

Here's a simple example in the case of strings. If you want to return a personalised message, it is obvious the below function will not give you what you want:

```
greet <- function(name) {
  "Hello, name"
}

greet("Bob")
```

```
## [1] "Hello, name"
```

The function returns the string exactly as it is written because it is in quotes. With the `glue` package you can use non-standard evaluation to return what you want:

```
greet <- function(name) {
  glue("Hello, {name}")
}

greet("Bob")
```

```
## Hello, Bob
```

The `glue()` function "unquotes" the "name" and evaluates it as a variable.

6.1.2 Base R evaluation

6.1.2.1 Expressions and quoting

A similar approach can be taken to more general expressions in R. An **expression** is some R code has not been evaluated yet. It is captured and saved for later. In base R this can be achieved using the `quote()` function. Expressions can be one of four classes:

- "constants" (e.g. 4, TRUE).
- "names" (e.g. variable names), that have type "symbols".
- "calls" (e.g. unevaluated function calls), that have type "language".
- "pairlists" (not really used anymore).

Constants are not very interesting, so we will concentrate on names and calls. As with strings, we can capture the expression, rather than the evaluated result, using the `quote()` function:

```
quote(x) %>% class()
```

```
## [1] "name"
```

```
quote(mean(1:10)) %>% class()
```

```
## [1] "call"
```

To evaluate a quoted expression use the `eval()` function:

```
x <- 10

quote(x) %>% eval()
```

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

```
quote(mean(1:10)) %>% eval()
```

```
## [1] 5.5
```

6.1.2.2 Names and Environments

An **environment** is a container for variables, binding a set of names to a set of values. Every environment also has a parent environment, except the **empty** environment, which is the ultimate ancester of all environments. If a name is not found in the current environment R looks in its parent and so on through the generations. This is known as **lexical scoping**.

When using `eval()` as above the expression is evaluated in the current environment. However, the environment to evaluate the expression in can be set as a parameter to `eval()`.

Note: `data.frames` can be treated as environments containing the column names binded to the column vectors.

6.1.2.3 Calls

A **call** is a delayed evaluation of a function call. The function and the argument names are stored, but not evaluated. So you can change the values associated with the arguments before evaluation. In fact they need not be defined at creation time at all:

```
wait_for_it <- quote(x + y)

x <- 3
y <- 8

eval(wait_for_it)
```

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

6.1.2.4 Parsing and Deparsing

Parsing turns text into expressions and **deparsing** turns expressions into text. The `parse()` function converts text, but its default argument is a file connection, so we must use the text argument:

```
parse(text = "8 + 10") %>% eval()
```

```
## [1] 18
```

Deparse might be useful for returning information to the user:

```
friendly_eval <- function(expr) {
  str_c("The value of ", deparse(expr), " is ", eval(expr))
}

quote(8 + 10) %>% friendly_eval()
```

```
## [1] "The value of 8 + 10 is 18"
```

6.1.2.5 Functions and closures

When you execute a function it creates a new, temporary, environment. The named arguments to the function, plus any variables created within its body are stored in this environment. Thus it cannot effect the variables outside its scope.

The parent for the function's environment is the environment in which the function was created in, not the one in which it was executed in. Thus it has access to all the variables in the parent environment. This is known as **closure**, as it *encloses* the parent environment, and can be a powerful tool.

6.1.2.6 Substitute and promise

The `friendly_eval()` function above requires its argument to be quoted. It would be nice if we could write the expression directly as an argument. However, `quote()` makes a literal quote of its input, in this case `expr`. What we need is `substitute()`. This will lookup all the object names provided to it, and if it finds a value for that name, it will substitute the name for its value:

```
friendly_eval <- function(expr) {
  expr_sub <- substitute(expr)
  str_c("The value of ", deparse(expr_sub), " is ", eval(expr_sub))
}

friendly_eval(8 + 10)
```

```
## [1] "The value of 8 + 10 is 18"
```

The above function only works because in R function arguments are evaluated lazily – variables are not evaluated until they are used. R stores arguments as a **promise**, which contains the expression of an argument along with the value. `substitute()` capture the expression before it is evaluated.

6.1.2.7 Formula and overscoping

A **formula** is a *domain specific language* (DSL) to simplify expressing the relationship between variables. Just like functions, formulas enclose the environment they are created in. When a formula is evaluated later in a different environment, it can still access all the objects that lived in its original environment.

If an object exists in more than one accessible environment the enviroment the formula (or function) is evaluated in takes precidence. If the object is not found, R looks in the enclosed environment. This is known as **overscoping**, as the formula or function has scope beyond its execution environment.

6.1.3 Tidy Evaluation

To summarise the above, the following points are important to tidy evaluation:

1. `quote()` delays evaluation of an expression.
2. `eval()` evaluates an expression in the current (or a specified) environment.

Functions and formulas:

3. enclose the environment they were created in.
4. evaluate objects in their own, and enclosed, environments

Tidy evaluation has two new additions: **quasiquotation** and **quosures**.

6.1.3.1 Quasiquotation

Quasiquotation enables the user to evaluate parts of the expression right away, while quoting the rest. Say we have the expression `z - x + 4` and we know the value of `x` at the time of quoting, we can **unquote** `x` with the function `UQ()` or the operator `!!`. However, `quote()` will not work in this quasiquotation, we need to use the tidy evaluation equivalent, `expr()`:

```
x <- 10

expr(z - UQ(x) + 4)
```

```
## z - 10 + 4
```

```
expr(z - !!x + 4) %>% class()
```

```
## [1] "call"
```

The `expr()` function returns expressions, as does `quote()`, without any information on the environment it was created in. Some other useful functions for quasiquotation are:

- `UQS()` and the `!!!` operator unquote and splice their arguments.
- `enexpr()` takes an expression, looks up any symbols (names) within it and returns it unevaluated. It is equivalent to `substitute()`.
- `exprs()` captures multiple expressions and returns a list.

6.1.3.2 Quosures

As the name suggests, **quosures** are hybrids of quotes and closures. They are unevaluated expressions that enclose their creation environment. This is very similar to formulas and they are in fact implemented as one-sided formulas. Quosures are created with the `quo()` function:

```
quo(z - UQ(x) + 4)
```

```
## <quosure>
## expr: ^z - 10 + 4
## env:  global
```

```
quo(z - !!x + 4) %>% class()
```

```
## [1] "quosure" "formula"
```

The quosure equivalent of `substitute()` is `enquo()`. It takes a symbol referring to a function argument, quotes the R code that was supplied to this argument, captures the environment where the function was called (and thus where the R code was typed), and bundles them in a quosure.

Another useful function is `quos()`, it returns a list of quosures. You can supply several expressions directly, e.g. `quos(foo, bar)`, but more importantly you can also supply dots: `quos(...)`.

Note that quosures don't make a lower level distinction between calls and names. Every expression becomes a quosure.

To evaluate quosures we need a special function to implement the environment scoping properly. `rlang` provides such a function: `eval_tidy()`

```
x <- 10

quo(x) %>% eval_tidy()
```

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

```
quo(mean(1:10)) %>% eval_tidy()
```

```
## [1] 5.5
```

6.1.3.3 Parsing and Deparsing

`rlang` also provides functions for parsing and deparsing expressions. To turn a string into an expression use: - `parse_expr()`, which works in the same way as `parse(text = ...)`. - `parse_quosure()` to create a quosure.

To turn an expression into a string, `rlang` provides two functions: - `expr_text()` to turn an expression into a string. - `expr_label()` to produce a string that is more suited to use as a label.

```
friendly_eval <- function(expr) {
  str_c("The value of ", deparse(expr), " is ", eval(expr))
}

quote(8 + 10) %>% friendly_eval()
```

```
## [1] "The value of 8 + 10 is 18"
```

Chapter 7

Cheatsheets