

One Pager Data Science

Programming with Data: Introducing R

Graham.Williams@togaware.com

4th February 2019

Visit <https://essentials.togaware.com/onepagers> for more Essentials.

Data scientists write programs to ingest, fuse, clean, wrangle, visualise, analyse, and model data. Programming over data is a core task for the data scientist. We will primarily use R as our programming language (R Core Team, 2018a) and assume basic familiarity of R as may be gained from the many resources available on the Intranet, particularly from <https://cran.r-project.org/manuals.html>.

Programmers of data develop sentences or *code*. Code instructs a computer to perform specific tasks. A collection of sentences written in a language is what we might call a *program*. Through *programming by example* and *learning by immersion* we will share programs to deliver insights and outcomes from our data.

R is a large and complex ecosystem for the practice of data science. There is much freely available information on the Internet from which we can continually learn and borrow useful code segments that illustrate almost any task we might think of. We introduce here the basics for getting started with R, libraries and packages which extend the language, and the concepts of functions, commands, and operators. We also introduce the powerful concept of pipes as a foundation for building sophisticated data processing pipelines in R.

Through this guide new R commands will be introduced. The reader is encouraged to review the command's documentation and understand what the command does. Help is obtained using the `? command` as in:

```
?read.csv
```

Documentation on a particular package can be obtained using the `help=` option of `library()`:

```
library(help=rattle)
```

This chapter is intended to be hands on. To learn effectively you are encouraged to run R (e.g., RStudio or Emacs with ESS mode) and to replicate the commands. Check that output is the same and that you understand how it is generated. Try some variations. Explore.

Copyright © 2000-2019 Graham Williams. This work is licensed under a [Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License](https://creativecommons.org/licenses/by-nc-sa/4.0/) allowing this work to be copied, distributed, or adapted, with attribution and provided under the same license.



1 Tooling For R Programming

We will first need to have available to us the computer software that interprets our programs that we write as sentences in the R language. Usually we will install this software, called the R Interpreter, on our own computer and instructions for doing so are readily available on the Internet. The [R Project](https://www.r-project.org)¹ is a good place to start. Most GNU/Linux distributions freely provide R packages from their application stores and a commercial offering is available from Microsoft who also provide direct access within SQL Server and through Cortana Analytics on the Azure cloud. Now is a good time to install R on your own computer if that is required.

The open source and freely available [RStudio](https://www.rstudio.com)² software is recommended as a modern **integrated development environment** (IDE) for writing R programs. It can be installed on the common desktop or server computers running GNU/Linux, Microsoft/Windows, or Apple/MacOS. A browser version of RStudio also allows a desktop to access R running on a back-end cloud server.

The server version of RStudio runs on a remote computer (e.g., a server in the cloud) with the graphical interface presented within a web browser on our own desktop. The interface is much the same as the desktop version but all of the commands are run on the server rather than on our own desktop. Such a setup is useful when we require a powerful server on which to analyse very large datasets. We can then control the analyses from our own desktop with the results sent from the server back to our desktop whilst all the computation is performed on the server itself which may be a considerably more powerful computer than our usually less capable personal computers.

¹<https://www.r-project.org>

²<https://www.rstudio.com>

2 Introducing RStudio

The RStudio interface includes an **R Console** on the left through which we directly communicate with the R software. The window pane on the top right provides access to the **Environment** within which R is operating. This includes data and datasets that are defined as we interact with R. Initially it is empty. A **History** of all the commands we have asked R to run is available on another tab within the top right pane.

The bottom right pane provides direct access to an extensive collection of **Help**. On another tab within this same pane we can access the **Files** on our hard disk. Other tabs provide access to **Plots**, **Packages**, and a **Viewer** to view documents that we might generate.

In the **Help** tab displayed in the bottom right pane of Figure 1 we can see numerous links to R documentation. The Manual titled *An Introduction to R* within the **Help** tab is a good place to start if you are unfamiliar with R. There are also links to learning more about RStudio and these are recommended if you are new to RStudio.

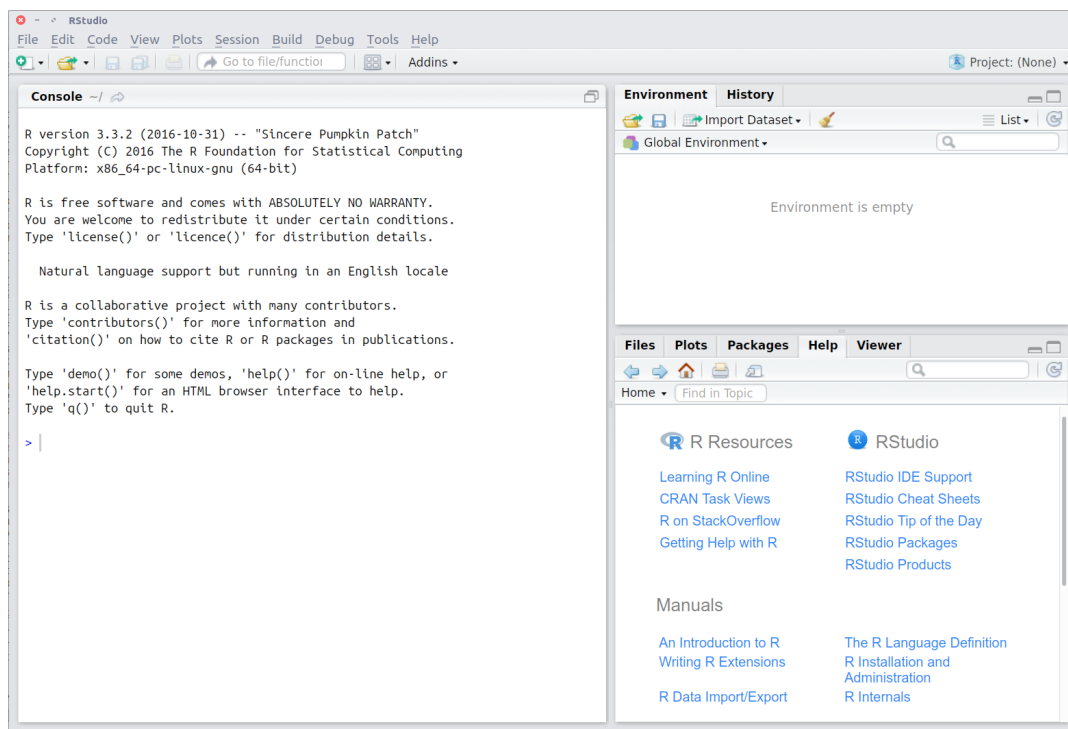


Figure 1: The initial layout of the RStudio window panes showing the **Console** pane on the left with the **Environment** and **Help** panes on the right.

3 Editing Code

Often we will be interacting with R by writing code and sending that code to the R Interpreter so that it can be run (locally or remotely). It is always good practice to store this code into a file so that we have a record of what we have done and are able to replicate the work at a later time. Such a file is called an R Script.

We can create a new R Script by clicking on the first icon on the RStudio toolbar and choosing the first item in the resulting menu as in Figure 2. A keyboard shortcut is also available to do this: **Ctrl+Shift+N** (hold the **Ctrl** and the **Shift** keys down and press the **N** key).

A sophisticated file editor is presented as the top left pane within RStudio. The tab will initially be named **Untitled1** until we actually save the script to a file on the disk. When we do so we will be asked to provide a name for the file.

The editor is where we write our R code and compose the programs that instruct the computer to perform particular tasks. The editor provides numerous features that are expected in a modern program editor. These include syntax colouring, automatic indentation to improve layout, automatic command completion, interactive command documentation, and the ability to send specific commands to the R Console to have them run by the R Interpreter.

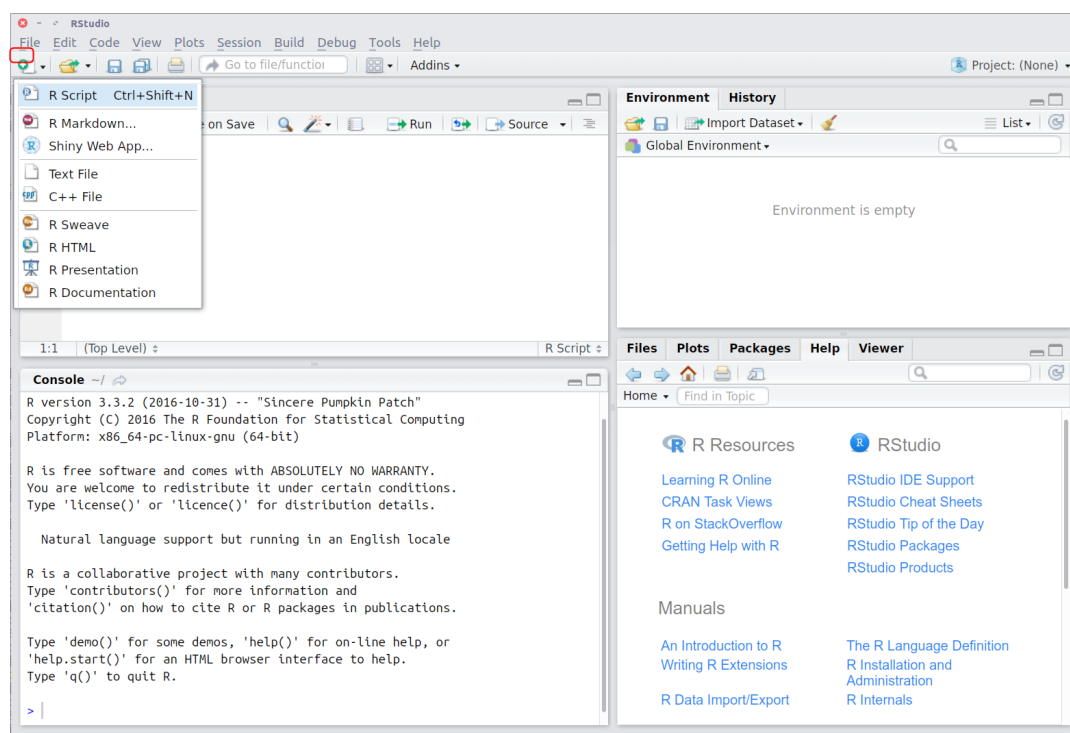


Figure 2: Ready to edit R scripts in RStudio using the **Editor** pane on the top left created when we choose to add a new R Script from the toolbar menu displayed by clicking the icon highlighted in red.

4 Executing Code

To send R commands to the R Console we can use the appropriate toolbar buttons. One line or a highlighted region can be sent using the Run button. Having opened a new R script file we can enter the commands below. The four commands make up a program which begins with an `install.packages()` to ensure we have installed two requisite software packages for R. The second and third commands use `library()` to load those packages into the current R session. The fourth command produces a plot of the `weatherAUS` dataset. This dataset contains observations of weather related variables across almost 50 weather stations in Australia over multiple years.

```
install.packages(c("ggplot2", "rattle"))
library(ggplot2)
library(rattle)
qplot(data=weatherAUS, x=MinTemp, y=MaxTemp)
```

`qplot()` allows us to quickly code a plot. Using the `weatherAUS` dataset we plot the minimum daily temperature (`MinTemp`) on the x-axis against the maximum daily temperature (`MaxTemp`) on the y-axis. The resulting plot is a **scatter plot** as we see the plot in the lower right pane.

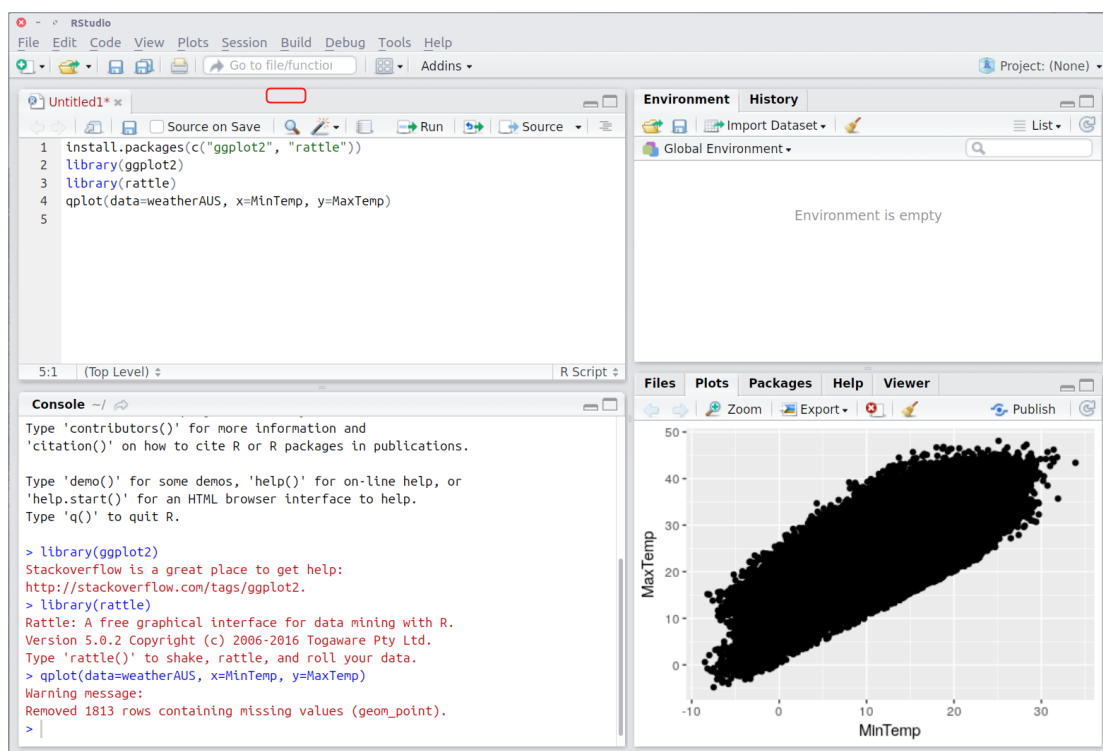


Figure 3: Running R commands in RStudio. The R programming code is written into a file using the editor in the top left pane. With the cursor on the line containing the code we click the Run button to pass the code on to the R Console to have it run by the R Interpreter to produce the plot we see in the bottom right pane.

5 RStudio Review

Figure 3 shows the RStudio editor as it appears after we have typed the above commands into the **R Script** file in the top left pane. We have sent the commands to the **R Console** to have it run by R. We have done this by ensuring the cursor within the **R Script** editor is on the same line as the command to be run and then clicking the **Run** button. We will notice that the command is sent to the **R Console** in the bottom left pane and the cursor advances to the next line within the **R Script** editor.

After each command is run any text output by the command is displayed in the **R Console** whilst graphic output is displayed in the **Plots** tab of the bottom right pane.

This is now our first program in R. We can now provide our first observations of the data. It is not too hard to see from the plot that there appears to be quite a strong relationship between the minimum temperature and the maximum temperature: with higher values of the minimum temperature recorded on any particular day we see higher values of the maximum temperature. There is also a clear lower boundary that might suggest, as logic would dictate, that the maximum temperature can not be less than the minimum temperature. If we were to observe data points below this line then we would begin to explore issues with the quality of the data.

As data scientists we have begun our observation and understanding of the data, taking our first steps toward immersing ourselves in and thereby beginning to understand the data.

6 Packages and Libraries

The power of the R ecosystem comes from the ability to extend the language by its nature as **open source software**. Anyone is able to contribute to the R ecosystem and by following stringent guidelines they can have their contributions included in the comprehensive R archive network, abbreviated as CRAN³. Such contributions are collected by an author into what is called a *package*. Over the decades many researchers and developers have contributed very many *packages* to CRAN.

A **package** is how R collects together *commands* for a specific collection of tasks. A **command** is a verb in the computer language used to tell the computer to do something. There are over 13,000 packages available for R from almost as many different authors. Hence there are very many verbs available to build our sentences to command R appropriately. With so many packages there is bound to be a package or two covering essentially any kind of processing we could imagine though we will also find packages offering the same or similar commands (verbs) perhaps even with very different meanings.

Beginning with Chapter 7 we will list at the beginning of each chapter the R *packages* that are required for us to be able to replicate the examples presented in that chapter. Packages are installed from the Internet (from the securely managed CRAN⁴ package repository) into a local *library* on our own computer. A **library** is a folder on our computer's storage which contains sub-folders corresponding to each of the installed packages.

To install a package from the Internet we can use the command `install.packages()` and provide to it as an *argument* the name of the package to install. The package name is provided as a **string** of **characters** within quotes and supplied as the `pkgs=` *argument* to the command as in the following code. Here we choose to install a package called `dplyr`—a very useful package for data manipulations.

```
# Install a package from a CRAN repository.  
  
install.packages(pkgs="dplyr")
```

Once a package is installed we can access the commands provided by that package by prefixing the command name with the package name as in `ggplot2::qplot()`. This is to say that `qplot()` is provided by the `ggplot2` (Wickham *et al.*, 2018a) package.

³CRAN: <https://cran.r-project.org>.

⁴<https://cran.r-project.org>

7 A Glimpse of the Dataset

Another example of a useful command that we will find ourselves using often is the `glimpse()` command from the `dplyr` (Wickham *et al.*, 2018b) package. This command can be accessed in the R console as `dplyr::glimpse()` once the `dplyr` package has been installed. This particular command accepts an *argument* `x=` which names the dataset we wish to glimpse. In the following R example we use the `weatherAUS` dataset from the `rattle` (Williams, 2018) package.

```
# Review the dataset.

dplyr::glimpse(x=rattle::weatherAUS)

## Observations: 145,463
## Variables: 24
## $ Date      <date> 2008-12-01, 2008-12-02, 2008-12-03, 2008-12-04, 200...
## $ Location  <fct> Albury, Albury, Albury, Albury, Albury, Albury, Albu...
## $ MinTemp   <dbl> 13.4, 7.4, 12.9, 9.2, 17.5, 14.6, 14.3, 7.7, 9.7, 13...
## $ MaxTemp   <dbl> 22.9, 25.1, 25.7, 28.0, 32.3, 29.7, 25.0, 26.7, 31.9...
....
```

We can see here the command that was run and the output from running that command. As a convention used in this book the output from running R commands is prefixed with “## ”. The “#” introduces a comment in an R script file and tells R to ignore everything that follows on that line. We use the “## ” convention throughout the book to clearly identify output produced by R. When we run these commands ourselves in R this prefix is not displayed.

Long lines of output are also truncated for our presentation here. The ... at the end of the lines and the at the end of the output indicate that the output has been truncated for the sake of keeping our example output to a minimum.

8 Attaching a Package

We can *attach* a package to our *current* R session from our local *library* for added convenience. This will make the command available during this specific R session without the requirement to specify the package name each time we use the command. Attaching a package tells the R software to look within that package (and then to look within any other attached packages) when it needs to find out what the command should do (the *definition* of the command).

We can see which packages are currently attached using `base::search()`. The order in which they are listed here corresponds to the order in which R searches for the definition of a command.

```
base::search()

## [1] ".GlobalEnv"          "package:stats"
## [3] "package:graphics"    "package:grDevices"
## [5] "package:utils"        "package:datasets"
## [7] "package:methods"     "Autoloads"
## [9] "package:base"
```

Notice that a collection of *packages* are installed by default among a couple of other special objects (`.GlobalEnv` and `Autoloads`).

A package is attached using the `base::library()` command. The `base::library()` command takes an argument to identify the `package=` we wish to *attach*.

```
# Load packages from the local library into the R session.

library(package=dplyr)
library(package=rattle)
```

Running these two commands will affect the search path by placing these packages early within the path.

```
base::search()

## [1] ".GlobalEnv"          "package:rattle"
## [3] "package:dplyr"        "package:stats"
## [5] "package:graphics"    "package:grDevices"
## [7] "package:utils"        "package:datasets"
## [9] "package:methods"     "Autoloads"
## [11] "package:base"
```

9 Simplifying Commands

By attaching the `dplyr` package we can drop the package name prefix for any commands from the package. Similarly by attaching `rattle` we can drop the package name prefix from the name of the dataset. Our previous `dplyr::glimpse()` command can be simplified.

```
# Review the dataset.

glimpse(x=weatherAUS)

## Observations: 145,463
## Variables: 24
## $ Date      <date> 2008-12-01, 2008-12-02, 2008-12-03, 2008-12-04, 200...
## $ Location  <fct> Albury, Albury, Albury, Albury, Albury, Albury, Albu...
## $ MinTemp   <dbl> 13.4, 7.4, 12.9, 9.2, 17.5, 14.6, 14.3, 7.7, 9.7, 13...
## $ MaxTemp   <dbl> 22.9, 25.1, 25.7, 28.0, 32.3, 29.7, 25.0, 26.7, 31.9...
....
```

We can actually simplify this a little more. Often for a command we don't have to explicitly name all of the arguments. In the following example we drop the `package=` and the `x=` arguments as the commands themselves know what to expect implicitly.

```
# Load packages from the local library into the R session.

library(dplyr)
library(rattle)

# Review the dataset.

glimpse(weatherAUS)

## Observations: 145,463
## Variables: 24
## $ Date      <date> 2008-12-01, 2008-12-02, 2008-12-03, 2008-12-04, 200...
## $ Location  <fct> Albury, Albury, Albury, Albury, Albury, Albury, Albu...
## $ MinTemp   <dbl> 13.4, 7.4, 12.9, 9.2, 17.5, 14.6, 14.3, 7.7, 9.7, 13...
## $ MaxTemp   <dbl> 22.9, 25.1, 25.7, 28.0, 32.3, 29.7, 25.0, 26.7, 31.9...
....
```

A number of packages are automatically attached when R starts. The first `base::search()` command above returned a vector of packages and since we had yet to attach any further packages those listed are the ones automatically attached. One of those is the `base` ([R Core Team, 2018a](#)) package which provides the `base::library()` command.

10 Working with the Library

To summarise then, when we interact with R we can usually drop the package prefix for those commands that can be found in one (or more) of the attached packages. Throughout the running text in this book we retain the package prefix to clarify where each command comes from. However, within the code we tend to drop the package name prefix.

A prefix can still be useful in larger programs to ensure we are using the correct command and to communicate to the human reader where the command comes from. Some packages do implement commands with the same names as commands defined differently and found in other packages and the prefix notation is then useful to specify which command we are referring to.

Each of the following chapters will begin with a list of *packages* to *attach* from the *library* into the R session. Below is an example of attaching five common packages for our work. Attaching the listed packages will allow the examples presented within the chapter to be replicated. In the code below take note of the use of the hash (#) to introduce a comment which is ignored by R—R will not attempt to understand the comments as commands. Comments are there to assist the human reader in understanding our programs.

```
# Load packages required for this script.  
  
library(rattle)    # The weatherAUS dataset and normVarNames().  
library(readr)     # Efficient reading of CSV data.  
library(dplyr)     # Data wrangling and glimpse().  
library(ggplot2)   # Visualise data.  
library(magrittr)  # Pipes %>%, %<>%, %T>%, %$%.
```

In starting up an R session (for example, by starting up RStudio) we can enter the above `library()` commands into an R script file created using the **New R Script File** menu option in RStudio and then ask RStudio to **Run** the commands. RStudio will send each command to the **R Console** which sends the command on to the R software. It is the R software that then runs the commands. If R responds that the package is not available then the package will need to be installed, which we can do from RStudio's **Tools** menu or by directly using `utils::install.packages()` as we saw above.

11 Functions

commands in R instruct the R software to perform particular actions. Formally commands are actually *functions* and we generally use this term to convey the mathematical concept of a function. A function simply takes a set of inputs and produces an output. R provides a wealth of *functions*, and we will generally think of each one as either a *function*, a *command* or an *operator*.

All R *functions* take arguments and return values. When we run a function (as distinct from what we will identify shortly as a *command*) we are interested in the value that it returns. Below is a simple example where we `base::sum()` two numbers.⁵

```
# Add two numbers.  
  
sum(1, 2)  
## [1] 3
```

As previously we precede the output from the function by the double hash (`##`). We will however not see the double hash in the R **Console** when we run the functions ourselves. Instead, the output we see will begin with the `[1]` which indicates that the returned value is a *vector* which starts counting from index 1. A *vector* is a collection of items which we can access through a sequential index—in this case the vector has only a single item.

We can store the result value from running the function (the value being a vector of length 1 containing just the item 3) into a *variable*. A *variable* is a name that we can use to refer to some location in the computer's memory where we store data while our programs are running. To store data in the computer's memory so that we can refer to that data by the specific variable name we use the *assignment operator* `base::<-`.

```
# Add two numbers and assign the result into a variable.  
  
v <- sum(1, 2)
```

⁵It is a good idea to replicate all of the examples presented here in R. Simply open an R script file to edit and type the text `sum(1, 2)` using the editor in RStudio. Then instruct RStudio to Run the command in the R Console.

12 Accessing Stored Values

We can access the value stored in the variable (or actually stored in the computer's memory that the variable name refers to) simply by requesting through the R Console that R “run” the command `v`. We do so below with a line containing just `v`. In fact the R software actually runs the `base::print()` function to display the contents of the computer's memory that `v` refers to—this is a convenience that the R software provides for us.

```
# Print the value stored in a variable.

v
## [1] 3
print(v)
## [1] 3
```

By indexing the variable with `[1]` we can ask for the first item referred to by `v`. Noting that the variable `v` is a vector with only a single item when we try to index a second item we get an `NA` (meaning not available or a missing value).

```
# Access a particular value from a vector of data.

v[1]
## [1] 3
v[2]
## [1] NA
```

13 Commands

For *commands* (rather than *functions*) we are generally more interested in the actions or side-effects that are performed by the command rather than the value returned by the command. For example, the `base::library()` command is run to attach a package from the library (the action) and consequently modifies the search path that R uses to find commands (the side-effect).

```
# Load package from the local library into the R session.  
  
library(rattle)
```

A *command* is also a function and does in fact return a value. In the above example we do not see any value printed. Some functions in R are implemented to return values invisibly. This is the case for `base::library()`. We can ask R to print the value when the returned result is invisible using the function `base::print()`.

```
# Demonstrate that library() returns a value invisibly.  
  
print(library(rattle))  
## [1] "readr"      "extrafont" "Hmisc"      "Formula"    "survival"   "lattice"  
## [7] "magrittr"   "xtable"    "rattle"     "ggplot2"    "tidyr"      "stringr"  
## [13] "dplyr"      "knitr"     "stats"      "graphics"   "grDevices"  "utils"  
## [19] "datasets"  "methods"   "base"
```

We see that the value returned by `base::library()` is a vector of character strings. Each character string names a package that has been attached during this session of R. Notice that the vector begins with item [1] and then item [5] continues on the second line, and so on. We can save the resulting vector into a variable and then index the variable to identify packages at specific locations within the vector.

```
# Load a package and save the value returned.  
  
l <- library(rattle)  
  
# Review one of the returned values.  
  
l[7]  
## [1] "magrittr"
```

14 Operators

The third type of function is the *operator*. We use **operators** in the flow of an expression. Technically they are called *infix* operators.

In the following example we use the `base::+` infix operator.

```
# Add two numbers using an infix operator.  
  
1 + 2  
## [1] 3
```

Internally R actually converts the operator into a functional prefix form to be run in the same way as any other function.

```
# Add two numbers using the equivalent functional form.  
  
`+`(1, 2)  
## [1] 3
```

Note that R also supports using quotes and double quotes instead of the backtick we used in the code above. The backtick is preferred as per page 142 of [Wickham \(2014\)](#).

The key thing to remember is that all commands and operators are functions and we instruct R to take action by running functions.

15 Pipes

A function (Section 11) performs some action on some input and returns some output. We can think of functions as the verbs of the language—the action words of our sentences.

As learn many new functions useful for the data scientist we will construct longer sentences that string together a sequence of verbs to deliver the results. This is a powerful programming concept, combining dedicated and well designed and implemented functions, each function focused on achieving a specific outcome.

To combine single focus functions into more complex operations we can use *pipes*. Pipes are a powerful concept for combining functions. The concept will be familiar to most who have used Unix and Linux. The idea is to pass the output of one function on to another function as its input. Each function does one task and aims to do that task very well, very accurately, and very simply from a user's point of view. We can then pipe together many such specialist functions to deliver very complex and quite sophisticated data transformations in an easily accessible manner. Pipes were introduced in R through the `magrittr` (Bache and Wickham, 2014) package.

To illustrate the concept of pipes we will again use the `rattle::weatherAUS` dataset. We review the basic contents of the `rattle::weatherAUS` dataset by printing it.

```
# Review the dataset of weather observations.
```

```
weatherAUS
```

##	Date	Location	MinTemp	MaxTemp	Rainfall	Evaporation	Sunshine
## 1	2008-12-01	Albury	13.4	22.9	0.6	NA	NA
## 2	2008-12-02	Albury	7.4	25.1	0.0	NA	NA
....							

We might be interested in the distribution of specific numeric variables from the dataset. For that we will `dplyr::select()` a few numeric variables by *piping* the dataset into `dplyr::select()`.

```
# Select variables from the dataset.
```

```
weatherAUS %>%
  select(MinTemp, MaxTemp, Rainfall, Sunshine)
```

##	MinTemp	MaxTemp	Rainfall	Sunshine
## 1	13.4	22.9	0.6	NA
## 2	7.4	25.1	0.0	NA
## 3	12.9	25.7	0.0	NA
## 4	9.2	28.0	0.0	NA
## 5	17.5	32.3	1.0	NA
....				

Notice how `weatherAUS` by itself will list the whole of the dataset. Piping the whole dataset to the `dplyr::select()` function using the *pipe* operator `purrr::%>%` tells R to send the `rattle::weatherAUS` dataset on the left to the `dplyr::select()` function on the right. We provided `dplyr::select()` with a list of the variables we wish to select. The end result returned as the output of the pipeline is a subset of the original dataset containing just the named columns.

16 Multiple Pipes

Suppose we now want to produce a `base::summary()` of these numeric variables. We will *pipe* the dataset produced by `dplyr::select()` into `base::summary()`.

```
# Select variables from the dataset and summarise the result.

weatherAUS %>%
  select(MinTemp, MaxTemp, Rainfall, Sunshine) %>%
  summary()

##      MinTemp      MaxTemp      Rainfall      Sunshine
##  Min.   :-8.70   Min.   :-4.10   Min.    : 0.000   Min.    : 0.00
## 1st Qu.: 7.40   1st Qu.:17.90   1st Qu.: 0.000   1st Qu.: 4.90
##  Median :11.90   Median :22.50   Median : 0.000   Median : 8.40
##   Mean  :12.04   Mean  :23.14   Mean   : 2.298   Mean   : 7.61
## 3rd Qu.:16.70   3rd Qu.:28.20   3rd Qu.: 0.600   3rd Qu.:10.60
##   Max.   :33.90   Max.   :48.10   Max.    :371.000   Max.    :14.50
##
## ..
```

Perhaps we would like to summarise only those observations where there is more than a little rain on the day of the observation. To do so we will `stats::filter()` the observations.

```
# Select specific variables and observations from the dataset.

weatherAUS %>%
  select(MinTemp, MaxTemp, Rainfall, Sunshine) %>%
  filter(Rainfall >= 1)

##      MinTemp MaxTemp Rainfall Sunshine
## 1      17.5     32.3       1.0       NA
## 2      13.1     30.1       1.4       NA
## 3      15.9     21.7       2.2       NA
## 4      15.9     18.6     15.6       NA
## 5      12.6     21.0       3.6       NA
##
## ..
```

We can see that this sequence of functions operating on the original `rattle::weatherAUS` dataset returns a subset of that dataset where all observations have some rain.

17 Backward Assignment

We saw earlier in this chapter the assignment operator `base::<-` which is used to save a result into the computer's memory and give it a name that we can refer to later. We can use this operator to save the result of the sequence of operations from the pipe.

```
# Select variables/observations and save the result.

rainy.days <-
  weatherAUS %>%
    select(MinTemp, MaxTemp, Rainfall, Sunshine) %>%
    filter(Rainfall >= 1)
```

An alternative that makes sense within the pipe paradigm is to use the forward assignment operator `base::>` to save the resulting data into a variable in a logically consistent way.

```
# Demonstrate use of the forward assignment operator.

weatherAUS %>%
  select(MinTemp, MaxTemp, Rainfall, Sunshine) %>%
  filter(Rainfall >= 1) ->
  rainy.days
```

In this example we see that using the forward assignment operator tends to hide the actual variable into which the assignment is made. The important side effect of the series of commands, the assignment to `rainy.days`, could easily be missed. Compare that to the prior code.

Instead, when using the forward assignment, be sure to un-indent the final variable name

```
# Best to unindent the final assigned variable for clarity.

weatherAUS %>%
  select(MinTemp, MaxTemp, Rainfall, Sunshine) %>%
  filter(Rainfall >= 1) ->
rainy.days
```

18 Comparing Distributions

Continuing with our pipeline example, we might want a `base::summary()` of the dataset.

```
# Summarise subset of variables for observations with rainfall.

weatherAUS %>%
  select(MinTemp, MaxTemp, Rainfall, Sunshine) %>%
  filter(Rainfall >= 1) %>%
  summary()

##      MinTemp      MaxTemp      Rainfall      Sunshine
##  Min.   :-8.50   Min.   :-4.10   Min.    : 1.000   Min.    : 0.000
## 1st Qu.: 8.40   1st Qu.:15.50   1st Qu.: 2.200   1st Qu.: 2.300
##  Median :12.20   Median :19.30   Median : 4.800   Median : 5.500
##  Mean   :12.71   Mean   :20.13   Mean    : 9.714   Mean    : 5.345
## 3rd Qu.:17.10   3rd Qu.:24.30   3rd Qu.:11.000   3rd Qu.: 8.100
##   Max.   :28.30   Max.   :46.30   Max.    :371.000   Max.    :14.200
##
## ..
```

It could be useful to contrast this with a `base::summary()` of those observations where there was virtually no rain.

```
# Summarise observations with little or no rainfall.

weatherAUS %>%
  select(MinTemp, MaxTemp, Rainfall, Sunshine) %>%
  filter(Rainfall < 1) %>%
  summary()

##      MinTemp      MaxTemp      Rainfall      Sunshine
##  Min.   :-8.70   Min.   :-2.10   Min.    :0.00000   Min.    : 0.00
## 1st Qu.: 7.10   1st Qu.:18.90   1st Qu.:0.00000   1st Qu.: 6.10
##  Median :11.80   Median :23.60   Median :0.00000   Median : 9.30
##  Mean   :11.83   Mean   :24.08   Mean    :0.05936   Mean    : 8.33
## 3rd Qu.:16.60   3rd Qu.:29.10   3rd Qu.:0.00000   3rd Qu.:10.90
##   Max.   :33.90   Max.   :48.10   Max.    :0.90000   Max.    :14.50
##
## ..
```

Any number of functions can be included in a *pipeline* to achieve the results we desire. In the following chapters we will see many examples and some will string together ten or more functions. Each step along the way is of itself generally easily understandable. The power is in what we can achieve by stringing together many simple steps to produce something more complex.

19 Pipeline Syntactic Sugar

For the technically minded we note that what is actually happening here is that the syntax, or how we write the sentences, is changed by R from the normal embedded functional approach in order to increase the ease with which we can read the code. This is an important goal as we need to always keep in mind that we write our code for others (and ourselves later on) to read.

Below we see a pipeline version of a series of commands operating on the dataset and not how it is mapped by R into the functional construct. For many of us it will take quite a bit of effort to parse this traditional functional form of the expression so as to understand what it is doing. The pipeline alternative provides a clearer narrative

```
# Summarise observations with little or no rainfall.

weatherAUS %>%
  select(MinTemp, MaxTemp, Rainfall, Sunshine) %>%
  filter(Rainfall < 1) %>%
  summary()
```

##	MinTemp	MaxTemp	Rainfall	Sunshine
##	Min. : -8.70	Min. : -2.10	Min. : 0.00000	Min. : 0.00
##	1st Qu.: 7.10	1st Qu.: 18.90	1st Qu.: 0.00000	1st Qu.: 6.10
##	Median : 11.80	Median : 23.60	Median : 0.00000	Median : 9.30
##	Mean : 11.83	Mean : 24.08	Mean : 0.05936	Mean : 8.33
##	3rd Qu.: 16.60	3rd Qu.: 29.10	3rd Qu.: 0.00000	3rd Qu.: 10.90
##	Max. : 33.90	Max. : 48.10	Max. : 0.90000	Max. : 14.50
....				

```
# Functional form equivalent to the pipeline above.

summary(filter(select(weatherAUS,
                      MinTemp, MaxTemp, Rainfall, Sunshine),
            Rainfall < 1))
```

##	MinTemp	MaxTemp	Rainfall	Sunshine
##	Min. : -8.70	Min. : -2.10	Min. : 0.00000	Min. : 0.00
##	1st Qu.: 7.10	1st Qu.: 18.90	1st Qu.: 0.00000	1st Qu.: 6.10
##	Median : 11.80	Median : 23.60	Median : 0.00000	Median : 9.30
##	Mean : 11.83	Mean : 24.08	Mean : 0.05936	Mean : 8.33
##	3rd Qu.: 16.60	3rd Qu.: 29.10	3rd Qu.: 0.00000	3rd Qu.: 10.90
##	Max. : 33.90	Max. : 48.10	Max. : 0.90000	Max. : 14.50
....				

Anything that improves the readability of our code is useful. Computers are quite capable of doing the hard work of transforming a simpler sentence into this much more complex looking sentence.

20 Assignment Pipe

There are several variations of the pipe operator available. A particularly handy operator implemented by `magrittr` is the assignment pipe `magrittr::%<>%`. This operator should be the left most pipe of any sequence of pipes. In addition to piping the dataset on the left onto the function on the right the result coming out of the right-hand pipeline is piped back to the original dataset overwriting its original contents in memory with the results from the pipeline.

A simple example is to replace a dataset with the same dataset after removing some observations (rows) and variables (columns). In the example below we `stats::filter()` and `dplyr::select()` the dataset to reduce it to just those observations and variables of interest. The result is piped backwards to the original dataset and thus overwrites the original data (which may or may not be a good thing). We do this on a temporary copy of the dataset and use the `base::dim()` function to report on the dimensions (rows and columns) of the resulting datasets.

```
# Copy the dataset into the variable ds.

ds <- weatherAUS

# Report on the dimensions of the dataset.

dim(ds)
## [1] 145463      24

# Demonstrate an assignment pipeline.

ds %<>%
  filter(Rainfall==0) %>%
  select(MinTemp, MaxTemp, Sunshine)

# Confirm that the dataset has changed.

dim(ds)
## [1] 91879      3
```

Once again this is so-called *syntactic sugar*. The commands are effectively translated by the computer into the following code.

```
# Functional form equivalent to the pipeline above.

ds <- select(filter(weatherAUS, Rainfall==0),
             MinTemp, MaxTemp, Sunshine)
```

21 Tee Pipe

Another useful operation is the tee-pipe `magrittr::%T>%` which causes the command that follows to be run as a side-pipe whilst piping the same data into that command and also into the next following command. The output from the first command is ignored.

Note that a new operator `magrittr::%T>%` from `magrittr` has been introduced here to process the transformed dataset in two ways—once with `dplyr::glimpse()` and then with `base::summary()`. The so-called *tee* pipe will take the flow of the dataset coming out of the previous junction which is the `dplyr::select()` and pipe it in two directions, like a T pipe. The first flow proceeds to the junction where `dplyr::glimpse()` is run. The second flow proceeds directly to the following junction, without going via the `dplyr::glimpse()`, where `base::summary()` is then run. Any further pipes would flow the output of the `base::summary()` command onward.

One use case is to generate a summary of an intermediate dataset whilst continuing to process the dataset.

```
weatherAUS %>%
  select(Rainfall, MinTemp, MaxTemp, Sunshine) %>%
  filter(Rainfall==0) %T>%
  head() %>%
  select(MinTemp, MaxTemp, Sunshine) %>%
  summary()
```

##	MinTemp	MaxTemp	Sunshine
## Min.	:-8.70	Min. :-2.10	Min. : 0.00
## 1st Qu.:	7.30	1st Qu.:19.60	1st Qu.: 6.80
## Median :	12.00	Median :24.40	Median : 9.60
## Mean :	12.02	Mean :24.75	Mean : 8.69
## 3rd Qu.:	16.80	3rd Qu.:29.70	3rd Qu.:11.10
## Max. :	33.90	Max. :48.10	Max. :14.50
## NA's :	386	NA's :400	NA's :44629

A common use case is to `base::print()` the result of some data processing steps whilst continuing on to assign the dataset itself to a variable. We will often see the following example.

```
# Demonstrate usage of a tee-pipe.

no.rain <-
  weatherAUS %>%
  filter(Rainfall==0) %T>%
  print()
```

##	Date	Location	MinTemp	MaxTemp	Rainfall	Evaporation	Sunshine
## 1	2008-12-02	Albury	7.4	25.1	0	NA	NA
## 2	2008-12-03	Albury	12.9	25.7	0	NA	NA
## 3	2008-12-04	Albury	9.2	28.0	0	NA	NA
## 4	2008-12-07	Albury	14.3	25.0	0	NA	NA
## 5	2008-12-08	Albury	7.7	26.7	0	NA	NA
....							

22 Tee Pipe Alternatives

We could simply `base::print()` the value of the variable `no.rain` after the assignment or else enclose the assignment within round brackets but there is some elegance in including it explicitly as part of the single pipeline.

```
# Equivalent alternatives to the above tee-pipe example.
```

```
(
  weatherAUS %>%
    filter(Rainfall==0) ->
    no.rain
)
```

```
##           Date      Location MinTemp MaxTemp Rainfall Evaporation Sunshine
## 1  2008-12-02    Albury      7.4    25.1        0          NA          NA
## 2  2008-12-03    Albury     12.9    25.7        0          NA          NA
## 3  2008-12-04    Albury      9.2    28.0        0          NA          NA
## 4  2008-12-07    Albury     14.3    25.0        0          NA          NA
## 5  2008-12-08    Albury      7.7    26.7        0          NA          NA
....
```

```
print(no.rain)
```

```
##           Date      Location MinTemp MaxTemp Rainfall Evaporation Sunshine
## 1  2008-12-02    Albury      7.4    25.1        0          NA          NA
## 2  2008-12-03    Albury     12.9    25.7        0          NA          NA
## 3  2008-12-04    Albury      9.2    28.0        0          NA          NA
## 4  2008-12-07    Albury     14.3    25.0        0          NA          NA
## 5  2008-12-08    Albury      7.7    26.7        0          NA          NA
....
```

23 Tee Pipe Use Case: Load all CSV Files

Example of usage of the tee operator. Load all `.csv.gz` files in the data folder into a single data frame and include a message for each file loaded to monitor progress.

```
fpath <- "data"
files <- dir(fpath, "*.csv.gz")
ds <- data.frame()
for (i in seq_len(length(files)))
{
  ds <- fpath %>%
    file.path(files[i]) %T>%
    cat("\n") %>%
    readr::read_csv() %>%
    rbind(ds, .)
}
```

This can also be useful when combined with an `on the fly` function which is introduced with curly braces. We can use the global assignment operator to define variables that can be used later on in the pipeline. This example is a little contrived but illustrates its use. The `on the fly` function calculates the order of wind gust directions based on the maximum temperature for any day within each group defined by the wind direction. This is then used to order the wind directions and saving that order to a global variable `lvls`. That variable is then used to mutate the original data frame (note the use of the tee pipe) to reorder the levels of the `WindGustDir` variable. This is typically done within a pipeline that feeds into a plot where we want to reorder the levels so that there is some meaning to the order of the bars in a bar plot, for example.

```
# List the levels of a factor.

levels(weatherAUS$WindGustDir)

## [1] "N" "NNE" "NE" "ENE" "E" "ESE" "SE" "SSE" "S" "SSW" "SW" "WSW"
## [13] "W" "WNW" "NW" "NNW"

#

weatherAUS %>%
  filter(Rainfall>0) %T>%
  {
    lvls <- select(., WindGustDir, MaxTemp) %>%
      group_by(WindGustDir) %>%
      summarise(MaxMaxTemp=max(MaxTemp)) %>%
      arrange(MaxMaxTemp) %>%
      pull(WindGustDir)
  } %>%
  mutate(WindGustDir=factor(WindGustDir, levels=lvls)) %$$
  levels(WindGustDir)

## [1] "ESE" "SW" "ENE" "NNE" "N" "NE" "E" "SE" "SSE" "S" "SSW" "WSW"
## [13] "W" "WNW" "NW" "NNW"
```

24 Exposition Pipe

We also see here yet another useful pipe operation `%$%`. This is the exposition pipe which is also useful for commands that do not take a dataset as their argument but instead some columns from the dataset. It evaluates the following function within the context of the dataset passed to it, so that the variables of the dataset become available without the need to quote them

```
weatherAUS %>%  
  filter(Rainfall==0) %$%  
  cor(MinTemp, MaxTemp)  
## [1] NA
```

We might otherwise use with.

```
weatherAUS %>%  
  filter(Rainfall==0) %>%  
  with(cor(MinTemp, MaxTemp))  
## [1] NA
```

25 Pipe and Plot

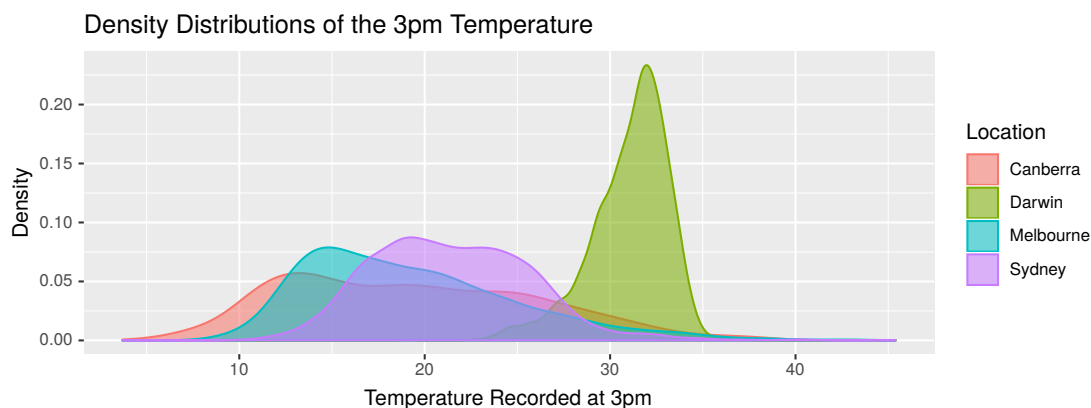
A common scenario for pipeline processing is to prepare data for plotting. Indeed, plotting itself has a pipeline type concept where we build a plot by adding layers to it.

Below the `rattle::weatherAUS` dataset is `stats::filter()`ed for observations from four Australian cities. We `stats::filter()` observations that have missing values for the variable `Temp3pm` using an embedded pipeline. The embedded pipeline pipes the `Temp3pm` data through the `base::is.na()` function which tests if the value is missing. These results are then piped to `magrittr::not()` which inverts the true/false values so that we include those that are not missing.

A plot is generated using `ggplot2::ggplot()` into which we pipe the processed dataset. We add a geometric layer using `ggplot2::geom_density()` which consists of a density plot with transparency specified through the `alpha=` argument. We also add a title and label the axes using `ggplot2::labs()`.

```
cities <- c("Canberra", "Darwin", "Melbourne", "Sydney")

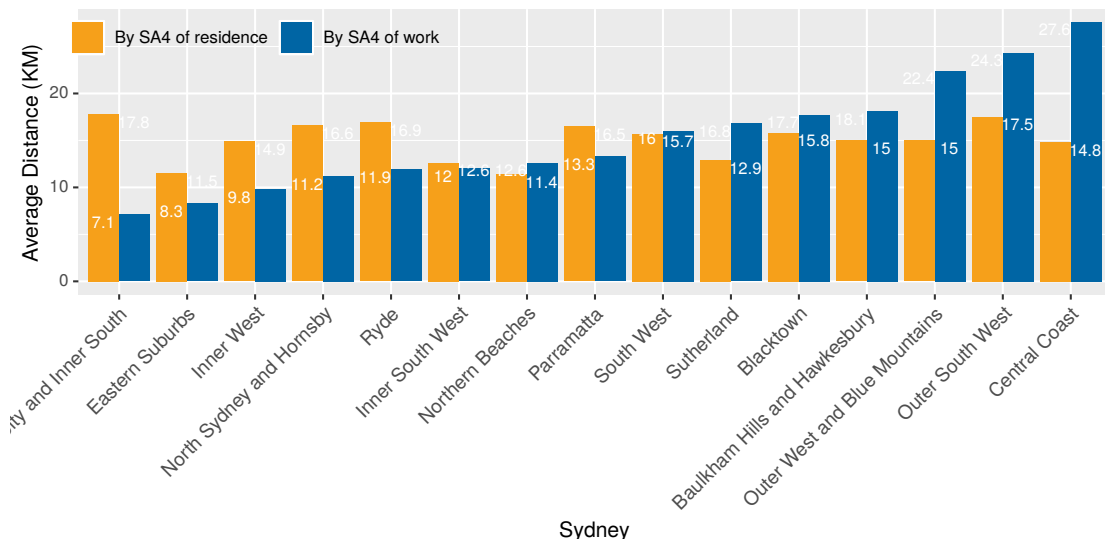
weatherAUS %>%
  filter(Location %in% cities) %>%
  filter(Temp3pm %>% is.na() %>% not()) %>%
  ggplot(aes(x=Temp3pm, colour=Location, fill=Location)) +
  geom_density(alpha=0.55) +
  labs(title = "Density Distributions of the 3pm Temperature",
       x     = "Temperature Recorded at 3pm",
       y     = "Density")
```



We now observe and tell a story from the plot. Our narrative will begin with the observation that Darwin has quite a different and warmer pattern of temperatures at 3pm than Canberra, Melbourne and Sydney.

26 Sophisticated Pipeline

```
avgdis %>%
  filter(GCCSA_CODE_2011=="1GSYD") %>%
  select(O_SA4name, O_Avg_KM, D_Avg_KM) %>%
  mutate(O_SA4name=str_replace(O_SA4name, "~Sydney - ", "")) %$%
  {
    nms <- O_SA4name[order(O_Avg_KM)] ; .
  } %>%
  mutate(O_SA4name=factor(O_SA4name, levels=nms)) %>%
  gather("OD", "Distance", -O_SA4name) %>%
  arrange(O_SA4name) %$%
  {
    lbls <- data.frame(x = sort(c(seq(0.75, 14.75), seq(1.2, 15.2))+0.02),
                      y = (Distance-0.75),
                      lbl = round(Distance,1)) ; .
  } %>%
  ggplot(aes(x=O_SA4name, y=Distance)) +
  geom_bar(stat="identity", position="dodge", aes(fill=OD)) +
  geom_text(data=lbls, aes(x=x, y=y, label=lbl), colour="white",
           size=3, hjust=0.5) +
  theme(axis.text.x=element_text(angle=45, hjust=1, size=10)) +
  theme(legend.position=c(.15, .9)) +
  scale_fill_manual(values=c(rgb(246, 160, 26, max=255),
                              rgb( 0, 101, 164, max=255)),
                  labels=c("By SA4 of residence", "By SA4 of work")) +
  theme(legend.title=element_text(colour="transparent"),
        legend.background=element_rect(fill="transparent"),
        legend.direction="horizontal") +
  labs(x="Sydney", y="Average Distance (KM)")
```



27 Pipeline Identity Operator

A handy trick when building a pipeline is to use what is effectively the identity operator. An identity operator simply takes the data being communicated through the pipeline, and without changing it passes it on to the next operator in the pipeline. An effective identity operator is constructed with the syntax `{.}`. In R terms this is a compound statement containing just the period whereby the period represents the data. Effectively this is an operator that passes the data through without processing it—an identity operator.

Why is this useful? Whilst we are building our pipeline, one line at a time, we will be wanting to put a pipe at the end of each line, but of course we can not do so if there is no following operator. Also, whilst debugging a pipeline, we may want to execute only a part of it, and so the identity operator is handy there too.

As a typical scenario we might be in the process of building a pipeline as here and find that including the `purrr::%>%` at the end of the line of the `dplyr::select()` operation:

```
weatherAUS %>%
  select(Rainfall, MinTemp, MaxTemp, Sunshine) %>%
  {.}
```

##		Rainfall	MinTemp	MaxTemp	Sunshine
## 1		0.6	13.4	22.9	NA
## 2		0.0	7.4	25.1	NA
## 3		0.0	12.9	25.7	NA
## 4		0.0	9.2	28.0	NA
## 5		1.0	17.5	32.3	NA
....					

We then add the next operation into the pipeline without having to modify any of the code already present:

```
weatherAUS %>%
  select(Rainfall, MinTemp, MaxTemp, Sunshine) %>%
  summary() %>%
  {.}
```

##		Rainfall	MinTemp	MaxTemp	Sunshine
##	Min. :	0.000	Min. : -8.70	Min. : -4.10	Min. : 0.00
##	1st Qu.: :	0.000	1st Qu.: 7.40	1st Qu.: 17.90	1st Qu.: 4.90
##	Median :	0.000	Median : 11.90	Median : 22.50	Median : 8.40
##	Mean :	2.298	Mean : 12.04	Mean : 23.14	Mean : 7.61
##	3rd Qu.: :	0.600	3rd Qu.: 16.70	3rd Qu.: 28.20	3rd Qu.: 10.60
##	Max. :	371.000	Max. : 33.90	Max. : 48.10	Max. : 14.50
##	NA's :	3218	NA's : 1545	NA's : 1335	NA's : 70820

And so on. Whilst it appears quite a minor convenience, over time as we build more pipelines, this becomes quite a handy trick.

28 Getting Help

A key skill of any programmer, including those programming over data, is the ability to identify how to access the full power of our tools. The breadth and depth of the capabilities of R means that there is much to learn around both the basics of R programming and the multitude of packages that support the data scientist.

R provides extensive in-built documentation with many introductory manuals and resources accessible through the RStudio Help tab. These are a good adjunct to our very brief introduction here to R. Further we can use RStudio's search facility for documentation on any action and we will find manual pages that provide an understanding of the purpose, arguments, and return value of *functions*, *commands*, and *operators*. We can also ask for help using the `utils::?` operator as in:

```
# Ask for documentation on using the library command.  
  
? library
```

The `utils` package which provides this operator is another package that is attached by default into R. Thus we can drop its prefix as we did here when we run the command in the R Console.

RStudio provides a search box which can be used to find specific topics within the vast collection of R documentation. Now would be a good time to check the documentation for the `base::library()` command.

Beginning with this chapter we will also list at the conclusion of each chapter the functions, commands, and operators introduced within the chapter together with a brief synopsis of the purpose. The following section reviews all of the functions introduced so far.

29 Command Summary

This chapter has introduced, demonstrated and described the following R packages, functions, commands, operators, and datasets:

- `<-` *operator from base*. The base R assignment operator. The expression on the right hand side is evaluated and the result is stored into the memory location identified on the left hand side. The left hand side is usually the name of a variable and has associated with it memory where it stores its values.
- `->` *operator from base*. The forward assignment operator. The expression on the left hand side is evaluated and the result is stored into the memory location identified on the right hand side. This assignment is not commonly used in R however it is a natural for use in a pipeline expression where the pipeline processes a dataset and we then store the resulting dataset into a variable.
- `?` *operator from utils*. Display the documentation provided for a command. The documentation will be displayed in the lower right pane of the RStudio window and often includes links to other documentation.
- `%<>%` *operator from magrittr*. The assignment pipe will pipe a dataset through to a function and the result is piped backwards to overwrite the original dataset.
- `%>%` *operator from magrittr*. Pipe a dataset through a series of functions. Each function takes the dataset as its input, processes the dataset in some way, and then returns a processed dataset which is piped onto the next function, if there is another pipe.
- `%T>%` *operator from magrittr*. The Tee-Pipe operator pipes a dataset through to the next function and also on to the following function. The results of the immediately following function are ignored and the pipeline then continues with the function following the immediately following function.
- `+` *operator from base*. Determine the sum of two arguments. The arguments are usually numbers but we have also seen an example where we add layers to a plot using this operator. This is an infix binary operator in that it is used between the two arguments that are to be added.
- `dim()` *function from base*. Returns a count of the number of rows and the number of columns in a dataset—the dimensions.
- `filter()` *function from stats*. Choose only those observations from the input dataset that satisfy a specified criteria. The output dataset consists of all of the same variables as the input dataset but a subset of the observations from the input dataset.
- `geom_density()` *function from ggplot2*. Add a density plot to a ggplot. A density plot shows the distribution of the different values of the variable over the observations in the dataset.
- `ggplot()` *function from ggplot2*. Initiate a new plot and return an object that represents that (currently empty) plot. In order to be able to display the plot we need to add layers. We have illustrated the layering in this chapter where we added density geometries, a title and labels. The `gg` here refers to the *grammar of graphics* which is implemented by `ggplot2`.
- `glimpse()` *command from dplyr*. Summarise a dataset. This is a convenient command for quickly obtaining a view of a potentially very large dataset.

install.packages() *command from utils*. Install a package into a library. Often the package will be downloaded from the Internet though it can also be a locally available file that was separately downloaded. The file will be an archive and this command unpacks the archive and stores the result in a folder that is referred to as a library.

is.na() *function from base*. Test if a value (or for that matter a vector) is “not available” (missing). Returns TRUE or FALSE appropriately or a vector of TRUE/FALSE if the argument is a vector.

labs() *function from ggplot2*. Change the text of the labels used for the x or y axis of a plot built using `ggplot2::ggplot()`.

library() *command from base*. Attach a package from an R library. The name of a package is provided as the argument to this command. That package is searched for within the available libraries on our local computer. When it is found the functions, commands, operators and datasets are made available to this R session.

not() *function from magrittr*. Invert the meaning of TRUE and FALSE, so that TRUE becomes FALSE and FALSE becomes TRUE. This is useful in a pipe as an alternative to the `base::!` operator.

print() *command from base*. Print a representation of the memory referred to by the named object supplied as the argument.

qplot() *command from ggplot2*. Quickly code a plot of the dataset. The generated plot is displayed as a side-effect of the command.

search() *function from base*. List the currently attached packages. This can be useful when we wish to see which packages have been attached to the current R session. The order in which the packages are listed is also the order in which R searches for a function. When the same function name is defined in two packages the first that is found is the one that will be used.

select() *function from dplyr*. Choose specified variables from a dataset. This function returns a smaller dataset than the input dataset consisting of all of the same observations as the input dataset but only of the selected variables.

sum() *function from base*. Determine the sum of two arguments (usually numbers). This is the traditional functional form of the `+` operator.

summary() *function from base*. From the input dataset this function generates a simple statistical summary of each variable.

weather *dataset from rattle*. The weather dataset contains daily observations of numerous weather variables from a weather station at the Canberra airport in Australia. It is a small dataset consisting of just 366 observations of 24 variables. It is used here to illustrate complex processing when size is not relevant for demonstrating the processing.

weatherAUS *dataset from rattle*. The weather dataset contains daily observations of numerous weather variables from nearly 50 weather stations across Australia. This is a moderately sized dataset consisting of 145,463 observations of 24 variables.

30 Exercises

Exercise 1 Exploring RStudio.

1. Install R and RStudio.
2. Start up RStudio and explore the menus, tool bars, and panes and tabs.
3. Explore the various options for learning about R.

Exercise 2 Getting help in RStudio.

1. Attach the `rattle` package in RStudio.
2. Obtain a list of the functions available from the `rattle` package.
3. Review the documentation for `rattle::normVarNames()` and then apply the function to the `base::names()` of the `rattle::weatherAUS` dataset from `rattle`.

Exercise 3 Interacting with RStudio.

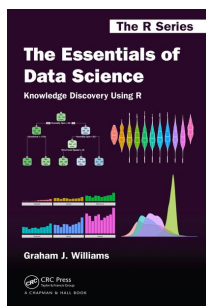
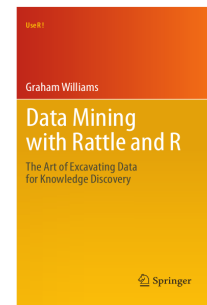
1. Create a new R script file and add a comment at the top with your name and today's data. Then save the file to disk.
2. By interacting with the R script file, attach the `rattle` package.
3. Plot the distribution of `WindSpeed9am` against `Humidity9am` from the `rattle::weatherAUS` dataset.
4. Filter the dataset by different locations and then review and compare the plots.
5. View and write a short report on the `rattle::weatherAUS` dataset. The report should describe the dataset, including a summary of the number of variables and observations. Describe each variable and summarise its distribution. Include some interesting plots.

Exercise 4 Using pipes.

1. Load the `rattle::weatherAUS` dataset into R.
2. Explore the rainfall for a variety of locations, generating different plots similar to the two examples provided in this chapter.
3. Write a short report on what you discovery including the R code you develop (using pipelines) and include sample plots that support your narrative.

31 Further Reading

The [Rattle](#) book (Williams, 2011), published by Springer, provides a comprehensive introduction to data mining and analytics using Rattle and R. It is available from [Amazon](#). Rattle provides a graphical user interface through which the user is able to load, explore, visualise, and transform data, and to build, evaluate, and export models. Through its Log tab it specifically aims to provide an R template which can be exported and serve as the starting point for further programming with data in R.



The [Essentials of Data Science](#) book (Williams, 2017), published by CRC Press, provides a comprehensive introduction to data science through programming with data using R. It is available from [Amazon](#). The book provides a template based approach to doing data science and knowledge discovery. Templates are provided for data wrangling and model building. These serve as generic starting points for programming with data, and are designed to require minimal effort to get started. Visit <https://essentials.togaware.com> for further guides and templates.

We list below other resources that augment the material we have presented in this chapter.

- The manual titled *An Introduction to R* is a good place to learn some of the basics of R. Follow the link provided within RStudio.
- <http://www.rstudio.org/> is the home of RStudio. The RStudio software can be downloaded from here as well as finding documentation and discussion forums. For a guide to using RStudio visit <https://support.rstudio.com>.
- Documentation within RStudio through the Help tab in the bottom right pane provides entry points to many of the standard resources for getting started with R and RStudio, as well as more advanced references.
- [Stack Overflow](#)⁶ is a great question and answer resource. There we find crowd sourced answers with crowd sourced ratings. It is always a good first port of call for any questions about using R.

⁶<https://stackoverflow.com>

32 References

Bache SM, Wickham H (2014). *magrittr: A Forward-Pipe Operator for R*. R package version 1.5, URL <https://CRAN.R-project.org/package=magrittr>.

R Core Team (2018a). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. URL <https://www.R-project.org/>.

R Core Team (2018b). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. URL <https://www.R-project.org/>.

Wickham H (2014). *Advanced R*. Chapman & Hall/CRC The R Series. Chapman & Hall.

Wickham H, Chang W, Henry L, Pedersen TL, Takahashi K, Wilke C, Woo K (2018a). *ggplot2: Create Elegant Data Visualisations Using the Grammar of Graphics*. R package version 3.1.0, URL <https://CRAN.R-project.org/package=ggplot2>.

Wickham H, François R, Henry L, Müller K (2018b). *dplyr: A Grammar of Data Manipulation*. R package version 0.7.8, URL <https://CRAN.R-project.org/package=dplyr>.

Williams GJ (2009). “Rattle: A Data Mining GUI for R.” *The R Journal*, **1**(2), 45–55. URL http://journal.r-project.org/archive/2009-2/RJournal_2009-2_Williams.pdf.

Williams GJ (2011). *Data Mining with Rattle and R: The art of excavating data for knowledge discovery*. Use R! Springer, New York.

Williams GJ (2017). *The Essentials of Data Science: Knowledge discovery using R*. The R Series. CRC Press.

Williams GJ (2018). *rattle: Graphical User Interface for Data Science in R*. R package version 5.2.5, URL <https://rattle.togaware.com/>.

This document, sourced from IntroRO.Rnw bitbucket revision 312, was processed by KnitR version 1.21 of 2018-12-10 23:00:03 UTC and took 26.6 seconds to process. It was generated by gjw on Ubuntu 18.04.1 LTS.

