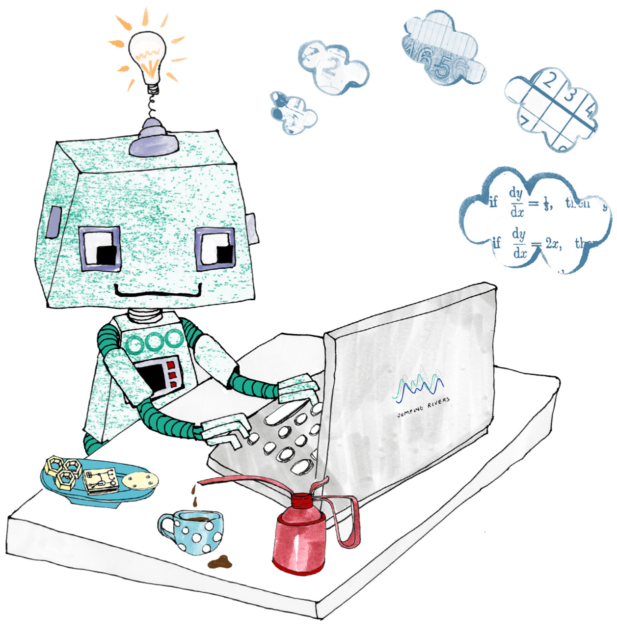
R BEST PRACTICES

jumping rivers

jumpingrivers.com

CREATING CLARITY WITH YOUR DATA.

Saving organisations like yours thousands of work hours.



Jumping Rivers is an analytics company whose passion is data and machine learning. We help our clients move from data storage to data insights. Our trainers and consultants come with over 40 years combined experience in R, Python, Stan, Scala and other programming languages.

Consultancy

- RStudio certified - one of only seven full-service partners.
- We have plans for managing all RStudio products. From on-demand support to full care packages.
- As an RStudio reseller, we can provide free consultancy.
- Experts in both data science and data engineering.
- Creation of bespoke dashboards and Shiny apps.
- Optimisation of your algorithms.

R Studio®

Training

- R/Python/Stan/Scala
- The Tidyverse
- Machine Learning and Tensorflow
- Dashboards with Shiny
- Statistical Modelling
- And many more!

jumping rivers
jumpingrivers.com
info@jumpingrivers.com
@jumping_uk

“GOD IS IN THE DETAILS.”

LUDWIG MIES VAN DER ROHE.

“BEWARE OF BUGS IN THE ABOVE CODE; I HAVE ONLY PROVED IT CORRECT, NOT TRIED IT.”

DONALD E. KNUTH.

Meaningful Names

1.1 What's in a name?

Names influence many aspects of programming. We name our variables, functions, scripts and packages. A good name can make the intention of your code clear and easy to read. A bad name can lead to confusion, misunderstanding and bugs. You might:

- forget which of `data1`, `data2`, ... contains the customer data you need;
- forget that `my_plot()` can make the box plot that you are currently writing code for;
- redefine `x` in multiple places.

I HATE WHEN I DON'T KNOW SOMEONE'S NAME, BUT IT'S BEEN LONG ENOUGH THAT IT'S TOO AWKWARD TO ASK.
DO YOU, RACHEL, TAKE THIS MAN...
AHA! RACHEL!

Figure 1.1: `https://xkcd.com/302/`

Thinking of a good name for your variable can be really difficult[1], so the first thing to consider is if you *need* to think of a good name at all! For example, often whilst working with {ggplot2} we might need to manipulate our data to get it into the correct format before plotting. If you don't need to save the modified data, it's possible to use the {magrittr} pipe `%>%` to pass your data directly to {ggplot2} without pausing to name in-between data.

[1] In this course, we refer to "variable"s in the programming sense—meaning a named container for, or reference to, some data in a program—rather than to statistical or mathematical variables.

```
library("dplyr")
library("ggplot2")
data(gapminder, package = "gapminder")

gapminder %>%
  filter(country == "United Kingdom") %>%
  mutate(pop_millions = pop / 1000000) %>%
  ggplot(aes(x = year, y = pop_millions)) +
  geom_point() +
  geom_line()
```

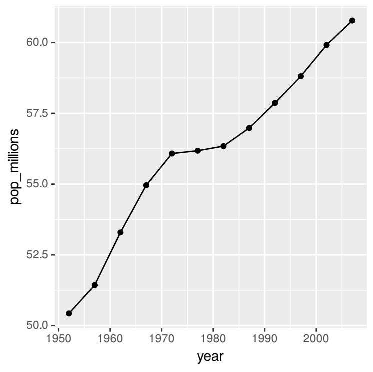


Figure 1.2: Population growth in the U.K. from 1952 to 2007.

rather than

```
gapminder_uk = filter(gapminder,
                      country == "United Kingdom")
gapminder_uk_mill = mutate(gapminder_uk,
                           pop_millions = pop / 1000000)

ggplot(gapminder_uk_mill,
       aes(x = year, y = pop_millions)) +
```

```r
  geom_point() +
  geom_line()
```

By using the `%>%` pipe to pass our data through to {ggplot2}, we can avoid pausing to name the intermediary datasets `gapminder_uk` and `gapminder_uk_millions`.

1.2 *Naming conventions*

When trying to decide a meaningful name for a variable, it is sometimes hard to think of a single-word description for its contents. For example, you've summarised the quarterly sales for a firm, should you store these as:

- `quarterlies` (will you make other quarterly summaries?); or
- `sales` (wouldn't that be more appropriate for the raw data?);

You could use several words to better contextualise your variable names. However, often in programming white space is not allowed in variable names. We could `putallthewordstogether` without spaces, but that wouldn't be very readable and might be confusing.[2]

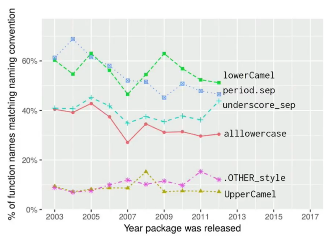


Figure 1.3: The current state of naming conventions in R—UseR 2017 Rasmus Bååth.

[2] Would `cutest()` be for estimating cuts, the result of running a CU test, or just really cute?

Instead, a number of naming conventions exist. The common ones are listed in Table 1.1. Different programming languages use different conventions. For example, Java and C# tend to use `PascalCase`, whilst R and Python tend to use `snake_case`. However it's not always so straight forward.

Table 1.1: Common naming conventions in programming.

Formatting	Name
two_words	snake_case
two.words	period.separated
twoWords	camelCase
TwoWords	PascalCase, UpperCamelCase
two-words	kebab-case

R is very liberal when it comes to names for objects and functions. This freedom is a great blessing and a great burden at the same time. Nobody is obliged to follow strict rules, so everybody who programs something in R can basically do as they please.[3] Many of the older functions in base R packages use the dot as a separator[4], whilst Bioconductor projects and {shiny} tend to use `camelCase` [5].

[3] Some R functions aren't even consistent across their own arguments! For example, `fisher.test()` has arguments `conf.int`, `hybridPars` and `Emin` — three different naming conventions!

[4] For example, `stats::t.test()` and `base::is.numeric()`.

[5] E.g., `shiny::updateNumericInput()`.

In R, you can only use letters, numbers, underscore characters (`_`) and dots (`.`). Although you *can* force R to accept other characters in names, you shouldn't, because these characters often have a special meaning. The most important thing is to decide on naming conventions for your project or your team and be *consistent* throughout your code. We'll see how to enforce these naming conventions in Chapter 3.

1.3 *Renaming*

Consistent style is not only about naming new objects. You may also want to rename elements in existing data frames. Useful R packages for this include {stringr} for general string manipulation, {snakecase} for reformatting strings to match a specific convention and {janitor} to clean column names.

One challenge in case conversion are odd looking "mixed cases". These might be introduced due to country codes or other abbreviations, which are usually written in upper case. The {snakecase} package provides a number of parsing options as well as the abbreviations argument to handle these issues.

```r
library("snakecase")
to_snake_case(c("KEYvar2", "IDTable1", "newUSElections"),
              abbreviations = c("ID", "KEY", "US"))
#> [1] "key_var_2"        "id_table_1"
#> [3] "new_us_elections"
```

The {janitor} package has a number of useful functions for cleaning up dataframes. In particular the `clean_names()` function tidies problematic column names by parsing letter cases and separators to a consistent format.[6] and appending numbers to duplicated column names.

Figure 1.4: `https://sfirke.github.io/janitor/`

[6] The default is `snake_case`, but other cases are available.

```r
library("janitor")
data(dirty_col_names, package = "jrBestPractices")
names(dirty_col_names)
#> [1] "firstName"   "SECOND.name" "Address"
#> "Address.1"
#> [5] "Post.Code"

dirty_col_names %>%
  clean_names() %>%
  names()
#> [1] "first_name"  "second_name" "address"
#> "address_1"
#> [5] "post_code"
```

The `clean_names()` function works well with `readr::read_csv()`.

If a variable or function name ever seems confusing, and you can think up a better name, feel free to change its name.[7] Don't be afraid to rename your colleagues variables either! However, if you spend time choosing a good name in the first place, you shouldn't need to spend time later renaming inappropriate names.

[7] Even the most experienced developers do this all the time—see the recent change from `tidyr::gather()` to `tidyr::pivot_longer()`. But make sure you update all the sites where that object is used.

1.4 *Write for humans not computers*

Computers can handle reading all sorts of awful code. Humans tend to be less tolerant. Clean code is code that a human can read easily. Using consistent naming conventions will help, but also try reading it out loud.

Ideally variable names and function names should be pronounceable, easy to spell, informative and easy to recall. Imagine you are talking through your code with your boss, or doing some pair programming[8] with a colleague. How would you read the following code? [9]

[8] See Chapter 5.

[9] GS 5734 is actually the name of a type of anti-viral medication.

```r
gs5734_2021.res_mock = data.frame(
  ptCls = sample(c("ctrl", "trt"), 7, replace = TRUE),
  pft = sample.int(15, 7, replace = TRUE)
)
```

Acronyms and abbreviations are sometimes tempting, as it allows for shorter names which are easier to type. However, these contractions can be difficult to say out loud and may not make sense to someone who hasn't used them before—especially if they speak a different language.

For example, which of these two scripts is easier to say, and easier to understand?

```r
library("dplyr")
defac = function(x) {
  as.numeric(as.character(x))
}
crm_summ = crimtab %>%
  as.data.frame() %>%
    mutate(
      fng = defac(Var1),
      ht = defac(Var2),
      n = Freq
    ) %>%
    select(fng, ht, n)
```

Or this one:

```r
extract_numeric_from_level = function(x) {
  as.numeric(as.character(x))
}
criminal_summary = crimtab %>%
  as.data.frame() %>%
  mutate(
    finger_length = extract_numeric_from_level(Var1),
    height = extract_numeric_from_level(Var2),
    count = Freq
  ) %>%
  select(finger_length, height, count)
```

Nonetheless, when an expressive contraction can be found, it can be very handy: a great example of this is the {lubridate} function `ymd()`, which converts date strings (`"2021-02-28"`) into `Date` objects.

All in all, try saying before naming.

1.5 *Name length*

Good names should ideally be short, as it keeps the code easier to read. However, in the quest to make a short name, you might end up losing understandability.

Which is easier to understand below?

```r
pick_random_file(path)
rndF(path)
f_rand(path)
sample_file(path)

get_todays_date()
today()

strip(strings)
trimws(strings)
```

```
str_trim(strings)
chomp(strings)
```

Another thing to consider in terms of name length is how *searchable* your name is. Let's say we need to find all instances of our variable which defines the date of an event. Which is easier to search for `d` or `start_date`?

If there is a trade off between a short variable or an understandable variable name, we would always opt for the longer, more interpretable variable name. It's better to be clear than short. And remember in most good IDEs[10] you can use tab to auto-complete long names. A good rule of thumb is that the length of a name should correspond to the size of its scope.

[10] Integrated Development Environment, a programme you use to write scripts, e.g. RStudio or Visual Studio.

1.6 *Function names*

When naming functions, there are some additional things to consider beyond the general advice above. Functions perform an action (e.g., computing, retrieving or saving something), and as such, their names should typically start with a verb: `create_graph()`, `update_db()`, `clean_data()`. But, any given action may have multiple synonyms. So, which of the following would be the most appropriate name for your data import function?

```
load_clients()
import_clients()
fetch_clients()
retrieve_clients()
```

Similarly, which of these has the best suffix?

```
load_clients()
load_client_dataset()
```

It doesn't really matter. What does matter is *consistency*:

- if you use `load_clients()` to import the data for your clients,
- ... then use `load_products()`, `load_suppliers()` (and so on) for importing other datasets,
- ... and use `filter_clients()`, `save_clients()` (and so on) for other functions pertaining to client-related datasets.

Having consistently-named functions makes it easier to find the right function when you need to use it.

Nonetheless, accurate naming is more important than consistent naming: if your function is called `check_url_format()` it should *only* check that the URL(s) that you pass to it have a valid format—it shouldn't check if a web page exists and it shouldn't check whether a set of email addresses are correctly formatted.

Cute names and in-jokes might seem a fun idea at the time, but when you look at your code later, you might struggle to work out the intention of the function. [11]

[11] Personally I can never remember what `recipes::juice()` does!

The `verb()` and `verb_noun()` function-naming convention is just a recommendation. R contains many examples of well-named functions that do not conform to this style which you could consider replicating in your own code. For example:

- `custom %op% erator` syntax:
  - A custom operator might make a neat shorthand
  - Compare `x %in% the_vector` to `contains(the_vector, x)` or `is_present(x, the_vector)`.
- `barenoun()`
  - A really non-specific verb may add little value to a function name, especially where a well-established value or property is returned
  - Compare `compute_mean()` / `get_length()` to `mean()` / `length()`.
- `prefix_verb()` and `prefix_verb_noun()`
  - When the verb you want to use is already in use,
  - or, you have a suite of functions that all apply to the same data type
  - For example, common prefixes are used in {stringr} (`str_*()`) and {forcats} (`fct_*()`) and indicate that the functions work on strings and factors.

1.7 *File names*

Files are another important object that we deal with in R. When creating or renaming files, these ideas may help ensure the future usability of those files:

- It can be helpful to store metadata as part of the filename e.g. `2016-05-alaska.csv`
- If you use manuscript location in figure filenames e.g `fig_3_a.png`—be aware things might change!
- Use filenames that are portable across operating systems:
  - Never use spaces in filenames, this aids portability to Linux and OS-X
  - Use lower-case names (since Windows directories are case-insensitive by default)
- Use consistent file name extensions (e.g., use `*.R`, `*.Rmd`, `*.jpeg` rather than `*.r`, `*.rmd`, `*.jpg` or `*.Jpeg`)

1.8 *Clutter*

To recap: when it comes to naming objects, there are three main forces in play. A name should:

- describe the purpose of the object;
- be as short as possible;
- be consistent with the other object-names in a project.

… but the first two qualities compete with each other.

Sometimes the path to creating concise, meaningful names involves removing clutter.

Unnecessary prefixes / suffixes

Suppose `outages_table_2021` is the name of a `data.frame` containing information about power outages in 2021 in your local area.

The name is self-explanatory, but is rather long. Consider each part of the name:

- `outages` : the table is about power-outages, so this part seems necessary
- `table` : we know (or can determine) that this is a `data.frame` , so including `table` in the name is superfluous.[12]
- `2021` : whether this is of value depends on context—is there an `outages_table_2020` in the same script? If not, maybe we can remove `2021` from the variable name. If so, we ought to keep the year in the name for now.

[12] This might even be misleading to those who have worked with the contingency table data-structure in R.

So, if part of a variable name is uninformative or is obvious from the context, that can usually be removed.

Clutter isn't always clutter

Don't always view prefixes or long object names as clutter—sometimes they indicate a missing abstraction.

Suppose multiple variables with a shared prefix were mentioned in a script: `dataset_raw` , `dataset_clean` , `dataset_train` , `dataset_test` . You could view the `dataset_` prefix as clutter and rename these variables (`raw` , `clean` , `train` , `test`). This isn't always wise. Shared prefixes (or suffixes) sometimes indicate when a set of variables have a shared context. Removing that prefix may take away an important mental cue and slow down your future coding. Here, rearranging those datasets into a larger dataset (e.g., by putting them in a list) that makes the shared context more explicit may have more value (though refactoring in this way[13] is out of scope for this chapter).

[13] Refactoring is the process of rewriting code without altering its behaviour.

Tangentially, if you have defined functions that have overly-long or highly-specific names, this might also indicate a missing abstraction. Rather than defining `read_outages_2020_data()` and `read_outages_2019_data()` you should aim to define `read_outages_data(year)` . Identifying when to generalise a function is rarely so simple, but a rule of thumb is "Three strikes and refactor".[14] That is, if you find yourself writing similar code multiple times, try to write a function that generalises that code. Again, this is beyond the scope of a chapter on renaming. But, you never know, the generalised function might even be easier to name.

[14] Fowler, Refactoring.

2 Modular Code

The most fundamental problem in computer science is problem decomposition. John Ousterhout, 2018, “A Philosophy of Software Design”.

Modularity, in addition to picking good names, is a key ingredient when trying to improve the readability of code. Modularity is about two main ideas, splitting your programming challenges into smaller components or tasks, and collecting together those components that have related purposes.

By separating a program up in this way you help yourself because a small, independent, task:

- is easier to describe (finding pre-existing solutions is simplified);
- has fewer moving parts and less to think about (solving it yourself is simplified);
- has solutions that are easier to verify (checking it with colleagues is simplified);
- may arise in several projects (so the solutions may be reusable);
- can be worked on by other members of a team (so a solution may arise faster).

Ultimately, when scanning through a script or report that has been nicely subdivided, the aim of the code is more apparent and easier to describe to others. So, before you start programming

- *Think:* What are the main datasets, tasks and outputs involved here? For those tasks, which pose the greatest challenges (and can they be split into smaller challenges)?
- *Draw:* Make a flowchart of a collection of the tasks—connect in any datasets that are required or produced
- *Discuss:* Has a colleague (or yourself), or a pre-existing R package solved the challenges involved here?

In R, the main approaches for separating-up parts of a program or project are through writing functions, scripts and packages.

2.1 *Write for people, not computers*

There are many ways to generate a particular value using code. Each of these expressions generate the integers between 1 and 10 (inclusive).

```r
1:10
seq_len(10)
c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
```

When you can see several different ways to write the same code, how should you proceed? You could choose the version of the code that:

- runs the fastest
- uses the least computational memory
- is the easiest to modify
- requires the least typing
- is the easiest to understand.

When faced with this choice, you should always aim to write code that is as easily understood as possible So, although optimising for readability might mean that your code sacrifices speed or memory, the benefits gained when you come back to read your code are vast.[1]

"*We are constantly reading old code as part of the effort to write new code.*" Robert C. Martin, 2009, "Clean Code".

[1] If you ever find it hard to follow some code, when re-reading it, then it might be worth investing some time to rewrite it. But be careful to ensure that your new program works the same afterwards.

Also, humans have a much smaller working memory than computers. Trying to hold lots of new information in our heads can be challenging. By chunking functions into smaller, digestible pieces, it's generally easier for others to follow the workflow.

2.2 *Don't Repeat Yourself (DRY)*

We often repeat ourselves in code. Sometimes this happens by accident: you might not realise some previously-written code exists that is very similar to your newly written code. Indeed, since two blocks of code can look different but compute the same expression, you might not even realise that your new code duplicates some old functionality. Then, sometimes it happens on purpose. A couple of lines here and there doesn't really matter, but when you find yourself copy and pasting substantial code blocks within a report, or between scripts or projects then something is amiss.

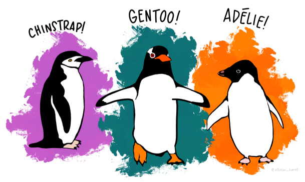


Figure 2.1: The Palmer Penguins. Artwork by @allison_horst.

The problems with duplicated code are that it makes code harder to read, more time consuming to maintain, and increases the chances of bugs. For example, let's say you're studying different species of penguins and want to compare the widths and lengths of their bills. You might want to create a scatter plot for each different penguin species. We can load the data from {palmerpenguins}.[2]

[2] The data used in this example comes from the {palmerpenguins} package. For simplicity, we've tweaked some of the data and added it to {jrBestPractices}.

```r
data(penguins_chinstrap, package = "jrBestPractices")
data(penguins_adelie, package = "jrBestPractices")
```

The following code converts the measurements from millimetres to centimetres, and then creates a scatter plot.

```r
library("ggplot2")
library("dplyr")
penguins_chinstrap %>%
  mutate(bill_length_cm = bill_length_mm / 10) %>%
  mutate(bill_depth_cm = bill_depth_mm / 10) %>%
  ggplot(aes(x = bill_length_cm, y = bill_depth_cm)) +
  geom_point() +
  labs(title = "Bill length and depth",
       subtitle = "Chinstrap Penguins")
```

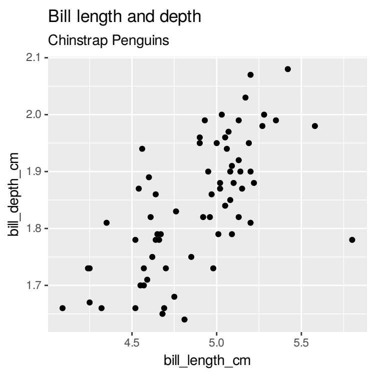


Figure 2.2: Bill length vs Bill depth.

If we wanted to create the same plot for the Adelie penguins, we could copy and paste all the code—changing the word Chinstrap to Adelie.

This is dangerous for a couple of reasons. We might remember to change the data set from `penguins_chinstrap` to `penguins_adelie` but forget to change the penguin species in the subtitle (or vice versa). We might have a typo in our original code (such as typing `goem_point()` instead of `geom_point()`) which gets copied across every time we paste our code. Finally, if we wanted to change the plot style, perhaps changing the colour of the points from black to green, we would need to change this in multiple places (and not make any typos!). Another issue with duplicating this code, is it will become much harder to read—and therefore harder to spot any mistakes.

Any time you find yourself using copy and paste, take a pause and consider if there is a way for you to not duplicate your code. The best course of action depends on how the code is being reused. When you find this kind of duplication, ask yourself which commands are common to the different code-blocks and which vary the most. In R, the main routes for deduplicating code are to introduce variables and to define functions.

If in the different code-blocks, you are using the same processing steps on several different datasets you may benefit from introducing a function to handle the processing pipeline. In the penguin example, we convert the measurements from cm to mm for all three datasets. We could move these two lines into a single function `convert_to_cm()` .

```
convert_to_cm = function(data) {
  data %>%
    mutate(bill_length_cm = bill_length_mm / 10) %>%
    mutate(bill_depth_cm = bill_depth_mm / 10)
}

penguins_adelie %>%
  convert_to_cm() %>%
  ggplot(aes(x = bill_length_cm, y = bill_depth_cm)) +
  geom_point() +
  labs(title = "Bill length and depth",
       subtitle = "Adelie Penguins")
```

If there are minor variations in your pipeline you can use variables to handle the variety. If we wanted to move the entire plotting code into a function, we would need to account for the fact that each plot requires a unique subtitle. We can do this with a function argument.

```
plot_bill_scatter = function(data, subtitle) {
  data %>%
    ggplot(aes(x = bill_length_cm, y = bill_depth_cm)) +
    geom_point() +
    labs(title = "Bill length and depth",
         subtitle = subtitle)
}

penguins_adelie %>%
  convert_to_cm() %>%
  plot_bill_scatter(subtitle = "Adelie Penguins")
```

2.3 *Writing good functions*

There are a number of guidelines that will help when designing and using functions.

Structure

A well written function should "Do one thing, do it well, and do it only". [3] If you have written a function `regress()` that performs linear regression, that function shouldn't also generate plots of the dataset.[4] When naming your function, if you cannot create a succinct name it may simply be that the function is doing too many things and should be split up.

[3] Error handling is one thing!

[4] You should also have a good reason not to just use the `lm()` function.

Ideally, function definitions should be small. To some, *small* might mean that the main body of the function can be read without scrolling on a laptop screen, to others it might mean that the function body contains only a handful of variable declarations and function calls. This may be difficult to achieve, but it is only a rule-of-thumb.

A reasonable upper limit, is 100 lines of code in a function.

More important is that a function should be readable. While reading a function, you may find there is a lot of low-level complexity in there that makes it hard to follow why the function is structured as it is (including things like validity checking). If so, consider moving that low-level work into a new function and calling it.

For example, the following function has boilerplate check code:

```r
get_age = function(age_days, units = "years") {
  if (is.character(age_days)) stop("Age must be a number")
  if (age_days < 0) stop("Negative age")
  if (age_days > 150 * 365) stop("A bit too old?")

  if (units == "years") {
    age = age_days / 365
  } else if (units == "days") {
    age = age_days
  } else if (units == "hours") {
    age = age_days * 24
  }
  return(age)
}
```

We could move the first three lines to

```r
check_age = function(age_days) {
  if (is.character(age_days)) stop("Age must be a number")
  if (age_days < 0) stop("Negative age")
  if (age_days > 150 * 365) stop("A bit too old?")
}
```

This would significantly simplify the function.

Contract

The *contract* defines what a function does (not how it does it) and the interface of that function (the data-types that it receives and returns, the parameter names). Fight the impulse to write the body of your function—think about the contract of that function first.

- If the function acts on numeric values, should it accept negative values?
- If it accepts data frames as input, does it need to know which columns of that data frame it should act upon?
- If you have written other functions with similar functionality, are the arguments named consistently, do they return consistent data-types?
- Should the output of the function be of the same structure as the input?
- If the user passes in invalid input, should the function throw an error?

Take a second to think about {dplyr}. One of the main reasons it is so successful is that the key functions all have the same input/output: `mutate()`, `summary()`, and `select()`. This makes it very easy to use.

For each argument that the user can pass into your function, there will be extra code to write and the resulting function will be more complicated and harder to test. So it simplifies your life if your functions have as few parameters as possible. But, if adding extra parameters increases the applicability of your function, reduces the need for extra functions, or reduces the burden on the user then the change can be justified.

The contract is very important. The minute someone else starts using your code, if you change something you have now given them more work.

No side-effects!

A pure function is one that has no side-effects and is self-contained. These are really useful properties that make it easier to reason about your code.

Side-effects include things like file-access, printing to the command line, setting global options or modifying values outside of the scope of the function. A function that is self-contained is one where, for any given input arguments, it always provides the same output. So a function that uses a random number generator[5], that depends on a global configuration option or other value that isn't passed into the function as an argument, would not be pure.[6]

[5] When you call a random number generator, you alter the variable `.Random.seed`.

[6] Other examples are plotting functions such as `ggplot()` and writing files to disks, `write_csv()`.

Because they neither influence, nor are influenced by global state, pure functions play nicely with the rest of your code.

You will, however, need to access files, databases, webpages and random numbers when programming with data, so it is hard to eliminate impure function calls from your code. But, when writing functions that do these things, if you are able to separate the impure behaviour away from calls to a pure function you'll benefit.

For example, suppose you have a function that reads some data and processes it. By separating the processing steps into a pure function, you benefit—you are no longer tied to the data-storage format and if you've already imported the data for some other reason you can pass that into the pure function. For example, if we move line 2 of this function

```r
f = function(path) {
  values = read.csv(path)
  do(some(thing(to(values))))
}
```

into a separate function

```
process_the_data = function(x) {
  do(some(thing(to(x))))
}
```

This makes it easier to move from a CSV file to a database to Dropbox.

A particularly dangerous, but relatively common, way that a function might have a side-effect is by loading a package dependency. Just don't do that.

Write generalisable code

If you find yourself writing a function that could *only* be used for a particular project, ask yourself if there is a way to make that function *slightly more* general. For example, if you were to write a function to import and clean-up data from the file `"./data/health.csv"`, which of the following would be better:

We use `file.path()` to ensure file-path portability. Notice that we have used the `file` argument to match `read_csv`.

```
import = function() {
  file = file.path("data", "health.csv")
  raw_data = readr::read_csv(file = file)
  clean_the_data(raw_data)
}

import = function(file = file.path("data", "health.csv")) {
  raw_data = readr::read_csv(file = file)
  clean_the_data(raw_data)
}
```

3 Code Style

3.1 Introduction

The trouble with proposing a series of programming best-practices is that any such guidelines are subjective. Some authors recommend that functions should be split up into really small units,[1] however, doing so might mean you end up with a huge number of functions, each of which does very little and may only be called from a single place. Other authors recommend against writing shallow functions, considering it better to write functions that do a lot (lots of–well structured–code) with very little (few parameters).[2]

[1] *… and however small that is, should be even smaller than that.* Robert C. Martin, 2009, Clean Code.

[2] John Ousterhout, 2018, A Philosophy of Software Design.

An area that is particularly subjective is code *style*. What do we mean by code style? There are the surface qualities:[3]

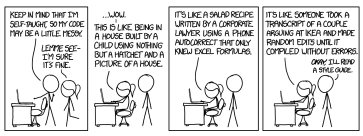
[3] As we'll discuss, these are still important.

- Should you `"double-quote"` or `'single-quote'` your strings?
- Should you indent using tabs or spaces, and if you indent, how much and where should you indent?
- Should you use `camelCase` or `snake_case` ?
- In R, should you assign variables with `<-` or `=` ?

If you are ever thinking about the above, always consider the rest of your team. Try to be consistent within your team.

Then there are the structural qualities:

- How to organise your script?
- Where should your functions be defined?
- Where should documentation live?
- When there are several ways to do the same thing (e.g., for-loops, vectorisation, map-style functional iterators, python's list-comprehensions) which should you choose?


KEEP IN MIND THAT I'M SELF-TAUGHT, SO MY CODE MAY BE A LITTLE MESSY.
LEMME SEE- I'M SURE IT'S FINE.
...WOW.
THIS IS LIKE BEING IN A HOUSE BUILT BY A CHILD USING NOTHING BUT A HATCHET AND A PICTURE OF A HOUSE.
IT'S LIKE A SALAD RECIPE WRITTEN BY A CORPORATE LAWYER USING A PHONE AUTOCORRECT THAT ONLY KNEW EXCEL FORMULAS.
IT'S LIKE SOMEONE TOOK A TRANSCRIPT OF A COUPLE ARGUING AT IKEA AND MADE RANDOM EDITS UNTIL IT COMPILED WITHOUT ERRORS.
OKAY, I'LL READ A STYLE GUIDE.
Figure 3.1: `https://xkcd.com/1513/`

Many of these things seem trivial. It certainly has no functional impact if you use single-quotes on Monday, and double-quotes on Tuesday; or tabs in one file and spaces in another. And indeed, if you joined a project where you were obliged to follow a style-guide that differed from your normal style, wouldn't your productivity wane?

Why even discuss style?

At Jumping Rivers, our code style enforces using double quotes when calling the `library()` function. This avoids complex discussions in our introductory courses.

As with many of the things discussed in this course, code style impacts upon the long-term success of a project or team, rather than your productivity on any given day. If on a given project, all variables and functions are `snake_case` , then you will find it easier to find the `db_connection` you needed, and because this limits the universe of choice, you'll find it easier to name the `parse_tweets()` function you are defining. When scrolling through code that is visually heterogeneous, the different style of two neighbouring code blocks can be off-putting: by maintaining a consistent code style[4], you will find it easier to focus on what the code *does* rather than how it *looks* when you come back to read it again.

[4] Shockingly, coders can be pedantic at times. So having a single uniform style avoids the "styling wars".

3.2 *Automation to the rescue*

Depending on the team and project, sticking to a style guide could feel like a burden: writing code may take you longer (you are both solving a problem, and thinking how to style your solution), and reviewing code may both take longer and identify seemingly minor issues.

Automation can prevent code-style from becoming a burden. This has a two-fold benefit: people find it less intrusive when an automated tool tells them they've misaligned their code than when their colleagues do, and, code-reviews become more pleasant since you and your colleagues can focus on the high-level design of your code.

There are two main types of code-style automation tools:

- Code formatters (stylers)
  - These tools *do* change your source code;
  - They fix things that won't influence the running of your code, e.g., white-space, line-wrapping, quoting;
  - Limited configuration options;
  - Examples: R {styler} and python {black}.
- Code analysers (linters)[5]
  - These tools check your code against a style-configuration for your project;
  - They do not modify your code;
  - They can identify the same issues that are fixed by formatters;
  - But also identify things that require human-input to fix them: variable-names, use of undesirable functions, absolute-paths, system calls etc.;
  - Highly configurable;
  - Examples: R {lintr}[6] and python {pylint} & {flake8}.

[5] Lint being the loose fluff that comes away from your jumpers...

[6] All Jumping Rivers course notes are put through a lint check, to ensure a consistent style across courses. Our trainers usually curse the lintr.

Many IDEs (RStudio, VS Code, PyCharm) and text editors provide integrations for these tools. But, when and where these tools are used is up to you—to ensure they are used, it is better to set up an automated pipeline. A typical work flow is to run code-formatters on-the-fly (e.g., whenever a file is saved) and to use code-analysers as part of a continuous integration pipeline, so that running this step doesn't depend upon the local computer set up of each member of your team.

Be consistent

To maintain the aesthetic consistency of the code in your project, you will need to decide on a code style, document it, and then encode it in such a way that it can be enforced upon the project (by configuring a formatter and analyser).

However, you shouldn't reinvent the wheel. Instead, adopt an existing style guide and tweak as necessary. There are several style guides for R, including those used by the tidyverse team and Google. It is a good idea to use such a well-defined style guide in your own projects for several reasons: the style guide is well-documented, many open-source projects already use that style-guide, and configuration options for the styling tools can readily be found by looking within those projects. The default options for both `{styler}` and `{lintr}` are based on the tidyverse style guide.

IDEs often provide tools for automatic styling. For example, in RStudio, you can use `Ctrl + I` to fix indentation and `Ctrl + Shift + A` to reformat a selected section of code.

Running your own linter

At Jumping Rivers we use an R style guide that is based on the tidyverse-style, but where the equals sign is used for assignment instead of the tidyverse default of left arrow assignment: `<-` .[7]

[7] Our rationale is that we switch between languages: R to Python to Javascript, and this just helps muscle memory.

If we had an R script containing the following code, then we would want our automated tools to identify and/or fix where

- the use of white space is inconsistent
- the left-assignment operator is used.

```r
# my-script.R
a<- 123
b =    234
mean(c(a,b))
```

Table 3.1: Scoping options in {styler}.

Scope	Description
none	No transformation
spaces	Spacing between tokens on the same line
indentation	“spaces” option + indention level
line_breaks	“indention” option + line breaks.
tokens	“line_breaks” option + tokens.

To run `{styler}` and `{lintr}` on a file you would use these commands:

```r
styler::style_file(file_path, "... styler options ...")
lintr::lint(file_path, "... lintr options ...")
```

The tools also have commands for working with whole directories or packages, and can work with code as plain text. If we store the text for the above script as `code` , here is what it looks like after running it through `{styler}` with default settings:

```r
styler::style_text(code)
# my-script.R
a <- 123
b <- 234
mean(c(a, b))
```

That has made the spacing more consistent between lines, but unfortunately, means that the equals-assignments have been replaced with left-assignments.

You can configure `{styler}` to have less drastic effects on your code by using the `scope` argument. For simplicity, we will use `scope="line_breaks"` which is the second-most severe transformation that `{styler}` can apply.[8]

[8] It changes spacing, indentation and line-breaks, but doesn’t replace any symbols.

```r
styled_code = styler::style_text(
  code,
  scope = "line_breaks"
)
```

The code now looks like this:

```r
# my-script.R
a <- 123
b = 234
mean(c(a, b))
```

For those things that code-stylers cannot readily fix, you may be able to configure code-linters to highlight. Let's see what `{lintr}` identifies in our newly-styled code: [9]

```r
# Running {lintr} with default configuration
collapsed_code = paste0(styled_code, collapse = "\n")
lintr::lint(collapsed_code)
#> <text>:3:3: style: Use <-, not =, for assignment.
#> b = 234
#>   ^
```

[9] `lintr::lint` accepts a character string as its first argument, it determines whether this is a file-path or plain-code based on the presence of new-lines characters. Hence why we newline-collapse the code here.

Again, since `{lintr}` isn't natively configured for our style-guide, it is telling us where our code should be changed to match the tidyverse style, rather than where it should be changed to match *our* style. We can configure the rules that `{lintr}` follows, as follows:

```r
linters = lintr::with_defaults(
  # Disable rule that tells us to use "<-" instead of "="
  assignment_linter = NULL,
  # Define a rule that says not to use "<-" in the script
  arrow_linter = lintr::undesirable_operator_linter(
    list("<-" = "please use '=' to assign variables")
  )
)

lintr::lint(collapsed_code, linters = linters)
#> <text>:2:3: warning: Operator `<-` is undesirable. As an
#> alternative, please use '=' to assign variables.
#> a <- 123
#>   ^~
```

This time, linting has identified the use of the left-assignment operator.

3.3 *Layout of scripts*

There should be a logical order to the structure of your scripts. You should decide on conventions for ordering these, however there are some general tips about packages, functions and spacing.

Handling dependencies

In R and python, scripts are ran from top-to-bottom. When a code block runs, all the dependencies of that code block (the variables, functions, classes and packages that it requires) must already be available. This constrains the structure of your files. Since each package must be loaded before it is used, it makes sense to put *all* package-loading code at the very start of a script. Organising package-loading code like this is standard practice in these languages: it gives the reader a quick overview of the external dependencies of the script, and since these lines run first, means that a script will fail early if any package-dependencies are missing.

A common anti-pattern in R is to load packages within the body of a function, using the following syntax.[10]

```
f = function(x) {
  if (!require("dplyr")) {
    stop("Couldn't load {dplyr}")
  }
  # {dplyr} is now loaded and attached
  filter(x, is.na(some_column))
}
```

[10] `library()` raises an error when a package is missing, where as `require()` raises a warning. Typically you want to use `library()` —you want to fail as early as possible.

Doing this is dangerous. Loading a package modifies the global namespace[11] and may change the definitions of functions used later in the script. For example, both {dplyr} and {biomaRt} export a `select()` function, and the `select` function that is available to the user depends on the order in which they are loaded. If you need to optionally depend on installed packages, it is better to check if the package is installed (using `requireNamespace()`) and then call the optional function using its fully-specified name e.g., `myPackage::function_from_the_package()`.

[11] As a general rule, a function call should never alter the global namespace.

This structure of placing all imports in a single place can be taken further. Consider the other internal and external dependencies within a script. The other external dependencies might include user-defined options, files, databases and web pages that are accessed while running the code—by handling these in a single place, early during the running of your script, you help the reader understand the inputs, outputs and requirements of that script. [12]

[12] The running of the script may be improved by this too: imagine running a lengthy computation before checking whether you can write the results to a given directory.

Organising functions

Similarly, each function must be defined before it is called. As such, it is common to put all function definitions together in a single place towards the start of a script. A sensible choice would be to collect together functions by related tasks, or by the type of dataset that they work upon, so that navigating your functions is simplified.

Collecting your functions together still leaves a lot of style choice. Note that functions aren't *called* if they are used in the definition of another function. So this code is invalid because you can't call a function before it is defined:

```
double(2)
double = function(x) x * 2
```

But, this code is perfectly valid, because `double()` is not *called* during the definition of `triple()` (it is only called when `triple()` is evaluated):

```
triple = function(x) double(x) + x
double = function(x) x * 2
triple(3)
```

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

So, when organising your functions in a script or package, you might organise them in a top-down way:

```
my_important_function = function(x) {
  cleaned_up_data = clean_me(x)
  summarised_data = summarise_me(cleaned_up_data)
  summarised_data
}
clean_me = function(x) {
  # how to clean the data
}
summarise_me = function(x) {
  # how to summarise the data
}
```

Or, you might organise them in a bottom-up way:

```
summarise_me = function(x) {
  # how to summarise the data
}
clean_me = function(x) {
  # how to clean the data
}
my_important_function = function(x) {
  cleaned_up_data = clean_me(x)
  summarised_data = summarise_me(cleaned_up_data)
  summarised_data
}
```

Using vertical space and declaring variables

When it comes to code layout, vertical spacing can be useful to help the user follow the code structure. Similar concepts and functions can be located close together, whilst code that is unrelated can be kept further apart.

Often, we need to define variables for example defining a global colour for use in plotting `jumping_rivers_blue = "#516e7a"` or specifying a common file path `path_to_data = "data/user-data"`. It's a good idea to define these variables near the code where they are used. If the variables are used throughout the script, you might choose to define them either near the first instance of their usage, or at the top of the script.

For example the following code works but could be spaced out and reordered to be easier on the eye.

```
library("dplyr")
library("ggplot2")
library("readr")
data(starwars, package = "dplyr")
maximum_mass = 500
starwars_yellow = "#FFE81F"
```

```
starwars_anon =
  starwars %>%
  select(height, mass)
path = "starwars"
height_weight_scatter =
  starwars_anon %>%
  filter(mass < maximum_mass) %>%
  ggplot(aes(x = height, y = mass)) +
  geom_point(fill = starwars_yellow)
write_csv(x = starwars_anon,
          file = file.path(path, "starwars_anon.csv"))
ggsave(filename = file.path(path, "height_weight.png"),
       height_weight_scatter)
```

We can group the library calls together, chunk the pre-processing, the plotting and the saving. By separating these sections of code using vertical space (new lines) we can help the reader quickly search through the code and find the relevant lines. We can also move the `path` variable near to the code where it is first used. Keeping variables defined close to where they are used can help debugging.[13]

```
library("dplyr")
library("ggplot2")
library("readr")

data(starwars, package = "dplyr")
starwars_anon =
 starwars %>%
 select(height, mass)

starwars_yellow = "#FFE81F"
maximum_mass = 500
height_weight_scatter =
 starwars_anon %>%
 filter(mass < maximum_mass) %>%
 ggplot(aes(x = height, y = mass)) +
 geom_point(fill = starwars_yellow)

path = "starwars"
write_csv(x = starwars_anon,
          file = file.path(path, "starwars_anon.csv"))
ggsave(filename = file.path(path, "height_weight.png"),
      height_weight_scatter)
```

Ultimately, how you organise the large-scale structure of your scripts is up to you. But since there are few automated tools available to maintain this global structure, it is important that you write in a way that means your design choices will be continued as the code develops.

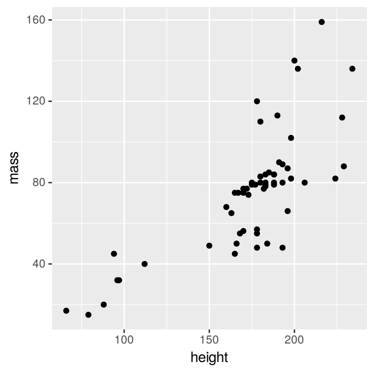


Figure 3.2: Height vs Mass (Starwars characters).

[13] If `starwars_yellow` was used throughout the script, we might choose to define it at the at the top of the script instead.

3.4 *Code comments*

Comments and function-level documentation provide extra information about your source code. Your code would work perfectly well without any comments or documentation, however, well placed comments can help a programmer have a better understanding of your code. Comments can provide a means to elaborate:

- what the code does; and
- why the code was written the way it was.

These are details of the *contract*, *constraints* and *design* of the code: aspects of your code that change relatively slowly.

A mistake that many people make is to comment on the *mechanics* of how their code works. But that's what the code is for, so this kind of comment just duplicates information that could be read from the source files. Also, since code evolves over time, unless efforts are made to check comments whenever the code changes, these mechanistic comments can quickly become out-of-date.

Code-comments can be very useful, for example, alerting a programmer that the model fit code is slow to run. However, comments can also be over-used and they can get out-of-sync with the code they are supposed to clarify. Before writing a comment, ask yourself the following questions:

- Would this information be better in the function documentation?
- Is what I'm typing accurate?[14]
- Is my comment encapsulated by my version control (e.g. author, date, change log)
- Can I refactor my code so it's clearer, making the comment redundant.

[14] No comments are better than an inaccurate comment!

If you have written a paragraph of comments to explain the code, you probably need to refactor the code instead. Sometimes, splitting up functions into smaller components with clear names will mean the comment is no longer required.

Commented code

Why do we have the tendency to comment out code instead of just deleting it? The reason is that we don't want to lose a piece of code that might come in handy in the future. It all boils down to fear of losing information. But commented out code will get in the way more than it will help you, and it's better off being deleted.

Commented out code is distracting when you are reading a script, making it harder for your brain to build a mental model of what the code is doing. Also, as our commented out code gets older, the higher the chances it no longer fits when uncommented which might result in failing code or bugs.

If you are using version control software such as git, you can safely delete those lines of code, knowing you can always retrieve them later if you need them. If that isn't an option, you might want to consider pulling the commented code into a knowledge repository associated with the code base.

Code banners

Some people like to split their code up into sections with *code banners*. These are comments whose only purpose is to split the script up visually, allowing the user to scan across the different sections of a script. Some programmers think these banners are unnecessary code junk. Again, it's up to you to decide what works for you. If you like code banners, you can add them in RStudio with `Ctrl+Shift+R` and jump between sections with `Shift+Alt+J`.

```r
# Load packages -------------------------------------
library("dplyr")
library("ggplot2")

# Analysis -------------------------------------------
# Start data analysis
```

4 Reproducible Workflows

A minimal standard for data analysis is that they be reproducible: that the code and data are assembled in a way so that another person can recreate all of the results. Adopting a work flow that will make your results reproducible will ultimately make your life easier; if a problem arises, or your data updates, it will be much easier to correct.

Organising analyses so that they are reproducible is not easy. It requires diligence and a considerable investment of time, but *partially reproducible* is better than not at all reproducible. Just try to make your next project better organised than the last.

You can think of different levels of reproducibility.

1. You can rerun the analysis next week on the same computer
2. You can rerun the analysis on a different computer
3. Someone else can rerun the analysis (today)
4. Someone else can rerun the analysis in 5 years time

4.1 *Organise your files with RStudio projects*

Perhaps the most important step to take towards reproducibility is to be organised. Each project should live in its own directory, which is named after the project. This directory should contain *everything* needed to reproduce your work, including all scripts, data, documentation and results. This is such a common practice that RStudio has built-in support for this via RStudio Projects.

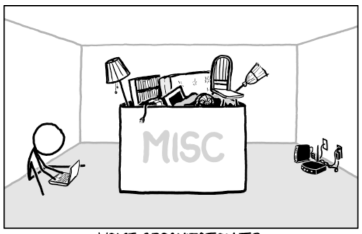


Figure 4.1: `https://xkcd.com/1077/`

The main advantage of using RStudio Projects is that it manages your working directories for you. Your working directory is where R looks for files that you ask it to load, and where it will put any files that you ask it to save. You can find your current working directory in R code by running:

```r
getwd()
```

Beginner R users commonly specify their the working directory at the top of R scripts.[1]

```r
setwd("C:/Documents/analysis/path/to/my/CoolProject")
cats = readr::read_csv("data/cat-species.csv")
```

[1] As a general rule of thumb, if your username appears *anywhere* in R code, you are doing it wrong.

Although this might work for a while, if you move your files around, or try to share them with a colleague those line of code won't work, because the file path won't exist.

When you create an RStudio Project, a `.Rproj` file is created. Whenever you open your Project, your working directory will automatically be set to where the `.Rproj` file lives—your *project home*. Whenever you refer to a file with a relative path[2], R will look for it here. Therefore, you should never need to use `setwd()`.

[2] A *relative path* points to a specific location in a file system relative to the current directory you are working on.

You can create a new project in RStudio by clicking

File → New Project

4.2 *Document*

Documentation of code occurs at several scales. Suppose you have written a new package or an analysis project. You might write:

- a README, website[3] or publication to help new users get an overview of the purpose of the project;
- tutorials and other long-form documentation, to help users understand how to use your code to achieve an overall goal;
- help pages to explain how to use specific components of the code (specific functions or classes);
- comments to help programmers understand the inner workings of your code.

[3] {pkgdown} is a package designed to make it quick and easy to build a website for your package.

For projects like open-source packages, it is hard to imagine prospective users or contributors persevering if the documentation is hard to follow, misleading or missing.

But for most projects, the user and maintenance-programmer described above will be somewhat closer to home: either yourself or members of your team. So, much like choosing informative names and writing well-structured code, spending some time to document your code will aid your future progress.

As with many things, documentation can be taken too far. Too much documentation and it may become a maintenance burden—becoming out-of-sync with the source code that it describes—this is a particularly common problem with code comments.

4.3 *Everything with a script*

Ideally, get the data in the rawest form possible. If you need to download external data, such as from the internet, it's best to do so via a script.[4] If that's not feasible, at least document the source of the data and when it was downloaded.

[4] Checkout the R package {httr}.

Write scripts for every stage of data processing — cleaning, renaming, filtering. You should *never* edit data files by hand. If you need to convert a data file from one form to another, use a script. Rather than opening the file in excel and saving each sheet by hand, do this with code.

All aspects of data cleaning should be in scripts. If you decide that you need to omit a couple of subjects, don't go into the data file and delete those rows, but rather write a script that will delete those subjects and create a separate, revised data file. If you find some errors in the data, don't go into the file and change it by hand, but rather make those changes via a script.[5]

[5] Don't be afraid to rename column names!

4.4 *Keep things fresh*

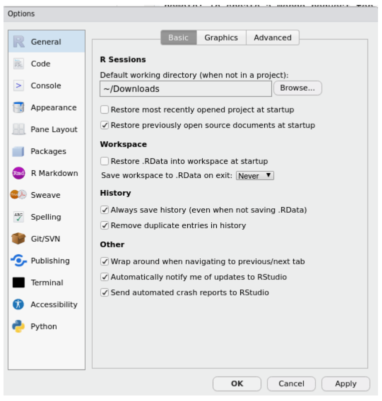
When beginners start using R, they tend to think of their environment variables as the *real* bit of their programming. They want to save their environment at the end of the session and are nervous about the idea of refreshing their R session. However, in the long run, it's better to think of your R scripts as *real* and not to worry about your environment variables. If you've got your scripts—you can always recreate your environment. It's much harder to recreate your R scripts from your environment!

R can save and reload the user's workspace between sessions via an `.RData` file in the current directory. However, we recommend turning this feature off and clearing R's memory at every restart. The {usethis} package has a function to disable saving the user workspace.

Refresh your session often with Ctrl + Shift + F10 to restart RStudio. In general, avoid `rm(list = ls())`.

```
usethis::use_blank_slate()
#> v Setting RStudio preference save_workspace to 'never'
#> v Setting RStudio preference load_workspace to FALSE
#> v Creating '/root/.config/rstudio/'
```

This will cause you some short-term pain, because now when you restart RStudio it will not remember the results of the code that you ran last time. But this short-term pain will save you long-term agony because it forces you to capture all important interactions in your code. There's nothing worse than discovering three months after the fact that you've only stored the results of an important calculation in your workspace, not the calculation itself in your code. Starting with a blank slate provides timely feedback that encourages the development of scripts that are complete and self-contained.



R Sessions

Default working directory (when not in a project): ~/Downloads

☐ Restore most recently opened project at startup

✓ Restore previously open source documents at startup

Workspace

☐ Restore .RData into workspace at startup

Save workspace to .RData on exit: Never

History

✓ Always save history (even when not saving .RData)

✓ Remove duplicate entries in history

Other

✓ Wrap around when navigating to previous/next tab

✓ Automatically notify me of updates to RStudio

✓ Send automated crash reports to RStudio

Figure 4.2: Don't save your workspace.

4.5 *Manage your dependencies*

Code that you write depends on a number of things, including the version of R and the R packages installed on your laptop. If you share your code with your colleague—will it run on their computer? What if {dplyr} updates and changes some key functions? Will your code run still in two years?

Figure 4.3: `https://rstudio.github.io/renv/`

{renv} is a dependency manager for R. It captures the state of your current R packages within a *lockfile*. This allows you to easily share and collaborate on projects with others, and ensure that everyone is working from a common base.

The general workflow when working with {renv} is:

1. Run `renv::init()` to initialise a new project-local environment with a private R library;
2. Work in the project as normal, installing and removing new R packages as they are needed in the project;
3. Run `renv::snapshot()` to save the state of the project library to the lockfile;[6]
4. Call `renv::restore()` to revert to the previous state as encoded in the lockfile.

[6] Called `renv.lock`.

4.6 *Version control*

Working with others in the same space can often get messy, especially when it comes to code. Imagine that you and your colleague Maria are working in the same file.

If you work in a large organisation, your I.T. department will almost certainly use some form of version control. Don't reinvent the maintenance wheel. Ask what they use!

Do any of these file names look familiar to you?

```
code_v1.R
code_v1b_add_plot.R
code_v1b_add_plot_FINAL.R
code_v1b_add_plot_FINAL_FINALv2_sept2020.R
code_v1b_add_plot_FINAL_FINALv2_sept2020_Maria_edits.R
```

If so, you need a version control system such as Git! What are the benefits?

- Backups of your files are stored remotely using an online host such as GitHub or GitLab.
- If you are working in a team, backups are stored on multiple locations.
- Git tracks the changes you make to files, so you have a record of what has been done, and you can revert to specific versions should you ever need to. This is called version control.
- Git allows changes by multiple people to be merged into a single source, thus improving collaborative working.

Git is a whole language in itself, and not something we can go into in this course. However it's definitely worth considering adding version control to your toolbox.

Jenny Bryan's Happy Git and GitHub for the useR book is a good place to start.

4.7 *Write an R package*

As you write more and more functions, you may find yourself wanting to document and share your functions. By far the best way to do this in R is by building an R package. Writing an R package can seem daunting at first. If you've browsed the source code of a popular CRAN package then you can be forgiven for feeling overwhelmed. But a package doesn't need to be the next {data.table} or have the audience of {dplyr} to be worthwhile. If you have a handful of functions that *you* find useful, you can create a *personal R package.*

In RStudio you can create a new package by clicking

New Project → New Directory → R Package

The R packages book `https://r-pkgs.org/` by Hadley Wickham and Jenny Bryan is an excellent resource.

This will populate a skeleton package containing a simple function that has been documented. The {usethis} packages also contains a useful helper function which will help take an existing function and build the infrastructure for an R package around it.

```
usethis::create_package("path_to_r_scripts")
```

Using {usethis} you can add a new R script with `usethis::use_r("name-of-script")`, this will add the file `./R/name-of-script.R` to your package (the `./R/` directory being where most of your package code should go). Your package might have a script for each function, or for each collection of related functions.

Within the package structure, you can add tests for your code. Tests help increase your confidence that your code works as expected and help during subsequent maintenance—if a function is generalised or rewritten, the tests should continue to pass. They also provide additional documentation regarding how your code should be used. Test files are typically stored in the `./tests/` directory of your package. For testing R packages, we recommend investigating the {testthat} package. {usethis} will add a test file and some testing infrastructure to a package using `usethis::use_test("some_function")`, this will add the file `./tests/testthat/test-some_function.R` along with some boilerplate code to help you write your first test.

If you want other people to be able to use your R package, you'll want to release it under a software license. Software licenses are about two things: copyright, and protecting yourself from being held liable if your software screws something up somewhere down the line. There are lots of different software licenses to choose from.[7] At Jumping Rivers we tend to use GNU General Public License (GPL3), but it's worth reading about the different ones and choosing the appropriate one for your project.

[7] See `https://tldrlegal.com/` for a user-friendly overview.

5 Working Collaboratively

One of the best ways you can improve your coding style, is by learning from other programmers. In this chapter we'll look at techniques you can use for working well together and learning from each other.

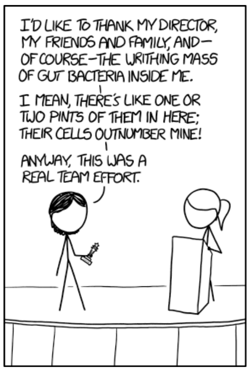

I'D LIKE TO THANK MY DIRECTOR, MY FRIENDS AND FAMILY, AND—OF COURSE—THE WRITHING MASS OF GUT BACTERIA INSIDE ME.
I MEAN, THERE'S LIKE ONE OR TWO PINTS OF THEM IN HERE; THEIR CELLS OUTNUMBER MINE!
ANYWAY, THIS WAS A REAL TEAM EFFORT.

Figure 5.1: `https://xkcd.com/1543/`

5.1 *Code review*

Code review is simply the process of someone else reviewing code that you have written. The reviewer might make comments, offer constructive criticism and resolve small issues. Code reviews are beneficial because they can help eliminate bugs, improve code readability and share knowledge across a team.

If you are using a version control system[1], the code should be reviewed pre-merge, i.e. before it is merged into the main branch. If you are using a git platform like GitLab or GitHub, it is easy to build code review into the merge request process. It is also possible to assign a *code owner*[2] who is responsible for signing off on any changes to the code base.

[1] And you should!

[2] GitLab allows different owners for code. In the `CODEOWNERS` file, we can list usernames and the associated directories.

If you are asked to review someone's code you should try to respond promptly, explain your reasoning and balance being directive with flexibility.[3] If you aren't in the middle of deep work, you should respond to a code review request shortly after it comes in. Otherwise wait until a natural break in your day. As a rule of thumb, we expect code review requests to be responded to within a day. If you are too busy to do a full review, you may be able to provide some initial broad comments or suggest an alternative reviewer.[4]

[3] At Jumping Rivers, we have *merge-monday* for website updates. This allows multiple non-urgent merge requests to be reviewed at once.

[4] The key thing is communication and expectations. If you can't look at something in the near future—let people know!

If you have comments to make, it's important to explain why those changes are necessary.

- **Bad**: "This is slow"
- **Good**: "As the data is so large, running can slow things down. Consider using caching to speed things up"

If the code review is too large, ask the developer to split it up. If the code is too complex, ask them to make it simpler to understand rather than asking them to explain the complexity to you.[5]

[5] The common joke is the larger the merge request, the fewer the number of comments.

Sometimes it's easier to fix something than explain what is wrong. For example, a typo in a variable name, or a violation of your internal code style. In these cases, you might choose to just fix the code yourself rather than asking the developer to fix it. In some cases giving explicit fixes with just pointing out problems and letting the developer decide

Bad	Good
Maria, why did you use a for loop here? There is obviously a much better way to answer that	Have you considered using {purrr}—it could simplify the code.
Why did you use threads here when there's obviously no benefit to be gained from concurrency?	The concurrency model here is adding complexity to the system without any actual performance benefit that I can see. Because there's no performance benefit, it's best for this code to be single-threaded instead of using multiple threads.

Table 5.1: Good and bad code reviews

For example a reviewer might respond as follows.

- I've fixed the linting issues and clarified some variables names[6].
- Consider using `stringr::str_to_lower()` to pre-process the naming field
- The script is getting long, can the code be modularised?

[6] It's essential to have a group policy for this to avoid snake_case vs camelCase wars.

Finally be kind. Sharing code can sometimes make you feel vulnerable and it's important to be courteous and kind. One trick to help writing constructive code reviews to is to only make comments about the code and never make comments about the individual developer.

If you see something nice in the code being reviewed do tell the programmer. Too often, code review just focus on mistakes, but they should offer encouragement and appreciation for good practices as well.

Using your criticism points

If a reviewer makes multiple suggestions, this places a significant burden on the reviewer's time and can also be demoralising the for the original author. We can think of this as *reviewer points*; a reviewer can only spend these points before other issues creep in. A good way of saving these points for key issues is to use continuous integration (CI). This process is useful for picking up on minor issues, like linting, thereby saving the *reviewer points* for other more important topics.

5.2 *Pair programming*

A real-time form of code review is pair programming, where two programmers sit together while writing code. One programmer, *the driver* actually writes the code and focusses solely on the task at hand. The other, called *the navigator* does not type, and is free to detect bugs, issues, design and think about the wider structure of the project.

Pair programming is an efficient way of detecting bugs, writing better code and understanding the task at hand better. Having two people working simultaneously tends to mean you don't get "lost" in the code, and having another person to work with tends to keep both developers on track and productive.

I check Twitter less when I pair program with you. Anonymous Data Scientist at Jumping Rivers.

Pairing also helps with knowledge transfer across a team. You always have someone else who understands the code base. It's a great way to train up junior members of the team, and a good way to get to know your colleagues better.

There can be some downsides of pair programming. Sometimes pair programming can feel a little slow and awkward at the start, especially if you do not know the other programmer well. It can also be tiring and feel intrusive for programmers who find social interactions tiring. Pair programming is a great tool for the right tasks. We recommend that teams use pair programming when bringing a junior team member up to speed and when tackling particularly complex problems.

Pairing is also nice when you are working remotely!

5.3 *Asking great questions*

We all get stuck sometimes and need to ask for help. That's why Q&A forums such as Stack Overflow, the R4DS slack channel and RStudio Community are so popular.[7] Programmers regularly visit sites like these weekly to ask for help or look at the previous answers.

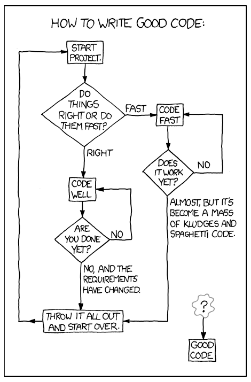

HOW TO WRITE GOOD CODE:
START PROJECT.
DO THINGS RIGHT OR DO THEM FAST?
FAST
CODE FAST
DOES IT WORK YET?
NO
ALMOST, BUT IT'S BECOME A MASS OF KLUDGES AND SPAGHETTI CODE.
RIGHT
CODE WELL.
ARE YOU DONE YET?
NO
NO, AND THE REQUIREMENTS HAVE CHANGED.
THROW IT ALL OUT AND START OVER.
?
GOOD CODE

Figure 5.2: `https://xkcd.com/844/`

[7] A good way of learning how to ask questions, is to try and answer other existing questions. You quickly realise what makes a good question.

If you can't find an existing question that helps you out, you might want to ask others for help. This could be explaining the problem to a colleague or putting out a question in public like on Stack Overflow. But how do you explain your problem in the simplest way so that others will help you? The answer is to create a reprex, or *reproducible example*. The goal of a reprex is to package your problematic code in such a way that other people can run it and feel your pain. Then, hopefully, they can provide a solution and put you out of your misery. There are two parts to creating a reprex, making it reproducible and making it simple.

It's important that someone trying to help you can replicate your problem exactly, this means understanding what packages you have loaded and any existing variables you might have in your workspace. You then have to make it simple. Remove any code which isn't directly related to the issue you are having. Instead of using your actual data, use a built-in R data set. Often in the process or writing the reprex, you actually solve your own problem.

Example reprex

Suppose you have an issue with some {ggplot2} code

```r
source("constants.R")
library("dplyr")
library("ggplot2")
library("lubridate")
library("readr")
my_data = read_csv("my_data.csv")
my_data = my_data %>%
  filter(age > min_age)
ggplot(my_data, aes(age, mitochondrial_load)) +
  geom_point(aes(colour = "yellow"))
```

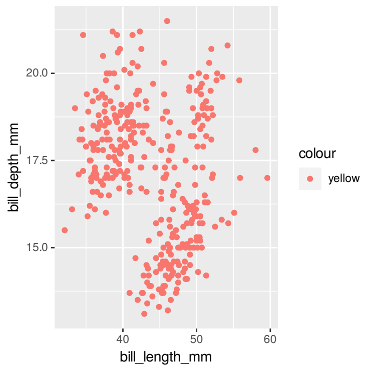
However, instead of the points being yellow, they're red!

The above code doesn't make a good reprex. There are a number of barriers that make it hard for people to help you:

1. The `source()` file is not accessible by most people and likely contains code unrelated to the issue.
2. There are many library calls and the majority are not related to the issue.
3. No-one can run the code!
4. The code contains technical and potentially distracting names, `mitochondrial_load`.

The key idea is to have reproducible code. The above code can be significantly simplified, by removing unnecessary library calls and using standard and accessible data set:

```r
library("ggplot2")
data(penguins, package = "palmerpenguins")
ggplot(penguins, aes(bill_length_mm, bill_depth_mm)) +
  geom_point(aes(colour = "yellow"))
```



The example is now complete reproducible and the problem is clear.

The {reprex} package

Writing a good reprex can be hard work, luckily there is an R package to lend a helping hand. When you want to make a reprex, you need to load the reprex package.

```r
library("reprex")
```

Write a bit of code and copy it to the clipboard. Type `reprex()` in the R Console. In RStudio, you'll see a preview of your rendered reprex. It is now ready and waiting on your clipboard, so you can paste it into, say, a GitHub issue.[8]

Figure 5.3: `https://reprex.tidyverse.org/`

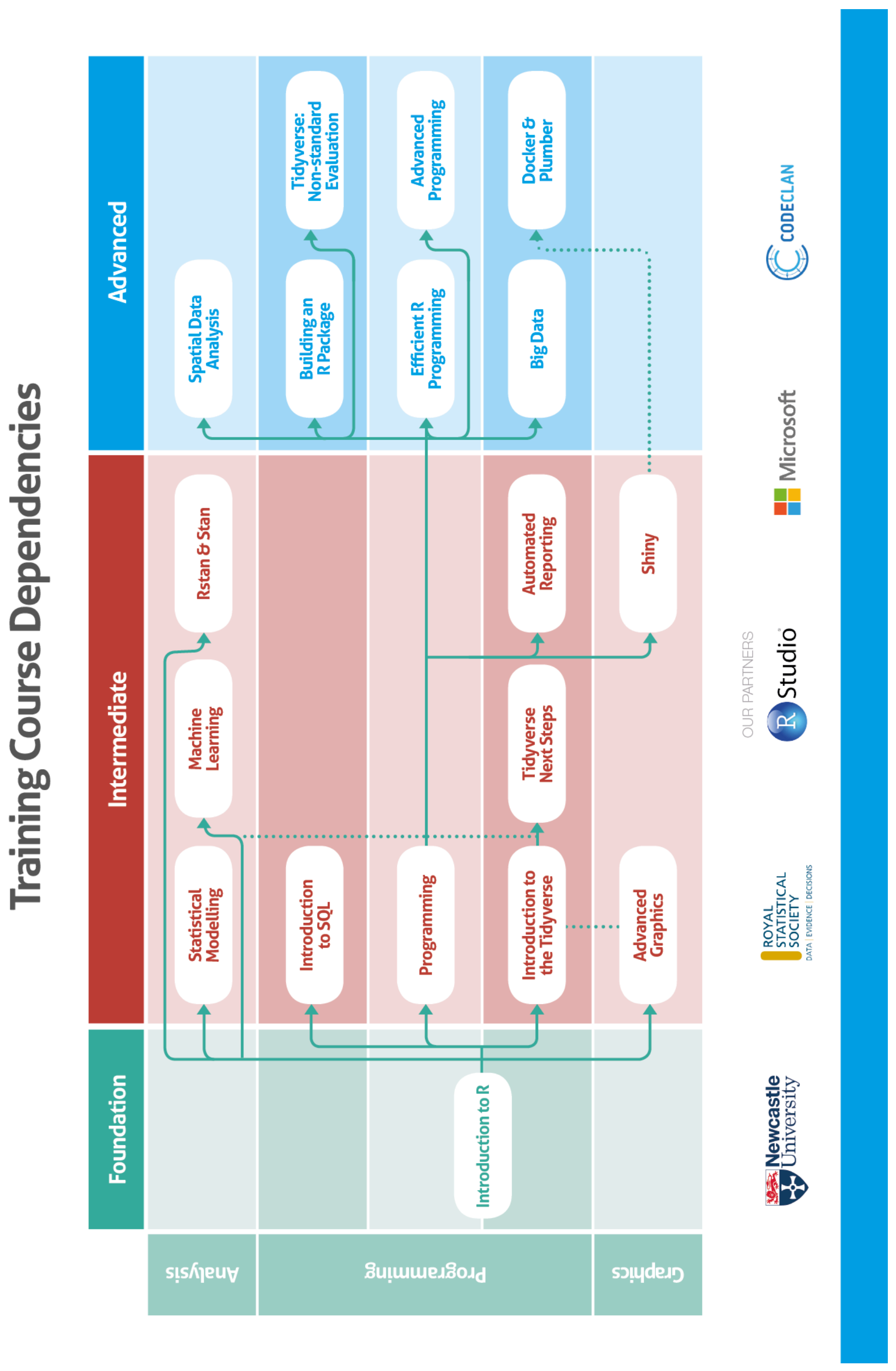
[8] In RStudio, you can access reprex from the addins menu.

5.4 *How to work in a group*

Communication and agreed ground rules are essential for successful group working. For example, should every project have a README? If so, then enforcing a standard rule reduces overall friction, than leaving it to individual opinions.

- Create an overview of your project. (README)
- Create a shared "to-do" list (issue tracker)
- Decide on communication strategies
- Make small changes with frequent feedback

Once a team grows beyond a certain size, it becomes difficult to keep track of what needs to be reviewed, or of who's doing what. Teams can avoid a lot of duplicated effort if they use an issue tracking tool to maintain a list of tasks to be performed and bugs to be fixed. This helps avoid duplicated work and makes it easier for tasks to be transferred to different people. Free repository hosting services like GitHub include issue tracking tools.


Training Course Dependencies
Foundation
Intermediate
Advanced
Analysis
Programming
Graphics
Introduction to R
Statistical Modelling
Machine Learning
Rstan & Stan
Introduction to SQL
Programming
Introduction to the Tidyverse
Tidyverse Next Steps
Automated Reporting
Advanced Graphics
Shiny
Spatial Data Analysis
Building an R Package
Tidyverse: Non-standard Evaluation
Efficient R Programming
Advanced Programming
Big Data
Docker & Plumber
OUR PARTNERS
Newcastle University
ROYAL STATISTICAL SOCIETY DATA | EVIDENCE | DECISIONS
R Studio
Microsoft
CODECLAN