# Best Practice for R : : CHEAT SHEET

## Software

| | |
|---|---|
| RStudio | Write code in the **RStudio** IDE |
| quarto | Use **quarto** for literate programming |
| git | Use **git** to version-control your code and analysis |
| | Use **GitHub** to collaborate with other people |

## Projects

### PROJECT CREATION

- **Create** a new project in RStudio using `File > New Project > New Directory`
- **Do** put projects in a single, local folder like `C:\Users\your-name\Documents`
- **Don't** put projects in locations controlled by OneDrive / iCloud (these don't play well with Git)

### PROJECT STRUCTURE

Most projects should be structured like this:

```
my-project/
├ .gitignore          .gitignore tells git which files not to track
├ .Rprofile           R code to run on startup
├ R/                  Scripts in R/ should define functions for use elsewhere
│ ├ 01-import.R
│ └ 02-tidy.R
├ SQL/                Use folders SQL/, data/ etc for other file types
│ └ costs.sql
├ run-all.R           Use a top-level R script to run everything
├ renv/               Records of package versions; created using renv::init()
├ renv.lock
├ my-project.Rproj    A .Rproj file makes this directory an RStudio project
└ README.md           Write the main facts about the project here
```

NB, `usethis::use_description()` + `usethis::use_namespace()` will turn this structure into a package!

## Packages

Packages should be loaded in one place with successive calls to `library()`

- Use the tidyverse for normal wrangling, plotting etc
- Use tidymodels for modelling and machine learning
- Use {**shiny**}, {**bslib**} and {**bs4Dash**} for app development
- Use r-lib packages like {**rlang**}, {**cli**} & {**glue**} for low-level programming
- Use {**renv**} in long-term projects to track dependency packages

GitHub stars are a good proxy for a package's quality. Not sure whether to use a package? If it has >200 stars on GitHub it's probably good!

## Getting Help

### CREATE A REPREX

- A **minimal, reproducible example** should demonstrate the issue as simply as possible
- Copy your example code and run **`reprex::reprex()`** to embed errors/ messages/outputs as comments
- Use your reprex in a question on Teams or Stackoverflow

```
print("Hello " + "world!")
#> Error in "Hello " + "world!": non-
numeric argument to binary operator
```

This reprex minimally demonstrates an error when attempting to use + for Python-style string concatenation

### ETIQUETTE WHEN ASKING QUESTIONS

| Don't | Do |
|---|---|
| Post screenshots of your code | Use **`reprex::reprex()`** and paste your code as text |
| Include big files | Use **`dput()`** or **`tibble::tribble()`** to include a data sample |
| Ignore messages or warnings | **Ensure your code only fails where you're expecting it to** |

## Databases

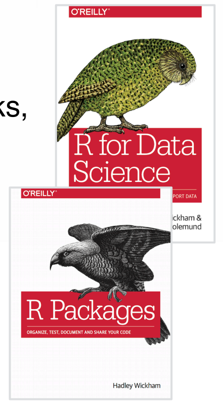
- Use {**DBI**} and {**odbc**} to connect to SQL
- Use **helper functions** to create connections

```
connect_to_db <- function(db) {
  DBI::dbConnect(
    odbc::odbc(), Database = db,
    # Hard-code common options here
  )
}
```

```
# Connect using the helper
con <- connect_to_db("DWH")
```

## Learning More

- For common data science tasks, see *R for Data Science (2e)*
- For package development, see *R Packages (2e)*
- For advanced programming, see *Advanced R (2e)*
- For app development, see *Mastering Shiny*



## Functions

- Write functions to **reduce repetition** or **increase clarity**
- Write many **small** functions that **call each other**
- Define functions in **dedicated scripts** with corresponding names

### NAMING CONVENTIONS

| ✗ Bad (noun-like) | ✓ Good (verb-like) |
|---|---|
| `totals_getter()` | `compute_totals()` |
| `modeller_func()` | `fit_model()` |
| `project_data()` | `import_datasets()` |

### WRITING FUNCTIONS: WORKFLOW

```
a <- complex operation on a
b <- complex operation on b
c <- complex operation on c
d <- complex operation on d
```

1. **Repetitive, complex code; purpose clarified by comments**

```
operate_on <- function(x) {
  complex operation on x
}
```

2. **Complex logic abstracted into functions**

```
a <- operate_on(a)
b <- operate_on(b)
c <- operate_on(c)
d <- operate_on(d)
```

3. **Repetition reduced; clearer code; less need for comments**

## Styling

*For other styling guidance, refer to the Tidyverse style guide*

### NAMING THINGS

- Use **`lower_snake_case`** *for most objects* (functions, variables etc)
- **`Title_Snake_Case`** may be used for *column names*
- Use only **syntactic** names where possible (include only *numbers*, *letters*, *underscores* and *periods*, and don't start with a number)

```
# Good (lower_snake_case everywhere):
add1          <- function(x) x + 1
first_letters <- letters[1:3]
iris_sample   <- slice_sample(iris, n = 5)

# Bad (non-syntactic, not lower_snake_case):
`add 1`       <- function(x) x + 1
FirstLetters  <- letters[1:3]
iris.sample   <- slice_sample(iris, n = 5)
```

### WHITESPACE

- **Add spaces** after commas and around operators like `|>`, `%>%`, `+`, `-`, `*`, `/`, `=` and `<-`
- **Indentation increases** should always be by *exactly* 2 spaces
- **Add linebreaks** when lines get longer than **80** characters.
- When there are many arguments in a call, **give each argument its own line** (including the first one!)

```
# Good (lots of spaces, indents always by +2):
df <- iris |>
  mutate(
    Sepal.Area = Sepal.Width * Sepal.Length,
    Petal.Area = Petal.Width * Petal.Length
  )

# Bad (inconsistent spacing and indentation):
df<-iris |>
  mutate(Sepal.Area=Sepal.Width*Sepal.Length,
      Petal.Area=Petal.Width*Petal.Length)
```