Writing Clean and Efficient Code

An Introduction

Benjamín Muñoz

Massachusetts Institute of Technology

March 07, 2025
PML Workshop

Table of Contents

What is Clean Code?

- Your code should read like a recipe for mediocre cooks.

What is Clean Code?

- Your code should read like a recipe for mediocre cooks.
  - ⋆ **Why mediocre?**

What is Clean Code?

- Your code should read like a recipe for mediocre cooks.

  ⋆ **Why mediocre?** Some base knowledge is expected, i.e. you do not need to explain what `x <- 2` means, but do not assume they are deeply familiar with your esoteric choice of packages.

What is Clean Code?

- Your code should read like a recipe for mediocre cooks.

  ⋆ **Why mediocre?** Some base knowledge is expected, i.e. you do not need to explain what `x <- 2` means, but do not assume they are deeply familiar with your esoteric choice of packages.

  ⋆ **Why recipe?**

What is Clean Code?

- Your code should read like a recipe for mediocre cooks.

  ⋆ **Why mediocre?** Some base knowledge is expected, i.e. you do not need to explain what `x <- 2` means, but do not assume they are deeply familiar with your esoteric choice of packages.

  ⋆ **Why recipe?** Your code should be well-organized so that a reader knows what to expect. Maybe at first glance I want to know what is the input (ingredients), output (dish), and what are the steps to get there.

What is Clean Code?

- Your code should read like a recipe for mediocre cooks.

  ⋆ **Why mediocre?** Some base knowledge is expected, i.e. you do not need to explain what `x <- 2` means, but do not assume they are deeply familiar with your esoteric choice of packages.

  ⋆ **Why recipe?** Your code should be well-organized so that a reader knows what to expect. Maybe at first glance I want to know what is the input (ingredients), output (dish), and what are the steps to get there.

- Three important aspects that make a recipe good:

What is Clean Code?

- Your code should read like a recipe for mediocre cooks.

  ⋆ **Why mediocre?** Some base knowledge is expected, i.e. you do not need to explain what `x <- 2` means, but do not assume they are deeply familiar with your esoteric choice of packages.

  ⋆ **Why recipe?** Your code should be well-organized so that a reader knows what to expect. Maybe at first glance I want to know what is the input (ingredients), output (dish), and what are the steps to get there.

- Three important aspects that make a recipe good:
  1. **Annotation:** I can understand the instructions even if I have never made the recipe before.

What is Clean Code?

- Your code should read like a recipe for mediocre cooks.

  ⋆ **Why mediocre?** Some base knowledge is expected, i.e. you do not need to explain what `x <- 2` means, but do not assume they are deeply familiar with your esoteric choice of packages.

  ⋆ **Why recipe?** Your code should be well-organized so that a reader knows what to expect. Maybe at first glance I want to know what is the input (ingredients), output (dish), and what are the steps to get there.

- Three important aspects that make a recipe good:
  1. **Annotation:** I can understand the instructions even if I have never made the recipe before.
  2. **Organization:** I know where to find the input and where the outputs will be located.

What is Clean Code?

- Your code should read like a recipe for mediocre cooks.

  ⋆ **Why mediocre?** Some base knowledge is expected, i.e. you do not need to explain what `x <- 2` means, but do not assume they are deeply familiar with your esoteric choice of packages.

  ⋆ **Why recipe?** Your code should be well-organized so that a reader knows what to expect. Maybe at first glance I want to know what is the input (ingredients), output (dish), and what are the steps to get there.

- Three important aspects that make a recipe good:
  1. **Annotation:** I can understand the instructions even if I have never made the recipe before.
  2. **Organization:** I know where to find the input and where the outputs will be located.
  3. **Abstraction:** Combine basic instructions into more abstract functions or states.

Annotation

1. **Variable/Function Naming:**
   - One Ring to Rule them All: Be Consistent!
   - Good Idea to use a Style Guide and following it closely for the best results.

Annotation

1. **Variable/Function Naming:**
   - One Ring to Rule them All: Be Consistent!
   - Good Idea to use a Style Guide and following it closely for the best results.
   - Even if you deviate, make sure you do so consistently. For instance you prefer `camelCase` over snake_case naming.

Annotation

1. **Variable/Function Naming:**
   - One Ring to Rule them All: Be Consistent!
   - Good Idea to use a Style Guide and following it closely for the best results.
   - Even if you deviate, make sure you do so consistently. For instance you prefer `camelCase` over snake_case naming.
   - Style Guides: Tydiverse and Google are pretty common choices.
   - Practical Option: use the `styler` package. You can check the documentation here. Another option is formatR.
2. **Commenting:** Include general phrases to describe your steps (logical sequence) or important details you want to remember.
   - I like to err on the more conservative side, so more comments.

Annotation

1. **Variable/Function Naming:**
   - One Ring to Rule them All: Be Consistent!
   - Good Idea to use a Style Guide and following it closely for the best results.
   - Even if you deviate, make sure you do so consistently. For instance you prefer `camelCase` over snake_case naming.
   - Style Guides: Tydiverse and Google are pretty common choices.
   - Practical Option: use the `styler` package. You can check the documentation here. Another option is formatR.

2. **Commenting:** Include general phrases to describe your steps (logical sequence) or important details you want to remember.
   - I like to err on the more conservative side, so more comments.
   - The bare minimum is: For functions explain input, output, and purpose; For blocks of code explain purpose and maybe some more esoteric steps within.

Annotation

1. **Variable/Function Naming:**
   - One Ring to Rule them All: Be Consistent!
   - Good Idea to use a Style Guide and following it closely for the best results.
   - Even if you deviate, make sure you do so consistently. For instance you prefer `camelCase` over snake_case naming.
   - Style Guides: Tydiverse and Google are pretty common choices.
   - Practical Option: use the `styler` package. You can check the documentation here. Another option is formatR.

2. **Commenting:** Include general phrases to describe your steps (logical sequence) or important details you want to remember.
   - I like to err on the more conservative side, so more comments.
   - The bare minimum is: For functions explain input, output, and purpose; For blocks of code explain purpose and maybe some more esoteric steps within.
   - Key point: The more you abstract away, and the better your naming, the fewer comments you will need!

Organization

- It's not only important to organize your code. It's also necessary to do so with your working directory (folder).

1. **Code organization:** Just like your paper, your code will likely have sections, highlight them.

Organization

- It's not only important to organize your code. It's also necessary to do so with your working directory (folder).

1. **Code organization:** Just like your paper, your code will likely have sections, highlight them.
   - E.g. your analysis corresponds to different tables, figures, can label them as such in the code.

Organization

- It's not only important to organize your code. It's also necessary to do so with your working directory (folder).

1. **Code organization:** Just like your paper, your code will likely have sections, highlight them.
   - E.g. your analysis corresponds to different tables, figures, can label them as such in the code.
   - Commonly use some type of separators `######` or `----------`.

Organization

- It's not only important to organize your code. It's also necessary to do so with your working directory (folder).

1. **Code organization:** Just like your paper, your code will likely have sections, highlight them.
   - E.g. your analysis corresponds to different tables, figures, can label them as such in the code.
   - Commonly use some type of separators `######` or `----------`.
   - In RStudio, can use headings directly by commenting `# HEADING NAME ------`
2. **Folder organization:** Keep paper, presentation, data, data wrangling, analysis, and simulations separate.

Organization

- It's not only important to organize your code. It's also necessary to do so with your working directory (folder).

1. **Code organization:** Just like your paper, your code will likely have sections, highlight them.
   - E.g. your analysis corresponds to different tables, figures, can label them as such in the code.
   - Commonly use some type of separators `######` or `----------`.
   - In RStudio, can use headings directly by commenting `# HEADING NAME ------`
2. **Folder organization:** Keep paper, presentation, data, data wrangling, analysis, and simulations separate.
   - Not all projects will include all of these, but try to keep them as separate as possible.

Organization

- It's not only important to organize your code. It's also necessary to do so with your working directory (folder).

1. **Code organization:** Just like your paper, your code will likely have sections, highlight them.
   - E.g. your analysis corresponds to different tables, figures, can label them as such in the code.
   - Commonly use some type of separators `######` or `----------`.
   - In RStudio, can use headings directly by commenting `# HEADING NAME ------`
2. **Folder organization:** Keep paper, presentation, data, data wrangling, analysis, and simulations separate.
   - Not all projects will include all of these, but try to keep them as separate as possible.
   - Benefits: Easy to find replication data and sought after code.

Organization

- It's not only important to organize your code. It's also necessary to do so with your working directory (folder).

1. **Code organization:** Just like your paper, your code will likely have sections, highlight them.
   - E.g. your analysis corresponds to different tables, figures, can label them as such in the code.
   - Commonly use some type of separators `######` or `----------`.
   - In RStudio, can use headings directly by commenting `# HEADING NAME ------`
2. **Folder organization:** Keep paper, presentation, data, data wrangling, analysis, and simulations separate.
   - Not all projects will include all of these, but try to keep them as separate as possible.
   - Benefits: Easy to find replication data and sought after code.
   - Costs: It is painful to work with paths, especially across operating systems.

Organization

- It's not only important to organize your code. It's also necessary to do so with your working directory (folder).

1. **Code organization:** Just like your paper, your code will likely have sections, highlight them.
   - E.g. your analysis corresponds to different tables, figures, can label them as such in the code.
   - Commonly use some type of separators `######` or `----------`.
   - In RStudio, can use headings directly by commenting `# HEADING NAME ------`
2. **Folder organization:** Keep paper, presentation, data, data wrangling, analysis, and simulations separate.
   - Not all projects will include all of these, but try to keep them as separate as possible.
   - Benefits: Easy to find replication data and sought after code.
   - Costs: It is painful to work with paths, especially across operating systems. $\Longrightarrow$ `here` package.

Abstraction

- ★ Do not repeat code, write functions instead.

Abstraction

★ Do not repeat code, write functions instead.

- Be practical. If the estimated time to create a function is more than 2 lines of code, there is no problem in repeating it. But it is not a good idea to repeat a line of code hundreds of times.

Abstraction

★ Do not repeat code, write functions instead.

- Be practical. If the estimated time to create a function is more than 2 lines of code, there is no problem in repeating it. But it is not a good idea to repeat a line of code hundreds of times.
- This is an iterative process. You will build more complex functions as you need them for your code.

Abstraction

★ Do not repeat code, write functions instead.

- Be practical. If the estimated time to create a function is more than 2 lines of code, there is no problem in repeating it. But it is not a good idea to repeat a line of code hundreds of times.
- This is an iterative process. You will build more complex functions as you need them for your code.
- Your functions don't have to be created solely with base R. You can use functions present in other packages. However, three basic tips:

Abstraction

★ Do not repeat code, write functions instead.

- Be practical. If the estimated time to create a function is more than 2 lines of code, there is no problem in repeating it. But it is not a good idea to repeat a line of code hundreds of times.
- This is an iterative process. You will build more complex functions as you need them for your code.
- Your functions don't have to be created solely with base R. You can use functions present in other packages. However, three basic tips:
  - Be explicit about the function dependency. Instead of writing `mutate`, write `dplyr::mutate`.

Abstraction

★ Do not repeat code, write functions instead.

- Be practical. If the estimated time to create a function is more than 2 lines of code, there is no problem in repeating it. But it is not a good idea to repeat a line of code hundreds of times.
- This is an iterative process. You will build more complex functions as you need them for your code.
- Your functions don't have to be created solely with base R. You can use functions present in other packages. However, three basic tips:
  - Be explicit about the function dependency. Instead of writing `mutate`, write `dplyr::mutate`.
  - Prioritize functions from established/popular packages on CRAN. Often, over time, packages become deprecated and you will no longer be able to use their code.

Abstraction

★ Do not repeat code, write functions instead.

- Be practical. If the estimated time to create a function is more than 2 lines of code, there is no problem in repeating it. But it is not a good idea to repeat a line of code hundreds of times.
- This is an iterative process. You will build more complex functions as you need them for your code.
- Your functions don't have to be created solely with base R. You can use functions present in other packages. However, three basic tips:
  - Be explicit about the function dependency. Instead of writing `mutate`, write `dplyr::mutate`.
  - Prioritize functions from established/popular packages on CRAN. Often, over time, packages become deprecated and you will no longer be able to use their code.
  - Document the version (number) of the packages you use.

Basics of Reproducibility

- Today most studies already submit code and data is that not enough?

Basics of Reproducibility

- Today most studies already submit code and data is that not enough? Sometimes and in the short term.

Basics of Reproducibility

- Today most studies already submit code and data is that not enough? Sometimes and in the short term.
- R tries to be very compatible with past versions, but that is not necessarily the case for other packages.
- We are in the long-term business, so it might make sense to make sure I can run code from 2013 in 2021 without running into weird hiccups.

Basics of Reproducibility

- Today most studies already submit code and data is that not enough? Sometimes and in the short term.
- R tries to be very compatible with past versions, but that is not necessarily the case for other packages.
- We are in the long-term business, so it might make sense to make sure I can run code from 2013 in 2021 without running into weird hiccups.
- **The reproducibility spectrum:** Snapshot of full operating system > Snapshot of R and all R packages > Code and Data > Nothing.
- `renv`, a hopeful `packrat` replacement fits in the 2nd best spot and should be good enough for most.

★ Reproducibility in the Social Sciences.

★ How to Improve your Relationship with your Future Self

Organization 2.0: Workflow

- We've already discussed file and folder organization. There are a couple of extra tools that are useful.

  ⋆ At a minimum, follow the R4DS recommendations on workflow.

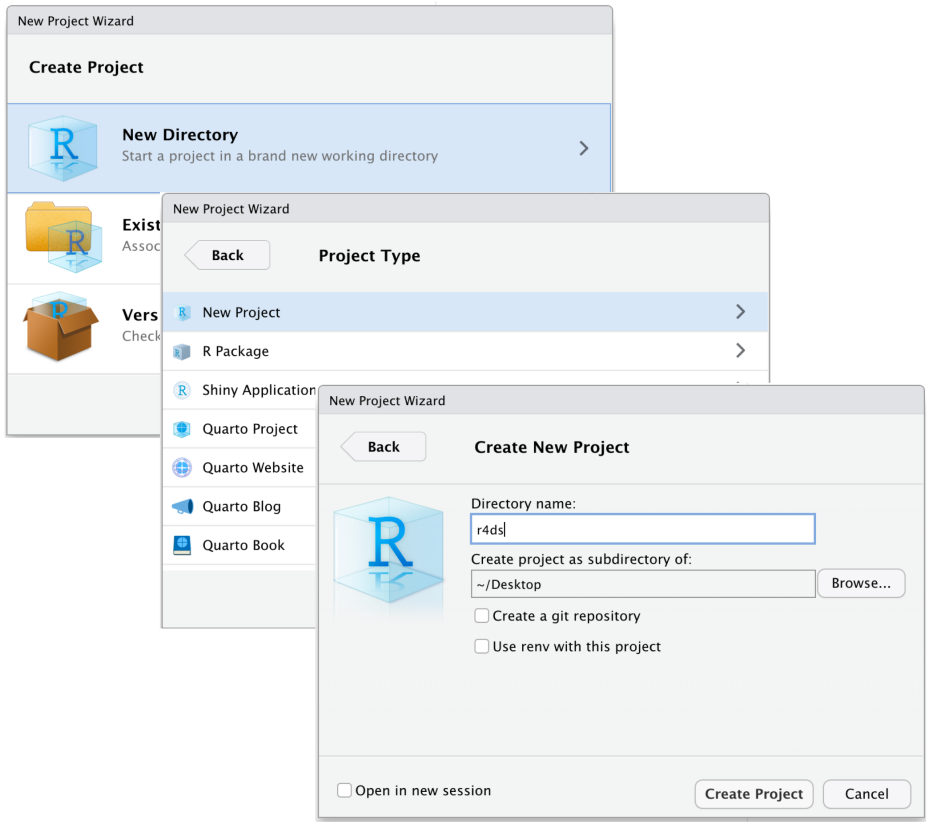
  1. Standardized names for scripts.
  2. Standardized names for outputs (also for formats).
  3. Use `.Rproj` files to set your **working directory**.

  ⋆ Other extreme (maybe too much) use tools as workflowr.

  - Create a website with all the documentation of your project, which helps ensure transparency, reproducibility and shareability.

Organization 2.0: Workflow

- **working directory**. Use the function `getwd()` to check it.
- It is not a good idea to use `setwd(«/path/to/my/CoolProject»)`. Instead, you can use `.Rproj` files. This type of file allows the use of relative paths.



Debugging

- **Debugging hierarchy:** Unit tests > Breakpoint setting and traceback > Printing intermediary output > Explaining what code does in words (Rubber-ducky debugging) > Giving up.

Debugging

- **Debugging hierarchy:** Unit tests > Breakpoint setting and traceback > Printing intermediary output > Explaining what code does in words (Rubber-ducky debugging) > Giving up.
- Goal is to not give up. Usually printing intermediary output is effective enough.

Debugging

- **Debugging hierarchy:** Unit tests > Breakpoint setting and traceback > Printing intermediary output > Explaining what code does in words (Rubber-ducky debugging) > Giving up.
- Goal is to not give up. Usually printing intermediary output is effective enough.
- Some more concrete advice:

Debugging

- **Debugging hierarchy:** Unit tests > Breakpoint setting and traceback > Printing intermediary output > Explaining what code does in words (Rubber-ducky debugging) > Giving up.
- Goal is to not give up. Usually printing intermediary output is effective enough.
- Some more concrete advice:
  - If you followed the advice until now, most of your code will be functions, which makes debugging easier, since you can trace the error to some of those functions.

Debugging

- **Debugging hierarchy:** Unit tests > Breakpoint setting and traceback > Printing intermediary output > Explaining what code does in words (Rubber-ducky debugging) > Giving up.
- Goal is to not give up. Usually printing intermediary output is effective enough.
- Some more concrete advice:
  - If you followed the advice until now, most of your code will be functions, which makes debugging easier, since you can trace the error to some of those functions.
  - Note that you can always run each line of a function individually and print the output to see if it matches what you expect.

Debugging

- **Debugging hierarchy:** Unit tests > Breakpoint setting and traceback > Printing intermediary output > Explaining what code does in words (Rubber-ducky debugging) > Giving up.
- Goal is to not give up. Usually printing intermediary output is effective enough.
- Some more concrete advice:
  - If you followed the advice until now, most of your code will be functions, which makes debugging easier, since you can trace the error to some of those functions.
  - Note that you can always run each line of a function individually and print the output to see if it matches what you expect.
  - When you code, always debug as you go. I.e. don't write 50 functions and then start testing your code. Rather do it, a function or two at a time.
  - Use Generative Intelligence intensively (ChatGPT, Copilot, Deepseek, etc.).

Efficiency in R

- Why is some code «fast» and other code «slow» in R?
- Two main culprits:
  - Bad algorithms.
  - Inefficient memory usage.
- Does not suffice to use and understand `apply()` family of functions.

  ⋆ Need to understand basics of computational complexity and memory allocation.
- **Practical Advice:**
  1. Consider appropriate algorithms (canned functions).
  2. Compartmentalize your code and evaluate its efficiency piece by piece.
  3. Vectorize whenever possible (try to avoid nested loops).
  4. If possible, parallelize.

Computational Complexity Examples

- Good starting point ⟶ measure time from start to end.
  - ⋆ Use the `tictoc` package.
- However, this is not enough. Many other factors impact the speed.

R Code

```
# Block 1: Simple for loop
for (i in 1:n) {
  foo()
}

# Block 2: Sequential for loops
for (i in 1:n) {
  foo()
}
for (i in 1:n) {
  bar()
}

# Block 3: Nested for-loops
for (i in 1:n) {
  foo()
  for (i in 1:n) {
    bar()
  }
}

# Block 4: Simple bootstrap
for (i in 1:b) {
  # Hint: mle with BFGS takes O(n^2)
  estimate <- mle(X_1, ..., X_n, method = BFGS)
}
```

Memory Allocation in R

- Even if the algorithms are efficient, we can lose real-life performance due to inefficient memory usage.
- Will go over basics of memory (in R), data types, and vectorization to motivate efficiency prescriptions.

What is memory?

- Computer consists of three main parts: CPU core(s) (different circuits), Memory (RAM, Hard drive), GPU (can ignore for this session).
- CPU is wicked fast.
- Every R session lives on RAM memory.
- RAM is actually very very fast for memory standards, but painstakingly slow compared to CPUs, particularly saving new things to it (writes).
- To write efficient code, we **need to avoid unnecessary memory writes!**

Memory Usage in R

- What happens when I delete an object with `rm()`?
  - You tell R that this reference to an object is no longer needed.
  - If there are no more, it will be «garbage collected» *automatically*.

  ⋆ **Big Culprit:** Copying of objects.
  - R uses a «copy if modified» framework so if `y <- x`, and `y[1] <- 0`, `y` will no longer point to `x`, but will allocate a new chunk of memory and point to that.
    $\Longrightarrow$ Avoid implicit and explicit copying of objects whenever possible.
    $\Longrightarrow$ «Vectorizing» is mostly faster because it avoids this implicit copying.

Implicit Copying of Objects

R Code

```
# Example from: http://adv-r.had.co.nz/memory.html
library(pryr)
x <- data.frame(matrix(runif(100 * 1e4), ncol = 100))
medians <- vapply(x, median, numeric(1))

for (i in seq_along(medians) |> head(5)) {
  x[, i] <- x[, i] - medians[i]
  print(c(pryr::address(x), pryr::refs(x)))
}

## [1] "0x55ba813f6e80" "1"
## [1] "0x55ba8457d8f0" "1"
## [1] "0x55ba8460eb00" "1"
## [1] "0x55ba8460ee60" "1"
## [1] "0x55ba8493dad0" "1"
```

Explicit Copying of Objects

- Define the size and type of object to avoid copying object.
- Example: creating a vector of ones.

R Code

```
    f1 <- function() { # Adding ones one by one
  x <- 1.0
  for (i in 2:10000) {
    x <- c(x, 1.0)
  }
  return(x)
}
f2 <- function() { # Define length + type of vector
  x <- numeric(10000) # Can be character(), interger() etc
  for (i in 1:10000) {
    x[i] <- 1.0
  }
  return(x)
}
f3 <- function() { # Vectorization
  x <- rep(1.0, 10000)
  return(x)
}
bench::mark( # Sum two columns for each row
  f1(), f2(), f3()
) |> dplyr::select(expression, median)

## # A tibble: 3 x 2
## expression median
## <bch:expr> <bch:tm>
## 1 f1() 162ms
## 2 f2() 452.4us
## 3 f3() 14.4us
```

Functions in R

- From last time, arguments are not copied until modified, yay!
- What if we return a function in a function?
- **Turns out:** Functions save their surrounding environment!
- So what? Imagine you create a large variable (`x <- 1:1e+50`) before you create your function, then your function will also copy the large variable into memory.
- Now on to more applied examples of why data types matter and how to vectorize in R.

Memory Prescription 1: Data Types Matter

- List vs. Matrix vs. Dataframe (or Tibble).
  - Matrix algebra is heavily optimized!
  - Neither lists nor matrices have memory overhead.
  - Dataframes have many specialized functions such as grouping that are much faster than DIY approaches on lists or matrices.

R Code

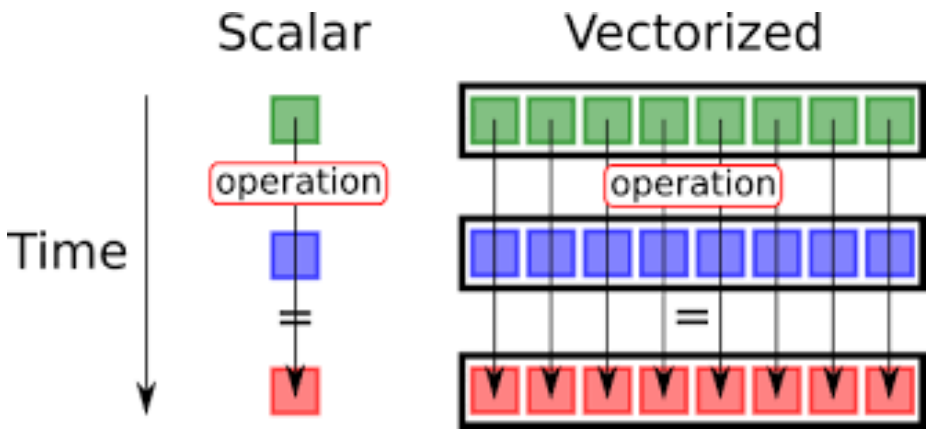
```
x <- runif(10000)
y <- rnorm(10000)
DF <- data.frame(x = x, y = y) # Dataframe
TIB <- tibble::as_tibble(DF) # Tidyverse tibble
MAT <- cbind(x, y) # Matrix

bench::mark( # Sum two columns for each row
  apply(DF, 2, sum),
  apply(TIB, 2, sum),
  apply(MAT, 2, sum),
  colSums(MAT) # Implemented in C
) |> dplyr::select(expression, median)

## # A tibble: 4 x 2
## expression median
## <bch:expr> <bch:tm>
## 1 apply(DF, 2, sum) 271.2us
## 2 apply(TIB, 2, sum) 279.9us
## 3 apply(MAT, 2, sum) 165.2us
## 4 colSums(MAT) 27.9us
```

Memory Prescription 2: Vectorize

- **Vectorize** your codes as much as you can.
- Most R functions allow you to vectorize by default.



★ `apply`, `lapply`, `sapply`, `...` come built-in with R and allow you to apply **any** function in a vectorized way to a matrix/dataframe, list, or vector.

★ `purrr`'s `map`, `map_dbl`, `map_dfc`, `...` are the `tidyverse` equivalents and extensions to the `apply()` family.

Vectorization: apply and purrr

R Code

```
library(purrr)
d <- list(
  data.frame(quant = c("danny", "insong", "teppei")),
  data.frame(schools = c("MIT", "Harvard"))
)
# We want a list as an output
lapply(d, nrow)

# default map returns a list
purrr::map(d, nrow)

# The tidy version
d |> purrr::map(nrow)

## Second Example
v <- 1:10

# We want a vector as an output
sapply(v,                                          (x) x * x)

# The tidy version
v |> purrr::map_dbl(                                    (x) x * x)
```

Vectorization using purrr

- `map_*` return different object type and will fail if it is not appropriate.
- Makes the code more predictable, and thus easier to debug.
- Of particular interest are `map_dfr` and `map_dfc`, which run a function on the input and then row/column bind the outputs together.
- Inspired popular parallelization package `furrr`.

Parallelization: Why and when to parallelize

- Enable multiple computations to take place at the same time.
- Useful when you have time-consuming, unordered tasks.
  - Data cleaning.
  - Bootstrap.
  - Monte Carlo simulations.
  - Any tasks with lots of loops, apply, or maps

Often need two different codes for parallel and non-parallel computing ⟶ Increase potential bugs.

Methods for parallel computing differ across operating systems and different packages implements different methods:

- `mclapply`, `parallel`, `doParallel`, etc.
- Parallelization with `mclapply` (forking) does not work in Windows.
- Your choice of package would decide how to run in parallel.

`future` package

- `future` provides a simple and uniform tool for async. parallel, and distributed processing in R.
- Same coding style between sequential and parallel tasks.
- Users decide how to parallelize: the code does not depend on how to run in parallel.
  - Use the same code for parallel computing in different operating systems

R Code

```
f <- future::future(expr) # Evaluate in parallel
r <- future::resolved(f) # Check if done
v <- future::value(f) # Get result
```