Paper CT04

Beyond Basics: Essential R Code Tips and Tricks for Clinical Programmers

Sanjana Jayacumar, Pfizer Healthcare India Private Limited, Chennai, India

ABSTRACT

As the clinical research industry increasingly embraces open-sources tools, R has become a powerful tool for statistical programming. As the adoption of R continues to grow within the pharmaceutical industry, clinical programmers are increasingly required to manage diverse datasets and implement complex decision logic with precision with efficiency. Many programmers transitioning from SAS or starting in R often face challenges with syntax optimization and reproducibility. This paper offers a collection of practical R code tips and tricks aimed at simplifying the challenges commonly faced in clinical data workflows. Some of the key topics include importing and managing multiple datasets simultaneously, handling join and column operations, regex for dynamic column selection, version control for reproducibility, functional iteration for scalability and other logic structures and some strategies to avoid redundancy and increase performance in routine data checks. This paper explores the growing relevance of R in clinical programming and its potential to streamline processes across data cleaning, transformation, and validation.

INTRODUCTION

The pharmaceutical industry is undergoing a significant transformation in the way clinical data is analyzed and managed. With the growing complexity of clinical trials and the increasing volume of data generated, there is a heightened need for tools that enable flexibility, scalability, and reproducibility in statistical programming. Traditionally, SAS has been the dominant language for clinical programming; however, recent years have witnessed a steady shift toward open-source solutions, driven by their adaptability and cost-effectiveness. Among these solutions, R has emerged as a powerful and versatile language for data analysis and visualization. Its extensive ecosystem of packages, active community support, and integration capabilities make it particularly well-suited for modern clinical workflows. Regulatory agencies and industry consortia have also begun to recognize and endorse the use of R, further accelerating its adoption in clinical research environments.

Despite these advantages, transitioning to R presents unique challenges for programmers accustomed to legacy systems. Clinical data often involves diverse structures, stringent quality requirements, and complex decision logic, all of which demand efficient and reproducible programming practices. As organizations embrace R, understanding its role in improving data handling, compliance, and operational efficiency becomes critical for ensuring successful implementation. This paper highlights the broader implications of adopting open-source tools in regulated environments, emphasizing the need for robust workflows that balance innovation with compliance.

DYNAMIC DATA LOADING

In SAS, the LIBNAME statement provides a convenient way to reference an entire directory of datasets as a library. Once defined, all datasets within that library can be accessed by name without additional code. This feature simplifies workflows and reduces redundancy when working with multiple datasets.In contrast, R does not have an equivalent built-in mechanism. When using R, programmers often rely on functions like read_sas() from the haven package to import SAS datasets. However, this approach typically requires manual, repetitive code for each file as shown below:

```
dm <-read_sas("<library path>/dm.sas7bdat")

adsl <-read_sas("<library path>/adsl.sas7bdat")

adsl_s002 <-read_sas("<library path>/adsl_s002.sas7bdat")
```

This method is time consuming as each dataset requires a separate line of code, error-prone where manual typing increases the risk of typos and inconsistencies and is not scalable, that is, adding new datasets need to modify the code repeatedly.

To bring SAS-like flexibility into R and eliminate repetition, we can adopt a dynamic loading pattern that scans one or more directories for .sas7bdat files and loads them automatically into a named R list. The approach begins by specifying the relevant library paths and programmatically enumerating all matching files as shown in the code below:

```
paths <- c(  "<library path>", "<library path>", "<library path>")
 all_files <- unlist(lapply(paths, list.files, pattern = "\\.sas7bdat$", full.names = TRUE))
 data_list <- setNames(lapply(all_files,read_sas,tools::file_path_sans_ext (basename(all_files))))
adsl <- data_list[["adsl"]]
```

This sequence replaces repetitive import statements with a compact routine that discovers datasets, reads each file using read_sas(), and assigns the respective names derived from the filenames (for example, dm, adsl, adsl_s002). The code initially scans the directories and identifies all .sas7bdat files across the specified path and combines them into one character vector of full file paths. After creating the vector, it reads each SAS file into R and assigns list names without the .sas7bdat extension and pulls the dataset from the list created. The benefit of this code is that the new datasets appear in the workflow without requiring code changes, human error is reduced by removing repetitive lines and manual naming, and multiple library locations can be handled uniformly within the same construct. In practice, this pattern delivers a maintainable, scalable mechanism for clinical programming teams transitioning to R, enabling efficient ingestion of SDTM and ADaM domains while preserving clarity and reproducibility across evolving data deliveries.

HANDLING JOIN AND COLUMN OPERATIONS

When we join two data frames in R using dplyr—for example, left_join() or inner_join() the function needs to resolve clashes when the same variable name appears in both inputs but is not a join key. To avoid overwriting one source with the other, dplyr appends suffixes to the overlapping variable names, typically .x for the left data frame and .y for the right, so we might see SUBJID.x and SUBJID.y alongside the rest of the columns.

SUBJID.x	AGE.x	SEX.x	RACE.x	SUBJID.y	AGE.y	SEX.y	RACE.y
Subject Identifier for the Study	Age	Sex	Race	Subject Identifier for the Study	Age	Sex	Race
10011001	43	M	WHITE	*NA*	*NA*	*NA*	*NA*
10011002	53	M	WHITE	10011002	53	M	WHITE
10011002	53	M	WHITE	10011002	53	M	WHITE

However, in almost all downstream analytical or reporting steps, we want a single, clean column per variable rather than carrying two nearly redundant versions. That is where a systematic coalescing step becomes essential. A practical pattern is to identify all columns that end in .x, derive their base names by stripping the suffix, and then coalesce each pair of .x and .y into one consolidated column. Coalescing here means taking the value from the left-hand dataset when present and falling back to the right-hand dataset when the left value is missing, reflecting the usual preference to preserve the left source while still recovering information from the right. The following compact pattern implements this approach.

```
<input dataframe> <- <input dataframe> |>
(\(df) {bases <- names(df) |>
keep(~ str_ends(.x, ".x")) |>
str_remove("\\.x$")
df |>
mutate(!!!set_names(
map(bases, ~ coalesce(df[[paste0(.x, ".x")]], df[[paste0(.x, ".y")]])), bases))
})() |>
select(-ends_with(".x"), -ends_with(".y"))
```

This coalescing code should be placed immediately after the join step, so that the joined data is normalized before any further transformation, derivation, or validation. The following code demonstrates the above-mentioned approach. First, it identifies all columns in the joined data that end with .x; these are placeholders for variables that exist in both source tables. By stripping the .x suffix, we recover the “base” variable names (for example, ASEV), which define the set of columns we intend to consolidate. Second, for each base name, we construct a new column by coalescing the two suffixed versions: if ASEV.x is non-missing, we take it; otherwise, we take ASEV.y. This strategy aligns with the semantics of a left join—prefer information from the left table but fill gaps from the right when available—while remaining general enough to be used after inner or full joins as well. Finally, we remove the temporary .x and .y columns after creating the consolidated variable, leaving the data frame tidy and ready for subsequent transformations, derivations, or checks.

In routine clinical programming, this significantly simplifies post-join normalization, reduces the risk of accidental variable selection errors, and yields consistent, reproducible column handling across diverse datasets.

REGEX PATTERNS FOR DYNAMIC COLUMN SELECTION

In clinical datasets, variable names often follow conventions that embed meaningful tokens—domain prefixes, or date fragments such as DT, DTC, and DY. In many ADaM structures columns encode period-specific treatments and analysis flags using predictable naming patterns. Typical examples include treatment variables like TRT01P, TRT02P, TRT03P (period assignment) and TRT01A, TRT02A, TRT03A (actual treatment), as well as analysis flags such as ANL01FL, ANL02FL, ANL03FL.

Rather than enumerating columns manually, regex-based selection allows you to describe these naming rules once and let R dynamically discover the relevant variables. Every study may have a different number of periods, teams may choose zero-padding inconsistently, and new columns appear as deliveries evolve. This leads to repeated code edits, higher risk of typos, and difficulty keeping transformations reproducible across studies. In practice, this means using pattern expressions to select sets of columns by prefix, suffix, embedded token, or structured variants (for example, all date-like variables, all analysis values for a family of parameters, or all flags that end in FL). A robust alternative is to describe the naming convention once using regular expressions and let the code discover columns dynamically as shown below

```
<input dataframe> <- <input dataframe> %>%
  select(USUBJID,
    matches("^TRT\\d+P$"),   # TRT + digits + P  → captures TRT01P, TRT02P,...
    matches("^TRT\\d+A$"),   # TRT + digits + A  → captures TRT01A, TRT02A,...
    matches("^ANL\\d+FL$")   # ANL + digits + FL → captures ANL01FL, ANL02FL,...
```

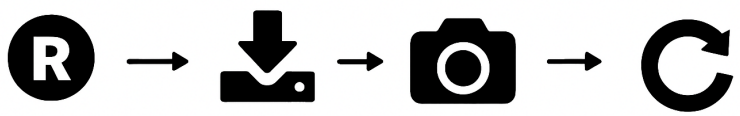
In dplyr::select(), the matches() helper interprets a string as a regular expression, giving us precise control. Anchoring the expression with ^ (start of name) and $ (end of name) prevents accidental partial matches, while tokens such as TRT or ANL define the prefixes, and \\d+ (one or more digits) captures the period or index—whether it is zero-padded. Finally, constraining the suffix to either P, A, or FL ensures we only select the intended period assignment, actual treatment, and analysis flag columns. This single, declarative selection achieves what manual lists cannot: it automatically includes newly delivered periods (e.g., TRT04P, TRT04A) and flags (e.g., ANL04FL) the moment they appear, with no edits required.

Beyond these examples, there are many other regex patterns that can be applied to clinical programming workflows. Common constructs include ^ for the start of a string, $ for the end, . for any character, \\d for digits, and \\w for word characters. Quantifiers like + (one or more), * (zero or more), and ? (optional) allow flexible matching as these patterns enable powerful column selection strategies

VERSION CONTROL FOR REPRODUCIBILITY

Reproducibility is not just the best practice it is a regulatory expectation. Analyses must produce consistent results across teams, machines, and time, even as software environments evolve. However, R packages and their dependencies change frequently, and without version control, the same code may yield different outputs or fail entirely when executed on another system. This variability introduces audit risks and undermines confidence in the integrity of clinical workflows.

Version control for R environments addresses these challenges by locking package versions and their dependency graph at a specific point in time. This ensures that the analytical pipeline remains stable and traceable, regardless of future updates to CRAN or internal repositories. It also supports collaboration by providing a mechanism for teams to share identical environments, and it improves auditability by documenting the exact configuration used for any given analysis. The renv package is the most widely adopted solution for environment management in R. It creates a project-specific library and a lockfile (renv.lock) that records the precise versions of all packages in use.



`renv::init()` `renv:install()` `renv:snapshot()` `renv:restore()`

```
# Initialize renv for the project
renv::init()

# Install required packages
renv::install

# Capture the environment state
renv::snapshot()

# Restore environment on another machine
renv::restore()
```

The workflow is simple yet powerful:

- renv::init() initializes a local environment for the project and generates the lockfile.
- renv::install() install packages into this isolated environment.
- renv::snapshot() captures the current state of all packages and writes it to renv.lock.
- renv::restore() recreates the environment on any machine by installing the versions recorded in the lockfile.

To ensure reproducibility and compliance in clinical environments, always commit the renv.lock file to version control so the environment can be restored consistently. Use project-specific libraries instead of global installations to avoid conflicts and run renv::snapshot() frequently after package changes to keep the lockfile current. Before validations or submissions, execute renv::restore() to confirm the environment matches the validated state.

This approach guarantees that analyses remain reproducible across time and systems, prevents unexpected changes due to package updates, and meets traceability requirements for regulatory compliance. By embedding renv into clinical programming workflows, teams can confidently share code and results, knowing that the underlying environment is fully controlled and auditable.

FUNCTIONAL ITERATION FOR SCALABILITY

Clinical programming workflows frequently involve performing the same transformation, summary, or validation logic across multiple datasets, treatments, parameters. A common but limited approach is to rely on explicit loops or repeated code blocks, which quickly become difficult to read, maintain, and scale as new requirements are introduced. As studies grow in complexity, such patterns increase the likelihood of errors. The table below illustrates a typical clinical safety reporting output, Adverse Events by Maximum Severity, where adverse events are summarized by System Organ Class and Preferred Term, and stratified across multiple treatments (Drug A, Drug B, Drug C) and severity categories (Mild, Moderate, Severe). Each cell in the table represents the result of the same underlying logic—deriving subject counts and percentages—for a specific combination of treatment and severity.

Adverse Events by Maximum Severity

	Drug A			Drug B			Drug C		
SYSTEM ORGAN CLASS Preferred Term	Mild	Moderate	Severe	Mild	Moderate	Severe	Mild	Moderate	Severe
CARDIAC DISORDERS	xx (xx.x)	xx (xx.x)	xx (xx.x)	xx (xx.x)	xx (xx.x)	xx (xx.x)	xx (xx.x)	xx (xx.x)	xx (xx.x)
Atrioventricular block	xx (xx.x)	xx (xx.x)	xx (xx.x)	xx (xx.x)	xx (xx.x)	xx (xx.x)	xx (xx.x)	xx (xx.x)	xx (xx.x)
GASTROINTESTINAL DISORDERS	xx (xx.x)	xx (xx.x)	xx (xx.x)	xx (xx.x)	xx (xx.x)	xx (xx.x)	xx (xx.x)	xx (xx.x)	xx (xx.x)
Salivary hypersecretion	xx (xx.x)	xx (xx.x)	xx (xx.x)	xx (xx.x)	xx (xx.x)	xx (xx.x)	xx (xx.x)	xx (xx.x)	xx (xx.x)
Glossodynia	xx (xx.x)	xx (xx.x)	xx (xx.x)	xx (xx.x)	xx (xx.x)	xx (xx.x)	xx (xx.x)	xx (xx.x)	xx (xx.x)
INFECTIONS AND INFESTATION	xx (xx.x)	xx (xx.x)	xx (xx.x)	xx (xx.x)	xx (xx.x)	xx (xx.x)	xx (xx.x)	xx (xx.x)	xx (xx.x)
Influenza	xx (xx.x)	xx (xx.x)	xx (xx.x)	xx (xx.x)	xx (xx.x)	xx (xx.x)	xx (xx.x)	xx (xx.x)	xx (xx.x)
Gastroenteritis	xx (xx.x)	xx (xx.x)	xx (xx.x)	xx (xx.x)	xx (xx.x)	xx (xx.x)	xx (xx.x)	xx (xx.x)	xx (xx.x)

Instead of writing separate logic for Drug A–Mild, Drug A–Moderate, Drug A–Severe, and so on, a functional programming provides a scalable alternative by shifting the focus from iterating over objects to iterating over parameters. In R, this paradigm is supported by the purrr package, which offers a family of mapping functions designed to apply logic consistently and efficiently across structured inputs. Rather than duplicating similar code for each scenario, programmers can define the logic once and systematically apply it over all required combinations as shown in below code.

	TRTA	TRTAN	SEVERITY
1	DRUG A	1	MILD
2	DRUG A	1	MODERATE
3	DRUG A	1	SEVERE
4	Drug B	2	MILD
5	Drug B	2	MODERATE
6	Drug B	2	SEVERE
7	Drug C	3	MILD
8	Drug C	3	MODERATE
9	Drug C	3	SEVERE

```
param_grid <- tibble(
  TRTA = list(
    "DRUG A",
    "Drug B",
    "Drug C"),
  TRTAN = c(1, 2, 3)) %>%
  tidyr::crossing(
    SEVERITY = list(
    "MILD",
    "MODERATE",
    "SEVERE"))

df <- pmap_dfr(param_grid, <analysis_fn>)
```

In this structure, param_grid defines the parameter combinations, and analysis_fn is the function created to calculate counts, percentages for each treatment–severity combination. The _dfr suffix ensures that outputs are automatically row-bound into a tidy data frame, eliminating the need for post-processing steps. This approach offers several advantages in clinical programming contexts. First, it significantly improves **code readability and maintainability**. All analytical logic resides in one function, making updates or corrections straightforward and auditable. Second, it enforces **consistency**, as every parameter combination is processed using identical logic, reducing the risk of discrepancies introduced by manual edits. Third, it enhances **scalability**—new treatments, categories, or parameters can be incorporated simply by extending the parameter grid, without modifying the core computation.

Functional iteration also aligns well with validation and regulatory expectations. By using a declarative parameter grid and a single controlled function, the transformation pathway for each output becomes transparent and reproducible. This structure simplifies code review, facilitates peer validation, and supports traceability across evolving study designs. In practice, replacing traditional loops with functional iteration enables clinical programmers to build robust, extensible workflows that adapt seamlessly to changing data requirements. As clinical datasets grow larger and more complex, this paradigm becomes essential for sustaining efficient, maintainable, and compliant R-based programming solutions.

CONCLUSION

Transitioning to R in clinical programming offers significant advantages when approached with the right strategies. Automating repetitive tasks through functional programming and regex improves efficiency, reducing manual coding and errors. Implementing version control with tools like renv ensures reproducibility, meeting regulatory and audit requirements by locking environments for consistent results. Techniques such as dynamic dataset handling and .x/.y coalescing enhance scalability, allowing workflows to adapt seamlessly to evolving data structures. Optimizing processes to avoid redundancy and streamline checks boosts performance, making large-scale clinical data handling more robust. functional iteration illustrated how parameter-driven workflows can replace repetitive logic when generating complex clinical tables, improving consistency, maintainability, and scalability.

Final Thought: Mastering these practices not only simplifies the transition from SAS to R but also empowers clinical programmers to build workflows that are efficient, compliant, and future-ready.

REFERENCES

- Wickham H, Henry L (2026). purrr: Functional Programming Tools. R package version 1.2.1, https://purrr.tidyverse.org/.
- Wickham H, Henry L (2026). renv: Introduction to renv. R package version 1.1.6, Introduction to renv • renv
- Pharmaverse: https://pharmaverse.org/
- R for Clinical Study Reports: https://r4csr.org/

ACKNOWLEDGMENTS

The author would like to thank the company Pfizer for providing an environment that supports continuous learning and innovation. Special thanks are extended to Debjit Biswas, Priya Venkatesan Iyer and Jeba Kumar for their support and encouragement throughout this work. Heartfelt thanks to Karthik P and Dhivya Kanagaraj for their invaluable guidance and constructive feedback throughout the development of this paper. Finally, the author wishes to thank A. Alex Jegadeep Raja. for his guidance and support throughout the submission process.

CONTACT INFORMATION

Your comments and questions are valued and encouraged.
Contact the author at:

Author name: Sanjana Jayacumar
Company: Pfizer Healthcare India Private Limited
Address: New No.237, Anna Salai, Thousand Lights, Chennai, Tamil Nadu 600014
Email: sanjana.jayacumar@pfizer.com