

6 Programming Foundations in R and Python for Health Data Science

Choosing (or Combining) R and Python

R and Python are both first-class languages for health data science, and this book uses both deliberately rather than favouring one, because Nigerian research groups, journals, and collaborators use both, and a data scientist who can read and write either is more employable and more collaborative. As a rough guide: R's tidyverse and Bioconductor ecosystems, along with its long-standing dominance in classical biostatistics (survival analysis, mixed models, epidemiological packages), make it a natural default for statistical inference and reproducible reporting (R Markdown/Quarto); Python's scikit-learn, pandas, and deep learning ecosystems (TensorFlow, PyTorch) make it a natural default for machine learning pipelines and production deployment. Chapters 9-15 of this book give code examples in whichever language best fits the technique, and often both, and Chapter 18 is devoted entirely to a deeper, standalone treatment of Python for disease epidemiology.

Core Data Structures

The data frame (R's 'data.frame'/'tibble', Python's pandas 'DataFrame') is the central object of health data analysis: a rectangular table where rows are observations (patients, facility-

months, survey clusters) and columns are variables. Mastery of a small set of operations — filtering rows, selecting columns, creating derived variables, grouping and summarising, and joining/merging tables on a shared identifier — covers perhaps 80% of the data manipulation that a working analyst performs.

R example

```
# R (tidyverse): computing state-level average TPR by quarter from facility data
library(dplyr); library(lubridate)

state_quarterly <- malaria_clean %>%
  mutate(quarter = paste0(year(report_month), "-Q", quarter(report_month))) %>%
  group_by(state, quarter) %>%
  summarise(mean_tpr = mean(tpr, na.rm = TRUE),
            total_tests = sum(tests_performed),
            n_facilities = n_distinct(facility_id),
            .groups = "drop")
```

PYTHON example

```
# Python (pandas): equivalent computation
import pandas as pd

df["quarter"] = pd.to_datetime(df["report_month"]).dt.to_period("Q")
state_quarterly = (
    df.groupby(["state", "quarter"])
      .agg(mean_tpr=("tpr", "mean"),
           total_tests=("tests_performed", "sum"),
           n_facilities=("facility_id", "nunique"))
      .reset_index()
)
```

Writing Reproducible Analysis Scripts

A script is reproducible when a colleague (or the author, six months later) can re-run it from raw data to final output and obtain identical results. This requires: never manually editing a dataset outside of code; setting a random seed wherever randomness is used (train/test splits, bootstrap resampling, Monte Carlo simulation — Chapter 13); recording package/library versions (via ‘renv’ in R or a ‘requirements.txt’/virtual environment in Python); and

structuring a project directory consistently (separate folders for raw data, cleaned data, scripts, and outputs). R Markdown and Jupyter/Quarto notebooks that interleave narrative, code, and output are the standard tools for producing a reproducible analytic report — the kind of artefact expected in a thesis chapter, a journal submission, or a technical report to the FMoH.

Functions, Iteration, and Avoiding Repetition

A recurring inefficiency in applied health analysis is copy-pasting the same block of cleaning or plotting code for each of 36 states or hundreds of facilities. Writing a function once, and applying it iteratively (via a loop, or more idiomatically via `'purrr::map()'` in R or a list comprehension/`'apply()'` in Python), is not merely a stylistic preference: it means a bug fix or a methodological refinement need only be made in one place, and it dramatically reduces the risk of an inconsistency creeping in between, say, the Lagos and Kano state analyses in a 13-state Global Fund malaria diagnostics report.

Worked Example

A Reusable Reporting Function Applied Across States

Suppose a monthly report must be produced for each of Nigeria's 36 states plus the FCT, each showing the same three summary figures (mean TPR, total tests, number of reporting facilities) and a small trend plot. Writing this once as a function, and applying it iteratively, both saves time and guarantees that every state's report is generated by exactly the same logic — eliminating the risk that a manual copy-paste error causes, say, Kano's report to accidentally use Lagos's date filter.

R example

```
# R: a reusable state reporting function applied iteratively with purrr
library(purrr); library(dplyr); library(ggplot2)

make_state_report <- function(state_name, data) {
  state_data <- data %>% filter(state == state_name)
  summary_stats <- state_data %>%
    summarise(mean_tpr = mean(tpr, na.rm = TRUE),
              total_tests = sum(tests_performed),
              n_facilities = n_distinct(facility_id))
  plot <- ggplot(state_data, aes(x = report_month, y = tpr)) +
    geom_line(stat = "summary", fun = mean, colour = "firebrick") +
    labs(title = paste("TPR Trend:", state_name))
  list(state = state_name, stats = summary_stats, plot = plot)
}

all_reports <- map(unique(malaria_clean$state), make_state_report, data = malaria_clean)
names(all_reports) <- unique(malaria_clean$state)
all_reports[["Kano"]]$stats # access Kano's summary statistics directly
```

PYTHON example

```
# Python: the equivalent reusable function applied via a dictionary comprehension
import matplotlib.pyplot as plt

def make_state_report(state_name, data):
    state_data = data[data["state"] == state_name]
    stats = {
        "mean_tpr": state_data["tpr"].mean(),
        "total_tests": state_data["tests_performed"].sum(),
        "n_facilities": state_data["facility_id"].nunique(),
    }
    return stats

all_reports = {s: make_state_report(s, df) for s in df["state"].unique()}
print(all_reports["Kano"])
```

Version Control Basics for Analysis Projects

Even a solo researcher benefits from basic version control (Git), because it provides a complete, timestamped history of every change to an analysis script, allows safe experimentation on a separate branch without risking a working version, and is the mechanism by which a supervisor, co-author, or reviewer can be given precise, auditable access to the exact code that produced a

given result. A minimal but sufficient workflow commits changes with a short, descriptive message after each meaningful unit of work (a completed cleaning step, a new model specification), rather than in one enormous commit at the end of a project.

R example

```
# Terminal (not R code): a minimal Git workflow for an analysis project
# git init                                # start version control in a project
#                                           folder
# git add cleaning_script.R              # stage a file
# git commit -m "Add facility-month cleaning and TPR plausibility flag"
# git log --oneline                      # view a compact history of all commits
```

Debugging

A Systematic Approach

Every analyst, regardless of experience, spends substantial time debugging code that does not run or that runs but produces an unexpected result. A systematic approach saves far more time than ad-hoc trial and error: first, read the actual error message carefully rather than skimming it, since it usually names the specific line and type of problem; second, isolate the smallest piece of code that reproduces the problem, rather than debugging inside a hundred-line script; third, check your assumptions about the data at the point of failure (print the object's dimensions, type, and a few values) rather than assuming the data looks the way you expect; and fourth, when a result runs without error but looks wrong, work backward from the output through each transformation step, checking intermediate results, rather than staring at the final number and guessing at the cause.

Extended Case Study

Refactoring an Ad-Hoc Script into a Maintainable Pipeline

Consider a realistic starting point: a 400-line R script, written under deadline pressure to produce a single technical report, that reads three different data files, applies a long sequence of manual

filtering and recoding steps directly in the main script body, produces six plots via copy-pasted and individually hand-edited ggplot2 calls, and saves a final Word document. The script works, in the narrow sense that it produced the report that was needed at the time, but six months later, when the same report needs to be regenerated with an additional quarter of data, the original author (or a colleague standing in for them) faces a script that is difficult to safely modify: it is unclear which of the many sequential filtering steps depend on assumptions specific to the original date range, the six plots' copy-pasted code makes it easy to update five of them correctly while accidentally leaving the sixth using stale logic, and there is no way to verify that the refactored script still produces the same results on the original data before trusting it on the new, extended data.

Refactoring this script following the principles of Chapter 6 involves several concrete steps: extracting the repeated plot-generation logic into a single parameterised function (Section 6.5's pattern) that takes a date range and a plot title as arguments, rather than six independently hand-edited copies; separating the data loading and cleaning steps into their own function with clearly named intermediate outputs, so that a colleague reviewing the code can see exactly what transformation happens at each stage; and, critically, before making any of these changes, running the original script's own results against the refactored version on the identical original dataset to confirm they produce bit-for-bit identical output — a regression test in the software engineering sense, applied here to a data analysis pipeline, which provides concrete evidence that the refactoring has not silently changed the analysis's substantive results. Only after this verification step does the refactored, function-based pipeline get used to generate the updated report with the additional quarter of data, at which point extending the date range and re-running the six plots requires changing a single date-range argument rather than hand-editing

six separate blocks of copy-pasted code — precisely eliminating the fragility that made the original ad-hoc script risky to maintain.

Choosing Between a Script and a Notebook

A recurring practical question for a Nigerian health data scientist setting up a new project is whether to work primarily in a plain script (a .R or .py file, run from start to finish) or an interactive notebook (R Markdown/Quarto, or Jupyter). Neither is universally superior: a notebook's ability to interleave narrative text, code, and rendered output makes it excellent for exploratory work (Chapter 5) and for producing a shareable analytic report with embedded explanation, but its non-linear, cell-by-cell execution model can silently produce results that depend on the order cells happened to be run in during an editing session, rather than the order they appear in the file — a subtle reproducibility risk unless the notebook is periodically re-run from a completely fresh session ('restart and run all') to confirm it still produces the same results top to bottom. A plain script, by contrast, always executes in a single fixed order and is generally preferred for production pipelines (Chapter 18's field-ready tools) precisely because it removes this specific risk, at the cost of losing the notebook's convenient interleaving of narrative and output during the exploratory phase of a project.

Testing Your Own Code

A Brief Introduction

Beyond the informal, ad-hoc checking implicit in the debugging approach of Section 6.7, a more systematic practice — increasingly common in serious applied data science work, though still under-used in much academic health research code — is writing explicit, automated tests for key functions: small pieces of code that check a function produces the expected, known-correct output for a specific, deliberately simple input where the

correct answer can be verified by hand. For example, a custom function computing test positivity rate with a plausibility flag (Section 4.4) could be tested by feeding it a small, hand-constructed dataset with a known TPR of exactly 50% and confirming the function returns exactly that value, and by feeding it a deliberately implausible case (more positives than tests performed) and confirming the plausibility flag correctly triggers. Writing even a handful of such tests for a project's most important, most-reused functions provides a fast, automatic way to confirm that a later change to the function (a bug fix, a refactoring per Section 6.8) has not silently broken its correct behaviour for the cases that matter most.

Continuous Integration for Analysis Pipelines

A Light-Touch Introduction

Continuous integration (CI) — automatically re-running a project's code (and, ideally, a small set of automated checks) every time a change is committed to version control — is standard practice in professional software engineering and is increasingly accessible to health data science projects through free platforms such as GitHub Actions. A light-touch CI setup for a typical analysis project might simply re-run the core cleaning and summary-statistics script on every commit and fail loudly if the script errors or if a key summary statistic (such as total row count) falls outside an expected range, catching an accidental regression (for example, a cleaning script silently breaking after an update to a package it depends on) far earlier than a manual, occasional re-run would.

Environment Management

Why 'It Works on My Machine' Is a Warning Sign

A script that runs correctly on the original analyst's computer but fails, or produces subtly different results, on a colleague's machine

is a common and specifically diagnosable reproducibility failure: differing package versions between the two environments. R’s ‘renv’ and Python’s virtual environments (or ‘conda’ environments) address this by recording the exact versions of every package used in a project and allowing any future user (including the original analyst, on a new machine) to recreate an identical computational environment with a single command, rather than manually reinstalling packages and hoping the versions happen to match. Treating ‘it works on my machine’ as a warning sign rather than a satisfactory endpoint — and routinely capturing the project’s environment specification alongside its code — is a small additional step that closes one of the most common and otherwise invisible gaps in analytic reproducibility.

Adding Unit Tests to the Reporting Function

Section 6.10 introduced testing as a brief orientation; applying it concretely to the reusable reporting function from Section 6.5 shows what a genuinely useful test looks like: not a test that checks the function runs without crashing, but one that checks the function returns the correct numeric answer on a small, hand-constructed input whose correct output the analyst already knows by hand-calculation, catching a logic error (such as an accidental sum instead of a mean) that a mere “does it run” check would miss entirely.

PYTHON example

```
# Python: a genuine unit test for the state reporting function (Section 6.5)
import pandas as pd

def test_make_state_report_computes_correct_mean():
    toy_data = pd.DataFrame({
        "state": ["Oyo", "Oyo", "Oyo"],
        "tpr": [0.2, 0.4, 0.6],
        "tests_performed": [100, 100, 100],
        "facility_id": ["F1", "F2", "F3"],
    })
    result = make_state_report("Oyo", toy_data)
    assert abs(result["mean_tpr"] - 0.4) < 1e-9, "Mean TPR calculation is incorrect"
    assert result["n_facilities"] == 3

test_make_state_report_computes_correct_mean() # raises AssertionError if the logic regresses
```

Chapter Summary

- R and Python are complementary tools; this book uses both, matched to the strength of each ecosystem, with a dedicated Python chapter (18) at the close of the book.
- The data frame and a small set of manipulation verbs (filter, select, mutate, group/summarise, join) underlie most applied analysis.
- Reproducibility requires scripted (not manual) workflows, fixed random seeds, recorded dependencies, and consistent project structure.
- Functions and iteration prevent the errors and inconsistency that arise from copy-pasted, state-by-state or facility-by-facility code.
- Basic version control (Git) provides an auditable history of an analysis project and supports safe collaboration.

Mini-Project

Mini-Project 6: A Multi-State Automated Reporting Pipeline

Build a small pipeline (R or Python) that automatically generates a one-page summary for each of at least five states/groups in a dataset, using a single reusable function.

Step 1. Simulate or obtain a facility-month dataset covering at least five states.

Step 2. Write a single function (following Section 6.5) that computes three summary statistics and produces one plot for a given state.

Step 3. Apply the function iteratively (via `map()`/`apply()`), not copy-paste) across all states, storing the results in a list or dictionary.

Step 4. Extract and display the results for two specific states to demonstrate the pipeline works correctly.

Step 5. Initialise a Git repository for your project (if not already done) and make at least three meaningful, separately-described commits as you build the pipeline.

Deliverable: *The reusable function, its iterative application across states, and a short Git commit history (a screenshot or ``git log`` output is sufficient).*

End-of-Chapter Exercises

1. Convert a manual, spreadsheet-based process you currently use for a repeated task (e.g., producing a monthly report

for each LGA) into a plan for a scripted, reproducible pipeline.

2. Write an R or Python function that takes a facility-month data frame and returns TPR with a flag for implausible values (see Section 4.4), then apply it across multiple states using iteration rather than copy-paste.
3. Explain, to a non-programmer colleague, why setting a random seed matters for a machine learning pipeline that will be run again next month on updated data.
4. In the worked example of Section 6.5, explain what would go wrong (in terms of reproducibility and correctness) if each state's report were instead produced by copy-pasting and manually editing the filter condition 36 times.
5. Using the Python code in Section 6.5, modify `'make_state_report()'` to also return the number of months with zero reported tests, and explain why this additional check might be useful for a state-by-state HMIS data quality audit.
6. Explain, in your own words, the difference between committing all your work in one single Git commit at the end of a project versus committing incrementally after each meaningful step.
7. A colleague says version control is 'only for software engineers, not for a statistics thesis.' Write a short response, drawing on the reproducibility arguments made across Chapters 4 and 6.
8. Using the systematic debugging approach in Section 6.7, describe the first two steps you would take if a script produced an error you did not immediately understand.
9. Explain why isolating the smallest piece of code that reproduces a bug is more efficient than debugging within a full, long script.

10. A script runs without any error but produces an obviously wrong summary statistic. Using Section 6.7's fourth step, describe your debugging approach.
11. In the Section 6.8 case study, explain why running the refactored pipeline against the original script's results on the SAME original data was an essential verification step, not an optional extra.
12. Describe one risk that would remain even after this verification step, when the refactored pipeline is later run on the new, extended quarter of data.
13. Why does extracting the six copy-pasted plots into a single parameterised function reduce the risk of the specific 'five out of six updated correctly' failure mode described in Section 6.8?
14. If you inherited a 400-line ad-hoc script like the one in Section 6.8, would you refactor the whole script at once or incrementally? Justify your choice.
15. Explain, in your own words, what a 'regression test' means in the data-analysis context described in Section 6.8, as distinct from its meaning in Chapter 8's regression modelling.
16. Using Section 6.9, explain the specific reproducibility risk introduced by a notebook's non-linear, cell-by-cell execution model.
17. Why does 'restart and run all' meaningfully reduce the risk described in Section 6.9, compared to simply saving the notebook after editing?
18. For which of the tasks in this book (e.g., exploratory analysis, a field-ready SEIR tool) would you personally choose a notebook, and for which a plain script? Justify your choices.

19. Propose one hybrid workflow that gets some of the benefits of both notebooks and scripts for a single project.
20. Explain to a student who has only ever used notebooks why a production tool (like the Section 18.11 dashboard) is more often built as a script or application than as a notebook.
21. Using Section 6.10, write (in words) two test cases you would use to check a function that calculates case fatality ratio (Section 18.5).
22. Why is testing a function with a deliberately implausible input (e.g., more deaths than total cases) useful, in addition to testing with a normal, expected input?
23. Explain how automated tests specifically support the refactoring process described in Section 6.8.
24. Why are automated tests described as ‘under-used’ in academic health research code, and what barrier do you think most contributes to this?
25. Propose one function from any chapter in this book that you would prioritise writing a test for first, and explain your reasoning.
26. Summarise, in your own words, why this chapter treats programming practice (functions, version control, testing) as inseparable from good statistical analysis.
27. Explain the common thread linking the refactoring case study (Section 6.8) and the notebook-versus-script discussion (Section 6.9).
28. Why does this chapter argue that reproducibility habits benefit even a solo researcher, not just large teams?
29. Propose one habit from this chapter (functions, testing, version control, or documentation) you consider most valuable for your own current work, and explain why.

30. If you had to teach this chapter's single most important lesson to a new graduate student in one sentence, what would it be?
31. Explain, using Section 6.12, why 'it works on my machine' is a warning sign rather than a satisfactory outcome.
32. What specific information does an environment specification (via renv or a Python virtual environment) capture that plain code does not?
33. Describe a scenario where a package version mismatch between two collaborators' machines could silently produce different analysis results.
34. Why might capturing an environment specification be considered as important as version-controlling the code itself?
35. If you have never used an environment management tool before, what is the first concrete step you would take to start?

Part 3

Statistical Inference and Modelling

