Python for R Programmers

A One-Week Boot Camp for R Users Learning Python

The rgtlab Curriculum Project

2026-11-13

Python for R Programmers

A One-Week Boot Camp

The rgtlab Curriculum Project

Welcome

This is the online version of **Python for R Programmers: A One-Week Boot Camp for R Users Learning Python** by The rgtlab Curriculum Project. We have written it for statisticians and data analysts who are already fluent in R and want to become productive in Python.

We begin from what a working R user needs in order to read, write, and run everyday scientific Python: the Python mental model and how it differs from R; the core data structures and control flow; NumPy and pandas for numerical and tabular work; the visualization and modeling stack; and reproducible project structure with interoperability back to R. The book is designed for **five consecutive days** of work, roughly one hour of lecture content each day and two hours of homework, with no examinations.

The organizing principle is translation. We assume the reader already knows the R idiom, so we introduce every Python construct by contrast with its R counterpart. That is, we do not teach programming from scratch. We teach a second language to a reader who already thinks in a first one. Each chapter opens with an R idiom the reader knows, develops the Python equivalent, and returns to compare what each language makes easy and what each makes awkward.

This workshop is a companion to a broader curriculum series:

- **R for Biostatistics: A One-Week Boot Camp** (https://r-bootcamp.rgtlab.org), the preparatory volume for the R language this workshop assumes.
- **Git and GitHub for Biostatistics: A One-Week Boot Camp** (https://git-bootcamp.rgtlab.org), the parallel preparatory volume for version control.
- *Statistical Computing in the Age of AI*, introductory methods (linear models, GLM, mixed models, survival, Bayesian, bootstrap, simulation).

- *Applied Generative AI for Health Sciences Research*, capability classes, RAG, agents, evaluation, regulation, deployment.

A reader who has completed *R for Biostatistics* has the R fluency this workshop assumes. A reader who completes this workshop should be able to move between the two languages as a given problem demands.

The reader is referred to the Preface for the design rationale and to the Conventions page for the visual cues used throughout.

License

This book is licensed under Creative Commons Attribution 4.0 International.

The code samples printed in this book are licensed under the MIT License. The repository code (package functions and build scripts) is licensed under GPL-3.

Python for R Programmers

A One-Week Boot Camp for R Users Learning Python.

Draft, November 13, 2026. Content may change.

Copyright

Python for R Programmers: A One-Week Boot Camp by rgtlab curriculum team is licensed under a Creative Commons Attribution 4.0 International License.

The code samples printed in this book are licensed under the MIT License. The repository code (package functions and build scripts) is licensed under GPL-3.

To cite this book, please use:

> rgtlab curriculum team (2026). *Python for R Programmers: A One-Week Boot Camp*. Available at https://python-for-r.rgtlab.org.

Table of contents

License 2

Copyright **5**

Preface **11**
Why an R user should learn Python 11
What this book covers 12
What this book does not cover 13
How this book is meant to be used 13
Prerequisites 14
Acknowledgments 14

Conventions **15**
Code and output 15
Callouts 16
Naming the two languages 16
Cross-references 17
Chapter structure 17

How to use this book **19**
The daily cadence 19
Chapter template 19
A note on translation 20
Conventions 20
Companion volumes 21

I. Five Days **23**

1. Day 1: Orientation and Setup **25**
1.1. Learning objectives 25

1.2. Lecture . . . 25
1.2.1. The Python mental model for R users . . . 26
1.2.2. Installing Python . . . 26
1.2.3. Virtual environments versus renv . . . 26
1.2.4. The REPL, scripts, and notebooks . . . 27
1.2.5. import versus library() . . . 28
1.2.6. Key differences that trip up R users . . . 28
1.3. Worked example: from an R script to a first Python session 29
1.4. Homework . . . 30
1.5. Solutions . . . 31
1.6. What's next . . . 32

2. Day 2: Data Structures and Control Flow 33
2.1. Learning objectives . . . 33
2.2. Lecture . . . 33
2.2.1. list, tuple, dict, set . . . 34
2.2.2. Strings . . . 34
2.2.3. Comprehensions versus vectorization and purrr . . . 35
2.2.4. Defining functions . . . 35
2.2.5. Loops and iterables . . . 36
2.2.6. Truthiness, None, NA, NULL . . . 36
2.3. Worked example: tallying subjects by arm . . . 37
2.4. Homework . . . 37
2.5. Solutions . . . 38
2.6. What's next . . . 39

3. Day 3: NumPy and pandas 41
3.1. Learning objectives . . . 41
3.2. Lecture . . . 41
3.2.1. NumPy arrays versus R vectors and matrices . . . 41
3.2.2. pandas Series and DataFrame versus the tibble . . . 42
3.2.3. Indexing with loc and iloc versus filter and select . . . 43
3.2.4. groupby versus group_by and summarize . . . 43
3.2.5. merge versus joins . . . 43
3.2.6. Missing data . . . 44
3.2.7. Method chaining versus the pipe . . . 44
3.3. Worked example: a grouped summary end to end . . . 45
3.4. Homework . . . 46

Draft, November 13, 2026. Content may change.

3.5. Solutions 47
3.6. What's next 47

4. Day 4: Visualization, Modeling, and the Scientific Stack 49
4.1. Learning objectives 49
4.2. Lecture 49
4.2.1. matplotlib, seaborn, and plotnine versus ggplot2 . . 49
4.2.2. statsmodels formula API versus lm and glm 50
4.2.3. scikit-learn basics 51
4.2.4. scipy.stats 51
4.2.5. reticulate for interoperability 51
4.3. Worked example: a fitted model and its diagnostic plot . . 52
4.4. Homework 53
4.5. Solutions 54
4.6. What's next 54

5. Day 5: Reproducible Python Projects and Interoperability 55
5.1. Learning objectives 55
5.2. Lecture 55
5.2.1. Project structure 56
5.2.2. Environments and lockfiles 56
5.2.3. Testing with pytest 57
5.2.4. Quarto with Python 57
5.2.5. Calling R from Python and Python from R 58
5.2.6. A note on migration strategy 58
5.3. Worked example: migrating a small R analysis to Python . 59
5.4. Homework 60
5.5. Solutions 61
5.6. What's next 61

References 63

Appendices 65

Credits 65

Colophon 67

Preface

This volume is a one-week workshop for readers who are already comfortable in R and want to become productive in Python. We assume the R fluency developed in the companion volume *R for Biostatistics*. Where that book makes a student functional in R, this one makes an R user functional in a second language. We do not assume any prior Python.

The design constraint here is deliberate. Five lectures and ten hours of homework should leave the reader able to read and write everyday scientific Python and to move an analysis between the two languages. We meet this constraint by leaning on what the reader already knows. Python is a large language, and we omit the great majority of its surface area. What remains is the subset that a practicing data analyst uses daily, taught by contrast with the R the reader has already internalized.

Why an R user should learn Python

The case for a second language is not that Python is better than R. It is, rather, that the two languages have different centers of gravity. A researcher who commands both can choose the right tool for a given task rather than forcing every task into one. Consider three situations a working analyst meets:

- A collaborator ships a model as a scikit-learn pipeline, and the analyst must read, run, and modify it rather than reimplement it.
- A data source arrives with a Python client library and no R equivalent, so the ingestion step is naturally written in Python.
- A deep-learning or large-language-model component has no mature R interface, and the surrounding analysis must meet it where it lives.

Each of these is a case where fluency in a second language removes a barrier rather than adds one. We note that the goal is not to abandon R. It is to make the boundary between the two languages a bridge the analyst can cross in either direction.

What this book covers

We organize the material as five days, one chapter each:

1. **Orientation and setup.** The Python mental model for R users; installing Python with `uv`, `venv`, and conda; the REPL, scripts, and notebooks; `import` versus `library()`; virtual environments versus `renv`; and the key differences (zero-indexing, mutability, significant whitespace, methods on objects rather than functions).
2. **Data structures and control flow.** `list`, `tuple`, `dict`, and `set` versus R vectors and lists; strings; comprehensions versus vectorization and `purrr`; defining functions with default and variadic arguments; loops and iterables; truthiness; and `None` versus `NA` and `NULL`.
3. **NumPy and pandas.** NumPy arrays versus R vectors and matrices; the pandas `Series` and `DataFrame` versus the tibble; `loc` and `iloc` versus `dplyr filter` and `select`; `groupby` versus `group_by` and `summarize`; `merge` versus joins; missing data; and method chaining versus the pipe.
4. **Visualization, modeling, and the scientific stack.** matplotlib, seaborn, and plotnine versus `ggplot2`; the statsmodels formula API versus `lm` and `glm`; the basics of scikit-learn; `scipy.stats`; and `reticulate` for interoperability.
5. **Reproducible projects and interoperability.** Project structure; environments and lockfiles with `uv`, `requirements.txt`, and conda; testing with `pytest` versus `tinytest`; Quarto with Python; calling R from Python and Python from R; and a worked migration of a small R analysis to Python.

Each chapter is approximately 1 hour of reading and worked examples plus 2 hours of homework problems with worked solutions provided.

Draft, November 13, 2026. Content may change.

What this book does not cover

We deliberately omit, and point elsewhere for, nearly everything beyond the daily core:

- Python-language internals (the data model in depth, metaclasses, descriptors, the C API), asynchronous programming, and packaging for distribution on PyPI.
- Deep learning and large-language-model frameworks (PyTorch, TensorFlow, JAX, the transformer libraries). See *Applied Generative AI for Health Sciences Research*.
- Production engineering (containers, web services, workflow orchestration), which belongs to a software-engineering course rather than a language workshop.
- The statistical methods themselves. This book teaches the Python that implements methods taught elsewhere; see *Statistical Computing in the Age of AI*.

The workshop is intentionally narrow. The reader finishes Day 5 able to work fluently in everyday scientific Python. This leaves the follow-on volumes free to teach the methods and systems that surround it.

How this book is meant to be used

The five-day cadence is the spine of the book. A reader starting the week having never run a Python script should be able to finish the week moving an analysis between R and Python. The cadence assumes roughly 3 hours of focused work per day:

- **Hour 1:** Read the chapter, running every example in your own Python session as you reach it. A second language is learned by typing, not by reading.
- **Hours 2-3:** Complete the homework problems. Solutions are at the end of each chapter; check yourself only after attempting each problem.

Draft, November 13, 2026. Content may change.

We show the examples in this book for a Unix-style shell (macOS Terminal, Linux, or a Unix shell on Windows such as Git Bash or WSL). Where a step differs on Windows, we note the difference.

Prerequisites

We assume:

- A working laptop running macOS, Windows, or Linux.
- Working fluency in R at the level of the companion volume *R for Biostatistics*: vectors, data frames or tibbles, writing functions, `dplyr` and `ggplot2`, and running a script from the console.
- Basic familiarity with the command line: opening a terminal, changing directories with `cd`, and listing files with `ls`.
- No prior exposure to Python.

Prior completion of *R for Biostatistics* is the natural entry point, since we introduce every Python construct here by contrast with its R counterpart. A reader without that R background can still follow the Python, but the translational scaffolding will do less work for such a reader.

Acknowledgments

This workshop consolidates patterns we developed while teaching R users to read and write scientific Python. Wes McKinney's *Python for Data Analysis* and Jake VanderPlas's *Python Data Science Handbook* provide the technical scaffolding. The translational pedagogy reflects our experience with what an established R user actually needs, as distinct from what a first-time programmer needs.

Draft, November 13, 2026. Content may change.

Conventions

This page summarizes the visual conventions used throughout the book.

Code and output

This book is example-first. Python code appears in syntax-highlighted blocks. Interactive sessions at the Python REPL show the `>>>` prompt for input; output appears on the following lines, without a prompt:

```
>>> 1 + 1
2
```

When a block is a script rather than an interactive session, the prompts are omitted and the code is shown as it would appear in a `.py` file:

```
import numpy as np
x = np.arange(10)
print(x.mean())
```

Shell commands appear in their own blocks. A command the reader types is shown after a `$` prompt; the prompt itself is not typed:

```
$ python --version
Python 3.12.4
```

Where a construct has a direct R counterpart, the R idiom is shown alongside the Python one for comparison:

```r
# R
x <- 1:10
mean(x)
```

Inline code, file names, function names, and method names are set in `monospace`. Python methods are written with a trailing pair of parentheses (`df.head()`) to distinguish them from attributes (`df.shape`).

Placeholders that the reader must replace with their own values are written in angle brackets, for example `pip install <package>` or `import <module>`. The angle brackets are not typed.

Callouts

Three callout types appear:

> **Tip**
>
> A small practical recommendation.

> **Check your understanding: example**
>
> A short question testing comprehension of the just-read material. Click to expand the answer.

> **Warning**
>
> A pitfall the reader may otherwise hit, most often a place where an R habit produces a surprising result in Python.

Naming the two languages

Throughout, we mark which language a construct belongs to whenever it could be ambiguous. We name Python objects and methods in full

(`list.append()`, not merely `append`) so that they remain unambiguous. We name R functions with their package where it matters (`dplyr::filter`), since the same bare name may mean different things in the two languages.

Cross-references

Within this book, sections are referenced by their Quarto label (`@sec-day1`, `@sec-pandas-groupby`). These resolve to clickable links in HTML and section numbers in PDF.

For references to the companion volumes *R for Biostatistics* and *Git and GitHub for Biostatistics*, we use **prose pointers** rather than Quarto cross-references, since cross-references do not resolve across separate books. For example: 'see the data-frames chapter of the companion *R for Biostatistics* volume'.

Chapter structure

Every content chapter follows the same five-section template:

1. **Learning objectives.** What you will be able to do after the day.
2. **Lecture.** The substantive content, with code to run in your own Python session as you read.
3. **Worked example.** A small but realistic scenario that uses the day's content end to end, contrasting the R idiom with the Python idiom.
4. **Homework.** Problems organized from easier to harder.
5. **Solutions.** Worked solutions to all homework problems. Read only after attempting the problem.

Each chapter closes with a short **What's next** pointer. The pattern repeats deliberately; by the third chapter you know where to find each component.

How to use this book

We begin with a short orientation. In this chapter we set out the daily cadence, the chapter template, and the conventions used throughout, so that the reader knows what to expect.

The daily cadence

Each day has the same shape:

- **Lecture content (~1 hour).** Read the chapter, running each example in your own Python session as you encounter it. Do not skip the code. The chapter is designed to be read with a REPL open and a scratch script at hand.
- **Homework (~2 hours).** Five to eight problems, with worked solutions at the end of the chapter. Attempt each problem before checking the solution.

The cadence assumes one chapter per day for five consecutive days. The workshop may be compressed (two chapters per day across two-and-a-half days) or expanded (one chapter per week across a month) depending on the reader's other commitments; we leave the choice to the reader.

Chapter template

Each chapter follows a five-section template, and we shall keep to it consistently:

1. **Learning objectives.** A bulleted list of what the reader should be able to do by the end of the day.

2. **Lecture.** The substantive content for the day, with code to run.
3. **Worked example.** A small but realistic scenario that uses the day's content end to end, contrasting the R idiom the reader already knows with the Python idiom being learned.
4. **Homework.** Problems organized from easier to harder.
5. **Solutions.** Worked solutions to all homework problems. Read only after attempting the problem.

There are no quizzes, no examinations, and no tests in this book. Self-assessment proceeds entirely through the homework solutions.

A note on translation

This book teaches Python to a reader who already thinks in R. That premise shapes every page:

- We rarely explain a programming concept from first principles. Instead we name the R construct the reader knows (a vector, a data frame, `dplyr::filter`, `purrr::map`) and present its Python counterpart beside it.
- Where the two languages agree, we say so briefly and move on. Where they differ in a way that trips up R users, we slow down. That is, the differences carry the weight, not the similarities.

Recall that the goal is not to decide which language is better. It is to make the reader able to move a given problem into whichever language serves it, and back again.

Conventions

The visual cues used throughout the book are described on the Conventions page. In brief:

- Interactive Python input follows a `>>>` prompt; shell commands follow a `$` prompt; output has no prompt.
- R code shown for comparison is labeled as such.
- Placeholders you replace are written in `<angle-brackets>`.

Draft, November 13, 2026. Content may change.

- Code lines and file listings are wrapped at 78 characters.

Companion volumes

This workshop is part of a broader curriculum series. The volumes most relevant to it are:

- *R for Biostatistics: A One-Week Boot Camp* (https://r-bootcamp.rgtlab.org), the source of the R fluency this workshop assumes.
- *Git and GitHub for Biostatistics: A One-Week Boot Camp* (https://git-bootcamp.rgtlab.org), the parallel preparatory volume for version control.
- *Statistical Computing in the Age of AI* (https://scai.rgtlab.org), introductory statistical methods and computing.
- *Applied Generative AI for Health Sciences Research* (https://applied-genai.rgtlab.org), generative AI for research.

The end of Chapter 5 describes how a small R analysis migrates to Python and where the reader can carry the two-language workflow next.

Part I.

Five Days

Draft, November 13, 2026. Content may change.

1. Day 1: Orientation and Setup

1.1. Learning objectives

Learning objectives

By the end of this day you should be able to:

- Describe the Python mental model and name the three or four places where it departs sharply from R.
- Install Python and create an isolated environment with `uv`, `venv`, or conda, and relate each to `renv`.
- Choose among the REPL, a script, and a notebook for a given task, and run each from the command line.
- Import functionality with `import` and explain how it differs from `library()`.
- Predict the effect of zero-indexing, mutability, and significant whitespace on code an R user would otherwise write by habit.
- Call a method on an object and explain how method syntax differs from R's function-call style.

1.2. Lecture

We begin with the mental model, because most early confusion for an R user is not syntax but a mismatch of expectations. We note four departures up front and return to each below.

1.2.1. The Python mental model for R users

R is a language built around vectors and functions that act on whole vectors at once. Python is a language built around objects that carry methods, where a plain number is a scalar and looping is ordinary. That is, R nudges you toward vectorized thinking; Python does not, though its numerical libraries restore it. We shall keep this contrast in view all week.

```
# Python: a scalar is a scalar, not a length-one vector
x = 3
type(x)
```

```
# R: everything is a vector; 3 is a length-one numeric vector
x <- 3
length(x)
```

1.2.2. Installing Python

We recommend `uv`, a fast installer and environment manager that plays the role `renv` plays in R. Install it, then use it to install a Python version.

```
# install uv (macOS/Linux)
curl -LsSf https://astral.sh/uv/install.sh | sh
uv python install 3.12
```

The older `venv` module ships with Python itself and is worth knowing, and conda remains common in scientific settings. We introduce all three so the reader can read any project they meet.

```
python -m venv .venv        # the standard-library approach
```

1.2.3. Virtual environments versus renv

An R user already understands the problem an environment solves: a project should pin its own package versions so that it does not break when another

Draft, November 13, 2026. Content may change.

project upgrades a dependency. That is exactly `renv`. In Python the same discipline is carried by a virtual environment, a self-contained directory of packages activated per project.

```
uv venv                          # create an environment in .venv
source .venv/bin/activate        # activate it (Unix shells)
uv pip install numpy pandas
```

> **Tip**
>
> Treat activating an environment as the Python equivalent of opening an `renv`-enabled project. Work outside an environment only for throwaway experiments.

1.2.4. The REPL, scripts, and notebooks

Python offers three ways to run code, and choosing well matters. The REPL is the interactive prompt, analogous to the R console. A script is a `.py` file run start to finish, analogous to `Rscript`. A notebook interleaves code and prose, analogous to a Quarto or R Markdown document.

```
python                           # start the REPL
python analysis.py               # run a script
```

We orient briefly on when to reach for each, and note that production analysis usually lives in scripts, not notebooks.

For an editor we recommend Positron, Posit's data science IDE, ahead of the classic Jupyter interface. Positron will feel familiar to an RStudio user. It offers the same script-and-console workflow, a Variables pane, and a Data Explorer for pandas data frames, and it edits and runs both `.py` scripts and `.ipynb` notebooks in one place. It is multilingual, so the reader's R and Python work can live in a single environment. A reader who prefers a notebook-only experience may still use Jupyter. For an R user coming from RStudio, however, we find Positron the more natural home, and it is the environment we recommend throughout this book.

1.2.5. import versus library()

In R, `library(dplyr)` attaches a package and makes its functions available by bare name. Python's `import` keeps names namespaced by default, which an R user should read as a feature rather than an inconvenience.

```
import numpy as np            # np.array(), np.mean(), ...
from math import sqrt         # sqrt() by bare name
```

```
library(dplyr)                # filter(), select() by bare name
```

That is, Python defaults to `package::function` style, where R defaults to attaching. We explain the trade-off and the common aliases (`np`, `pd`, `plt`) the reader will see everywhere.

1.2.6. Key differences that trip up R users

We collect here the departures most likely to produce a surprising result.

Zero-indexing. Python counts from zero, and ranges exclude their upper bound.

```
xs = [10, 20, 30]
xs[0]                         # 10, the first element
xs[0:2]                       # [10, 20], stops before index 2
```

```
xs <- c(10, 20, 30)
xs[1]                         # 10, R counts from one
```

Mutability. Many Python objects can be changed in place, and two names can refer to the same object. R's copy-on-modify semantics do not apply.

```
a = [1, 2, 3]
b = a                         # b and a are the same list
b.append(4)                   # a is now [1, 2, 3, 4] as well
```


Draft, November 13, 2026. Content may change.

> **Warning**
>
> Assigning `b = a` for a list does not copy it, unlike R. To copy, write `b = a.copy()` or `b = list(a)`. This is the single most common source of surprise for R users.

Significant whitespace. Indentation is syntax, not style. A block is defined by its indentation rather than by braces.

```
if x > 0:
    print('positive')       # the indentation defines the block
```

Objects and methods. Python calls many operations as methods on an object rather than as free functions.

```
s = 'hello'
s.upper()                   # method call: 'HELLO'
```

```
toupper('hello')            # R: a free function
```

1.3. Worked example: from an R script to a first Python session

We take a task an R user would write without thinking, reading a small vector and summarizing it, and reproduce it in Python to expose the four departures at once.

In R the reader would write:

```
ages <- c(34, 41, 29, 55, 38)
mean(ages)
ages[1]
```

The Python translation, run at the REPL, looks close but differs in indexing and in where the summary lives:

```
ages = [34, 41, 29, 55, 38]
sum(ages) / len(ages)       # no built-in vector mean yet
ages[0]                     # zero-indexed first element
```

We note that `sum(ages) / len(ages)` is deliberately clumsy: it shows that base Python lacks R's vectorized `mean`. On Day 3 we restore the convenience with NumPy, where `np.mean(ages)` recovers the R idiom exactly. For now the point is the mental model, not the ergonomics.

Finally, we package the same work as a script and run it from the shell, the Python counterpart of `Rscript summary.R`:

```
$ python summary.py
39.4
34
```

1.4. Homework

Attempt each problem in your own Python session before checking the solution. Create a scratch environment so that you can experiment freely.

1. **Install and verify.** Install `uv`, use it to install Python 3.12, and confirm the version with `python --version`. Report the exact version string you see.

2. **An isolated environment.** Create a virtual environment, activate it, install `numpy`, and confirm from the REPL that `import numpy` succeeds. State the R (`renv`) step that each command corresponds to.

3. **Indexing.** Given `xs = [5, 6, 7, 8, 9]`, write the expressions that return the first element, the last element, and the middle three elements. Explain how each differs from the R equivalent.

4. **Aliasing.** Predict the value of `a` after running `a = [1, 2]; b = a; b.append(3)`. Then predict it after `a = [1, 2]; b = a.copy(); b.append(3)`. Run both and explain the difference in terms of R's copy-on-modify.

Draft, November 13, 2026. Content may change.

5. **Whitespace.** Write a short `if/else` that prints `'even'` or `'odd'` for a variable `n`. Then break it deliberately by mis-indenting one line and record the error message Python reports.

6. **Methods versus functions.** For the string `s = 'Trial Data'`, find the method that lowercases it and the method that replaces spaces with underscores. Contrast the calling syntax with the R functions that do the same.

7. **REPL, script, notebook.** Write a two-line computation, run it three ways (REPL, `.py` script, and a notebook cell in Positron, or in Jupyter if you prefer), and note one advantage of each.

1.5. Solutions

Solutions

Problem 1. Placeholder: the version string has the form `Python 3.12.x`; any 3.11 or later suffices for this book.
Problem 2. Placeholder: `uv venv` corresponds to `renv::init`, activation corresponds to opening the project, and `uv pip install numpy` corresponds to `renv::install` plus `renv::snapshot`.
Problem 3. Placeholder: `xs[0]`, `xs[-1]`, and `xs[1:4]`; the R equivalents are `xs[1]`, `xs[length(xs)]`, and `xs[2:4]`, with the index base and half-open range as the differences.
Problem 4. Placeholder: `[1, 2, 3]` in the first case because `b` aliases `a`; `[1, 2]` in the second because `.copy()` makes an independent list, which matches R's default behavior.
Problem 5. Placeholder: the mis-indentation raises an `IndentationError` naming the offending line.
Problem 6. Placeholder: `s.lower()` and `s.replace(' ', '_')`; R uses `tolower(s)` and `gsub(' ', '_', s)`.
Problem 7. Placeholder: the REPL suits exploration, the script suits reproducible runs, and the notebook suits narrative reporting.

Draft, November 13, 2026. Content may change.

1.6. What's next

Day 2 fills in the everyday vocabulary of the language: the built-in data structures (`list`, `tuple`, `dict`, `set`), strings, comprehensions as the Python answer to vectorization and `purrr`, defining functions with default and variadic arguments, loops and iterables, truthiness, and how `None` relates to R's `NA` and `NULL`.

Draft, November 13, 2026. Content may change.

2. Day 2: Data Structures and Control Flow

2.1. Learning objectives

> **Learning objectives**
>
> By the end of this day you should be able to:
>
> - Choose among `list`, `tuple`, `dict`, and `set` and relate each to R's vectors, lists, and named lists.
> - Manipulate strings with Python's method-based string API.
> - Write a comprehension and explain how it substitutes for R vectorization and `purrr::map`.
> - Define functions with positional, keyword, default, and variadic (`*args`, `**kwargs`) arguments.
> - Write `for` and `while` loops over iterables and explain what makes an object iterable.
> - Reason about truthiness and about `None` in contrast with R's `NA` and `NULL`.

2.2. Lecture

We begin with the containers, since almost everything else builds on them. An R user should read this section as a map from four familiar structures to their Python counterparts.

2.2.1. list, tuple, dict, set

A Python `list` is an ordered, mutable sequence, closest to an R list but also doing the work of an atomic vector. A `tuple` is an ordered but immutable sequence. A `dict` is a mapping from keys to values, closest to a named R list. A `set` is an unordered collection of unique elements.

```
xs = [1, 2, 3]               # list: ordered, mutable
pt = (40.7, -74.0)           # tuple: ordered, immutable
ages = {'ann': 34, 'bo': 41}  # dict: key -> value
seen = {1, 2, 3}             # set: unique, unordered
```

```
xs <- list(1, 2, 3)          # R list
ages <- list(ann = 34, bo = 41)  # named list ~ dict
```

We note that a Python `list` is not vectorized: `xs * 2` repeats the list rather than doubling its elements. That surprise motivates NumPy on Day 3.

2.2.2. Strings

Python strings are objects with a rich method API, where R leans on functions from base and `stringr`.

```
s = 'Trial 001'
s.lower()                    # 'trial 001'
s.split(' ')                 # ['Trial', '001']
f'subject {s}'               # f-string interpolation
```

```
tolower('Trial 001')
strsplit('Trial 001', ' ')
```


Draft, November 13, 2026. Content may change.

2.2.3. Comprehensions versus vectorization and purrr

Because base Python does not vectorize, the idiomatic way to transform a sequence is a comprehension, which an R user should read as the counterpart of `purrr::map` or of vectorized arithmetic.

```
squares = [x**2 for x in range(5)]      # [0, 1, 4, 9, 16]
evens = [x for x in range(10) if x % 2 == 0]
```

```
squares <- (0:4)^2                       # vectorized
evens <- purrr::keep(0:9, ~ .x %% 2 == 0)
```

> **Tip**
>
> Read a list comprehension right after its opening bracket: the expression on the left is 'what to keep', the `for` clause is 'over what', and the optional `if` clause is 'when'. This maps cleanly onto `map` plus `filter`.

2.2.4. Defining functions

Functions are defined with `def`. Arguments may be positional or keyword, may carry defaults, and may be gathered variadically with `*args` and `**kwargs`.

```
def summarize(x, trim=0.0, *args, **kwargs):
    # body indented; last expression is not auto-returned
    return sum(x) / len(x)
```

```
summarize <- function(x, trim = 0, ...) {
  mean(x, trim = trim)
}
```


Draft, November 13, 2026. Content may change.

> **Warning**
>
> Python does not return the last expression implicitly; an R habit of relying on that will return `None`. Write an explicit `return`.

We also note the classic pitfall of a mutable default argument and show the safe `None` sentinel pattern.

2.2.5. Loops and iterables

Looping is ordinary and idiomatic in Python, unlike in R where it is often avoided in favor of vectorization. An object is iterable if it can be consumed by a `for` loop.

```
for name, age in ages.items():
    print(name, age)

i = 0
while i < 3:
    i += 1
```

2.2.6. Truthiness, None, NA, NULL

Python treats empty containers, zero, and `None` as false in a boolean context. `None` is the single 'no value' object, and it is worth mapping carefully onto R's two distinct notions.

```
if not xs:                  # true when xs is empty
    print('empty')
value = None                # absence of a value
```

We note the mapping: Python's `None` is closest to R's `NULL` (a genuine absence), whereas R's `NA` (a present-but-missing measurement) has its most faithful counterpart later, in pandas and NumPy missing-data markers, covered on Day 3.

Draft, November 13, 2026. Content may change.

2.3. Worked example: tallying subjects by arm

We take a task an R user would solve with `table()` and a named list, counting trial subjects by treatment arm, and write it in idiomatic Python using a `dict` and a comprehension.

In R:

```r
arms <- c('A', 'B', 'A', 'A', 'B')
table(arms)
```

In Python we build the tally explicitly, which exposes the `dict` and the loop, and then show the concise idiom:

```python
arms = ['A', 'B', 'A', 'A', 'B']
counts = {}
for a in arms:
    counts[a] = counts.get(a, 0) + 1
counts                      # {'A': 3, 'B': 2}
```

The standard library offers the concise form, closer in spirit to `table()`:

```python
from collections import Counter
Counter(arms)               # Counter({'A': 3, 'B': 2})
```

We note the trade-off: the explicit loop teaches the mechanics, while `Counter` is what one writes in practice. Both return a `dict`-like object the reader can index by arm.

2.4. Homework

Attempt each problem before checking the solution.

1. **Container choice.** For each of the following, name the Python container you would use and its closest R analogue: an ordered set of subject IDs that will grow; a fixed coordinate pair; a lookup from subject ID to age; the set of distinct sites in a study.

2. **List is not a vector.** Predict the result of `[1, 2, 3] * 2` and explain why it differs from the R result of `c(1, 2, 3) * 2`.
3. **String methods.** Given `s = 'HbA1c_change'`, produce `'hba1c change'` using string methods only. State the `stringr` or base R calls that would do the same.
4. **Comprehension.** Write a comprehension that returns the squares of the even numbers in `range(20)`. Then write the equivalent using an explicit loop, and say which you find clearer.
5. **Default arguments.** Write a function `greet(name, greeting='Hello')` and call it both with and without the second argument. Then explain why `def f(x, acc=[])` is a trap and how to fix it.
6. **Variadic arguments.** Write a function that accepts any number of positional numbers and returns their mean, using `*args`. Relate `*args` to R's `...`.
7. **None versus NA.** Write an expression that is true when a variable `v` is `None`. Explain, in one or two sentences, why Python's `None` maps more naturally onto R's `NULL` than onto R's `NA`.

2.5. Solutions

Solutions

Problem 1. Placeholder: a `list` (growable IDs), a `tuple` (fixed pair), a `dict` (ID to age), and a `set` (distinct sites); the R analogues are a list or vector, a length-two vector, a named list, and `unique()`.
Problem 2. Placeholder: `[1, 2, 3, 1, 2, 3]`, list repetition rather than elementwise doubling, because base Python does not vectorize.
Problem 3. Placeholder: `s.lower().replace('_', ' ')`; R uses `gsub('_', ' ', tolower(s))`.
Problem 4. Placeholder: `[x**2 for x in range(20) if x % 2 == 0]`; the loop form appends inside a `for`.
Problem 5. Placeholder: the default list is created once and shared across calls; use `acc=None` and set it inside.

Draft, November 13, 2026. Content may change.

Problem 6. Placeholder: `def mean(*xs): return sum(xs) / len(xs)`; `*args` gathers extra positionals as R's `...` does.
Problem 7. Placeholder: `v is None`; `None` denotes a genuine absence like `NULL`, not a missing measurement like `NA`.

2.6. What's next

Day 3 restores the vectorized comfort of R. We introduce NumPy arrays as the counterpart of R vectors and matrices, then pandas `Series` and `DataFrame` as the counterpart of the tibble, and translate `dplyr` verbs (`filter`, `select`, `group_by`, joins) into their pandas equivalents.

Draft, November 13, 2026. Content may change.

3. Day 3: NumPy and pandas

3.1. Learning objectives

Learning objectives

By the end of this day you should be able to:

- Create and operate on NumPy arrays and relate them to R vectors and matrices.
- Build a pandas `Series` and `DataFrame` and relate them to the R tibble.
- Select rows and columns with `loc` and `iloc` and relate them to `dplyr::filter` and `dplyr::select`.
- Aggregate with `groupby` and relate it to `group_by` and `summarize`.
- Combine tables with `merge` and relate it to `dplyr` joins.
- Handle missing data and read a method chain as the counterpart of the pipe.

3.2. Lecture

We begin with NumPy, because pandas is built on it and because it restores the vectorized arithmetic an R user expects.

3.2.1. NumPy arrays versus R vectors and matrices

A NumPy array is a homogeneous, n-dimensional block of numbers. A one-dimensional array plays the role of an R atomic vector; a two-dimensional

array plays the role of an R matrix. Unlike a Python `list`, an array vectorizes.

```python
import numpy as np
x = np.array([1, 2, 3])
x * 2                       # array([2, 4, 6]), elementwise
x.mean()                    # 2.0
```

```r
x <- c(1, 2, 3)
x * 2                       # 2 4 6
mean(x)
```

We note that array indexing is zero-based and that broadcasting, NumPy's rule for combining arrays of different shapes, is the analogue of R's recycling but stricter.

3.2.2. pandas Series and DataFrame versus the tibble

A pandas `Series` is a one-dimensional labeled array; a `DataFrame` is a two-dimensional table of columns, each a `Series`. The `DataFrame` is the working counterpart of the tibble, with one notable addition: an explicit row index.

```python
import pandas as pd
df = pd.DataFrame({
    'id': [1, 2, 3],
    'arm': ['A', 'B', 'A'],
    'age': [34, 41, 29],
})
df.head()
```

```r
df <- tibble::tibble(
  id = c(1, 2, 3),
  arm = c('A', 'B', 'A'),
  age = c(34, 41, 29)
)
```


Draft, November 13, 2026. Content may change.

> **Warning**
>
> The pandas row index is a feature with no clean tibble counterpart. Many surprises for R users trace back to an index that is carried along silently; `reset_index()` is often the fix.

3.2.3. Indexing with loc and iloc versus filter and select

pandas separates label-based selection (`loc`) from position-based selection (`iloc`). Row filtering uses a boolean mask, which reads much like `dplyr::filter`; column selection by name reads like `dplyr::select`.

```
df.loc[df['age'] > 30]              # filter rows
df.loc[:, ['id', 'arm']]            # select columns
df.iloc[0:2]                        # first two rows by position
```

```
dplyr::filter(df, age > 30)
dplyr::select(df, id, arm)
```

3.2.4. groupby versus group_by and summarize

The split-apply-combine pattern an R user knows from `group_by` plus `summarize` is `groupby` plus an aggregation in pandas.

```
df.groupby('arm')['age'].mean()     # mean age by arm
```

```
df |>
  dplyr::group_by(arm) |>
  dplyr::summarize(age = mean(age))
```

3.2.5. merge versus joins

Table joins use `merge`, whose `how` argument names the join type an R user knows from `dplyr::left_join` and its relatives.

```
pd.merge(df, sites, on='id', how='left')
```

```
dplyr::left_join(df, sites, by = 'id')
```

3.2.6. Missing data

pandas marks missing values with `NaN` (and, in newer types, `pd.NA`), the counterpart of R's `NA`. We note that most pandas aggregations skip missing values by default, which matches the behavior of R functions called with `na.rm = TRUE`.

```
df['age'].isna().sum()              # count missing
df['age'].fillna(df['age'].mean()) # impute with the mean
```

3.2.7. Method chaining versus the pipe

Where R threads a data frame through `dplyr` verbs with the pipe, pandas threads a `DataFrame` through methods with a chain. Wrapping the chain in parentheses lets it span lines readably.

```
result = (
    df
    .loc[df['age'] > 30]
    .groupby('arm')['age']
    .mean()
)
```

```
result <- df |>
  dplyr::filter(age > 30) |>
  dplyr::group_by(arm) |>
  dplyr::summarize(age = mean(age))
```


Draft, November 13, 2026. Content may change.

> **Tip**
>
> Read a pandas method chain exactly as you read a `dplyr` pipeline: top to bottom, one verb per line. The parentheses are only there to let the chain break across lines.

3.3. Worked example: a grouped summary end to end

We take a task an R user would write as a `dplyr` pipeline, computing mean age by arm among subjects over thirty, and build it in pandas so the correspondence is explicit line by line.

The R version the reader already knows:

```r
enroll |>
  dplyr::filter(age > 30) |>
  dplyr::group_by(arm) |>
  dplyr::summarize(mean_age = mean(age), n = dplyr::n())
```

The pandas translation, built as a chain:

```python
summary = (
    enroll
    .loc[enroll['age'] > 30]
    .groupby('arm')
    .agg(mean_age=('age', 'mean'), n=('age', 'size'))
    .reset_index()
)
summary
```

We note two points of friction for an R user. First, the grouped result carries `arm` in its index, so we call `reset_index()` to return it to an ordinary column, recovering a tibble-shaped result. Second, the named aggregation syntax `agg(mean_age=('age', 'mean'))` is the pandas counterpart of naming summaries inside `summarize`. With those two mappings in hand, the pipeline and the chain say the same thing.

Draft, November 13, 2026. Content may change.

3.4. Homework

Attempt each problem before checking the solution. Use a small `DataFrame` of your own construction unless one is given.

1. **Array arithmetic.** Create a NumPy array of the integers 1 through 10 and compute its mean and standard deviation. Show the R vector code that does the same.

2. **Build a DataFrame.** Construct a `DataFrame` with columns `id`, `site`, and `weight` for five subjects. Display its shape and its column dtypes, and name the tibble code that corresponds.

3. **loc versus iloc.** From your `DataFrame`, select the rows where `weight` exceeds its mean using `loc`, and the first three rows using `iloc`. Explain the difference between label-based and position-based selection.

4. **groupby.** Compute the mean `weight` by `site`. Then add a count of subjects per site in the same result. Show the `dplyr` equivalent.

5. **merge.** Build a second table mapping `site` to a region and left-join it onto your subject table. Predict what happens to a subject whose site is absent from the mapping.

6. **Missing data.** Set one `weight` to `NaN`. Count the missing values, then compute the mean with and without the missing value included, and relate the behavior to `na.rm`.

7. **Method chain.** Rewrite problems 3 through 5 as a single parenthesized method chain and compare its readability with the equivalent `dplyr` pipeline.

Draft, November 13, 2026. Content may change.

3.5. Solutions

Solutions

Problem 1. Placeholder: `np.arange(1, 11)` with `.mean()` and `.std()`; R uses `1:10`, `mean`, and `sd`, noting that NumPy's default `std` divides by n rather than n - 1.
Problem 2. Placeholder: `pd.DataFrame({...})`; `.shape` and `.dtypes` report dimensions and column types, matching `dim()` and column classes on a tibble.
Problem 3. Placeholder: `df.loc[df['weight'] > df['weight'].mean()]` and `df.iloc[0:3]`; `loc` selects by label or mask, `iloc` by integer position.
Problem 4. Placeholder: `df.groupby('site').agg(mean_w=('weight', 'mean'), n=('weight', 'size'))`.
Problem 5. Placeholder: a `how='left'` merge; an unmatched site yields `NaN` in the region column.
Problem 6. Placeholder: `.isna().sum()` counts missing; the mean skips `NaN` by default, matching `na.rm = TRUE`.
Problem 7. Placeholder: a single chain wrapped in parentheses, one verb per line, read like a pipe.

3.6. What's next

Day 4 turns to the scientific stack: plotting with matplotlib, seaborn, and plotnine as the counterpart of `ggplot2`; modeling with the statsmodels formula API as the counterpart of `lm` and `glm`; the basics of scikit-learn; `scipy.stats`; and `reticulate` for calling Python from R.

Draft, November 13, 2026. Content may change.

4. Day 4: Visualization, Modeling, and the Scientific Stack

4.1. Learning objectives

Learning objectives

By the end of this day you should be able to:

- Produce a plot with matplotlib and seaborn, and reproduce a `ggplot2` figure with plotnine.
- Fit a linear or generalized linear model with the statsmodels formula API and read its summary against `lm` and `glm`.
- Fit and evaluate a simple model with scikit-learn and explain its estimator interface.
- Run common tests and distributions from `scipy.stats`.
- Call Python from R with `reticulate` and describe when interoperability is preferable to reimplementation.

4.2. Lecture

We begin with visualization, since it is where the R user's `ggplot2` fluency transfers most directly, then move to modeling.

4.2.1. matplotlib, seaborn, and plotnine versus ggplot2

Python offers three plotting styles worth knowing. matplotlib is the low-level engine, verbose but complete. seaborn is a statistical layer over it,

concise for common charts. plotnine is a near-literal port of `ggplot2`, so an R user can often paste a grammar-of-graphics figure with minimal change.

```python
import seaborn as sns
sns.scatterplot(data=df, x='age', y='weight', hue='arm')
```

```python
from plotnine import ggplot, aes, geom_point
(ggplot(df, aes('age', 'weight', color='arm')) + geom_point())
```

```r
ggplot2::ggplot(df, ggplot2::aes(age, weight, color = arm)) +
  ggplot2::geom_point()
```

> **Tip**
>
> If you think in the grammar of graphics, start with plotnine; the translation cost from `ggplot2` is nearly zero. Reach for seaborn when you want a common statistical chart in one line.

4.2.2. statsmodels formula API versus lm and glm

statsmodels offers a formula interface that an R user will find familiar, including the `y ~ x1 + x2` notation and a regression summary table.

```python
import statsmodels.formula.api as smf
model = smf.ols('weight ~ age + arm', data=df).fit()
model.summary()
```

```r
model <- lm(weight ~ age + arm, data = df)
summary(model)
```

Generalized linear models follow the same shape, with a family argument that parallels R's `family=`.

Draft, November 13, 2026. Content may change.

```
smf.glm('event ~ age', data=df,
        family=sm.families.Binomial()).fit()
```

4.2.3. scikit-learn basics

scikit-learn is oriented toward prediction rather than inference, and its estimators share one interface: construct, `fit`, then `predict`. An R user should read it as closer to `caret` or `tidymodels` than to `lm`.

```
from sklearn.linear_model import LinearRegression
X = df[['age']]
y = df['weight']
reg = LinearRegression().fit(X, y)
reg.predict(X)
```

We note that scikit-learn expects a two-dimensional feature matrix `X` and a one-dimensional target `y`, and that it does not print a coefficient table by default; its focus is the fitted predictor, not the inferential summary.

4.2.4. scipy.stats

`scipy.stats` provides distributions and classical tests, the counterpart of R's `d/p/q/r` functions and its test suite.

```
from scipy import stats
stats.ttest_ind(group_a, group_b)
stats.norm.ppf(0.975)              # ~ qnorm(0.975) in R
```

4.2.5. reticulate for interoperability

Interoperability is often preferable to rewriting. The R package `reticulate` lets an R session call Python directly, so a team can keep an analysis in R while borrowing a single Python component.

Draft, November 13, 2026. Content may change.

```
library(reticulate)
np <- import('numpy')
np$mean(c(1, 2, 3))
```

We note the design question this raises, and answer it: rewrite only when the component is small and central; otherwise call across the boundary and keep each piece in the language that serves it.

4.3. Worked example: a fitted model and its diagnostic plot

We take a task an R user would write with `lm` and `ggplot2`, fitting weight on age and arm and plotting the fit, and build it with statsmodels and plotnine so both halves mirror the R workflow.

The R version:

```
fit <- lm(weight ~ age + arm, data = enroll)
summary(fit)
ggplot2::ggplot(enroll, ggplot2::aes(age, weight)) +
  ggplot2::geom_point() +
  ggplot2::geom_smooth(method = 'lm')
```

The Python translation keeps the formula and the grammar of graphics intact:

```
import statsmodels.formula.api as smf
from plotnine import ggplot, aes, geom_point, geom_smooth

fit = smf.ols('weight ~ age + arm', data=enroll).fit()
print(fit.summary())

(
    ggplot(enroll, aes('age', 'weight'))
    + geom_point()
```


Draft, November 13, 2026. Content may change.

```
    + geom_smooth(method='lm')
)
```

We note that the formula string, the summary table, and the plot layers each have a one-to-one R counterpart. The reader who knows `lm` and `ggplot2` is, in effect, already reading this Python; only the surrounding syntax has changed.

4.4. Homework

Attempt each problem before checking the solution.

1. **A first plot.** Using seaborn, draw a scatterplot of two numeric columns colored by a categorical column. Then draw the same figure with plotnine and note which felt closer to `ggplot2`.

2. **Faceting.** Extend the plotnine figure with a facet by a grouping variable. Show the `ggplot2::facet_wrap` call it corresponds to.

3. **Linear model.** Fit a linear model with statsmodels using a formula with two predictors. Read off the coefficient estimates and compare the printed summary with `summary()` on the R fit.

4. **Logistic regression.** Fit a logistic regression with the statsmodels GLM interface. Name the `family` argument and its R counterpart.

5. **scikit-learn.** Fit a `LinearRegression` to the same data as problem 3. Extract the coefficients and explain why the scikit-learn output differs in emphasis from the statsmodels output.

6. **A classical test.** Use `scipy.stats` to run a two-sample t-test and to compute a normal quantile. State the R functions that match each.

7. **Interoperability.** From an R session, use `reticulate` to call a NumPy function on an R vector. Describe one situation where you would prefer this to rewriting the computation in R.

Draft, November 13, 2026. Content may change.

4.5. Solutions

Solutions

Problem 1. Placeholder: `sns.scatterplot(...)` versus a plotnine `ggplot(...) + geom_point()`; plotnine mirrors `ggplot2` most closely.
Problem 2. Placeholder: add `+ facet_wrap('~ group')`, matching `ggplot2::facet_wrap(~ group)`.
Problem 3. Placeholder: `smf.ols('y ~ x1 + x2', data=df).fit()`; the coefficient table lines up with `summary(lm(...))`.
Problem 4. Placeholder: `family=sm.families.Binomial()`, matching `glm(..., family = binomial)` in R.
Problem 5. Placeholder: `LinearRegression().fit(X, y)` exposes `.coef_` and `.intercept_`; scikit-learn emphasizes prediction, so it omits the inferential table.
Problem 6. Placeholder: `stats.ttest_ind` matches `t.test`; `stats.norm.ppf` matches `qnorm`.
Problem 7. Placeholder: `reticulate::import('numpy')` then call the function; prefer interoperability when the component is large or has no R equivalent.

4.6. What's next

Day 5 assembles the pieces into a reproducible project: directory structure, environments and lockfiles with `uv`, `requirements.txt`, and conda, testing with `pytest`, authoring with Quarto and Python, and calling R from Python and Python from R. We close with a worked migration of a small R analysis to Python.

Draft, November 13, 2026. Content may change.

5. Day 5: Reproducible Python Projects and Interoperability

5.1. Learning objectives

Learning objectives

By the end of this day you should be able to:

- Lay out a Python project directory and relate it to an R package or research compendium.
- Pin dependencies with `uv`, `requirements.txt`, and conda, and relate each to `renv.lock`.
- Write and run tests with `pytest` and relate them to `tinytest` and `testthat`.
- Author a Quarto document with Python code cells.
- Call R from Python and Python from R, choosing a direction for a given task.
- Migrate a small R analysis to Python end to end.

5.2. Lecture

We begin with project structure, since a reproducible analysis is a matter of layout and pinning before it is a matter of code.

5.2.1. Project structure

A Python analysis benefits from the same discipline an R user applies to a research compendium: a clear directory layout with code, data, and outputs separated. We describe a minimal layout and relate each part to its R counterpart.

```
project/
  pyproject.toml        # project + dependency metadata
  src/analysis/         # importable package code
  tests/                # pytest tests
  data/                 # inputs (often gitignored)
  notebooks/            # exploratory work
```

We note the parallel to an R package's `R/`, `tests/`, and `DESCRIPTION`, and to a zzcollab-style compendium.

5.2.2. Environments and lockfiles

Reproducibility requires pinning versions, the role `renv.lock` plays in R. Python offers several mechanisms, and the reader will meet all of them.

```
uv lock                         # write uv.lock (preferred)
uv pip freeze > requirements.txt   # a portable pin
conda env export > environment.yml # the conda equivalent
```

```
renv::snapshot()                # write renv.lock
```

> **Tip**
>
> Treat `uv.lock` (or a committed `requirements.txt`) as the Python `renv.lock`: commit it, and recreate the environment from it on another machine rather than reinstalling by hand.

Draft, November 13, 2026. Content may change.

> **A language-agnostic option: zzcollab**
>
> The rgtlab `zzcollab` tool (https://github.com/rgt47/zzcollab) builds a Docker-based reproducible compendium. Although it grew out of R practice, the container it produces is language-agnostic, so a reader who already uses it for R work can host a Python project in the same framework rather than maintaining two separate reproducibility stacks.

5.2.3. Testing with pytest

Automated tests are as valuable in Python as in R. `pytest` discovers functions named `test_*` and runs them, reporting failures. An R user should read it as the counterpart of `tinytest` or `testthat`.

```
# tests/test_summary.py
from analysis.summary import mean_age

def test_mean_age():
    assert mean_age([30, 40, 50]) == 40
```

```
# tinytest
expect_equal(mean_age(c(30, 40, 50)), 40)
```

```
pytest                          # discover and run all tests
```

5.2.4. Quarto with Python

Quarto renders documents with Python code cells exactly as it renders R cells, so the reader's Quarto experience transfers directly. A cell is marked with `{python}` in place of `{r}`.

```
::: {.cell execution_count=1}
``` {.python .cell-code}
import pandas as pd
```


Draft, November 13, 2026. Content may change.
```

```
pd.DataFrame({'x': [1, 2, 3]}).describe()
```
```
:::
```

We note that a single Quarto project can mix R and Python cells, which is often the cleanest path during a migration.

5.2.5. Calling R from Python and Python from R

Interoperability runs in both directions. From R, `reticulate` calls Python, as we saw on Day 4. From Python, the `rpy2` package calls R, so an established R routine can be invoked without rewriting it. This matters for the reader's own toolbox: the rgtlab R packages (`zztable1`, `zzlongplot`, and the rest of the suite at https://github.com/rgt47) remain callable from Python through `rpy2`, so adopting Python need not mean abandoning an R routine that already works.

```
import rpy2.robjects as ro
ro.r('summary(c(1, 2, 3))')
```

```
reticulate::py_run_string('x = [1, 2, 3]')
```

We answer the natural question of direction: keep the analysis in whichever language holds most of its logic, and cross the boundary only for the specific component the other language serves better.

5.2.6. A note on migration strategy

We close the lecture with a strategy rather than a command. A migration need not be all-or-nothing. The safe path is to move one component at a time, checking each Python result against the R result it replaces, until the R code is either gone or deliberately retained behind an interoperability call.

Draft, November 13, 2026. Content may change.

5.3. Worked example: migrating a small R analysis to Python

We take a compact R analysis the reader could have written on any project, reading a CSV, filtering, summarizing by group, and fitting a model, and migrate it to Python step by step, verifying as we go.

The R original:

```r
enroll <- readr::read_csv('enroll.csv')
summary_tbl <- enroll |>
  dplyr::filter(age > 30) |>
  dplyr::group_by(arm) |>
  dplyr::summarize(mean_wt = mean(weight))
fit <- lm(weight ~ age + arm, data = enroll)
```

The Python migration, component by component, keeps each step beside the R line it replaces:

```python
import pandas as pd
import statsmodels.formula.api as smf

enroll = pd.read_csv('enroll.csv')

summary_tbl = (
    enroll
    .loc[enroll['age'] > 30]
    .groupby('arm')
    .agg(mean_wt=('weight', 'mean'))
    .reset_index()
)

fit = smf.ols('weight ~ age + arm', data=enroll).fit()
```

We verify the migration by comparing outputs, not by inspection alone. The grouped means should match the R `summary_tbl` row for row, and the fitted coefficients should match `coef(fit)` from R to several decimal

places. Where they disagree, the usual culprits are a different default (for example, a standard deviation divisor) or an index carried along silently, both of which we met earlier in the week. When the numbers agree, the migration is done, and the reader now has the same analysis in a second language.

5.4. Homework

Attempt each problem before checking the solution.

1. **Project skeleton.** Create a project directory with the layout shown in the lecture. State which parts correspond to an R package's `R/`, `tests/`, and `DESCRIPTION`.

2. **Pin an environment.** In a fresh environment, install two packages and write a lockfile with `uv lock` (or a `requirements.txt`). Explain what committing that file buys you, in the language of `renv.lock`.

3. **Recreate an environment.** From the lockfile of problem 2, recreate the environment in a new directory and confirm the same versions install. Name the `renv` step that corresponds.

4. **Write a test.** Write a small function and a `pytest` test for it, then run `pytest` and read the report. Show the `tinytest` assertion that would test the same behavior.

5. **A failing test.** Deliberately break the function so the test fails, and read what `pytest` reports. Compare the failure output with what `tinytest` reports.

6. **Quarto with Python.** Write a one-cell Quarto document with a `{python}` cell that prints a small `DataFrame`, and render it. Note what changed relative to an `{r}` cell.

7. **Round trip.** Using `reticulate` from R or `rpy2` from Python, compute a value in the other language and bring it back. Describe when you would choose each direction.

Draft, November 13, 2026. Content may change.

8. **A full migration.** Take a short R script of your own (read, transform, summarize) and migrate it to Python, verifying each intermediate result against the R output. Report the one step that required the most care.

5.5. Solutions

Solutions

Problem 1. Placeholder: `src/analysis/` maps to `R/`, `tests/` maps to `tests/`, and `pyproject.toml` maps to `DESCRIPTION`.
Problem 2. Placeholder: `uv lock` writes `uv.lock`; committing it fixes exact versions as `renv.lock` does.
Problem 3. Placeholder: `uv sync` from the lockfile, matching `renv::restore()`.
Problem 4. Placeholder: a `test_*` function using `assert`; the `tinytest` form is `expect_equal(...)`.
Problem 5. Placeholder: `pytest` prints the failing assertion with the observed and expected values.
Problem 6. Placeholder: only the cell language tag changes from `{r}` to `{python}`.
Problem 7. Placeholder: use `reticulate` when the analysis lives in R and borrows Python; use `rpy2` when it lives in Python and borrows R.
Problem 8. Placeholder: the step needing most care is typically a silent default difference or a carried index.

5.6. What's next

This is the final day of the workshop. The reader can now read and write everyday scientific Python and move an analysis between R and Python in either direction. To go further, the companion volume *Statistical Computing in the Age of AI* (https://scai.rgtlab.org) develops the statistical methods these tools implement, and *Applied Generative AI for Health Sciences*

Draft, November 13, 2026. Content may change.

Research (https://applied-genai.rgtlab.org) takes up the deep-learning and large-language-model libraries that this workshop deliberately set aside.

Draft, November 13, 2026. Content may change.

References

Credits

The chapter list was constructed by surveying publicly available materials on scientific Python for readers coming from R, including Wes McKinney's *Python for Data Analysis*, Jake VanderPlas's *Python Data Science Handbook*, the pandas and NumPy documentation, and the `reticulate` package documentation, and adopting the topics common to the majority of them.

Cover artwork generated procedurally by the R script at `images/build-cover.R`. Watercolor palette: deep slate through steel and periwinkle to pale mist, anchored on a brand slate `#4b6584` chosen to sit between the Python and R visual identities. Typography: Avenir family (Apple system font); body text in Source Serif 4.

Colophon

This book was produced with Quarto, typeset in Source Serif 4, with code blocks set in JetBrains Mono. The examples target Python 3.11 or later, with NumPy 1.26 or later and pandas 2.0 or later.

The book is hosted at https://python-for-r.rgtlab.org on Netlify, with continuous deployment via GitHub Actions from the `rgt47/python-for-r` repository on every push to `main`. The deployment recipe is in `HOSTING.md`.

The cover is generated procedurally by `images/build-cover.R` from a watercolor gradient anchored on `#4b6584` (the volume's brand slate) and overlaid with Avenir typography. Re-running the script regenerates the cover; the procedural approach makes the cover reproducible and editable in version control rather than dependent on a one-shot image-generation step. That a book about crossing between two languages carries a cover scripted in one of them is a small deliberate joke.

Last rendered: see the build timestamp on the homepage.