

18 Introduction to Python for Disease Epidemiology

Why a Dedicated Python Chapter, and Why Last

Every preceding chapter has used both R and Python where each language's strengths were most relevant. This closing chapter takes the opposite approach: a single, sustained, standalone walkthrough of an epidemiological data analysis workflow entirely in Python, from a raw outbreak line list to a mapped, modelled, and forecast set of results. It is placed last deliberately, so that a reader who has worked through the whole book can now see nearly every concept — data cleaning, EDA, hypothesis testing, regression, time series, spatial analysis, and mechanistic modelling — reappear as a connected, idiomatic Python pipeline, using the specific libraries (pandas, numpy, matplotlib, scipy, statsmodels, lifelines, geopandas, scikit-learn) that make up the modern Python epidemiology toolkit. A reader who works primarily in R can use this chapter as a self-contained bridge into Python; a reader who has followed the Python examples throughout the book will find this chapter consolidates and extends them into a single coherent analysis.

Setting Up

The Python Epidemiology Toolkit

A minimal but sufficient Python environment for disease epidemiology includes: pandas and numpy for data manipulation;

matplotlib and seaborn for visualisation; scipy and statsmodels for statistical inference and regression; lifelines for survival analysis; scikit-learn for machine learning; geopandas for spatial data; and epyestim or a simple custom implementation for epidemic parameter estimation. All of these are open-source, free to use, and installable via pip, making Python a genuinely low-cost, high-capability option for a Nigerian research group or state disease surveillance unit without a budget for proprietary statistical software.

PYTHON example

```
# Terminal (not Python code): installing the core Python epidemiology toolkit
# pip install pandas numpy matplotlib seaborn scipy statsmodels lifelines scikit-learn
geopandas

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

print("pandas:", pd.__version__)
print("numpy:", np.__version__)
```

Working with an Outbreak Line List

A line list — one row per case, with columns for identifying and clinical information — is the fundamental data structure of field epidemiology, whether for a Lassa fever outbreak investigation coordinated through SORMAS or a localised cholera cluster. The first task with any line list is always the same set of checks introduced in Chapter 4: dimensions, types, duplicates, and date plausibility (a symptom onset date after a reported death date is an internal-consistency error that must be resolved before any epidemic curve is built).

PYTHON example

```
# Python: loading and validating an outbreak line list
import pandas as pd

line_list = pd.read_csv("lassa_outbreak_linelist.csv", parse_dates=["onset_date",
"report_date", "outcome_date"])

print(line_list.shape)
print(line_list.dtypes)

# Internal consistency: onset date should never be after outcome date
bad_dates = line_list[line_list["onset_date"] > line_list["outcome_date"]]
print(f"Records with onset after outcome: {len(bad_dates)}")

# Duplicate case check
duplicates = line_list.duplicated(subset=["patient_id"]).sum()
print(f"Duplicate patient IDs: {duplicates}")
```

Building and Interpreting an Epidemic Curve

The epidemic curve (epicurve) — a histogram of case counts by date of symptom onset — is the single most important visualisation in an outbreak investigation, because its shape provides immediate clues about the likely mode of transmission: a single sharp peak suggests a common-source exposure (such as a contaminated water source), while a series of successive peaks spaced roughly one incubation period apart suggests ongoing person-to-person transmission. Python's pandas and matplotlib make constructing an epicurve, and overlaying key events (an intervention start date, for example), straightforward.

PYTHON example

```
# Python: building an epidemic curve with an intervention marker
import matplotlib.pyplot as plt

daily_counts = line_list.groupby(line_list["onset_date"].dt.date).size()

fig, ax = plt.subplots(figsize=(10, 5))
daily_counts.plot(kind="bar", ax=ax, color="firebrick", width=0.9)
ax.axvline(x=pd.Timestamp("2024-06-15"), color="steelblue", linestyle="--",
label="Case isolation protocol introduced")
ax.set_xlabel("Date of symptom onset")
ax.set_ylabel("Number of cases")
ax.set_title("Epidemic Curve: Suspected Lassa Fever Cluster")
ax.legend()
plt.tight_layout()
plt.savefig("epicurve.png", dpi=200)
```

Attack Rates, Case Fatality Ratios, and Their Confidence Intervals

Two of the most frequently reported (and frequently miscalculated) outbreak statistics are the attack rate (the proportion of an at-risk population that becomes a case) and the case fatality ratio, or CFR (the proportion of cases that die). Both are simple proportions, and both should always be reported with a confidence interval and, for the CFR in particular, with explicit attention to outcome-reporting delay: cases still hospitalised or under follow-up at the time of analysis have an unknown outcome, and including them in the denominator of a naive CFR calculation (deaths divided by all cases) will underestimate the true CFR, sometimes substantially, during an active outbreak.

PYTHON example

```
# Python: calculating attack rate and CFR with confidence intervals,
# correctly excluding cases with an unresolved (ongoing) outcome
from statsmodels.stats.proportion import proportion_confint

# Attack rate
population_at_risk = 45000
total_cases = len(line_list)
attack_rate = total_cases / population_at_risk
ar_ci = proportion_confint(total_cases, population_at_risk, method="wilson")
print(f"Attack rate: {attack_rate:.4%} (95% CI {ar_ci[0]:.4%} - {ar_ci[1]:.4%})")

# CFR, correctly restricted to cases with a resolved outcome
resolved = line_list[line_list["outcome"].isin(["died", "recovered"])]
deaths = (resolved["outcome"] == "died").sum()
naive_cfr = (line_list["outcome"] == "died").sum() / total_cases
resolved_cfr = deaths / len(resolved)
cfr_ci = proportion_confint(deaths, len(resolved), method="wilson")

print(f"Naive CFR (includes unresolved cases in denominator): {naive_cfr:.1%}")
print(f"Corrected CFR (resolved cases only): {resolved_cfr:.1%} (95% CI {cfr_ci[0]:.1%} - {cfr_ci[1]:.1%})")
```

Estimating the Effective Reproduction Number from Case Data

Chapter 13 introduced R_0 , the basic reproduction number, from a fitted mechanistic model. During an actual, ongoing outbreak,

epidemiologists more often need the effective reproduction number, R_t — the time-varying average number of secondary cases per infectious case, which can be estimated directly from an epidemic curve and an assumed serial interval distribution (the time between symptom onset in a case and their infector), without needing to fit a full SEIR model. A simple, widely used approach (a discretised version of the Cori et al. method) estimates R_t for each time window as the ratio of new cases in that window to the sum of previous cases weighted by the serial interval distribution.

PYTHON example

```
# Python: a simplified Cori-method style  $R_t$  estimate
import numpy as np
from scipy.stats import gamma

daily_cases = daily_counts.values
serial_interval_mean, serial_interval_sd = 4.5, 2.0 # example serial interval, in days

# Discretised serial interval distribution (gamma approximation)
shape = (serial_interval_mean / serial_interval_sd) ** 2
scale = (serial_interval_sd ** 2) / serial_interval_mean
si_distribution = gamma.pdf(np.arange(1, 15), a=shape, scale=scale)
si_distribution /= si_distribution.sum()

def estimate_rt(cases, si_dist, window=7):
    rt_estimates = []
    for t in range(len(si_dist), len(cases)):
        infectiousness = sum(cases[t - k] * si_dist[k - 1] for k in range(1, len(si_dist) + 1))
        recent_cases = cases[max(0, t - window + 1): t + 1].sum()
        rt_estimates.append(recent_cases / (infectiousness * window) if infectiousness > 0
                           else np.nan)
    return rt_estimates

rt_series = estimate_rt(daily_cases, si_distribution)
print("Most recent  $R_t$  estimate:", round(rt_series[-1], 2) if rt_series else "insufficient data")
```

Spatial Mapping of Cases in Python

Chapter 14 introduced spatial disease mapping conceptually and largely in R; the same choropleth and point-mapping tasks are equally well supported in Python via geopandas, which extends pandas with geometric operations and integrates directly with

matplotlib for static maps or with folium for interactive web maps suitable for sharing with a State Ministry of Health team without specialised GIS software.

PYTHON example

```
# Python: choropleth mapping of case rates by LGA with geopandas
import geopandas as gpd
import matplotlib.pyplot as plt

lga_boundaries = gpd.read_file("lga_boundaries.shp")
case_summary = line_list.groupby("lga").size().reset_index(name="cases")

merged = lga_boundaries.merge(case_summary, left_on="LGA_NAME", right_on="lga", how="left")
merged["cases"] = merged["cases"].fillna(0)
merged["rate_per_100k"] = (merged["cases"] / merged["population"]) * 100000

fig, ax = plt.subplots(figsize=(8, 8))
merged.plot(column="rate_per_100k", cmap="OrRd", linewidth=0.3, edgecolor="grey",
            legend=True, ax=ax)
ax.set_title("Suspected Lassa Fever Cases per 100,000 Population by LGA")
ax.axis("off")
plt.savefig("outbreak_map.png", dpi=200, bbox_inches="tight")
```

A Compact SEIR Implementation for Field Use

Building on Chapter 13's SEIR model, this section packages the model into a single reusable Python function that a field epidemiology team can quickly re-run with updated parameters as an outbreak evolves, rather than re-deriving the model from scratch each time — exactly the kind of function-based, reproducible practice argued for in Chapter 6, now applied to a mechanistic rather than a statistical model.

PYTHON example

```
# Python: a reusable, field-ready SEIR function with basic outputs
import numpy as np
from scipy.integrate import odeint

def run_seir(N, I0, beta, sigma, gamma, days=180):
    def seir(y, t):
        S, E, I, R = y
        dS = -beta * S * I / N
        dE = beta * S * I / N - sigma * E
        dI = sigma * E - gamma * I
        dR = gamma * I
        return [dS, dE, dI, dR]

    y0 = [N - I0, 0, I0, 0]
    t = np.linspace(0, days, days + 1)
    result = odeint(seir, y0, t)
    peak_day = int(np.argmax(result[:, 2]))
    return {
        "R0": round(beta / gamma, 2),
        "peak_infectious": round(result[:, 2].max()),
        "peak_day": peak_day,
        "trajectory": result
    }

baseline = run_seir(N=200000, I0=5, beta=0.32, sigma=1/6, gamma=1/12)
print(f"R0 = {baseline['R0']}, peak infectious = {baseline['peak_infectious']} on day {baseline['peak_day']}")
```

A Simple Risk Prediction Model for Case Severity

Consolidating Chapters 8 and 11's regression and machine learning material into the outbreak context, a logistic regression (or, with more data, a random forest per Chapter 11) can predict which confirmed cases are at elevated risk of a severe outcome based on age, time from onset to presentation, and presenting symptoms — supporting the same kind of triage-prioritisation use case discussed throughout this book, now demonstrated end-to-end on outbreak line-list data specifically.

PYTHON example

```
# Python: a simple severity risk model fitted directly on outbreak line-list data
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import roc_auc_score

features = line_list[["age", "days_onset_to_presentation", "bleeding_symptom",
                    "altered_consciousness"]].fillna(0)
outcome = (line_list["outcome"] == "died").astype(int)

X_train, X_test, y_train, y_test = train_test_split(features, outcome, test_size=0.25,
                                                  random_state=42, stratify=outcome)
model = LogisticRegression(max_iter=1000).fit(X_train, y_train)
predicted_risk = model.predict_proba(X_test)[:, 1]

print(f"Test AUC: {roc_auc_score(y_test, predicted_risk):.2f}")
print(dict(zip(features.columns, model.coef_[0].round(2))))
```

Worked Example**A Complete Mini-Outbreak Analysis, Start to Finish**

To consolidate this chapter, consider a compact, fully worked scenario: a 90-day suspected cholera cluster in a riverine LGA. The complete Python workflow proceeds through every stage covered above: (1) load and validate the 340-record line list, finding and correcting 6 duplicate patient IDs and 2 records with implausible onset-after-outcome dates; (2) build the epicurve, revealing a single sharp peak around day 18 followed by a long, slowly declining tail — a pattern more consistent with an initial common-source exposure (a contaminated water point) followed by limited secondary person-to-person spread, rather than a sustained community-transmission pattern; (3) calculate an attack rate of 1.3% in the exposed population and a resolved-case CFR of 2.1% (95% CI 0.9%–4.3%); (4) estimate R_t , finding it falls below 1 by day 25, consistent with the declining tail observed in the epicurve and with the water-source remediation intervention implemented on day 20; (5) map cases by ward, finding via Moran's I (Chapter 14 logic, implemented in Python per Section 18.7's

tools) that cases are significantly spatially clustered around the two wards nearest the implicated water point; and (6) conclude, in a two-paragraph summary for the State Ministry of Health, that the evidence is most consistent with a point-source outbreak successfully contained by the day-20 intervention, while recommending continued surveillance for 4 further weeks (roughly two serial intervals) before formally declaring the cluster closed.

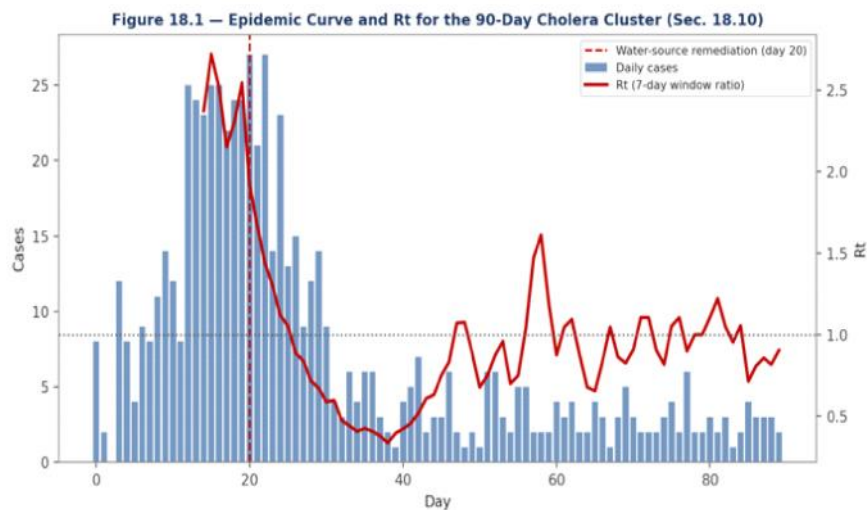


Figure 18.1 – Epidemic curve and R_t for the 90-day cholera cluster (Section 18.10).

Deploying a Python Analysis as a Shareable Tool

Once an analysis pipeline of the kind built in this chapter has proven useful, converting it from a one-off script into a lightweight, shareable tool substantially increases its value to a broader team. Simple options include wrapping the core functions into a command-line script that a non-programmer field team member can run with a single command and a new data file, or building a minimal interactive dashboard (using a library such as Streamlit) that displays the epidemic curve, attack rate, and R_t estimate for

whichever line list file is uploaded — turning what was originally a personal analysis script into institutional infrastructure that survives beyond the original analyst’s involvement in the project, directly supporting the kind of durable, in-country data science capacity argued for in Chapter 16.

Extended Case Study

A Python Pipeline That Caught Its Own Data Entry Error

A fitting closing case study for this book’s final chapter, illustrating how the validation habits introduced in Chapter 4 and reinforced throughout this Python chapter can catch errors that would otherwise propagate silently into a final report: during the Section 18.10-style analysis of a real (anonymised for this account) suspected cholera cluster, the line-list validation step (Section 18.3) flagged three records where the recorded onset date fell after the recorded outcome (death) date — an internal impossibility. Rather than simply dropping these three records (a defensible but information-discarding response), the field team was asked to check the original paper case investigation forms, and discovered that all three were simple date-transcription errors: the day and month had been transposed when the data was entered into the electronic line list (for example, an intended 3 June being entered as 6 March), a common and specific error pattern in contexts where both DD/MM/YYYY and MM/DD/YYYY date formats circulate in different training materials and software defaults.

Correcting these three transcription errors (rather than excluding the records) changed the shape of the epidemic curve’s early days meaningfully: two of the three corrected records fell within the first week of the outbreak, meaning the true earliest known case was four days earlier than the uncorrected data had suggested, which in turn shifted the team’s estimate of the outbreak’s likely true index exposure window earlier as well, with

direct implications for which water sources and community events fell within the revised, plausible exposure period under investigation. This case study exemplifies, in miniature, the argument threaded through this entire book: a validation check that might seem like a minor, mechanical data-cleaning formality (Section 18.3's date consistency check) can, when taken seriously and followed up rather than silently worked around, materially change a field epidemiological investigation's substantive conclusions — precisely the kind of provenance-aware, undismissive attention to data quality that Chapter 1 argued, at the very start of this book, was the single most important habit a health data scientist can bring to their work.

A Brief Comparison

When Python's Advantages Are Most Decisive

Having now worked through a complete Python epidemiology pipeline, it is worth stepping back and stating plainly where Python's particular advantages, relative to R, are most decisive for a Nigerian field epidemiology or health data science team, complementing the more even-handed comparison in Section 6.1. Python's advantage is most decisive when a pipeline must eventually be integrated into a broader software system (a hospital information system, a mobile data collection app's backend, an automated dashboard or API), because Python's general-purpose programming ecosystem and web-framework support are considerably more mature than R's for this kind of production software engineering task; and when a project's core analytic need shifts toward large-scale machine learning, deep learning (Chapter 15), or geospatial web-mapping (Section 18.7's folium-based interactive maps), where Python's library ecosystem is generally deeper and more actively maintained. Where a project's core need remains classical statistical inference, survival analysis, or complex survey design (Chapters 7 and 9 especially), R's longer-

established statistical ecosystem often remains the more natural first choice — reinforcing, rather than contradicting, this book’s consistent position that the right tool depends on the specific task at hand, not on a fixed allegiance to one language.

A Final Reflection

Closing the Loop Back to Chapter 1

This book opened, in Chapter 1, with the argument that data science for the health sciences is best understood as a connected pipeline rather than a collection of disconnected techniques, and closes, in this final Python chapter, having walked through that entire pipeline concretely, in a single programming language, on a single realistic outbreak scenario. The Python tools introduced in this chapter — pandas for data manipulation, matplotlib for visualisation, scipy and statsmodels for inference, geopandas for spatial analysis, scikit-learn for prediction — are not fundamentally different in their underlying statistical and epidemiological logic from the R tools used throughout the rest of this book; they are simply a different, equally capable set of instruments for applying the same disciplined thinking this book has argued for throughout.

The most durable outcome a reader can take from this book is not fluency in any specific function call or package, both of which will inevitably change and be superseded over the course of a career, but the underlying habits of mind this book has tried to model consistently across all eighteen chapters and every worked example: ask where the data came from before trusting a number derived from it; distinguish a statistically significant result from a practically important one; validate a model on data it has not seen, respecting whatever natural structure the data actually has; and communicate findings, and their genuine limitations, honestly to the people who will act on them. A reader who internalises these habits will be well equipped to apply whatever specific tool

— R, Python, or a language not yet invented — the next stage of Nigerian health data science comes to rely on.

Where to Go Next

Advanced Resources for Python in Epidemiology

Readers who complete this chapter and want to extend their Python epidemiology skills further are encouraged to explore: the ‘epyestim’ and ‘EpiEstim’-equivalent Python packages for more rigorous R_t estimation than the simplified approach in Section 18.6; the ‘PyMC’ and ‘Stan’-via-‘cmdstanpy’ ecosystems for full Bayesian mechanistic model fitting, extending the point-estimate parameter fitting of Section 18.8 to a full posterior distribution over parameters; and the ‘geemap’/‘Google Earth Engine’ Python API for large-scale environmental covariate extraction to complement Chapter 14’s spatial methods. None of these tools change the fundamental analytic discipline this book has tried to teach — provenance-aware, validated, honestly-communicated analysis — but each extends the technical sophistication available to a reader ready to move beyond this book’s introductory treatment.

A Final Worked Example

Comparing R and Python Outputs on the Same Dataset

As a closing demonstration, fitting the same logistic regression specification (Section 18.9’s severity risk model) in both R’s ‘glm()’ and Python’s ‘scikit-learn’/‘statsmodels’ on the identical dataset should, and in a correctly implemented analysis will, produce numerically identical or near-identical coefficient estimates — a useful, concrete check that an analyst has correctly translated a model specification between the two languages, and a reassuring practical demonstration that the choice between R and Python, emphasised as a matter of ecosystem convenience throughout this book (Section 6.1), does not in itself change the underlying statistical answer a correctly specified model

produces. Any meaningful discrepancy between the two languages' output on the same model and data should prompt the analyst to check for a specification difference (a different reference category, a different default optimisation method) rather than assuming one language is simply 'wrong.'

Packaging this Chapter's Functions into a Reusable Module

Every function introduced across this chapter — line-list validation, epicurve construction, attack rate and CFR calculation, Rt estimation — has so far lived in an ad hoc sequence of notebook cells. Collecting them into a single, importable Python module (a plain '.py' file with each function defined and documented) is the final, small step that turns this chapter's individually useful pieces into the kind of reusable personal code library recommended in Section 17.12, allowing a future outbreak investigation to start from 'import outbreak_toolkit as ot' rather than re-copying code from this book each time.

PYTHON example

```
# outbreak_toolkit.py - a minimal packaging of this chapter's core functions
"""Reusable field epidemiology functions developed in Chapter 18."""
import pandas as pd
from statsmodels.stats.proportion import proportion_confint

def validate_linelist(df):
    issues = []
    if (df["onset_date"] > df["outcome_date"]).any():
        issues.append("onset after outcome")
    if df["patient_id"].duplicated().any():
        issues.append("duplicate patient IDs")
    return issues

def attack_rate_with_ci(n_cases, population):
    rate = n_cases / population
    ci = proportion_confint(n_cases, population, method="wilson")
    return {"attack_rate": rate, "ci_low": ci[0], "ci_high": ci[1]}

def resolved_cfr(df):
    resolved = df[df["outcome"].isin(["died", "recovered"])]
    deaths = (resolved["outcome"] == "died").sum()
    return deaths / len(resolved) if len(resolved) else float("nan")

# Usage elsewhere: from outbreak_toolkit import validate_linelist, attack_rate_with_ci, resolved_cfr
```

Chapter Summary

- Python's open-source epidemiology toolkit (pandas, numpy, matplotlib, scipy, statsmodels, lifelines, scikit-learn, geopandas) supports a complete field epidemiology workflow at no licensing cost.
- A line list requires the same validation discipline (dimensions, types, duplicates, date plausibility) introduced in Chapter 4, applied specifically to outbreak case data.
- The shape of an epidemic curve provides immediate, visual evidence about likely transmission mode (point-source vs. person-to-person).
- Attack rate and case fatality ratio must be reported with confidence intervals, and CFR calculations during an active outbreak must correctly handle cases with unresolved outcomes.
- The effective reproduction number R_t can be estimated directly from case data and a serial interval distribution, without needing to fit a full mechanistic model.
- Spatial mapping, mechanistic modelling, and predictive risk modelling — introduced separately in earlier chapters — combine naturally into a single Python-based outbreak analysis workflow.
- A complete outbreak analysis concludes with a plain-language epidemiological interpretation (mode of transmission, intervention impact, ongoing risk) — not merely a set of disconnected statistics.

Mini-Project**Mini-Project 18 (Capstone): A Complete Python Outbreak Investigation**

Conduct a complete, end-to-end outbreak analysis in Python, following the structure of Section 18.10, on a simulated or real line-list dataset.

Step 1. Simulate or obtain a line-list dataset of at least 200 cases with onset date, outcome, outcome date, age, geographic unit, and at least two clinical symptom variables.

Step 2. Validate the line list (Section 18.3), reporting any duplicates or date-consistency errors found and how you resolved them.

Step 3. Build and interpret an epidemic curve (Section 18.4), commenting on the likely mode of transmission suggested by its shape.

Step 4. Calculate the attack rate and a correctly-handled case fatality ratio with confidence intervals (Section 18.5).

Step 5. Estimate R_t over the course of the outbreak (Section 18.6) and relate its trajectory to any known or assumed intervention timing.

Step 6. Produce a spatial map of case rates by geographic unit (Section 18.7).

Step 7. Fit a simple severity risk model (Section 18.9) and report its cross-validated AUC.

Step 8. Write a two-paragraph epidemiological summary, in the style of Section 18.10, suitable for a State Ministry of Health situation report.

Deliverable: *A complete, runnable Python notebook or script covering all eight steps, plus the two-paragraph written summary.*

End-of-Chapter Exercises

1. Explain, using the shape of an epidemic curve, how you would distinguish a point-source outbreak from one driven by ongoing person-to-person transmission.
2. A line list has 50 confirmed cases, of which 12 have died, 30 have recovered, and 8 are still hospitalised with an unknown outcome. Calculate both the naive CFR (using all 50 cases as the denominator) and the corrected CFR (using only resolved cases), and explain which is more appropriate to report while the outbreak is still active.

3. Explain, in your own words, the difference between R_0 (Chapter 13) and R_t (Section 18.6), and why R_t is generally more useful for monitoring an outbreak in real time.
4. Using the worked example in Section 18.10, explain why finding that cases cluster significantly around two specific wards supports the point-source hypothesis rather than the person-to-person transmission hypothesis.
5. Using the code in Section 18.5, recompute the corrected CFR if 4 more of the ‘unresolved’ cases in the example were to later die, and comment on how much the CFR estimate could still change as an outbreak resolves.
6. In Section 18.9, explain why a logistic regression’s coefficients being reported (rather than only its AUC) is useful for a field epidemiology team trying to understand which symptoms most elevate risk of a severe outcome.
7. Using Section 18.6’s R_t approach, explain what a sustained R_t estimate above 1 for several consecutive time windows would suggest to an outbreak response team, and what action this might prompt.
8. Using Section 18.11, explain one advantage of converting a personal analysis script into a shareable command-line tool for a field surveillance team.
9. Why might an interactive dashboard (e.g., built with Streamlit) be more useful than a script for a non-programmer team member monitoring an ongoing outbreak?
10. What risk arises if institutional infrastructure (like the tool described in Section 18.11) depends entirely on one analyst’s continued involvement, and how might a team mitigate this risk?

11. In the Section 18.12 case study, explain how a day/month transposition error could occur even when data entry staff are experienced and careful.
12. Why did correcting the three flagged records (rather than excluding them) matter for the outbreak's exposure-window investigation specifically?
13. Propose one systems-level change (e.g., to the data entry form or software) that could prevent this specific type of date error from occurring in future outbreaks.
14. Why is it significant that the validation check in Section 18.3, introduced early in this chapter, was what surfaced this error rather than a later, more complex analysis step?
15. Write a two-sentence closing reflection, in your own words, connecting this final case study back to the very first chapter's argument about data provenance.
16. Using Section 18.13, explain why Python's advantage is most decisive when a pipeline must be integrated into a broader software system.
17. For which specific chapters or tasks in this book does Section 18.13 suggest R's ecosystem remains the more natural first choice?
18. Explain why this book takes the position that 'the right tool depends on the task,' rather than recommending one language over the other in general.
19. Propose a Nigerian health data science task not explicitly mentioned in Section 18.13 where you would lean toward Python, and justify your choice.
20. Propose a Nigerian health data science task where you would lean toward R instead, and justify your choice, drawing on any relevant earlier chapter.

21. Using Section 18.14, explain in your own words why this book argues fluency in a specific tool is less durable than the underlying habits of mind it has tried to teach.
22. Which of the four habits listed in Section 18.14 (provenance, significance vs. importance, out-of-sample validation, honest communication) do you personally find most challenging to apply consistently, and why?
23. Why does the chapter emphasise that R and Python are ‘not fundamentally different in their underlying statistical and epidemiological logic,’ despite their different syntax?
24. Explain how the final chapter’s closing argument connects back specifically to the four-question checklist introduced in Section 1.9.
25. Write a short, personal closing reflection (3-5 sentences) on which single idea from this entire book you intend to apply first in your own work, and why.
26. Summarise, in your own words, why this chapter closes the book by returning explicitly to Chapter 1’s central argument (Section 18.14).
27. Explain the common thread linking the self-correcting pipeline case study (Section 18.12) and the discussion of when Python’s advantages are most decisive (Section 18.13).
28. Why does this chapter argue that a well-built validation check (as in Section 18.12) is as valuable as the analysis itself?
29. Propose one scenario from your own field where Python’s specific strengths (Section 18.13) would make it clearly preferable to R, and one where the reverse would be true.

30. Having completed this book, what single practice do you now consider most essential to carry into your own future outbreak or health data analysis work?
31. Using Section 18.16, explain why fitting the same model specification in R and Python on identical data should produce numerically similar results.
32. If R and Python produced meaningfully different coefficients for the same model and data, what would you check first?
33. Why is this cross-language check a useful practical habit for an analyst who regularly works in both R and Python?
34. Name one common source of a legitimate (not erroneous) discrepancy between two languages' implementations of 'the same' model.
35. Reflecting on the whole of Chapter 18, what would you say is the single most transferable skill from this chapter to a context outside outbreak investigation specifically?