# Python for Clinical Study Reports and Submission

Yilong Zhang  Nan Xiao

# Table of contents

# Welcome

Welcome to Python for Clinical Study Reports and Submission. Clinical study reports (CSR) are crucial components in clinical trial development. A CSR is an "integrated" full scientific report of an individual clinical trial.

The ICH E3: Structure and Content of Clinical Study Reports offers comprehensive instructions to sponsors on the creation of a CSR. This book is a clear and straightforward guide on using Python to streamline the process of preparing CSRs. Additionally, it provides detailed guidance on the submission process to regulatory agencies. Whether you are a beginner or an experienced developer, this book is an indispensable asset in your clinical reporting toolkit.

**This is a work-in-progress draft.**

# Preface

## In this book

This book is designed for people who are interested in using Python for clinical development. Each part of the book makes certain assumptions about the readers' background:

- Part 1, titled "Environment and toolchain" and "Reporting packages", provides general information on setting up Python development environments for clinical reporting.
- Part 2, titled "Delivering TLFs in CSR", provides general information and examples on creating tables, listings, and figures using Python. It assumes that readers are individual contributors to a clinical project with prior experience in Python. Familiarity with data manipulation using Polars is expected. Recommended references for this part include Python Polars: The Definitive Guide, theand the rtflite documentation.
- Part 3, titled "Clinical trial project", provides general information and examples on managing a clinical trial A&R project using Python. It assumes that readers are project leads who have experience in Python package development.
- Part 4, titled "eCTD submission package", provides general information on preparing submission packages related to the CSR in the electronic Common Technical Document (eCTD) format using Python. It assumes that readers are project leads of clinical projects who possess experience in Python package development and regulatory submission processes.

## Philosophy

We share the same philosophy described in the introduction of the R Packages book (Wickham and Bryan 2023), which we quote below:

- "Anything that can be automated, should be automated."
- "Do as little as possible by hand. Do as much as possible with functions."

## Authors and contributors

This document is a collaborative effort maintained by a community. As you read through it, you also have the opportunity to contribute and enhance its quality. Your input and involvement play a vital role in shaping the excellence of this document.

- Authors: made significant contributions to at least one chapter, constituting the majority of the content.

  Yilong Zhang, Nan Xiao,

# Part I

# Environment and toolchain

# 1 Python developer setup

Objective

Set up a productive Python development environment for clinical study reporting. Learn about IDE options, essential extensions, and workflow tools.

## 1.1 Development environments

For this book, you have several options for your development environment. Choose the one that best fits your current setup and constraints.

### 1.1.1 GitHub Codespaces

GitHub Codespaces provides a cloud-based development environment with everything pre-configured. This is the easiest option if you don't have a local Python setup.

We will provide a dev container configuration that includes:

- Python with uv pre-installed.
- All necessary VS Code extensions.
- Consistent environment across all readers, useable in the web browser.

To use Codespaces, simply click the "Code" button in the repository and select "Create codespace on main".

> **Note**
>
> Codespaces currently offers 120 hours of free compute time per month for personal accounts. This is more than sufficient for this book.

### 1.1.2 Positron

Positron is Posit's next-generation data science IDE, built on Code OSS (the open source core of VS Code), with specific improvements for R and Python development.

Key features for Python work:

- Native notebook support.
- Interactive variable explorer.
- Integrated plot viewer.
- Built-in data viewer for DataFrames.

Download Positron from https://positron.posit.co/.

Positron uses Open VSX instead of the Microsoft VS Code marketplace. Most essential Python extensions are available, but the selection is more limited.

### 1.1.3 VS Code

Visual Studio Code remains the most popular choice for Python development. It offers a rich ecosystem of extensions and tools.

Essential extensions for this book:

- Python: Core Python language support.
- Pylance: Fast, feature-rich Python language server.
- Ruff: Lightning-fast linting and formatting.
- Even Better TOML: Syntax highlighting for TOML files (`pyproject.toml`).
- Quarto: Authoring support for Quarto documents.

## 1.2 VS Code settings

### 1.2.1 Unicode highlighting

Python allows Unicode characters in strings and identifiers. AI coding tools might also generate code with non-ASCII characters. For regulatory work, you should highlight non-ASCII characters to find these hidden issues early and avoid problems in submissions.

**Via Settings UI:**

1. Open Command Palette (Cmd/Ctrl + Shift + P)
2. Search for "Preferences: Open Settings (UI)"
3. Search for "Unicode Highlight"
4. Enable "Non Basic ASCII" for both trusted and untrusted workspaces

**Via Settings JSON:**

Open Command Palette with Cmd/Ctrl + Shift + P, select "Preferences: Open User Settings (JSON)", then add:

```
"editor.unicodeHighlight.nonBasicASCII": true
```

This highlights characters like curly quotes, em dashes, and other non-ASCII characters that could cause issues in eCTD submission packages.

## 1.3 Terminal setup

For local development, you will interact with uv and Quarto through the terminal.

### 1.3.1 Shell

Any modern shell works well:

- macOS/Linux: zsh (default on macOS), bash
- Windows: PowerShell, Windows Terminal

### 1.3.2 Terminal emulator

If you are on macOS and want a faster terminal experience, consider Ghostty. It is written in Zig for exceptional performance.

## 1.4 AI coding assistants

Modern agentic AI coding tools can accelerate statistical and clinical coding tasks, especially for popular programming languages like Python. We encourage you to use them, for example:

- Codex (command-line interface, VS Code extension)
- Claude Code (command-line interface)
- Cursor (AI-first editor)
- GitHub Copilot (VS Code extension)

### 1.4.1 Effective use of AI tools

To use AI assistants effectively for programming, you need:

**Product manager mindset**: Know exactly what you want to build. In clinical reporting, this means understanding the table shell, statistical method, and regulatory requirements.

**Software architect mindset**: Evaluate model outputs critically. Can you spot issues with data transformations? Do the statistical computations match the statistical analysis plan? Is the output format submission-ready?

> ⚠ Warning
>
> AI tools are assistants, not replacements for domain expertise. Always verify outputs against statistical analysis plans and regulatory guidance.

## 1.5 What's next

With your development environment configured, you are ready to learn about uv, the modern project management tool for Python.

In the next chapter, we will cover:

- Creating and managing Python projects.
- Pinning Python versions.
- Installing dependencies.
- Understanding the modern Python packaging ecosystem.

# 2 Python projects with uv

Objective

Learn how to use uv to create, manage, and maintain Python projects. Understand virtual environments, dependency management, and the modern Python packaging ecosystem.

## 2.1 Why virtual environments

In Python, virtual environments are not optional. They are essential for any serious project work.

Unlike R's renv (which primarily helps with reproducibility), Python virtual environments serve a fundamental purpose: **isolating project dependencies from the system Python**.

Here is why this matters:

- Different projects need different package versions.
- System Python library should never be modified directly.
- Dependency conflicts are common and destructive.
- Reproducibility requires exact version control.

Warning

Installing packages globally with `pip install` without a virtual environment will cause conflicts and break system tools. Always use virtual environments. To install Python packages as global command-line tools, use `pipx`.

## 2.2 What is uv

uv is a modern Python package and project manager written in Rust. It replaces and improves upon a scattered toolchain:

- `pip` (package installation)
- `venv` (virtual environment creation)
- `pyenv` (Python version management)
- `pip-tools` (dependency locking)
- `setuptools` (package building)

Benefits of uv:

- **Fast**: 10-100x faster than pip due to Rust implementation.
- **Complete**: Manages Python versions, dependencies, and builds.
- **Modern**: Uses `pyproject.toml` as the single source of truth.
- **Reliable**: Automatic dependency resolution and lock files.

In R terms, uv combines functionality from renv, devtools, usethis, and pak into a single, cohesive tool.

## 2.3 Python packaging standards

Python has standardized on `pyproject.toml` as the configuration file for all projects. This is similar to R's `DESCRIPTION` file but uses TOML format.

The Official Python packaging guide is available at https://packaging.python.org/.

Key concepts:

- `pyproject.toml` defines project metadata and dependencies.
- `uv.lock` records exact versions (like `renv.lock`).
- Build backends (like `hatchling`) create distributable packages.

## 2.4 Installing uv

Follow the official installation guide.

**macOS and Linux:**

```
curl -LsSf https://astral.sh/uv/install.sh | sh
```

**Windows:**

```
powershell -ExecutionPolicy ByPass -c "irm https://astral.sh/uv/install.ps1 | iex"
```

**Via Homebrew (macOS):**

```
brew install uv
```

Verify installation:

```
uv --version
```

## 2.5 Updating uv

uv can update itself:

```
uv self update
```

Regular updates are important because uv frequently adds support for new Python versions and features.

> **Note**
>
> uv uses Python distributions from the python-build-standalone project. These are optimized, portable Python builds that work consistently across platforms.

## 2.6 Initialize a project

Create a new Python project:

```
uv init pycsr-example
cd pycsr-example
```

This creates a basic structure:

```
pycsr-example/
   .python-version    # Pinned Python version
   pyproject.toml     # Project metadata and dependencies
   README.md          # Project documentation
   src/
       pycsr_example/
           __init__.py
```

### 2.6.1 Project structure

Notice the directory name uses hyphens (`pycsr-example`) while the package name uses underscores (`pycsr_example`). This is Python convention.

The `pyproject.toml` file contains project configuration:

```
[project]
name = "pycsr-example"
version = "0.1.0"
description = "Example clinical study report project"
dependencies = []

[build-system]
requires = ["hatchling"]
build-backend = "hatchling.build"
```

Key sections:

- `[project]`: Package metadata.
- `[project.dependencies]`: Hard, runtime dependencies.
- `[dependency-groups.dev]`: Development dependencies.
- `[build-system]`: How to build the package.

## 2.7 Pin Python version

Specify the exact Python version for your project:

```
uv python pin 3.13.9
```

This updates `.python-version` file so everyone uses the same Python version when they restore the environment.

> **!** Important
>
> Use the full `MAJOR.MINOR.PATCH` version (for example, `3.13.9`) rather than just `MAJOR.MINOR` (for example, `3.13`). This prevents drift as new patch versions are released.

Why pin the exact version:

- Patch releases can introduce subtle behavior changes.
- Reproducibility requires exact version matching.
- Regulatory submissions should document the exact Python version.

Check which Python versions are available:

```
uv python list
```

Install a specific Python version if needed:

```
uv python install 3.13.9
```

## 2.8 Managing dependencies

### 2.8.1 Adding dependencies

Add runtime dependencies:

```
uv add polars plotnine rtflite
```

Add development-only dependencies:

```
uv add --dev ruff pytest mypy
```

This updates `pyproject.toml`:

```
[project]
dependencies = [
    "polars>=1.34.0",
    "plotnine>=0.15.0",
    "rtflite>=1.0.2",
]

[dependency-groups.dev]
dependencies = [
    "ruff>=0.14.1",
    "pytest>=8.4.2",
    "mypy>=1.18.2",
]
```

> **Note**
>
> By default, uv adds dependencies with `>=` constraints. This allows updates within compatible versions. The lock file ensures exact versions are used.

### 2.8.2 Removing dependencies

Remove a package:

```
uv remove pandas
```

This removes the package from both `pyproject.toml` and the environment.

## 2.9 Lock files and syncing

### 2.9.1 Creating and updating the lock file

Generate or update the lock file:

```
uv sync
```

This creates `uv.lock`, which records:

- Exact version of every package.
- All transitive dependencies.
- Package hashes for verification.

The lock file ensures reproducibility across different machines and over time.

### 2.9.2 Upgrading dependencies

To update packages while respecting constraints in `pyproject.toml`:

```
uv lock --upgrade
```

Then synchronize the environment:

```
uv sync
```

This is similar to:

- R: `renv::update()` followed by `renv::snapshot()`.
- Node.js: `npm update` followed by `npm install`.

The two-step process (lock & sync) gives you control: you can review lock file changes before updating your environment.

## 2.10 Running commands

You have two options for running commands in your project environment.

### 2.10.1 Option 1: Activate the virtual environment

```
source .venv/bin/activate  # macOS/Linux
# or
.venv\Scripts\activate     # Windows
```

Then run commands directly:

```
python -m pycsr_example
pytest
ruff check
```

Deactivate when done:

```
deactivate
```

### 2.10.2 Option 2: Use `uv run`

Run commands without activation:

```
uv run python -m pycsr_example
uv run pytest
uv run ruff check
```

> **Tip**
>
> `uv run` is convenient for one-off commands and CI/CD scripts. For interactive work, activating the environment is often more ergonomic.

### 2.10.3 `uv run` and `uvx`

`uvx` runs tools in isolated, temporary environments:

```
uvx ruff check .
uvx black --check .
```

Use `uvx` when:

- Running tools you don't want to install in the project.
- Trying packages without adding them as dependencies.
- Running scripts that declare their own dependencies.

Use `uv run` when:

- Running project code.
- Running tests.
- Using project dependencies.

See using tools in uv for details.

## 2.11 Building and publishing

For creating distributable packages, you need a build backend. The simplest option is `hatchling`.

Add to `pyproject.toml`:

```
[build-system]
requires = ["hatchling"]
build-backend = "hatchling.build"
```

### 2.11.1 Build wheel

Create distribution files:

```
uv build
```

This creates: - `dist/pycsr_example-0.1.0.tar.gz` (source distribution) - `dist/pycsr_example-0.1.0-py3-none-any.whl` (wheel)

### 2.11.2 Publish to PyPI

Publish to the Python Package Index:

```
uv publish
```

> **Note**
>
> Building and publishing are not typically needed for internal clinical reporting projects. However, if you develop reusable tools like table generation packages, open sourcing in a GitHub repository and publishing on PyPI will make them more visible.

## 2.12 Exercise

Create a small project to practice uv commands:

1. Initialize a new project called `csr-practice`.
2. Pin Python to version 3.13.9 (or latest available).
3. Add `polars` as a dependency.
4. Add `pytest` as a development dependency.
5. Examine the generated `pyproject.toml` and `uv.lock` files.
6. Run Python using `uv run python --version`.

View solution

```
# Initialize project
uv init csr-practice
cd csr-practice

# Pin Python version
uv python pin 3.13.9

# Add dependencies
uv add polars
uv add --dev pytest

# View configuration
cat pyproject.toml

# Check lock file
```

```
cat uv.lock

# Run Python
uv run python --version
```

Your `pyproject.toml` should look similar to:

```
[project]
name = "csr-practice"
version = "0.1.0"
description = "Add your description here"
dependencies = [
    "polars>=1.18.0",
]

[project.optional-dependencies]
dev = [
    "pytest>=8.3.4",
]

[build-system]
requires = ["hatchling"]
build-backend = "hatchling.build"
```

## 2.13 What's next

Now that you understand uv basics, the next chapter covers the Python package toolchain:

- Formatting and linting with Ruff.
- Type checking with mypy.
- Testing with pytest.
- Documentation generation.
- Development workflows for clinical reporting.

# 3 Python package toolchain

Objective

Learn the essential development tools for Python projects: formatting, linting, type checking, testing, and documentation. Build a professional development workflow for clinical reporting.

## 3.1 The modern Python toolchain

In R, packages like `devtools`, `usethis`, `styler`, `lintr`, and `testthat` provide development infrastructure. Python's ecosystem distributes these functions across specialized tools.

For clinical reporting projects, we recommend:

- **uv**: Package and environment management.
- **Ruff**: Code formatting and linting.
- **mypy**: Static type checking.
- **pytest**: Unit testing framework.
- **quartodoc**: Documentation and reporting.

All tools are installed as development dependencies and configured through `pyproject.toml`.

For R users, think of this as: uv = renv + pak + devtools, Ruff = styler + lintr, pytest = testthat, mypy = (no direct R equivalent).

## 3.2 Ruff: Formatting and linting

Ruff is an super fast linter and formatter written in Rust. It replaces multiple legacy tools (Black, isort, Flake8, pyupgrade) with a single, consistent interface.

### 3.2.1 Installation

Add Ruff as a development dependency:

```
uv add --dev ruff
```

### 3.2.2 Code formatting

Format your code:

```
uv run ruff format
```

Or using `uvx`:

```
uvx ruff format
```

Ruff format:

- Enforces consistent style (like Black).
- Sorts imports automatically.
- Removes trailing whitespace.
- Ensures consistent line lengths.

### 3.2.3 Linting

Check for linting issues:

```
uv run ruff check
```

Fix auto-fixable issues:

```
uv run ruff check --fix
```

Ruff detects:

- Unused imports and variables.
- Undefined names.
- Style violations.
- Common anti-patterns.
- Security issues.

### 3.2.4 Configuration

Add Ruff configuration to `pyproject.toml`:

```toml
[tool.ruff]
line-length = 88
target-version = "py313"

[tool.ruff.format]
quote-style = "double"
indent-style = "space"

[tool.ruff.lint]
select = [
    "E",    # pycodestyle
    "F",    # Pyflakes
    "UP",   # pyupgrade
    "B",    # flake8-bugbear
    "SIM",  # flake8-simplify
    "I",    # isort
]
ignore = []
```

> **Note**
>
> Line length of 88 characters is the Python community standard. It balances readability with modern screen sizes.

## 3.3 Type checking with mypy

Python supports optional type annotations through PEP 484. Type annotations improve code clarity and catch errors before runtime.

### 3.3.1 Why type checking matters

For clinical programming:

- Catch data transformation errors at development time.
- Document expected DataFrame structures.
- Improve IDE autocomplete and refactoring.
- Reduce runtime errors in production.

### 3.3.2 Installation

Add mypy as a development dependency:

```
uv add --dev mypy
```

### 3.3.3 Basic usage

Check types in your code:

```
uv run mypy .
```

### 3.3.4 Type annotation example

Without types:

```
def calculate_bmi(weight, height):
    return weight / (height ** 2)
```

With types:

```
def calculate_bmi(weight: float, height: float) -> float:
    """Calculate BMI from weight (kg) and height (m)."""
    return weight / (height ** 2)
```

The type checker verifies:

- Arguments are the correct type.
- Return value matches the declared type.
- Operations are valid for the types used.

### 3.3.5 Configuration

Add mypy settings to `pyproject.toml`:

```
[tool.mypy]
python_version = "3.13"
warn_return_any = true
warn_unused_configs = true
disallow_untyped_defs = false
disallow_incomplete_defs = true
check_untyped_defs = true
no_implicit_optional = true
```

Start with lenient settings (`disallow_untyped_defs = false`) and progressively tighten as you add type annotations to your codebase.

### 3.3.6 Type stubs for libraries

Some libraries don't include type information. Install type stubs when available:

```
uv add --dev types-tabulate
```

> **Note**
>
> Popular data science libraries like `polars` include built-in type annotations. Older libraries like `pandas` require separate stub packages (`pandas-stubs`).

## 3.4 Testing with pytest

pytest is Python's de facto standard testing framework. It's more powerful and ergonomic than the built-in `unittest` module.

### 3.4.1 Installation

Add pytest and coverage tools:

```
uv add --dev pytest pytest-cov
```

### 3.4.2 Writing tests

Create a `tests/` directory:

```
pycsr-example/
   src/
       pycsr_example/
           __init__.py
   tests/
       test_calculations.py
```

Write a simple test in `tests/test_calculations.py`:

```
from pycsr_example.calculations import calculate_bmi
import pytest

def test_calculate_bmi():
    # Normal BMI calculation
    assert calculate_bmi(70, 1.75) == pytest.approx(22.857142857142858)

def test_calculate_bmi_underweight():
    # BMI < 18.5 indicates underweight
    assert calculate_bmi(50, 1.75) < 18.5
```

### 3.4.3 Running tests

Run all tests:

```
uv run pytest
```

Run with verbose output:

```
uv run pytest -v
```

Run specific test file:

```
uv run pytest tests/test_calculations.py
```

### 3.4.4 Code coverage

Generate coverage report:

```
uv run pytest --cov=pycsr_example --cov-report=term
```

Generate HTML coverage report:

```
uv run pytest --cov=pycsr_example --cov-report=html
```

This creates `htmlcov/index.html` showing which lines are tested.

> **!** Important
>
> For regulatory submissions, high test coverage demonstrates code quality. Aim for >80% coverage for critical data transformation and statistical computation functions.

### 3.4.5 pytest configuration

Add pytest settings to `pyproject.toml`:

```
[tool.pytest.ini_options]
testpaths = ["tests"]
python_files = ["test_*.py"]
python_functions = ["test_*"]
addopts = [
    "--strict-markers",
    "--strict-config",
    "-ra",
]
```

## 3.5 Documentation generation

For clinical reporting projects, documentation serves two purposes:

1. **Code documentation**: Function and module documentation.
2. **Report generation**: Analysis reports and TLFs.

### 3.5.1 Quarto for reports

We use Quarto for creating reproducible analysis documents:

```
# Install Quarto separately (not via uv)
# See: https://quarto.org/docs/get-started/
```

Quarto documents (`.qmd` files) combine:

- Markdown text.
- Python code cells.
- Generated outputs (tables, listings, figures).

This book itself is written in Quarto.

### 3.5.2 quartodoc for API documentation

For packages that need API documentation (similar to R's `pkgdown`), use `quartodoc`:

```
uv add --dev quartodoc
```

quartodoc generates documentation from docstrings and integrates with Quarto for full website generation.

For analysis projects (rather than reusable packages), Quarto alone is usually sufficient. Use `quartodoc` when building analysis packages for team to collaborate on.

## 3.6 Development workflow

Putting it all together, a typical development cycle looks like:

1. **Format code**: `uv run ruff format`
2. **Check linting**: `uv run ruff check --fix`
3. **Verify types**: `uv run mypy .`
4. **Run tests**: `uv run pytest --cov=pycsr_example`
5. **Generate reports**: `quarto render`

### 3.6.1 Pre-commit automation

You can automate these checks using Git hooks (not covered in this book), but manual execution provides better learning and control during development.

## 3.7 Clinical project structure guidelines

In case you need clinical reporting projects using both R and Python:

**Separate R and Python directories:**

```
project/
   r-package/          # R package for R-based analyses
       DESCRIPTION
       R/
       tests/
   python-package/     # Python package for Python-
based analyses
       pyproject.toml
       src/
       tests/
   data/               # Shared input data (SDTM, ADaM)
   output/             # Shared output (TLFs, reports)
```

**Why separate?**

As John Carmack noted: "It's almost always a mistake to mix languages in a single project."

Reasons:

- Different build systems.
- Different dependency management.
- Different testing frameworks.
- Different IDE configurations.

**Shared resources:**

- Input datasets (SDTM, ADaM) can be in a common `data/` directory.
- Output deliverables can go to a common `output/` directory.
- Documentation can reference both implementations.

> **Note**
>
> For this book, we focus exclusively on Python. Mixed R/Python workflows are beyond scope but follow the same principles.

## 3.8 Exercise

Set up a complete development environment:

1. Create a new project with `uv init dev-practice`.
2. Add development dependencies: `ruff`, `mypy`, `pytest`, `pytest-cov`.
3. Create a simple function in `src/dev_practice/stats.py`:

```
def mean(values: list[float]) -> float:
    return sum(values) / len(values)
```

4. Write a test in `tests/test_stats.py`.
5. Run Ruff format and check.

6. Run mypy type checking.
7. Run pytest with coverage.

View solution

```
# Create project
uv init dev-practice
cd dev-practice

# Add dev dependencies
uv add --dev ruff mypy pytest pytest-cov

# Create stats module
mkdir -p src/dev_practice
cat > src/dev_practice/stats.py << 'EOF'
def mean(values: list[float]) -> float:
    """Calculate the arithmetic mean of a list of numbers."""
    if not values:
        raise ValueError("Cannot calculate mean of empty list")
    return sum(values) / len(values)
EOF

# Create test file
mkdir -p tests
cat > tests/test_stats.py << 'EOF'
import pytest
from dev_practice.stats import mean

def test_mean_basic():
    assert mean([1.0, 2.0, 3.0]) == 2.0

def test_mean_single_value():
    assert mean([5.0]) == 5.0

def test_mean_empty_raises():
    with pytest.raises(ValueError):
        mean([])
EOF

# Run checks
```

```
uv run ruff format .
uv run ruff check .
uv run mypy src/
uv run pytest --cov=dev_practice --cov-report=term
```

Expected output from pytest:

```
============================= test session starts ==============================
collected 3 items

tests/test_stats.py ...                                                  [100%]

---------- coverage: platform darwin, python 3.13.9-
final-0 ----------
Name                        Stmts   Miss  Cover
---------------------------------------------
src/dev_practice/__init__.py    0      0   100%
src/dev_practice/stats.py       4      0   100%
---------------------------------------------
TOTAL                           4      0   100%

============================== 3 passed in 0.05s ===============================
```

## 3.9 Example repositories

Demo project repositories have been created:

- **Python package example**: [Link to be added]
- **eCTD package example**: [Link to be added]

With the knowledge from this chapter, you can understand how these projects are organized and develop similar professional Python packages for clinical reporting.

## 3.10 What's next

You now have a complete Python development environment with:

- uv for project and dependency management.
- Ruff for code quality.
- mypy for type safety.
- pytest for testing.
- Quarto for documentation.

Next part will introduce how to create real clinical study reports, demonstrating TLF generation with `polars` and `rtflite`.

# Part II

# Clinical trial project

# 4 TLF overview

Objective

Understand the regulatory context and importance of TLFs in clinical study reports. Learn basic concepts of Polars and rtflite for TLF generation.

## 4.1 Overview

Tables, listings, and figures (TLFs) are essential components of clinical study reports (CSRs) and regulatory submissions. Following ICH E3 guidance, TLFs provide standardized summaries of clinical trial data that support regulatory decision-making.

This chapter provides an overview of creating TLFs using Python, focusing on the tools and workflows demonstrated throughout this book.

## 4.2 Background

Submitting clinical trial results to regulatory agencies is a crucial aspect of clinical development. The Electronic Common Technical Document (eCTD) has emerged as the global standard format for regulatory submissions. For instance, the United States Food and Drug Administration (US FDA) mandates the use of eCTD for new drug applications and biologics license applications.

A CSR provides comprehensive information about the methods and results of an individual clinical study. To support the statistical analysis, numerous tables, listings, and figures are included within the main text and appendices. The creation of a

CSR is a collaborative effort that involves various professionals such as clinicians, medical writers, statisticians, and statistical programmers.

Within an organization, these professionals typically collaborate to define, develop, validate, and deliver the necessary TLFs for a CSR. These TLFs serve to summarize the efficacy and/or safety of the pharmaceutical product under study. In the pharmaceutical industry, Microsoft Word is widely utilized for CSR preparation. As a result, the deliverables from statisticians and statistical programmers are commonly provided in formats such as `.rtf`, `.doc`, `.docx` to align with industry standards and requirements.

> **Note**
>
> Each organization may define specific TLF format requirements that differ from the examples in this book. It is advisable to consult and adhere to the guidelines and specifications set by your respective organization when preparing TLFs for submission.

By following the ICH E3 guidance, most TLFs in a CSR are located at:

- Section 10: Study participants
- Section 11: Efficacy evaluation
- Section 12: Safety evaluation
- Section 14: Tables, listings, and figures referenced but not included in the text
- Section 16: Appendices

## 4.3 Datasets

The dataset structure follows CDISC Analysis Data Model (ADaM).

In this project, we use publicly available CDISC pilot study data, which is accessible through the CDISC GitHub repository.

We have converted these datasets from the `.xpt` format to the `.parquet` format for ease of use and compatibility with Python tools. The dataset structure adheres to the CDISC Analysis Data Model (ADaM) standard.

## 4.4 Tools

To exemplify the generation of TLFs in RTF format, we rely on the functionality provided by two Python packages:

- **Polars**: Preparation of datasets in a format suitable for reporting purposes. Polars offers a comprehensive suite of tools and functions for data manipulation and transformation, ensuring that the data is structured appropriately.
- **rtflite**: Creation of RTF files. The rtflite package offers functions specifically designed for generating RTF files, allowing us to produce TLFs in the desired format.

## 4.5 Polars

Polars is an open-source library for data manipulation implemented in Rust with Python bindings. It offers exceptional performance while maintaining a user-friendly interface for interactive data analysis.

Key advantages of Polars include:

- **Performance**: 10-100x faster than pandas for most operations due to Rust implementation
- **Memory efficiency**: Lazy evaluation and columnar storage reduce memory usage
- **Familiar syntax**: Similar to tidyverse-style pipelines, making it accessible to R users
- **Type safety**: Strong typing system that catches errors early in development

The creators of Polars have provided exceptional documentation and tutorials that serve as valuable resources for learning and mastering the functionalities of the library.

Furthermore, several books are available that serve as introductions to Polars:

- Python Polars: The Definitive Guide

> **Note**
>
> In this book, we assume that the reader has some experience with data manipulation concepts. This prior knowledge enables a more efficient and focused exploration of the clinical reporting concepts presented throughout the book.

To illustrate the basic usage of Polars, let's work with a sample ADSL dataset. This dataset contains subject-level information from a clinical trial, which will serve as a practical example for generating summaries using Polars.

```
import polars as pl
```

```
polars.config.Config
```

```
# Read clinical data
adsl = pl.read_parquet("data/adsl.parquet")

# Select columns
adsl = adsl.select(["USUBJID", "TRT01A", "AGE", "SEX"])

# Basic data exploration
adsl
```

| USUBJID<br>str | TRT01A<br>str | AGE<br>f64 | SEX<br>str |
|---|---|---|---|
| "01-701-1015" | "Placebo" | 63.0 | "Female" |
| "01-701-1023" | "Placebo" | 64.0 | "Male" |
| "01-701-1028" | "Xanomeline High Dose" | 71.0 | "Male" |
| … | … | … | … |
| "01-718-1371" | "Xanomeline High Dose" | 69.0 | "Female" |
| "01-718-1427" | "Xanomeline High Dose" | 74.0 | "Female" |

Key Polars operations for clinical reporting include:

### 4.5.1 I/O

Polars supports multiple data formats for input and output (see the I/O guide). For clinical development, we recommend the `.parquet` format because tools in Python, R, and Julia can read and write it without conversion. The example below loads subject-level ADSL data with Polars.

```
import polars as pl
```

```
adsl = pl.read_parquet("data/adsl.parquet")
adsl = adsl.select("STUDYID", "USUBJID", "TRT01A", "AGE", "SEX") # select columns
adsl
```

| STUDYID<br>str | USUBJID<br>str | TRT01A<br>str | AGE<br>f64 | SEX<br>str |
|---|---|---|---|---|
| ”CDISCPILOT01” | ”01-701-1015” | ”Placebo” | 63.0 | ”Female” |
| ”CDISCPILOT01” | ”01-701-1023” | ”Placebo” | 64.0 | ”Male” |
| ”CDISCPILOT01” | ”01-701-1028” | ”Xanomeline High Dose” | 71.0 | ”Male” |
| … | … | … | … | … |
| ”CDISCPILOT01” | ”01-718-1371” | ”Xanomeline High Dose” | 69.0 | ”Female” |
| ”CDISCPILOT01” | ”01-718-1427” | ”Xanomeline High Dose” | 74.0 | ”Female” |

### 4.5.2 Filtering

Filtering in Polars uses the `.filter()` method with column expressions. Below are examples applied to the ADSL data.

```
# Filter female subjects
adsl.filter(pl.col("SEX") == "Female")
```

| STUDYID<br>str | USUBJID<br>str | TRT01A<br>str | AGE<br>f64 | SEX<br>str |
|---|---|---|---|---|
| ”CDISCPILOT01” | ”01-701-1015” | ”Placebo” | 63.0 | ”Female” |
| ”CDISCPILOT01” | ”01-701-1034” | ”Xanomeline High Dose” | 77.0 | ”Female” |
| ”CDISCPILOT01” | ”01-701-1047” | ”Placebo” | 85.0 | ”Female” |
| … | … | … | … | … |

| STUDYID<br>str | USUBJID<br>str | TRT01A<br>str | AGE<br>f64 | SEX<br>str |
|---|---|---|---|---|
| ”CDISCPILOT01” | ”01-718-1371” | ”Xanomeline High Dose” | 69.0 | ”Female” |
| ”CDISCPILOT01” | ”01-718-1427” | ”Xanomeline High Dose” | 74.0 | ”Female” |

```
# Filter subjects with Age >= 65
adsl.filter(pl.col("AGE") >= 65)
```

| STUDYID<br>str | USUBJID<br>str | TRT01A<br>str | AGE<br>f64 | SEX<br>str |
|---|---|---|---|---|
| ”CDISCPILOT01” | ”01-701-1028” | ”Xanomeline High Dose” | 71.0 | ”Male” |
| ”CDISCPILOT01” | ”01-701-1033” | ”Xanomeline Low Dose” | 74.0 | ”Male” |
| ”CDISCPILOT01” | ”01-701-1034” | ”Xanomeline High Dose” | 77.0 | ”Female” |
| … | … | … | … | … |
| ”CDISCPILOT01” | ”01-718-1371” | ”Xanomeline High Dose” | 69.0 | ”Female” |
| ”CDISCPILOT01” | ”01-718-1427” | ”Xanomeline High Dose” | 74.0 | ”Female” |

### 4.5.3 Deriving

Deriving new variables is common in clinical data analysis for creating age groups, BMI categories, or treatment flags. Polars uses `.with_columns()` to add new columns while keeping existing ones.

```
# Create age groups
adsl.with_columns([
    pl.when(pl.col("AGE") < 65)
      .then(pl.lit("<65"))
      .otherwise(pl.lit(">=65"))
      .alias("AGECAT")
])
```

| STUDYID<br>str | USUBJID<br>str | TRT01A<br>str | AGE<br>f64 | SEX<br>str | AGECAT<br>str |
|---|---|---|---|---|---|
| ”CDISCPILOT01” | ”01-701-1015” | ”Placebo” | 63.0 | ”Female” | ”<65” |
| ”CDISCPILOT01” | ”01-701-1023” | ”Placebo” | 64.0 | ”Male” | ”<65” |

| STUDYID<br>str | USUBJID<br>str | TRT01A<br>str | AGE<br>f64 | SEX<br>str | AGECAT<br>str |
|---|---|---|---|---|---|
| ”CDISCPILOT01” | ”01-701-1028” | ”Xanomeline High Dose” | 71.0 | ”Male” | ”>=65” |
| … | … | … | … | … | … |
| ”CDISCPILOT01” | ”01-718-1371” | ”Xanomeline High Dose” | 69.0 | ”Female” | ”>=65” |
| ”CDISCPILOT01” | ”01-718-1427” | ”Xanomeline High Dose” | 74.0 | ”Female” | ”>=65” |

### 4.5.4 Grouping

Grouping operations are fundamental for creating summary statistics in clinical reports. Polars uses `group_by()` followed by aggregation functions to compute counts, means, and other statistics by categorical variables like treatment groups.

The `.count()` method provides a quick way to get subject counts by group.

```
# Count by treatment group
adsl.group_by("TRT01A").len().sort("TRT01A")
```

| TRT01A<br>str | len<br>u32 |
|---|---|
| ”Placebo” | 86 |
| ”Xanomeline High Dose” | 84 |
| ”Xanomeline Low Dose” | 84 |

You can also use `.agg()` with multiple aggregation functions:

```
# Age statistics by treatment group
adsl.group_by("TRT01A").agg([
    pl.col("AGE").mean().round(1).alias("mean_age"),
    pl.col("AGE").std().round(2).alias("sd_age")
]).sort("TRT01A")
```

| TRT01A<br>str | mean_age<br>f64 | sd_age<br>f64 |
|---|---|---|
| ”Placebo” | 75.2 | 8.59 |
| ”Xanomeline High Dose” | 74.4 | 7.89 |
| ”Xanomeline Low Dose” | 75.7 | 8.29 |

### 4.5.5 Joining

Joining datasets is essential for combining subject-level data (ADSL) with event-level data (e.g. ADAE, ADLB). Polars supports various join types including inner, left, and full joins.

Here is a toy example that splits ADSL and joins it back by `USUBJID`.

```
# Create a simple demographics subset
demo = adsl.select("USUBJID", "AGE", "SEX").head(3)

# Create treatment info subset
trt = adsl.select("USUBJID", "TRT01A").head(3)

# Left join to combine datasets
demo.join(trt, on="USUBJID", how="left")
```

| USUBJID<br>str | AGE<br>f64 | SEX<br>str | TRT01A<br>str |
|---|---|---|---|
| ”01-701-1015” | 63.0 | ”Female” | ”Placebo” |
| ”01-701-1023” | 64.0 | ”Male” | ”Placebo” |
| ”01-701-1028” | 71.0 | ”Male” | ”Xanomeline High Dose” |

### 4.5.6 Pivoting

Pivoting transforms data from long to wide format, commonly needed for creating tables. Use `.pivot()` to reshape grouped data into columns.

```
# Create summary by treatment and sex
(
    adsl
        .group_by(["TRT01A", "SEX"])
        .agg(pl.len().alias("n"))
        .pivot(
            values="n",
            index="SEX",
            on="TRT01A"
        )
)
```

| SEX<br>str | Placebo<br>u32 | Xanomeline Low Dose<br>u32 | Xanomeline High Dose<br>u32 |
|---|---|---|---|
| ”Male” | 33 | 34 | 44 |
| ”Female” | 53 | 50 | 40 |

Having covered the essential Polars operations for data manipulation, we now turn to the second component of our clinical reporting workflow: formatting and presenting the processed data in regulatory-compliant RTF format.

## 4.6 rtflite

```
import rtflite as rtf
```

rtflite is a Python package for creating production-ready tables and figures in RTF format. While Polars handles the data processing and statistical calculations, rtflite focuses exclusively on the presentation layer. The package is designed to:

- Provide simple Python classes that map to table elements (title, headers, body, footnotes) for intuitive table construction.
- Offer a canonical Python API with a clear, composable interface.

- Focus exclusively on **table formatting and layout**, leaving data manipulation to dataframe libraries like polars or pandas.
- Minimize external dependencies for maximum portability and reliability.

Creating an RTF table involves three steps:

- Design the desired table layout and structure.
- Configure the appropriate rtflite components.
- Generate and save the RTF document.

This guide introduces rtflite's core components and demonstrates how to turn dataframes into Tables, Listings, and Figures (TLFs) for clinical reporting.

### 4.6.1 Data: adverse events

To explore the RTF generation capabilities in rtflite, we will use the dataset `data/adae.parquet`. This dataset contains adverse event (AE) information from a clinical trial.

Below are the meanings of relevant variables:

- `USUBJID`: Unique Subject Identifier
- `TRTA`: Actual Treatment
- `AEDECOD`: Dictionary-Derived Term

```
# Load adverse events data
df = pl.read_parquet("data/adae.parquet")

df.select(["USUBJID", "TRTA", "AEDECOD"])
```

| USUBJID<br>str | TRTA<br>str | AEDECOD<br>str |
|---|---|---|
| "01-701-1015" | "Placebo" | "APPLICATION SITE ERYTHEMA" |
| "01-701-1015" | "Placebo" | "APPLICATION SITE PRURITUS" |
| "01-701-1015" | "Placebo" | "DIARRHOEA" |
| … | … | … |
| "01-718-1427" | "Xanomeline High Dose" | "DECREASED APPETITE" |

| USUBJID<br>str | TRTA<br>str | AEDECOD<br>str |
|---|---|---|
| ”01-718-1427” | ”Xanomeline High Dose” | ”NAUSEA” |

### 4.6.2 Table-ready data

In this AE example, we provide the number of subjects with each type of AE by treatment group.

```
tbl = (
    df.group_by(["TRTA", "AEDECOD"])
    .agg(pl.len().alias("n"))
    .sort("TRTA")
    .pivot(values="n", index="AEDECOD", on="TRTA")
    .fill_null(0)
    .sort("AEDECOD")  # Sort by adverse event name to match R output
)

tbl
```

| AEDECOD<br>str | Placebo<br>u32 | Xanomeline High Dose<br>u32 | Xanomeline Low Dose<br>u32 |
|---|---|---|---|
| ”ABDOMINAL DISCOMFORT” | 0 | 1 | 0 |
| ”ABDOMINAL PAIN” | 1 | 2 | 3 |
| ”ACROCHORDON EXCISION” | 0 | 1 | 0 |
| … | … | … | … |
| ”WOUND” | 0 | 0 | 2 |
| ”WOUND HAEMORRHAGE” | 0 | 1 | 0 |

### 4.6.3 Table component classes

rtflite provides dedicated classes for each table component. Commonly used classes include:

- `RTFPage`: RTF page information (orientation, margins, pagination).

- `RTFPageHeader`: Page headers with page numbering (compatible with r2rtf).
- `RTFPageFooter`: Page footers for attribution and notices.
- `RTFTitle`: RTF title information.
- `RTFColumnHeader`: RTF column header information.
- `RTFBody`: RTF table body information.
- `RTFFootnote`: RTF footnote information.
- `RTFSource`: RTF data source information.

These component classes work together to build complete RTF documents. A full list of all classes and their parameters can be found in the API reference.

### 4.6.4 Simple example

A minimal example below illustrates how to combine components to create an RTF table.

- `RTFBody()` defines table body layout.
- `RTFDocument()` transfers table layout information into RTF syntax.
- `write_rtf()` saves encoded RTF into a `.rtf` file.

```
rtf/tlf_overview1.rtf
```

```
PosixPath('pdf/tlf_overview1.pdf')
```

### 4.6.5 Column width

If we want to adjust the width of each column to provide more space to the first column, this can be achieved by updating `col_rel_width` in `RTFBody`.

The input of `col_rel_width` is a list with the same length as the number of columns. This argument defines the relative length of each column within a pre-defined total column width.

In this example, the defined relative width is 3:2:2:2. Only the ratio of `col_rel_width` is used. Therefore it is equivalent to use `col_rel_width = [6,4,4,4]` or `col_rel_width = [1.5,1,1,1]`.

```
rtf/tlf_overview2.rtf
```

```
PosixPath('pdf/tlf_overview2.pdf')
```

### 4.6.6 Column headers

In `RTFColumnHeader`, the `text` argument provides the column header content as a list of strings.

```
rtf/tlf_overview3.rtf
```

```
PosixPath('pdf/tlf_overview3.pdf')
```

We also allow column headers to be displayed in multiple lines. If an empty column name is needed for a column, you can insert an empty string. For example, `["name 1", "", "name 3"]`.

In `RTFColumnHeader`, the `col_rel_width` can be used to align column headers with different numbers of columns.

By using `RTFColumnHeader` with `col_rel_width`, one can customize complex column headers. If there are multiple pages, the column header will repeat on each page by default.

```
rtf/tlf_overview4.rtf
```

```
PosixPath('pdf/tlf_overview4.pdf')
```

### 4.6.7 Titles, footnotes, and data source

RTF documents can include additional components to provide context and documentation:

- `RTFTitle`: Add document titles and subtitles
- `RTFFootnote`: Add explanatory footnotes
- `RTFSource`: Add data source attribution

```
rtf/tlf_overview5.rtf
```

```
PosixPath('pdf/tlf_overview5.pdf')
```

Note the use of `\\line` in column headers to create line breaks within cells.

### 4.6.8 Text formatting and alignment

rtflite supports various text formatting options:

- **Text formatting**: Bold (`b`), italic (`i`), underline (`u`), strikethrough (`s`)
- **Text alignment**: Left (`l`), center (`c`), right (`r`), justify (`j`)
- **Font properties**: Font size, font family

```
rtf/tlf_overview6.rtf
```

```
PosixPath('pdf/tlf_overview6.pdf')
```

### 4.6.9 Border customization

Table borders can be customized extensively:

- **Border styles**: `single`, `double`, `thick`, `dotted`, `dashed`
- **Border sides**: `border_top`, `border_bottom`, `border_left`, `border_right`
- **Page borders**: `border_first`, `border_last` for first/last rows across pages

```
rtf/tlf_overview7.rtf
```

```
PosixPath('pdf/tlf_overview7.pdf')
```

## 4.7 Next Steps

Having covered the fundamental concepts and tools for creating clinical TLFs with Python, readers can explore specific implementations based on their requirements:

Each chapter provides step-by-step tutorials with reproducible code examples that can be adapted for specific clinical reporting requirements.

# 5 Disposition of participants

Objective

Create participant disposition tables to track how participants flow through the study from enrollment to completion. Learn to analyze completion status and discontinuation reasons using Polars and create regulatory-compliant disposition tables with rtflite.

## 5.1 Overview

Clinical trials needs to track how participants flow through a study from enrollment to completion. Following ICH E3 guidance, regulatory submissions require a disposition table in Section 10.1 that summarizes:

- **Enrolled**: Total participants who entered the study
- **Completed**: Participants who finished the study protocol
- **Discontinued**: Participants who left early and their reasons

This tutorial shows you how to create a regulatory-compliant disposition table using Python's `rtflite` package.

```
import polars as pl # Manipulate data
import rtflite as rtf # Reporting in RTF format
```

```
polars.config.Config
```

## 5.2 Step 1: Load Data

We start by loading the Subject-level Analysis Dataset (ADSL), which contains all participant information needed for our disposition table.

The ADSL dataset stores participant-level information including treatment assignments and study completion status. We're using the `parquet` format for data storage.

```
adsl = pl.read_parquet("data/adsl.parquet")
```

Let's examine the key variables we'll use to build our disposition table:

- **USUBJID**: Unique identifier for each participant
- **TRT01P**: Treatment name (text)
- **TRT01PN**: Treatment group (numeric code)
- **DISCONFL**: Flag indicating if participant discontinued (Y/N)
- **DCREASCD**: Specific reason for discontinuation

```
adsl.select(["USUBJID", "TRT01P", "TRT01PN", "DISCONFL", "DCREASCD"])
```

| USUBJID<br>str | TRT01P<br>str | TRT01PN<br>i64 | DISCONFL<br>str | DCREASCD<br>str |
|---|---|---|---|---|
| "01-701-1015" | "Placebo" | 0 | "" | "Completed" |
| "01-701-1023" | "Placebo" | 0 | "Y" | "Adverse Event" |
| "01-701-1028" | "Xanomeline High Dose" | 81 | "" | "Completed" |
| … | … | … | … | … |
| "01-718-1371" | "Xanomeline High Dose" | 81 | "Y" | "Adverse Event" |
| "01-718-1427" | "Xanomeline High Dose" | 81 | "Y" | "Lack of Efficacy" |

## 5.3 Step 2: Count Total Participants

First, we count how many participants were enrolled in each treatment group.

We group participants by treatment arm and count them using `.group_by()` and `.agg()`. The `.pivot()` operation reshapes our data from long format (rows for each treatment) to wide format (columns for each treatment), which matches the standard disposition table layout.

```
n_rand = (
    adsl
    .group_by("TRT01PN")
    .agg(n = pl.len())
    .with_columns([
        pl.lit("Participants in population").alias("row"),
        pl.lit(None, dtype=pl.Float64).alias("pct") # Placeholder for percentage (not applicabl
    ])
    .pivot(
        index="row",
        on="TRT01PN",
        values=["n", "pct"],
        sort_columns=True
    )
)

n_rand
```

| row | n_0 | n_54 | n_81 | pct_0 | pct_54 | pct_81 |
|---|---|---|---|---|---|---|
| str | u32 | u32 | u32 | f64 | f64 | f64 |
| ”Participants in population” | 86 | 84 | 84 | null | null | null |

## 5.4 Step 3: Count Completed Participants

Next, we identify participants who successfully completed the study and calculate what percentage they represent of each treatment group.

We filter for participants where `DCREASCD == "Completed"`, then calculate both counts and percentages. The `.join()` operation brings in the total count for each treatment group so we can compute percentages.

```
n_complete = (
    adsl
    .filter(pl.col("DCREASCD") == "Completed")
    .group_by("TRT01PN")
    .agg(n = pl.len())
    .join(
        adsl.group_by("TRT01PN").agg(total = pl.len()),
        on="TRT01PN"
    )
    .with_columns([
        pl.lit("Completed").alias("row"),
        (100.0 * pl.col("n") / pl.col("total")).round(1).alias("pct")
    ])
    .pivot(
        index="row",
        on="TRT01PN",
        values=["n", "pct"],
        sort_columns=True
    )
)

n_complete
```

| row | n_0 | n_54 | n_81 | pct_0 | pct_54 | pct_81 |
|---|---|---|---|---|---|---|
| str | u32 | u32 | u32 | f64 | f64 | f64 |
| ”Completed” | 58 | 25 | 27 | 67.4 | 29.8 | 32.1 |

## 5.5 Step 4: Count Discontinued Participants

Now we count participants who left the study early, regardless of their specific reason.

We filter for participants where the discontinuation flag `DISCONFL == "Y"`, then follow the same pattern of counting and calculating percentages within each treatment group.

```
n_disc = (
    adsl
    .filter(pl.col("DISCONFL") == "Y")
    .group_by("TRT01PN")
    .agg(n = pl.len())
    .join(
        adsl.group_by("TRT01PN").agg(total = pl.len()),
        on="TRT01PN"
    )
    .with_columns([
        pl.lit("Discontinued").alias("row"),
        (100.0 * pl.col("n") / pl.col("total")).round(1).alias("pct")
    ])
    .pivot(
        index="row",
        on="TRT01PN",
        values=["n", "pct"],
        sort_columns=True
    )
)

n_disc
```

| row<br>str | n_0<br>u32 | n_54<br>u32 | n_81<br>u32 | pct_0<br>f64 | pct_54<br>f64 | pct_81<br>f64 |
|---|---|---|---|---|---|---|
| ”Discontinued” | 28 | 59 | 57 | 32.6 | 70.2 | 67.9 |

## 5.6 Step 5: Break Down Discontinuation Reasons

For regulatory reporting, we need to show the specific reasons why participants discontinued.

We filter out completed participants, then group by both treatment and discontinuation reason. The indentation (four spaces) in the row labels helps show these are subcategories under “Discontinued”. We also use `.fill_null(0)` to handle cases where

certain discontinuation reasons don't occur in all treatment groups.

```
n_reason = (
    adsl
    .filter(pl.col("DCREASCD") != "Completed")
    .group_by(["TRT01PN", "DCREASCD"])
    .agg(n = pl.len())
    .join(
        adsl.group_by("TRT01PN").agg(total = pl.len()),
        on="TRT01PN"
    )
    .with_columns([
        pl.concat_str([pl.lit("    "), pl.col("DCREASCD")]).alias("row"),
        (100.0 * pl.col("n") / pl.col("total")).round(1).alias("pct")
    ])
    .pivot(
        index="row",
        on="TRT01PN",
        values=["n", "pct"],
        sort_columns=True
    )
    .with_columns([
        pl.col(["n_0", "n_54", "n_81"]).fill_null(0),
        pl.col(["pct_0", "pct_54", "pct_81"]).fill_null(0.0)
    ])
    .sort("row")
)

n_reason
```

| row<br>str | n_0<br>u32 | n_54<br>u32 | n_81<br>u32 | pct_0<br>f64 | pct_54<br>f64 | pct_81<br>f64 |
|---|---|---|---|---|---|---|
| "    Adverse Event" | 8 | 44 | 40 | 9.3 | 52.4 | 47.6 |
| "    Death" | 2 | 1 | 0 | 2.3 | 1.2 | 0.0 |
| "    I/E Not Met" | 1 | 0 | 2 | 1.2 | 0.0 | 2.4 |
| … | … | … | … | … | … | … |
| "    Sponsor Decision" | 2 | 2 | 3 | 2.3 | 2.4 | 3.6 |
| "    Withdrew Consent" | 9 | 10 | 8 | 10.5 | 11.9 | 9.5 |

## 5.7 Step 6: Combine All Results

Now we stack all our individual summaries together to create the complete disposition table.

Using `pl.concat()`, we combine the enrollment counts, completion counts, discontinuation counts, and detailed discontinuation reasons into a single table that flows logically from top to bottom.

```
tbl_disp = pl.concat([
    n_rand,
    n_complete,
    n_disc,
    n_reason
])

tbl_disp
```

| row<br>str | n_0<br>u32 | n_54<br>u32 | n_81<br>u32 | pct_0<br>f64 | pct_54<br>f64 | pct_81<br>f64 |
|---|---|---|---|---|---|---|
| "Participants in population" | 86 | 84 | 84 | null | null | null |
| "Completed" | 58 | 25 | 27 | 67.4 | 29.8 | 32.1 |
| "Discontinued" | 28 | 59 | 57 | 32.6 | 70.2 | 67.9 |
| … | … | … | … | … | … | … |
| "  Sponsor Decision" | 2 | 2 | 3 | 2.3 | 2.4 | 3.6 |
| "  Withdrew Consent" | 9 | 10 | 8 | 10.5 | 11.9 | 9.5 |

## 5.8 Step 7: Generate Publication-Ready Output

Finally, we format our table in RTF format using the `rtflite` package.

The `RTFDocument` class handles the complex formatting required for clinical reports, including proper column headers, borders, and spacing. The resulting RTF file can be directly included in regulatory submissions or converted to PDF for review.

```
doc_disp = rtf.RTFDocument(
    df=tbl_disp.select("row", "n_0", "pct_0", "n_54", "pct_54", "n_81", "pct_81"),
    rtf_title=rtf.RTFTitle(text=["Disposition of Participants"]),
    rtf_column_header=[
        rtf.RTFColumnHeader(
            text=["", "Placebo", "Xanomeline Low Dose", "Xanomeline High Dose"],
            col_rel_width=[3] + [2] * 3,
            text_justification=["l"] + ["c"] * 3,
        ),
        rtf.RTFColumnHeader(
            text=["", "n", "(%)", "n", "(%)", "n", "(%)"],
            col_rel_width=[3] + [1] * 6,
            text_justification=["l"] + ["c"] * 6,
            border_top=[""] + ["single"] * 6,
            border_left=["single"] + ["single", ""] * 3
        )
    ],
    rtf_body=rtf.RTFBody(
        col_rel_width=[3] + [1] * 6,
        text_justification=["l"] + ["c"] * 6,
        border_left=["single"] + ["single", ""] * 3
    ),
    rtf_source=rtf.RTFSource(text=["Source: ADSL dataset"]) # Required source attribution
)

doc_disp.write_rtf("rtf/tlf_disposition.rtf")  # Save as RTF for submission
```

```
rtf/tlf_disposition.rtf
```

```
PosixPath('pdf/tlf_disposition.pdf')
```

# 6 Study population

Objective

Create study population summary tables to document participant counts across different analysis populations. Learn to use population flags in ADSL data and generate regulatory-compliant population tables with rtflite.

## 6.1 Overview

Clinical trials define multiple analysis populations based on different inclusion criteria. Following ICH E3 guidance, regulatory submissions must clearly document the number of participants in each analysis population to support the validity of statistical analyses.

The key analysis populations typically include:

- **All Randomized**: Total participants who entered the study
- **Intent-to-Treat (ITT)**: Participants included in the primary efficacy analysis
- **Efficacy Population**: Participants who meet specific criteria for efficacy evaluation
- **Safety Population**: Participants who received at least one dose of study treatment

This tutorial shows you how to create a population summary table using Python's `rtflite` package.

```
import polars as pl # Data manipulation
import rtflite as rtf # RTF reporting
```

```
polars.config.Config
```

## 6.2 Step 1: Load Data

We start by loading the Subject-level Analysis Dataset (ADSL), which contains population flags for each participant.

```
adsl = pl.read_parquet("data/adsl.parquet")
```

Let's examine the key population flag variables we'll use:

- **USUBJID**: Unique participant identifier
- **TRT01P**: Planned treatment group
- **ITTFL**: Intent-to-treat population flag (Y/N)
- **EFFFL**: Efficacy population flag (Y/N)
- **SAFFL**: Safety population flag (Y/N)

```
adsl.select(["USUBJID", "TRT01P", "ITTFL", "EFFFL", "SAFFL"])
```

| USUBJID<br>str | TRT01P<br>str | ITTFL<br>str | EFFFL<br>str | SAFFL<br>str |
|---|---|---|---|---|
| "01-701-1015" | "Placebo" | "Y" | "Y" | "Y" |
| "01-701-1023" | "Placebo" | "Y" | "Y" | "Y" |
| "01-701-1028" | "Xanomeline High Dose" | "Y" | "Y" | "Y" |
| … | … | … | … | … |
| "01-718-1371" | "Xanomeline High Dose" | "Y" | "Y" | "Y" |
| "01-718-1427" | "Xanomeline High Dose" | "Y" | "Y" | "Y" |

## 6.3 Step 2: Calculate Treatment Group Totals

First, we calculate the total number of randomized participants in each treatment group, which will serve as the denominator for percentage calculations.

```
totals = adsl.group_by("TRT01P").agg(
    total = pl.len()
)

totals
```

| TRT01P | total |
|---|---|
| str | u32 |
| ”Placebo” | 86 |
| ”Xanomeline Low Dose” | 84 |
| ”Xanomeline High Dose” | 84 |

## 6.4 Step 3: Define Helper Function

We create a reusable function to count participants by treatment group for any population subset.

```
def count_by_treatment(data, population_name):
    """Count participants by treatment group and add population label"""
    return data.group_by("TRT01P").agg(
        n = pl.len()
    ).with_columns(
        population = pl.lit(population_name)
    )
```

## 6.5 Step 4: Count Each Population

Now we calculate participant counts for each analysis population.

### 6.5.1 All Randomized Participants

```
pop_all = count_by_treatment(
    data=adsl,
    population_name="Participants in population"
)

pop_all
```

| TRT01P<br>str | n<br>u32 | population<br>str |
|---|---|---|
| ”Xanomeline High Dose” | 84 | ”Participants in population” |
| ”Placebo” | 86 | ”Participants in population” |
| ”Xanomeline Low Dose” | 84 | ”Participants in population” |

### 6.5.2 Intent-to-Treat Population

```
adsl_itt = adsl.filter(pl.col("ITTFL") == "Y")
pop_itt = count_by_treatment(
    data=adsl_itt,
    population_name="Participants included in ITT population"
)

pop_itt
```

| TRT01P<br>str | n<br>u32 | population<br>str |
|---|---|---|
| ”Placebo” | 86 | ”Participants included in ITT p… |
| ”Xanomeline Low Dose” | 84 | ”Participants included in ITT p… |
| ”Xanomeline High Dose” | 84 | ”Participants included in ITT p… |

### 6.5.3 Efficacy Population

```
adsl_eff = adsl.filter(pl.col("EFFFL") == "Y")
pop_eff = count_by_treatment(
```

```
    data=adsl_eff,
    population_name="Participants included in efficacy population"
)

pop_eff
```

| TRT01P<br>str | n<br>u32 | population<br>str |
|---|---|---|
| ”Xanomeline Low Dose” | 81 | ”Participants included in effic… |
| ”Xanomeline High Dose” | 74 | ”Participants included in effic… |
| ”Placebo” | 79 | ”Participants included in effic… |

### 6.5.4 Safety Population

```
adsl_saf = adsl.filter(pl.col("SAFFL") == "Y")
pop_saf = count_by_treatment(
    data=adsl_saf,
    population_name="Participants included in safety population"
)

pop_saf
```

| TRT01P<br>str | n<br>u32 | population<br>str |
|---|---|---|
| ”Xanomeline High Dose” | 84 | ”Participants included in safet… |
| ”Placebo” | 86 | ”Participants included in safet… |
| ”Xanomeline Low Dose” | 84 | ”Participants included in safet… |

## 6.6 Step 5: Combine All Populations

We stack all population counts together into a single dataset.

```
all_populations = pl.concat([
    pop_all,
    pop_itt,
    pop_eff,
    pop_saf
])

all_populations
```

| TRT01P<br>str | n<br>u32 | population<br>str |
|---|---|---|
| ”Xanomeline High Dose” | 84 | ”Participants in population” |
| ”Placebo” | 86 | ”Participants in population” |
| ”Xanomeline Low Dose” | 84 | ”Participants in population” |
| … | … | … |
| ”Placebo” | 86 | ”Participants included in safet… |
| ”Xanomeline Low Dose” | 84 | ”Participants included in safet… |

## 6.7 Step 6: Calculate Percentages

We join with the total counts and calculate what percentage each population represents of the total randomized participants.

```
stats_with_pct = all_populations.join(
    totals,
    on="TRT01P"
).with_columns(
    pct = (100.0 * pl.col("n") / pl.col("total")).round(1)
)

stats_with_pct
```

| TRT01P<br>str | n<br>u32 | population<br>str | total<br>u32 | pct<br>f64 |
|---|---|---|---|---|
| ”Xanomeline High Dose” | 84 | ”Participants in population” | 84 | 100.0 |
| ”Placebo” | 86 | ”Participants in population” | 86 | 100.0 |

| TRT01P<br>str | n<br>u32 | population<br>str | total<br>u32 | pct<br>f64 |
|---|---|---|---|---|
| ”Xanomeline Low Dose” | 84 | ”Participants in population” | 84 | 100.0 |
| … | … | … | … | … |
| ”Placebo” | 86 | ”Participants included in safet… | 86 | 100.0 |
| ”Xanomeline Low Dose” | 84 | ”Participants included in safet… | 84 | 100.0 |

## 6.8 Step 7: Format Display Values

For the final table, we format the display text. The total randomized count shows just “N”, while subset populations show “N (%)”.

```
formatted_stats = stats_with_pct.with_columns(
    display = pl.when(pl.col("population") == "Participants in population")
        .then(pl.col("n").cast(str))
        .otherwise(
            pl.concat_str([
                pl.col("n").cast(str),
                pl.lit(" ("),
                pl.col("pct").round(1).cast(str),
                pl.lit(")")
            ])
        )
)

formatted_stats
```

| TRT01P<br>str | n<br>u32 | population<br>str | total<br>u32 | pct<br>f64 | display<br>str |
|---|---|---|---|---|---|
| ”Xanomeline High Dose” | 84 | ”Participants in population” | 84 | 100.0 | ”84” |
| ”Placebo” | 86 | ”Participants in population” | 86 | 100.0 | ”86” |
| ”Xanomeline Low Dose” | 84 | ”Participants in population” | 84 | 100.0 | ”84” |
| … | … | … | … | … | … |
| ”Placebo” | 86 | ”Participants included in safet… | 86 | 100.0 | ”86 (100.0)” |
| ”Xanomeline Low Dose” | 84 | ”Participants included in safet… | 84 | 100.0 | ”84 (100.0)” |

## 6.9 Step 8: Create Final Table

We reshape the data from long format (rows for each treatment-population combination) to wide format (columns for each treatment group).

```
df_overview = formatted_stats.pivot(
    values="display",
    index="population",
    on="TRT01P",
    maintain_order=True
).select(
    ["population", "Placebo", "Xanomeline Low Dose", "Xanomeline High Dose"]
)

df_overview
```

| population<br>str | Placebo<br>str | Xanomeline Low Dose<br>str | Xanomeline High Dose<br>str |
|---|---|---|---|
| ”Participants in population” | ”86” | ”84” | ”84” |
| ”Participants included in ITT p… | ”86 (100.0)” | ”84 (100.0)” | ”84 (100.0)” |
| ”Participants included in effic… | ”79 (91.9)” | ”81 (96.4)” | ”74 (88.1)” |
| ”Participants included in safet… | ”86 (100.0)” | ”84 (100.0)” | ”84 (100.0)” |

## 6.10 Step 9: Generate Publication-Ready Output

Finally, we format the population table for regulatory submission using the `rtflite` package.

```
doc_overview = rtf.RTFDocument(
    df=df_overview,
    rtf_title=rtf.RTFTitle(
        text=["Analysis Population", "All Participants Randomized"]
    ),
    rtf_column_header=rtf.RTFColumnHeader(
        text=["", "Placebo\nn (%)", "Xanomeline Low Dose\nn (%)", "Xanomeline High Dose\nn (%)"
```

```
        col_rel_width=[4, 2, 2, 2],
        text_justification=["l", "c", "c", "c"],
    ),
    rtf_body=rtf.RTFBody(
        col_rel_width=[4, 2, 2, 2],
        text_justification=["l", "c", "c", "c"],
    ),
    rtf_source=rtf.RTFSource(text=["Source: ADSL dataset"])
)

doc_overview.write_rtf("rtf/tlf_population.rtf")
```

```
rtf/tlf_population.rtf
```

```
PosixPath('pdf/tlf_population.pdf')
```

# 7 Baseline characteristics

Objective

Create baseline characteristics tables to summarize demographic and clinical characteristics of study participants at enrollment. Learn to calculate descriptive statistics by treatment group using Polars and format regulatory-compliant tables with rtflite.

## 7.1 Overview

Baseline characteristics tables summarize the demographic and clinical characteristics of study participants at enrollment. Following ICH E3 guidance, these tables are essential for understanding the study population and assessing comparability between treatment groups.

This tutorial shows you how to create a baseline characteristics table using Python's `rtflite` package.

```
import polars as pl # Data manipulation
import rtflite as rtf # RTF reporting
```

```
polars.config.Config
```

## 7.2 Step 1: Load Data

We start by loading the Subject-level Analysis Dataset (ADSL) and filtering to the safety population.

```
adsl = (
    pl.read_parquet("data/adsl.parquet")
    .select(["USUBJID", "TRT01P", "AGE", "SEX", "RACE"])
)

adsl
```

| USUBJID<br>str | TRT01P<br>str | AGE<br>f64 | SEX<br>str | RACE<br>str |
|---|---|---|---|---|
| ”01-701-1015” | ”Placebo” | 63.0 | ”Female” | ”White” |
| ”01-701-1023” | ”Placebo” | 64.0 | ”Male” | ”White” |
| ”01-701-1028” | ”Xanomeline High Dose” | 71.0 | ”Male” | ”White” |
| … | … | … | … | … |
| ”01-718-1371” | ”Xanomeline High Dose” | 69.0 | ”Female” | ”White” |
| ”01-718-1427” | ”Xanomeline High Dose” | 74.0 | ”Female” | ”Black Or African American” |

## 7.3 Step 2: Calculate Summary Statistics

We’ll create separate functions to handle continuous and categorical variables.

### 7.3.1 Continuous Variables (Age)

For continuous variables, we calculate mean (SD) and median [min, max].

```
def summarize_continuous(df, var):
    """Calculate summary statistics for continuous variables"""
    return df.group_by("TRT01P").agg([
        pl.col(var).mean().round(1).alias("mean"),
        pl.col(var).std().round(2).alias("sd"),
        pl.col(var).median().alias("median"),
        pl.col(var).min().alias("min"),
        pl.col(var).max().alias("max"),
        pl.len().alias("n")
    ])
```

```
age_stats = summarize_continuous(adsl, "AGE")
age_stats
```

| TRT01P<br>str | mean<br>f64 | sd<br>f64 | median<br>f64 | min<br>f64 | max<br>f64 | n<br>u32 |
|---|---|---|---|---|---|---|
| ”Xanomeline High Dose” | 74.4 | 7.89 | 76.0 | 56.0 | 88.0 | 84 |
| ”Xanomeline Low Dose” | 75.7 | 8.29 | 77.5 | 51.0 | 88.0 | 84 |
| ”Placebo” | 75.2 | 8.59 | 76.0 | 52.0 | 89.0 | 86 |

### 7.3.2 Categorical Variables (Sex, Race)

For categorical variables, we calculate counts and percentages.

```
def summarize_categorical(df, var):
    """Calculate counts and percentages for categorical variables"""
    # Get counts by treatment and category
    counts = df.group_by(["TRT01P", var]).len()

    # Get treatment totals for percentage calculations
    totals = df.group_by("TRT01P").len().rename({"len": "total"})

    # Calculate percentages
    result = counts.join(totals, on="TRT01P").with_columns([
        (100.0 * pl.col("len") / pl.col("total")).round(1).alias("pct")
    ])

    return result

sex_stats = summarize_categorical(adsl, "SEX")
sex_stats
```

| TRT01P<br>str | SEX<br>str | len<br>u32 | total<br>u32 | pct<br>f64 |
|---|---|---|---|---|
| ”Xanomeline High Dose” | ”Female” | 40 | 84 | 47.6 |
| ”Placebo” | ”Male” | 33 | 86 | 38.4 |
| ”Xanomeline Low Dose” | ”Female” | 50 | 84 | 59.5 |

| TRT01P<br>str | SEX<br>str | len<br>u32 | total<br>u32 | pct<br>f64 |
|---|---|---|---|---|
| … | … | … | … | … |
| ”Xanomeline High Dose” | ”Male” | 44 | 84 | 52.4 |
| ”Xanomeline Low Dose” | ”Male” | 34 | 84 | 40.5 |

```
race_stats = summarize_categorical(adsl, "RACE")
race_stats
```

| TRT01P<br>str | RACE<br>str | len<br>u32 | total<br>u32 | pct<br>f64 |
|---|---|---|---|---|
| ”Placebo” | ”Black Or African American” | 8 | 86 | 9.3 |
| ”Xanomeline High Dose” | ”American Indian Or Alaska Nati… | 1 | 84 | 1.2 |
| ”Xanomeline Low Dose” | ”Black Or African American” | 6 | 84 | 7.1 |
| … | … | … | … | … |
| ”Xanomeline Low Dose” | ”White” | 78 | 84 | 92.9 |
| ”Xanomeline High Dose” | ”White” | 74 | 84 | 88.1 |

## 7.4 Step 3: Format Results

Now we format the statistics into the standard baseline table format.

### 7.4.1 Format Age Statistics

```
# Format age as "Mean (SD)" and "Median [Min, Max]"
age_formatted = age_stats.with_columns([
    pl.format("{} ({})", pl.col("mean"), pl.col("sd")).alias("mean_sd"),
    pl.format("{} [{}, {}]", pl.col("median"), pl.col("min"), pl.col("max")).alias("median_rang
]).select(["TRT01P", "mean_sd", "median_range"])

age_formatted
```

| TRT01P<br>str | mean_sd<br>str | median_range<br>str |
|---|---|---|
| ”Xanomeline High Dose” | ”74.4 (7.89)” | ”76.0 [56.0, 88.0]” |
| ”Xanomeline Low Dose” | ”75.7 (8.29)” | ”77.5 [51.0, 88.0]” |
| ”Placebo” | ”75.2 (8.59)” | ”76.0 [52.0, 89.0]” |

### 7.4.2 Format Categorical Statistics

```
# Format categorical as "n (%)"
sex_formatted = sex_stats.with_columns(
    pl.format("{} ({}%)", pl.col("len"), pl.col("pct")).alias("n_pct")
).select(["TRT01P", "SEX", "n_pct"])

race_formatted = race_stats.with_columns(
    pl.format("{} ({}%)", pl.col("len"), pl.col("pct")).alias("n_pct")
).select(["TRT01P", "RACE", "n_pct"])

sex_formatted
```

| TRT01P<br>str | SEX<br>str | n_pct<br>str |
|---|---|---|
| ”Xanomeline High Dose” | ”Female” | ”40 (47.6%)” |
| ”Placebo” | ”Male” | ”33 (38.4%)” |
| ”Xanomeline Low Dose” | ”Female” | ”50 (59.5%)” |
| … | … | … |
| ”Xanomeline High Dose” | ”Male” | ”44 (52.4%)” |
| ”Xanomeline Low Dose” | ”Male” | ”34 (40.5%)” |

## 7.5 Step 4: Create Table Structure

We’ll build the table row by row following the standard baseline table format.

```
# Helper function to get value for a treatment group
def get_value(df, treatment):
    """Get value for a specific treatment group or return default"""
    result = df.filter(pl.col("TRT01P") == treatment)
    return result[result.columns[-1]][0] if result.height > 0 else "0 (0.0%)"

# Build the baseline table structure
table_rows = []

# Age section
table_rows.append(["Age (years)", "", "", ""])

# Age Mean (SD) row
age_mean_row = ["  Mean (SD)"] + [
    get_value(age_formatted.select(["TRT01P", "mean_sd"]), trt).replace("0 (0.0%)", "")
    for trt in ["Placebo", "Xanomeline Low Dose", "Xanomeline High Dose"]
]
table_rows.append(age_mean_row)

# Age Median [Min, Max] row
age_median_row = ["  Median [Min, Max]"] + [
    get_value(age_formatted.select(["TRT01P", "median_range"]), trt).replace("0 (0.0%)", "")
    for trt in ["Placebo", "Xanomeline Low Dose", "Xanomeline High Dose"]
]
table_rows.append(age_median_row)

# Sex section
table_rows.append(["Sex", "", "", ""])

for sex_cat in ["Female", "Male"]:
    sex_data = sex_formatted.filter(pl.col("SEX") == sex_cat)
    sex_row = [f"  {sex_cat}"] + [
        get_value(sex_data, trt)
        for trt in ["Placebo", "Xanomeline Low Dose", "Xanomeline High Dose"]
    ]
    table_rows.append(sex_row)

# Race section
table_rows.append(["Race", "", "", ""])
```

```
for race_cat in ["White", "Black Or African American", "American Indian Or Alaska Native"]:
    race_data = race_formatted.filter(pl.col("RACE") == race_cat)
    race_row = [f"  {race_cat}"] + [
        get_value(race_data, trt)
        for trt in ["Placebo", "Xanomeline Low Dose", "Xanomeline High Dose"]
    ]
    table_rows.append(race_row)

# Create DataFrame from table rows
baseline_table = pl.DataFrame(
    table_rows,
    schema=["Characteristic", "Placebo", "Xanomeline Low Dose", "Xanomeline High Dose"],
    orient="row"
)

baseline_table
```

| Characteristic<br>str | Placebo<br>str | Xanomeline Low Dose<br>str | Xanomeline High Dose<br>str |
|---|---|---|---|
| "Age (years)" | "" | "" | "" |
| "  Mean (SD)" | "75.2 (8.59)" | "75.7 (8.29)" | "74.4 (7.89)" |
| "  Median [Min, Max]" | "76.0 [52.0, 89.0]" | "77.5 [51.0, 88.0]" | "76.0 [56.0, 88.0]" |
| … | … | … | … |
| "  Black Or African American" | "8 (9.3%)" | "6 (7.1%)" | "9 (10.7%)" |
| "  American Indian Or Alaska Na… | "0 (0.0%)" | "0 (0.0%)" | "1 (1.2%)" |

## 7.6 Step 5: Generate Publication-Ready Output

Finally, we format the baseline table for regulatory submission using the `rtflite` package.

```
# Get treatment group sizes for column headers
treatment_n = adsl.group_by("TRT01P").len().sort("TRT01P")
n_placebo = treatment_n.filter(pl.col("TRT01P") == "Placebo")["len"][0]
n_low = treatment_n.filter(pl.col("TRT01P") == "Xanomeline Low Dose")["len"][0]
n_high = treatment_n.filter(pl.col("TRT01P") == "Xanomeline High Dose")["len"][0]
```

```
doc_baseline = rtf.RTFDocument(
    df=baseline_table,
    rtf_title=rtf.RTFTitle(
        text=[
            "Baseline Characteristics of Participants",
            "(All Participants Randomized)"
        ]
    ),
    rtf_column_header=rtf.RTFColumnHeader(
        text=[
            "Characteristic",
            f"Placebo\n(N={n_placebo})",
            f"Xanomeline Low Dose\n(N={n_low})",
            f"Xanomeline High Dose\n(N={n_high})"
        ],
        text_justification=["l", "c", "c", "c"],
        col_rel_width=[3, 2, 2, 2]
    ),
    rtf_body=rtf.RTFBody(
        text_justification=["l", "c", "c", "c"],
        col_rel_width=[3, 2, 2, 2]
    ),
    rtf_source=rtf.RTFSource(text=["Source: ADSL dataset"])
)

doc_baseline.write_rtf("rtf/tlf_baseline.rtf") # Save as RTF for submission
```

```
rtf/tlf_baseline.rtf
```

```
PosixPath('pdf/tlf_baseline.pdf')
```

# 8 Adverse events summary

Objective

Create adverse event summary tables to provide high-level safety overview across treatment groups. Learn to calculate AE rates and percentages using Polars and create comprehensive safety summary tables with rtflite.

## 8.1 Overview

Adverse events (AE) summary tables are critical safety assessments required in clinical study reports. Following ICH E3 guidance, these tables summarize the overall safety profile by showing the number and percentage of participants experiencing various categories of adverse events across treatment groups.

Key categories typically include:

- **Any adverse event**: Total participants with at least one AE
- **Drug-related events**: Events potentially related to study treatment
- **Serious adverse events**: Events meeting regulatory criteria for seriousness
- **Deaths**: Fatal outcomes
- **Discontinuations**: Participants who stopped treatment due to AEs

This tutorial shows you how to create an AE summary table using Python's `rtflite` package.

```
import polars as pl
import rtflite as rtf
```

```
polars.config.Config
```

## 8.2 Step 1: Load Data

We need two datasets for AE analysis: the subject-level dataset (ADSL) and the adverse events dataset (ADAE).

```
# Load datasets
adsl = pl.read_parquet("data/adsl.parquet")
adae = pl.read_parquet("data/adae.parquet")

# Display key variables from ADSL
adsl.select(["USUBJID", "TRT01A", "SAFFL"])
```

| USUBJID<br>str | TRT01A<br>str | SAFFL<br>str |
|---|---|---|
| ”01-701-1015” | ”Placebo” | ”Y” |
| ”01-701-1023” | ”Placebo” | ”Y” |
| ”01-701-1028” | ”Xanomeline High Dose” | ”Y” |
| … | … | … |
| ”01-718-1371” | ”Xanomeline High Dose” | ”Y” |
| ”01-718-1427” | ”Xanomeline High Dose” | ”Y” |

```
# Display key variables from ADAE
adae.select(["USUBJID", "AEREL", "AESER", "AEOUT", "AEACN"])
```

| USUBJID<br>str | AEREL<br>str | AESER<br>str | AEOUT<br>str | AEACN<br>str |
|---|---|---|---|---|
| ”01-701-1015” | ”PROBABLE” | ”N” | ”NOT RECOVERED/NOT RESOLVED” | ”” |
| ”01-701-1015” | ”PROBABLE” | ”N” | ”NOT RECOVERED/NOT RESOLVED” | ”” |
| ”01-701-1015” | ”REMOTE” | ”N” | ”RECOVERED/RESOLVED” | ”” |
| … | … | … | … | … |

| USUBJID<br>str | AEREL<br>str | AESER<br>str | AEOUT<br>str | AEACN<br>str |
|---|---|---|---|---|
| ”01-718-1427” | ”POSSIBLE” | ”N” | ”RECOVERED/RESOLVED” | ”” |
| ”01-718-1427” | ”POSSIBLE” | ”N” | ”RECOVERED/RESOLVED” | ”” |

Key ADAE variables used in this analysis:

- **USUBJID**: Unique subject identifier to link with ADSL
- **AEREL**: Relationship of adverse event to study drug (e.g., “RELATED”, “POSSIBLE”, “PROBABLE”, “DEFINITE”, “NOT RELATED”)
- **AESER**: Serious adverse event flag (“Y” = serious, “N” = not serious)
- **AEOUT**: Outcome of adverse event (e.g., “RECOVERED”, “RECOVERING”, “NOT RECOVERED”, “FATAL”)
- **AEACN**: Action taken with study treatment (e.g., “DOSE NOT CHANGED”, “DRUG WITHDRAWN”, “DOSE REDUCED”)

## 8.3 Step 2: Filter Safety Population

For safety analyses, we focus on participants who received at least one dose of study treatment.

```
# Filter to safety population
adsl_safety = adsl.filter(pl.col("SAFFL") == "Y").select(["USUBJID", "TRT01A"])

# Get treatment counts for denominators
pop_counts = adsl_safety.group_by("TRT01A").agg(
    N = pl.len()
).sort("TRT01A")

# Preserve the treatment level order for downstream joins
treatment_levels = pop_counts.select(["TRT01A"])

# Safety population by treatment
pop_counts
```

| TRT01A | N |
|---|---|
| str | u32 |
| ”Placebo” | 86 |
| ”Xanomeline High Dose” | 84 |
| ”Xanomeline Low Dose” | 84 |

```
# Join treatment information to AE data
adae_safety = adae.join(adsl_safety, on="USUBJID")

# Total AE records in safety population
adae_safety.height
```

```
1191
```

## 8.4 Step 3: Define AE Categories

We’ll calculate participant counts for standard AE categories used in regulatory submissions.

```
def count_participants(df, condition=None):
    """
    Count unique participants meeting a condition

    Args:
        df: DataFrame with adverse events
        condition: polars expression for filtering (None = count all)

    Returns:
        DataFrame with counts by treatment
    """
    if condition is not None:
        df = df.filter(condition)

    counts = df.group_by("TRT01A").agg(
        n = pl.col("USUBJID").n_unique()
    )
```

```
    return treatment_levels.join(counts, on="TRT01A", how="left").with_columns(
        pl.col("n").fill_null(0)
    )

# Calculate each category
categories = []

# 1. Participants in population (no filtering)
pop_row = pop_counts.with_columns(
    category = pl.lit("Participants in population")
).rename({"N": "n"})
categories.append(pop_row)

# 2. With any adverse event
any_ae = count_participants(adae_safety).with_columns(
    category = pl.lit("With any adverse event")
)
categories.append(any_ae)
```

```
# 3. With drug-related adverse event
drug_related = count_participants(
    adae_safety,
    pl.col("AEREL").is_in(["POSSIBLE", "PROBABLE", "DEFINITE", "RELATED"])
).with_columns(
    category = pl.lit("With drug-related adverse event")
)
categories.append(drug_related)

# 4. With serious adverse event
serious = count_participants(
    adae_safety,
    pl.col("AESER") == "Y"
).with_columns(
    category = pl.lit("With serious adverse event")
)
categories.append(serious)
```

```
# 5. With serious drug-related adverse event
serious_drug_related = count_participants(
    adae_safety,
```

```
    (pl.col("AESER") == "Y") &
    pl.col("AEREL").is_in(["POSSIBLE", "PROBABLE", "DEFINITE", "RELATED"])
).with_columns(
    category = pl.lit("With serious drug-related adverse event")
)
categories.append(serious_drug_related)

# 6. Who died
deaths = count_participants(
    adae_safety,
    pl.col("AEOUT") == "FATAL"
).with_columns(
    category = pl.lit("Who died")
)
categories.append(deaths)

# 7. Discontinued due to adverse event
discontinued = count_participants(
    adae_safety,
    pl.col("AEACN") == "DRUG WITHDRAWN"
).with_columns(
    category = pl.lit("Discontinued due to adverse event")
)
categories.append(discontinued)
```

## 8.5 Step 4: Combine and Calculate Percentages

Now we combine all categories and calculate percentages based on the safety population.

```
# Combine all categories
ae_summary = pl.concat(categories, how="diagonal")

# Add population totals and calculate percentages
ae_summary = ae_summary.join(
    pop_counts.select(["TRT01A", "N"]),
    on="TRT01A",
```

```
    how="left"
).with_columns([
    # Fill missing counts with 0
    pl.col("n").fill_null(0),
    # Calculate percentage
    pl.when(pl.col("category") == "Participants in population")
        .then(None)  # No percentage for population row
        .otherwise((100.0 * pl.col("n") / pl.col("N")).round(1))
        .alias("pct")
])

ae_summary.sort(["category", "TRT01A"])
```

| TRT01A<br>str | n<br>u32 | category<br>str | N<br>u32 | pct<br>f64 |
|---|---|---|---|---|
| ”Placebo” | 0 | ”Discontinued due to adverse ev… | 86 | 0.0 |
| ”Xanomeline High Dose” | 0 | ”Discontinued due to adverse ev… | 84 | 0.0 |
| ”Xanomeline Low Dose” | 0 | ”Discontinued due to adverse ev… | 84 | 0.0 |
| … | … | … | … | … |
| ”Xanomeline High Dose” | 1 | ”With serious drug-related adve… | 84 | 1.2 |
| ”Xanomeline Low Dose” | 1 | ”With serious drug-related adve… | 84 | 1.2 |

## 8.6 Step 5: Format for Display

We’ll format the counts and percentages for the final table display.

```
# Format display values
ae_formatted = ae_summary.with_columns([
    # Show counts as strings, including zeros
    pl.col("n").cast(str).alias("n_display"),
    # Format percentages with parentheses; blank out population row
    pl.when(pl.col("category") == "Participants in population")
      .then(pl.lit(""))
      .otherwise(
          pl.format("({})", pl.col("pct").fill_null(0).round(1).cast(str))
      )
```

```
      .alias("pct_display")
])

ae_formatted.select(["category", "TRT01A", "n_display", "pct_display"])
```

| category<br>str | TRT01A<br>str | n_display<br>str | pct_display<br>str |
|---|---|---|---|
| ”Participants in population” | ”Placebo” | ”86” | ”” |
| ”Participants in population” | ”Xanomeline High Dose” | ”84” | ”” |
| ”Participants in population” | ”Xanomeline Low Dose” | ”84” | ”” |
| … | … | … | … |
| ”Discontinued due to adverse ev… | ”Xanomeline High Dose” | ”0” | ”(0.0)” |
| ”Discontinued due to adverse ev… | ”Xanomeline Low Dose” | ”0” | ”(0.0)” |

## 8.7 Step 6: Create Final Table Structure

We reshape the data to create the final table with treatments as columns.

```
# Define category order for consistent display
category_order = [
    "Participants in population",
    "With any adverse event",
    "With drug-related adverse event",
    "With serious adverse event",
    "With serious drug-related adverse event",
    "Who died",
    "Discontinued due to adverse event"
]

# Pivot to wide format
ae_wide = ae_formatted.pivot(
    values=["n_display", "pct_display"],
    index="category",
    on="TRT01A",
    maintain_order=True
)
```

```
# Reorder columns for each treatment group
treatments = ["Placebo", "Xanomeline Low Dose", "Xanomeline High Dose"]
column_order = ["category"]
for trt in treatments:
    column_order.extend([f"n_display_{trt}", f"pct_display_{trt}"])

# Create final table with proper column order
final_table = ae_wide.select(column_order).sort(
    pl.col("category").cast(pl.Enum(category_order))
)

final_table
```

| category<br>str | n_display_Placebo<br>str | pct_display_Placebo<br>str | n_display_Xanomeline Low D<br>str |
|---|---|---|---|
| ”Participants in population” | ”86” | ”” | ”84” |
| ”With any adverse event” | ”69” | ”(80.2)” | ”77” |
| ”With drug-related adverse even… | ”44” | ”(51.2)” | ”73” |
| … | … | … | … |
| ”Who died” | ”2” | ”(2.3)” | ”1” |
| ”Discontinued due to adverse ev… | ”0” | ”(0.0)” | ”0” |

## 8.8 Step 7: Generate Publication-Ready Output

Finally, we format the AE summary table for regulatory submission using the `rtflite` package.

```
# Get population sizes for column headers
n_placebo = pop_counts.filter(pl.col("TRT01A") == "Placebo")["N"][0]
n_low = pop_counts.filter(pl.col("TRT01A") == "Xanomeline Low Dose")["N"][0]
n_high = pop_counts.filter(pl.col("TRT01A") == "Xanomeline High Dose")["N"][0]

doc_ae_summary = rtf.RTFDocument(
    df=final_table.rename({"category": ""}),
    rtf_title=rtf.RTFTitle(
```

```
        text=[
            "Analysis of Adverse Event Summary",
            "(Safety Analysis Population)"
        ]
    ),
    rtf_column_header=[
        rtf.RTFColumnHeader(
            text = [
                "",
                "Placebo",
                "Xanomeline Low Dose",
                "Xanomeline High Dose"
            ],
            col_rel_width=[4, 2, 2, 2],
            text_justification=["l", "c", "c", "c"],
        ),
        rtf.RTFColumnHeader(
            text=[
                "",          # Empty for first column
                "n", "(%)",  # Placebo columns
                "n", "(%)",  # Low Dose columns
                "n", "(%)"   # High Dose columns
            ],
            col_rel_width=[4] + [1] * 6,
            text_justification=["l"] + ["c"] * 6,
            border_left = ["single"] + ["single", ""] * 3,
            border_top = [""] + ["single"] * 6
        )
    ],
    rtf_body=rtf.RTFBody(
        col_rel_width=[4] + [1] * 6,
        text_justification=["l"] + ["c"] * 6,
        border_left = ["single"] + ["single", ""] * 3
    ),
    rtf_footnote=rtf.RTFFootnote(
        text=[
            "Every subject is counted a single time for each applicable row and column."
        ]
    ),
    rtf_source=rtf.RTFSource(
```

```
        text=["Source: ADSL and ADAE datasets"]
    )
)

doc_ae_summary.write_rtf("rtf/tlf_ae_summary.rtf")
```

```
rtf/tlf_ae_summary.rtf
```

```
PosixPath('pdf/tlf_ae_summary.pdf')
```

# 9 Specific adverse events

Objective

Create detailed adverse event tables organized by System Organ Class and Preferred Term to support safety evaluation. Learn to process ADAE data with hierarchical grouping using Polars and generate regulatory-compliant AE listings with rtflite.

## 9.1 Overview

Specific adverse events tables provide detailed safety information organized by System Organ Class (SOC) and Preferred Term (PT) following the Medical Dictionary for Regulatory Activities (MedDRA) hierarchy. Following ICH E3 guidance, these tables are essential components of clinical study reports that present participant-level adverse event data across treatment groups.

Key features of specific AE tables include:

- **Hierarchical structure**: SOC categories with nested specific AE terms
- **Participant counts**: Number of participants experiencing each AE type
- **Treatment comparison**: Side-by-side counts across treatment groups
- **MedDRA compliance**: Standardized medical terminology for regulatory submissions

This tutorial demonstrates how to create a regulatory-compliant specific adverse events table using Python's `rtflite` package.

## 9.2 Setup

```
import polars as pl
import rtflite as rtf
```

```
polars.config.Config
```

```
adsl = pl.read_parquet("data/adsl.parquet")
adae = pl.read_parquet("data/adae.parquet")
treatments = ["Placebo", "Xanomeline Low Dose", "Xanomeline High Dose"]
```

## 9.3 Step 1: Load and Explore Data

We start by examining the adverse events data structure and understanding the MedDRA hierarchy.

```
# Display key variables in ADAE dataset
adae_vars = adae.select(["USUBJID", "TRTA", "AEBODSYS", "AEDECOD", "AESEV", "AESER"])
# Key ADAE variables
adae_vars
```

| USUBJID<br>str | TRTA<br>str | AEBODSYS<br>str | AEDECOD<br>str |
|---|---|---|---|
| ”01-701-1015” | ”Placebo” | ”GENERAL DISORDERS AND ADMINIST… | ”APPLICATION SI |
| ”01-701-1015” | ”Placebo” | ”GENERAL DISORDERS AND ADMINIST… | ”APPLICATION SI |
| ”01-701-1015” | ”Placebo” | ”GASTROINTESTINAL DISORDERS” | ”DIARRHOEA” |
| … | … | … | … |
| ”01-718-1427” | ”Xanomeline High Dose” | ”METABOLISM AND NUTRITION DISOR… | ”DECREASED AP |
| ”01-718-1427” | ”Xanomeline High Dose” | ”GASTROINTESTINAL DISORDERS” | ”NAUSEA” |

```
# Examine the MedDRA hierarchy structure
# System Organ Classes (SOCs) in the data
soc_summary = adae.group_by("AEBODSYS").agg(
    n_participants=pl.col("USUBJID").n_unique(),
    n_events=pl.len()
```

```
).sort("n_participants", descending=True)
soc_summary
```

| AEBODSYS<br>str | n_participants<br>u32 | n_events<br>u32 |
|---|---|---|
| ”GENERAL DISORDERS AND ADMINIST… | 108 | 292 |
| ”SKIN AND SUBCUTANEOUS TISSUE D… | 105 | 276 |
| ”NERVOUS SYSTEM DISORDERS” | 59 | 101 |
| … | … | … |
| ”HEPATOBILIARY DISORDERS” | 1 | 1 |
| ”SOCIAL CIRCUMSTANCES” | 1 | 1 |

## 9.4 Step 2: Prepare Analysis Population

Following regulatory standards, we focus on the safety analysis population.

```
# Define safety population
adsl_safety = adsl.filter(pl.col("SAFFL") == "Y").select(["USUBJID", "TRT01A"])
# Safety population size
adsl_safety.height

# Get safety population counts by treatment
pop_counts = adsl_safety.group_by("TRT01A").agg(N=pl.len()).sort("TRT01A")
# Safety population by treatment
pop_counts
```

| TRT01A<br>str | N<br>u32 |
|---|---|
| ”Placebo” | 86 |
| ”Xanomeline High Dose” | 84 |
| ”Xanomeline Low Dose” | 84 |

```
# Filter adverse events to safety population
adae_safety = adae.join(adsl_safety, on="USUBJID", how="inner")
# AE records in safety population
adae_safety.height
```

```
1191
```

## 9.5 Step 3: Data Preparation and Standardization

We standardize the adverse event terms and prepare the hierarchical data structure.

```
# Standardize AE term formatting for consistency
ae_counts = (
    adae_safety
    .with_columns([
        pl.col("AEDECOD").str.to_titlecase().alias("AEDECOD_STD"),
        pl.col("AEBODSYS").str.to_titlecase().alias("AEBODSYS_STD")
    ])
    .group_by(["TRT01A", "AEBODSYS_STD", "AEDECOD_STD"])
    .agg(n=pl.col("USUBJID").n_unique())
    .sort(["AEBODSYS_STD", "AEDECOD_STD", "TRT01A"])
)

# Sample of prepared AE counts
ae_counts
```

| TRT01A<br>str | AEBODSYS_STD<br>str | AEDECOD_STD<br>str | n<br>u32 |
|---|---|---|---|
| ”Placebo” | ”Cardiac Disorders” | ”Atrial Fibrillation” | 1 |
| ”Xanomeline High Dose” | ”Cardiac Disorders” | ”Atrial Fibrillation” | 3 |
| ”Xanomeline Low Dose” | ”Cardiac Disorders” | ”Atrial Fibrillation” | 1 |
| … | … | … | … |
| ”Placebo” | ”Vascular Disorders” | ”Orthostatic Hypotension” | 1 |
| ”Xanomeline High Dose” | ”Vascular Disorders” | ”Wound Haemorrhage” | 1 |

## 9.6 Step 4: Build Hierarchical Table Structure

We create a nested table structure with SOC headers and indented specific terms.

```
# Initialize table with population counts
table_data = [
    ["Participants in population"] + [
        str(pop_counts.filter(pl.col("TRT01A") == t)["N"][0])
        for t in treatments
    ],
    [""] * 4  # Blank separator row
]

# Build hierarchical structure: SOC -> Specific AE terms
for soc in ae_counts["AEBODSYS_STD"].unique().sort():
    # Add SOC header row (bold formatting will be applied later)
    table_data.append([soc] + [""] * 3)

    # Get all AE terms within this SOC
    soc_data = ae_counts.filter(pl.col("AEBODSYS_STD") == soc)

    # Add each specific AE term with counts
    for ae_term in soc_data["AEDECOD_STD"].unique().sort():
        row = [f"  {ae_term}"]  # Indent specific terms

        # Add counts for each treatment group
        for trt in treatments:
            count_data = soc_data.filter(
                (pl.col("AEDECOD_STD") == ae_term) &
                (pl.col("TRT01A") == trt)
            )
            count = count_data["n"][0] if count_data.height > 0 else 0
            row.append(str(count))

        table_data.append(row)

# Convert to Polars DataFrame
df_ae_specific = pl.DataFrame(
```

```
    table_data,
    schema=[""] + treatments,
    orient="row"
)

# Final table structure
df_ae_specific.shape
df_ae_specific
```

| column_0<br>str | Placebo<br>str | Xanomeline Low Dose<br>str | Xanomeline High Dose<br>str |
|---|---|---|---|
| ”Participants in population” | ”86” | ”84” | ”84” |
| ”” | ”” | ”” | ”” |
| ”Cardiac Disorders” | ”” | ”” | ”” |
| … | … | … | … |
| ”  Orthostatic Hypotension” | ”1” | ”0” | ”0” |
| ”  Wound Haemorrhage” | ”0” | ”0” | ”1” |

## 9.7 Step 5: Create Regulatory-Compliant RTF Output

We format the table following regulatory submission standards with proper hierarchy and formatting.

```
# Create comprehensive RTF document
doc_ae_specific = rtf.RTFDocument(
    df=df_ae_specific,
    rtf_title=rtf.RTFTitle(
        text=[
            "Adverse Events by System Organ Class and Preferred Term",
            "(Safety Analysis Set)"
        ]
    ),
    rtf_column_header=rtf.RTFColumnHeader(
        text=[
            "System Organ Class\\line   Preferred Term",
            f"Placebo\\line (N={pop_counts.filter(pl.col('TRT01A') == 'Placebo')['N'][0]})",
```

```
            f"Xanomeline Low Dose\\line (N={pop_counts.filter(pl.col('TRT01A') == 'Xanomeline L
            f"Xanomeline High Dose\\line (N={pop_counts.filter(pl.col('TRT01A') == 'Xanomeline
        ],
        col_rel_width=[4, 1.5, 1.5, 1.5],
        text_justification=["l", "c", "c", "c"],
        text_format="b",  # Bold headers
        border_bottom="single"
    ),
    rtf_body=rtf.RTFBody(
        col_rel_width=[4, 1.5, 1.5, 1.5],
        text_justification=["l", "c", "c", "c"],
        # Apply bold formatting to SOC headers (rows without indentation)
        text_font_style=lambda df, i, j: "b" if j == 0 and not str(df[i, j]).startswith("  ") a
    ),
    rtf_footnote=rtf.RTFFootnote(
        text=[
            "MedDRA version 25.0.",
            "Each participant is counted once within each preferred term and system organ class
            "Participants with multiple events in the same preferred term are counted only once
        ]
    ),
    rtf_source=rtf.RTFSource(
        text=["Source: ADAE Analysis Dataset (Data cutoff: 01JAN2023)"]
    )
)

# Generate RTF file
doc_ae_specific.write_rtf("rtf/tlf_ae_specific.rtf")
# RTF file created: rtf/tlf_ae_specific.rtf
```

```
rtf/tlf_ae_specific.rtf
```

```
PosixPath('pdf/tlf_ae_specific.pdf')
```

# 10 ANCOVA efficacy analysis

Objective

Create ANCOVA efficacy analysis tables to evaluate treatment effects while controlling for baseline covariates. Learn to perform ANCOVA modeling with missing data imputation using Python statistical libraries and present results in regulatory format with rtflite.

## 10.1 Overview

Analysis of Covariance (ANCOVA) is the primary statistical method for efficacy evaluation in clinical trials. Following ICH E9 guidance on statistical principles for clinical trials, ANCOVA provides a robust framework for comparing treatment effects while controlling for baseline covariates.

Key features of ANCOVA efficacy analysis include:

- **Covariate adjustment**: Controls for baseline values to reduce variability
- **Least squares means**: Provides treatment effect estimates adjusted for covariates
- **Missing data handling**: Implements appropriate imputation strategies (e.g., LOCF, MMRM)
- **Pairwise comparisons**: Tests specific treatment contrasts with confidence intervals
- **Regulatory compliance**: Follows statistical analysis plan specifications

This tutorial demonstrates how to create a comprehensive ANCOVA efficacy table for glucose change from baseline us-

ing Python's statistical libraries and `rtflite` for regulatory-compliant formatting.

## 10.2 Setup

```
import polars as pl
import rtflite as rtf
import pandas as pd
import numpy as np
import statsmodels.formula.api as smf
from scipy import stats as scipy_stats
```

```
polars.config.Config
```

```
adsl = pl.read_parquet("data/adsl.parquet")
adlbc = pl.read_parquet("data/adlbc.parquet")
treatments = ["Placebo", "Xanomeline Low Dose", "Xanomeline High Dose"]
```

## 10.3 Step 1: Explore Laboratory Data Structure

We start by understanding the glucose data structure and endpoint definitions.

```
# Display key laboratory variables
# Key ADLBC variables for efficacy analysis
lab_vars = adlbc.select(["USUBJID", "PARAMCD", "PARAM", "AVISIT", "AVISITN", "AVAL", "BASE", "
lab_vars
```

| USUBJID<br>str | PARAMCD<br>str | PARAM<br>str | AVISIT<br>str | AVISITN<br>f64 | AVAL<br>f64 | BA<br>f6 |
|---|---|---|---|---|---|---|
| ”01-701-1015” | ”SODIUM” | ”Sodium (mmol/L)” | ” Baseline” | 0.0 | 140.0 | 14 |
| ”01-701-1015” | ”K” | ”Potassium (mmol/L)” | ” Baseline” | 0.0 | 4.5 | 4.5 |
| ”01-701-1015” | ”CL” | ”Chloride (mmol/L)” | ” Baseline” | 0.0 | 106.0 | 10 |

| USUBJID<br>str | PARAMCD<br>str | PARAM<br>str | AVISIT<br>str | AVISITN<br>f64 | AVAL<br>f64 | BA<br>f6 |
|---|---|---|---|---|---|---|
| … | … | … | … | … | … | … |
| ”01-718-1427” | ”_CK” | ”Creatine Kinase (U/L) change f… | ” Week 8” | 8.0 | -0.5 | nu |
| ”01-718-1427” | ”_CK” | ”Creatine Kinase (U/L) change f… | ”End of Treatment” | 99.0 | -0.5 | nu |

```
# Focus on glucose parameter
gluc_visits = adlbc.filter(pl.col("PARAMCD") == "GLUC").select("AVISIT", "AVISITN").unique().so
# Available glucose measurement visits
gluc_visits
```

| AVISIT<br>str | AVISITN<br>f64 |
|---|---|
| ” .” | null |
| ” Baseline” | 0.0 |
| ” Week 2” | 2.0 |
| … | … |
| ” Week 26” | 26.0 |
| ”End of Treatment” | 99.0 |

## 10.4 Step 2: Define Analysis Population and Endpoint

Following the Statistical Analysis Plan, we focus on the efficacy population for the primary endpoint.

```
# Clean data types and prepare datasets
adlbc_clean = adlbc.with_columns([
    pl.col(c).cast(str).str.strip_chars()
    for c in ["USUBJID", "PARAMCD", "AVISIT", "TRTP"]
])

# Define efficacy population
adsl_eff = adsl.filter(pl.col("EFFFL") == "Y").select(["USUBJID"])
# Efficacy population size
adsl_eff.height
```

```
# Filter laboratory data to efficacy population
adlbc_eff = adlbc_clean.join(adsl_eff, on="USUBJID", how="inner")
# Laboratory records in efficacy population
adlbc_eff.height
```

71882

```
# Examine glucose data availability by visit and treatment
gluc_availability = (
    adlbc_eff.filter(pl.col("PARAMCD") == "GLUC")
    .group_by(["TRTP", "AVISIT"])
    .agg(n_subjects=pl.col("USUBJID").n_unique())
    .sort(["TRTP", "AVISIT"])
)
# Glucose data availability by visit
gluc_availability
```

| TRTP<br>str | AVISIT<br>str | n_subjects<br>u32 |
|---|---|---|
| ”Placebo” | ”.” | 13 |
| ”Placebo” | ”Baseline” | 79 |
| ”Placebo” | ”End of Treatment” | 79 |
| … | … | … |
| ”Xanomeline Low Dose” | ”Week 6” | 62 |
| ”Xanomeline Low Dose” | ”Week 8” | 59 |

## 10.5 Step 3: Implement LOCF Imputation Strategy

We apply Last Observation Carried Forward (LOCF) for missing Week 24 glucose values.

```
# Prepare glucose data with LOCF for Week 24 endpoint
gluc_data = (
    adlbc_eff
    .filter((pl.col("PARAMCD") == "GLUC") & (pl.col("AVISITN") <= 24))
```

```
    .sort(["USUBJID", "AVISITN"])
    .group_by("USUBJID")
    .agg([
        pl.col("TRTP").first(),
        pl.col("BASE").first(),
        pl.col("AVAL").filter(pl.col("AVISITN") == 0).first().alias("Baseline"),
        pl.col("AVAL").last().alias("Week_24_LOCF"),  # LOCF: last available value <= Week 24
        pl.col("AVISITN").max().alias("Last_Visit")   # Track actual last visit
    ])
    .filter(pl.col("Baseline").is_not_null() & pl.col("Week_24_LOCF").is_not_null())
    .with_columns((pl.col("Week_24_LOCF") - pl.col("Baseline")).alias("CHG"))
)

# Subjects with baseline and Week 24 (LOCF) glucose
gluc_data.height
# Sample of prepared analysis data
gluc_data
```

| USUBJID<br>str | TRTP<br>str | BASE<br>f64 | Baseline<br>f64 | Week_24_LOCF<br>f64 | Last_Visit<br>f64 | CHG<br>f64 |
|---|---|---|---|---|---|---|
| ”01-701-1015” | ”Placebo” | 4.71835 | 4.71835 | 4.49631 | 24.0 | -0.22204 |
| ”01-701-1023” | ”Placebo” | 5.32896 | 5.32896 | 5.43998 | 4.0 | 0.11102 |
| ”01-701-1028” | ”Xanomeline High Dose” | 4.77386 | 4.77386 | 5.43998 | 24.0 | 0.66612 |
| … | … | … | … | … | … | … |
| ”01-718-1371” | ”Xanomeline High Dose” | 6.27263 | 6.27263 | 5.10692 | 12.0 | -1.16571 |
| ”01-718-1427” | ”Xanomeline High Dose” | 4.55182 | 4.55182 | 3.71917 | 8.0 | -0.83265 |

```
# Assess LOCF imputation impact
locf_summary = (
    gluc_data
    .group_by(["TRTP", "Last_Visit"])
    .agg(n_subjects=pl.len())
    .sort(["TRTP", "Last_Visit"])
)
# LOCF imputation summary (last actual visit used)
locf_summary
```

| TRTP<br>str | Last_Visit<br>f64 | n_subjects<br>u32 |
|---|---|---|
| "Placebo" | 2.0 | 2 |
| "Placebo" | 4.0 | 2 |
| "Placebo" | 6.0 | 2 |
| … | … | … |
| "Xanomeline Low Dose" | 20.0 | 5 |
| "Xanomeline Low Dose" | 24.0 | 25 |

## 10.6 Step 4: Calculate Descriptive Statistics

We compute baseline, Week 24, and change from baseline statistics by treatment group.

```
# Calculate comprehensive descriptive statistics
desc_stats = []
for trt in treatments:
    # Analysis data for this treatment
    trt_data = gluc_data.filter(pl.col("TRTP") == trt)

    # Original baseline data (all subjects with baseline)
    baseline_full = adlbc_eff.filter(
        (pl.col("PARAMCD") == "GLUC") &
        (pl.col("AVISIT") == "Baseline") &
        (pl.col("TRTP") == trt)
    )

    desc_stats.append({
        "Treatment": trt,
        "N_Baseline": baseline_full.height,
        "Baseline_Mean": baseline_full["AVAL"].mean() if baseline_full.height > 0 else np.nan,
        "Baseline_SD": baseline_full["AVAL"].std() if baseline_full.height > 0 else np.nan,
        "N_Week24": trt_data.height,
        "Week24_Mean": trt_data["Week_24_LOCF"].mean() if trt_data.height > 0 else np.nan,
        "Week24_SD": trt_data["Week_24_LOCF"].std() if trt_data.height > 0 else np.nan,
        "N_Change": trt_data.height,
        "Change_Mean": trt_data["CHG"].mean() if trt_data.height > 0 else np.nan,
        "Change_SD": trt_data["CHG"].std() if trt_data.height > 0 else np.nan
```

```
    })

# Display descriptive statistics
desc_df = pl.DataFrame(desc_stats)
# Descriptive statistics by treatment
desc_df
```

| Treatment<br>str | N_Baseline<br>i64 | Baseline_Mean<br>f64 | Baseline_SD<br>f64 | N_Week24<br>i64 | Week24_Mean<br>f64 | Week24 |
|---|---|---|---|---|---|---|
| ”Placebo” | 79 | 5.656399 | 2.229324 | 79 | 5.639535 | 1.65180 |
| ”Xanomeline Low Dose” | 79 | 5.419603 | 0.946102 | 79 | 5.352148 | 1.05800 |
| ”Xanomeline High Dose” | 74 | 5.388971 | 1.374893 | 74 | 5.831551 | 2.21466 |

## 10.7 Step 5: Perform ANCOVA Analysis

We fit the ANCOVA model with treatment and baseline glucose as covariates.

```
# Convert to pandas for statsmodels compatibility
ancova_df = gluc_data.to_pandas()
ancova_df["TRTP"] = pd.Categorical(ancova_df["TRTP"], categories=treatments)

# Fit ANCOVA model: Change = Treatment + Baseline
model = smf.ols("CHG ~ TRTP + BASE", data=ancova_df).fit()

# Display model summary
# ANCOVA Model Summary
model.rsquared
model.fvalue, model.f_pvalue
# Model coefficients
model.summary().tables[1]
```

| | coef | std err | t | P> \|t\| | [0.025 | 0.975] |
|---|---|---|---|---|---|---|
| **Intercept** | 2.9972 | 0.392 | 7.642 | 0.000 | 2.224 | 3.770 |
| **TRTP[T.Xanomeline Low Dose]** | -0.1768 | 0.243 | -0.729 | 0.467 | -0.655 | 0.301 |
| **TRTP[T.Xanomeline High Dose]** | 0.3169 | 0.247 | 1.284 | 0.200 | -0.169 | 0.803 |
| **BASE** | -0.5329 | 0.062 | -8.543 | 0.000 | -0.656 | -0.410 |

```
# Calculate adjusted means (LS means) at mean baseline value
base_mean = ancova_df["BASE"].mean()
var_cov = model.cov_params()
ls_means = []

# LS means calculated at baseline mean
base_mean

for i, trt in enumerate(treatments):
    # Create prediction vector for LS mean calculation
    # Model: CHG = intercept + trt_effect1*(trt==1) + trt_effect2*(trt==2) + base_effect*basel
    x_pred = np.array([1, int(i==1), int(i==2), base_mean])

    # Calculate LS mean
    ls_mean = model.predict(pd.DataFrame({"TRTP": [trt], "BASE": [base_mean]}))[0]

    # Calculate standard error for confidence interval
    se_pred = np.sqrt(x_pred @ var_cov @ x_pred.T)

    ls_means.append({
        "Treatment": trt,
        "LS_Mean": ls_mean,
        "SE": se_pred,
        "CI_Lower": ls_mean - 1.96 * se_pred,
        "CI_Upper": ls_mean + 1.96 * se_pred
    })

ls_means_df = pl.DataFrame(ls_means)
# LS Means (95% CI)
ls_means_df
```

| Treatment<br>str | LS_Mean<br>f64 | SE<br>f64 | CI_Lower<br>f64 | CI_Upper<br>f64 |
|---|---|---|---|---|
| ”Placebo” | 0.071554 | 0.171563 | -0.264709 | 0.407818 |
| ”Xanomeline Low Dose” | -0.105215 | 0.171308 | -0.440977 | 0.230548 |
| ”Xanomeline High Dose” | 0.388498 | 0.177055 | 0.041471 | 0.735525 |

## 10.8 Step 6: Pairwise Treatment Comparisons

We calculate treatment differences and their statistical significance.

```
# Calculate pairwise comparisons vs. placebo
tbl2_data = []
comparisons = [
    ("Xanomeline Low Dose vs. Placebo", "TRTP[T.Xanomeline Low Dose]"),
    ("Xanomeline High Dose vs. Placebo", "TRTP[T.Xanomeline High Dose]")
]

for comp_name, trt_coef in comparisons:
    # Extract coefficient estimates
    coef = model.params[trt_coef]
    se = model.bse[trt_coef]
    t_stat = coef / se
    df = model.df_resid
    p_value = 2 * (1 - scipy_stats.t.cdf(abs(t_stat), df))

    # Calculate confidence interval
    ci_lower = coef - scipy_stats.t.ppf(0.975, df) * se
    ci_upper = coef + scipy_stats.t.ppf(0.975, df) * se

    tbl2_data.append({
        "Comparison": comp_name,
        "Estimate": coef,
        "SE": se,
        "CI_Lower": ci_lower,
        "CI_Upper": ci_upper,
        "t_stat": t_stat,
```

```
        "p_value": p_value
    })

comparison_df = pl.DataFrame(tbl2_data)
# Treatment comparisons vs. placebo
comparison_df
```

| Comparison<br>str | Estimate<br>f64 | SE<br>f64 | CI_Lower<br>f64 | CI_Upper<br>f64 | t_stat<br>f64 | p_value<br>f64 |
|---|---|---|---|---|---|---|
| ”Xanomeline Low Dose vs. Placeb… | -0.176769 | 0.242635 | -0.654862 | 0.301324 | -0.728539 | 0.467031 |
| ”Xanomeline High Dose vs. Place… | 0.316943 | 0.246806 | -0.169369 | 0.803256 | 1.284179 | 0.200383 |

## 10.9 Step 7: Prepare Tables for RTF Output

We format the analysis results into publication-ready tables.

```
# Table 1: Descriptive Statistics and LS Means
tbl1_data = []
for s, ls in zip(desc_stats, ls_means):
    tbl1_data.append([
        s["Treatment"],
        str(s["N_Baseline"]),
        f"{s['Baseline_Mean']:.1f} ({s['Baseline_SD']:.2f})" if not np.isnan(s['Baseline_Mean']
        str(s["N_Week24"]),
        f"{s['Week24_Mean']:.1f} ({s['Week24_SD']:.2f})" if not np.isnan(s['Week24_Mean']) else
        str(s["N_Change"]),
        f"{s['Change_Mean']:.1f} ({s['Change_SD']:.2f})" if not np.isnan(s['Change_Mean']) else
        f"{ls['LS_Mean']:.2f} ({ls['CI_Lower']:.2f}, {ls['CI_Upper']:.2f})"
    ])

tbl1 = pl.DataFrame(tbl1_data, orient="row", schema=[
    "Treatment", "N_Base", "Mean_SD_Base", "N_Wk24", "Mean_SD_Wk24",
    "N_Chg", "Mean_SD_Chg", "LS_Mean_CI"
])

# Table 1 - Descriptive Statistics and LS Means
tbl1
```

| Treatment<br>str | N_Base<br>str | Mean_SD_Base<br>str | N_Wk24<br>str | Mean_SD_Wk24<br>str | N_Chg<br>str | Mean_SD_C<br>str |
|---|---|---|---|---|---|---|
| "Placebo" | "79" | "5.7 (2.23)" | "79" | "5.6 (1.65)" | "79" | "-0.0 (2.32)" |
| "Xanomeline Low Dose" | "79" | "5.4 (0.95)" | "79" | "5.4 (1.06)" | "79" | "-0.1 (1.02)" |
| "Xanomeline High Dose" | "74" | "5.4 (1.37)" | "74" | "5.8 (2.21)" | "74" | "0.4 (1.65)" |

```
# Table 2: Pairwise Comparisons (formatted for output)
tbl2_formatted = []
for row in tbl2_data:
    tbl2_formatted.append([
        row["Comparison"],
        f"{row['Estimate']:.2f} ({row['CI_Lower']:.2f}, {row['CI_Upper']:.2f})",
        f"{row['p_value']:.4f}" if row['p_value'] >= 0.0001 else "<0.0001"
    ])

tbl2 = pl.DataFrame(tbl2_formatted, orient="row", schema=["Comparison", "Diff_CI", "P_Value"])

# Table 2 - Pairwise Comparisons
tbl2
```

| Comparison<br>str | Diff_CI<br>str | P_Value<br>str |
|---|---|---|
| "Xanomeline Low Dose vs. Placeb… | "-0.18 (-0.65, 0.30)" | "0.4670" |
| "Xanomeline High Dose vs. Place… | "0.32 (-0.17, 0.80)" | "0.2004" |

## 10.10 Step 8: Create Regulatory-Compliant RTF Document

We generate a comprehensive efficacy table following regulatory submission standards.

```
# Create comprehensive RTF document with multiple table sections
doc_ancova = rtf.RTFDocument(
    df=[tbl1, tbl2],
    rtf_title=rtf.RTFTitle(
        text=[
```

```
            "Analysis of Covariance (ANCOVA) of Change from Baseline in",
            "Fasting Glucose (mmol/L) at Week 24 (LOCF)",
            "Efficacy Analysis Population"
        ]
    ),
    rtf_column_header=[
        # Header for descriptive statistics table
        [
            rtf.RTFColumnHeader(
                text=["", "Baseline", "Week 24 (LOCF)", "Change from Baseline", ""],
                col_rel_width=[3, 2, 2, 3, 2],
                text_justification=["l", "c", "c", "c", "c"],
                text_format="b"
            ),
            rtf.RTFColumnHeader(
                text=[
                    "Treatment Group",
                    "N", "Mean (SD)",
                    "N", "Mean (SD)",
                    "N", "Mean (SD)",
                    "LS Mean (95% CI){^a}"
                ],
                col_rel_width=[3, 0.7, 1.3, 0.7, 1.3, 0.7, 1.3, 2],
                text_justification=["l"] + ["c"] * 7,
                border_bottom="single",
                text_format="b"
            )
        ],
        # Header for pairwise comparisons table
        [
            rtf.RTFColumnHeader(
                text=[
                    "Pairwise Comparison",
                    "Difference in LS Mean (95% CI){^a}",
                    "p-Value{^b}"
                ],
                col_rel_width=[5, 4, 2],
                text_justification=["l", "c", "c"],
                text_format="b",
                border_bottom="single"
```

```
            )
        ]
    ],
    rtf_body=[
        # Body for descriptive statistics
        rtf.RTFBody(
            col_rel_width=[3, 0.7, 1.3, 0.7, 1.3, 0.7, 1.3, 2],
            text_justification=["l"] + ["c"] * 7
        ),
        # Body for pairwise comparisons
        rtf.RTFBody(
            col_rel_width=[5, 4, 2],
            text_justification=["l", "c", "c"]
        )
    ],
    rtf_footnote=rtf.RTFFootnote(
        text=[
            "{^a}LS means and differences in LS means are based on an ANCOVA model with treatme
            f"{{^b}}p-values are from the ANCOVA model testing treatment effects (overall F-tes
            "LOCF (Last Observation Carried Forward) approach is used for missing Week 24 value
            "ANCOVA = Analysis of Covariance; CI = Confidence Interval; LS = Least Squares; SD
        ]
    ),
    rtf_source=rtf.RTFSource(
        text=[
            "Source: ADLBC Analysis Dataset",
            f"Analysis conducted: {pd.Timestamp.now().strftime('%d%b%Y').upper()}",
            "Statistical software: Python (statsmodels)"
        ]
    )
)

# Generate RTF file
doc_ancova.write_rtf("rtf/tlf_efficacy_ancova.rtf")
```

```
rtf/tlf_efficacy_ancova.rtf
```

```
PosixPath('pdf/tlf_efficacy_ancova.pdf')
```

# Part III

# eCTD submission package

# References

Wickham, Hadley, and Jennifer Bryan. 2023. *R Packages*. O'Reilly Media, Inc.