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 trials.

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.

1 Introduction

This is a book created from Markdown and executable code.

See Knuth (1984) for additional discussion of literate programming.

```
import polars as pl
from datetime import datetime

df = pl.DataFrame(
    {
        "integer": [1, 2, 3],
        "date": [
            datetime(2025, 1, 1),
            datetime(2025, 1, 2),
            datetime(2025, 1, 3),
        ],
        "float": [4.0, 5.0, 6.0],
        "string": ["a", "b", "c"],
    }
)

df
```

integer i64	date datetime[s]	float f64	string str
1	2025-01-01 00:00:00	4.0	”a”
2	2025-01-02 00:00:00	5.0	”b”
3	2025-01-03 00:00:00	6.0	”c”

Part I

Environment and toolchain

2 Python developer setup

Objective

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

2.1 Development environments

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

2.1.1 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.

2.1.2 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.

2.1.3 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.

2.2 VS Code settings

2.2.1 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.

2.3 Terminal setup

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

2.3.1 Shell

Any modern shell works well:

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

2.3.2 Terminal emulator

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

2.4 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)

2.4.1 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 SAP? 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.

2.5 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.

3 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.

3.1 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`.

3.2 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.

3.3 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.

3.4 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
```

3.5 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.

3.6 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
```

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

3.6.1 Project structure

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.

3.7 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
```

3.8 Managing dependencies

3.8.1 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.

3.8.2 Removing dependencies

Remove a package:

```
uv remove pandas
```

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

3.9 Lock files and syncing

3.9.1 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.

3.9.2 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.

3.10 Running commands

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

3.10.1 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
```

3.10.2 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.

3.10.3 `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.

3.11 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"
```

3.11.1 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)

3.11.2 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.

3.12 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"
```

3.13 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.

4 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.

4.1 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).

4.2 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.

4.2.1 Installation

Add Ruff as a development dependency:

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

4.2.2 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.

4.2.3 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.

4.2.4 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.

4.3 Type checking with mypy

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

4.3.1 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.

4.3.2 Installation

Add mypy as a development dependency:

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

4.3.3 Basic usage

Check types in your code:

```
uv run mypy .
```

4.3.4 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.

4.3.5 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.

4.3.6 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`).

4.4 Testing with pytest

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

4.4.1 Installation

Add pytest and coverage tools:

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

4.4.2 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
```

4.4.3 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
```

4.4.4 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.

4.4.5 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",
]
```

4.5 Documentation generation

For clinical reporting projects, documentation serves two purposes:

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

4.5.1 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.

4.5.2 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.

4.6 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`

4.6.1 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.

4.7 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.

4.8 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 ===============================
```

4.9 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.

4.10 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.

Part 2 of this book will use these tools to build real clinical study reports, demonstrating TLF generation with `polars` and `rtflite`.

Part II

Reporting packages

5 Polars

5.1 Polars

We use Polars in this book. The syntax aligns with tidyverse-style pipelines. The sections below cover basic examples. For deeper dives, see the Polars user guide or the book Python Polars: The Definitive Guide.

5.2 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.head()
```

STUDYID str	USUBJID str	TRT01A str	AGE f64	SEX 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”	”01-701-1033”	”Xanomeline Low Dose”	74.0	”Male”
”CDISCPILOT01”	”01-701-1034”	”Xanomeline High Dose”	77.0	”Female”

5.3 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").head()
```

STUDYID str	USUBJID str	TRT01A str	AGE f64	SEX 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”
”CDISCPILOT01”	”01-701-1111”	”Xanomeline Low Dose”	81.0	”Female”
”CDISCPILOT01”	”01-701-1133”	”Xanomeline High Dose”	81.0	”Female”

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

STUDYID str	USUBJID str	TRT01A str	AGE f64	SEX 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”	”01-701-1047”	”Placebo”	85.0	”Female”
”CDISCPILOT01”	”01-701-1097”	”Xanomeline Low Dose”	68.0	”Male”

5.4 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")
]).head()
```

STUDYID str	USUBJID str	TRT01A str	AGE f64	SEX str	AGECAT str
”CDISCPILOT01”	”01-701-1015”	”Placebo”	63.0	”Female”	”<65”
”CDISCPILOT01”	”01-701-1023”	”Placebo”	64.0	”Male”	”<65”
”CDISCPILOT01”	”01-701-1028”	”Xanomeline High Dose”	71.0	”Male”	”>=65”
”CDISCPILOT01”	”01-701-1033”	”Xanomeline Low Dose”	74.0	”Male”	”>=65”
”CDISCPILOT01”	”01-701-1034”	”Xanomeline High Dose”	77.0	”Female”	”>=65”

5.5 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").count().sort("TRT01A")
```

```
/tmp/ipykernel_7436/3512064118.py:2: DeprecationWarning: `GroupBy.count` was renamed; use `Grou
  adsl.group_by("TRT01A").count().sort("TRT01A")
```

TRT01A str	count u32
”Placebo”	86
”Xanomeline High Dose”	84

TRT01A str	count u32
”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 str	mean_age f64	sd_age f64
”Placebo”	75.2	8.59
”Xanomeline High Dose”	74.4	7.89
”Xanomeline Low Dose”	75.7	8.29

5.6 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 str	AGE f64	SEX str	TRT01A str
”01-701-1015”	63.0	”Female”	”Placebo”
”01-701-1023”	64.0	”Male”	”Placebo”
”01-701-1028”	71.0	”Male”	”Xanomeline High Dose”

5.7 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 str	Xanomeline High Dose u32	Xanomeline Low Dose u32	Placebo u32
”Male”	44	34	33
”Female”	40	50	53

Part III

Tables, Listings, and Figures

6 Disposition of Participants Table

6.1 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
```

6.2 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 str	TRT01P str	TRT01PN i64	DISCONFL str	DCREASCD str
"01-701-1015"	"Placebo"	0	""	"Completed"
"01-701-1023"	"Placebo"	0	"Y"	"Adverse Event"
"01-701-1028"	"Xanomeline High Dose"	81	""	"Completed"
"01-701-1033"	"Xanomeline Low Dose"	54	"Y"	"Sponsor Decision"
"01-701-1034"	"Xanomeline High Dose"	81	""	"Completed"
…	…	…	…	…
"01-718-1254"	"Xanomeline Low Dose"	54	""	"Completed"
"01-718-1328"	"Xanomeline High Dose"	81	"Y"	"Withdrew Consent"
"01-718-1355"	"Placebo"	0	""	"Completed"
"01-718-1371"	"Xanomeline High Dose"	81	"Y"	"Adverse Event"
"01-718-1427"	"Xanomeline High Dose"	81	"Y"	"Lack of Efficacy"

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

6.4 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

6.5 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 str	n_0 u32	n_54 u32	n_81 u32	pct_0 f64	pct_54 f64	pct_81 f64
”Discontinued”	28	59	57	32.6	70.2	67.9

6.6 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 str	n_0 u32	n_54 u32	n_81 u32	pct_0 f64	pct_54 f64	pct_81 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
” Lack of Efficacy”	3	0	1	3.5	0.0	1.2
” Lost to Follow-up”	1	1	0	1.2	1.2	0.0
” Physician Decision”	1	0	2	1.2	0.0	2.4
” Protocol Violation”	1	1	1	1.2	1.2	1.2
” Sponsor Decision”	2	2	3	2.3	2.4	3.6
” Withdrew Consent”	9	10	8	10.5	11.9	9.5

6.7 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 str	n_0 u32	n_54 u32	n_81 u32	pct_0 f64	pct_54 f64	pct_81 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
" Adverse Event"	8	44	40	9.3	52.4	47.6
" Death"	2	1	0	2.3	1.2	0.0
…	…	…	…	…	…	…
" Lost to Follow-up"	1	1	0	1.2	1.2	0.0
" Physician Decision"	1	0	2	1.2	0.0	2.4
" Protocol Violation"	1	1	1	1.2	1.2	1.2
" Sponsor Decision"	2	2	3	2.3	2.4	3.6
" Withdrew Consent"	9	10	8	10.5	11.9	9.5

6.8 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
```

7 Study Population Table

7.1 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
```

7.2 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 str	TRT01P str	ITTFL str	EFFFL str	SAFFL str
"01-701-1015"	"Placebo"	"Y"	"Y"	"Y"
"01-701-1023"	"Placebo"	"Y"	"Y"	"Y"
"01-701-1028"	"Xanomeline High Dose"	"Y"	"Y"	"Y"
"01-701-1033"	"Xanomeline Low Dose"	"Y"	"Y"	"Y"
"01-701-1034"	"Xanomeline High Dose"	"Y"	"Y"	"Y"
…	…	…	…	…
"01-718-1254"	"Xanomeline Low Dose"	"Y"	"Y"	"Y"
"01-718-1328"	"Xanomeline High Dose"	"Y"	"Y"	"Y"
"01-718-1355"	"Placebo"	"Y"	"Y"	"Y"
"01-718-1371"	"Xanomeline High Dose"	"Y"	"Y"	"Y"
"01-718-1427"	"Xanomeline High Dose"	"Y"	"Y"	"Y"

7.3 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 High Dose”	84
”Xanomeline Low Dose”	84

7.4 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)
    )
```

7.5 Step 4: Count Each Population

Now we calculate participant counts for each analysis population.

7.5.1 All Randomized Participants

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

pop_all
```

TRT01P str	n u32	population str
”Xanomeline High Dose”	84	”Participants in population”
”Xanomeline Low Dose”	84	”Participants in population”
”Placebo”	86	”Participants in population”

7.5.2 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 str	n u32	population str
”Xanomeline High Dose”	84	”Participants included in ITT p…
”Placebo”	86	”Participants included in ITT p…
”Xanomeline Low Dose”	84	”Participants included in ITT p…

7.5.3 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 str	n u32	population str
”Xanomeline High Dose”	74	”Participants included in effic…
”Xanomeline Low Dose”	81	”Participants included in effic…
”Placebo”	79	”Participants included in effic…

7.5.4 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 str	n u32	population str
”Xanomeline High Dose”	84	”Participants included in safet…
”Placebo”	86	”Participants included in safet…
”Xanomeline Low Dose”	84	”Participants included in safet…

7.6 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 str	n u32	population str
”Xanomeline High Dose”	84	”Participants in population”
”Xanomeline Low Dose”	84	”Participants in population”
”Placebo”	86	”Participants in population”
”Xanomeline High Dose”	84	”Participants included in ITT p…
”Placebo”	86	”Participants included in ITT p…
…	…	…
”Xanomeline Low Dose”	81	”Participants included in effic…
”Placebo”	79	”Participants included in effic…
”Xanomeline High Dose”	84	”Participants included in safet…
”Placebo”	86	”Participants included in safet…
”Xanomeline Low Dose”	84	”Participants included in safet…

7.7 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 str	n u32	population str	total u32	pct f64
”Xanomeline High Dose”	84	”Participants in population”	84	100.0
”Xanomeline Low Dose”	84	”Participants in population”	84	100.0
”Placebo”	86	”Participants in population”	86	100.0
”Xanomeline High Dose”	84	”Participants included in ITT p…	84	100.0
”Placebo”	86	”Participants included in ITT p…	86	100.0
…	…	…	…	…
”Xanomeline Low Dose”	81	”Participants included in effic…	84	96.4

TRT01P str	n u32	population str	total u32	pct f64
”Placebo”	79	”Participants included in effic…	86	91.9
”Xanomeline High Dose”	84	”Participants included in safet…	84	100.0
”Placebo”	86	”Participants included in safet…	86	100.0
”Xanomeline Low Dose”	84	”Participants included in safet…	84	100.0

7.8 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 str	n u32	population str	total u32	pct f64	display str
”Xanomeline High Dose”	84	”Participants in population”	84	100.0	”84”
”Xanomeline Low Dose”	84	”Participants in population”	84	100.0	”84”
”Placebo”	86	”Participants in population”	86	100.0	”86”
”Xanomeline High Dose”	84	”Participants included in ITT p…	84	100.0	”84 (100.0)”
”Placebo”	86	”Participants included in ITT p…	86	100.0	”86 (100.0)”
…	…	…	…	…	…
”Xanomeline Low Dose”	81	”Participants included in effic…	84	96.4	”81 (96.4)”

TRT01P str	n u32	population str	total u32	pct f64	display str
”Placebo”	79	”Participants included in effic…	86	91.9	”79 (91.9)”
”Xanomeline High Dose”	84	”Participants included in safet…	84	100.0	”84 (100.0)”
”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)”

7.9 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 str	Placebo str	Xanomeline Low Dose str	Xanomeline High Dose 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)”

7.10 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

8 Baseline Characteristics Table

8.1 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
```

8.2 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 str	TRT01P str	AGE f64	SEX str	RACE str
"01-701-1015"	"Placebo"	63.0	"Female"	"White"
"01-701-1023"	"Placebo"	64.0	"Male"	"White"

USUBJID str	TRT01P str	AGE f64	SEX str	RACE str
”01-701-1028”	”Xanomeline High Dose”	71.0	”Male”	”White”
”01-701-1033”	”Xanomeline Low Dose”	74.0	”Male”	”White”
”01-701-1034”	”Xanomeline High Dose”	77.0	”Female”	”White”
…	…	…	…	…
”01-718-1254”	”Xanomeline Low Dose”	78.0	”Male”	”White”
”01-718-1328”	”Xanomeline High Dose”	86.0	”Male”	”White”
”01-718-1355”	”Placebo”	79.0	”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”

8.3 Step 2: Calculate Summary Statistics

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

8.3.1 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 str	mean f64	sd f64	median f64	min f64	max f64	n u32
”Placebo”	75.2	8.59	76.0	52.0	89.0	86
”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

8.3.2 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 str	SEX str	len u32	total u32	pct f64
”Xanomeline High Dose”	”Female”	40	84	47.6
”Xanomeline Low Dose”	”Male”	34	84	40.5
”Placebo”	”Male”	33	86	38.4
”Xanomeline Low Dose”	”Female”	50	84	59.5
”Xanomeline High Dose”	”Male”	44	84	52.4
”Placebo”	”Female”	53	86	61.6

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

TRT01P str	RACE str	len u32	total u32	pct f64
”Xanomeline High Dose”	”American Indian Or Alaska Nati…	1	84	1.2
”Placebo”	”White”	78	86	90.7
”Xanomeline Low Dose”	”Black Or African American”	6	84	7.1
”Placebo”	”Black Or African American”	8	86	9.3
”Xanomeline High Dose”	”Black Or African American”	9	84	10.7
”Xanomeline Low Dose”	”White”	78	84	92.9
”Xanomeline High Dose”	”White”	74	84	88.1

8.4 Step 3: Format Results

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

8.4.1 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 str	mean_sd str	median_range str
”Placebo”	”75.2 (8.59)”	”76.0 [52.0, 89.0]”
”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]”

8.4.2 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 str	SEX str	n_pct str
”Xanomeline High Dose”	”Female”	”40 (47.6%)”
”Xanomeline Low Dose”	”Male”	”34 (40.5%)”
”Placebo”	”Male”	”33 (38.4%)”
”Xanomeline Low Dose”	”Female”	”50 (59.5%)”
”Xanomeline High Dose”	”Male”	”44 (52.4%)”
”Placebo”	”Female”	”53 (61.6%)”

8.5 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 str	Placebo str	Xanomeline Low Dose str	Xanomeline High Dose 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]”
”Sex”	””	””	””
” Female”	”53 (61.6%)”	”50 (59.5%)”	”40 (47.6%)”
” Male”	”33 (38.4%)”	”34 (40.5%)”	”44 (52.4%)”
”Race”	””	””	””
” White”	”78 (90.7%)”	”78 (92.9%)”	”74 (88.1%)”
” 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%)”

8.6 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
```

9 Adverse Events Summary Table

9.1 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
```

9.2 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"]).head()
```

USUBJID str	TRT01A str	SAFFL str
”01-701-1015”	”Placebo”	”Y”
”01-701-1023”	”Placebo”	”Y”
”01-701-1028”	”Xanomeline High Dose”	”Y”
”01-701-1033”	”Xanomeline Low Dose”	”Y”
”01-701-1034”	”Xanomeline High Dose”	”Y”

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

USUBJID str	AEREL str	AESER str	AEOUT str	AEACN 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”	””
”01-701-1023”	”POSSIBLE”	”N”	”NOT RECOVERED/NOT RESOLVED”	””
”01-701-1023”	”PROBABLE”	”N”	”NOT RECOVERED/NOT 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”)

9.3 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
```

9.4 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)
```

9.5 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	n	category	N	pct
str	u32	str	u32	f64
”Placebo”	0	”Discontinued due to adverse ev…	86	0.0

TRT01A str	n u32	category str	N u32	pct f64
”Xanomeline High Dose”	0	”Discontinued due to adverse ev…	84	0.0
”Xanomeline Low Dose”	0	”Discontinued due to adverse ev…	84	0.0
”Placebo”	86	”Participants in population”	86	null
”Xanomeline High Dose”	84	”Participants in population”	84	null
…	…	…	…	…
”Xanomeline High Dose”	2	”With serious adverse event”	84	2.4
”Xanomeline Low Dose”	1	”With serious adverse event”	84	1.2
”Placebo”	0	”With serious drug-related adve…	86	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

9.6 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 str	TRT01A str	n_display str	pct_display str
”Participants in population”	”Placebo”	”86”	””
”Participants in population”	”Xanomeline High Dose”	”84”	””

category str	TRT01A str	n_display str	pct_display str
”Participants in population”	”Xanomeline Low Dose”	”84”	””
”With any adverse event”	”Placebo”	”69”	”(80.2)”
”With any adverse event”	”Xanomeline High Dose”	”79”	”(94.0)”
…	…	…	…
”Who died”	”Xanomeline High Dose”	”0”	”(0.0)”
”Who died”	”Xanomeline Low Dose”	”1”	”(1.2)”
”Discontinued due to adverse ev…	”Placebo”	”0”	”(0.0)”
”Discontinued due to adverse ev…	”Xanomeline High Dose”	”0”	”(0.0)”
”Discontinued due to adverse ev…	”Xanomeline Low Dose”	”0”	”(0.0)”

9.7 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 str	n_display_Placebo str	pct_display_Placebo str	n_display_Xanomeline Low D str
"Participants in population"	"86"	""	"84"
"With any adverse event"	"69"	"(80.2)"	"77"
"With drug-related adverse even…	"44"	"(51.2)"	"73"
"With serious adverse event"	"0"	"(0.0)"	"1"
"With serious drug-related adve…	"0"	"(0.0)"	"1"
"Who died"	"2"	"(2.3)"	"1"
"Discontinued due to adverse ev…	"0"	"(0.0)"	"0"

9.8 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
```

10 Specific Adverse Events Table

This article demonstrates how to create a specific adverse events table by System Organ Class and Preferred Term.

10.1 Setup

```
import polars as pl
import rtflite as rtf

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

10.2 Prepare AE Summary Data

```
# Get safety population counts and AE data
adsl_safety = adsl.filter(pl.col("SAFFL") == "Y").select(["USUBJID", "TRT01A"])
adae_safety = adae.join(adsl_safety, on="USUBJID", how="inner")
pop_counts = adsl_safety.group_by("TRT01A").agg(N=pl.len()).sort("TRT01A")

# Calculate AE counts by SOC and term
ae_counts = (
    adae_safety.with_columns(pl.col("AEDECOD").str.to_titlecase())
    .group_by(["TRT01A", "AEBODSYS", "AEDECOD"])
    .agg(n=pl.col("USUBJID").n_unique())
    .sort(["AEBODSYS", "AEDECOD", "TRT01A"])
)
```

```
# Build table rows
table_data = [
    ["Participants in population"] + [str(pop_counts.filter(pl.col("TRT01A") == t)["N"][0]) fo
    [""] * 4  # Blank row
]

# Add SOC and AE term rows
for soc in ae_counts["AEBODSYS"].unique().sort():
    table_data.append([soc] + [""] * 3)
    soc_data = ae_counts.filter(pl.col("AEBODSYS") == soc)

    for ae in soc_data["AEDECOD"].unique().sort():
        row = [f"  {ae}"]
        for trt in treatments:
            count = soc_data.filter((pl.col("AEDECOD") == ae) & (pl.col("TRT01A") == trt))
            row.append(str(count["n"][0]) if count.height > 0 else "0")
        table_data.append(row)

df_ae_specific = pl.DataFrame(table_data, schema=[""] + treatments, orient="row")
```

10.3 Create RTF Output

```
doc_ae_specific = rtf.RTFDocument(
    df=df_ae_specific,
    rtf_title=rtf.RTFTitle(text=["Specific Adverse Events", "(Safety Analysis Population)"]),
    rtf_column_header=rtf.RTFColumnHeader(
        text=["", "Placebo\nn", "Xanomeline Low Dose\nn", "Xanomeline High Dose\nn"],
        col_rel_width=[4, 1, 1, 1],
        text_justification=["l", "c", "c", "c"],
    ),
    rtf_body=rtf.RTFBody(
        col_rel_width=[4, 1, 1, 1],
        text_justification=["l", "c", "c", "c"],
        text_font_style=lambda df, i, j: "bold" if j == 0 and "  " not in str(df[i, j]) else "
    ),
    rtf_footnote=rtf.RTFFootnote(text=["Number of participants with specific adverse events."]
    rtf_source=rtf.RTFSource(text=["Source: ADSL and ADAE datasets"])
```

```
)

doc_ae_specific.write_rtf("rtf/tlf_ae_specific.rtf")
```

```
rtf/tlf_ae_specific.rtf
```

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

11 ANCOVA Efficacy Analysis

This article demonstrates how to create an ANCOVA efficacy table for glucose levels at Week 24 with LOCF imputation.

11.1 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
from importlib.resources import files

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

11.2 Prepare Analysis Data

```
# Clean data types and filter for efficacy population
adlbc_clean = adlbc.with_columns(
    [pl.col(c).cast(str).str.strip_chars() for c in ["USUBJID", "PARAMCD", "AVISIT", "TRTP"]]
)
adsl_eff = adsl.filter(pl.col("EFFFL") == "Y").select(["USUBJID"])
adlbc_eff = adlbc_clean.join(adsl_eff, on="USUBJID", how="inner")

# Apply LOCF for glucose data up to Week 24
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")
    ])
    .filter(pl.col("Baseline").is_not_null() & pl.col("Week 24").is_not_null())
    .with_columns((pl.col("Week 24") - pl.col("Baseline")).alias("CHG"))
)

# Calculate descriptive statistics
desc_stats = []
for trt in treatments:
    trt_data = gluc_data.filter(pl.col("TRTP") == trt)
    baseline_full = adlbc_eff.filter(
        (pl.col("PARAMCD") == "GLUC") & (pl.col("AVISIT") == "Baseline") & (pl.col("TRTP") == t
    )

    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"].mean() if trt_data.height > 0 else np.nan,
        "Week24_SD": trt_data["Week 24"].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
    })

# Perform ANCOVA
ancova_df = gluc_data.to_pandas()
ancova_df["TRTP"] = pd.Categorical(ancova_df["TRTP"], categories=treatments)
model = smf.ols("CHG ~ TRTP + BASE", data=ancova_df).fit()

# Calculate LS means and confidence intervals
```

```
base_mean = ancova_df["BASE"].mean()
var_cov = model.cov_params()
ls_means = []

for i, trt in enumerate(treatments):
    x_pred = np.array([1, int(i==1), int(i==2), base_mean])
    ls_mean = model.predict(pd.DataFrame({"TRTP": [trt], "BASE": [base_mean]}))[0]
    se_pred = np.sqrt(x_pred @ var_cov @ x_pred.T)

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

11.3 Create Tables for RTF Output

```
# Table 1: Descriptive Statistics
tbl1_data = [
    [
        s["Treatment"],
        str(s["N_Baseline"]),
        f"{s['Baseline_Mean']:.1f} ({s['Baseline_SD']:.2f})",
        str(s["N_Week24"]),
        f"{s['Week24_Mean']:.1f} ({s['Week24_SD']:.2f})",
        str(s["N_Change"]),
        f"{s['Change_Mean']:.1f} ({s['Change_SD']:.2f})",
        f"{ls['LS_Mean']:.2f} ({ls['CI_Lower']:.2f}, {ls['CI_Upper']:.2f})"
    ]
    for s, ls in zip(desc_stats, ls_means)
]

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 2: Pairwise Comparisons
tbl2_data = []
for comp_name, trt_name in [("Xanomeline Low Dose - Placebo", "TRTP[T.Xanomeline Low Dose]"),
                            ("Xanomeline High Dose - Placebo", "TRTP[T.Xanomeline High Dose]")]:
    coef = model.params[trt_name]
    se = model.bse[trt_name]
    t_stat = coef / se
    p_value = 2 * (1 - scipy_stats.t.cdf(abs(t_stat), model.df_resid))

    tbl2_data.append([
        comp_name,
        f"{coef:.2f} ({coef - 1.96*se:.2f}, {coef + 1.96*se:.2f})",
        f"{p_value:.3f}"
    ])

tbl2 = pl.DataFrame(tbl2_data, orient="row", schema=["Comparison", "Diff_CI", "P_Value"])
```

11.4 Create RTF Document

```
# Create RTF document with two sections
doc_ancova = rtf.RTFDocument(
    df=[tbl1, tbl2],
    rtf_title=rtf.RTFTitle(text=[
        "ANCOVA of Change from Baseline Glucose (mmol/L) at Week 24", "LOCF",
        "Efficacy Analysis Population"
    ]),
    rtf_column_header=[
        [rtf.RTFColumnHeader(text=["", "Baseline", "Week 24", "Change from Baseline"],
                             col_rel_width=[3, 2, 2, 4], text_justification=["l", "c", "c", "c"]),
         rtf.RTFColumnHeader(text=["Treatment", "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")],
        [rtf.RTFColumnHeader(text=["Pairwise Comparison", "Difference in LS Mean (95% CI){^a}"
                             col_rel_width=[5, 4, 2], text_justification=["l", "c", "c"])]
    ],
    rtf_body=[
        rtf.RTFBody(col_rel_width=[3, 0.7, 1.3, 0.7, 1.3, 0.7, 1.3, 2],
```

```
                    text_justification=["l"] + ["c"] * 7),
        rtf.RTFBody(col_rel_width=[5, 4, 2], text_justification=["l", "c", "c"])
    ],
    rtf_footnote=rtf.RTFFootnote(text=[
        "{^a}Based on an ANCOVA model after adjusting baseline value. LOCF approach is used to
        "ANCOVA = Analysis of Covariance, LOCF = Last Observation Carried Forward",
        "CI = Confidence Interval, LS = Least Squares, SD = Standard Deviation"
    ]),
    rtf_source=rtf.RTFSource(text=["Source: ADLBC dataset"])
)

doc_ancova.write_rtf("rtf/tlf_efficacy_ancova.rtf")
```

```
rtf/tlf_efficacy_ancova.rtf
```

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

12 Summary

In summary, this book has no content whatsoever.

```
1 + 1
```

```
2
```

References

Knuth, Donald E. 1984. “Literate Programming.” *Comput. J.* 27 (2): 97–111.