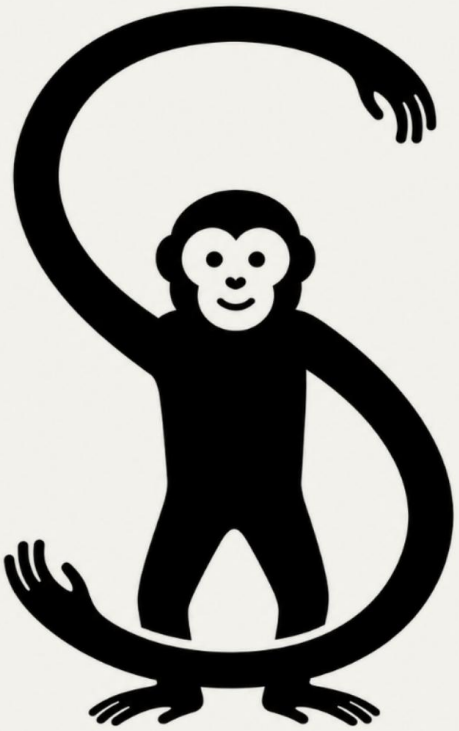
Siamang and Siamang Cloud: The Complete Guide

Designing, deploying, and analyzing sociological surveys with research-as-code



Siamang Labs

Based on the official Siamang documentation

Contents

Siamang and Siamang Cloud: The Complete Guide

Designing, Deploying, and Analyzing Sociological Surveys with Research-as-Code

A comprehensive user guide and reference, based on the official Siamang documentation (README, Manual, docs, and wiki of the Siamang-Labs/siamang repository).

Preface

Siamang is a source-available, research-as-code framework for sociological surveys: you define, deploy, and analyze questionnaires in pure Python. Siamang Cloud is the hosted platform built around the same engine. Siamang is source-available, not open source: the code is public and free for noncommercial use, while commercial use is available only through Siamang Cloud or under a separate corporate license requested by email. This textbook teaches both, from first installation to advanced analysis and reporting, using only the officially documented, user-facing features of the framework and the cloud platform.

Who This Book Is For

This book is written for researchers, survey methodologists, data analysts, and students who want to run professional survey studies with Siamang. It assumes basic familiarity with Python (writing and running simple scripts). No prior experience with Siamang, survey software, or web deployment is required.

How to Use This Book

- **Part I — Getting Started** (Chapters 1–3) takes you from zero to a working first survey.
- **Part II — Authoring Surveys** (Chapters 4–11) is the complete guide to questionnaire design: structure, question types, logic, quotas, variables, theming, and validation.
- **Part III — Deployment and Data** (Chapters 12–14) covers publishing surveys, simulating respondents, and moving data in and out.
- **Part IV — Analysis and Reporting** (Chapters 15–19) teaches statistics, tables, banners, charts, report documents, and a full end-to-end capstone tutorial.
- **Part V — Siamang Cloud** (Chapters 20–24) is the user guide to the hosted platform: accounts, the web app, teams, cloud deployment, analysis, connectors, scheduling, webhooks, and plans.
- **Appendices A–C** provide the CLI reference, the configuration reference, and a glossary with further resources.

Every chapter opens with learning objectives, walks through step-by-step procedures with complete code examples, and closes with a chapter summary and pointers to the original documentation. Chapters are self-contained enough to be consulted out of order, but Parts I–IV build on each other and are best read in sequence.

Scope Note

This book documents user-facing functionality only. Server-side, infrastructure, and internal architectural details of Siamang Cloud are intentionally out of scope. All content is derived from the official project documentation; where the documentation itself is inconsistent, the discrepancy is flagged in a note rather than silently resolved. Siamang is at version 0.6.0 (Beta) at the time of writing — expect details to evolve, and consult the online documentation for the latest changes.

Part I — Getting Started

Chapter 1. Introducing Siamang

Learning Objectives

By the end of this chapter, you will be able to:

- Describe what Siamang is and what problem it solves for survey researchers.
- Explain the research-as-code philosophy and its practical consequences for your day-to-day work.
- Name the stages of the Siamang pipeline — validate, preview, simulate, deploy, collect, analyze — and what happens at each one.
- Distinguish the source-available engine from the hosted Siamang Cloud platform.
- Determine which license applies to your situation and what each license permits.
- Decide whether Siamang fits your study, given its strengths and its current Beta limitations.

1.1 What Siamang Is

Siamang is a *research-as-code framework for sociological surveys*. Its tagline states the core idea directly: “Define variables, questionnaires, and logic in pure Python — then deploy, collect, and analyze in a single pipeline.”

In practical terms, Siamang gives you a Python library in which you declare everything that makes up a survey study — the measured quantities (called Variables), the questions respondents see, the pages those questions appear on, the routing logic between them, and the quotas that shape your sample — and then a set of commands and methods that carry that same declaration through validation, preview, deployment, data collection, and analysis. Nothing about your study lives in a proprietary file format or a point-and-click interface. Everything is ordinary Python code that you can read, edit, and run like any other Python program.

A first taste of what that looks like (you do not need to understand every line yet; Chapter 3 builds this example up properly):

```python
import siamang as sg

    age = sg.Variable("age", scale="ratio", label="Age")
    fav = sg.Variable(
        "fav_color", scale="nominal",
        label="Favourite colour",
        labels={1: "Red", 2: "Blue", 3: "Green"},
    )

    survey = sg.Questionnaire(
        title="Hello, siamang",
        pages=[
            sg.Page(
                name="main",
                title="Tell us a bit about yourself",
                items=[
                    sg.NumericInput("How old are you?", var=age, required=True),
                    sg.SingleChoice("What is your favourite colour?", var=fav),
                ],
            ),
        ],
    )
```

A few terms worth fixing now, because this book uses them consistently:

- A **Variable** is an atomic measurement unit — a named quantity with a measurement scale (`nominal`, `ordinal`, `interval`, or `ratio`), a human-readable label, and optionally value labels and missing-value conventions. It becomes a column in your analysis dataset.
- A **Question** binds a prompt to one (or several) Variables. Siamang ships seven question types: `SingleChoice`, `MultiChoice`, `LikertScale`, `NumericInput`, `OpenText`, `Matrix`, and `Ranking`.
- A **Page** is one screen of the survey as the respondent experiences it.
- A **Questionnaire** is the aggregate object that owns the pages and provides the operations of the pipeline: `validate()`, `simulate()`, `preview` (via the command line), and `deploy()`.

1.1.1 Research-as-code philosophy

The defining decision of Siamang's design is that a questionnaire is a **plain Python module** — not a row in a proprietary database, not an opaque export from a graphical builder. The project's own phrasing is blunt: "No GUI builders. No drag-and-drop. No lock-in."

This is a philosophical stance with concrete, practical consequences. Because your survey is a source file:

1) **It lives in version control.** Your questionnaire sits in Git alongside the rest of your research code. Every change to question wording, scale points, or routing is a commit with an author and a timestamp.
2) It can be reviewed, diffed, branched, and rolled back. A co-investigator's change to a skip rule shows up as a line-level diff in a pull request, exactly like a change to any other code.
3) **It can be tested with standard tooling.** You can exercise your survey definition with `pytest`, type-check it, and lint it — the same quality gates you already trust for code.
4) **It can be imported and inspected anywhere.** A notebook or an analysis pipeline can `from my_survey import survey` and examine the instrument directly.

Two further design commitments reinforce the philosophy. First, the *same* module flows through every stage of the survey lifecycle — there is no export/import step between authoring and fielding where information can drift. Second, all of Siamang's core data classes are declared `frozen=True, slots=True`: once constructed, a questionnaire cannot be mutated. This immutability keeps definitions side-effect free, thread-safe, and deterministic across compilation, simulation, and analysis — you can be certain that the survey you validated is byte-for-byte the survey respondents see.

> **Tip:** If you have ever lost track of "which version of the questionnaire was in the field when these responses arrived," the research-as-code model answers that question permanently: the survey is a commit hash, and the data export carries the same lineage as any other build artifact.

1.1.2 The single pipeline

Siamang organizes the entire survey lifecycle into one pipeline, and the *same* Python module moves through every stage of it:

1) **Validate** — `siamang validate my_survey.py` (or `survey.validate()` in Python) checks the questionnaire for structural and logical problems: duplicate question IDs, broken navigation targets, expressions that reference unknown variables, and malformed scripts. It reports "OK" if the questionnaire is well-formed and lists problems otherwise.
2) **Preview** — `siamang preview my_survey.py` builds the web frontend and serves it locally so you can fill the survey in your own browser. Responses land in a local SQLite database (`survey.db`).
3) **Simulate** — `survey.simulate(n=...)` generates a synthetic dataset that respects each variable's scale, labels, and ranges, so you can rehearse the full analysis before the survey ever goes into the field.

4) **Deploy** — `siamang deploy my_survey.py` provisions a backend and publishes the frontend, then prints the public URL where respondents can take the survey.
5) **Collect** — responses accumulate in the backend while the survey is in the field; you pull them back at any time.
6) **Analyze and report** — collected data loads as a `SurveyData` object that wraps a pandas `DataFrame` together with your variable metadata, so frequency tables, cross-tabulations, charts, and statistical tests come out labeled and publication-ready.

In its most compact form, the canonical pipeline from the project's README is:

```
pip install siamang
    siamang validate my_survey.py
    siamang preview  my_survey.py        # local preview
    siamang deploy   my_survey.py --backend supabase --frontend vercel
```

Two properties of this pipeline matter for how you will work. Because every stage consumes the same module, there is no "design file" that can fall out of sync with "the fielded instrument." And because the analysis layer reads your Variable metadata — labels, scales, missing-value codes — it can produce SPSS-like outputs (labeled frequency tables, cross-tabs with automatic statistical tests) directly in Python, without you re-entering codebook information downstream.

1.2 Editions and Licensing

1.2.1 Source-available engine and hosted Siamang Cloud

Siamang comes in two editions, and it is important to know which one a given chapter of this book is talking about.

The source-available engine is the Python package `siamang`, distributed through PyPI and developed in a public GitHub repository. At the time of writing, the latest release is version 0.6.0 (released 2026-08-30), and its development status is Beta. You install it with a single command and run everything — validation, preview, deployment, collection, analysis — from your own Python environment and command line:

```
pip install siamang
```

Siamang Cloud is the hosted, managed platform built on the same source-available engine, available in open beta at `https://app.siamang.org/login`. Be precise about the terminology: Siamang is source-available, not open source. The engine's source code is public — you can read it, audit it, and use it freely for noncommercial purposes — but it is not distributed under an OSI-approved open-source license, and commercial use is possible only through the two routes described in Section 1.2.2. Where the engine ex-

pects you to run commands yourself, the Cloud lets you author surveys as code in a browser, deploy to a public link with one click, collect responses, run analyses, view dashboards and reports, and collaborate with a team — without operating any of the underlying machinery yourself.

The two editions share one core: the same questionnaire model, the same validation rules, the same analysis concepts. The opening chapters of this book teach the shared foundation — concepts, question types, logic — using the source-available engine, because that is where the model is most visible. Later chapters cover the Cloud's browser-based workflow in detail.

> **Note:** Siamang is developed and operated by Siamang Labs LLC (Wyoming, USA). The source repository is `github.com/Siamang-Labs/siamang`.

1.2.2 Dual licensing

The Siamang engine is **dual-licensed**, and the license that applies to you depends on the kind of use you make of it.

TABLE 1.1

License	Who it covers	What you may do
PolyForm Noncommercial License 1.0.0	Noncommercial users	Use Siamang free of charge for personal projects, research, education, and use by noncommercial organizations such as charities, universities, public research bodies, and government institutions.
Commercial license	Commercial users	Required for any commercial self-hosted use of the engine; negotiated individually and requested by email. Alternatively, commercial teams can use Siamang through Siamang Cloud under its own Terms of Use.

The practical reading is straightforward. If you are a student, an academic researcher, a teacher, or you work for a charity, university, public research body, or government institution, you can use Siamang for your surveys at no cost under the PolyForm Noncommercial license. If your use is commercial — for example, running surveys as a paid service or inside a for-profit product — there are exactly two routes: (1) use Siamang through Siamang Cloud, the hosted platform, which is governed by its own Terms of Use and subscription plans; or (2) for self-hosted use of the engine, obtain a separate corporate license, negotiated individually and requested by emailing info@siamang-team.org (the terms are set out in `LICENSE-COMMERCIAL.md` in the repository). Until a written commercial license agreement is in place, no commercial rights to the self-hosted engine are granted.

One version nuance is worth knowing: the license switch is recent. Releases up to and including 0.5.0 remain available under the permissive MIT license; the dual-licensing model (PolyForm Noncom-

mercial plus commercial) applies from release 0.6.0 (2026-08-30), which records the change in the project's changelog. If your organization has compliance requirements, check the license file shipped with the exact version you install.

> **Warning:** "Free for research" does not mean "no license." PolyForm Noncommercial is a real license with a defined noncommercial boundary. If your survey work is funded, commissioned, or embedded in a commercial product, confirm which side of the boundary you are on before fielding. Remember the rule: Siamang is source-available, not open source — noncommercial use is free, and commercial use is available only through Siamang Cloud or under a separate corporate license requested by email.

1.3 When to Use Siamang

1.3.1 Target users, strengths, and current limitations

Siamang is built first and foremost for sociologists and social-science researchers who run surveys and want the kind of outputs they are used to from SPSS — labeled frequency tables, cross-tabulations, significance tests — but produced in Python. If you recognize yourself in any of the following, Siamang was designed with you in mind:

- You write (or want to write) reproducible research code and are comfortable editing Python files.
- You want your questionnaire under version control, reviewed like code, and testable with standard tooling.
- You want variable metadata — labels, scales, missing-value conventions — declared once and carried through to publication-ready tables and charts automatically.
- You want one pipeline from questionnaire definition to analyzed data, rather than a chain of disconnected tools.

The feature set matches that ambition across the full lifecycle:

TABLE 1.2

Area	Capabilities
Core model	Variables at all four measurement scales; seven question types; pages and blocks; skip logic via `show_if`/`hide_if`; quotas; built-in validation.
Reporting	Declarative tables (`FreqTable`, `CrossTable`, `GroupMeanTable`) and charts (`BarChart`, `BoxPlot`, `HeatMap`, `ScatterPlot`) with automatic labels, statistical tests, and metadata awareness.
Data I/O	Import/export for CSV, Excel (.xlsx), SPSS (.sav), Stata (.dta), and R. SPSS and Stata round-trip labels and missing values in both directions; CSV/Excel carry data only (labels via the accompanying JSON dictionary); R export is a bundle of CSV + JSON dictionary + .R loader script.
Scripts	Inline JavaScript at seven lifecycle trigger points for survey-side behavior such as option randomization.
Delivery	Local preview out of the box; deployment to hosted backends/frontends; a self-contained HTML bundle for offline use.

A particularly valuable capability at the design stage is **simulation**: `survey.simulate()` generates a synthetic dataset that respects each variable's scale, labels, and ranges — no server and no real respondents needed. You can rehearse your entire analysis before the survey ever goes into the field.

Typical studies that fit this model well include cross-sectional attitude surveys, multi-wave panel instruments (where version control pays for itself), experiments with randomized question or option order, and any study where an ethics board or funder expects an exact, auditable record of the instrument.

Against these strengths, set the current limitations honestly:

- Beta status. Version 0.6.0 is explicitly marked Beta. The public API and behavior can still evolve between releases, and you should expect to consult the changelog when upgrading.
- **Code-first workflow.** There is no graphical questionnaire builder in the source-available engine. If your team cannot or does not want to work in Python, that is a real barrier (the hosted Cloud softens it with browser-based authoring, but the artifacts are still code).
- **You operate the pieces yourself.** With the source-available engine you choose and connect the deployment backends and frontends; the hosted Cloud exists precisely for teams that would rather not.
- **One integration is experimental.** The Google Sheets backend is flagged as experimental for public web deployments; the documentation recommends the more conventional hosted options for production use.

Try it yourself: Before reading on, make a shortlist of one survey you have run (or plan to run). Write down its three most important measured quantities and, for each, decide its measurement scale — nominal, ordinal, interval, or ratio. In Chapter 3 you will turn ex-

actly this list into Siamang Variables, and the scale decision you make now will determine which statistics Siamang computes for you later.

If Siamang sounds like the right fit, the natural next step is to get it running. Chapter 2 walks you through installation requirements, the exact install commands, optional extras for charts and integrations, and the configuration file that holds your deployment settings — ending with a smoke test that proves your environment is ready for the quickstart in Chapter 3.

Chapter Summary

- Siamang is a research-as-code framework for sociological surveys: you define variables, questionnaires, and logic in pure Python, then deploy, collect, and analyze in a single pipeline.
- The research-as-code philosophy means your questionnaire is a plain Python module — version-controlled, reviewable, testable, and free of GUI lock-in.
- All core data classes are immutable (`frozen=True, slots=True`), so a validated questionnaire is exactly the questionnaire respondents see.
- The pipeline runs through six stages — validate, preview, simulate, deploy, collect, analyze/report — and the same module flows through all of them.
- The source-available engine (version 0.6.0, Beta) is installed from PyPI; Siamang Cloud (`app.siamang.org`) is the hosted platform that runs everything around the same engine, in open beta.
- Licensing is dual: PolyForm Noncommercial 1.0.0 covers free noncommercial use (research, education, charities, universities, government); commercial use is available only through Siamang Cloud or under a separate corporate license requested via info@siamang-team.org. Versions up to and including 0.5.0 remain MIT-licensed.
- Siamang suits researchers who want SPSS-like, metadata-driven outputs in Python with full reproducibility; its current limitations are Beta status, a code-first workflow, and an experimental Google Sheets backend.

Documentation References

- `README.md` — project definition and tagline, “No GUI builders. No drag-and-drop. No lock-in.”, feature matrix, deployment combinations, license section, canonical pipeline commands, Siamang Cloud section.
- `docs/index.md` — framework definition and the validate → preview → simulate → deploy → collect → analyze pipeline description.

- `wiki/Home.md` — project overview, feature matrix including the Cloud row, install command.
- `wiki/Core-Concepts.md` and `docs/concepts.md` — research-as-code philosophy, immutability of core dataclasses, the five-layer model.
- `wiki/Quickstart.md` and `docs/getting-started.md` — the four-layer authoring flow and lifecycle commands.
- `MANUAL.md` — install-from-Git instructions and single-file feature tour.
- `CHANGELOG.md` — version 0.6.0 Beta status, release history, and the license switch from MIT to dual licensing recorded in the 0.6.0 release entry.

Chapter 2. Installation and Configuration

Learning Objectives

By the end of this chapter, you will be able to:

- Check that your Python environment meets Siamang's requirements.
- Install Siamang from PyPI or directly from the Git repository.
- Choose and install the optional extras (`charts`, `gsheets`, `dev`) that match your work.
- Verify the installation with a version check and a runnable smoke test.
- Create and read the `~/.siamang.toml` configuration file with its four table types.
- Apply environment-variable overrides and protect the file's secrets with correct permissions.
- Use the four CLI subcommands — `validate`, `preview`, `deploy`, `init` — and discover their options through `--help`.

2.1 Installing Siamang

2.1.1 Requirements and the base install

Siamang requires **Python 3.11 or newer**. Versions 3.11, 3.12, and 3.13 are supported and tested, on both POSIX systems (Linux, macOS) and Windows; the package is OS-independent and ships type hints (via a `py.typed` marker) for editors and type checkers.

The base installation is a single command:

```
pip install siamang
```

If you prefer to install the development state directly from the Git repository:

```
pip install git+https://github.com/Siamang-Labs/siamang.git
```

The base package is deliberately self-contained: its core dependencies install automatically, so analysis, local preview, deployment, and data import/export all work out of the box. The table below lists what comes along for the ride and why.

TABLE 2.1

Package	Minimum version	Purpose
`pandas`	2.0	Data manipulation; the backbone of the `SurveyData` object you analyze.

Package	Minimum version	Purpose
`scipy`	1.11	Statistical tests (chi-square, t-test, ANOVA, and more).
`openpyxl`	3.1	Excel (.xlsx) import and export.
`pyreadstat`	1.2	SPSS (.sav) and Stata (.dta) import and export.
`fastapi`	0.110	The local preview server behind `siamang preview`.
`uvicorn`	0.29	The ASGI server that runs the local preview.
`markdown`	3.5	Rendering of `Report` documents to HTML.
`tabulate`	0.9	Markdown output from `to_markdown()` on report tables.
`supabase`	2.0	The Supabase production backend.
`requests`	2.31	HTTP client for the Netlify and Vercel deployment APIs.

One dependency area is *not* bundled: charts. Frequency tables and cross-tabulations (`data.report.freq()`, `data.report.crosstab()`) work without any plotting library, but rendering a chart requires the optional `charts` extra described below. If you try to render a chart without it, Siamang raises a clear error message telling you what to install — nothing fails silently.

2.1.2 Optional extras and verification

Optional features are installed through pip *extras* — bracketed suffixes on the package name that pull in additional dependencies. Three extras are meaningful today:

```
pip install "siamang[charts]"    # matplotlib >= 3.7 + seaborn >= 0.13 — required for data.plot.* (bar, boxplot, heatmap, scatter)
pip install "siamang[gsheets]"   # google-auth >= 2.0, google-auth-httplib2 >= 0.1, google-api-python-client >= 2.0
pip install "siamang[dev]"       # ruff >= 0.4, mypy >= 1.10, pytest >= 8.0, plus matplotlib/seaborn
```

TABLE 2.2

Extra	Installs	When you need it
`charts`	matplotlib, seaborn	You want to render charts via `data.plot.bar()`, `data.plot.boxplot()`, `data.plot.heatmap()`, or `data.plot.scatter()`.
`gsheets`	Google auth and API client libraries	You deploy with the Google Sheets backend (flagged as experimental for public web deployments).
`dev`	ruff, mypy, pytest, plus the chart libraries	You contribute to Siamang itself or run its test suite.

Tip: Quote the bracketed form — `pip install "siamang[charts]"` — so your shell does not interpret the square brackets as a glob pattern.

You may also encounter legacy extra names — `all`, `excel`, `pyreadstat`, `server`, `supabase`, `vercel`, `scipy`. These still resolve so that old install scripts do not break, but they are **empty no-ops**: their dependencies are already bundled with the base package. You do not need them.

If you want to hack on the framework source itself, an editable install from a clone is the standard route:

```
git clone https://github.com/Siamang-Labs/siamang.git
    cd siamang
    pip install -e ".[dev]"
```

Verifying the installation. Two quick checks confirm that the package and the command-line tool are on your path:

```
import siamang as sg

    print(sg.__version__)        # e.g. "0.6.0"
    print(sg.SingleChoice)       # <class 'siamang.core.question.SingleChoice'>

siamang --help
```

For a stronger guarantee, run the smoke test from the installation guide. It builds a one-question survey, validates it, simulates fifty synthetic respondents — no server needed — and prints the resulting dataset:

```
import siamang as sg

    age = sg.Variable("age", scale="ratio", label="Age")
    survey = sg.Questionnaire(
        title="Smoke test",
        pages=[sg.Page("main", items=[sg.NumericInput("How old are you?", var=age)])],
    )
    survey.validate()                # raises if anything is wrong
    data = survey.simulate(n=50)     # synthetic respondents — no server needed
    print(data.frame.head())
```

If this prints a five-row table of simulated ages, your installation is fully functional: the model layer, validation, simulation, and the pandas-backed data layer are all working.

> **Try it yourself:** Install Siamang with the charts extra, then run the smoke test above. Change `n=50` to `n=500` and add a fixed `seed=42` — Chapter 3 explains why the seed makes your synthetic data reproducible.

2.2 The Configuration File

2.2.1 Anatomy of `~/.siamang.toml`

When you move beyond local preview to real deployment, Siamang needs to know two things: which backend (where responses are stored) and frontend (where the survey web page is hosted) to use by default, and the credentials for each. These live in a single TOML file at **`~/.siamang.toml`** in your home directory. TOML is a human-readable configuration format built from `[table]` headers and `key = "value"` pairs.

You create the file once with the `siamang init` command (Section 2.3); `siamang deploy` then reads it on every run. The file has four kinds of top-level tables:

TABLE 2.3

Table	Contents
`[defaults]`	The default backend and frontend names.
`[backends.<name>]`	Keyword arguments forwarded to that backend adapter's constructor (typically credentials).
`[frontends.<name>]`	Keyword arguments forwarded to that frontend adapter's constructor.
`[profiles.<name>]`	Named overrides of `[defaults]`, selected with `siamang deploy --profile <name>`.

Here is a representative file, exactly as documented in the configuration guide:

```toml
# ~/.siamang.toml

    [defaults]
    backend  = "local"
    frontend = "local"

    # Adapter kwargs — forwarded to the backend/frontend constructor.
    [backends.supabase]
    url         = "https://abcdef.supabase.co"
    anon_key    = "eyJ..."
    service_key = "eyJ..."

    [frontends.vercel]
    token        = "vercel-token-..."
    project_name = "political-trust-2026"

    # A named profile, selected with `siamang deploy --profile production`.
    # Keys here override [defaults].
    [profiles.production]
    backend  = "supabase"
    frontend = "vercel"
```

Reading it top to bottom: day-to-day work defaults to the fully local setup (`local` backend, `local` frontend) — the safe choice while you develop. Credential blocks for Supabase and Vercel sit ready but inactive. When you deploy with `--profile production`, the profile overrides the defaults and the deploy uses Supabase and Vercel, picking up their credentials from the matching adapter blocks.

A helpful detail: the local backend and local frontend require no mandatory arguments — the local backend accepts an optional path (default survey.db), and the local frontend optional host, port, and open_browser settings — so a minimal configuration is just `[defaults]` with both set to `"local"` — which is exactly what `siamang init --non-interactive` writes. Adapter credential blocks are shared across all profiles, so you enter each secret once.

You will rarely need to edit this file by hand after `init`, but if you do, keep it out of version control.

> **Warning:** `~/.siamang.toml` contains live credentials. Never commit it to Git, and prefer environment variables over file values on shared or CI machines (see below).

2.2.2 Precedence and security

Environment variables override file values. When Siamang loads the configuration, it overlays matching environment variables onto the adapter credentials from the file — the environment wins. The naming scheme is a prefix per adapter, with the remainder of the variable name (lowercased) becoming the credential key: `SIAMANG_SUPABASE_URL` sets the Supabase backend's `url`, and so on.

TABLE 2.4

Prefix	Overrides credentials for
`SIAMANG_SUPABASE_*`	the Supabase backend
`SIAMANG_GSHEETS_*`	the Google Sheets backend
`SIAMANG_VERCEL_*`	the Vercel frontend
`SIAMANG_NETLIFY_*`	the Netlify frontend
`SURVLIB_*` (same suffixes)	legacy alternative names for each of the above — recognized equally; set only one form of a given variable, as precedence between the two prefixes is undefined

For example:

```
export SIAMANG_SUPABASE_URL="https://abcdef.supabase.co"
    export SIAMANG_SUPABASE_ANON_KEY="eyJ..."
    export SIAMANG_SUPABASE_SERVICE_KEY="eyJ..."
    export SIAMANG_GSHEETS_CREDENTIALS_FILE="./service-account.json"
```

Some adapters also read specific environment variables directly when their credentials are blank — notably `VERCEL_TOKEN` for the Vercel frontend and `NETLIFY_AUTH_TOKEN` for the Netlify frontend — so tokens can stay out of the file entirely. This is the recommended pattern for continuous-integration systems, where the file might not exist at all.

File permissions. Because the file holds secrets, Siamang helps you keep it tight: `siamang init` and the programmatic `save()` set permissions to `0600` (readable and writable only by you) automatically. On POSIX systems, loading the file checks those permissions and logs a warning if the file is group- or world-accessible, suggesting `chmod 600`. If you ever see that warning, fix it immediately:

```
chmod 600 ~/.siamang.toml
```

The working rule of thumb: keep secrets out of version control, prefer environment variables in CI, and reserve the file for your local machine with `chmod 600`.

> **Note:** This chapter covers the file from a user's perspective — what to put in it and how precedence works. The full reference, including the `siamang.config` Python API (`load`, `save`, `use_profile`, `current`) and every method on the `Config` object, is in Appendix B. All current documentation sources describe the file in a single, consistent layout — [defaults] plus [backends.<name>] / [frontends.<name>] blocks with [profiles.<name>] overrides, exactly as shown here. Since version 0.6.0, siamang deploy also loads this file automatically on every run; generating it with siamang init remains safer than hand-editing.

2.3 The Command-Line Interface at a Glance

2.3.1 The four subcommands

The `siamang` command-line tool is the control surface for the pipeline you met in Chapter 1. It has four subcommands, one per operational task:

TABLE 2.5

Subcommand	What it does
`siamang validate <file.py>`	Checks the questionnaire in the given module for structural and logical problems; prints "OK" if well-formed, lists problems otherwise.
`siamang preview <file.py>`	Builds the web frontend and serves it locally (FastAPI + uvicorn) so you can fill the survey in a browser; responses land in `survey.db` (SQLite).
`siamang deploy <file.py>`	Provisions the backend and publishes the frontend according to your configuration (or `--backend`/`--frontend` flags), then prints the public URL.
`siamang init`	One-time interactive setup that creates `~/.siamang.toml` with your defaults and credentials.

A typical session weaves them together in pipeline order:

```
siamang init                                    # one-time: store credentials
    siamang validate my_survey.py                  # structural check
    siamang preview  my_survey.py --port 8000      # try it at http://127.0.0.1:8000
    siamang deploy   my_survey.py --backend supabase --frontend vercel
```

Two discovery conventions make the CLI friendly to learn. First, the CLI locates your questionnaire automatically: it looks for a **module-level variable named `survey`** in the file you pass it, and if you named yours something else, `--attribute` tells it where to look. Second, every level of the tool documents itself — `siamang --help` lists the subcommands, and `siamang <subcommand> --help` lists that subcommand's flags (such as `--port` and `--open` for `preview`, or `--profile` for `deploy`). When in doubt, ask the tool.

> **Try it yourself:** Run `siamang --help` and then `siamang preview --help`. Identify the flags you would use to (a) serve a preview on port 9000 and (b) open it in your browser automatically. You will use both in Chapter 3.

With Siamang installed, verified, and configured, everything is in place for the hands-on core of this book. Chapter 3 builds your first complete survey — from Variable definitions through validation, simulation, preview, and a first analysis — using exactly the commands you have just seen.

Chapter Summary

- Siamang requires Python 3.11+ (3.11–3.13 tested) on POSIX or Windows; install with `pip install siamang`, or from Git for the development state.
- The base package bundles its analysis, preview, deployment, and I/O dependencies; only charts are truly optional.

- Three extras matter: `[charts]` (matplotlib/seaborn for `data.plot.*`), `[gsheets]` (Google Sheets backend), and `[dev]` (ruff, mypy, pytest). Legacy extras still resolve but are empty no-ops.
- Verify with `sg.__version__`, `siamang --help`, and the validate-plus-simulate smoke test.
- Deployment settings live in `~/.siamang.toml` with four table types: `[defaults]`, `[backends.*]`, `[frontends.*]`, and `[profiles.*]`; create it once with `siamang init`.
- SIAMANG_* environment variables override file values (legacy SURVLIB_* names are recognized as well); `VERCEL_TOKEN` and `NETLIFY_AUTH_TOKEN` are read directly by their adapters, keeping tokens out of the file.
- `init` and `save()` set file permissions to `0600`; fix any group/world-accessible config file with `chmod 600` and never commit it.
- The CLI has four subcommands — `validate`, `preview`, `deploy`, `init` — and discovers your questionnaire via the module-level `survey` variable; `--help` documents every flag.

Documentation References

- `wiki/Installation.md` — Python version support, install commands, extras (including legacy no-ops), dependencies table, editable install, verification and smoke test.
- `README.md` — Requirements, Dependencies table, install command, license contact, cloud deployment account requirements, charts error message note.
- `docs/getting-started.md` — Git install, no-op legacy extras, `--attribute` override, local preview and `survey.db`.
- `wiki/Configuration.md` — `~/.siamang.toml` format, the four table types, environment override prefixes, profiles, and permissions/security guidance.
- `MANUAL.md` — environment variables list, preview flags, `LICENSE-COMMERCIAL.md` pointer.
- `wiki/Quickstart.md` — CLI lifecycle commands and the module-level `survey` variable convention.
- `CHANGELOG.md` — release context for the current Beta and `SIAMANG_*`/`SURVLIB_*` environment naming.

Chapter 3. Quickstart: Your First Survey

Learning Objectives

By the end of this chapter, you will be able to:

- Write a complete Siamang survey module from scratch — Variables, Questions, a Page, and a Questionnaire.
- Validate the questionnaire and interpret the validator's output.
- Preview the survey in your browser and know where the responses are stored.
- Generate a synthetic dataset with `simulate()` and produce your first SPSS-like report.
- Read responses back from the local SQLite store.
- Navigate the remaining parts of this textbook and know which chapters to read for your goals.

3.1 A Minimal End-to-End Example

3.1.1 Defining variables, questions, and a page in code

Everything you build in Siamang follows one authoring flow, stated in the quickstart guide as four layers: define **Variables**, bind them to **Questions**, place those on a **Page**, and assemble a **Questionnaire** — then `validate`, `simulate`, `preview`, and `deploy`. Let us walk through that flow once, end to end, with the smallest survey that still shows every layer.

Create a file named `hello.py` with exactly this content:

```
import siamang as sg

    age = sg.Variable("age", scale="ratio", label="Age")
    fav = sg.Variable(
        "fav_color", scale="nominal",
        label="Favourite colour",
        labels={1: "Red", 2: "Blue", 3: "Green"},
    )

    survey = sg.Questionnaire(
        title="Hello, siamang",
        pages=[
            sg.Page(
                name="main",
                title="Tell us a bit about yourself",
                items=[
                    sg.NumericInput("How old are you?", var=age, required=True),
                    sg.SingleChoice("What is your favourite colour?", var=fav),
                ],
            ),
        ],
    )
```

This listing rewards a careful top-to-bottom reading, because every survey you ever write will have the same skeleton:

1) **Import the library as `sg`.** The convention throughout the documentation is `import siamang as sg`, and this book follows it everywhere.
2) **Define Variables.** `age` is a ratio-scale Variable (a true-zero numeric quantity) with the label "Age". `fav_color` is nominal (unordered categories) and carries *value labels* — the mapping `{1: "Red", 2: "Blue", 3: "Green"}` that turns stored codes into readable text in every table and export later. This metadata is not decoration; it is what makes Siamang's outputs labeled and publication-ready without further work.
3) **Bind Variables to Questions.** `sg.NumericInput("How old are you?", var=age, required=True)` ties the prompt to the `age` Variable and makes an answer mandatory. `sg.SingleChoice("What is your favourite colour?", var=fav)` does the same for the color question; its answer options come from the Variable's labels.
4) **Place Questions on a Page.** `sg.Page(name="main", title=..., items=[...])` is one screen of the survey, holding the two questions in order.
5) **Assemble the Questionnaire.** `sg.Questionnaire(title=..., pages=[...])` is the aggregate root — the object every later stage operates on.

One convention matters more than any other at this point: the module-level variable is named **`survey`**, deliberately. The CLI discovers your questionnaire by looking for a module-level variable with that name;

if you ever choose a different name, the `--attribute` flag tells the CLI where to look. Sticking with `survey` keeps every command in this book working verbatim.

3.1.2 Running `siamang validate` and `siamang preview`

With `hello.py` saved, the first two stages of the pipeline are one command each.

Step 1 — validate. Structural and logical checking before anything goes near a respondent:

```
siamang validate hello.py
```

A well-formed questionnaire prints a one-line confirmation:

```
OK — no warnings.
```

If something is wrong — a duplicate question ID, a navigation target that does not exist, an expression referencing an unknown variable — the validator lists the problem instead. The same check is available in Python as `survey.validate()`, which raises `ValueError` on the first problem and returns `None` when the questionnaire is well-formed; the separate `lint()` method surfaces softer warnings such as empty pages, which the command line prints line by line in the form `[warning] [code] message`; `strict=True` goes further: it runs the strict lint level and raises `ValueError` for its error-severity findings, such as empty pages or categorical Variables without labels. Get into the habit of validating after every edit — it is the cheapest feedback loop in the whole pipeline.

Step 2 — preview. See the survey exactly as a respondent will:

```
siamang preview hello.py --port 8000
```

This builds the web frontend, serves it with FastAPI and uvicorn, and prints a local address — open `http://127.0.0.1:8000` in your browser and fill the survey in. Every response you submit lands in **`survey.db`**, a SQLite database in the working directory, so the preview is not a mock-up: it is the real instrument writing real data to a local store. Add `--open` to launch the browser automatically.

> **Tip:** Preview is also a design tool. Because the frontend is compiled from the same module you just validated, anything that looks wrong in the browser is fixed in `hello.py` — never in a separate design file — and re-previewed within seconds.
>
> **Try it yourself:** Run the preview and submit three test responses with deliberately different answers — including one attempt to skip the required age question. Watch how the

runtime enforces `required=True`, then keep `survey.db` around: you will read it back in Section 3.2.

3.2 From Definition to Data

3.2.1 Deploying locally, collecting responses, and a first look at SurveyData

You already have data without deploying anything: the preview server stored your test responses in `survey.db`. "Deploying locally" in Siamang means exactly this arrangement — the `local` backend (SQLite) and `local` frontend — which is also the default in a fresh `~/.siamang.toml`. Hosted deployment to a public URL is a one-flag change (`siamang deploy hello.py --backend supabase --frontend vercel`), and Chapter 1's pipeline diagram shows where it sits; this book dedicates Part III to it. For now, the local store is enough to complete the loop from definition to data.

Reading the responses back. The local backend exposes stored responses through a small Python API; the survey ID is printed once when the preview server starts, on the survey_id: line under the Preview ready at … message:

```
from siamang.deploy.backends.local import LocalBackend

    backend = LocalBackend(path="survey.db")
    df = backend.get_responses(survey_id="<id from server log>")
```

The result is a pandas DataFrame — one row per respondent, one column per Variable name, plus two bookkeeping columns added by the backend: _response_id and _submitted_at.

Rehearsing with simulated data. You do not need to wait for respondents to develop your analysis. `simulate()` generates a synthetic dataset that respects each Variable's scale, labels, and ranges — defaults are `simulate(n=100, seed=42)`, one hundred rows with a fixed random seed, and `seed=None` gives fresh randomness on each run:

```
from hello import survey

    data = survey.simulate(n=500, seed=42)
    print(data.frame.head())
```

SurveyData: your first look. What `simulate()` returns — and what collected responses load into downstream — is a **`SurveyData`** object: a pandas `DataFrame` (accessed as `data.frame`) wrapped together with your Variable metadata. That pairing is the heart of Siamang's analysis layer. Because the object knows that `fav_color` is nominal with labels Red/Blue/Green, a frequency table needs no further annotation:

```
print(data.report.freq("fav_color").to_markdown())
```

The `data.report` accessor is the declarative, SPSS-like reporting layer: tables come out with value labels attached and, where appropriate, statistical tests already computed — `data.report.crosstab("gender", "party")`, for instance, adds a Chi-square test and Cramér's V automatically. Charts live one accessor over (`data.plot.bar(...)`) and require the `[charts]` extra from Chapter 2. And when the analysis is done, `SurveyData.export()` writes the data to CSV, Excel, SPSS, Stata, or R — but only the SPSS and Stata formats embed the full metadata round-trip (variable labels, value labels, missing-value conventions); CSV and Excel carry data only, so pair them with the JSON dictionary to preserve the codebook, and the R export is a directory containing a CSV file, a JSON dictionary, and an .R loader script:

```
data.export("csv",   path="hello.csv")
    data.export("xlsx",  path="hello.xlsx")
    data.export("spss",  path="hello.sav")
    data.export("stata", path="hello.dta")
    data.export("r",     path="hello_R/")   # CSV + JSON dict + .R loader
```

That is the entire promise of Chapter 1 made concrete: one plain Python module carried through validate, preview, collect, and analyze, with metadata intact at every step.

> **Warning:** Simulated data is for rehearsing your analysis code, never for drawing conclusions. It is generated to be structurally plausible (right scales, labels, ranges), not substantively meaningful. Always re-run your analysis scripts against real collected data before reporting results.

> **Try it yourself:** Extend `hello.py` with a third Variable — an ordinal `satisfaction` Variable with labels 1 to 5 — and a `sg.LikertScale` question bound to it. Re-validate, re-preview, then simulate 500 respondents and print `data.report.freq("satisfaction").to_markdown()`. You have just repeated the exact workflow used for real studies.

3.3 Where to Go Next

3.3.1 Roadmap through the rest of the textbook

You now hold the complete loop: write a module, validate it, preview it, collect data, and analyze. The rest of this book deepens each stage, organized into parts that mirror the pipeline:

- **Part II — Authoring surveys.** The mental model (five layers, the model tree, immutability), then Variables and measurement in depth, all seven question types, pages, blocks, and navigation, visibility and branching logic (`show_if`/`hide_if`, Expressions), quotas, lifecycle scripts, theming with `UIConfig`, and validation as a quality gate. Read this part cover to cover if you design instruments.
- **Part III — Deployment and data.** Configuration beyond the basics, hosted backends and frontends, collecting responses, simulating respondents, and the full import/export system for CSV, Excel, SPSS, Stata, and R.
- **Part IV — Analysis and reporting.** The `SurveyData` object properly, the `data.report` and `data.plot` accessors, low-level statistics via `data.analysis`, recoding and derived variables, banner tables, weights, and composable `Report` documents.
- **Part V — Siamang Cloud.** The hosted platform at `app.siamang.org`: browser-based authoring, one-click deploy, dashboards, team collaboration, and plans — everything the managed service adds around the engine you already know.
- **Appendices.** Reference material to consult rather than read: the full configuration reference (Appendix B) and other lookup chapters alongside it.

If you are in a hurry, the shortest path is Part II's mental-model chapter next — it explains *why* the four-layer flow you just used is shaped the way it is — followed by the question-types chapter, and then straight to whatever Part III or IV chapter matches your current task. The book's cross-references assume you have read Chapters 1–3; everything else is designed to be entered directly.

For a complete worked example beyond this quickstart, the repository ships `examples/full_pipeline/`, a Jupyter notebook with twelve variables, five pages, conditional routing, matrix questions, 300 synthetic respondents, frequency tables, cross-tabs, correlation heatmaps, and charts. Run it with:

```
cd examples/full_pipeline
    jupyter notebook full_pipeline_demo.ipynb
```

Chapter Summary

- Authoring follows four layers — Variables, Questions, Page, Questionnaire — in one plain Python module, conventionally exposing a module-level `survey` variable.
- Variable metadata (scale, labels) is declared once and powers labeled outputs through the entire lifecycle.

- `siamang validate` (or `survey.validate()`) checks structure and logic before fielding; lint() surfaces softer warnings; strict=True promotes strict-level findings to validation errors.
- `siamang preview hello.py --port 8000` serves the real survey locally at `http://127.0.0.1:8000`, storing responses in `survey.db` (SQLite).
- Responses are read back via `LocalBackend(path="survey.db").get_responses(...)`; `simulate(n=..., seed=42)` generates reproducible synthetic data for analysis rehearsal.
- `SurveyData` pairs a pandas `DataFrame` with Variable metadata; `data.report.freq(...).to_markdown()` is the first taste of SPSS-like reporting.
- `SurveyData.export()` writes CSV, Excel, SPSS, Stata, and R; full metadata round-trips only in SPSS/Stata, CSV/Excel carry data only (labels via the JSON dictionary), and R export is CSV + dictionary + .R loader.
- The rest of the book follows the pipeline: Part II authoring, Part III deployment and data, Part IV analysis and reporting, Part V Siamang Cloud, plus reference appendices.

Documentation References

- `wiki/Quickstart.md` — the four-layer authoring flow, `hello.py` walkthrough, validate/simulate/preview/deploy/export steps, `simulate()` defaults.
- `docs/getting-started.md` — the minimal survey module, CLI lifecycle, module-level `survey` convention, export examples, SPSS round-trip check.
- `MANUAL.md` — “Hello, siamang” listing, validation and lint details, simulation semantics, analysis examples.
- `README.md` — canonical pipeline commands, quick-start variant, reporting examples, Full Pipeline Example description.
- `wiki/Installation.md` — smoke test pattern reused for verification.
- `docs/concepts.md` and `wiki/Core-Concepts.md` — the five-layer model behind the `SurveyData` first look and backend/frontend inventory.

Part II — Authoring Surveys

Chapter 4. Core Concepts and the Mental Model

Learning Objectives

By the end of this chapter, you will be able to:

- Describe the five layers of the Siamang architecture and the responsibility of each one.
- Explain why every building block in `siamang.core` is an immutable (frozen) dataclass, and why that matters for reproducible research.
- Name the objects of the authoring model — Variable, Question, Page, Block, and Questionnaire — and describe how they nest into a single tree.
- Distinguish the role of a Variable (the definition of a measurement) from an answer (one respondent's value).
- Use a VariableMap as the central registry and codebook for a survey, and explain when Siamang builds one for you automatically.
- Trace how the same Python module flows through validation, simulation, preview, deployment, and analysis.

4.1 The Five Layers

When you ran the quickstart in Chapter 3, you typed four commands — `validate`, `simulate` (in Python), `preview`, and `deploy` — against one small Python module. Behind those commands, Siamang is organized as five layers, each with a strict responsibility (docs/concepts.md; wiki/Core-Concepts.md). Understanding the layers is the single most useful piece of mental scaffolding you can build, because every feature in the rest of this book belongs to exactly one of them.

TABLE 4.1

Layer	Module	Responsibility
1. Model	`siamang.core`	The questionnaire as a tree of immutable dataclasses.
2. Frontend	`siamang.frontend`	Compile the model to a schema and a self-contained bundle.
3. Deploy	`siamang.deploy`	Provision a backend, publish the bundle, return a URL.
4. Data	`siamang.data`	Wrap a pandas `DataFrame` plus a VariableMap with analysis accessors.
5. I/O	`siamang.io`	Round-trip data and metadata with CSV, Excel, SPSS, Stata, and R.

4.1.1 Core model, frontend bundle, deploy pipeline, data analysis, I/O round-trip

Layer 1 — the core model. This is where you spend your time as an author. Everything you write in a survey module — Variables, Questions, Pages, Blocks, Expressions, Quotas, Scripts — is a Python object from `siamang.core`. The Questionnaire is the aggregate root (the top-level object that owns everything else), and it exposes the verbs you already know: `validate()`, `lint()`, `simulate()`, `compile()`, and `deploy()`. Layer 1 knows nothing about browsers, servers, or file formats; it is a pure, in-memory description of a study.

Layer 2 — the frontend bundle. A Questionnaire object cannot be shown to a respondent directly. Layer 2 compiles it into a SurveySchema — a frozen, platform-agnostic snapshot of the survey — and then into a SurveyBundle: a self-contained set of static files (`index.html`, `style.css`, `env.js`, and friends) that any web server can host. A `FrontendBuilder` combines three ingredients to do this: a runtime adapter (the bundled SurveyJS or React 18 runtime), a UIConfig (the theme and branding you will meet in Chapter 10), and a backend client template that tells the browser where to send responses. You rarely call this layer by hand — `siamang preview` and `survey.deploy(...)` drive it for you — but it is why a survey you preview locally looks and behaves identically to the one you deploy publicly.

Layer 3 — the deploy pipeline. Deployment is an orchestrated sequence: validate the model, provision a backend (the place responses are stored), build the frontend bundle, and publish it to a frontend host. The result is handed back to you as a `DeployResult`, which carries the public URL and lets you pull accumulated responses at any time with `result.collect()`. Siamang bundles three backends (`local`, `supabase`, `gsheets`) and three frontends (`local`, `vercel`, `netlify`), and discovers additional ones through Python entry points if you install plugins.

Layer 4 — data analysis. When responses come back — from `survey.simulate(...)`, from `result.collect()`, or read from a file — they arrive as a `SurveyData`: a pandas `DataFrame` of answers joined to the VariableMap that defined them. SurveyData gives you accessor objects rather than loose functions: `data.report` and `data.plot` for declarative, SPSS-like tables and charts, `data.analysis` for low-level statistics, `data.processing` for recodes and derived variables, and `data.tables` for banner tables. Part III of this book is devoted to this layer.

Layer 5 — the I/O round-trip. The final layer moves data and metadata in and out of the statistical packages you already use. Readers and writers exist for CSV, Excel (.xlsx), SPSS (.sav), and Stata (.dta), plus a JSON dictionary format; for R there is a writer only — a CSV/JSON dictionary bundle plus an R loader script. The full metadata round-trip — variable labels, value labels, and missing-value conventions preserved in both directions — belongs to SPSS and Stata: a .sav file produced by Siamang opens in SPSS exactly as you defined it, and a .sav you import comes back with its metadata intact. CSV and Excel carry data only; pair them with the JSON dictionary (DictionaryWriter/DictionaryReader) to preserve the metadata.

The key insight is that *the same module flows through every stage*. You never export your questionnaire to another tool and lose the connection; the Variable you define in layer 1 is the column header in layer 4 and the variable label in the SPSS file in layer 5.

> **Tip:** When you are unsure where a Siamang feature lives, ask which layer it serves. Question text, branching, and quotas are authoring (layer 1). Colors and button labels are UIConfig (layer 2). Credentials and hosts are deployment (layer 3). Recodes, weights, and tests are analysis (layer 4).

4.2 Immutable Building Blocks

Every building-block class of the model in siamang.core is declared as a frozen Python dataclass — `@dataclass(frozen=True, slots=True)` (docs/reference/core.md). Frozen means that once an object is constructed, its fields cannot be reassigned; `slots=True` means instances carry no per-object dictionary, so accidental new attributes cannot be attached either. This applies to the model classes uniformly: Variable, Question and its seven subclasses, Page, Block, Option, Media, Expression, Quota, Script, and the Questionnaire itself are all immutable after instantiation. The deliberate exception is VariableMap, the registry you will meet in §4.3.2 — a mutable, specialized dictionary subclass rather than a frozen dataclass.

```
import siamang as sg

    age = sg.Variable("age", scale="ratio", label="Age")
    # age.label = "Age in years"      # raises FrozenInstanceError
```

4.2.1 Why immutability matters for reproducibility

This is not an implementation detail; it is a design guarantee, and the documentation states it plainly: immutability keeps survey definitions "side-effect free, thread-safe, and deterministic across compilation, simulation, and analysis" (wiki/Core-Concepts.md; docs/concepts.md). Three practical consequences follow.

First, **a definition cannot change underneath you**. When you reference the same Variable from several expressions and analyses, or pass a Block into a Page, you are sharing a reference — but no code path can mutate that shared object. (Note that a Variable may be bound to at most one Question — `validate()` rejects duplicates.) The questionnaire you validated is bit-for-bit the questionnaire that gets compiled, deployed, and used to label your analysis. There is no class of bugs where "the survey changed between preview and deploy."

Second, **determinism becomes testable**. Because `simulate(n=100, seed=42)` starts from an immutable model and a fixed seed, it produces the same synthetic dataset on every run, on every machine. You can commit an expected frequency table to your test suite and know it will still pass after a refactor of the questionnaire code.

Third, **version control becomes meaningful**. A survey that cannot be mutated at runtime can only be changed by editing the source file. Every change is therefore a diff in Git: reviewable in a pull request, attributable to an author and a date, and reversible with a rollback. This is the heart of the "research-as-code" philosophy from Chapter 1.

When you genuinely need a variant of an object — say, a theme preset with your institution's colors — you do not mutate; you derive a new object with `dataclasses.replace`, which copies the frozen instance and overrides the fields you name:

```
import dataclasses
    from siamang.frontend import get_preset

    ui = dataclasses.replace(
        get_preset("modern"),
        institution_name="Independent Polling Lab",
        primary_color="#a8324b",
    )
```

Note: If you catch yourself wanting to append a question to a page after the fact, stop and rebuild instead. The idiomatic Siamang pattern is to construct the complete tree bottom-

up in one pass — Variables first, then Questions, then Pages, then the Questionnaire — exactly as the quickstart in Chapter 3 did.

4.3 The Authoring Object Model

Layer 1 deserves a closer look, because it is the vocabulary of the entire authoring part of this book. Siamang models a survey as a tree (docs/concepts.md; wiki/Core-Concepts.md):

```
Questionnaire
    ├── pages: list[Page]
    │       ├── items: list[Question | Block]
    │       └── show_if / hide_if : Expression | str | None
    └── variables: VariableMap
```

4.3.1 Variable, Question, Page and Block, and the Questionnaire aggregate root

The tree is built from five kinds of objects, each with a distinct role (docs/concepts.md; wiki/Core-Concepts.md):

TABLE 4.2

Object	Role
`Variable`	An atomic measurement unit: `name`, `scale`, `label`, `labels`, `missing_values`, and more.
`Question` (and subclasses)	Binds a respondent-facing prompt to one variable — or to many, for `Matrix` and wide-mode `MultiChoice`.
`Page`	One screen in the survey; carries its own visibility gate and navigation rules.
`Block`	A named container of questions (and nested blocks) inside a page; can be randomized or hidden as a unit (applies to top-level blocks; nested blocks are flattened into their parent at compile time — see §5.2.1).
`Questionnaire`	The aggregate root; owns the pages and provides `validate()`, `compile()`, `simulate()`, and `deploy()`.

The **Variable** is the atomic unit. The most important distinction in the whole mental model is this one: a Variable carries the *definition* of a measurement — its name, its measurement scale (nominal, ordinal, interval, or ratio), its value labels, and its missing-value conventions — while an *answer* is one respondent's value, stored as a row in the collected `DataFrame` under the column named by `variable.name` (MANUAL.md; wiki/Core-Concepts.md). You define Variables once; answers accumulate later, at collection time, and are validated and labeled against the definitions.

A **Question** is what the respondent actually sees: a prompt text bound to a Variable via its `var=` field. Siamang ships seven concrete types — `SingleChoice`, `MultiChoice`, `LikertScale`, `NumericInput`, `OpenText`, `Matrix`, and `Ranking` — and Chapter 6 covers each in depth. Most bind ex-

actly one Variable; `Matrix` binds one Variable per row, and a wide-mode `MultiChoice` binds one per option. Two helper objects enrich questions: `Option`, a single answer choice that can carry its own visibility condition or media attachment, and `Media`, an image, video, or audio clip attached to a prompt or an option.

Pages and Blocks provide structure. A Page is one screen shown to the respondent; it has a unique `name` used as a branching target, an ordered `items` list, and optional routing fields (`next_if`, `default_next`). A Block groups questions inside a page — to apply one visibility rule to several items at once, to place a shared title above them, or to randomize their order. Blocks nest, so you can build sections within sections.

The **Questionnaire** is the aggregate root — the single object that owns the whole tree and that the CLI discovers as the module-level variable named `survey`. Two structural rules are worth memorizing now (wiki/Pages-Blocks-and-Structure.md): you pass either `pages=[...]` for a multi-page survey or `blocks=[...]` for a quick single-page survey, never both (passing both raises `ValueError`); and the Questionnaire is also where cross-cutting concerns attach — `variables=` for an explicit registry, `scripts=` for lifecycle JavaScript, and `deadline=` for a closing date.

Two more objects hang off the tree without being part of the respondent-facing flow. An **Expression** is the typed condition language for visibility and routing — `age.ge(18) & gender.eq(2)` — that attaches to `show_if`/`hide_if` fields at the Page, Block, Question, and even Option level. A **Quota** caps the number of accepted responses in a category; notably, quotas are attached at deploy time (`survey.deploy(..., quota=quotas)`), not stored on the Questionnaire, because they describe fielding policy rather than instrument design. Chapters 7 and 8 treat both in detail.

Here is the object model in one compact, complete example — the same shapes you will use for the rest of the book:

```python
import siamang as sg

    # 1. Variables: the measurement definitions
    age = sg.Variable("age", scale="ratio", label="Age", valid_range=(18, 99))
    trust = sg.Variable(
        "trust", scale="ordinal", label="Trust in government",
        labels={1: "No trust", 2: "Low", 3: "Medium", 4: "High", 5: "Full trust"},
    )

    # 2. Questions: prompts bound to variables
    q_age = sg.NumericInput("How old are you?", var=age, required=True)
    q_trust = sg.LikertScale(
        "How much do you trust the government?", var=trust, points=5,
        left_label="No trust", right_label="Full trust",
    )

    # 3. Structure: a block of questions on a page
    attitudes = sg.Block(title="Attitudes", items=[q_trust], show_if=age.ge(18))

    # 4. The aggregate root
    survey = sg.Questionnaire(
        title="Political Trust — 2026",
        pages=[sg.Page(name="main", items=[q_age, attitudes])],
    )

    survey.validate()
    data = survey.simulate(n=100, seed=42)
```

Notice the direction of construction: bottom-up, from atoms to aggregate. Each level only ever references objects you have already built, which is what makes the tree easy to reason about — and easy for `validate()` to check exhaustively (unique question ids and page names, valid navigation targets, expressions referencing only known variables, and more).

> **Try it yourself:** Take the example above and restructure it without running any new commands. Move `q_age` and `q_trust` onto two separate pages named `"demographics"` and `"attitudes"`, with the second page gated by `show_if=age.ge(18)`. Then predict: what does `len(survey.all_questions())` return, and will `survey.validate()` still pass? Check your prediction in a Python session.

4.3.2 The VariableMap registry and the codebook idea

One node of the tree sits slightly apart: `variables: VariableMap`. A VariableMap is a `dict[str, Variable]` subclass — a registry of every Variable in the survey, indexed by name (wiki/Core-Concepts.md; wiki/Variables-and-Measurement.md). It plays two roles at once.

First, it is the **bridge between model and data**. When responses arrive, they are plain columns in a pandas `DataFrame`; the VariableMap is what tells Siamang that the `trust` column is ordinal, that code

`5` means “Full trust”, and that code `9` (if you declared it) is a missing value rather than a substantive answer. Every reporting table, chart, and export in Parts III and IV reads its labels from this registry. You can also validate a raw frame against the schema directly with `vm.validate_frame(data.frame)`, which reports issues such as missing columns, out-of-range values, or codes with no label.

Second, it is your **codebook** — the machine-readable documentation of the study. Fields such as `label`, `description`, `construct` (the latent construct a variable measures), and `source` (the provenance of the question wording, e.g. `"ESS Wave 10"`) exist to document what each column means, and they travel with the data: `codebook()` includes the `label` and `description` columns, and on SPSS/Stata export `label` becomes the variable label. `construct` and `source` are not shown by `codebook()` but are preserved in `VariableMap.to_dict()`. A well-populated VariableMap means your dataset is self-describing.

You can build a registry explicitly and query it:

```
from siamang.core import VariableMap, Variable

    vm = VariableMap()
    vm.add_many([
        Variable("age", scale="ratio", label="Age"),
        Variable("gender", scale="nominal", label="Gender", labels={1: "M", 2: "F"}),
    ])
    vm.require("age")          # lookup; raises KeyError if absent
    vm.by_scale("nominal")     # [Variable('gender', ...)]
    issues = vm.validate_frame(data.frame)   # validate a DataFrame against the schema
```

Crucially, you do not have to. If you pass `Questionnaire(variables=VariableMap([...]))`, that registry is the source of truth — and validate() then enforces that every question variable matches the definition registered there — the comparison is by value, field by field, not by object identity. If you omit it, Siamang **auto-builds** the VariableMap by walking every question in the tree (wiki/Core-Concepts.md; docs/concepts.md). The practical rule of thumb: let Siamang infer the registry while you are drafting, and pass an explicit one when your study matures — for example, when several questionnaire modules in a panel study must share one canonical codebook.

> **Warning:** Because a VariableMap is a registry keyed by `name`, registering two different Variables under the same name is an error (`add()` raises `KeyError`), and with an explicit registry, a question whose Variable is not equal to the registered one fails validation (two separate but identical definitions pass). Build each Variable once, at module level, and reuse that single object everywhere — a good habit that keeps the registry tidy, even though validation itself compares definitions by value.

Try it yourself: Build the survey from §4.3.1, then construct an explicit `VariableMap` containing only `age` and pass it via `Questionnaire(..., variables=vm)`. Run `survey.validate()` and observe the registry-consistency error for `trust`. Fix it by adding `trust` to the map, and confirm validation passes. You have just experienced exactly what the auto-built registry saves you from during drafting.

Chapter Summary

- Siamang is organized into five layers with strict responsibilities: the core model (`siamang.core`), the frontend bundle (`siamang.frontend`), the deploy pipeline (`siamang.deploy`), data analysis (`siamang.data`), and the I/O round-trip (`siamang.io`).
- The same Python module flows through every layer: the Variable you define is the survey question, the dataset column, and the SPSS variable label.
- Every building-block class of the model is a frozen, slotted dataclass (VariableMap, the mutable registry dictionary, is the exception); immutability makes definitions side-effect free, thread-safe, and deterministic across compilation, simulation, and analysis — the technical foundation of reproducible research-as-code.
- To derive a variant of a frozen object, use `dataclasses.replace(...)`; never attempt mutation.
- The authoring model is a tree: Variables are atomic measurement definitions, Questions bind prompts to variables, Pages are screens, Blocks group questions within pages, and the Questionnaire is the aggregate root with `validate()`, `compile()`, `simulate()`, and `deploy()`.
- A Variable defines a measurement; an answer is one respondent's value in a DataFrame column named after the variable.
- A VariableMap is the central registry and codebook — the bridge between model and data. Pass one explicitly for a canonical schema, or let Siamang auto-build it by walking the questions.
- Quotas attach at deploy time, not on the Questionnaire, because they describe fielding policy rather than instrument design.

Documentation References

- `docs/concepts.md` — the five-layer architecture, the model tree, and object roles.
- `wiki/Core-Concepts.md` — research-as-code philosophy, immutability guarantees, the model tree, VariableMap, and variables versus answers.

- `docs/reference/core.md` — frozen dataclass declarations and the canonical import surface for `siamang.core`.
- `wiki/Pages-Blocks-and-Structure.md` — the Questionnaire aggregate root, `pages` versus `blocks`, and validation responsibilities.
- `wiki/Variables-and-Measurement.md` — Variable fields, VariableMap methods, and the codebook concept.
- `MANUAL.md` ("Variables"; "Pages, blocks, and navigation") — Variable argument reference and structural examples.
- `wiki/Frontend-and-Theming.md` — SurveySchema, SurveyBundle, FrontendBuilder, and the `dataclasses.replace` pattern for frozen UIConfig objects.
- `wiki/Quotas.md` — quotas as deploy-time attachments rather than questionnaire fields.

Chapter 5. Structuring Questionnaires: Pages and Blocks

Learning Objectives

By the end of this chapter, you will be able to:

- Construct `Page` objects with names, titles, items, and navigation fields, and explain what each field controls.
- Choose correctly between the `pages=[...]` and `blocks=[...]` modes of a `Questionnaire` — and predict the error when they are mixed.
- Use the four page factory helpers — `ContentPage`, `DisqualificationPage`, `FinalPage`, and `RedirectPage` — to build intro, screen-out, thank-you, and redirect screens.
- Distinguish terminal from non-terminal pages using `is_terminal`, and gate terminal pages with `show_if`.
- Group questions into `Block` containers to share a title, apply bulk visibility rules, and randomize item order.
- Flatten a structured questionnaire back into an ordered list of questions with `all_questions()` and `flatten_questions()`.

5.1 Pages

In Chapter 4 you learned that a Questionnaire is a tree whose respondent-facing nodes are Pages. A **Page** is one screen shown to the respondent — the unit of navigation, progress display, and (as you will see in Chapter 7) a major attachment point for visibility logic. This chapter covers the concrete API for building pages and for grouping their contents into Blocks.

5.1.1 Page anatomy; the pages-XOR-blocks rule

`Page` is a frozen dataclass in `siamang.core` (wiki/Pages-Blocks-and-Structure.md). Its full field set:

TABLE 5.1

Field	Type	Default	Description
`name`	`str`	*required*	Unique page identifier; used as a branching/skip target.
`title`	`str | None`	`None`	Title shown at the top of the page.

Field	Type	Default	Description
`items`	`list[Question | Block]`	`[]`	Questions and blocks rendered on the page.
`next_if`	`list[tuple[str, str]]`	`[]`	Conditional routing: `[(condition, target_page_name), ...]`, evaluated in order.
`default_next`	`str | None`	`None`	Fallback next page when no `next_if` matches.
`randomize_blocks`	`bool`	`False`	Shuffle the order of immediate `Block` items.
`show_if`	`Expression | str | None`	`None`	Render the page only when this is true.
`hide_if`	`Expression | str | None`	`None`	Hide the page when this is true.
`kind`	`str | None`	`None`	Page kind (see §5.1.2); `None` is an ordinary question page.
`body`	`str | None`	`None`	HTML body for content/terminal pages.
`redirect_url`	`str | None`	`None`	Target URL for terminal pages that redirect.
`redirect_delay`	`int | None`	`None`	Seconds before redirecting (runtime default 5).

Three conventions are worth internalizing. First, **`name` is the page's address**: every next_if and default_next target in the questionnaire is a page name; skip_to additionally accepts a question id, so names must be unique — `Questionnaire.validate()` enforces this, along with the existence of every navigation target, the reachability of every page from the first page, and the absence of cycles in the navigation graph. Second, **`items` mixes Questions and Blocks freely** in the order they should appear on screen. Third, **navigation defaults to sequential**: if no `next_if` rule matches and no `default_next` is set, the respondent advances to the next page in the `pages` list. Conditional routing itself is the subject of Chapter 7; here you only need to know where the fields live.

A typical multi-page skeleton looks like this:

```
import siamang as sg

    survey = sg.Questionnaire(
        title="Political Trust — 2026",
        pages=[
            sg.Page(name="welcome", title="Welcome", items=[q_consent]),
            sg.Page(name="main",    title="About you", items=[q_age, q_gender]),
            sg.Page(name="attitudes", items=[attitudes_block],
                    show_if=age.ge(18), default_next="thanks"),
            sg.Page(name="thanks",  items=[q_comment]),
        ],
    )
```

Two utility members help you work with pages programmatically. `Page.flatten_questions()` returns all questions on the page, recursing into nested blocks; and `Questionnaire.all_questions()` flattens every page and block of the whole survey into one ordered list — handy for counting items or checking coverage:

```
print(len(survey.all_questions()))   # every question, in display order
```

The pages-XOR-blocks rule. The Questionnaire accepts its content through exactly one of two fields (wiki/Pages-Blocks-and-Structure.md):

TABLE 5.2

Field	Content	Use case
`pages`	`list[Page]`	Multi-page surveys — the common case.
`blocks`	`list[Question | Block]`	Quick surveys without explicit pages; bare questions render on one screen, while each Block in the list becomes its own page.

You must use one **or** the other, never both: passing both raises `ValueError` at construction. Most surveys use `pages`; reach for blocks only when a quick form without explicit page structure is all you need:

```
quick = sg.Questionnaire(
        title="One-question poll",
        blocks=[sg.SingleChoice("Do you approve?", var=approval, required=True)],
    )
```

Warning: An empty page — `Page(name="p2", items=[])` — does not fail validation, but linting flags it with an `EMPTY_PAGE` warning (promoted to an error at strict level). If a page exists only as a navigation placeholder, give it content or remove it. Chapter 11 covers validation and linting in full.

5.1.2 Special page kinds: ContentPage, DisqualificationPage, FinalPage, RedirectPage

Not every screen in a survey holds questions. Real studies need consent forms, screen-outs for ineligible respondents, thank-you screens, and redirects back to a panel provider. Siamang models these through the `kind` field of `Page`, which selects a rendering mode; constructing a `Page` with an unknown kind raises `ValueError` (wiki/Pages-Blocks-and-Structure.md).

TABLE 5.3

`kind`	Terminal?	Behavior
`None`	no	Ordinary question page (default).
`"content"`	no	Renders arbitrary HTML (`body`) instead of questions — intro/consent etc. Stays in normal Next/Prev flow.
`"disqualification"`	yes	Terminal screen-out; records the response as screened-out.
`"final"`	yes	Terminal custom thank-you screen (`body`).
`"redirect"`	yes	Terminal screen that redirects to `redirect_url` (e.g. a panel completion URL).

The distinction that matters is **terminal versus non-terminal**. Terminal pages — disqualification, final, redirect — end the survey when reached, which is why they are typically gated by `show_if` (a disqualification page should only appear to respondents who fail the screener). Content pages, by contrast, remain in the normal Next/Prev navigation flow; they simply render HTML instead of questions. The property `Page.is_terminal` reports which side of this line a page sits on.

Although you could set `kind=` by hand, the documented idiom is the four **page factory helpers**, each of which returns a `Page` with the appropriate `kind` preset and uses keyword-only arguments to keep call sites explicit:

```
def ContentPage(name, *, body, title=None, show_if=None, hide_if=None) -> Page
    def DisqualificationPage(name, *, title=None, body=None, redirect_url=None,
                             redirect_delay=None, show_if=None, hide_if=None) ->
    Page
    def FinalPage(name, *, title=None, body=None, redirect_url=None,
                  redirect_delay=None, show_if=None) -> Page
    def RedirectPage(name, *, redirect_url, title=None, body=None,
                     redirect_delay=None, show_if=None) -> Page
```

TABLE 5.4

Factory	kind	Required keyword
`ContentPage`	`"content"`	`body`
`DisqualificationPage`	`"disqualification"`	—
`FinalPage`	`"final"`	—
`RedirectPage`	`"redirect"`	`redirect_url`

> **Note:** The factory helpers live in `siamang.core` but are **not** re-exported from the top-level `siamang` namespace. Import them explicitly: `from siamang.core import ContentPage, ...` — writing `sg.ContentPage(...)` will fail.

A complete questionnaire skeleton with all four special screens:

```
import siamang as sg
    from siamang.core import ContentPage, DisqualificationPage, FinalPage, RedirectPage

    age = sg.Variable("age", scale="ratio", label="Age")

    intro = ContentPage("intro", body="<p>Welcome to the study.</p>")

    dq = DisqualificationPage(
        "dq", title="Sorry", body="<p>You are not eligible.</p>",
        show_if=age.lt(18),                      # gate the screen-out
    )

    done = FinalPage(
        "done", title="Thanks", body="<p>Thank you for participating.</p>",
        redirect_url="https://panel.example.com/done", redirect_delay=3,
    )

    rd = RedirectPage("rd", redirect_url="https://panel.example.com/complete")

    intro.is_terminal     # False
    dq.is_terminal        # True
```

Note how `FinalPage` can also redirect: pass `redirect_url` (and optionally `redirect_delay`, in seconds) and the thank-you screen forwards the respondent automatically after the delay — the standard pattern for returning panelists to their provider with a completion link.

> **Tip:** Keep terminal pages at the end of the `pages` list and gate each with `show_if`. Validation checks that every page is reachable and that navigation has no cycles; a clearly ordered list — intro, screeners, main pages, terminal screens — makes those checks trivially satisfiable and the survey easy to reason about.

5.2 Blocks

A **Block** is a named container of questions (and nested blocks) inside a page (wiki/Pages-Blocks-and-Structure.md; docs/reference/core.md). Blocks do not create new screens — everything on a page, including the contents of its blocks, renders on that one screen. What blocks give you is *structure within* the screen: a shared title, a single visibility rule applied to many items at once, and randomization of item order.

5.2.1 Grouping, bulk show_if/hide_if, and randomization

`Block` is a frozen dataclass with five fields:

TABLE 5.5

Field	Type	Default	Description
`title`	`str | None`	`None`	Optional header above the block items.
`items`	`list[Question | Block]`	`[]`	Questions or nested blocks inside the block.
`randomize`	`bool`	`False`	Shuffle the order of items within the block.
`show_if`	`Expression | str | None`	`None`	Show the whole block only when this is true.
`hide_if`	`Expression | str | None`	`None`	Hide the whole block when this is true.

Three use cases cover nearly everything blocks are for.

1. Grouping under a shared heading. A titled block renders its `title` above its items — the wiki's notion of a "section" maps to titled Blocks; there is no separate `Section` class. Use blocks to give a long page readable structure:

```
demographics = sg.Block(
        title="About you",
        items=[q_age, q_gender, q_region],
    )
```

2. Bulk visibility. `show_if` and `hide_if` on a block gate *every item inside it*, which is far clearer than repeating the same condition on five questions. The condition itself is an `Expression` such as `age.ge(18)`; Chapter 7 develops the expression language in depth — here it is enough to see the attachment point:

```
attitudes = sg.Block(
        title="Political attitudes",
        items=[q_trust, q_party],
        show_if=age.ge(18),            # whole block gated to adults
        randomize=True,                # shuffle the two questions
    )
```

3. Randomization. Setting `randomize=True` shuffles the order of the block's items for each respondent — the standard defense against order effects in batteries of attitude items. Randomization composes upward: `Block(randomize=True)` shuffles questions *within* one block, while `Page(randomize_blocks=True)` shuffles the order of the page's immediate `Block` items, leaving standalone questions in place.

```
page = sg.Page(name="main", items=[demographics, attitudes], randomize_blocks=True)
```

Blocks **nest**: an item of a block may itself be a block, so you can build sections within sections — for example, a "Trust in institutions" block containing one sub-block per institution group, each with its own title and visibility rule. Two members help you see through the nesting: `Block.flatten_questions()` returns the questions inside a block, recursing into nested blocks, and — as you saw in §5.1.1 — `Page.flatten_questions()` and `Questionnaire.all_questions()` do the same at the page and survey levels. Because blocks are structure, not identity, flattening always yields the same questions regardless of how deeply they were grouped; the grouping affects presentation, not the dataset. One caveat for the current React runtime: titles, visibility conditions, and randomization attached to a block take effect only for top-level blocks of a page — the contents of a nested sub-block are merged into its parent's items at compile time. Treat nesting as a modeling convenience, not a rendering feature.

One design note on how blocks interact with the rest of the model. Because every object is frozen (Chapter 4), you compose blocks and pages bottom-up: define the questions, wrap them in blocks, place the blocks in pages, and pass the pages to the Questionnaire. Validation then checks the assembled whole — and simulation respects your structure at the page and question level, since invisible questions are not sampled. One limitation to keep in mind: visibility conditions attached to a Block itself are a runtime-only gate — the simulator flattens blocks and does not model block-level visibility.

Putting it together, here is a complete multi-page questionnaire that uses an ordinary question page, a content page, a gated terminal page, and a randomized block:

```python
import siamang as sg
    from siamang.core import ContentPage, DisqualificationPage, FinalPage

    age = sg.Variable("age", scale="ratio", label="Age")
    trust = sg.Variable("trust", scale="ordinal", label="Trust",
                        labels={1: "No trust", 2: "Low", 3: "Medium", 4: "High", 5: "Full"})

    survey = sg.Questionnaire(
        title="Political Trust — 2026",
        pages=[
            ContentPage("welcome", body="<p>Welcome.</p>"),
            sg.Page(
                name="main",
                title="About you",
                items=[
                    sg.NumericInput("How old are you?", var=age, required=True),
                    sg.Block(
                        title="Attitudes",
                        items=[sg.LikertScale("How much do you trust the government?",
                                              var=trust, points=5)],
                        show_if=age.ge(18),
                        randomize=True,
                    ),
                ],
            ),
            DisqualificationPage("dq", body="<p>Not eligible.</p>", show_if=age.lt(18)),
            FinalPage("done", title="Thanks", body="<p>Thank you.</p>"),
        ],
    )

    survey.validate()                       # raises if structure/logic is wrong
    print(len(survey.all_questions()))      # 2 (content/terminal pages hold no questions)
```

Try it yourself: Extend the example above with a second attitudes block — `items=[q_party, q_efficacy]` for two variables you define — and set `randomize_blocks=True` on the `"main"` page. Then answer: (a) does the standalone `NumericInput` move when the blocks shuffle? (b) What does `survey.all_questions()` return, and in what order? Verify both answers in a Python session.

Chapter Summary

- A `Page` is one respondent-facing screen. Its `name` is the unique address used by all branching and skip targets; `items` mixes Questions and Blocks in display order.
- Navigation defaults to the next page in sequence; `next_if` (ordered condition/target pairs) and `default_next` override it — full routing logic is covered in Chapter 7.
- A Questionnaire takes **either** `pages=[...]` (multi-page) **or** `blocks=[...]` (single-page); passing both raises `ValueError`.

- `Questionnaire.validate()` enforces unique page names, existing navigation targets, page reachability, and a cycle-free navigation graph; an empty page earns an `EMPTY_PAGE` lint warning.
- The `kind` field selects special rendering modes: `"content"`, `"disqualification"`, `"final"`, and `"redirect"`. The last three are terminal (`is_terminal` is `True`) and are typically gated with `show_if`.
- Build special screens with the factory helpers `ContentPage`, `DisqualificationPage`, `FinalPage`, and `RedirectPage` — imported explicitly from `siamang.core`, not from the top-level `siamang` namespace.
- A `Block` groups questions (and nested blocks) within one screen for a shared title, bulk `show_if`/`hide_if`, and randomization via `Block(randomize=True)` (within a block) or `Page(randomize_blocks=True)` (across a page's blocks).
- Blocks affect presentation only: `flatten_questions()` and `all_questions()` always recover the same flat, ordered list of questions.

Documentation References

- `wiki/Pages-Blocks-and-Structure.md` — `Page` fields and kinds, page factory helpers, `Block`, the Questionnaire aggregate root, and the pages-XOR-blocks rule.
- `docs/reference/core.md` — frozen dataclass declarations for `Page` and `Block`, `flatten_questions()`, and Questionnaire methods.
- `wiki/Validation-and-Linting.md` — structural checks on page names and navigation, and the `EMPTY_PAGE` lint rule.
- `wiki/Visibility-and-Branching.md` — attachment of `show_if`/`hide_if` at the Page and Block levels (expression syntax itself deferred to Chapter 7).

Chapter 6. Question Types

Learning Objectives

By the end of this chapter, you will be able to:

- Use the fourteen fields shared by every Question — `text`, `var`, `required`, `hint`, `show_if`/`hide_if`, `skip_to`, `randomize`, `other_specify`, `tag`, `id`, `name`, `media`, and `metadata` — and predict each one's default.
- Enrich answer choices with `Option` (per-option visibility and media) and attach `Media` (image, video, audio) to prompts and options.
- Construct all seven question types — `SingleChoice`, `MultiChoice`, `LikertScale`, `NumericInput`, `OpenText`, `Matrix`, and `Ranking` — with their type-specific parameters.
- Choose correctly between MultiChoice array mode (`var=`) and wide mode (`vars=`), and state the data shape each produces.
- Predict each question's `id` and output column name, which you will need for the visibility expressions of Chapter 7.
- Select the right question type for a given measurement goal and anticipate the resulting dataset shape.

6.1 Shared Question Fields and Helpers

Every question in Siamang is an instance of one of seven concrete classes, and all seven inherit from the shared base class Question (wiki/Question-Types.md; docs/reference/core.md). Question itself is not meant to be used directly — you always instantiate one of the seven subclasses — but its fields are the shared vocabulary of every question you will ever write.

A **Question** binds a respondent-facing prompt (the `text`) to one Variable (or several, for `Matrix` and wide-mode `MultiChoice`) via the `var` field; answers are stored in the dataset under the bound variables' names. Before the seven types in §6.2, this section covers the shared fields and the two helper objects, `Option` and `Media`, that work with any of them.

6.1.1 The fourteen shared Question fields; Option and Media helpers

The shared fields of `Question` (wiki/Question-Types.md; docs/reference/core.md):

TABLE 6.1

Field	Type	Default	Description
`text`	`str`	*required*	The prompt shown to the respondent. Must be non-empty.
`var`	`Variable | list[Variable]`	*required*	The bound variable(s); answers are stored under their names.
`required`	`bool`	`False`	Respondent must answer before advancing.
`hint`	`str | None`	`None`	Helper text shown beneath the prompt.
`show_if`	`Expression | str | None`	`None`	Render only when this evaluates true.
`hide_if`	`Expression | str | None`	`None`	Hide when this evaluates true.
`skip_to`	`str | None`	`None`	Jump to a target page/question id after answering.
`randomize`	`bool`	`False`	Shuffle the answer choices.
`other_specify`	`bool`	`False`	Add an “Other (please specify)” free-text choice.
`tag`	`str | list[str] | None`	`None`	Tag(s) for categorization/filtering.
`id`	`str | None`	`None`	Explicit question id; defaults to `name` if that is set, then to the variable name — except for Matrix and wide-mode MultiChoice, where the fallback is `matrix_<first var>` / `multi_<first var>`.
`name`	`str | None`	`None`	Output column name; defaults to the id.
`media`	`Media | list[Media] | None`	`None`	Image/video/audio attached to the prompt.
`metadata`	`dict[str, Any]`	`{}`	Free-form extra parameters (e.g. `{"other_placeholder": "Specify..."}`).

Three of these fields deserve special attention because later chapters depend on them.

`id` and `name` — the question’s address. The `id` is how the rest of the questionnaire refers to the question: `skip_to` targets, script targets, and special keys in lifecycle scripts all use question ids. If you do not set `id`, it defaults to the bound variable’s name — so a question bound to `Variable("gender", ...)` is addressable as `"gender"`. The exceptions are `Matrix` and wide-mode `MultiChoice`, which bind *lists* of variables and therefore fall back to `matrix_<first var>` and `multi_<first var>` respective-

ly. The `name` field, defaulting to the id, controls the output column name. Keep the defaults unless you have a reason: predictable ids make Chapter 7's expressions and Chapter 8's scripts far easier to write.

`show_if` / `hide_if` / `skip_to` — the logic hooks. Every question can be conditionally rendered and can redirect the respondent after being answered. These fields accept the `Expression` objects you met in Chapters 4 and 5 (e.g. `show_if=age.ge(18)`); Chapter 7 is devoted to building them.

`required` and validation interplay. A required question blocks navigation until answered. Note the interaction linting checks for: a question that is both `required` and conditionally visible earns a `REQUIRED_CONDITIONAL` warning at strict lint level, since a hidden required question could trap a respondent.

`Option` — the enriched answer choice. For `SingleChoice`, `MultiChoice`, and `Ranking`, the choices normally come from the bound variable's `labels` mapping. Pass an explicit `choices=[Option(...), ...]` list instead when a choice needs its own visibility condition or a media attachment (wiki/Question-Types.md; docs/reference/core.md):

TABLE 6.2

Field	Default	Description
`code`	*required*	The stored code if chosen. Must match the type used in `Variable.labels`.
`label`	*required*	Displayed text; overrides the variable's registered label. Non-empty.
`show_if` / `hide_if`	`None`	Per-option visibility condition.
`media`	`None`	A single `Media` attachment for this option.

Option codes within one `choices` list must be unique; duplicates raise `ValueError`.

```
from siamang.core import Option, Media

    q_color = sg.SingleChoice(
        "Pick a colour", var=fav,
        choices=[
            Option(1, "Red",   media=Media("https://cdn.example.com/red.png")),
            Option(2, "Blue",  media=Media("https://cdn.example.com/blue.png")),
            Option(3, "Pink",  hide_if=gender.eq(1)),
            Option(4, "Green", show_if=age.ge(18)),
        ],
    )
```

`Media` — image, video, and audio attachments. A `Media` object attaches a media file to a question prompt or to an individual option:

TABLE 6.3

Field	Default	Description
`url`	*required*	URL to the media file. Non-empty.
`kind`	`None`	`"image"`/`"video"`/`"audio"`; inferred from the URL extension if omitted.
`alt`	`None`	Accessibility text (HTML `alt`).
`caption`	`None`	Caption shown beneath the media.
`autoplay`	`False`	Auto-play video/audio when visible.
`loop`	`False`	Loop playback.
`controls`	`True`	Show playback controls.

The `kind` is inferred from the file extension (`png`/`jpg` and friends map to image, `mp4`/`webm` to video, `mp3`/`wav` to audio). A URL **without a recognizable extension** cannot be inferred and raises `ValueError` at construction — pass `kind=` explicitly for such URLs:

```
from siamang.core import Media

    q_clip = sg.SingleChoice(
        "Did you find the clip persuasive?", var=persuasion,
        media=Media("https://cdn.example.com/intro.mp4", caption="Watch before
    answering."),
    )

    # Extensionless URL: kind is required
    logo = Media("https://cdn.example.com/asset?id=42", kind="image", alt="Brand logo")
```

Tip: Always set `alt` on informative images. It costs one line and makes your survey usable with screen readers — and `alt` exists on `Media` precisely for this.

6.2 The Seven Question Types

Siamang ships exactly seven concrete question types, all frozen dataclasses inheriting the shared `Question` fields (wiki/Question-Types.md):

```
from siamang.core import (
        SingleChoice, MultiChoice, LikertScale, NumericInput,
        OpenText, Matrix, Ranking, Option, Media,
    )
```

Each subsection below gives the type-specific fields and a complete example. Everything from §6.1 — `required`, `hint`, `show_if`, `randomize`, and so on — applies on top.

6.2.1 SingleChoice and MultiChoice

`SingleChoice` asks for exactly one answer from a set of mutually exclusive options. Its `var` must be a single `Variable`.

TABLE 6.4

Field	Default	Description
`display`	`"radio"`	UI style: `"radio"`, `"dropdown"`, or `"buttons"` (segmented).
`none_of_above`	`False`	Append a “None of the above” option that deselects others.
`choices`	`None`	Explicit `Option` list; if `None`, derived from the variable’s `labels`.

```
import siamang as sg

    gender = sg.Variable("gender", scale="nominal", label="Gender",
                         labels={1: "Male", 2: "Female", 3: "Other"})

    q_gender = sg.SingleChoice(
        "What is your gender?", var=gender,
        display="buttons", required=True,
    )
```

When `choices` is omitted, the options are derived from `var.labels` — one more reason to keep value labels complete on categorical variables. Two related conveniences: `other_specify=True` (a shared field) appends an “Other (please specify)” free-text choice, and `none_of_above=True` appends a “None of the above” option that deselects the others.

Storage note: a respondent who picks “None of the above” is recorded with the sentinel code "__none__" (analogous to "__other__" from other_specify) — a string that is not part of Variable.labels. Declare such sentinels as missing values, or recode them, before any numeric analysis.

`MultiChoice` allows one or more selections — and it is the one question type with two distinct *storage layouts*, which change the shape of your dataset (wiki/Question-Types.md; docs/reference/core.md):

TABLE 6.5

Field	Default	Description
`min_answers`	`1`	Minimum selections. Advisory in the shipped runtime: it drives the "Select at least N more" hint (shown when `max_answers` is also set) but does not block navigation; `required` enforces only that at least one option is selected.
`max_answers`	`None`	Maximum selections; in wide mode cannot exceed the number of variables.
`exclusive`	`[]`	Codes that clear all other selections when chosen (e.g. "None").
`mode`	`"array"`	`"array"` stores a list in one column; `"wide"` spreads binary flags across columns.
`choices`	`None`	Explicit `Option` list; if `None`, derived from `labels`.

You select the layout by how you bind variables, not by setting `mode` directly. Pass **`var=`** (a single Variable) for **array mode**: one dataset column holding the list of selected codes. Pass the keyword-only **`vars=`** (a non-empty list of Variables) and the question switches to **wide mode** automatically: one binary-flag column per option. Passing both `var` and `vars` raises `ValueError`.

```
# Array mode — one column, list of selected codes
    hobbies = sg.Variable("hobbies", scale="nominal",
                          labels={1: "Music", 2: "Sport", 3: "Reading", 99: "None"})
    q_hobbies = sg.MultiChoice(
        "Which hobbies do you have?", var=hobbies,
        min_answers=1, max_answers=3,
        exclusive=[99],            # picking "None" clears the others
    )

    # Wide mode — one binary column per source
    sources = [sg.Variable(f"src_{n}", scale="nominal", labels={0: "No", 1: "Yes"},
                           label=f"Source {n}") for n in ("tv", "radio", "web")]
    q_sources = sg.MultiChoice("Where do you get news from?", vars=sources)
```

The `exclusive` list is the standard way to implement a "None of these" answer: selecting an exclusive code clears all other selections. In wide mode, remember that `max_answers` cannot exceed the number of bound variables, and the question `id` falls back to `multi_<first var>` (here, `multi_src_tv`).

> **Note:** Choose wide mode when downstream analysis treats each option as its own variable (the common convention for "select all that apply" items in SPSS-style workflows). Choose array mode when the selection is naturally one value — a set of codes in a single column.

6.2.2 LikertScale and Matrix

`LikertScale` renders a symmetric ordinal rating scale — the workhorse of attitude measurement. Its `var` must be a single Variable, ideally `scale="ordinal"`.

TABLE 6.6

Field	Default	Description
`points`	`5`	Number of scale points. Must be `>= 2`.
`left_label`	`None`	Anchor label on the far left (e.g. `"Strongly disagree"`).
`right_label`	`None`	Anchor label on the far right (e.g. `"Strongly agree"`).
`na_option`	`False`	`True` adds a "Not applicable" choice; a string sets its label.

```
trust = sg.Variable("trust", scale="ordinal", label="Trust in government",
                        labels={1: "No trust", 2: "Low", 3: "Medium", 4: "High", 5: "Full"})


    q_trust = sg.LikertScale(
        "How much do you trust the government?", var=trust, points=5,
        left_label="No trust", right_label="Full trust",
        na_option=True,
    )
```

The scale binding is not merely a suggestion: strict linting flags a `LikertScale` whose variable is not ordinal (the `INCOMPATIBLE_QUESTION_SCALE` error). Declare the full `labels` mapping on the variable so reporting tables and charts carry the anchor texts automatically.

`Matrix` renders a grid of subquestions — one row per variable — that all share a common column scale. It is the efficient way to field a battery of Likert items, and it is the second question type (besides wide-mode `MultiChoice`) whose `var` is a *list* of Variables.

TABLE 6.7

Field	Default	Description
`var`	*required*	Non-empty list of `Variable` — one per row.
`subquestions`	`None`	Row labels; default to each variable's `label`.
`column_labels`	`None`	Column headers; default to the variables' value `labels`.
`na_option`	`False`	`True` adds a "Not applicable" column; a string sets its header.

```
def trust_dim(name, label):
        return sg.Variable(name, scale="ordinal", label=label,
                           labels={1: "No trust", 2: "Low", 3: "Medium", 4: "High", 5:
    "Full"})

    q_trust_matrix = sg.Matrix(
        "How much do you trust each of the following?",
        var=[
            trust_dim("trust_govt",   "The government"),
            trust_dim("trust_courts", "The courts"),
            trust_dim("trust_media",  "The press"),
        ],
    )
```

Because each row is its own Variable, a matrix produces **one dataset column per row** (`trust_govt`, `trust_courts`, `trust_media`) — which is exactly what scale-reliability analysis and correlation heatmaps in Part III expect. Row labels default to each variable's `label` and column headers to the shared value `labels`, so well-annotated variables give you a fully labeled grid for free. The question `id` falls back to `matrix_<first var>` (here, `matrix_trust_govt`).

Storage note: when a respondent answers a LikertScale item — or a Matrix row — via the "Not applicable" option, the stored value is the string "na" rather than a numeric code. Recode "na" to a declared missing value before scale-reliability analysis or correlation heatmaps.

6.2.3 NumericInput, OpenText, and Ranking

`NumericInput` collects a continuous numeric value. Its `var` must be a single Variable, on an `interval` or `ratio` scale (strict linting flags anything else).

TABLE 6.8

Field	Default	Description
`display`	`"input"`	`"input"` (number box) or `"slider"`.
`unit`	`None`	Unit shown next to the field (e.g. `"years"`, `"%"`, `"USD"`).
`step`	`1`	Increment for sliders/number inputs. Must be `> 0`.

```
age = sg.Variable("age", scale="ratio", label="Age", valid_range=(18, 99))


    q_age = sg.NumericInput(
        "How old are you?", var=age,
        display="input", step=1, unit="years", required=True,
    )
```

A useful cross-layer connection: if the bound variable declares `valid_range=(min, max)`, the React runtime forwards it as the input's `min`/`max`, so the respondent's browser constrains entries to the same range your data validation will later enforce. Declare the range once on the Variable and both ends respect it.

`OpenText` collects free-form text.

TABLE 6.9

Field	Default	Description
`multiline`	`False`	Render a multiline `<textarea>` instead of a single line.
`max_chars`	`None`	Character limit. Must be `> 0` when set.
`placeholder`	`None`	Ghost text shown while the field is empty.

```
comments = sg.Variable("comments", scale="nominal")


    q_open = sg.OpenText(
        "Anything else you would like to add?", var=comments,
        multiline=True, max_chars=500, placeholder="Optional comments…",
    )
```

`Ranking` asks the respondent to order items by preference, drag-and-drop style. Its `var` must be a single Variable.

TABLE 6.10

Field	Default	Description
`max_ranked`	`None`	How many items must be ranked (e.g. "rank your top 3"). Must be `> 0`.
`choices`	`None`	Explicit `Option` list; if `None`, derived from `labels`.

```
brands = sg.Variable("brand_rank", scale="ordinal",
                     labels={1: "Acme", 2: "Globex", 3: "Initech"})

q_rank = sg.Ranking("Rank these brands from best to worst", var=brands, max_ranked=3)
```

Warning: Keep `points`, `step`, `max_chars`, and `max_ranked` within their documented constraints — `points >= 2`, `step > 0`, `max_chars > 0`, `max_ranked > 0`. These are validated at construction, and since question objects are frozen dataclasses, an invalid value fails immediately when the module is imported, not later at deploy time. That early failure is a feature: `siamang validate` catches it before any respondent ever could.

6.3 Choosing the Right Type

With seven types, the choice is usually determined by two questions: *what is the measurement goal?* and *what data shape do you want back?*

6.3.1 Guidance: measurement goal to question type to data shape

The measurement scale of the bound variable and the question type should agree — remember that strict linting enforces `LikertScale` on ordinal variables, `NumericInput` on interval/ratio variables, and value labels on categorical (`SingleChoice`/`MultiChoice`) variables. Use this table as the decision aid:

TABLE 6.11

Measurement goal	Question type	Variable scale	Resulting data shape
One category from a short list	`SingleChoice` (`display="radio"` or `"buttons"`)	`nominal` or `ordinal`	One column with the selected code
One category from a long list	`SingleChoice` (`display="dropdown"`)	`nominal` or `ordinal`	One column with the selected code
“Select all that apply,” analyzed as one set	`MultiChoice` with `var=` (array mode)	`nominal`	One column holding a list of codes
“Select all that apply,” one flag per option	`MultiChoice` with `vars=` (wide mode)	`nominal` (0/1 labels per option)	One binary column per option
Single agreement/satisfaction rating	`LikertScale`	`ordinal`	One column with the scale point
Battery of items on a shared rating scale	`Matrix`	`ordinal` (one variable per row)	One column per row — ready for reliability analysis

Measurement goal	Question type	Variable scale	Resulting data shape
Age, income, counts, percentages	`NumericInput`	`ratio` or `interval`	One numeric column (optionally bounded by `valid_range`)
Verbatim comments, explanations	`OpenText`	any	One text column
Preference ordering of items	`Ranking`	`ordinal`	One column with the ranking

Two final habits will save you rework in later chapters. First, put the metadata on the Variable, not the question: value `labels`, `valid_range`, and missing-value codes belong to the measurement definition, and every question type derives its defaults (choices, row labels, column headers, input bounds) from them. Second, **let ids default to variable names** wherever possible. Every expression in Chapter 7, every script target in Chapter 8, and every report call in Part III refers to variables and questions by name — a consistent naming convention across your module is the cheapest documentation you will ever write.

> **Try it yourself:** Model this mini-questionnaire: (1) "Do you consent to participate?" — yes/no, required; (2) "Which of these news sources do you use?" — select all, with a "None" option that clears the others, one column per source; (3) a 4-item trust battery on a 5-point scale; (4) "Roughly how many minutes per day do you spend on news?" Define the Variables first, then the four questions, place them on one Page, and run `survey.validate()` followed by `survey.simulate(n=50)`. Inspect `data.frame.columns` and confirm each question produced the data shape you predicted from the table above.

Chapter Summary

- All seven question types inherit fourteen shared `Question` fields; you instantiate subclasses such as `SingleChoice`, never `Question` itself.
- A question binds a prompt (`text`) to measurement definitions (`var`); answers land in dataset columns named after the bound variables.
- Question id defaults to `name` if set, else the variable name (`matrix_<first var>` / `multi_<first var>` for multi-variable types) and is the address used by `skip_to`, scripts, and logic — keep the defaults predictable.
- `Option` enriches a choice with per-option visibility and media; codes must be unique within a `choices` list. `Media` attaches image/video/audio to prompts and options, inferring `kind` from the URL extension — extensionless URLs require an explicit `kind`.

- `SingleChoice` takes one answer with `display "radio"/"dropdown"/"buttons"`; `MultiChoice` offers array mode (`var=`, one list column) versus wide mode (`vars=`, one binary column per option), with `min_answers`/`max_answers`/`exclusive` constraints.
- `LikertScale` (ordinal, `points >= 2`, anchor labels, `na_option`) and `Matrix` (one variable per row, shared column scale) cover rating measurement; `NumericInput` (interval/ratio, honors `valid_range`), `OpenText`, and `Ranking` cover numbers, verbatims, and orderings.
- Strict linting enforces scale–type agreement (`INCOMPATIBLE_QUESTION_SCALE`) and labels on categorical variables (`CATEGORICAL_WITHOUT_LABELS`); constructor constraints like `points >= 2` fail fast at import time.
- Choose types by measurement goal and desired data shape, and put reusable metadata on the Variable so questions derive choices, labels, and bounds automatically.

Documentation References

- `wiki/Question-Types.md` — the `Question` base class and its shared fields, all seven question types with parameters and examples, `Option`, and `Media`.
- `docs/reference/core.md` — frozen dataclass declarations, the `id`/`name` fallback rules, and `metadata` usage.
- `wiki/Variables-and-Measurement.md` — measurement scales per question type and the lint rules tying scale to type.
- `wiki/Validation-and-Linting.md` — `REQUIRED_CONDITIONAL`, `INCOMPATIBLE_QUESTION_SCALE`, and `CATEGORICAL_WITHOUT_LABELS` lint rules.
- `wiki/Pages-Blocks-and-Structure.md` — how questions sit inside pages and blocks (Chapter 5).

Chapter 7. Branching and Visibility Logic

Real surveys rarely show every question to every respondent. A follow-up about children only makes sense for respondents who have children; a screen-out page should only appear to ineligible respondents; some respondents should skip whole sections entirely. Siamang implements all of this with a small, typed expression language. You build a condition once, attach it to a page, block, question, or option, and the same condition is evaluated consistently in Python (for simulation and validation) and in the respondent's browser.

In the previous chapters you learned the object model (Chapter 4), how to assemble pages and blocks (Chapter 5), and the seven question types (Chapter 6). This chapter adds the logic layer on top of those building blocks.

Learning Objectives

By the end of this chapter, you will be able to:

- Build typed `Expression` objects from `Variable` comparison helpers and combine them with `AND`, `OR`, and `NOT`.
- Construct conditions from variable-name strings with compare, and fall back to verbatim SurveyJS string gates when needed.
- Apply `show_if` and `hide_if` gates at the page, block, question, and option level, and explain exactly when an element is rendered.
- Route respondents with question-level `skip_to` jumps and page-level `next_if`/`default_next` rules.
- Use `FilterRule` for Python-side predicates and know when it is — and is not — the right tool.

7.1 The Expression DSL

A *DSL* (domain-specific language) is a small, purpose-built notation embedded in a host language. Siamang's expression DSL lets you write conditions such as "age is at least 18 and region is 1 or 2" as ordinary Python objects. The central type is `Expression`, a frozen dataclass (immutable after creation) that stores an operator and one or two operands:

```
@dataclass(frozen=True, slots=True)
    class Expression:
        op: str
        left: Any = None
        right: Any = None
```

You almost never construct `Expression` directly. Instead, you build trees from `Variable` comparison helpers and the logical combinators, and Siamang assembles the `Expression` tree for you. Inside the tree, a variable is represented by a `VarRef` — a tiny wrapper holding the variable's name. Its string form is `{name}`, which is exactly the token used when the expression is compiled to the SurveyJS dialect understood by the frontend.

The import surface for everything in this chapter is:

```
from siamang.core import (
        Expression, VarRef, compare, AND, OR, NOT, FilterRule,
    )
```

7.1.1 Typed expressions: var.eq/ge/…, compare, AND/OR/NOT, FilterRule

Comparison helpers on Variable. Every `Variable` provides eight helper methods that produce an `Expression`:

TABLE 7.1

Helper	Operator	Example	SurveyJS form
`var.eq(v)`	`=`	`gender.eq(1)`	`{gender} = 1`
`var.ne(v)`	`!=`	`gender.ne(1)`	`{gender} != 1`
`var.gt(v)`	`>`	`age.gt(18)`	`{age} > 18`
`var.ge(v)`	`>=`	`age.ge(18)`	`{age} >= 18`
`var.lt(v)`	`<`	`age.lt(65)`	`{age} < 65`
`var.le(v)`	`<=`	`age.le(65)`	`{age} <= 65`
`var.isin(vs)`	`in`	`region.isin([1, 2])`	`{region} in [1, 2]`
`var.notin(vs)`	`not in`	`region.notin([99])`	`{region} not in [99]`

The ordering operators are also overloaded on `Variable`, so you may write `age >= 18` directly and get the same `Expression(">=", VarRef("age"), 18)` as `age.ge(18)`. Equality and inequality are the exception: == and != are not overloaded on Variable — the frozen dataclass uses them for field-

by-field comparison of whole objects — so you must write gender.eq(1) rather than gender == 1, and gender.ne(1) rather than gender != 1.

> **Warning:** gender == 1 silently produces a plain Python bool, not an Expression. So does gender != 1, which silently evaluates to True — an even more dangerous outcome, because a show_if gate receiving it would show the element to everyone. If you accidentally pass either to show_if, you will not get the behavior you expect. Always use .eq() and .ne() for equality and inequality.

Logical combinators. Conditions are combined with `AND`, `OR`, and `NOT`, or with the equivalent operator forms `&`, `|`, and `~`:

```
def AND(*expressions: Expression) -> Expression    # >= 2 args
    def OR(*expressions: Expression) -> Expression      # >= 2 args
    def NOT(expression: Expression) -> Expression
```

`AND` and `OR` require at least two arguments — passing fewer raises `ValueError` — and fold left to right. The two styles are interchangeable:

```
age >= 18                       # Expression(">=", VarRef("age"), 18)
    age.ge(18) & gender.eq(2)       # Expression("and", ..., ...)
    ~age.isin([18, 19])             # NOT(...)
    AND(age.ge(18), gender.eq(2), region.isin([1, 2]))

    gate = AND(age.ge(18), OR(region.eq(1), region.eq(2)), NOT(party.eq(99)))
    # same as:
    gate = age.ge(18) & (region.eq(1) | region.eq(2)) & ~party.eq(99)
```

> **Tip:** When using `&`, `|`, and `~`, parenthesize every comparison. Python's operator precedence makes `age.ge(18) & gender.eq(2)` fine, but `age.ge(18) | gender.eq(2) & region.eq(1)` groups in a way you probably do not intend. Explicit parentheses make the intent unambiguous.

Working with an Expression. Typed expressions are more than syntax — they can be inspected, evaluated, validated, and serialized:

TABLE 7.2

Method	Purpose
`evaluate(answers) -> bool`	Evaluate against an answers dict. Raises `ValueError` for `raw` expressions.
`variables() -> set[str]`	All variable names referenced in the tree.
`validate(variables) -> None`	Check that every referenced variable exists and the operator tree is well-formed; raises `ValueError` otherwise (also rejects `raw`).
`to_surveyjs() -> str`	Compile to a SurveyJS expression string, e.g. `"{age} >= 18"`.
`to_dict() -> dict` / `Expression.from_dict(payload)`	Lossless JSON serialization round-trip.
`Expression.raw(text)` (classmethod)	Wrap a verbatim SurveyJS string. It cannot be evaluated or validated in Python: as a show_if/hide_if gate it fails validate(), and the React runtime hides the gated element permanently — use a plain string gate instead.

```
expr = AND(age.ge(18), gender.eq(2))

    expr.evaluate({"age": 30, "gender": 2})    # True
    expr.evaluate({"age": 16, "gender": 2})    # False
    expr.variables()                            # {"age", "gender"}
    expr.validate({"age", "gender"})            # OK
    expr.to_surveyjs()                          # "({age} >= 18) and ({gender} = 2)"

    payload = expr.to_dict()
    restored = Expression.from_dict(payload)
    restored.to_surveyjs()                      # "({age} >= 18) and ({gender} = 2)"
```

This round-trip matters architecturally: `Expression.evaluate` backs `Questionnaire.simulate()` and `Questionnaire.validate()` on the Python side, and the frontend evaluates the *same* serialized expression tree in the respondent's browser. One definition, two runtimes, identical behavior.

`compare` — building from a string. When a variable name arrives as a string (from configuration, for example), use `compare` instead of the helper methods:

```
def compare(var_name: str, op: str, value: Any) -> Expression

from siamang.core import compare
    compare("age", ">=", 18)        # Expression(">=", VarRef("age"), 18)
```

Valid `op` values are `"="`, `"!="`, `">"`, `">="`, `"<"`, `"<="`, `"in"`, and `"not in"`.

String form. Every gate also accepts a plain string in the SurveyJS dialect:

```
sg.Page("adults", items=[...], show_if="age >= 18")
```

Strings are preserved verbatim and sent to the frontend, but they are not evaluated in Python. The React runtime evaluates them client-side: both the explicit {age} >= 18 form and bare names such as the age >= 18 above are supported, along with and/or/not, in [...], empty/notempty, and contains. A string the runtime cannot parse is treated as a true gate (with a console warning), and simulate() likewise treats every string gate as always true — so string-gated visibility is exercised only in the browser. `Questionnaire.validate()` still checks that `{name}` tokens inside a string gate reference known variables — bare names such as the `age` above are not extracted or checked. Typed expressions are therefore the recommended default: they are checked by `validate()`, evaluated by `simulate()`, and serialized losslessly.

`FilterRule` — the Python predicate escape hatch. Some logic is easier to write as arbitrary Python than as an expression tree. `FilterRule` wraps a callable:

```
@dataclass(frozen=True, slots=True)
    class FilterRule:
        predicate: Callable[[dict[str, Any]], bool]
        description: str | None = None
```

Its single method, `evaluate(answers) -> bool`, calls the predicate and coerces the result to a boolean:

```
from siamang.core import FilterRule

    adult_in_capital = FilterRule(
        predicate=lambda a: a.get("age", 0) >= 18 and a.get("region") == 1,
        description="Adults living in the capital",
    )
    adult_in_capital.evaluate({"age": 30, "region": 1})    # True
```

Note: `FilterRule` is a Python-side predicate only. Unlike `Expression`, it does **not** compile to a SurveyJS string, so it cannot drive visibility in the respondent's browser. Use it for analysis-time filtering and simulation logic, not frontend gating.

7.2 Applying Logic

Every visibility gate in Siamang follows one rule: an element is rendered **if and only if** `show_if` is true (or absent) **and** `hide_if` is not true (or absent). Routing is a separate mechanism: `skip_to` on a question and `next_if`/`default_next` on a page decide *where the respondent goes next* rather than *what is visible*.

TABLE 7.3

Level	Fields
`Page`	`show_if`, `hide_if`, plus `next_if` / `default_next` routing
`Block`	`show_if`, `hide_if`
`Question`	`show_if`, `hide_if`, `skip_to`
`Option`	`show_if`, `hide_if`

7.2.1 show_if/hide_if at page, block, question, and option level

The same pair of fields exists at four levels of the object model, and the semantics are identical everywhere. Choose the highest level that captures your intent: gating a page or block once is clearer than repeating the same condition on every question inside it.

1) **Page level.** Gate an entire page — most commonly a terminal page such as a screen-out:

```
import siamang as sg
    from siamang.core import AND, ContentPage, DisqualificationPage, FinalPage

    age = sg.Variable("age", scale="ratio", label="Age")

    survey = sg.Questionnaire(
        title="Eligibility Example",
        pages=[
            ContentPage("welcome", body="<p>Welcome.</p>"),
            sg.Page(
                name="main",
                title="About you",
                items=[sg.NumericInput("How old are you?", var=age, required=True)],
            ),
            DisqualificationPage("dq", body="<p>Not eligible.</p>",
                                 show_if=age.lt(18)),
            FinalPage("done", title="Thanks", body="<p>Thank you.</p>"),
        ],
    )
```

The disqualification page only appears to respondents under 18; everyone else flows past it to the final page. A page gate also accepts either form of typed expression:

```
sg.Page("political", items=[...], show_if=AND(age.ge(18), region.isin([1, 2])))
    sg.Page("adults",    items=[...], show_if=age >= 18)
```

2) **Block level.** Gate a group of questions with one condition (Chapter 5 introduced blocks as titled groups within a page):

```
attitudes = sg.Block(
        title="Political attitudes",
        items=[q_trust, q_party],
        show_if=age.ge(18),            # whole block gated to adults
    )
```

3) **Question level.** Gate an individual follow-up question, optionally combining `show_if` and `hide_if`:

```
q_kids = sg.NumericInput(
        "How many children do you have?", var=num_children,
        show_if=has_children.eq(1),
        hide_if=age.lt(18),
    )
```

4) **Option level.** Show or hide a single answer choice by building the `choices` list from `Option` objects instead of relying on the variable's `labels`:

```
from siamang.core import Option

    q_color = sg.SingleChoice(
        "Pick a colour", var=fav,
        choices=[
            Option(1, "Red"),
            Option(2, "Blue"),
            Option(3, "Pink",  hide_if=gender.eq(1)),
            Option(4, "Green", show_if=age.ge(18)),
        ],
    )
```

Tip: Keep conditions simple and readable. If a gate grows beyond two or three combined comparisons, name it (`adult_gate = AND(age.ge(18), region.isin([1, 2]))`) and reuse the name. validate() checks that every variable referenced in a gate is known to the questionnaire (in **`pages=`** mode; blocks-mode questionnaires skip this check), so typos in variable names are caught before deployment.

Try it yourself: Extend the eligibility example: add a `region` variable with `labels={1: "Capital", 2: "North", 3: "South"}`, then gate the disqualification page with `OR(age.lt(18), region.notin([1, 2, 3]))`. Call `survey.validate()` and then `expr.evaluate({"age": 30, "region": 2})` on the gate to confirm your logic in Python before simulating.

7.2.2 skip_to and next_if routing patterns with examples

Visibility gates hide or reveal elements *in place*. Routing jumps the respondent to a different point in the survey. Siamang offers two routing mechanisms, operating at different levels.

`skip_to` — question-level jump. The skip_to field on a Question jumps to a target page name or question id. The jump is applied when the respondent leaves the page by pressing Next: if the first answered visible question on the page has skip_to, the jump is taken regardless of the answer's value, and it takes priority over the page's next_if and default_next:

```
q_screen = sg.SingleChoice(
        "Do you currently live in the country?", var=resident,
        skip_to="dq",            # jump to a disqualification page on any answer
    )
```

The target must exist. `Questionnaire.validate()` raises a ValueError if skip_to references an unknown id. Reachability and cycle detection, however, cover only the navigation graph built from next_if/default_next rules and the implicit page order; skip_to edges are not part of that graph. A loop made purely of skip_to jumps is not caught by the validator — and a page reachable only via skip_to may be falsely reported as unreachable — so review your skip_to chains by hand.

`next_if` — page-level routing. `Page.next_if` is a list of `(condition, target_page_name)` pairs, evaluated in order; the first matching rule wins. If no rule matches, `default_next` is used; if that is also absent, the respondent advances to the next page in sequence:

```
sg.Page(
        name="screener",
        items=[q_age],
        next_if=[("{age} < 18", "dq")],   # SurveyJS-string condition + target page
        default_next="main",
    )
```

Note the shape of the data: `next_if` stores `(condition, target)` **string** pairs, whereas page-level `show_if`/`hide_if` accept either a typed `Expression` or a string. To keep your routing conditions typed and validated, build the strings from expressions with `to_surveyjs()`:

```
sg.Page(
        name="screener",
        items=[q_age],
        next_if=[(age.lt(18).to_surveyjs(), "dq")],
        default_next="main",
    )
```

A typical routing procedure. Putting both mechanisms together, a screening flow usually looks like this:

1) Define the variables and the screening questions on a `screener` page.
2) Add `next_if` rules to the screener page that send ineligible respondents to a terminal page, ordered most-specific first.

3) Set `default_next` to the first page of the main questionnaire so eligible respondents continue normally.
4) Use skip_to on individual questions when a single answered question should redirect the respondent no matter what was answered — the jump is unconditional, so express conditional branching with next_if instead.
5) Run survey.validate() — it verifies unique page names and question ids, valid skip_to and next_if/default_next targets, reachability and cycle detection across the next_if/default_next page graph (skip_to edges are excluded from both), and that all expressions reference known variables.
6) Run survey.simulate() to generate synthetic, logically valid responses and confirm that the visibility logic works; routing (next_if, skip_to, default_next) is not reproduced in the simulation — a known limitation of Siamang 0.6.0.

```
survey = sg.Questionnaire(
        title="Screening Flow",
        pages=[
            sg.Page(
                name="screener",
                items=[sg.NumericInput("How old are you?", var=age, required=True)],
                next_if=[(age.lt(18).to_surveyjs(), "dq")],
                default_next="main",
            ),
            sg.Page(name="main", title="Main questionnaire", items=[q_trust]),
            DisqualificationPage("dq", body="<p>Not eligible.</p>"),
            FinalPage("done", title="Thanks", body="<p>Thank you.</p>"),
        ],
    )
    survey.validate()          # checks targets, reachability, cycles (next_if/default_next
    graph)
    survey.simulate(n=200)     # exercises visibility gates; routing is not simulated
```

Warning: Because `next_if` rules are evaluated in order and the first match wins, order your rules from most specific to most general. A broad catch-all condition placed first will shadow every rule after it.

Try it yourself: Add a second `next_if` rule to the screener above that routes respondents aged 65 or older to a `"seniors"` page before `default_next` applies. Validate, then inspect the routing with the siamang preview CLI command — in Python, survey.preview() returns only a one-line summary such as Questionnaire<Title> with N questions. You can also read the routing directly off the Page objects via page.next_if and page.default_next.

Routing decides *who continues*; the next chapter introduces the complementary mechanism for deciding *how many* of each kind of respondent you accept. Chapter 8 covers quotas — server-side caps on re-

sponse categories, such as “no more than 200 male respondents” — together with scripts, which let you attach custom JavaScript logic to survey lifecycle events. Quota-based screening is the natural next step once your branching logic is in place.

Chapter Summary

- Conditional logic in Siamang is built from typed `Expression` trees: `Variable` comparison helpers (`eq`, `ne`, `gt`, `ge`, `lt`, `le`, `isin`, `notin`) combined with `AND`/`OR`/`NOT` or the `&`/`|`/`~` operators.
- Equality is not overloaded on Variable — == and != fall back to dataclass field-by-field comparison and return plain booleans; always use var.eq(value) and var.ne(value) rather than == and !=.
- Typed expressions can be evaluated in Python (`evaluate`), inspected (`variables`), checked (`validate`), compiled to SurveyJS (`to_surveyjs`), and serialized losslessly (`to_dict`/`from_dict`).
- `compare("age", ">=", 18)` builds an expression from a variable-name string; plain string gates are verbatim escape hatches that bypass Python evaluation. Expression.raw(text) is not a usable gate: it is rejected by validate(), and the React runtime hides the gated element permanently.
- An element is rendered if and only if `show_if` is true (or absent) and `hide_if` is not true (or absent); these fields exist at the page, block, question, and option levels.
- `FilterRule` wraps an arbitrary Python predicate for analysis-time filtering and simulation; it does not compile to the frontend.
- `skip_to` jumps after a question is answered; `next_if`/`default_next` route at the page level, with rules stored as `(condition, target)` string pairs — generate them with `to_surveyjs()`.
- `Questionnaire.validate()` checks routing targets, reachability and cycles in the next_if/default_next page graph, and variable references, making it safe to iterate on complex flows before deployment; skip_to edges are excluded from the reachability and cycle checks.

Documentation References

- `wiki/Visibility-and-Branching.md` — expression DSL, gates, skip/routing semantics, `FilterRule`.
- `wiki/Pages-Blocks-and-Structure.md` — `Page`, `Block`, terminal page factories, `next_if`/`default_next`, `validate()` checks.
- `wiki/Question-Types.md` — `Question` and `Option` visibility fields.
- `wiki/Variables-and-Measurement.md` — `Variable` comparison helpers.
- `docs/reference/core.md` — `Expression`, `VarRef`, `FilterRule`, and validation API reference.

Chapter 8. Quotas and Custom Scripts

Chapter 7 showed you how to control *which* respondents see *which* questions using expressions, gates, and routing. This chapter completes the control layer of your survey with two mechanisms that go beyond declarative branching. *Quotas* cap how many respondents of each kind you accept, keeping your sample within target proportions. *Scripts* inject small JavaScript snippets that run in the respondent's browser at well-defined moments, covering behavior the declarative model does not express.

Learning Objectives

By the end of this chapter, you will be able to:

- Define a `Quota` as a single variable/value/limit cell and compose multi-cell quota plans from several `Quota` objects.
- Attach quotas at deploy time with the `quota=` option and predict the enforcement behavior a respondent experiences.
- Test quota logic in Python with the `reached` predicate before deploying.
- Construct a `Script` with a validated lifecycle trigger, an optional target, and static context data.
- Choose the correct trigger for a task and use the `answers`, `utils`, `api`, and `context` globals safely inside a snippet.

8.1 Quotas

A *quota* caps the number of accepted responses matching a specific category. The classic use case is sample balancing: you want no more than 200 male and 200 female respondents so that the final dataset matches your target proportions. In the screening flow you built in Chapter 7, routing decides who is *eligible*; quotas decide *how many* of each eligible group you accept before closing the cell to further submissions.

8.1.1 Quota(variable, target_value, limit); passing quota= at deploy; enforcement behavior

A quota is a small frozen dataclass with three fields:

```
from siamang.core import Quota
    # or: import siamang as sg  →  sg.Quota

    @dataclass(frozen=True, slots=True)
    class Quota:
        variable: str
        target_value: Any
        limit: int
```

TABLE 8.1

Field	Type	Description
`variable`	`str`	Name of the variable to monitor; must match a `Variable.name`.
`target_value`	`Any`	The category code to count (e.g. `1` for "Male").
`limit`	`int`	Maximum accepted responses matching `variable == target_value`.

Each `Quota` describes exactly **one cell** — a single variable/value pair. There is only one quota "type": if your design constrains several cells or several variables, you compose the plan from multiple `Quota` objects. A balanced two-cell gender plan is:

```
from siamang.core import Quota

    quotas = [
        Quota("gender", target_value=1, limit=200),
        Quota("gender", target_value=2, limit=200),
    ]
```

Testing quota logic in Python. Before deploying, you can exercise a quota cell yourself with the `reached` predicate, which counts how many rows in a list of answer dicts match the cell and returns `True` once the count reaches the limit:

```
def reached(self, answers: list[dict[str, Any]]) -> bool

male_cap = Quota("gender", target_value=1, limit=200)

    responses = [{"gender": 1}, {"gender": 2}, {"gender": 1}]
    male_cap.reached(responses)              # False (2 < 200)
    male_cap.reached([{"gender": 1}] * 200)   # True
```

This is the same predicate used to decide whether a cell is full, so what you test in Python is what respondents will experience.

Attaching quotas at deploy time. Quotas are *not* stored on the `Questionnaire`. You pass them at deploy time via the `quota=` option:

```
quotas = [
        Quota("gender", target_value=1, limit=200),
        Quota("gender", target_value=2, limit=200),
        Quota("region", target_value=1, limit=400),   # cap on the capital region
    ]

    survey.deploy(backend="supabase", frontend="vercel", quota=quotas)
```

Enforcement behavior. When a respondent submits, each quota cell the submission falls into is checked. If a matching cell is already full, the submission is rejected for that cell and the respondent sees a "quota full" screen instead of having the response accepted. Quotas on *different variables are independent*: a submission must clear every cell it falls into. In the example above, a male respondent from the capital region must fit within both the `gender == 1` cell and the `region == 1` cell; if either is full, the submission is turned away.

```
quotas = [
        Quota("gender", 1, limit=200),
        Quota("gender", 2, limit=200),
        Quota("region", 1, limit=400),
        Quota("region", 2, limit=400),
    ]
    survey.deploy(backend="supabase", frontend="vercel", quota=quotas)
```

Note: All bundled backends enforce quotas at submission time — including the `local` SQLite backend, which keeps atomic quota counters. The difference is reachability: `local` is not publicly accessible, so use it for development and preview.

Adjusting quotas between deploys. Because quotas live in the deploy call, tightening or loosening a cell is a code change plus a fresh deploy. Each deploy provisions a **new survey instance** with fresh quota counters — counts do not carry over from the previous deployment:

```
quotas = [
        Quota("gender", 1, limit=150),   # lowered from 200
        Quota("gender", 2, limit=250),   # raised from 200
    ]
    survey.deploy(backend="supabase", frontend="vercel", quota=quotas)
```

Warning: Redeploying resets all quota counters to zero. If you adjust a limit mid-fieldwork, the new deployment starts counting from scratch — plan limit changes accordingly.

Try it yourself: Write a three-cell quota plan capping `region` codes 1, 2, and 3 at 100 responses each. Then verify in Python that `Quota("region", 2,`

`limit=100).reached([{"region": 2}] * 99)` is `False` and flipping the list to 100 rows makes it `True`. Deploy with `backend="local"` and confirm the plan is accepted.

8.2 Scripts

A `Script` injects a custom JavaScript snippet that runs in the respondent's browser at a chosen lifecycle trigger. Scripts exist for behavior the declarative model does not cover: option or page randomization, cross-field validation, timed questions, external API calls, A/B assignment, piped text, and custom event tracking — all without modifying the core runtime. If Chapter 7's expressions are the survey's *logic*, scripts are its *escape hatch* for everything else.

8.2.1 Script objects: lifecycle triggers, factory classmethods, and safe use cases

A script is a frozen dataclass:

```
from siamang.core import Script
    # or: import siamang as sg  →  sg.Script

    @dataclass(frozen=True, slots=True)
    class Script:
        code: str
        trigger: str = "onPageEnter"
        name: str | None = None
        target: str | None = None
        context: dict[str, Any] = {}
        sandbox: bool = True
```

TABLE 8.2

Field	Type	Default	Description
`code`	`str`	*required*	JavaScript source. Must be non-empty.
`trigger`	`str`	`"onPageEnter"`	Lifecycle event that runs the script.
`name`	`str | None`	`None`	Optional identifier shown in logs; useful for debugging.
`target`	`str | None`	`None`	Scope to a page name or question id; `None` runs globally at the trigger.
`context`	`dict[str, Any]`	`{}`	Static data passed into the script at runtime.
`sandbox`	`bool`	`True`	Declared intent to run in a restricted scope; the 0.6.0 React runtime does not enforce isolation — snippets can access window/DOM.

Construction validates the script: an unknown trigger raises `ValueError`, `code` must be non-empty, and an empty `name` is rejected. Scripts are attached to the survey via `Questionnaire(scripts=[...])`, and `Questionnaire.validate()` checks that every script has a known trigger and that a non-`None` `target` matches an existing question id or page name.

Factory classmethods. Four classmethods build ready-made scripts for the most common patterns; each returns a fully configured `Script` with the trigger, target, and name preset:

TABLE 8.3

Factory	Trigger	Scope	What it does
`Script.randomize_options(question_id, seed=None)`	`onQuestionShow`	`question_id`	Shuffle the question's answer options when it is shown; the optional `seed` is stored in `context`.
`Script.randomize_pages()`	`onInit`	global	Shuffle visible page order, keeping the first (welcome) and last page fixed.
`Script.validate_fields_match(field_a, field_b, message="Fields do not match.")`	`onAnswer`	`field_b`	Validate that two answer fields hold the same value (e.g. email confirmation); writes `message` to `answers.__errors__[field_b]` on mismatch.
`Script.timed_question(question_id, seconds=30)`	`onQuestionShow`	`question_id`	Show a question for a limited time, then auto-advance via the runtime's next-page hook.

```
shuffle_party = sg.Script.randomize_options("q_party")
    shuffle_pages = sg.Script.randomize_pages()
    match_emails  = sg.Script.validate_fields_match(
        "email_1", "email_2", message="Emails don't match.",
    )
    timer         = sg.Script.timed_question("q_party", seconds=30)
```

Factories and hand-written scripts mix freely in the `scripts=` list (`custom` is the hand-written script defined in the runtime-context example below):

```
survey = sg.Questionnaire(
        title="Political Trust — 2026",
        pages=[...],
        scripts=[shuffle_party, timer, match_emails, custom],
    )

    survey.validate()      # rejects unknown triggers or targets that don't exist
```

The seven lifecycle triggers. Choosing the right trigger is the most important design decision for a script:

TABLE 8.4

Trigger	When it runs
`onInit`	Once when the survey loads, before the first page.
`onPageEnter`	A page becomes visible.
`onPageExit`	A page is left.
`onQuestionShow`	A question becomes visible (its `show_if` resolved).
`onAnswer`	Any answer changes.
`onSubmit`	Before submission — can modify answers.
`onRandomize`	Randomization is requested.

The JavaScript runtime context. Inside a snippet, four globals are available:

TABLE 8.5

Global	Contents
`answers`	Current respondent answers (read/write). Special keys: `answers.__options__[qid]` (per-question option order), `answers.__pages__` (page order), `answers.__errors__[field]` (validation messages), `answers.__timers__` (timer handles).
`utils`	Helper functions: `shuffle`, `sample`, `clamp`, `debounce`, `now`, `formatDate`.
`api`	`{ get, post }` for external HTTP calls.
`context`	Exactly the static `context` dict passed on the `Script`; the runtime injects nothing else into it.

A complete example — posting a diagnostic timestamp whenever a page is left, with the endpoint supplied through `context`:

```
import siamang as sg

    custom = sg.Script(
        name="log_exit",
        trigger="onPageExit",
        context={"endpoint": "/diagnostics"},
        code="""
            api.post(context.endpoint, { left_at: utils.now() });
        """,
    )
```

Safe use cases and working practices. Scripts are powerful precisely because they bypass the declarative model, so discipline matters:

1) Prefer declarative features first. Reach for a script only when `show_if`/`hide_if`, routing, `randomize` flags, or question options genuinely cannot express the behavior.

2) Keep the sandbox on — as a matter of discipline. The default sandbox=True declares the intent to run the snippet in a restricted scope, but the 0.6.0 React runtime does not enforce isolation: snippets execute with full access to window and the DOM. Treat sandbox=True as a promise not to rely on that access; only disable it when you have a concrete reason.
3) Pass configuration through `context` rather than string-formatting values into `code`. The `context` dict arrives in the snippet unchanged, keeping the JavaScript generic and the Python call site explicit.
4) Always set `name`. It appears in logs and is the fastest way to identify which script fired when debugging.
5) Use `target` to scope scripts to the page or question they concern instead of running them globally at every trigger.
6) Run `survey.validate()` and `survey.simulate()` after adding scripts so invalid triggers and dangling targets are caught before deployment.

Tip: The `onSubmit` trigger can modify answers, and `onAnswer` fires on every change. For cross-field validation, write messages into answers.__errors__[field] from an onAnswer script — such a message is displayed under the question and blocks Next/Submit. An onSubmit script runs too late for this: it is suited to appending or transforming answers just before submission, not to blocking validation.

Try it yourself: Write a script named `"welcome_timer"` with `trigger="onInit"` that stores `utils.now()` into `answers.started_at`. Attach it to a small survey via `Questionnaire(scripts=[...])`, then run `validate()` — and confirm that changing the trigger to a made-up name raises `ValueError`.

With branching (Chapter 7) and quotas and scripts (this chapter) in place, your survey's *behavior* is fully specified. The next chapter turns from behavior to data: Chapter 9 covers variables, measurement scales, and how missing values are represented, so you can design the analytic output of your survey as carefully as its flow.

Chapter Summary

- A `Quota` is one cell — `variable`, `target_value`, `limit` — capping accepted responses where `variable == target_value`; multi-cell and multi-variable plans are composed from multiple `Quota` objects.

- Quotas are passed at deploy time via `survey.deploy(..., quota=quotas)`, not stored on the `Questionnaire`; each deploy starts a new instance with fresh counters.
- Cells on different variables are independent: a submission must clear every cell it falls into, and a full cell produces a "quota full" screen for the respondent.
- `Quota.reached(answers)` lets you test the same full/not-full predicate in Python before deploying.
- A `Script` is JavaScript running in the browser at one of seven lifecycle triggers (`onInit`, `onPageEnter`, `onPageExit`, `onQuestionShow`, `onAnswer`, `onSubmit`, `onRandomize`).
- Snippets access four globals — `answers` (with special `__options__`, `__pages__`, `__errors__`, `__timers__` keys), `utils`, `api`, and `context` — and run sandboxed by default.
- Construction validates triggers and code; `Questionnaire.validate()` additionally checks that a script's `target` matches a real page name or question id.
- Four factory classmethods — `randomize_options`, `randomize_pages`, `validate_fields_match`, and `timed_question` — return fully configured scripts for the most common patterns and mix freely with hand-written scripts.

Documentation References

- `wiki/Quotas.md` — `Quota` fields, `reached` predicate, deploy-time attachment, enforcement behavior, examples, backend support.
- `wiki/Scripts.md` — `Script` fields, trigger validation, lifecycle triggers, JavaScript runtime context, examples.
- `docs/reference/core.md` — `Quota` and `Script` class reference, sandbox behavior, `Questionnaire.deploy` and `validate`.

Chapter 9. Variables, Measurement, and Missing Data

Every question you have written so far binds its answers to a `Variable`. So far you have used variables lightly — a name, a scale, a few labels. This chapter goes deeper. A `Variable` is the atomic measurement unit in Siamang: it binds a data column to a measurement scale, a physical data type, an analytical role, and codebook metadata such as value labels, missing-value conventions, and provenance. Variables validate collected responses, generate dataset schemas, and drive labeled, SPSS-like analysis. Time invested in clean variable metadata pays off twice later: on export to SPSS/Stata (Chapter 14), where labels and missing codes travel with the data, and in analysis (Chapter 15), where reporting tables render with proper labels instead of raw codes.

Learning Objectives

By the end of this chapter, you will be able to:

- Define a fully specified `Variable` with scale, dtype, role, labels, and `valid_range`.
- Choose the correct measurement scale for each question type and predict what strict linting will flag.
- Represent non-substantive responses as structured `MissingValue` entries and distinguish the five missing kinds.
- Maintain a `VariableMap` registry as your survey's codebook and query it by scale or role.
- Validate a collected or simulated DataFrame against the registry with `validate_frame` and interpret `ValidationIssue` results.

9.1 Variables in Depth

The full `Variable` dataclass is larger than the subset used in earlier chapters:

```
from siamang.core import Variable, VariableMap, MissingValue, ValidationIssue

    @dataclass(frozen=True, slots=True)
    class Variable:
        name: str
        scale: str
        label: str | None = None
        labels: dict[Any, str] = {}
        missing_values: tuple[Any, ...] = ()
        dtype: str | None = None
        role: str | None = None
        description: str | None = None
        construct: str | None = None
        source: str | None = None
        valid_range: tuple[Any, Any] | None = None
        missing_labels: dict[Any, str] = {}
        missing: tuple[MissingValue, ...] = ()
```

Construction validates and normalizes your input: `name` must be non-empty; `scale`, `dtype`, and `role` are lowercased and checked against the allowed sets; `valid_range` must be an ordered `(min, max)` pair; and the legacy `missing_values`/`missing_labels` fields are merged into the structured `missing` tuple. Invalid values raise `ValueError` (or `TypeError` for malformed `missing` entries), so a malformed variable fails the moment you define it, not mid-fieldwork.

9.1.1 Scale/dtype/role enums, labels, valid_range

Scale — the level of measurement. The *level of measurement* describes what the numbers mean: whether they are unordered names, ordered ranks, or quantities with meaningful differences and ratios. Siamang supports the four classical scales:

TABLE 9.1

Scale	Meaning	Typical questions
`nominal`	Unordered categories	`SingleChoice`, `MultiChoice`
`ordinal`	Ordered categories	`LikertScale`, `Ranking`, ordinal `SingleChoice`
`interval`	Numeric, no true zero	`NumericInput`
`ratio`	Numeric with a true zero	`NumericInput`

The scale value is case-insensitive (`"Ordinal"` is normalized to `"ordinal"`). Scale choice is not cosmetic: strict linting flags a `LikertScale` bound to a non-ordinal variable, a `NumericInput` bound to a variable that is not `interval`/`ratio` (the `INCOMPATIBLE_QUESTION_SCALE` rule), and categorical variables without value labels (`CATEGORICAL_WITHOUT_LABELS`).

dtype — the physical type. The `dtype` field declares how values are stored and checked: `"int"`, `"float"`, `"str"`, `"bool"`, `"category"`, or `"datetime"`. If you omit it, no dtype constraint is applied.

> **Note:** When dtype is omitted (None), the dtype check is simply skipped during validation. In practice, declare dtype explicitly whenever you know it — an explicit declaration is checked.

role — the analytical purpose. The `role` field records how a variable is used downstream: `"input"`, `"target"`, `"weight"`, `"id"`, `"grouping"`, or `"derived"`. Roles are not just documentation — the `weight` and `id` roles carry constraints that are enforced when a dataset is validated against the registry (Section 9.3).

Labels and ranges. Two kinds of labels serve different audiences:

- `label` is the human-readable variable label. It is used as the SPSS/Stata variable label on export, so write it as you would want it to appear in a statistical package.
- `labels` is a `{code: label}` mapping of value labels for nominal and ordinal variables, e.g. `{1: "Male", 2: "Female", 3: "Other"}`. These drive the answer choices of `SingleChoice`, `MultiChoice`, and `Ranking` questions when no explicit `choices` list is given, and they appear in analysis output.

`valid_range` is an inclusive `(min, max)` tuple for numeric variables. Besides validating data later, a `NumericInput` question forwards the range to the browser as the input's `min`/`max`, so respondents cannot type an out-of-range number in the first place.

Provenance metadata. Three optional fields document where a measure comes from: `description` (long-form explanation of what the variable measures), `construct` (the latent construct being measured, e.g. `"trust_in_institutions"`, which groups indicators under one theoretical concept), and `source` (the provenance of the question wording, e.g. `"ESS Wave 10"`). These travel with the survey as part of the codebook.

```
import siamang as sg

    age = sg.Variable("age", scale="ratio", label="Age",
                      valid_range=(18, 99), dtype="int")

    gender = sg.Variable(
        "gender", scale="nominal", label="Gender",
        labels={1: "Male", 2: "Female", 3: "Other"},
    )

    trust = sg.Variable(
        "trust", scale="ordinal", label="Trust in government",
        labels={1: "No trust", 2: "Low", 3: "Medium", 4: "High", 5: "Full trust"},
        construct="institutional_trust", source="ESS Wave 10",
    )
```

Tip: When you attach a `VariableMap` to a `Questionnaire` via `variables=`, `validate()` checks registry consistency — every question variable must be equal to the registered definition (field-by-field). Defining each variable once and reusing the object everywhere remains the best practice.

9.2 Missing Data

Real respondents refuse questions, answer "don't know", or skip items that do not apply to them. Treating all of these as a blank or a single sentinel like `NaN` destroys analytic information: a refusal and a "not applicable" have very different interpretations. Siamang therefore represents non-substantive responses *structurally*, as declared codes with a label and a kind.

9.2.1 Structured MissingValue: the 5 kinds and how they flow into analysis

A `MissingValue` is a small frozen dataclass — a stored code, a human-readable label (which must be non-empty), and a kind that says *why* the value is missing:

```
@dataclass(frozen=True, slots=True)
    class MissingValue:
        code: Any
        label: str
        kind: str = "system_missing"
```

The `kind` field accepts exactly five values:

TABLE 9.2

Kind	Meaning
`"refusal"`	The respondent declined to answer.
`"dont_know"`	The respondent explicitly answered “don’t know”.
`"not_applicable"`	The question does not apply to this respondent.
`"not_asked"`	The respondent was never presented with the question (e.g. routed past it).
`"system_missing"`	Missing for any other, unspecified reason (the default).

You declare missing values on the variable itself:

```
from siamang.core import Variable, MissingValue

    employment = Variable(
        "employment", scale="nominal", label="Employment status",
        labels={1: "Employed", 2: "Unemployed", 3: "Retired"},
        missing=(
            MissingValue(code=98, label="Don't know", kind="dont_know"),
            MissingValue(code=99, label="Refused", kind="refusal"),
        ),
    )

    employment.is_missing(99)          # True
    employment.missing_kinds_dict()    # {98: 'dont_know', 99: 'refusal'}
```

Three methods expose the configuration: `is_missing(value)` reports whether a value matches a declared missing code; `structured_missing_values()` returns the canonical `MissingValue` tuple; and `missing_kinds_dict()` returns the `{code: kind}` mapping. `MissingValue` also serializes via `to_dict()` and `MissingValue.from_dict(payload)`.

Legacy and structured forms merge. You may also declare missing codes the legacy way, with `missing_values=[...]` plus `missing_labels={...}`:

```
income = sg.Variable(
        "income", scale="ratio", label="Monthly income (USD)",
        missing_values=[-1, -2],
        missing_labels={-1: "Refused", -2: "Don't know"},
    )
```

After construction, the two forms are normalized into one canonical state: `missing` always holds the structured `MissingValue` tuple, `missing_values` mirrors their codes, and `missing_labels` is back-filled from the structured entries. Populate the fields either way — the result is identical, though only the structured form carries a `kind`. Note that every key in `missing_labels` must correspond to a declared missing code.

How missing values flow into analysis. Because missing declarations live on the `Variable`, they are part of the dataset schema that Siamang derives when you `simulate()` or `deploy()`. The same codes and labels are preserved on export to SPSS/Stata (Chapter 14), where they become the package's native missing-value declarations, and they are available to analysis workflows (Chapter 15) through the codebook accessors described below. Declaring kinds up front means you never have to reverse-engineer what a `-1` or `99` meant six months after fieldwork.

> **Warning:** A missing code that is *not* declared is just an invalid value. When you validate a dataset, an undeclared code like `99` on a nominal variable triggers an `INVALID_LABEL_VALUE` error. Declare every non-substantive code your questionnaire can produce.

> **Try it yourself:** Define a `life_satisfaction` variable on a 0–10 scale (`scale="ordinal"`, `valid_range=(0, 10)`, `dtype="int"`) with value labels for the endpoints and two structured missing entries: `97` "Not applicable" and `99` "Refused". Verify that `is_missing(97)` is `True`, `is_missing(5)` is `False`, and `missing_kinds_dict()` returns `{97: 'not_applicable', 99: 'refusal'}`.

9.3 The Codebook

A *codebook* is the document that tells you what every column in a dataset means: its label, its value labels, its missing conventions. In Siamang the codebook is not a document you write separately — it is a live object, the `VariableMap`. A `VariableMap` is a `dict`-like registry of variables indexed by name; it is the centralized schema for a survey and the bridge to collected data.

```
from siamang.core import VariableMap

    vm = VariableMap()
    vm.add_many([age, gender, income, trust])

    vm.require("trust")            # -> Variable('trust', ...)
    vm.by_scale("ordinal")         # [Variable('trust', ...)]
    vm.by_role("weight")           # []
```

Registration and lookup follow strict rules: `add(variable)` registers one variable and raises `KeyError` if the name already exists; `add_many(variables)` registers a list in order; `require(name)` looks up by name and raises `KeyError` if absent; `by_scale(scale)` and `by_role(role)` filter case-insensitively. You can also rebuild a registry from a serialized payload with `VariableMap.from_dict`.

The codebook accessors flatten the registry into plain dictionaries — exactly the shapes that labelling and export tooling consume:

TABLE 9.3

Method	Returns
`labels_dict()`	`{name: label}` (falls back to `name` when no label)
`value_labels_dict()`	`{name: {code: label}}`
`missing_dict()`	`{name: [missing codes]}`
`missing_labels_dict()`	`{name: {code: label}}`
`missing_kinds_dict()`	`{name: {code: kind}}`
`to_dict()`	Full nested serialization of every variable

If you do not pass `variables=` to the `Questionnaire`, Siamang infers a map from your questions automatically — but an explicit registry is the only way to carry roles, constructs, provenance, and structured missing definitions.

9.3.1 VariableMap.validate_frame and ValidationIssue; keeping data and metadata in sync

Design-time checks (`Questionnaire.validate`/`lint`) verify the survey's structure; they say nothing about actual response data. `VariableMap.validate_frame` closes that gap by checking a pandas DataFrame against the registered schema:

```
issues = vm.validate_frame(data.frame)
    errors = [i for i in issues if i.severity == "error"]
    for issue in errors:
        print(issue.code, issue.message)
```

The signature is `validate_frame(frame, raise_on_error=False) -> list[ValidationIssue]`; with raise_on_error=True it raises ValueError after the check completes if any issue has severity == "error" (the message lists all error codes). The checks cover both shape and content:

TABLE 9.4

Code	Severity	Meaning
`MISSING_COLUMN`	error	A declared variable is absent from the frame.
`EXTRA_COLUMN`	warning	A frame column is not declared in the map.
`INVALID_DTYPE`	error	Values incompatible with the declared `dtype`.
`OUT_OF_RANGE`	error	Numeric values outside `valid_range`.
`INVALID_LABEL_VALUE`	error	A nominal/ordinal value not present in `labels`.
`MISSING_VALUE_WITHOUT_LABEL`	warning	A missing code has no label.
`INVALID_WEIGHT` / `DUPLICATE_ID`	error	Role constraints for `weight`/`id` variables.

Each finding is a `ValidationIssue`:

```
@dataclass(frozen=True, slots=True)
    class ValidationIssue:
        code: str
        severity: str              # "error" or "warning"
        message: str
        variable: str | None = None
        column: str | None = None
```

The `code` is machine-readable (e.g. `"OUT_OF_RANGE"`), and `variable`/`column` point to the offender when applicable, so you can filter, count, and route issues programmatically instead of parsing messages.

Keeping data and metadata in sync. The discipline this enables is simple: treat the `VariableMap` as the single source of truth and validate against it at every stage. A sound workflow is:

1) Define all variables once, with scales, dtypes, labels, ranges, and structured missing values.
2) Register them in a `VariableMap` and pass it to `Questionnaire(variables=vm)` so design-time validation checks registry consistency.
3) After `simulate()` or after collecting real responses, run `vm.validate_frame(...)` and triage errors before warnings.
4) Fix drift at the source: if the questionnaire changes, update the variable definition — never patch the data to match stale metadata.
5) Re-validate before every export or analysis run.

Because the same registry backs simulation, deployment, export, and analysis, a correction made in one place propagates everywhere. That is the practical payoff of the codebook-as-object design: metadata cannot silently diverge from the data it describes.

> **Try it yourself:** Build a `VariableMap` from the `age`, `gender`, and `employment` variables in this chapter, then call `validate_frame` on a DataFrame with one deliberate fault per row: an age of `12` (out of range), a gender code of `7` (invalid label value), and a missing `employment` column. Confirm you receive `OUT_OF_RANGE`, `INVALID_LABEL_VALUE`, and `MISSING_COLUMN` issues with the expected severities.

The next chapter shifts from data semantics to presentation: Chapter 10 covers theming and the frontend configuration that controls how your survey looks to respondents.

Chapter Summary

- A `Variable` binds a column to a measurement scale (`nominal`/`ordinal`/`interval`/`ratio`), a physical `dtype`, an analytical `role`, and codebook metadata; construction normalizes and validates all of it.
- Scale choice has teeth: strict linting flags `LikertScale` on non-ordinal variables, `NumericInput` on non-interval/ratio variables, and categorical variables without labels.
- `label` becomes the SPSS/Stata variable label on export; `labels` drives question choices and analysis output; `valid_range` validates data and constrains numeric inputs in the browser.
- `MissingValue(code, label, kind)` records *why* a value is missing, with five kinds: `refusal`, `dont_know`, `not_applicable`, `not_asked`, `system_missing`.
- Legacy `missing_values`/`missing_labels` and structured `missing=` declarations merge into one canonical state after construction.
- A `VariableMap` is the survey's codebook: a `dict`-like registry with `add`/`add_many`/`require`/`by_scale`/`by_role` and codebook accessors for labels and missing metadata.
- `validate_frame` checks a DataFrame against the registry — columns, dtypes, ranges, label values, missing labels, and weight/id role constraints — returning `ValidationIssue` objects with `error`/`warning` severities.
- Treating the `VariableMap` as the single source of truth keeps simulation, collection, SPSS/Stata export (Chapter 14), and analysis (Chapter 15) in sync.

Documentation References

- `wiki/Variables-and-Measurement.md` — `Variable` fields and enums, measurement scales, `MissingValue`, `VariableMap` registry and `validate_frame`, `ValidationIssue`, codebook usage.
- `wiki/Validation-and-Linting.md` — strict lint rules referencing variable scales and labels.
- `docs/reference/core.md` — `Variable`, `MissingValue`, `VariableMap`, and `ValidationIssue` class reference (including the `dtype` wording discrepancy noted above).

Chapter 10. Theming and the Respondent Experience

So far you have been working on the *content* of your survey: its pages, questions, logic, quotas, and variables. This chapter turns to what the respondent actually sees and touches: the colors, typography, layout, navigation chrome, and the runtime that renders your questionnaire in a browser. In Siamang, everything visual is controlled by a single configuration object, `UIConfig`, and everything about rendering is handled by the `siamang.frontend` subpackage, which compiles your `Questionnaire` into a deployable static bundle of HTML, CSS, and JavaScript.

Learning Objectives

By the end of this chapter, you will be able to:

- Explain what `UIConfig` is and navigate its roughly 66 theming fields by functional group.
- Build a custom theme for a survey, including palette, typography, layout, branding, and translated UI strings.
- List the six built-in `THEME_PRESETS` and customize a preset using `dataclasses.replace`.
- Describe the `SurveySchema` → `FrontendBuilder` → `SurveyBundle` compilation pipeline and what each stage produces.
- Choose between the SurveyJSRuntime (the default only for a hand-built FrontendBuilder) and the ReactRuntime (the default for survey.deploy() and siamang preview) for a given survey.

10.1 UIConfig

UIConfig is a frozen dataclass — immutable after creation, like all core model classes in Siamang — that controls the entire look of the deployed survey. Its defaults aim for a calm, research-grade aesthetic: a serif body font, a narrow line measure, a single accent color, and comfortable spacing. You pass it either directly to `survey.deploy(..., ui=UIConfig(...))` or to a `FrontendBuilder` when driving the frontend machinery by hand.

```
from siamang.frontend import UIConfig

    ui = UIConfig(primary_color="#2c5f8a")
```

10.1.1 The ~66 theming fields; worked theming example

`UIConfig` has roughly 66 fields, best learned in seven functional groups.

Palette. Ten color fields define the visual identity. Colors are given as CSS hex strings.

TABLE 10.1

Field	Default	Meaning
`primary_color`	`"#2c5f8a"`	Brand color for primary buttons and active states.
`accent_color`	`None`	Optional accent; falls back to `primary_color`.
`background_color`	`"#fbfbfb"`	Page background.
`surface_color`	`"#ffffff"`	Card, panel, and input background.
`text_color`	`"#1a1a1a"`	Body text.
`muted_text_color`	`"#5a5a5a"`	Secondary text.
`border_color`	`"#e6e4df"`	Dividers and input borders.
`error_color` / `error_soft_color`	`"#b3261e"` / `"#fdf1f0"`	Validation error text and its background tint.
`warn_color`	`"#9a6a1a"`	Warning states.

Typography. The high-level knob is `font_preset`, which selects a coordinated pair of typefaces, each shipping its own Google Fonts URL:

TABLE 10.2

`font_preset`	Typefaces
`"academic"` (default)	Source Serif 4 body text with Inter for UI elements.
`"modern"`	Inter everywhere.
`"humanist"`	Nunito.

For finer control you can override individual fields: `font_family`, `heading_font_family`, `ui_font_family`, `mono_font_family`, `font_size` (default `"15.5px"`), `line_height` (default `"1.6"`), and `font_pair` (`"serif"`, `"sans"`, or `"mixed"`).

Layout. Four fields shape the page geometry: `width` (default `"700px"`, deliberately kept under ~750px so lines stay within a readable 60–75 character measure), `radius` (corner rounding, default `"4px"`), `density` (`"compact"`, `"comfortable"`, or `"spacious"`), and `question_style` (`"plain"`, `"divided"`, `"carded"`, or `"accent"`), which controls how individual questions are visually separated.

Branding and header. These fields control the survey masthead: `logo_url`, `logo_text` (auto-derived from the initials of `institution_name` if unset), `logo_position`, `show_title`,

`institution_name`, `study_subtitle`, `show_section_numbers`, `show_progress_text`, and `estimated_minutes`.

Footer. Three fields add institutional accountability to the page footer: `privacy_url`, `contact_email`, and `ethics_statement` (for example, an IRB reference).

UI strings (internationalization). Twenty optional overrides let you translate or reword every piece of navigation chrome; all default to English:

TABLE 10.3

Group	Fields
Navigation buttons	`next_button_text`, `prev_button_text`, `submit_button_text`, `submitting_text`
Answering aids	`required_text`, `saving_text`, `select_placeholder`, `of_text`, `selected_text`
Resume/restart	`resume_title`, `resume_action`, `restart_action`
Progress and errors	`page_text`, `of_total_text`, `retry_title`, `retry_body`, `retry_action`, `save_local_action`
Completion screen	`completion_title`, `completion_body`

Advanced. The final group covers navigation behavior, access control, and analytics: `progress_style` (`"bar"`, `"dots"`, or `"both"`), `default_theme` (`"light"`, `"dark"`, or `"system"`), `redirect_url`, `allow_back`, **`enable_analytics`** (injects the Vercel Analytics script whenever True, whichever frontend you deploy to), and an access gate made up of `require_access_code`, `access_codes`, `access_title`, `access_body`, `access_placeholder`, and `access_button`. Finally, `custom_css` is a raw escape hatch: anything you put there is appended verbatim to the compiled stylesheet.

> **Warning:** `UIConfig` validates `logo_position`, `density`, `font_pair`, `question_style`, and `font_preset` in its `__post_init__`. An unknown value raises `ValueError` at construction time — a typo in `question_style="card"` fails immediately, not after deployment.

Worked theming example. The following builds a complete theme from scratch — no preset — for a fictional community-health study with a warm palette, humanist typography, carded questions, institutional branding, and German button labels:

```
from siamang.frontend import UIConfig

    ui = UIConfig(
        # Palette
        primary_color="#2e7d5b",
        background_color="#faf7f2",
        surface_color="#ffffff",
        # Typography and layout
        font_preset="humanist",
        question_style="carded",
        density="comfortable",
        # Branding
        institution_name="Riverside Health Collective",
        study_subtitle="Community Wellbeing Survey 2026",
        estimated_minutes=10,
        # Footer
        contact_email="study@riverside.example.org",
        ethics_statement="Approved by the Riverside IRB, protocol 2026-041.",
        # UI strings (German)
        next_button_text="Weiter",
        prev_button_text="Zurück",
        submit_button_text="Absenden",
        required_text="Pflichtfeld",
    )

    survey.deploy(backend="supabase", frontend="vercel", ui=ui)
```

Because `logo_text` was left unset, the header automatically shows “RH”, derived from the initials of the first two words of institution_name.

10.1.2 THEME_PRESETS: the 6 built-in presets

If you do not want to assemble a theme field by field, `get_preset(name)` returns a fully configured `UIConfig` from the six presets shipped in `THEME_PRESETS`:

TABLE 10.4

Name	Typography	Look	Best for
`default`	academic	Serif, light grey background.	Standard academic studies.
`academic`	academic	Explicit academic styling (680px width).	Alias of `default`.
`dark`	academic	Dark slate (`#10131a`), blue accents.	Night reading, tech studies.
`modern`	modern	White, indigo accents, spacious, 16px.	Public-facing / consumer surveys.
`humanist`	humanist	Warm off-white, green accents, rounded.	Community / non-profit research.
`high_contrast`	modern	Black/white, bold borders, 18px text.	WCAG AAA accessibility.

```
from siamang.frontend import get_preset


    ui = get_preset("dark")
```

An unknown preset name raises `KeyError` listing the valid options.

A preset is a starting point, not a straitjacket. Because `UIConfig` is frozen, you customize a preset with `dataclasses.replace`, which returns a modified copy:

```
import dataclasses
    from siamang.frontend import get_preset

    ui = dataclasses.replace(
        get_preset("modern"),
        institution_name="Independent Polling Lab",
        primary_color="#a8324b",
        estimated_minutes=8,
        # translate the navigation chrome to French
        next_button_text="Suivant",
        prev_button_text="Précédent",
        submit_button_text="Envoyer",
    )

    survey.deploy(backend="supabase", frontend="vercel", ui=ui)
```

Tip: To inspect what a theme actually generates, call `compile_css(ui)`. It compiles a `UIConfig` into a CSS stylesheet string built on CSS custom properties — the same stylesheet used as the fallback `style.css` when a runtime does not ship its own. Printing it is the fastest way to check that your color and font choices landed where you expected.

Try it yourself: Take a survey from an earlier chapter, build two themes — `get_preset("academic")` and a `dataclasses.replace` variant of `get_preset("high_contrast")` with your institution's name and a different `primary_color` — and compare the output of `compile_css` for each. Note which CSS custom properties change.

10.2 Frontends and Runtimes

The `siamang.frontend` subpackage compiles a `Questionnaire` into a deployable static bundle — HTML, CSS, JavaScript, and runtime configuration. In everyday work you rarely touch it directly: `survey.deploy(...)` and `siamang preview` drive the machinery for you. But understanding the pipeline helps when you want to build a bundle by hand, inspect its output, or swap the rendering runtime.

```
from siamang.frontend import (
        FrontendBuilder, UIConfig, get_preset, compile_css, compile_questionnaire,
        SurveySchema, SurveyBundle,
        SurveyJSRuntime, ReactRuntime,
        LocalClientTemplate, SupabaseClientTemplate, ClientEnv,
    )
```

10.2.1 SurveySchema → FrontendBuilder → SurveyBundle pipeline; SurveyJSRuntime vs ReactRuntime

The pipeline has three stages.

Stage 1: compile the questionnaire to a `SurveySchema`. `SurveySchema` is a frozen, platform-agnostic snapshot of your survey — an intermediate representation that backends, runtimes, and themes all consume. You produce it with `compile_questionnaire`:

```
from siamang.frontend import SurveySchema, compile_questionnaire

    schema = compile_questionnaire(survey, options={"language": "en"})
```

Its fields include `title`, `pages`, `variables`, `language`, `description`, `completion_text`, `show_progress`, `allow_back`, `one_question_per_page`, `deadline`, `max_responses`, `quotas`, and `metadata`. Two render methods expose the schema in different dialects: `schema.to_surveyjs()` returns a SurveyJS-compatible payload (mapping, for example, `show_progress` to `showProgressBar: "top"` and `one_question_per_page` to `questionsOnPageMode: "questionPerPage"`, after stripping internal keys such as `_quota_variable` and `_meta` from each page), while `schema.to_dict()` returns the full JSON serialization.

Stage 2: hand the schema to a `FrontendBuilder`. The builder is the orchestrator. It carries a runtime adapter and a `UIConfig`, and its `build(...)` method renders every artefact:

```
from siamang.frontend import FrontendBuilder, SurveyJSRuntime, UIConfig

    builder = FrontendBuilder(runtime=SurveyJSRuntime(), ui=UIConfig())
```

Both constructor arguments are optional and default to `SurveyJSRuntime()` and `UIConfig()`. The `build` signature is:

```
def build(
        self,
        schema: SurveySchema,
        *,
        client: BackendClientTemplate,
        env: ClientEnv,
        survey: Questionnaire | None = None,
    ) -> SurveyBundle: ...
```

The `client` is a backend client template — such as `LocalClientTemplate()` or `SupabaseClientTemplate()` — which emits the `env.js` snippet that wires the in-browser client to your response store. `ClientEnv` carries only frontend-safe values (URLs, anonymous keys); secrets such as service keys never reach the bundle. The optional `survey=` argument is only needed by runtimes that require the live `Questionnaire` — that is, the React runtime; `SurveyJSRuntime` works from the schema alone.

Stage 3: receive a `SurveyBundle`. The builder returns a `SurveyBundle` whose filenames are already content-hashed. It contains five base files; a runtime may add its own static assets on top — ReactRuntime contributes bundle.js plus vendored React files under vendor/, while SurveyJSRuntime adds none. The base files are:

TABLE 10.5

File	Purpose
`index.html`	The survey entry point.
`closed.html`	Shown when the survey has expired, hit `max_responses`, or quotas are full.
`style.css`	The compiled theme (from the runtime, or `compile_css(ui)` as a fallback).
`env.js`	Runtime config emitted by the backend client template.
`manifest.json`	Metadata: runtime, client, backend, `survey_id`, `schema_hash`, build time.

The bundle offers several convenience methods: `write_to(target)` writes every file under a directory, `to_zip()` returns a deflate-compressed ZIP, `manifest_json()` pretty-prints the manifest, `compute_digest()` returns a 16-character SHA-256 prefix over all contents, and `with_hashed_filenames()` renames `.js`/`.css` files to include a content hash (which `build(...)` already calls for you).

Here is the end-to-end flow by hand:

```python
import siamang as sg
    from siamang.frontend import (
        FrontendBuilder, UIConfig, LocalClientTemplate, ClientEnv, compile_questionnaire,
    )

    survey = sg.Questionnaire(title="Demo", pages=[...])

    schema = compile_questionnaire(survey, options={"language": "en"})
    builder = FrontendBuilder(ui=UIConfig(primary_color="#2c5f8a"))
    env = ClientEnv(survey_id="abc123", backend="local", settings={})

    bundle = builder.build(schema, client=LocalClientTemplate(), env=env)
    bundle.write_to("./dist")          # writes index.html, style.css, env.js, ...
```

Choosing a runtime. A `RuntimeAdapter` turns the compiled schema into HTML pages and client-side behavior. Two runtimes are bundled:

TABLE 10.6

	`SurveyJSRuntime` (default for `FrontendBuilder`)	**`ReactRuntime`**
Foundation	Lightweight, non-React runtime built on the SurveyJS core library.	Compiles the questionnaire into a standalone React 18 application with a bundled design-system stylesheet.
Why choose it	Highly compatible; needs zero build tooling. Note that it does not execute siamang's routing payload (skip_to / next_if / default_next) — surveys that rely on routing should use ReactRuntime.	Required for advanced interactive features (custom charts, custom widgets, complex animations) and for surveys that use routing (skip_to / next_if / default_next).
Needs live questionnaire?	No — works from the `SurveySchema` alone.	Yes — pass `survey=` to `build(...)`.

Note that both siamang preview and survey.deploy(...) use the React runtime by default, so the preview you see while developing reflects the same React design system your respondents will get. SurveyJSRuntime is the default only when you construct a FrontendBuilder yourself.

> **Note:** The respondent experience is also shaped at the edges by elements you met in earlier chapters: terminal `FinalPage`/`RedirectPage` screens (with `redirect_url` and `redirect_delay`), the "quota full" screen shown when a quota cell is closed, and the UI string overrides in `UIConfig` that reword the completion and retry screens. Theming is therefore not just colors — it is the whole surface a respondent touches.
>
> **Try it yourself:** Build the same survey twice with `FrontendBuilder` — once with the default `SurveyJSRuntime()` and once with `ReactRuntime()` (remembering to pass

`survey=` to `build`) — then compare the two bundles' `manifest.json` and `compute_digest()` values to confirm they are distinct artefacts.

One last word before you deploy a beautifully themed survey: theming changes nothing about structural soundness. A survey with a custom `UIConfig`, translated strings, and an access gate still needs to pass `validate()` and should be checked with `lint()` — ideally via `siamang validate my_survey.py --strict` — before every deployment. The next chapter covers those validation and linting gates in full.

Chapter Summary

- `UIConfig` is a frozen dataclass of roughly 66 fields that controls the entire look of a deployed survey; pass it via `survey.deploy(..., ui=...)` or to a `FrontendBuilder`.
- The fields fall into seven groups: palette, typography (anchored by `font_preset`), layout, branding/header, footer, twenty UI-string overrides for translation, and advanced settings including an access gate and `custom_css`.
- Invalid values for the enumerated fields (`density`, `font_pair`, `question_style`, `font_preset`, `logo_position`) raise `ValueError` at construction time.
- Six presets ship in `THEME_PRESETS` — `default`, `academic`, `dark`, `modern`, `humanist`, and `high_contrast`; retrieve one with `get_preset(name)` and customize it with `dataclasses.replace`.
- `compile_css(ui)` compiles a theme into a CSS string, useful for inspection and as the fallback `style.css`.
- The frontend pipeline is: `compile_questionnaire` → `SurveySchema` → `FrontendBuilder.build(...)` → `SurveyBundle` with `index.html`, `closed.html`, `style.css`, `env.js`, and `manifest.json`.
- `SurveyJSRuntime` is the default renderer only for a hand-built FrontendBuilder (compatible, zero build tooling) and does not execute routing; survey.deploy() and siamang preview default to ReactRuntime, which produces a standalone React 18 app for advanced interactivity and routing, and needs the live Questionnaire via survey=.
- Themed surveys must still pass `validate()` and `lint()` before deployment — the subject of Chapter 11.

Documentation References

- `wiki/Frontend-and-Theming.md` — the `siamang.frontend` subpackage, `SurveySchema`, `FrontendBuilder`, `SurveyBundle`, runtimes, client templates, `UIConfig` field groups, `THEME_PRESETS`, and `compile_css`.
- `docs/reference/core.md` — the frozen-dataclass architecture shared by all public classes, and the `Questionnaire` methods (`compile`, `deploy`) that drive the frontend pipeline.
- `wiki/Pages-Blocks-and-Structure.md` — terminal page kinds and redirect behavior that shape the respondent-facing end screens.
- `wiki/Quotas.md` — the "quota full" screen shown to respondents when a quota cell is full.
- `wiki/Validation-and-Linting.md` — the `validate`/`lint` gates and the `siamang validate` CLI referenced in the closing section.

Chapter 11. Validation and Linting

A survey that compiles is not necessarily a survey that works. Before you deploy, Siamang gives you two complementary quality gates, both running in pure Python: **structural validation** (`validate`), which enforces hard invariants and refuses to proceed when the design is broken, and **linting** (`lint`), which surfaces softer stylistic and best-practice issues without stopping you. Together they back the `siamang validate` command-line tool. This chapter — the last in Part II — teaches you to read and act on both, so that every survey you hand to the deployment machinery in Part III is sound by construction.

Learning Objectives

By the end of this chapter, you will be able to:

- Run `Questionnaire.validate()` and `Questionnaire.validate(strict=True)` and interpret their `ValueError` semantics.
- Enumerate the structural checks validation performs, from duplicate IDs to navigation cycles.
- Interpret the exit codes 0, 1, and 2 of the `siamang validate` CLI.
- Read a `LintWarning` and understand its four fields.
- Diagnose and fix each of the eleven basic-level and four strict-level lint rules.
- Distinguish design-level checks from dataset-level checks (`SurveyData.validate`).

11.1 Structural Validation

Structural validation guards the invariants that must hold for a survey to be deployable at all: unique identifiers, resolvable navigation, evaluable logic. It lives in one method:

```
def validate(self, strict: bool = False) -> None: ...
```

11.1.1 validate(strict=): ValueError semantics; CLI exit codes 0/1/2

`validate()` raises `ValueError` on the **first** structural problem it finds and returns `None` when the survey is sound. This fail-fast semantics means you fix errors one at a time: repair the reported problem, run validation again, and repeat until the call returns silently. The checks performed are:

- **Duplicate question IDs** and **duplicate variable names**.
- **Unknown `skip_to` targets** — a question may only skip to a known question ID or page name.

- **Page integrity** (in pages mode): no empty or duplicate page names; every `show_if`/`hide_if`/`next_if` expression references only known variables and is itself evaluable; all navigation targets exist; every page is reachable from the first page; and the navigation graph contains **no cycles**.
- **Export-safety** of page `show_if` expressions for the SurveyJS frontend.
- **Script triggers and targets** — each script must use a known trigger and point at a real question or page.
- **Registry consistency** — if the questionnaire carries a `VariableMap`, every question variable must be equal (field-by-field) to the definition registered there.

Passing `strict=True` layers linting on top: validation additionally runs `lint(level="strict")` and **promotes any lint finding with `severity == "error"` into a `ValueError`** — for example, failing the build on a `LikertScale` bound to a non-ordinal variable. Use strict mode in automated checks where you want a hard failure, not just a printed warning.

```
from siamang.core import Variable, LikertScale, SingleChoice, Page, Questionnaire

    consent = Variable("consent", scale="nominal", label="Consent", labels={1: "Yes", 0: "No"})
    autonomy = Variable("autonomy", scale="ordinal", label="Autonomy",
                        labels={1: "Very low", 2: "Low", 3: "Moderate", 4: "High", 5: "Very high"})

    survey = Questionnaire(
        title="Autonomy Study",
        pages=[
            Page(name="consent", items=[SingleChoice("Do you consent?", var=consent, required=True)]),
            Page(name="main", items=[LikertScale("How much autonomy?", var=autonomy, points=5)],
                 show_if=consent.eq(1)),
        ],
    )

    survey.validate()              # returns None — structurally valid
    survey.validate(strict=True)   # also fails on strict-level lint *errors*
```

> **Tip:** Make `survey.validate(strict=True)` the last line of your survey script before any `deploy(...)` call. It costs nothing, and it converts whole classes of deployment failures into immediate, local, readable errors.

The CLI. `siamang validate my_survey.py` is a thin wrapper around these methods: it loads the survey object, calls validate(strict=...), also validates the module-level options dict if the module exports one (quotas live there), then prints all lint() warnings. Its contract is expressed through exit codes, which makes it ideal for build scripts and continuous-integration checks:

TABLE 11.1

Exit code	Condition
`0`	Valid; no warnings, or only `warning`-severity lint findings.
`1`	Reserved: error-severity lint rules exist only at the strict level, and --strict promotes them to a ValueError, so with the current rule set they surface as exit code 2 instead.
`2`	`validate() (or the options check) raised a ValueError (structural failure).`

```
siamang validate my_survey.py            # basic lint
    siamang validate my_survey.py --strict   # strict validation + lint
```

Reading the codes: 2 means the survey is broken and cannot deploy — validate() (or the options check) raised a ValueError; 1 means validation passed but at least one lint finding had severity == "error"; 0 means clean or only advisory warnings. Note, however, that error-severity lint rules only exist at the strict level, and --strict runs validate(strict=True) first — which promotes those same findings into a ValueError (exit code 2). With the current rule set, exit code 1 is therefore a reserved part of the contract rather than an outcome you will commonly see.

11.2 Lint Warnings

Where `validate` asks “is this survey structurally sound?”, `lint` asks “is this survey well designed?”. Linting never raises (except on an invalid `level` argument) and, unlike validation, reports **all** findings at once:

```
def lint(self, level: str = "basic") -> list[LintWarning]: ...
```

Each finding is a `LintWarning`, imported from `siamang.core.questionnaire`:

TABLE 11.2

Field	Type	Description
`code`	`str`	Machine-readable rule code, e.g. `"EMPTY_PAGE"`.
`severity`	`str`	`"warning"` or `"error"`.
`message`	`str`	Human-readable description.
`location`	`str | None`	Page name or question ID, when applicable.

11.2.1 lint(level=): the 11 basic + 4 strict warning codes and how to fix them

With the default level="basic", eleven rules are checked — four structural rules (listed below) plus seven more that check codebook and logic consistency:

TABLE 11.3

Code	Severity	Meaning	How to fix
`EMPTY_QUESTIONNAIRE`	warning	No pages and no blocks.	Add pages (or blocks for a single-page survey).
`EMPTY_PAGE`	warning (error in strict)	A page has no items.	Add questions/blocks to the page, or remove it.
`REDUNDANT_NAVIGATION`	warning	`default_next` duplicates the implicit next page.	Drop the `default_next`; the sequential default already does this.
`MISSING_NAVIGATION`	warning	A page other than the last has no outgoing navigation edges.	Add `next_if`/`default_next`, or confirm sequential flow is intended.

With `level="strict"`, four further rules are added:

TABLE 11.4

Code	Severity	Meaning	How to fix
`REQUIRED_CONDITIONAL`	warning	A required question also has conditional visibility.	Make the question optional, or gate it at the page/block level instead.
`INCOMPATIBLE_QUESTION_SCALE`	error	`NumericInput` not on interval/ratio, or `LikertScale` not on ordinal.	Correct the variable's `scale` to match the measurement level.
`CATEGORICAL_WITHOUT_LABELS`	error	A `SingleChoice`/`MultiChoice` variable has no value labels.	Add a `labels={code: label}` mapping to the variable.
`UNUSED_VARIABLE`	warning	A registered variable is never used in the questionnaire.	Remove it from the `VariableMap`, or bind it to a question.

The two `error`-severity rules exist because they signal a measurement-model mistake: a `LikertScale` on a nominal variable or a choice question without value labels will produce data you cannot analyze correctly downstream. This is also why validate(strict=True) promotes them to hard failures — under siamang validate --strict they surface as exit code 2, not 1.

Here is lint in action on a deliberately flawed survey:

```
from siamang.core import Variable, SingleChoice, Page, Questionnaire

    gender = Variable("gender", scale="nominal", label="Gender", labels={1: "Male", 2:
    "Female"})

    survey = Questionnaire(
        title="Demo",
        pages=[
            Page(name="p1", items=[SingleChoice("Gender?", var=gender)]),
            Page(name="p2", items=[]),     # empty page!
        ],
    )

    for w in survey.lint():
        print(f"[{w.severity}] {w.code}: {w.message} ({w.location})")
    # [warning] EMPTY_PAGE: Page 'p2' has no items. (p2)
```

A practical workflow for every authoring session:

1) Write or modify the survey definition.
2) Run `survey.validate()` and fix each `ValueError` until it returns cleanly.
3) Run `survey.lint(level="strict")` and work through the findings — treating `error` severities as mandatory and weighing each `warning` on its merits.
4) Before handing the survey to the deployment machinery, run `siamang validate my_survey.py --strict` and require exit code `0`.

> **Warning:** Some warnings are genuinely intentional designs. A `REQUIRED_CONDITIONAL` finding, for instance, can be a deliberate choice — required-and-conditional is not *invalid*, merely worth a second look. Lint findings invite judgement; `ValueError`s do not.

> **Try it yourself:** Take a survey you built in an earlier chapter, introduce an empty page and a `LikertScale` bound to a `scale="nominal"` variable, then run `lint()` at both levels. Confirm that the empty page is a warning at basic level, and that the scale mismatch appears only at strict level with severity `error`. Finally run `validate(strict=True)` and observe it raise.

Validating data, not just the design. Everything in this chapter checks the questionnaire *design*. To check a collected or simulated *dataset* against your variable metadata — dtypes, ranges, value labels, weight constraints — use `SurveyData.validate()`, which returns a list of `ValidationIssue` objects; `VariableMap.validate_frame(frame)` performs the same family of checks against a pandas DataFrame (`MISSING_COLUMN`, `OUT_OF_RANGE`, `INVALID_LABEL_VALUE`, and so on). Those data-level gates belong to the analysis workflow and are covered later in the book.

Chapter Summary

- `validate()` enforces structural invariants — unique IDs, resolvable `skip_to`/navigation targets, reachability, acyclic page graphs, evaluable expressions, valid scripts, and registry consistency — raising `ValueError` on the first problem.
- `validate(strict=True)` additionally promotes `error`-severity lint findings into hard failures.
- The siamang validate CLI wraps both gates and communicates through exit codes: 0 clean or warnings only, 1 reserved (error-severity lint findings are promoted to exit code 2 under --strict), 2 a structural failure (validate() or the options check raised a ValueError).
- `lint(level="basic")` checks eleven rules — four structural (EMPTY_QUESTIONNAIRE, EMPTY_PAGE, REDUNDANT_NAVIGATION, MISSING_NAVIGATION) plus seven codebook and logic consistency codes; level="strict" adds four more (`REQUIRED_CONDITIONAL`, `INCOMPATIBLE_QUESTION_SCALE`, `CATEGORICAL_WITHOUT_LABELS`, `UNUSED_VARIABLE`), two of which carry `error` severity.
- A LintWarning carries code, severity, message, and location; severity is "warning" or "error".
- Lint findings invite judgement — some are intentional designs — while `ValueError`s must always be fixed.
- Design-level validation is the gate before deployment; dataset-level validation (`SurveyData.validate`, `VariableMap.validate_frame`) is a separate, later step.

With your survey authored, themed, and passing both quality gates, you are ready for Part III. The next chapter shows what happens when you call `survey.deploy(...)` — and how validation and linting form the final checkpoint before your survey goes live.

Documentation References

- `wiki/Validation-and-Linting.md` — `validate(strict=)` checks and `ValueError` semantics, `lint(level=)` and the basic/strict rule tables, `LintWarning` fields, and the `siamang validate` CLI with its exit codes.
- `docs/reference/core.md` — the core API surface (validate, lint, LintWarning).
- `wiki/Pages-Blocks-and-Structure.md` — the `Questionnaire.validate()`/`lint()` method entries and the navigation invariants validation enforces.
- `wiki/Variables-and-Measurement.md` — measurement-scale rules underlying `INCOMPATIBLE_QUESTION_SCALE` and `CATEGORICAL_WITHOUT_LABELS`, and `VariableMap.validate_frame` for data-level checks.

Part III — Deployment and Data

Chapter 12. Deploying Surveys

Part II ended with a survey that is authored, themed, and validated. Part III begins with the moment that survey meets its respondents. In Siamang, deployment is a single method call: you name a **backend** (where responses are stored) and a **frontend** (where the static survey is hosted), pass any credentials, and receive a `DeployResult` carrying the live URL. This chapter explains the deploy model, walks through the three bundled backends and three bundled frontends from a user's perspective — what credentials you need and where to put them — and shows how to pull accumulated responses back into Python for analysis.

Learning Objectives

By the end of this chapter, you will be able to:

- Call `survey.deploy(...)` with a backend, a frontend, adapter keyword arguments, and compile-time options.
- Explain how backend and frontend names are resolved through entry points, and list the available adapters programmatically.
- Choose among the `local`, `supabase`, and `gsheets` backends using their capability matrix, and supply each one's credentials via kwargs or environment variables.
- Choose among the `local`, `vercel`, and `netlify` frontends and configure their tokens and options.
- Read a `DeployResult` and use `collect()` to fetch responses as a pandas DataFrame.

12.1 The Deploy Model

*12.1.1 survey.deploy(backend, frontend, backend_kwargs, frontend_kwargs, **options); entry-point resolution*

The one-call entry point lives on the `Questionnaire`:

```
result = survey.deploy(
        backend="supabase",          # name → BackendAdapter via entry points
        frontend="vercel",           # name → FrontendAdapter via entry points
        backend_kwargs={...},        # forwarded to the backend adapter's __init__
        frontend_kwargs={...},       # forwarded to the frontend adapter's __init__
        **options,                   # quota=..., language=... → compile_questionnaire; ui=/runtime= →
    FrontendBuilder
    )  # -> DeployResult
```

Four ideas are packed into this signature:

1) **`backend` and `frontend` are names, not objects.** Each name is resolved to an adapter class through the `siamang.backends` and `siamang.frontends` entry-point groups. Plugins registered under those groups win; if no plugin matches, Siamang falls back to its built-in registry. You therefore deploy against exactly the same names whether the adapter ships with Siamang or comes from a third-party package.
2) **Both default to `"local"`.** A bare `survey.deploy()` writes a local SQLite store and serves the survey from a background FastAPI server — the fastest way to click through a real, working deployment on your own machine.
3) **`backend_kwargs` / `frontend_kwargs` go to the adapters.** Whatever you place in these dictionaries is forwarded verbatim to the corresponding adapter's constructor — for example `{"token": "...", "project_name": "political-trust-2026"}` for the Vercel frontend.
4) Most remaining **options go to compilation. These are the same options survey.compile(...) accepts: quota=[...] for quota enforcement (Chapter 8), language=, one_question_per_page=True (which maps to SurveyJS questionsOnPageMode: "questionPerPage"), show_progress, and allow_back. Two options are the exception: deploy() itself extracts ui=UIConfig(...) (theming, Chapter 10) and runtime= from **options and hands them to the FrontendBuilder — they never reach compile_questionnaire.

Here is the full pattern, taken from the deployment documentation:

```
import siamang as sg

    result = survey.deploy(
        backend="supabase", frontend="vercel",
        backend_kwargs={"url": "https://abc.supabase.co", "anon_key": "...", "service_key":
"..."},
        frontend_kwargs={"token": "...", "project_name": "political-trust-2026"},
    )
    print(result.url, result.dashboard)
    df = result.collect()   # pull accumulated responses later
```

You can inspect the registry and resolve factories yourself:

```
from siamang.deploy import list_backends, list_frontends, backend_factory,
    frontend_factory

    list_backends()    # ['gsheets', 'local', 'supabase']
    list_frontends()   # ['local', 'netlify', 'vercel']

    backend_factory("supabase")   # <class 'SupabaseBackend'>
    frontend_factory("netlify")   # <class 'NetlifyFrontend'>
```

Note: Under the hood, `deploy()` runs a six-step pipeline: compile the questionnaire to a `SurveySchema`; provision the backend (`backend.provision(schema)`); select the

matching client template; build the `SurveyBundle` with the `FrontendBuilder` machinery from Chapter 10; publish the bundle through the frontend adapter; and return a populated `DeployResult`. You will rarely orchestrate this by hand, but knowing the order explains the errors you see — a provisioning error happens before any frontend work begins.

The secret boundary. During provisioning, the backend produces a `BackendConfig` with two dictionaries: `settings` (frontend-safe values such as URLs and anonymous keys) and `internal` (server-only secrets). Only `settings` and `dashboard_url` ever cross into the deployed browser bundle; `internal` stays server-side. This is why it is safe to pass a `service_key` in `backend_kwargs` — it will not leak into the JavaScript your respondents download.

12.2 Backends and Frontends

12.2.1 Backends: local, supabase, gsheets — setup and credentials from a user perspective

Three backends ship with Siamang:

TABLE 12.1

Backend	name	Storage	Key kwargs
`LocalBackend`	`local`	SQLite file	`path="survey.db"`
`SupabaseBackend`	`supabase`	Postgres + RLS	`url`, `anon_key`, `service_key`, `auto_provision=True`
`GoogleSheetsBackend`	`gsheets`	Google Spreadsheet	`credentials_file`, `spreadsheet_id`, `apps_script_url`

Their trade-offs, from the deployment documentation's capability matrix:

TABLE 12.2

Capability	local	supabase	gsheets
Zero external setup	☑	✗	✗
Public web submissions	✗ (preview only)	☑	⚠ (needs Apps Script proxy)
Atomic quota counters	☑	☑	⚠ (small race window)
High concurrency	✗	☑	✗ (~100 req/100s)
Response dashboard	✗	☑	☑ (the spreadsheet)

`local` — zero setup, development only. `LocalBackend(path="survey.db")` creates a SQLite file (its parent directory is created automatically) and provisions three tables: `survey_meta`, `responses`, and `quota_counters`. It enforces quotas with atomic check-and-increment counters, so it behaves like a real backend — it simply is not reachable from the public internet. Use it for development, previews, and offline or air-gapped fielding where you distribute the HTML bundle yourself.

`supabase` — the production default. The constructor accepts `url`, `anon_key`, `service_key`, `table="responses"`, `quota_function="quota-check"`, and `auto_provision=True`. Any credential you omit falls back to an environment variable — `SIAMANG_SUPABASE_URL`, `SIAMANG_SUPABASE_ANON_KEY`, and `SIAMANG_SUPABASE_SERVICE_KEY` (legacy `SURVLIB_*` variants are also accepted) — and the constructor raises `ValueError` if a required value is still empty. Responses land in a single shared `responses` table keyed by `survey_id`; row-level security policies allow anonymous respondents to `INSERT` while only authenticated users can `SELECT` or `DELETE`.

Provisioning has three modes. In the default **auto** mode, Siamang creates the tables for you via an `exec_sql` function, which you create once in your Supabase project (Dashboard → SQL Editor):

```
-- Run once in Supabase Dashboard → SQL Editor:
    CREATE OR REPLACE FUNCTION exec_sql(query TEXT)
    RETURNS VOID AS $$
    BEGIN EXECUTE query; END;
    $$ LANGUAGE plpgsql SECURITY DEFINER;
```

If that function is missing, provisioning raises `SupabaseProvisionError` with instructions. In **manual** mode (`auto_provision=False`) you generate the SQL yourself and run it in the dashboard:

```
from siamang.deploy.backends.supabase import generate_migration_sql

    print(generate_migration_sql())
    # Copy output → Supabase Dashboard → SQL Editor → Run
```

Finally, passing `migration_dir="./migrations"` to `provision()` saves a timestamped `.sql` file — useful with either mode when you want a record of what was applied.

`gsheets` — a spreadsheet as the response store. `GoogleSheetsBackend` takes `credentials_file` (path to a service-account JSON key), `spreadsheet_id` (an existing spreadsheet; a new one is created if empty), `sheet_name="Responses"`, and `apps_script_url`. Setup requires a Google Cloud project with the Google Sheets API and Google Drive API enabled, a service account with a downloaded JSON key, and — if you use an existing spreadsheet — sharing it with the service-account email as Editor. Environment variables mirror the kwargs: `SIAMANG_GSHEETS_CREDENTIALS_FILE` (re-

quired), SIAMANG_GSHEETS_SPREADSHEET_ID and SIAMANG_GSHEETS_APPS_SCRIPT_URL (optional). The backend also needs extra packages, installable as pip install "siamang[gsheets]". Your responses appear as rows in the spreadsheet — with system columns _response_id and _submitted_at prepended — which doubles as the dashboard.

> **Warning:** Direct browser-to-Sheets submission uses a values.append call with no authorization headers and only works if the spreadsheet is publicly writable — which is not recommended. For secure public deployments you **must** route submissions through a Google Apps Script proxy: in the Sheet, open *Extensions → Apps Script*, paste a script that receives POSTs and appends rows, deploy it as a *Web App* executing as "Me" and accessible by "Anyone", and pass its URL as apps_script_url.

Mind the documented limits: roughly 100 requests per 100 seconds per user (switch to Supabase if you expect more than about 50 simultaneous respondents), quota increments that are not truly atomic, and a maximum of 10 million cells per spreadsheet.

12.2.2 Frontends: local, vercel, netlify — tokens and options

Three frontends ship with Siamang:

TABLE 12.3

Frontend	name	Host	Key kwargs
LocalFrontend	local	Background FastAPI server	host, port=0, open_browser
VercelFrontend	vercel	Vercel	token, team_id, project_name
NetlifyFrontend	netlify	Netlify CDN	token, site_id, site_name

local. Serves the bundle on all interfaces (host="0.0.0.0" by default; pass host="127.0.0.1" to bind loopback only) and forwards POST /responses and POST /quota-check to the backend. With port=0 it picks a free port; open_browser=True opens the survey for you. This is the frontend behind siamang preview, which blocks until Ctrl+C.

vercel. Accepts token (falling back to the VERCEL_TOKEN environment variable), an optional team_id, and project_name="siamang-survey". Publishing follows a three-step fallback strategy: (1) the Vercel REST API when a token is set; (2) the npx vercel CLI when a token is set but the REST path is unavailable; (3) without a token, the bundle is written straight to .vercel_deploy_<survey_id>/ for you to deploy manually, and the local path is returned. Every publish injects a strict vercel.json with a

Content-Security-Policy, `X-Frame-Options: DENY`, cache-control for the content-hashed assets; when your UIConfig sets enable_analytics=True, the Vercel Web Analytics script is injected into the bundle's page itself (the CSP already allows va.vercel-scripts.com).

`netlify`. Accepts `token` (falling back to `NETLIFY_AUTH_TOKEN`; the legacy alias `SIAMANG_NETLIFY_TOKEN` is also honoured), `site_id` for an existing site (a new one is created when empty), and `site_name="siamang-survey"`. Deployment has the same three-mode shape: a REST API ZIP upload that polls until the deploy is ready (requires a token), an `npx netlify deploy --prod` CLI fallback, and a local fallback writing `.netlify_deploy_<survey_id>/` for Netlify Drop or manual CLI use. Security headers are injected via a `_headers` file (Content-Security-Policy, `X-Frame-Options: DENY`, `X-Content-Type-Options: nosniff`, and others), and SPA routing is handled by a `_redirects` rule.

A complete Google Sheets + Netlify deployment looks like this:

```
result = survey.deploy(
        backend="gsheets",
        frontend="netlify",
        backend_kwargs={
            "credentials_file": "./key.json",
            "apps_script_url": "https://script.google.com/macros/s/AKfyc.../exec",
        },
        frontend_kwargs={"token": "nfp_...", "site_name": "my-research-survey"},
    )
    print(result.url)        # https://my-research-survey.netlify.app
    print(result.dashboard)  # https://docs.google.com/spreadsheets/d/...
```

Tip: Prefer environment variables (or the `~/.siamang.toml` config written by `siamang init`) over hard-coding tokens in survey scripts. Adapters initialise straight from the environment, so `backend_factory("gsheets")()` picks up `SIAMANG_GSHEETS_CREDENTIALS_FILE` and `SIAMANG_GSHEETS_SPREADSHEET_ID` with no kwargs at all — and your secrets stay out of version control.

Both the wiki and the API reference agree on recommended pairings:

TABLE 12.4

Use case	Backend	Frontend
Local development / testing	`local`	`local`
Small survey, shared with a team	`gsheets`	`netlify`
Production, high concurrency	`supabase`	`vercel` or `netlify`
Offline / air-gapped (HTML bundle)	`local`	`local`

Try it yourself: Deploy a validated survey with a bare `survey.deploy()` and open result.url. Submit a few test responses, then use `LocalBackend(path="survey.db").get_responses(survey_id=...)` (the survey ID is available as result.survey_id; the siamang preview and siamang deploy CLI commands print both) to read them back. This is exactly the loop `siamang preview` automates.

12.3 After Deployment

12.3.1 DeployResult.collect(): pulling responses as a DataFrame

`deploy()` returns a frozen `DeployResult` — your handle on the live survey:

TABLE 12.5

Field	Type	Description
`url`	`str`	The public survey URL to distribute.
`backend` / `frontend`	`str`	The adapter names used.
`survey_id`	`str`	Identifier linking responses to this deployment.
`dashboard`	`str | None`	A dashboard URL when the backend offers one (Supabase project, Google Sheet).
`deployed_at`	`datetime`	When the deployment happened.
`backend_ref` / `frontend_ref`	adapter or `None`	Cached adapter instances.
`extras`	`dict`	Additional adapter-specific data.

The method you will use most is `collect()`. It reuses the cached `backend_ref` to fetch accumulated responses as a pandas DataFrame — no separate credentials, no export step. One caveat: on the Supabase backend it delegates to `get_responses()`, whose default limit is 1000 rows; for larger surveys call `result.backend_ref.get_all_responses(result.survey_id)`, which paginates through everything:

```
responses = result.collect()
    data = sg.SurveyData(frame=responses, variables=survey.variables, questionnaire=survey)
    print(data.report.freq("trust").to_markdown())
```

`collect()` raises `RuntimeError` if the backend reference is missing, which can only happen for a hand-built `DeployResult`; anything returned by `deploy()` carries the reference. Because the result is a plain DataFrame, you can also collect at several points during fielding to monitor progress, and construct the `SurveyData` only when you are ready to analyze.

> **Note:** Deployments are fresh instances. As you saw with quotas in Chapter 8, each `deploy()` provisions a new survey instance — response counts and quota counters do not carry over from a previous deployment. Keep the `survey_id` (and, for `local`, the SQLite file) of any deployment whose data you intend to keep.

The DataFrame you get back, wrapped in a `SurveyData` with your `VariableMap` attached, is precisely the object the next two chapters are about: Chapter 13 uses `survey.simulate(...)` to produce the same kind of `SurveyData` synthetically before fielding, and Chapter 14 explores the analysis, reporting, and validation toolkits you can run on it.

Chapter Summary

- `survey.deploy(backend, frontend, backend_kwargs, frontend_kwargs, **options)` is the single entry point; names resolve through the `siamang.backends`/`siamang.frontends` entry-point groups with plugins taking precedence over built-ins.
- Both names default to `"local"`, so `survey.deploy()` gives you a working SQLite-backed deployment with zero setup.
- Compile-time options — quota, language, one_question_per_page, show_progress, allow_back — are forwarded through deploy() to compilation; deploy() itself consumes ui and runtime, passing them to the FrontendBuilder.
- Backends: `local` (SQLite, development only), `supabase` (Postgres + RLS, production default, auto/manual/migration-file provisioning, env-var credential fallback), `gsheets` (spreadsheet store; requires an Apps Script proxy for secure public submissions and has documented rate and atomicity limits).
- Frontends: `local` (background server), `vercel` and `netlify` (token via kwarg or environment variable; token-based REST API → CLI fallback, local folder when no token is set; strict security headers injected automatically).
- The `BackendConfig` secret boundary guarantees only frontend-safe `settings` reach the browser bundle; `internal` secrets stay server-side.
- `DeployResult` carries `url`, `survey_id`, and `dashboard`; `collect()` fetches accumulated responses as a pandas DataFrame, ready to wrap in a `SurveyData`.

Documentation References

- `wiki/Deployment.md` — the deploy model, entry-point resolution, backend capability matrix, recommended backend/frontend combinations, and env-var initialisation examples.
- `docs/reference/deploy.md` — adapter constructor signatures and defaults, Supabase provisioning modes, the Apps Script security notice, frontend fallback strategies and injected headers, `DeployResult` fields, and the `DeployPipeline` step order.
- `wiki/CLI-Reference.md` — `siamang preview` and `siamang deploy` commands, and reading preview-collected responses via `LocalBackend.get_responses`.
- `wiki/Cookbook.md` — compile/deploy options (`one_question_per_page`, `show_progress`, `allow_back`) accepted by both `compile()` and `deploy()`.

Chapter 13. Simulating Respondents

Learning Objectives

By the end of this chapter, you will be able to:

- Explain why generating synthetic respondents is a valuable step before fielding a survey.
- Call `simulate()` on a Questionnaire with the `n` and `seed` parameters and interpret the result.
- Describe how simulated values are drawn for each Question type.
- Predict the missingness pattern that page- and question-level skip logic produces in simulated data.
- Use a simulated SurveyData to test your analysis code, report layouts, and validation pipeline end to end.

13.1 Why Simulate

13.1.1 Testing skip logic, quotas, and analysis before fielding

Chapter 12 showed you how to deploy a Questionnaire with `survey.deploy(...)` and later pull the accumulated responses with `DeployResult.collect()`. That workflow raises an obvious question: how do you know the whole pipeline works *before* you invite a single real respondent? Fielding a survey costs time, money, and respondent goodwill. Discovering after launch that a page's `show_if` hides the wrong questions, or that your analysis script crashes on the column layout you actually collect, is expensive.

The answer is simulation. Every Questionnaire has a `simulate()` method that generates *synthetic respondents* — imaginary participants whose answers are drawn at random but who otherwise move through the survey much as real respondents would. The result is a fully populated **SurveyData** (the data container you met in Chapter 12, covered in depth in Chapter 14) with the questionnaire's VariableMap and the questionnaire itself already attached. If the questionnaire has no explicit `variables` registry, one is built on the fly from the variables bound to its questions.

> **Warning:** Simulation produces values drawn at random, *not* from any modeled correlation structure. Use it to exercise plumbing, validate logic, and preview report layouts — not to study real associations. A crosstab computed on simulated data tells you nothing about the population; it only tells you that your crosstab code runs.

Because a simulated SurveyData is structurally identical to one built from `result.collect()`, everything you learn in Chapters 14 through 18 — cleaning, recoding, validation, analysis, reporting, plotting — can be rehearsed today, on synthetic data, with zero respondents.

13.2 Using simulate()

13.2.1 simulate(n=100, seed=42): parameters, reproducibility, skip-logic awareness

The method signature is deliberately small:

```
def simulate(self, n: int = 100, seed: int | None = 42) -> "SurveyData": ...
```

TABLE 13.1

Parameter	Default	Meaning
`n`	`100`	Number of synthetic respondents (rows) to generate.
`seed`	`42`	Seed for the random number generator, for reproducibility. Pass `None` for a fresh, different draw on every call.

The default `seed=42` means that two calls to `survey.simulate()` — by you today, by a colleague next week, or by a test suite in continuous integration — produce *identical* data. This is what makes simulation suitable for unit tests and reproducible documentation. When you deliberately want variation, pass `seed=None`.

A first simulation takes three lines:

```
import siamang as sg

    data = survey.simulate(n=100, seed=42)
    print(data.frame.shape)   # (100, <number of variables>)
```

How values are drawn. The simulator picks a plausible value for each Question type. It does not model distributions, correlations, or respondent heterogeneity — it simply guarantees that every generated value is *valid* for its question:

TABLE 13.2

Question type	Simulated value
`NumericInput`	Uniform integer within the variable's `valid_range` (or 18–70 if no range is set)
`LikertScale`	Uniform integer in `1..points`
`SingleChoice`	A random option code
`MultiChoice`	A random subset honoring `min_answers`/`max_answers`/`exclusive`
`Ranking`	A random ranked subset up to `max_ranked`
`Matrix`	A random code per sub-variable from its labels
`OpenText`	The placeholder string `"sample text"`

Note how much of your question configuration is honored: numeric bounds, Likert point counts, and the `min_answers`/`max_answers`/`exclusive` constraints of multi-choice questions all shape the draw. This is why simulation is a genuine test of your instrument, not just a random-number dump.

Skip logic is respected. Simulation is skip-logic aware — with one caveat. The simulator models only page- and question-level visibility gates defined as Expressions. The routing directives (skip_to/next_if/default_next), string-form conditions, and option gates that the deployed runtime executes are not reproduced by simulation — a string condition is treated as always true — so for surveys that use routing, the skip pattern in real data will differ from the simulated one.

- A Page's `show_if`/`hide_if` (conditional Expression controlling visibility) is evaluated against the answers collected *so far* for that respondent. If the page is hidden, **every variable on it is `NaN`** for that respondent.
- Each Question's own `show_if`/`hide_if` is likewise evaluated; hidden questions produce `NaN`.

The result reproduces the *missingness pattern* your real data will have. If 40% of simulated respondents skip a page because of a consent gate, roughly 40% of real respondents will have `NaN` in those columns too — and your analysis code must cope with that.

> **Note:** Page-level skip logic applies only in **pages mode** (a Questionnaire built with `pages=[...]`). In legacy flat (`blocks`) mode, every question is answered for every respondent, so no structural `NaN` values appear.

13.2.2 A complete worked example

The following survey has a consent gate: only respondents who answer "Yes" see the main page. Simulating it demonstrates both reproducibility and the skip-logic missingness pattern:

```
from siamang.core import Variable, SingleChoice, LikertScale, Page, Questionnaire

    consent = Variable("consent", scale="nominal", label="Consent", labels={1: "Yes", 0: "No"})
    autonomy = Variable("autonomy", scale="ordinal", label="Autonomy",
                        labels={1: "Very low", 2: "Low", 3: "Moderate", 4: "High", 5: "Very high"})

    survey = Questionnaire(
        title="Autonomy Study",
        pages=[
            Page(name="consent", items=[SingleChoice("Do you consent?", var=consent, required=True)]),
            Page(name="main", items=[LikertScale("How much autonomy?", var=autonomy, points=5)],
                 show_if=consent.eq(1)),   # only consenters see this page
        ],
    )

    data = survey.simulate(n=200, seed=123)

    print(data.frame.shape)                # (200, 2)
    print(data.frame["consent"].value_counts(dropna=False).to_dict())
    print(int(data.frame["autonomy"].isna().sum()))   # NaN for every consent != 1
```

Follow the workflow step by step:

1) Define the Variables and assemble the Questionnaire with its pages and visibility conditions, exactly as you would for fielding.
2) Call `survey.simulate(n=200, seed=123)` to obtain a SurveyData.
3) Inspect `data.frame.shape` — it should be `(200, 2)`, one row per respondent and one column per variable.
4) Check the skip pattern: `data.frame["autonomy"].isna().sum()` counts respondents who never saw the autonomy question. Every one of them has `consent != 1`.
5) Re-run with the same `seed` and confirm the numbers are identical; re-run with `seed=None` and confirm they change.

   Tip: If your simulated `NaN` count looks wrong — for example, *everyone* is missing a variable, or *no one* is — the problem is almost always in your `show_if` Expression, not in the simulator. Simulation is an early warning system for exactly this class of bug.

13.2.3 Putting simulated data to work

Because the returned SurveyData carries the questionnaire's VariableMap, the full analysis and reporting stack works on it immediately:

```
print(data.report.freq("autonomy").to_markdown())   # publication-ready table from synthetic data
    print(data.describe_variables())                    # completeness per variable
    issues = data.validate()                            # should be clean
    assert all(i.severity != "error" for i in issues)
```

Three uses recur in practice:

- **Unit tests.** Fix `seed=42`, run your analysis and reporting code against the simulated SurveyData, and assert on the output. No network, database, or deployed survey is involved, so the tests are fast and deterministic.
- **Report layout previews.** Generate the frequency tables, banner tables, and charts you plan to publish. Labels, code formats, and column widths can all be reviewed before a single response exists.
- **Reproducible demos and documentation.** Any example you write down — like the ones in this book — can be regenerated byte-for-byte by a reader using the same seed.

Try it yourself: Take the questionnaire you deployed locally in Chapter 12 (or the autonomy example above) and run `survey.simulate(n=500, seed=7)`. Then compute `data.report.freq(...)` for one categorical variable and `data.analysis.mean(...)` for one numeric variable. Finally, change the `show_if` on one page so that it can never be true, re-simulate, and verify that the page's variables are now `NaN` for all 500 respondents.

Try it yourself: Write a three-line "smoke test" for your own survey: simulate with `seed=42`, run `data.validate()`, and assert that no issue has `severity == "error"`. Keep it next to your survey definition and run it after every edit to the questionnaire.

Chapter Summary

- `simulate()` generates synthetic respondents so you can test skip logic, validation, analysis, and report layouts before fielding — cheaply and without real participants.
- The signature is `simulate(n=100, seed=42)`; `n` sets the number of respondents and `seed` makes the draw reproducible (`seed=None` draws fresh data each call).
- The result is a fully populated SurveyData with the VariableMap and questionnaire attached, usable with every tool in Chapters 14–18.
- Values are drawn at random but validly per question type: numeric ranges, Likert points, and multi-choice constraints are all honored; `OpenText` yields the placeholder `"sample text"`.
- Simulated values carry no correlation structure — never interpret statistics computed on them as substantive findings.
- Page- and question-level `show_if`/`hide_if` conditions are evaluated respondent by respondent; hidden items become NaN, reproducing real-data missingness for Expression-gated items (pages mode only; routing, string conditions, and option gates are not simulated).

- Fixed seeds make simulation ideal for unit tests, report previews, and reproducible examples.

Documentation References

- `wiki/Simulation.md` — the `simulate()` API, per-type value drawing, skip-logic behavior, and typical uses.
- `wiki/Working-with-Data.md`, `docs/reference/data.md` — SurveyData structure, `describe_variables()`, and `validate()` used on simulated data.
- `wiki/Deployment.md`, `docs/reference/deploy.md` — context for `DeployResult.collect()` as the real-data counterpart to simulation.

Chapter 14. Importing, Exporting, and Working with Data

Learning Objectives

By the end of this chapter, you will be able to:

- Construct a SurveyData and explain its immutability model, including `with_frame()` and `with_weight()`.
- Locate the five lazy accessors (`processing`, `analysis`, `tables`, `report`, `plot`) and state what each one is for.
- Import and export data in CSV, Excel, SPSS, Stata, and R formats, choosing the right format for metadata preservation.
- Round-trip a dataset with a JSON data dictionary when using metadata-free formats.
- Validate a dataset against its metadata and interpret the resulting issues.
- Recode values, handle user-defined missing codes, and build composite indices.

14.1 SurveyData Fundamentals

Every response dataset you touch in Siamang lives inside a **SurveyData** — a container that binds a pandas DataFrame to the metadata that describes it. You obtained one from `survey.simulate(...)` in Chapter 13 and from `DeployResult.collect()` in Chapter 12 (wrapped with your variables). You can also read one from a file with the `siamang.io` readers, or construct one directly:

```
import pandas as pd
    from siamang.data import SurveyData
    from siamang.core import Variable, VariableMap

    variables = VariableMap()
    variables.add(Variable("gender", scale="nominal", label="Gender", labels={1: "Male", 2:
    "Female"}))
    variables.add(Variable("age", scale="ratio", label="Age", valid_range=(18, 99)))

    frame = pd.DataFrame({"gender": [1, 2, 1, 2], "age": [34, 28, 45, 52]})

    data = SurveyData(frame=frame, variables=variables)
    print(data.analysis.mean("age"))      # 39.75
```

The four fields of the container:

TABLE 14.1

Field	Type	Meaning
`frame`	`pd.DataFrame`	The raw response table, one row per respondent.
`variables`	`VariableMap | None`	The codebook: labels, scales, value labels, missing codes, valid ranges.
`questionnaire`	`Questionnaire | None`	The originating instrument, when known.
`weight`	`str | None`	Name of the default weight column for weighted statistics.

14.1.1 Immutability, with_frame/with_weight, the five lazy accessors

SurveyData is **frozen and immutable**: you never modify it in place. Every transformation — filtering rows, recoding values, setting a weight — returns a *new* SurveyData instance. This makes data pipelines safe to chain and easy to reason about, but it means you must always capture the result:

```
adults = data.with_frame(data.frame[data.frame["age"] >= 40])
```

`with_frame(frame)` replaces the DataFrame while carrying the metadata, questionnaire, and weight over unchanged — use it after any custom pandas operation such as dropping or filtering rows. Its companion, `with_weight(column)`, sets the default weight column used by every `weighted=True` statistic; it raises `ValueError` if the column is not in the frame:

```
weighted = data.with_weight("design_weight")
    weighted.analysis.mean("age", weighted=True)
```

> **Warning:** A bare `data.recode_values(...)` or `data.with_frame(...)` call without assignment silently discards its result. The idiom is always `data = data.some_transformation(...)`.

The power of SurveyData comes from its **five lazy accessors** — toolkits that are created on demand when you first touch them:

TABLE 14.2

Accessor	Type	Purpose
`data.processing`	`DataProcessing`	Ad-hoc value-level transforms
`data.analysis`	`DataAnalysis`	Descriptive and inferential statistics
`data.tables`	`SurveyTables`	Multi-cell banner/cross-break tables
`data.report`	`ReportAccessor`	Declarative, labeled tables
`data.plot`	`PlotAccessor`	Declarative, labeled charts

```
data.analysis.mean("age")
    data.report.freq("gender").to_markdown()
    data.plot.bar("gender").show()
    data.tables.banner(rows=["gender"], columns=["age"])
```

This chapter introduces the container, its metadata tools, and its processing methods. Chapter 15 takes a deep dive into `data.analysis`, and later chapters cover `report`, `plot`, and `tables` in detail.

Inspection. Two methods summarize your metadata without touching the data itself:

- `codebook()` returns a metadata-only DataFrame with one row per registered variable and columns `name`, `label`, `scale`, `dtype`, `role`, `description`, `missing_values`, `missing_kinds`, `missing`, `valid_range`. It raises `ValueError` if no VariableMap is attached.
- `describe_variables()` returns a completeness summary with `name`, `label`, `scale`, plus `n` (rows), `n_missing` (NaN count), and `n_unique`. It also requires metadata.

> **Note:** The reference page `docs/reference/data.md` describes the `codebook()` columns more briefly as `name`, `scale`, `label`, `labels`, `missing_values`. The fuller column list above comes from `wiki/Working-with-Data.md`; treat the exact column set as version-dependent.

14.2 Import and Export

The `siamang.io` layer round-trips datasets between SurveyData and CSV, Excel, SPSS, Stata, and R, preserving variable labels, value labels, missing-value codes and kinds, and column formats *wherever the format allows*. The whole layer follows one convention: every tabular reader (CSV, Excel, SPSS, Stata) exposes read(path, **kwargs) -> SurveyData, and every tabular writer exposes write(data, path, **kwargs) -> Path, returning the path it wrote. DictionaryReader/DictionaryWriter work with a VariableMap instead of SurveyData, and RScriptWriter.write(data, path) takes no extra kwargs. All I/O symbols are also re-exported at the top level, so `from siamang import read_spss` works.

```
from siamang.io import (
        SurveyDataReader,
        CSVReader, CSVWriter,
        ExcelReader, ExcelWriter,
        SPSSReader, SPSSWriter, read_spss,
        StataReader, StataWriter, read_stata,
        RScriptWriter,
        DictionaryReader, DictionaryWriter,
    )
```

14.2.1 CSV/Excel (data only) vs SPSS/Stata (full metadata round-trip)

The central design fact of the I/O layer is that some formats carry metadata and some do not.

CSV and Excel carry data only. Reading produces a SurveyData whose frame is populated but whose metadata is *not* reconstructed:

```
from siamang.io import CSVReader, CSVWriter, ExcelReader, ExcelWriter

    data = CSVReader().read("responses.csv")        # pd.read_csv under the hood
    CSVWriter().write(data, "out.csv")              # frame.to_csv(path, index=False)

    data = ExcelReader().read("responses.xlsx")     # pd.read_excel under the hood
    ExcelWriter().write(data, "out.xlsx")           # frame.to_excel(path, index=False)
```

Any keyword arguments you pass are forwarded to the underlying pandas call (`pd.read_csv`, `pd.read_excel`, and so on). Excel support requires `openpyxl`, which is bundled by default. To recover labels and missing codes after reading CSV or Excel, pair the file with a JSON dictionary (Section 14.2.2).

SPSS .sav and Stata .dta carry rich metadata, so both largely round-trip your codebook — with caveats. Stata only supports single-letter user missing codes (.a–.z), so numeric missing codes (e.g. 99) are dropped on write, and measurement levels are not stored in .dta files; an SPSS round-trip keeps labels, missing codes, and measurement levels but loses missing-value kinds and valid_range. Pair a .dta export with a JSON dictionary (Section 14.2.2) if you need them. Both are powered by `pyreadstat`, which is bundled by default:

```
from siamang.io import read_spss, SPSSWriter, read_stata, StataWriter

    data = read_spss("trust.sav")                    # convenience for
    SPSSReader().read(...)
    SPSSWriter().write(data, "trust_out.sav")

    data = read_stata("trust.dta")
    StataWriter().write(data, "trust_out.dta", version=15)
```

- `SPSSReader.read(path)` reads via `pyreadstat.read_sav(path, user_missing=True)` and rebuilds a VariableMap from the file's column-to-label mapping, value labels, missing ranges, and measurement levels.
- `SPSSWriter.write(data, path)` writes variable labels, value labels, missing values, and measurement levels (nominal/ordinal/scale) via pyreadstat.write_sav. Your `data.variables` must be set, or the columns are written bare.
- `StataReader.read(path)` likewise builds a SurveyData with a VariableMap. `StataWriter.write(data, path, version=15, **kwargs)` forwards version as the target Stata version (8–15 supported, default 15), passed through to pyreadstat.write_dta.

The canonical SPSS recode-and-export round-trip:

```
import pandas as pd
    from siamang.io import read_spss, SPSSWriter

    data = read_spss("input.sav")                       # metadata recovered
    data = data.with_frame(data.frame.replace({"age": {-1: pd.NA}}))  # treat -1 as missing (recode_values
    would write to a new column)
    SPSSWriter().write(data, "output.sav")
```

Auto-detection. If you do not want to pick a reader yourself, `SurveyDataReader` dispatches on the file extension:

```
from siamang.io import SurveyDataReader

    data = SurveyDataReader().read("responses.sav")   # picks the reader by suffix
```

TABLE 14.3

Extension	Reader used
`.csv`	`CSVReader`
`.xlsx`, `.xls`	`ExcelReader`
`.sav`	`SPSSReader`
`.dta`	`StataReader`

Unknown suffixes raise `ValueError`.

Tip: Choose your working format by metadata needs. If your collaborators use SPSS or Stata, use `.sav`/`.dta` end to end and never think about dictionaries. If you must exchange CSV or Excel, always ship a JSON dictionary alongside.

14.2.2 R three-file script bundle; JSON dictionaries

R export writes a three-file bundle into a target directory and returns the `Path` to the R script:

```
from siamang.io import RScriptWriter

    RScriptWriter().write(data, path="political_trust_R/")
```

The bundle contains a CSV of the responses, a JSON serialisation of the full VariableMap, and an R script that reads both, replaces missing-value codes with `NA`, and applies value labels with `factor(...)`, leaving you with a ready data frame. If `path` ends in `.R` (for example `trust.R`), the files are named after its stem (`trust.csv`, `trust_dictionary.json`, `trust.R`). In R, you load everything with one line:

```
source("political_trust_R/import_survey.R")
```

> **Note:** The bundle's file names are deterministic: import_survey.csv, import_survey_dictionary.json, and import_survey.R by default (the script uses jsonlite and leaves an object named survey_data); when path ends in .R, the files are named after its stem (trust.csv, trust_dictionary.json, trust.R).

JSON data dictionaries store the VariableMap itself, independently of any data file:

```
from siamang.io import DictionaryReader, DictionaryWriter

    DictionaryWriter().write(survey.variables, "dict.json")
    restored = DictionaryReader().read("dict.json")     # -> VariableMap
```

`DictionaryWriter.write(variables, path)` serialises `variables.to_dict()` to JSON; `DictionaryReader.read(path)` rebuilds the VariableMap and raises `ValueError` if the JSON root is not an object. Dictionaries solve the CSV/Excel metadata gap. The canonical pipeline:

```
from siamang.io import CSVWriter, DictionaryWriter, CSVReader, DictionaryReader

    CSVWriter().write(data, "out.csv")
    DictionaryWriter().write(data.variables, "out_dict.json")

    # later ...
    again = CSVReader().read("out.csv")
    again = again.__class__(frame=again.frame,
    variables=DictionaryReader().read("out_dict.json"))
```

High-level export helpers. SurveyData bundles the most common exports into two convenience methods:

```
data.export("spss",  path="out.sav")
    data.export("stata", path="out.dta")
    data.export("r",     path="out_R/")
    data.export_dictionary("dict.json")
```

`data.export(fmt, path=None, **kwargs)` supports `"csv"`, `"xlsx"` (alias `"excel"`), `"spss"` (alias `"sav"`), `"stata"` (alias `"dta"`), and "r"; an unknown format raises NotImplementedError. `data.export_dictionary(path) -> Path` writes the VariableMap metadata to a JSON schema file.

14.3 Data Processing

14.3.1 Validation, recoding, missing-value handling, index construction

Validation checks the DataFrame *against its metadata* — a different job from `Questionnaire.validate()`, which checks survey design:

```
def validate(self, raise_on_error: bool = False) -> list[ValidationIssue]: ...
```

It verifies column presence against the VariableMap; dtype compatibility, `valid_range` bounds, and value-label coverage for categorical variables; that the weight column exists and is numeric (when `weight` is set); and that every questionnaire variable is present in the frame (when a questionnaire is attached). Each `ValidationIssue` carries a `code`, a `severity` (`"error"` or `"warning"`), a `message`, and an optional `variable`/`column`. With `raise_on_error=True`, any error-severity issue raises `ValueError`; without a VariableMap you get a single `MISSING_METADATA` warning.

```
bad = SurveyData(frame=pd.DataFrame({"gender": [1, 5], "age": [34, 200]}), variables=variables)
    for issue in bad.validate():
        print(issue.code, "-", issue.message)
    # INVALID_LABEL_VALUE - Variable 'gender' has values not present in labels: 5.
    # OUT_OF_RANGE - Variable 'age' has values outside valid_range (18, 99).
```

Running `validate()` is a good first step after every import, every collection, and every simulation (recall from Chapter 13 that simulated data should validate clean).

Missing-value handling distinguishes *user-defined* missing codes (such as `99` for refusal, declared in the VariableMap) from plain NaN:

- `apply_missing_values(kinds=None)` replaces all user-defined missing codes with `pd.NA`. Pass a set of kinds — for example `{"refusal", "dont_know"}` — to replace only those kinds.
- `drop_missing(column)` returns a new instance with rows removed where the column is `NaN` or `pd.NA`.

Recoding and derivation cover the everyday transformations. Remember that each returns a new SurveyData:

TABLE 14.4

Method	Purpose
`recode(column, *, into, bins, labels=None, right=False, label=None)`	Bin a continuous variable into categories via `pandas.cut`; the new variable is registered with an `"ordinal"` scale.
`recode_values(column, mapping, *, into=None, label=None, scale=None)`	Collapse or remap discrete values (e.g. `{1: 0, 2: 0, 3: 1}`); the source column is never modified — with into, stores the result in a new registered column, otherwise in a new column named <column>_recoded; values absent from mapping become NaN, so list every code you want to keep.
`derive(*, name, expression, label=None, scale="nominal", labels=None)`	Evaluate a logical Expression row by row to create a binary 0/1 indicator variable.

Composite measures combine several items into a scale:

- `scale_alpha(items)` computes Cronbach's alpha (requires at least two items) — the standard reliability check before combining items.
- `create_index(name, *, items, method="mean", label=None)` builds the composite itself, by `"mean"` or `"sum"`, and registers the new variable with an `"interval"` scale.

A typical workflow: simulate or import data, validate it, apply missing codes, check `scale_alpha(...)` on your multi-item measure, then `create_index(...)` and analyze the composite.

The `processing` accessor is the low-level escape hatch: `data.processing.recode(column, mapping)` applies a raw {old: new} mapping and returns a new SurveyData carrying only the recoded frame — the VariableMap, questionnaire, and weight are dropped. Prefer `SurveyData.recode_values()` for metadata-aware recoding, and reach for `processing` only when you deliberately want a metadata-free transformation.

> **Tip:** Adopt the order *import* → *validate* → *apply_missing_values* → *recode/derive* → *analyze*. Validating before cleaning catches structural problems (missing columns, out-of-range codes) while they are still easy to attribute to the source file.

> **Try it yourself:** Export the simulated SurveyData from Chapter 13 twice: once via `CSVWriter` plus `DictionaryWriter`, and once via `SPSSWriter`. Read both back. Run `describe_variables()` on each restored dataset and confirm that the SPSS round-trip preserves labels, missing codes, and measurement levels (though not missing-value kinds

or valid_range), while the CSV version only recovers its labels after you attach the dictionary.

Try it yourself: Using any dataset with a numeric variable, call `validate()` on the raw data, then deliberately introduce an out-of-range value with a pandas assignment and `with_frame(...)`, and call `validate()` again. Confirm that you receive an `OUT_OF_RANGE` issue with severity `"error"`, and that `validate(raise_on_error=True)` raises `ValueError`.

Chapter Summary

- SurveyData binds a DataFrame to a VariableMap, an optional questionnaire, and an optional weight column; it is frozen, so every transformation returns a new instance that you must assign.
- `with_frame()` swaps the DataFrame after custom pandas work; `with_weight()` sets the default weight column for `weighted=True` statistics.
- Five lazy accessors — `processing`, `analysis`, `tables`, `report`, `plot` — provide all computation on the data; `analysis` is deepened in Chapter 15.
- The siamang.io tabular readers/writers (CSV, Excel, SPSS, Stata) follow one convention: read(path) -> SurveyData and write(data, path) -> Path; DictionaryReader/DictionaryWriter handle a VariableMap and RScriptWriter takes no extra kwargs; SurveyDataReader auto-detects by extension.
- CSV and Excel carry data only; SPSS and Stata round-trip full metadata via `pyreadstat`. Pair CSV/Excel with a JSON dictionary (`DictionaryWriter`/`DictionaryReader`) to preserve the codebook.
- `RScriptWriter` emits a three-file bundle (CSV + dictionary JSON + R loader script); the file names are deterministic — import_survey.* by default, or named after the stem when path ends in .R.
- `validate()` checks the frame against its metadata and reports `ValidationIssue` objects with codes and severities; `raise_on_error=True` turns errors into exceptions.
- `apply_missing_values()`, `drop_missing()`, `recode()`, `recode_values()`, `derive()`, `scale_alpha()`, and `create_index()` form the core cleaning-to-composite pipeline.

Documentation References

- `wiki/Data-Import-and-Export.md`, `docs/reference/io.md` — the `siamang.io` readers and writers, format capabilities, R bundle, and dictionary round-trips.
- `wiki/Working-with-Data.md`, `docs/reference/data.md` — SurveyData structure, accessors, `with_frame`/`with_weight`, inspection, validation, and processing methods.
- `wiki/Cookbook.md` — the high-level `data.export(...)` helpers (including the `"spss"` variant).
- `wiki/Simulation.md` — simulated SurveyData as practice input for everything in this chapter.
- `wiki/API-Reference-Index.md` — top-level re-exports of the I/O symbols.

Part IV — Analysis and Reporting

Chapter 15. Analyzing Survey Data

Learning Objectives

By the end of this chapter, you will be able to:

- Compute descriptive statistics with `data.analysis`: `mean`, `median`, and `grouped_mean`.
- Produce weighted statistics by setting a default weight column with `with_weight()` and passing `weighted=True`.
- Run non-parametric group-comparison tests with `kruskal()` and `mannwhitney()` and interpret their result dictionaries.
- Explain the automatic test-selection logic (t-test/ANOVA versus Mann–Whitney/Kruskal–Wallis) that Siamang applies based on measurement scale and group count.
- Assess the impact of weighting with Kish's `effective_sample_size()`.
- Choose between `data.processing.recode` and the metadata-aware `SurveyData.recode_values`, `recode`, and `derive` when preparing data for analysis.

15.1 Descriptive Statistics

Chapter 14 introduced the SurveyData container and its five lazy accessors. This chapter opens Part IV by putting the first of them to work. Two accessors matter for hands-on analysis:

- **`data.analysis`** → `DataAnalysis` — descriptive and inferential statistics that are aware of survey weights and variable metadata.
- **`data.processing`** → `DataProcessing` — quick value-level recoding.

Everything in this chapter runs on any SurveyData — collected, imported, or simulated. Following the documentation, the examples use a simulated "Work Study" survey with four variables: `age` (ratio), `it_role` (nominal), `remote_freq` (ordinal, five categories from "Never" to "Fully remote"), and `autonomy` (ordinal, five-point Likert):

```
from siamang.core import Variable, VariableMap, SingleChoice, NumericInput, LikertScale, Page, Questionnaire

    age = Variable("age", scale="ratio", label="Age", valid_range=(18, 75))
    it_role = Variable("it_role", scale="nominal", label="IT Role",
                       labels={1: "Engineer", 2: "Data Scientist", 3: "DevOps", 4: "PM"})
    remote_freq = Variable("remote_freq", scale="ordinal", label="Remote Frequency",
                           labels={1: "Never", 2: "Occasionally", 3: "Hybrid", 4: "Mostly remote", 5: "Fully
    remote"})
    autonomy = Variable("autonomy", scale="ordinal", label="Autonomy",
                        labels={1: "Very low", 2: "Low", 3: "Moderate", 4: "High", 5: "Very high"})

    variables = VariableMap()
    variables.add_many([age, it_role, remote_freq, autonomy])

    survey = Questionnaire(
        title="Work Study",
        pages=[Page(name="main", items=[
            NumericInput("Age?", var=age),
            SingleChoice("Role?", var=it_role),
            SingleChoice("Remote?", var=remote_freq),
            LikertScale("Autonomy?", var=autonomy, points=5),
        ])],
        variables=variables,
    )
    data = survey.simulate(n=200, seed=123)
```

The accessor itself is a small frozen dataclass that carries exactly what the statistics need — the frame, the weight column, and the metadata:

```
@dataclass(frozen=True, slots=True)
    class DataAnalysis:
        frame: pd.DataFrame
        weight_column: str | None = None
        variables: VariableMap | None = None
```

You never construct `DataAnalysis` by hand; `data.analysis` builds it lazily from the SurveyData, picking up the current weight column and VariableMap. The practical consequence is that the accessor always reflects the SurveyData you took it from: after `with_weight("w")`, the new dataset's `analysis` accessor is weight-aware, while the original dataset's accessor is not.

> **Warning:** Recall from Chapter 13 that simulated values are random draws with no modeled correlation structure. Every number below is valid for learning the API, but none of it is a substantive finding.

15.1.1 data.analysis: mean, median, grouped_mean; weighted statistics

The three core descriptive methods:

TABLE 15.1

Method	Signature	Returns
`mean`	`mean(column, weighted=False) -> float`	Arithmetic mean of a column (NaNs dropped).
`median`	`median(column) -> float`	Median of a column (NaNs dropped).
`grouped_mean`	`grouped_mean(column, by, weighted=False, labels=False) -> pd.DataFrame`	Mean of `column` within each category of `by`.

```
data.analysis.mean("autonomy")    # 3.06
    data.analysis.median("age")
```

`grouped_mean` is the descriptive workhorse: it compares a continuous measure across the categories of a grouping variable and returns a DataFrame with `group`, `mean`, and `n` columns. Passing `labels=True` adds a `label` column that resolves each group's value label from the VariableMap:

```
print(data.analysis.grouped_mean("autonomy", by="remote_freq", labels=True))
    #    group      mean     n          label
    # 0      1  3.222222  45.0          Never
    # 1      2  2.923077  39.0   Occasionally
    # 2      3  2.897959  49.0         Hybrid
    # 3      4  3.000000  31.0  Mostly remote
    # 4      5  3.277778  36.0   Fully remote
```

This raw output is ideal for further computation in pandas. When you want the same comparison *formatted and significance-tested* for a report, prefer the `data.report.means(...)` table covered in Chapter 16 — the documentation explicitly points you there for the publication version.

Weighted statistics. Survey samples often over- or under-represent population groups, and design weights correct for that. Siamang's model, introduced in Chapter 14, is: set the weight column once, then opt in per call.

1) Attach the weight column with `with_weight("column_name")`, which returns a new SurveyData.
2) Pass `weighted=True` to any supporting method.

```
import numpy as np

    # attach a synthetic design weight for illustration
    weighted = data.with_frame(data.frame.assign(w=np.linspace(0.5, 1.5, len(data.frame))))
    \
                    .with_weight("w")

    weighted.analysis.mean("autonomy", weighted=True)
```

With `weighted=True`, `mean` uses the configured weight column; calling it without a configured weight raises `ValueError`. In `grouped_mean(..., weighted=True)`, the `n` column becomes the *summed weight* per group rather than a raw count. Besides `mean` and `grouped_mean`, the analysis methods `frequencies`, `crosstab`, and `proportion_ci` are documented as weight-aware (they accept `weighted=True` and use the default weight column), though the documentation does not specify their full signatures; the labeled, publication-ready equivalents in Chapter 16 cover the same ground.

> **Tip:** Keep the unweighted and weighted SurveyData in separate variables (as with `data` and `weighted` above). Because SurveyData is immutable, both coexist cheaply, and you can compare weighted against unweighted results to judge how much the weights move your estimates.

15.2 Statistical Tests

15.2.1 kruskal, mannwhitney; automatic test selection logic; Kish effective_sample_size()

The analysis accessor exposes two **non-parametric** tests for comparing a continuous or ordinal measure across independent groups. Both require scipy (which ships with the base install and raises `ImportError` if somehow missing), and both return a plain dictionary rather than an object.

TABLE 15.2

Method	Test	Use when	Returns
`kruskal(column, group)`	Kruskal–Wallis H-test	3+ independent groups	`{"statistic", "p_value", "groups"}`
`mannwhitney(column, group)`	Mann–Whitney U-test (two-sided)	Exactly 2 independent groups	`{"statistic", "p_value", "group_a", "group_b"}`

The Kruskal–Wallis H-test is the non-parametric analogue of one-way ANOVA: it asks whether the *distributions* of a variable differ across groups, using ranks instead of raw values, so it makes no normality assumption. It is the safe default for ordinal measures such as Likert scales:

```
data.analysis.kruskal("autonomy", "remote_freq")
    # {'statistic': 2.1330..., 'p_value': 0.7112..., 'groups': 5.0}
```

Here `groups` reports how many groups were compared. The test raises `ValueError` if fewer than two non-empty groups are present.

The Mann–Whitney U-test compares **exactly two** independent groups and raises `ValueError` unless exactly two non-empty groups are present. To compare two levels of a multi-category variable, filter the frame first with `with_frame()`:

```
two_levels = data.with_frame(data.frame[data.frame["remote_freq"].isin([1, 5])])
    two_levels.analysis.mannwhitney("autonomy", "remote_freq")
    # {'statistic': 794.0, 'p_value': 0.8796..., 'group_a': 1, 'group_b': 5}
```

The `group_a`/`group_b` entries tell you exactly which two group values entered the comparison — worth checking whenever you filtered the frame yourself.

Automatic test selection: the logic behind the tests. You will meet these two tests again in Chapter 16, where `GroupMeanTable` runs significance tests for you and *chooses* among four candidates: the independent-samples t-test, one-way ANOVA, Mann–Whitney U, and Kruskal–Wallis H. Because you will rely on that automation constantly, it pays to understand the selection rule now. It uses exactly two pieces of information — both already in your metadata:

1) The measurement scale of the dependent variable. Parametric tests (t-test, ANOVA) compare means and assume interval- or ratio-scaled data. Non-parametric tests (Mann–Whitney, Kruskal–Wallis) compare rank distributions and are appropriate for ordinal data — or whenever the scale is unknown.
2) **The number of groups** in the grouping variable. Two groups call for a two-sample test (t-test or Mann–Whitney); three or more call for the omnibus variant (ANOVA or Kruskal–Wallis).

That yields a simple two-by-two table:

TABLE 15.3

Dependent scale	2 groups	3+ groups
`ordinal` (or scale unknown)	Mann–Whitney U	Kruskal–Wallis H
`interval` / `ratio`	Independent t-test	One-way ANOVA

The direct methods `kruskal()` and `mannwhitney()` you just used are the ordinal row of this table, exposed for when you want explicit control — for example, when you prefer a rank-based test for a skewed ratio variable, or when you are building a custom analysis pipeline. The t-test and ANOVA

branches exist only implicitly through the reporting table's automatic selection; they are not exposed as standalone `DataAnalysis` methods.

> **Note:** This is why declaring honest measurement scales in your VariableMap (Chapter 6) matters beyond documentation. The declared scale is what steers the automatic test selection between the parametric and non-parametric branches.

Kish effective sample size. Weighting reduces the information content of a sample: a weighted sample of 200 respondents typically carries less precision than an unweighted sample of 200. Kish's effective sample size (ESS) quantifies that loss:

```
weighted.analysis.effective_sample_size()   # Kish's ESS = (Σw)² / Σw²
```

The formula (Σw)² / Σw² compares the squared sum of weights to the sum of squared weights. When all weights are equal, ESS equals the actual sample size; the more unequal the weights, the further ESS drops below it. Report ESS alongside weighted estimates so readers can judge effective precision. The method raises `ValueError` if no weight column is set — it only makes sense for a weighted frame.

> **Try it yourself:** Using the Work Study data, filter to `remote_freq` levels 1 and 5 with `with_frame(...)`, run `mannwhitney("autonomy", "remote_freq")`, and note the p-value. Then run `kruskal("autonomy", "remote_freq")` on the *unfiltered* data. Which row of the selection table does each correspond to, and which test would `GroupMeanTable` pick in each case?

15.3 Recoding for Analysis

Real analysis rarely works on raw codes. You collapse scales, band ages, and build indicators before you estimate anything. Siamang offers two tiers of recoding tools, and choosing the right tier is mostly a question of whether your metadata should come along.

15.3.1 data.processing.recode; SurveyData.recode_values/recode/derive

Tier 1: the quick wrapper. `data.processing.recode` applies a raw `{old: new}` mapping to a column and returns a new SurveyData:

```
def recode(self, column: str, mapping: dict[Any, Any]) -> SurveyData: ...

# Collapse the 5-point remote-frequency scale into 3 levels
    collapsed = data.processing.recode("remote_freq", {1: 1, 2: 1, 3: 2, 4: 3, 5:
    3})
```

It is a thin convenience wrapper: the values change, but the variable metadata is not carried over at all: the returned SurveyData keeps only the recoded frame — the VariableMap, the questionnaire, and any configured weight are dropped. Your next labeled table on the result will show raw codes instead of labels, and weighted statistics lose the weight. Reach for this only in throwaway exploration, where labels do not matter.

Tier 2: metadata-aware recoding. The documentation explicitly recommends the three methods living on SurveyData itself for anything you will keep:

TABLE 15.4

Method	Purpose	Metadata effect
`SurveyData.recode_values(column, mapping, *, into=None, label=None, scale=None)`	Collapse/remap discrete codes	The source column is never modified: with into, the result is registered under that new name; without into, a new variable <column>_recoded is registered either way.
`SurveyData.recode(column, *, into, bins, labels=None, right=False, label=None)`	Bin a continuous variable into ordinal categories via `pandas.cut`	New variable registered with an `"ordinal"` scale.
`SurveyData.derive(*, name, expression, label=None, scale="nominal", labels=None)`	Build a 0/1 indicator from a logical Expression	New variable registered with the given metadata.

The binning example shows what "metadata-aware" buys you. Band `age` into brackets and the new variable arrives fully registered — with value labels ready for reporting:

```
# Metadata-aware: bin age into ordinal brackets, registered as a new variable
    banded = data.recode("age", into="age_band", bins=[18, 30, 45, 75],
                         labels=["18-29", "30-44", "45+"], label="Age band")
    banded.variables["age_band"].labels   # {1: '18-29', 2: '30-44', 3: '45+'}
```

Because `age_band` is a proper Variable with an ordinal scale and value labels, everything downstream works immediately: `banded.report.freq("age_band")` prints labeled categories, `grouped_mean(..., by="age_band", labels=True)` resolves its labels, and `validate()` checks its values against its metadata.

The same logic applies to the other two methods. Use `recode_values` when you collapse existing codes — for instance, merging the two ends of a five-point agreement scale into "disagree/neutral/agree" for a simpler breakdown — passing into only when you want a custom name for the new column (the original column is preserved either way). Use `derive` when the new variable is a yes/no

condition best expressed as a logical Expression (the same Expression objects you used for skip logic in earlier chapters): the result is a clean 0/1 indicator you can tabulate or use as a grouping variable.

A practical pre-analysis checklist. Before running any of the statistics in Sections 15.1 and 15.2:

1) Validate the raw data (`data.validate()`) so structural problems surface first.
2) Replace user-defined missing codes with `apply_missing_values()`.
3) Recode with the metadata-aware methods, so every derived variable is registered.
4) Attach weights with `with_weight()` if the design calls for them.
5) Check `effective_sample_size()` to understand the weighting cost.
6) Only then compute means, grouped means, and tests.

Warning: All of these methods return a *new* SurveyData. A line like `data.recode("age", into="age_band", ...)` without assignment computes the recoded dataset and immediately discards it. The idiom is always `data = data.recode(...)` or a new variable name.

Try it yourself: Collapse `remote_freq` into three levels twice: once with `data.processing.recode` and once with data.recode_values(..., into="remote3"). Run `data.report.freq(...)` on both results (you will meet `FreqTable` formally in Chapter 16) and compare how the two outputs are labeled. Which version would you put in a report, and why?

Chapter Summary

- `data.analysis` provides descriptive statistics that respect NaN exclusion, variable metadata, and optional design weights: `mean`, `median`, and `grouped_mean` (with `labels=True` for readable group labels).
- Weighted analysis follows a two-step model: `with_weight("col")` once, then `weighted=True` per call; weighted `grouped_mean` reports summed weights in its `n` column.
- `kruskal()` (3+ groups) and `mannwhitney()` (exactly 2 groups) are rank-based, non-parametric tests returning plain dictionaries with statistics, p-values, and the groups compared.
- Automatic test selection — shared with the reporting tables of Chapter 16 — uses the dependent variable's declared scale and the group count: ordinal/unknown scales get Mann–Whitney or Kruskal–Wallis; interval/ratio scales get t-test or ANOVA.

- `effective_sample_size()` reports Kish's ESS = $(\Sigma w)^2 / \Sigma w^2$, quantifying how much precision weighting costs; it requires a configured weight column.
- `data.processing.recode` is a quick, metadata-free value mapping for exploration; for anything kept, prefer the metadata-aware `SurveyData.recode_values`, `recode`, and `derive`.
- Metadata-aware recoding registers new variables with scales and value labels, so validation, grouping, and labeled reporting work on derived variables without extra setup.

Documentation References

- `wiki/Analysis.md` — the `data.analysis` and `data.processing` accessors, descriptive methods, inferential tests, weighted statistics, Kish ESS, and recoding guidance.
- `wiki/Working-with-Data.md`, `docs/reference/data.md` — `with_weight()`/immutability semantics and the SurveyData-level `recode_values`/`recode`/`derive` methods (via Brief 03).
- `wiki/Reporting-Tables.md` — the `GroupMeanTable` automatic test-selection rule previewed here and covered fully in Chapter 16.

Chapter 16. Tables and Banner Tables

In the previous chapter you computed statistics one at a time with `data.analysis` — a mean here, a Kruskal–Wallis test there. Those methods return raw numbers, DataFrames, and dictionaries, which is exactly what you want while exploring. But eventually you need output you can hand to a colleague, paste into a paper, or attach to an email: formatted frequency tables, cross-tabulations with significance tests, and wide banner tables that profile every subgroup at once.

Siamang provides two complementary table layers for this. The **reporting tables** — `FreqTable`, `CrossTable`, and `GroupMeanTable`, reached through the `data.report` accessor — produce single, labeled, publication-ready tables in Markdown, HTML, DataFrame, or Excel form. The **banner table**, built through the `data.tables` accessor, stacks many cross-breaks into one long, tidy, export-ready frame. This chapter covers both, and shows you how to choose between them.

Learning Objectives

By the end of this chapter, you will be able to:

- Build a labeled frequency table with counts, percentages, and cumulative percentages using `data.report.freq`.
- Produce a two-way cross-tabulation with row, column, or total percentages and an automatic Chi-square test using `data.report.crosstab`.
- Compare group means with `data.report.means` and explain how the significance test is selected automatically from the measurement scale and group count.
- Render any reporting table as a DataFrame, Markdown, HTML, or Excel file.
- Construct a weighted, multi-variable banner table with `data.tables.banner` and export it to CSV or XLSX.
- Decide when a banner table is the right tool and when a single `CrossTable` serves you better.

16.1 Reporting Tables

Declarative tables turn a `SurveyData` into publication-ready output. Each table reads variable labels, value labels, and measurement scales from the metadata you defined in your questionnaire, computes the appropriate statistics, and exports to DataFrame, Markdown, HTML, or Excel. There are three table types

— `FreqTable`, `CrossTable`, and `GroupMeanTable` — and all three are reachable through the fluent `data.report` accessor, so you rarely need to import the classes directly:

```
from siamang.reporting import FreqTable, CrossTable, GroupMeanTable  # optional; accessor preferred
```

All examples in this chapter use the simulated "Work Study" survey from Chapter 15, with variables `age`, `it_role`, `remote_freq`, and `autonomy` and `n = 200` simulated respondents:

```
data = survey.simulate(n=200, seed=123)
```

The common interface. All three table types subclass `SurveyTable` and share the same four output methods. The table is built lazily — nothing is computed until you ask for output.

TABLE 16.1

Method	Returns	Notes
`to_frame()`	`pd.DataFrame`	Raw result table, for further work in pandas.
`to_markdown()`	`str`	GitHub-flavored pipe table; appends a statistics footer.
`to_html()`	`str`	`<table class="dataframe siamang-table">`; renders inline in Jupyter.
`export_xlsx(path)`	`Path`	Writes an `.xlsx` file with a sheet named `"Table"`.

The statistics footer — Chi-square and Cramér's V for a cross-tabulation, the chosen mean-comparison test for a grouped-means table — is rendered automatically beneath the `to_markdown()` and `to_html()` output. You never format it yourself.

> **Note:** export_xlsx does not create directories — the destination directory must already exist, and both documentation sources agree on this. The safe habit is to create the output directory yourself before exporting.

16.1.1 FreqTable, CrossTable, and GroupMeanTable via data.report

The `data.report` accessor exposes one factory method per table type:

```
def freq(column, *, exclude_missing=True, sort="value") -> FreqTable
    def crosstab(row, col, *, pct="none", test=True) -> CrossTable
    def means(column, *, by, test=True) -> GroupMeanTable
```

FreqTable: univariate frequencies. A frequency table shows the distribution of a single categorical variable with absolute counts, percentages, cumulative percentages, and a `Total` row. Value labels are resolved automatically from the variable metadata. Its parameters:

TABLE 16.2

Parameter	Default	Meaning
`column`	—	Variable to tabulate.
`exclude_missing`	`True`	Drop `NaN` from the base.
`sort`	`"value"`	Row order: `"value"` (by code), `"freq"` (count descending), or `"label"` (alphabetical by label).

```
print(data.report.freq("it_role").to_markdown())

| Value | Label | N | % | Cumulative % |
    |---|---|---|---|---|
    | 1 | Engineer | 58 | 29.0 | 29.0 |
    | 2 | Data Scientist | 47 | 23.5 | 52.5 |
    | 3 | DevOps | 43 | 21.5 | 74.0 |
    | 4 | PM | 52 | 26.0 | 100.0 |
    |  | Total | 200 | 100.0 | 100.0 |

    Variable = IT Role; N valid = 200
```

Notice the footer: the variable label and the valid base size are carried through from the metadata without any work on your part. To sort by descending frequency instead — the usual choice for a “top mentions” table — pass `sort="freq"`:

```
data.report.freq("it_role", sort="freq").to_frame()
```

> **Warning:** sort accepts only "value", "freq", and "label". Any other value is silently ignored and the table keeps code order — a typo will not raise an error.

CrossTable: bivariate cross-tabulation. A cross-tabulation (contingency table) crosses a row variable with a column variable, shows row and column totals, and — by default — appends a Chi-square test of independence in the footer. Its parameters:

TABLE 16.3

Parameter	Default	Meaning
`row`	—	Row variable (usually the independent variable).
`col`	—	Column variable (usually the dependent variable).
`pct`	`"none"`	Percentage direction: `"none"` (counts), `"row"`, `"col"`, or `"total"`. The `Total` row and column always show raw counts.
`test`	`True`	Run the Chi-square test and append the footer. Requires scipy; without it the footer reports that scipy is missing.

```
print(data.report.crosstab("it_role", "remote_freq", pct="row").to_markdown())
```

```
| IT Role | Never | Occasionally | Hybrid | Mostly remote | Fully remote | Total |
    |---|---|---|---|---|---|---|
    | Engineer | 17.2 | 19.0 | 27.6 | 20.7 | 15.5 | 58 |
    | Data Scientist | 19.1 | 10.6 | 17.0 | 21.3 | 31.9 | 47 |
    | DevOps | 27.9 | 32.6 | 20.9 | 7.0 | 11.6 | 43 |
    | PM | 26.9 | 17.3 | 30.8 | 11.5 | 13.5 | 52 |
    | Total | 45.0 | 39.0 | 49.0 | 31.0 | 36.0 | 200 |

    χ² = 21.4850; df = 12; p = 0.0437; Cramér's V = 0.1890; N = 200
```

With `pct="row"` each row sums to 100 percent, so you read the table as "among Engineers, 15.5% work fully remote." The footer gives you everything a methods section needs: χ^2, degrees of freedom, the p-value, Cramér's V (an effect-size measure for the strength of association), and the base N.

> **Note:** The reference documentation describes the `CrossTable` test statistics as "Chi-square, Cramer's V, Phi", while the wiki's footer example shows χ^2, df, p, Cramér's V, and N only. The footer contents may therefore differ slightly between versions.

GroupMeanTable: grouped means with automatic test selection. A grouped-means table compares the mean of a continuous variable across the categories of a grouping variable and reports per-group `Mean`, `SD`, `Median`, and `N`. This is the formatted, significance-tested counterpart of `data.analysis.grouped_mean` from Chapter 15. Its parameters are `column` (the continuous dependent variable), `by` (the categorical grouping variable), and `test` (default `True`; requires scipy).

The important behavior is what happens with `test=True`: the table **selects the significance test automatically** from the dependent variable's measurement scale and the number of groups. This is the same logic Chapter 15 applied manually when you chose between `mannwhitney` and `kruskal` — here the table makes the choice for you:

TABLE 16.4

Dependent scale	2 groups	3+ groups
`ordinal` (or scale unknown)	Mann–Whitney U	Kruskal–Wallis H
`interval` / `ratio`	Independent t-test	One-way ANOVA

```
print(data.report.means("autonomy", by="remote_freq").to_markdown())

| Remote Frequency | Mean | SD | Median | N |
   |---|---|---|---|---|
   | Never | 3.222 | 1.38 | 4.0 | 45 |
   | Occasionally | 2.923 | 1.458 | 3.0 | 39 |
   | Hybrid | 2.898 | 1.447 | 3.0 | 49 |
   | Mostly remote | 3.0 | 1.653 | 2.0 | 31 |
   | Fully remote | 3.278 | 1.386 | 4.0 | 36 |

   Kruskal-Wallis H = 2.1330; p = 0.7113; N = 200; Variable = Autonomy
```

Because `autonomy` is ordinal and `remote_freq` has five categories, the table chose the Kruskal–Wallis H-test — exactly the test you ran by hand in Chapter 15. Had the dependent variable been `age` (ratio scale), the same call would have run a one-way ANOVA instead. This is the payoff of declaring measurement scales in your Variable definitions: correct test choice is automatic and consistent across the whole reporting layer.

> **Tip:** Use `data.analysis.grouped_mean` while you are exploring (it returns a plain DataFrame you can keep transforming), and switch to `data.report.means` once you want the formatted, significance-tested version for an audience.

Exporting. Every reporting table exports the same way:

```
table = data.report.crosstab("it_role", "remote_freq", pct="row")
    frame = table.to_frame()                    # pandas DataFrame
    md    = table.to_markdown()                 # str (with stats footer)
    html  = table.to_html()                     # str
    path  = table.export_xlsx("crosstab.xlsx")  # Path

    data.report.means("autonomy", by="remote_freq").export_xlsx("autonomy_means.xlsx")
```

Markdown output drops straight into a README, wiki page, or Quarto document; HTML renders inline in Jupyter; the Excel export suits collaborators who live in spreadsheets. In Chapter 18 you will see how the `Report` class embeds these tables directly into a narrative document — it calls `to_markdown()` for you.

> **Try it yourself:** Build `data.report.crosstab("it_role", "remote_freq", pct="col")` and compare it with the `pct="row"` version above. Which direction an-

swers the question “among fully remote workers, what share are Engineers?” Then rerun with `test=False` and confirm the footer disappears.

16.2 Banner Tables

A **banner table** (also called a *cross-break*) is the workhorse of survey reporting: it stacks one or more *row* variables against one or more *column* (banner) variables in a single wide structure, so every subgroup breakdown is readable at a glance. Where a `CrossTable` answers one question about one pair of variables, a banner table answers “how does every variable of interest break down across every subgroup?” — the format clients and stakeholders typically expect as a spreadsheet deliverable.

Siamang builds banner tables through the `data.tables` accessor (`SurveyTables`), which returns an export-ready `BannerTable`:

```
from siamang.data import SurveyTables, BannerTable

    data.tables   # -> SurveyTables(frame, variables, weight_column)
```

16.2.1 data.tables.banner(rows, columns, weight, labels)

The signature:

```
def banner(self, rows: list[str], columns: list[str],
               weight: str | None = None, labels: bool = True) -> BannerTable: ...
```

`banner` cross-tabulates **every** `rows` variable against **every** `columns` variable and concatenates the results into one long, tidy frame. For each row/column pair it computes the cell count `n` and the percentage **within each column category** — that is, column percentages. The parameters:

TABLE 16.5

Parameter	Default	Meaning
`rows`	—	Row variable names (the categories being profiled). Must be non-empty.
`columns`	—	Banner/column variable names (the cross-break groups). Must be non-empty.
`weight`	`None`	Optional weight column; cells become summed weights instead of raw counts. Defaults to the `SurveyData` weight set via `with_weight`. Raises `ValueError` if the column is not in the frame.
`labels`	`True`	Resolve variable and value labels into `row_label` / `column_label` columns.

The result is a frozen dataclass — `@dataclass(frozen=True, slots=True)` — wrapping a single DataFrame in its `frame` attribute. The frame has one row per row-variable × row-value × column-variable × column-value combination, with these columns:

```
row_variable | row_value | row_label | column_variable | column_value | column_label | n | percent
```

Here `percent` is a **proportion** within the column category — multiply by 100 for a percentage.

A first banner, one row variable against one column variable:

```
banner = data.tables.banner(rows=["it_role"], columns=["remote_freq"])
    print(banner.frame.head(6).to_string())
```

```
  row_variable  row_value       row_label column_variable  column_value   column_label     n   percent
    0      it_role          1        Engineer     remote_freq             1          Never  10.0
    0.222222
    1      it_role          1        Engineer     remote_freq             2   Occasionally  11.0
    0.282051
    2      it_role          1        Engineer     remote_freq             3         Hybrid  16.0
    0.326531
    3      it_role          1        Engineer     remote_freq             4  Mostly remote  12.0
    0.387097
    4      it_role          1        Engineer     remote_freq             5   Fully remote   9.0
    0.250000
    5      it_role          2  Data Scientist     remote_freq             1          Never   9.0
    0.200000
```

Read the first row as: among respondents whose `remote_freq` is *Never*, 22.2% are Engineers. Because the output is tidy (one observation per row, one variable per column), you can filter, pivot, and reshape it in pandas without any cleanup — `banner.frame` is an ordinary DataFrame.

Passing lists produces the full banner in one call. To profile both role and autonomy against the remote-frequency breakdown:

```
banner = data.tables.banner(rows=["it_role", "autonomy"], columns=["remote_freq"])
```

The frame simply grows: all `it_role` rows followed by all `autonomy` rows, each identified by `row_variable`.

Weighted banners. If you configured a default weight with `with_weight` (Chapter 15), the banner uses it automatically; you can also override the weight per call:

```
weighted = data.with_weight("design_weight")
    banner = weighted.tables.banner(rows=["it_role"], columns=["remote_freq"])
    # or override per call:
    banner = data.tables.banner(rows=["it_role"], columns=["remote_freq"],
                                weight="design_weight")
```

With a weight active, `n` holds summed weights rather than raw counts, and the column percentages are computed from those weighted sums.

The BannerTable container and export. `BannerTable` is immutable — the frozen dataclass guarantees the compiled frame cannot be mutated in place, so what you exported is always what you computed. It exposes two export methods, both of which return the written `Path`, write with `index=False`, and create parent directories as needed:

TABLE 16.6

Method	Returns	Notes
`export_csv(path, **kwargs)`	`Path`	Writes CSV; creates parent dirs. Extra keyword arguments pass through to the underlying writer.
`export_xlsx(path, **kwargs)`	`Path`	Writes Excel; creates parent dirs. Extra keyword arguments pass through to the underlying writer.

```
df = banner.frame                          # work with it in pandas
    banner.export_xlsx("out/banner.xlsx")   # publication export
    banner.export_csv("out/banner.csv")
```

Banner table or CrossTable? The two overlap deliberately; choose per audience:

- **Banner table** (`data.tables.banner`) — many subgroup breakdowns in one wide export, tidy long format, ideal for spreadsheets and deliverable appendices.
- **CrossTable** (`data.report.crosstab`) — a single, labeled two-way table with Chi-square, Cramér's V, and row/column percentages, ideal for inline Markdown/HTML and the Report-Document narratives of Chapter 18.

A practical workflow is to draft with a `CrossTable` — checking the Chi-square footer as you go — and then compile the final deliverable as one weighted banner table covering every variable of interest.

> **Tip:** Banner tables pair naturally with charts. A `BarChart` with `by=` (Chapter 17) visualizes exactly the grouped comparison a banner row quantifies, so you can export the banner as the data appendix and the chart as the headline figure.

> **Try it yourself:** Attach a synthetic weight with `data.with_frame(data.frame.assign(w=np.linspace(0.5, 1.5, len(data.frame)))).with_weight("w")` (as in Chapter 15), rebuild the `it_role` ×

`remote_freq` banner, and compare the weighted `percent` column against the unweighted one. Then export both versions to `out/` as CSV.

Chapter Summary

- Reporting tables (`FreqTable`, `CrossTable`, `GroupMeanTable`) turn a `SurveyData` into labeled, publication-ready output; all are built through the `data.report` accessor and subclass `SurveyTable`.
- Every reporting table shares the same interface — `to_frame()`, `to_markdown()`, `to_html()`, and `export_xlsx(path)` — and is built lazily on first use.
- `FreqTable` shows counts, percentages, and cumulative percentages with a `Total` row; `sort` accepts "value", "freq", or "label"; any other value is silently ignored and the table keeps code order.
- `CrossTable` produces a two-way contingency table with `pct` set to `"none"`, `"row"`, `"col"`, or `"total"`, plus a Chi-square and Cramér's V footer when `test=True`.
- `GroupMeanTable` compares means across groups and picks the significance test automatically: Mann–Whitney or Kruskal–Wallis for ordinal dependents, t-test or ANOVA for interval/ratio dependents.
- `data.tables.banner(rows, columns, weight, labels)` builds a banner (cross-break) table: every row variable against every column variable in one long, tidy frame with column percentages and an immutable `BannerTable` wrapper.
- Banner tables honor the default weight from `with_weight`, accept a per-call `weight=`, and export to CSV or XLSX with parent directories created.
- Use a banner table for wide spreadsheet deliverables and a `CrossTable` for single, significance-tested tables in narratives and reports.

Documentation References

- `wiki/Reporting-Tables.md` — FreqTable, CrossTable, GroupMeanTable, the `data.report` accessor, and the common `SurveyTable` export interface.
- `wiki/Banner-Tables.md` — banner table concept, `data.tables.banner` signature and result columns, the frozen `BannerTable` dataclass, CSV/XLSX export, and the banner-vs-CrossTable guidance.
- `docs/reference/reporting.md` — class inventory, base-class interfaces, typed accessor signatures, and the documented discrepancies on `sort` values, footer statistics, and directory creation.
- `wiki/Analysis.md` — `grouped_mean`, `kruskal`, and `mannwhitney` as the unformatted counterparts of the reporting tables (cross-referenced from Chapter 15).

Chapter 17. Charts and Visualization

Tables give your readers precision; charts give them pattern. In the previous chapter you built frequency tables, cross-tabulations, and banner tables with `data.report` and `data.tables`. This chapter introduces the visual counterpart: the four declarative chart types in `siamang.reporting` — `BarChart`, `BoxPlot`, `HeatMap`, and `ScatterPlot` — all reachable through the fluent `data.plot` accessor.

The design philosophy mirrors the reporting tables exactly. Each chart reads variable labels, value labels, and measurement scales from the metadata attached to your `SurveyData` and produces a publication-ready figure without manual axis labeling or legend construction. If you have already invested in well-labeled Variables (Part II), the charts inherit that work for free.

Learning Objectives

By the end of this chapter, you will be able to:

- Install the `charts` extra and recognize the errors raised when its dependencies are missing.
- Choose the right chart type — bar, boxplot, heat map, or scatter — for a given analytical question.
- Build each chart through the `data.plot` accessor with its specific parameters.
- Customize figures with `figsize`, `palette`, and `title`, and refine them further through the returned matplotlib Axes.
- Save figures to disk with `save()` and explain the documented discrepancy about directory creation.
- Describe how charts are embedded as `fig_<index>.png` assets in a Report document.

17.1 The Four Chart Types

The reporting charts mirror the reporting-tables API: each chart type is a class in `siamang.reporting`, and each is also reachable through a method on the `data.plot` accessor. As with the tables, the accessor is the everyday entry point; direct imports are there when you need them.

```
from siamang.reporting import BarChart, BoxPlot, HeatMap, ScatterPlot  # optional; accessor preferred
```

All examples continue with the simulated “Work Study” survey (`age`, `it_role`, `remote_freq`, `autonomy`; `n = 200`) used in Chapters 15 and 16.

Installing the dependency. Charts are an optional capability. They require the `charts` extra, which pulls in matplotlib (and seaborn, additionally needed by `HeatMap` and `ScatterPlot`):

```
pip install "siamang[charts]"
```

If matplotlib is missing, any chart raises a telling error: `ImportError:` matplotlib is required for chart generation. Install it with: pip install matplotlib. `HeatMap` specifically raises `ImportError:` seaborn is required for HeatMap when seaborn is unavailable. Install the extra once and both are satisfied.

The common interface. All four chart types subclass `SurveyChart` and share three presentation parameters and three output methods. The figure is built lazily — nothing is drawn until you call one of the output methods.

TABLE 17.1

Parameter	Default	Meaning
`figsize`	`(10, 6)`	Figure size `(width, height)` in inches.
`palette`	`"muted"`	Seaborn palette name (e.g. `"deep"`, `"pastel"`, `"colorblind"`).
`title`	`None`	Override the title; `None` derives one from the variable labels.

TABLE 17.2

Method	Returns	Notes
`plot()`	`matplotlib.axes.Axes`	Build and return the Axes for further tweaking.
`show()`	`None`	Display inline (Jupyter) or in a window.
`save(path, dpi=150)`	`Path`	Write to file with `bbox_inches="tight"`.

The `data.plot` accessor exposes one factory per chart type:

```
def bar(column, *, by=None, horizontal=False, show_values=True,
            figsize=(10, 6), palette="muted", title=None) -> BarChart
    def boxplot(column, *, by, show_points=False,
                figsize=(10, 6), palette="muted", title=None) -> BoxPlot
    def heatmap(columns, *, by=None, annot=True, cmap="YlOrRd",
                vmin=None, vmax=None, figsize=(10, 6), title=None) -> HeatMap
    def scatter(x, y, *, hue=None, trendline=True,
                figsize=(10, 6), palette="muted", title=None) -> ScatterPlot
```

17.1.1 BarChart, BoxPlot, HeatMap, ScatterPlot via data.plot

BarChart: frequencies or grouped means. A bar chart serves two purposes depending on whether you pass `by`. With only `column`, it draws the **frequency distribution** of a categorical variable — the visual twin of the `FreqTable` from Chapter 16. With `by` set, it switches to grouped-mean mode and plots the **mean of `column` within each category of `by`** — the visual twin of the `GroupMeanTable`.

```
BarChart(data, column="", by=None, horizontal=False, show_values=True,
             figsize=(10, 6), palette="muted", title=None)
```

TABLE 17.3

Parameter	Default	Meaning
`column`	—	Variable on the category axis (or whose mean is plotted).
`by`	`None`	Optional grouping variable; switches to grouped-mean mode.
`horizontal`	`False`	Draw horizontal bars.
`show_values`	`True`	Annotate each bar with its value.

```
data.plot.bar("it_role").show()
    data.plot.bar("autonomy", by="remote_freq",
    palette="pastel").save("autonomy_means.png")
```

Horizontal bars are worth remembering when value labels are long — "Data Scientist" reads far better along a vertical axis than rotated beneath one.

BoxPlot: distributions across groups. Where a bar chart of means compresses each group to a single number, a box plot shows the whole distribution — median, spread, and outliers — for a continuous variable across the categories of a grouping variable.

```
BoxPlot(data, column="", by="", show_points=False,
            figsize=(10, 6), palette="muted", title=None)
```

Here `column` is the continuous dependent variable (Y-axis) and `by` is the categorical grouping variable (X-axis); `show_points=True` overlays a jittered strip plot of the raw data points on top of the boxes, which helps you judge whether outliers are individual cases or a pattern.

```
data.plot.boxplot("autonomy", by="remote_freq", show_points=True).show()
```

HeatMap: group-mean matrices and correlations. The heat map has two distinct modes, selected by whether you pass `by`:

- With `by` set: a matrix of **group means** — each variable in `columns` averaged within the categories of `by`.
- With `by=None`: a **Spearman correlation matrix** of `columns`, drawn on a diverging `RdBu_r` scale centered at 0.

```
HeatMap(data, columns=[], by=None, annot=True, cmap="YlOrRd",
            vmin=None, vmax=None, figsize=(10, 6), title=None)
```

TABLE 17.4

Parameter	Default	Meaning
`columns`	`[]`	List of variables in the matrix.
`by`	`None`	Grouping variable for means; `None` for a correlation matrix.
`annot`	`True`	Write the numeric value in each cell.
`cmap`	`"YlOrRd"`	Colormap name.
`vmin` / `vmax`	`None`	Color-scale anchors.

```
# Mean autonomy and age across remote-frequency groups
    data.plot.heatmap(["autonomy", "age"], by="remote_freq", cmap="Blues").show()

    # Spearman correlation matrix of continuous measures
    data.plot.heatmap(["age", "autonomy"]).show()
```

Note: In correlation mode (by=None) the matrix is always drawn on the fixed diverging RdBu_r scale, centered at 0 over the range [-1, 1]; the cmap, vmin, and vmax properties are ignored. To recolor a correlation heatmap, restyle the matplotlib Axes returned by plot().

ScatterPlot: two continuous variables. A scatter plot charts two continuous variables against each other, with optional color grouping and a linear regression trendline.

```
ScatterPlot(data, x="", y="", hue=None, trendline=True,
                figsize=(10, 6), palette="muted", title=None)
```

TABLE 17.5

Parameter	Default	Meaning
`x`, `y`	—	Axis variables.
`hue`	`None`	Optional categorical variable to color points by.
`trendline`	`True`	Add a linear regression line; drawn **only when `hue` is `None`**.

```
data.plot.scatter("age", "autonomy", hue="remote_freq").show()
```

Note the interaction between `hue` and `trendline`: once points are colored by group, the single overall regression line is suppressed. If you want both the trend and the groups in an exploratory session, draw two figures — one with `trendline=True` and no `hue`, one with `hue` set.

> **Tip:** Choose the chart by the shape of your question. Distribution of one categorical variable: `bar`. Distribution of a continuous variable across groups: `boxplot`. Means of several variables across groups, or correlations among them: `heatmap`. Relationship between two continuous variables: `scatter`.
>
> **Try it yourself:** Reproduce the Chapter 16 analysis visually. Plot `data.plot.bar("autonomy", by="remote_freq")` next to the `GroupMeanTable` you built earlier and check that the bar heights match the table's `Mean` column. Then draw `data.plot.boxplot("autonomy", by="remote_freq", show_points=True)` and note what the box plot reveals that the table of means hides.

17.2 Customizing Charts

17.2.1 Parameters, styling, saving figures

Styling through parameters. The three shared parameters cover the most common adjustments. `figsize` controls physical size in inches — widen it for heat maps with many columns. `palette` accepts any seaborn palette name; `"colorblind"` is a considerate default for publications. `title` overrides the auto-generated title, which is otherwise derived from the variable labels you declared in your VariableMap:

```
data.plot.bar("autonomy", by="remote_freq",
                  figsize=(12, 5), palette="colorblind",
                  title="Mean autonomy by remote-work frequency").show()
```

Because labels flow from the metadata, the chart's category tick labels show "Never", "Occasionally", and so on — never raw codes like 1 and 2 — without any extra arguments.

Refining through the Axes. When parameters are not enough, `plot()` returns the matplotlib `Axes` object, giving you the full matplotlib API for final adjustments:

```
ax = data.plot.bar("it_role", horizontal=True).plot()   # get Axes to customize
    ax.set_xlabel("Respondents")
```

This is the escape hatch for journal-specific formatting: axis labels, tick formatting, gridlines, and anything else matplotlib supports. The chart builds the statistical content; you adjust the cosmetics.

Saving figures. `save(path, dpi=150)` writes the figure to disk and returns the `Path`, using `bbox_inches="tight"` so labels are not clipped. The default 150 dpi suits screen and most print use; raise it for typeset proceedings.

```
path = data.plot.boxplot("autonomy", by="remote_freq", show_points=True) \
                  .save("autonomy_by_remote.png")
```

> **Warning:** save() does not create directories — the destination directory must already exist, and both documentation sources agree on this. Create the output directory yourself before saving — for example with pathlib.Path("out").mkdir(exist_ok=True).

Charts inside a Report. You rarely save charts one by one for a deliverable. As you will see in Chapter 18, the `Report` class accepts any chart via `add(...)` and handles the files for you: when the document is rendered, each chart's `save()` is called automatically, the figure is written to a PNG named `fig_<index>.png` in the report's asset directory, and the image is linked into the Markdown or HTML output. The `caption` you pass to `add` becomes the image alt-text plus an italic caption beneath the figure:

```
report.add(data.plot.bar("autonomy", by="remote_freq"),
               caption="Figure 1. Mean autonomy by remote frequency")
```

So the workflow of this chapter — build the chart, tune it with parameters, preview with `show()` — transfers unchanged to report generation; only the terminal step moves from `save()` to `add()`.

> **Tip:** Keep `show()` in your exploratory notebook and remove it in the final script. A chart object passed to `Report.add` renders itself when the report is saved; calling `show()` there only pops up redundant windows.

> **Try it yourself:** Take the grouped-mean bar chart from the first exercise, restyle it with `palette="colorblind"` and a custom `title`, then save it twice: once via

`save("out/autonomy.png")` after creating `out/` yourself, and once by adding it to a quick `Report().add(...)` preview (you will formalize this in Chapter 18). Compare the two files.

Chapter Summary

- Siamang provides four declarative chart types — `BarChart`, `BoxPlot`, `HeatMap`, `ScatterPlot` — reachable through `data.plot` or imported from `siamang.reporting`; all subclass `SurveyChart` and build lazily.
- Charts require the `charts` extra (`pip install "siamang[charts]"`); missing matplotlib or seaborn raises a descriptive `ImportError`.
- Every chart shares `figsize` (default `(10, 6)`), `palette` (default `"muted"`), and `title` (auto-derived from variable labels), plus `plot()`, `show()`, and `save(path, dpi=150)`.
- `BarChart` shows frequencies of one variable, or grouped means when `by` is given; `BoxPlot` compares distributions across groups, optionally with raw points.
- `HeatMap` draws a group-mean matrix with by set, or a Spearman correlation matrix with by=None; in correlation mode cmap, vmin, and vmax are ignored (the scale is always RdBu_r, centered at 0 over [-1, 1]).
- `ScatterPlot` relates two continuous variables with optional `hue` coloring; the regression trend-line appears only when `hue` is `None`.
- `plot()` returns the matplotlib Axes for unlimited further customization; create output directories yourself before save(), which does not create them for you.
- In a `Report` document (Chapter 18), charts save themselves as `fig_<index>.png` assets — you add the chart object, never a file path.

Documentation References

- `wiki/Reporting-Charts.md` — the four chart types, the `charts` extra and its ImportErrors, the `SurveyChart` common interface, per-chart parameters and examples, and the `data.plot` accessor signatures.
- `docs/reference/reporting.md` — typed accessor signatures, the SurveyChart property list, and the additional HeatMap/BoxPlot/ScatterPlot examples.
- `wiki/Report-Document.md` — how `Report.add` embeds charts and materializes them as `fig_<index>.png` assets (previewed here, covered fully in Chapter 18).
- `wiki/Reporting-Tables.md` — the table counterparts (`FreqTable`, `GroupMeanTable`) whose statistics the bar chart visualizes (cross-referenced from Chapter 16).

Chapter 18. Generating Report Documents

In Chapters 16 and 17 you built the components of a survey deliverable: labeled frequency tables and cross-tabulations, significance-tested group-mean tables, banner tables, and publication-ready charts. Each of those lives in a separate Python object. Real projects, however, end with a single document — a narrative that walks the reader from sample description through findings, with tables and figures embedded at the right moments.

The `Report` class closes that gap. It composes narrative prose, reporting tables, and reporting charts into one document and exports it as Markdown or HTML. `Report` is deliberately a "thin orchestrator": it contains no statistics and no plotting logic of its own. Tables render themselves through `to_markdown()`, charts save themselves through `save()`, and `Report` stitches the pieces together in order and links the generated assets. Everything you learned in the previous two chapters transfers directly.

Learning Objectives

By the end of this chapter, you will be able to:

- Create a `Report` with a title and description and build it through fluent, chainable methods.
- Add narrative blocks — headings, prose, notes, key figures, and dividers — in document order.
- Insert reporting tables, charts, raw DataFrames, and external image files with captions.
- Export a report as Markdown or HTML, control chart asset handling, and recognize which formats are not supported.
- Merge several reports into one document with `Report.combine`, complete with a table of contents.
- Organize a multi-section publication workflow from per-topic reports to a combined final document.

18.1 The Report Builder

`Report` lives in `siamang.reporting` and is also importable from the top-level package:

```
from siamang.reporting import Report   # also: from siamang import Report
```

The constructor takes two optional arguments — a `title`, rendered as the document's top-level `# H1` heading, and a `description`, rendered in italics beneath it:

```
Report(title: str | None = None, description: str | None = None)
```

Every builder method appends a block to the document and returns `self`, so calls chain fluently. A terminal method — `save()`, `to_markdown()`, or `to_html()` — ends the chain by producing output.

18.1.1 Fluent builders, output formats, and chart assets

Narrative methods. Six methods add prose structure. Each appends one block and returns `self`:

TABLE 18.1

Method	Renders as
`heading(text, level=2)`	`## text` (the `level` argument controls the `#` count).
`markdown(md)`	Raw Markdown, verbatim.
`text(md)`	Alias for `markdown`.
`note(md)`	`> **Note:** md` — a blockquote callout.
`value(label, value)`	`**label:** value` — an inline key figure.
`divider()`	`---` — a horizontal rule.

`value` is small but useful: it surfaces headline numbers — sample size, fieldwork dates, response counts — right in the text flow where a reader expects them.

Two habits make narrative blocks work well. First, keep the default heading level: `heading(text)` renders at `## H2`, which sits correctly beneath the report's H1 title; only pass `level=3` when you genuinely need subsections within a section. Second, remember that `text` accepts any Markdown — emphasis, lists, and links all pass through verbatim — so you never need a separate templating step for prose. Reserve `note` for methodological caveats (base sizes, exclusions, weighting) that a reader must not miss, mirroring how the callouts in this textbook flag caveats to you.

Inserting components. The `add` method embeds the objects you built in Chapters 16 and 17:

```
def add(self, component: object, *, caption: str | None = None) -> Report: ...
```

`add` accepts three kinds of component: a `SurveyTable` (any of `FreqTable`, `CrossTable`, `GroupMeanTable`), a `SurveyChart` (any of `BarChart`, `BoxPlot`, `HeatMap`, `ScatterPlot`), or a raw `pandas.DataFrame`. Anything else raises `TypeError`. The `caption` renders in italics above tables, and as image alt-text plus an italic caption beneath charts.

For figures produced outside Siamang — a diagram from another tool, a scanned questionnaire page — `image` inserts an existing image file by path:

```
def image(self, path: str | Path, *, caption: str | None = None) -> Report: ...
```

A complete example. Continuing with the simulated "Work Study" survey from Chapters 15–17:

```
from siamang.reporting import Report

    report = (
        Report(title="Remote Work & Autonomy", description="Pilot wave, Q2 2026")
        .heading("Overview")
        .text("Perceived autonomy was measured on a 5-point scale.")
        .value("Respondents", len(data.frame))
        .add(data.report.means("autonomy", by="remote_freq"), caption="Table 1. Mean autonomy by remote
    frequency")
        .add(data.plot.bar("autonomy", by="remote_freq"), caption="Figure 1. Mean autonomy by remote
    frequency")
        .note("Non-consenting respondents were excluded from the base.")
        .divider()
    )

    # Write report.md plus fig_*.png into ./out/
    report.save("out/autonomy_report.md")
```

The generated Markdown begins:

```
# Remote Work & Autonomy

    *Pilot wave, Q2 2026*

    ## Overview

    Perceived autonomy was measured on a 5-point scale.

    **Respondents:** 200

    *Table 1. Mean autonomy by remote frequency*

    | Remote Frequency | Mean | SD | Median | N |
    |---|---|---|---|---|
    | Never | 3.222 | 1.38 | 4.0 | 45 |
    ...

    > **Note:** Non-consenting respondents were excluded from the base.
```

Notice how the `GroupMeanTable` appears as its fully formatted Markdown table, statistics footer included — `Report` simply asked the table to render itself.

This pass-through design has a practical consequence: everything you know about a component applies unchanged inside a report. The `CrossTable` still chooses percentages from its own `pct` argument, the `GroupMeanTable` still selects its significance test from the measurement scale, and the chart still derives its title from your variable labels. Configure the component the way you want it before you call `add`; the report never restyles what it embeds.

Exporting. Three terminal methods produce output:

```
def to_markdown(self, asset_dir: str | Path = ".", *, embed_images: bool = False) -> str: ...
    def to_html(self) -> str: ...
    def save(self, path: str | Path) -> Path: ...
```

- `to_markdown` renders the whole document to a string. Charts materialize as PNGs named fig_<index>.png, where <index> is the chart block's position in the whole document (narrative blocks count too) — the numbers are ordered but not consecutive. With the default `embed_images=False`, the PNGs are written into `asset_dir` and linked by relative filename; with `embed_images=True`, they are encoded inline as `data:` URIs and no files are written.
- `to_html` renders to HTML (via the `markdown` library with the `tables` extension), always embedding images inline. It requires the `markdown` package. Because images are embedded, an HTML report is a single portable string — convenient for pasting into a web page or an internal dashboard.
- `save` writes to disk and chooses the format from the file suffix: `.md` / `.markdown` (or no suffix) produces Markdown with the asset directory set to the file's parent; `.html` / `.htm` produces HTML. Parent directories are created automatically.

  Warning: Markdown and HTML are the only supported output formats. Saving with a `.pdf` suffix raises `NotImplementedError`, and any other suffix raises `ValueError`. No docx or pptx export is documented anywhere. If you need a PDF, export Markdown or HTML and convert it with an external tool.

  Tip: Prefer `embed_images=True` when you want a single self-contained file to email or paste into a wiki; prefer the default file-based assets when the report will live in a repository, so diffs on text stay small and images can be reviewed separately.

  Try it yourself: Rebuild the example report above, then render it twice: once with `report.save("out/report.md")` and once with `report.to_markdown(embed_images=True)`. Open both outputs and compare how Figure 1 is represented — a linked fig_4.png file versus an inline `data:` URI.

18.2 Combining Reports

A study with several workstreams rarely fits in one fluent chain. You might prepare one report for data cleaning, another for the main findings, and a third for subgroup analyses — possibly in different notebooks, possibly by different team members. `Report.combine` merges finished `Report` objects into a single document:

```
@classmethod
    def combine(cls, reports: list[Report], *, title: str, toc: bool = True) -> Report: ...
```

Each source report's `title` becomes a `## H2` section heading, with its `description` in italics below, and the source's blocks are appended in order. With `toc=True` (the default), a **Contents** list of anchor links is inserted at the top of the document. Section anchors are slugified from each title — lowercased, with non-alphanumeric characters replaced by `-` — so a report titled `"Data Cleaning"` is linked as `#data-cleaning`.

```
cleaning = Report(title="Data Cleaning", description="prep").text("49 cases dropped.")
    tables   = Report(title="Final Tables").add(data.report.freq("it_role"))

    full = Report.combine([cleaning, tables], title="Full Report")
    md = full.to_markdown()
    # "# Full Report"
    # "## Contents"  ->  "- [Data Cleaning](#data-cleaning)" / "- [Final Tables](#final-tables)"
    # "## Data Cleaning" ... "## Final Tables" ...
```

Because `combine` returns a `Report`, the merged document supports the same terminal methods — `save`, `to_markdown`, `to_html` — as any hand-built report.

18.2.1 A publication workflow

The pieces of this chapter assemble into a repeatable end-to-end pattern:

1) **Analyze and recode** with `data.analysis` and the metadata-aware recoders (Chapter 15) until the data frame is final.
2) **Build components**: reporting tables via `data.report` (Chapter 16) and charts via `data.plot` (Chapter 17), keeping each as a Python object — never as a screenshot or pasted file.
3) **Draft one `Report` per topic** — sample description, key findings, subgroup analyses — using `heading`, `text`, `value`, and `note` for the narrative and `add(..., caption=...)` for tables and figures. Number captions ("Table 1.", "Figure 1.") as you go.
4) **Combine** the topic reports with `Report.combine(..., title=..., toc=True)` so the final document gains a navigable contents list.
5) **Export** with `save("out/final_report.md")` (or `.html`). Charts are written automatically as `fig_<index>.png` next to the document; parent directories are created for you.
6) **Regenerate on demand.** Because the entire report is code, rerunning the script after a data update or a recode rebuilds every number, table, and figure identically — the core promise of research-as-code, and the pattern the capstone tutorial in Chapter 19 exercises in full.

Tip: Keep each topic report in its own function or notebook cell returning a `Report`. That way a colleague can review "Data Cleaning" in isolation, and `combine` stays a one-line final step.

Note: `Report` itself is documented only in `wiki/Report-Document.md`; it does not appear in the `docs/reference/reporting.md` class inventory alongside the tables and charts. Treat the wiki page as the authoritative source for its behavior.

Try it yourself: Split the autonomy report from Section 18.1 into two reports — one titled "Sample" (respondent count and the `it_role` frequency table), one titled "Findings" (the means table and bar chart). Combine them with a table of contents, save the result, and verify that the Contents links jump to the correct `## Sample` and `## Findings` sections.

Chapter Summary

- `Report` (importable from `siamang.reporting` or top-level `siamang`) composes narrative prose, reporting tables, and charts into one document; it is a thin orchestrator that lets components render themselves.
- The constructor takes an optional `title` (H1) and `description` (italics); every builder method returns `self` for fluent chaining, and `save`/`to_markdown`/`to_html` are the terminal steps.
- Narrative blocks: `heading`, `markdown`/`text`, `note` (blockquote callout), `value` (inline key figure), and `divider`.
- `add` inserts any `SurveyTable`, any `SurveyChart`, or a raw DataFrame — other types raise `TypeError` — with `caption` styled per component type; `image` inserts an external image file.
- Output is Markdown or HTML only: `.pdf` raises `NotImplementedError`, unknown suffixes raise `ValueError`, and no docx/pptx support is documented.
- Charts materialize as `fig_<index>.png` assets in `asset_dir`, or inline as `data:` URIs with `embed_images=True`; HTML always embeds images and requires the `markdown` package.
- `Report.combine(reports, title=..., toc=True)` merges reports under H2 section headings with slugified anchor links in a Contents list.
- Because a report is pure code, the publication workflow — analyze, build components, draft topic reports, combine, export — regenerates the entire document identically whenever the data changes.

Documentation References

- `wiki/Report-Document.md` — the `Report` class: constructor, narrative methods, `add` and `image`, the exporting API and suffix rules, chart asset naming, the full fluent example, and `Report.combine` with its table of contents.
- `wiki/Reporting-Tables.md` — the `SurveyTable` types embedded via `add` and their `to_markdown()` rendering (Chapter 16).
- `wiki/Reporting-Charts.md` — the `SurveyChart` types embedded via `add` and their self-`save()` behavior (Chapter 17).
- `docs/reference/reporting.md` — the reporting subpackage overview; noted here because it documents the tables and charts but not `Report` itself.

Chapter 19. End-to-End Tutorial: The Full Pipeline

You have now seen every piece of the Siamang engine in isolation: Variables and VariableMaps, questions and pages, expressions and routing, deployment, simulation, and the data-analysis and reporting stack. This chapter puts the pieces together. You will rebuild, step by step, the complete worked study that ships with Siamang as the full-pipeline tutorial (`wiki/Tutorial-Full-Pipeline.md`) and as the runnable notebook `examples/full_pipeline/full_pipeline_demo.ipynb` (34 cells, with all outputs saved inline). By the end you will have designed a questionnaire, checked it, exercised it with synthetic respondents, produced labeled tables and charts, and exported the results — one unbroken research-as-code workflow.

Learning Objectives

By the end of this chapter, you will be able to:

- Design a complete 12-variable, 5-page questionnaire with consent gating and `show_if` routing.
- Validate and preview the questionnaire before fielding it.
- Generate 300 synthetic respondents with `simulate()` and verify that skip logic produced the expected missingness pattern.
- Produce labeled frequency tables, cross-tabulations with Chi-square tests, and grouped means with automatically selected statistical tests.
- Render bar charts, boxplots, heatmaps, and scatter plots, and export tables to Markdown, HTML, and Excel.
- Apply selected cookbook recipes (quotas, weighting, index construction, R export) to extend the basic pipeline.

19.1 The Study

The tutorial's scenario is a sociological study of remote-working IT professionals: how do digital monitoring and algorithmic management affect job satisfaction and perceived autonomy? Two hypotheses structure the design. First, the *autonomy–surveillance paradox*: remote work raises subjective autonomy but invites invasive tracking (keystroke logging, webcam monitoring) that depresses satisfaction. Second, *professional stratification*: different IT roles experience different remote flexibility and monitoring pressure. The survey therefore has a consent gate, a demographics section, a remote-work measure, a surveillance-acceptance matrix, and two outcome scales.

19.1.1 Building the 12-variable, 5-page questionnaire with show_if routing

Everything begins with the codebook. You define twelve Variables — deliberately mixing scales (nominal for gender and role, ordinal for remote frequency, the surveillance items, and autonomy, interval for satisfaction, ratio for age and experience) so the reporting layer can pick statistical tests automatically later.

```
import siamang
    from siamang import (
        AND, LikertScale, Matrix, NumericInput, OpenText,
        Page, Questionnaire, SingleChoice, Variable, VariableMap,
    )

    # Consent
    consent = Variable("consent", scale="nominal", label="Informed Consent",
                       labels={1: "Yes", 0: "No"})

    # Demographics
    age = Variable("age", scale="ratio", label="Age (years)", dtype="int", valid_range=(18, 75))
    gender = Variable("gender", scale="nominal", label="Gender",
                      labels={1: "Male", 2: "Female", 3: "Non-binary"})
    it_role = Variable("it_role", scale="nominal", label="IT Role",
                       labels={1: "Software Engineer", 2: "Data Scientist", 3: "DevOps / SRE",
                               4: "Product Manager", 5: "QA / Tester"})
    experience = Variable("experience", scale="ratio", label="Years of Experience",
                          dtype="int", valid_range=(0, 50))

    # Remote work
    remote_freq = Variable("remote_freq", scale="ordinal", label="Remote Work Frequency",
                           labels={1: "Never (on-site)", 2: "1-2 days/week", 3: "Hybrid (3 days)",
                                   4: "Mostly remote (4 days)", 5: "Fully remote"})

    # Surveillance acceptance (three ordinal items, shown together as a matrix)
    _agree = {1: "Strongly disagree", 2: "Disagree", 3: "Neutral", 4: "Agree", 5: "Strongly agree"}
    surv_keystroke = Variable("surv_keystroke", scale="ordinal",
                              label="Keystroke logging acceptance", labels=_agree)
    surv_camera = Variable("surv_camera", scale="ordinal",
                           label="Webcam monitoring acceptance", labels=_agree)
    surv_git = Variable("surv_git", scale="ordinal",
                        label="Git commit metrics acceptance", labels=_agree)

    # Outcomes
    satisfaction = Variable("satisfaction", scale="interval", label="Job Satisfaction",
                            labels={1: "Very low", 2: "Low", 3: "Neutral", 4: "High", 5: "Very high"})
    autonomy = Variable("autonomy", scale="ordinal", label="Perceived Autonomy",
                        labels={1: "Very low", 2: "Low", 3: "Moderate", 4: "High", 5: "Very high"})
    story = Variable("story", scale="nominal", label="Open-ended comment")
```

Collect all twelve in a VariableMap — the machine-readable codebook that will travel with the data through simulation, deployment, and export:

```
variables = VariableMap()
    variables.add_many([
        consent, age, gender, it_role, experience, remote_freq,
        surv_keystroke, surv_camera, surv_git, satisfaction, autonomy, story,
    ])
    print(f"Defined {len(variables)} variables")   # Defined 12 variables
```

Next, bind the variables to questions. Note how each question type carries presentation hints (`display="buttons"`, `display="radio"`, `display="dropdown"`) without changing the underlying variable.

```
q_consent = SingleChoice("Do you consent to participate in this study?",
                         var=consent, required=True, display="buttons")
    q_age = NumericInput("What is your age?", var=age)
    q_gender = SingleChoice("What is your gender?", var=gender, display="radio")
    q_role = SingleChoice("What is your primary IT role?", var=it_role, display="dropdown")
    q_experience = NumericInput("Years of professional experience?", var=experience)
    q_remote = SingleChoice("How often do you work remotely?", var=remote_freq, display="radio")

    q_surveillance = Matrix(
        "To what extent do you accept the following monitoring practices?",
        var=[surv_keystroke, surv_camera, surv_git],
        subquestions=["Keystroke logging", "Webcam monitoring", "Git commit metrics"],
        column_labels=["Strongly disagree", "Disagree", "Neutral", "Agree", "Strongly agree"],
    )

    q_satisfaction = LikertScale("How satisfied are you with your current job?",
                                 var=satisfaction, points=5,
                                 left_label="Very dissatisfied", right_label="Very satisfied")
    q_autonomy = LikertScale("How much autonomy do you have in your daily work?",
                         var=autonomy, points=5, left_label="Very low", right_label="Very
    high")
    q_story = OpenText("Any additional comments about your work experience?",
                       var=story, multiline=True, max_chars=500)
```

Now assemble the five pages with routing. The consent page is always shown; every later page is hidden unless the respondent consented (`consent.eq(1)`), and the surveillance matrix additionally requires at least occasional remote work (`remote_freq.ge(2)`), composed with `AND`:

```
page_consent = Page(name="consent", title="Informed Consent", items=[q_consent])
    page_demo = Page(name="demographics", title="Demographics",
                     items=[q_age, q_gender, q_role, q_experience], show_if=consent.eq(1))
    page_remote = Page(name="remote", title="Remote Work",
                       items=[q_remote], show_if=consent.eq(1))
    page_surveillance = Page(name="surveillance", title="Workplace Monitoring",
                             items=[q_surveillance], show_if=AND(consent.eq(1),
    remote_freq.ge(2)))
    page_outcomes = Page(name="outcomes", title="Work Outcomes",
                         items=[q_satisfaction, q_autonomy, q_story], show_if=consent.eq(1))

    survey = Questionnaire(
        title="Remote Work, Autonomy & Digital Surveillance",
        pages=[page_consent, page_demo, page_remote, page_surveillance, page_outcomes],
        variables=variables,
    )
    print(f"Pages: {len(survey.pages)}")   # Pages: 5
```

The example directory ships `survey_preview.html`, a standalone SurveyJS render of exactly this questionnaire — conditional logic, validation, and progress bar included — that you can open in any browser with no server running.

> **Tip:** Save the questionnaire as a `.py` file with the module-level name `survey` (for example `my_survey.py`). The validate, preview, and deploy subcommands load that attribute by default, so the same file serves validation, preview, and deployment.

19.2 Fielding and Simulating

Before spending a single real respondent, you run two checks: structural validation and an interactive preview.

19.2.1 Validating, previewing, simulating 300 respondents

Validation runs `survey.validate()` followed by `survey.lint()` and prints any warnings as `[severity] [code] message (location)`:

```
siamang validate my_survey.py
    # OK — no warnings.

    siamang validate my_survey.py --strict   # expected to fail here (see the Note below)

Note: The --strict run above is expected to fail on this tutorial questionnaire:
    satisfaction is deliberately declared interval on a LikertScale (to enable ANOVA in
    Section 19.3), so strict lint reports INCOMPATIBLE_QUESTION_SCALE and the command exits
    with code 2. Plain siamang validate passes with no warnings and is the gate this
    pipeline relies on.
```

The preview command spins up the local FastAPI server with the React frontend and the SQLite backend, so you can click through the survey exactly as a respondent would; responses land in the local database until you stop the server with Ctrl+C:

```
siamang preview my_survey.py --port 8000 --open
    # Preview ready at http://0.0.0.0:8000
    #   survey_id: 42a1c0e9d3f5
    #   dashboard: sqlite:///survey.db
    #   [react] sucrase + esbuild minify available — fast path
    # Press Ctrl+C to stop.
```

With the design verified, simulate 300 respondents. `simulate(n, seed)` returns a fully populated SurveyData with the questionnaire's VariableMap and the questionnaire itself attached, and it respects every visibility rule: a respondent with `consent=0` gets `NaN` for all later variables, and a respondent with `remote_freq=1` gets `NaN` for the three surveillance items. This reproduces the missingness pattern your real data will have.

```
data = survey.simulate(n=300, seed=42)
    print(data.frame.shape)              # (300, 12)
    data.frame.head(10)

    desc = data.describe_variables()     # one row per variable: name, label, scale, n, n_missing,
    n_unique
```

Because `seed=42` is fixed, your run reproduces the tutorial's numbers exactly.

> **Warning:** Simulation draws values at random, not from any modeled correlation structure. The documentation is explicit: use it to exercise plumbing, validate logic, and preview report layouts — not to study real associations. Any "finding" in simulated data is noise by construction.
>
> **Note:** The tutorial notebook simulates 300 respondents. No SQLite database is shipped in the example directory — survey.db is a local artifact created once you run a local preview or deployment, and it stores responses exactly as in production.

19.3 Analysis and Reporting

The simulated SurveyData plugs directly into the entire reporting stack, with labels applied automatically from the VariableMap.

19.3.1 Frequencies, crosstabs, group means, charts, exports

Frequency tables. `data.report.freq()` computes N, %, and cumulative %, adds a Total row, and returns a rich FreqTable object:

```
print(data.report.freq("it_role").to_markdown())
    # sort categories by descending count
    print(data.report.freq("remote_freq", sort="freq").to_markdown())
```

Cross-tabulation with Chi-square. Because both variables are categorical, a Chi-square test and Cramér's V run automatically alongside the table:

```
print(data.report.crosstab("it_role", "remote_freq").to_markdown())              # counts
    print(data.report.crosstab("it_role", "remote_freq", pct="row").to_markdown())   # row
    %
```

The percentage modes are `"none"` (counts), `"row"`, `"col"`, and `"total"`.

Grouped means with automatic test selection. Here the scale mix you declared in the codebook pays off — the reporting layer picks the appropriate test from the variable's scale and the number of groups:

TABLE 19.1

Outcome scale	Groups	Test selected
ordinal	3+	Kruskal–Wallis H
ordinal	2	Mann–Whitney U
interval/ratio	3+	one-way ANOVA (F)
interval/ratio	2	independent t-test

```
print(data.report.means("autonomy", by="remote_freq").to_markdown())       # Kruskal-
    Wallis
    print(data.report.means("satisfaction", by="remote_freq").to_markdown())   # ANOVA
```

Charts. Plotting requires the charts extra (`pip install "siamang[charts]"`). The declarative `data.plot.*` calls produce labeled figures; `.plot()` renders them in a notebook, `.save("name.png")` writes a file.

```
data.plot.bar("it_role").plot()                          # category counts
    data.plot.bar("autonomy", by="remote_freq").plot()       # grouped means

    data.plot.boxplot("satisfaction", by="remote_freq",
                      title="Job Satisfaction by Remote Work Frequency").plot()
    data.plot.boxplot("autonomy", by="gender", show_points=True,
                      title="Perceived Autonomy by Gender").plot()

    # surveillance acceptance heatmap; vmin/vmax pin the 1-5 colour scale
    data.plot.heatmap(["surv_keystroke", "surv_camera", "surv_git"],
                      by="remote_freq", vmin=1, vmax=5,
                      title="Surveillance Acceptance by Remote Work Frequency").plot()

    # Spearman correlation matrix across the ordinal/interval outcomes
    data.plot.heatmap(["surv_keystroke", "surv_camera", "surv_git", "satisfaction", "autonomy"],
                      title="Spearman Correlation Matrix").plot()

    data.plot.scatter("satisfaction", "autonomy", hue="remote_freq",
                      title="Autonomy vs Satisfaction by Remote Frequency").plot()
```

A heatmap with `by=` shows grouped means; without `by=` it shows a Spearman correlation matrix. The notebook only renders these figures inline; to keep them as files (for example fig_outcomes_by_remote.png and fig_surveillance_heatmap.png), call .save(...) instead of .plot().

Exports. Every rich table converts to the format your write-up needs, and multi-cell banner tables export straight to Excel:

```
xtab = data.report.crosstab("it_role", "remote_freq")
    xtab.to_frame()        # pandas DataFrame
    xtab.to_markdown()     # publication-ready Markdown
    xtab.to_html()         # HTML fragment

    banner = data.tables.banner(rows=["satisfaction", "autonomy"],
                                columns=["remote_freq", "gender"], labels=True)
    banner.export_xlsx("banner_satisfaction_by_remote.xlsx")
```

Compare what this replaces. The notebook contrasts the traditional workflow with the declarative one:

TABLE 19.2

Traditional	Siamang declarative
`pd.crosstab(...)` + manual labels + `scipy.stats.chi2_contingency(...)`	`data.report.crosstab("it_role", "remote_freq")`
`df.groupby(...).agg(...)` + rename + `kruskal(...)`	`data.report.means("autonomy", by="remote_freq")`
`sns.boxplot(...)` + manual xtick labels + title	`data.plot.boxplot("autonomy", by="remote_freq")`

The notebook closes with a summary cell worth quoting, because it is a checklist of everything you have done:

```
Pipeline complete!
      - 12 variables with full metadata
      - 300 respondents simulated
      - Conditional routing: consent → demographics → remote → surveillance → outcomes
      - Tables: FreqTable, CrossTable, GroupMeanTable
      - Charts: BarChart, BoxPlot, HeatMap, ScatterPlot
      - Output: .to_frame(), .to_markdown(), .to_html(), .export_xlsx()
```

Try it yourself: Rerun the pipeline with `survey.simulate(n=300, seed=7)`. Confirm that the tables still render and that the missingness pattern (non-consenters and on-site workers with `NaN` surveillance items) is unchanged, even though every frequency differs. Then change the surveillance page's condition to `AND(consent.eq(1), remote_freq.ge(3))` and predict what happens to the matrix items' `n` in `describe_variables()` before you check.

19.4 Cookbook Highlights

The cookbook (`docs/cookbook.md`, `wiki/Cookbook.md`) collects short recipes that slot into this pipeline. All of them assume `import siamang as sg`. Four are especially relevant to the study you just built.

19.4.1 Selected cookbook recipes

Quotas. To balance the sample while fielding, attach Quota cells to the deploy call. When a quota cell is full, the respondent is routed to the closed screen: the runtime checks the backend's quota endpoint (`/quota-check` on the local server, the `quota-check` function on Supabase) and also honours a `{"status": "quota_full"}` submit response:

```
quotas = [
        sg.Quota("gender", 1, limit=200),
        sg.Quota("gender", 2, limit=200),
    ]
    survey.deploy(backend="supabase", frontend="vercel", quota=quotas)
```

Tightening a cell over time is cheap: updating a limit is just a code change followed by another deploy. Note that each deploy provisions a new survey instance with fresh quota counters.

Composite measures. The three surveillance items invite a summary scale. Check internal consistency with Cronbach's alpha, then build an index registered with an interval scale:

```
data.scale_alpha(["surv_keystroke", "surv_camera", "surv_git"])
    data = data.create_index(
        "surv_index",
        items=["surv_keystroke", "surv_camera", "surv_git"],
        method="mean",
        label="Surveillance acceptance",
    )
```

Weighting. Real fieldwork often needs post-stratification weights. `with_weight(col)` returns a view whose `analysis` accessor honours the weight when you pass `weighted=True` per call; `effective_sample_size()` reports Kish's ESS, which falls below N as weights become more uneven:

```
import numpy as np

    frame = data.frame.assign(
        weight=np.random.default_rng(0).uniform(0.5, 1.5, len(data.frame))
    )
    weighted = data.with_frame(frame).with_weight("weight")

    weighted.analysis.mean("satisfaction", weighted=True)
    weighted.analysis.proportion_ci("satisfaction", value=5, confidence=0.95,
    weighted=True)
    weighted.analysis.effective_sample_size()   # ESS <= N
```

Note: The declarative data.report.* tables are always unweighted — passing weighted=True raises TypeError; weights are honoured only by the data.analysis.* methods, as both cookbooks document. Also note that simulate() does not generate a weight column; the example above adds one manually.

Export for other tools. To hand the dataset to colleagues working in R, SPSS, or Stata, use the high-level export helpers:

```
data.export("spss",  path="out.sav")
    data.export("stata", path="out.dta")
    data.export("r",     path="out_R/")
    data.export_dictionary("dict.json")
```

Note: data.export supports the formats "csv", "xlsx" (alias "excel"), "spss" (alias "sav"), "stata" (alias "dta"), and "r". The R export writes import_survey.csv, import_survey_dictionary.json, and import_survey.R; sourcing import_survey.R in R builds a data frame named survey_data.

Try it yourself: Extend the tutorial survey with one cookbook technique: add `sg.Script.randomize_options("it_role")` to the Questionnaire's `scripts` list so the IT-role options appear in random order per respondent, then re-validate with siamang validate and confirm the script did not disturb the routing logic (plain validation passes; --strict still fails with INCOMPATIBLE_QUESTION_SCALE, as noted in Section 19.2).

This closes the source-available engine part of the textbook. Everything you built here — the questionnaire file, the validation and preview loop, the simulation, the reports — is pure Python under your control. In Part V you will see the same survey from a different angle: Siamang Cloud lets you manage, field, and monitor it from a hosted platform, without giving up the code-first definition you have just written.

Chapter Summary

- The full pipeline is: define Variables and a VariableMap, bind them to questions, assemble Pages with `show_if` routing, validate, preview, field (or simulate), analyze, and export.
- The tutorial study uses 12 variables across 5 pages, with a consent gate (`consent.eq(1)`) and a nested condition (`AND(consent.eq(1), remote_freq.ge(2))`) on the surveillance matrix.
- `siamang validate` (optionally `--strict`) checks structure and lint; `siamang preview` serves the survey locally with a SQLite backend for hands-on testing.
- `survey.simulate(n=300, seed=42)` produces a SurveyData that honours all visibility rules, reproducing the missingness pattern of real data — but its values are random and must not be interpreted substantively.
- `data.report.*` yields labeled FreqTable, CrossTable, and GroupMeanTable objects with automatic Chi-square/Cramér's V and scale-appropriate group tests; `data.plot.*` yields labeled BarChart, BoxPlot, HeatMap, and ScatterPlot figures.

- Tables convert via `.to_frame()`, `.to_markdown()`, `.to_html()`, and banner tables export to Excel with `.export_xlsx()`.
- Cookbook recipes extend the pipeline with quotas, composite indices, weighting (via the `analysis` accessor), and multi-format export.

Documentation References

- `wiki/Tutorial-Full-Pipeline.md` — the end-to-end tutorial reproduced in this chapter
- `examples/full_pipeline/` (README.md, full_pipeline_demo.ipynb, survey_preview.html) — the shipped runnable example
- `docs/cookbook.md`, `wiki/Cookbook.md` — cookbook recipes (quotas, weighting, indices, exports)
- `wiki/Simulation.md` — `simulate()` semantics and the random-data caveat
- `wiki/CLI-Reference.md`, `docs/reference/cli.md` — `siamang validate` and `siamang preview`
- `wiki/Working-with-Data.md`, `docs/reference/data.md` — SurveyData, report tables, and banner exports

Part V — Siamang Cloud

Chapter 20. Siamang Cloud: Overview, Accounts, and First Project

In Chapter 19 you ran a complete study on your own machine — questionnaire as Python, validation, simulation, tables, reports — but everything lived on your laptop, and publishing meant wiring up backends and frontends yourself. This chapter opens Part V and introduces **Siamang Cloud**, the hosted companion platform that keeps the same code-first workflow while taking over publishing, response storage, validation, analysis runs, and teamwork.

Learning Objectives

By the end of this chapter, you will be able to:

- Explain what Siamang Cloud is, how it relates to the `siamang` Python library, and what it adds for users.
- Create an account, configure your profile, and apply the documented security practices (passwords, API keys, SSH keys).
- Create an organization and a first project from the Example, Template, or Empty starting points.
- Follow the Quick Start loop: review the survey in the repository, deploy it, fill it in, watch responses arrive, and run an analysis to a report.
- Describe the open-beta rules: the 30-day Pro trial and what happens when it ends.

20.1 What Siamang Cloud Is

Siamang Cloud is, in the documentation's words, "the hosted home for your surveys." You author a survey as code with the `siamang` Python library — exactly as you did throughout Parts I–IV — keep it in a **project** on the platform, and from the browser you deploy it to a public link, collect responses, run analysis, read dashboards and reports, and manage your team and plan, all in one place.

20.1.1 The hosted platform around the same engine

The core idea is that your survey is *code in a hosted project*. Each project is backed by **Git**, the version-control system used for software, so the questionnaire, its logic, and analysis scripts are versioned like software: every change is a reviewable commit. Around your survey code, the platform publishes the survey to a live public URL (respondents need no account), collects responses into managed storage you can browse and export, validates your code automatically on every commit with a pass/fail status — so a bro-

ken survey never goes live by accident — runs analysis scripts with a history of every run and its output, shows dashboards and generates reports in Markdown and HTML, and manages your team and plan.

You never wire up servers, databases, or hosting. This is the key contrast with Part III: there you deployed with `survey.deploy(backend=..., frontend=...)` and managed credentials yourself; here the same survey lives in a project and goes online with one click. Everything can still be done locally with the library alone — the Cloud adds a public survey host, managed storage, automatic validation on every push, hosted analysis runs with history, and team collaboration, as the Overview page's comparison summarizes:

TABLE 20.1

What you want to do	In the library	In the Cloud
Write the survey	A Python module	The same module, in your project's repo
Check it is valid	`siamang validate` on your machine	Automatic on every commit, with a status badge
Put it online	Deploy to your own backend	One click → a hosted survey URL
Store responses	Local file / your own database	Managed storage you browse and export
Analyze data	Run scripts locally	Hosted runs, schedules, run history, reports
Work as a team	Files you pass around	Organizations, roles, shared projects

> **Note:** Siamang Cloud is an open-beta service operated by Siamang Labs LLC, the team behind the source-available `siamang` engine. Per the Terms of Use, it is free of charge during the beta, provided "as is" with no uptime or support guarantee, and features may change before the official release — keep your own copies of important data.

20.2 Accounts and Security

Everything in Siamang Cloud starts from an account, and your personal account settings follow you across every organization you belong to, so you configure them once.

20.2.1 Signing up, account settings, security features as documented for users

You sign in with **GitHub** or with **email and password**, landing in your **workspace** with your organization and account menu in the top bar. Open your personal settings from the account menu → **Profile**, which has six tabs:

TABLE 20.2

Profile tab	What you do there
Account	Set your display name — the name teammates see next to your activity.
Security	Change your password.
Appearance	Switch between the light and dark theme (the only place to do so).
API keys	Create and revoke personal access tokens for scripts and CLIs.
SSH keys	Register a public key to clone and push repositories over SSH.
Support	Contact the team, open documentation, report bugs, and reach the Terms of Use and Privacy Policy.

Changing and resetting passwords. On the Security tab you enter the new password twice (some configurations also ask for your current password) and click Save — you stay signed in. If you forget your password while logged out: enter your email on the sign-in screen and click **Forgot password?**, click **Send reset link**, follow the link from your inbox back to Siamang Cloud, and enter a new password twice. Password sign-in and reset apply to email accounts; if you sign in with GitHub, you manage your password at GitHub.

API keys. An API key is a personal token (format `sck_…`) that lets scripts, CLIs, or your own programs talk to Siamang Cloud as you, without a password. Any signed-in user can create one: open **Profile → API keys**, click **New key**, give it a name, and create it — then copy the token immediately, because it is shown only once; afterwards just its name and a hash remain. You send the token in a request header as `Authorization: Bearer sck_…`, and you can revoke a key at any time with **Revoke**, which stops it working immediately. The documentation's advice is direct: treat a key like a password — if one leaks, revoke it and create a new one.

SSH keys. If you prefer editing on your own machine instead of the browser, register an SSH public key so you can clone and push a project repository over SSH; the private half of the key pair never leaves your computer (`ssh-keygen` creates the pair). Copy your public key (for example the contents of `~/.ssh/id_ed25519.pub`), open **Profile → SSH keys → Add SSH key**, paste the key with a label, and save. For command-line clone and push, the Repository → Remotes dialog in your project shows the HTTPS clone URL, a username, and an access token; it does not currently display the SSH address.

Privacy and your responsibilities. Two points from the Privacy Policy matter from day one. First, there are two kinds of data: your account and usage data (which the policy covers) and survey respondent data — for the latter, *you* are the data controller and Siamang Cloud is only your processor, so you are responsible for lawful basis, notice, and consent. Second, the console uses no advertising trackers

and does not sell your data; cookies are limited to sign-in/session cookies and preference storage such as the theme. Account holders must be at least 16 years old.

> **Tip:** You own what you create — questionnaires, scripts, configuration, and the responses you collect — and you can export or delete it at any time. Response deletion is built into the app, which helps you honor respondent erasure (GDPR) requests as the controller.

20.3 Quick Start and Your First Project

The Quick Start walks you from sign-in to a live survey plus a first report, entirely in the web app, using the built-in example study. The only prerequisites are an account and an organization — a personal one is fine.

20.3.1 The documented quick-start flow; the example study "Digital Life & Wellbeing 2026"

1) **Sign in** with GitHub or email and password; you land in your workspace, with your organization and account menu in the top bar.
2) **Create an organization** via **Create organization** and a name. A personal organization is fine for solo work; it can become a team later.
3) **Create a project.** Inside the organization, open **Projects → New project** and choose a starting point:

TABLE 20.3

Starting point	What you get
Example	The complete "Digital Life & Wellbeing 2026" study (see below). Recommended for a first run.
Template	A minimal survey plus two starter analysis scripts, no sample data.
Empty	A one-question starter survey to build on from scratch.

   Whichever you pick, the platform sets up a Git-backed repository and a database automatically. The project workspace's left sidebar lists Dashboard, Repository, Database, Deployments, Analysis, and more.
4) **Review the survey in the repository.** Open **Repository**, then `survey/questionnaire.py` in the in-browser editor; other key files are `siamang.yaml` (project configuration) and `scripts/` (analysis scripts). To edit, make a change, click **Save & commit** (or **Ctrl/Cmd+S**), write a message, and commit — validation runs automatically on every commit, showing a green badge when the file is valid or the error count when not.

5) **Deploy.** Open **Deployments → New deployment**, choose an environment — the example ships with **pilot** (response cap 50) and **main** (cap 1200) — and deploy. The platform compiles the latest commit and publishes it; the finished card shows a **public survey URL** that respondents open in any browser, no account required. The **Preview** button builds a staging version you can fill in yourself; a preview does not accept real responses.
6) **Fill the survey** as a respondent, submitting a few times to add real responses alongside the sample data.
7) **Watch responses appear.** In **Database**, select the `responses` table: your submissions appear as rows next to the seeded samples, and you can filter, sort, page, and **Export** to CSV, Excel, SPSS (`.sav`), Parquet, or SQLite. On the Dashboard, a stats row (status, branch, responses, commits, deployments) is always visible. The Data insights section — live charts such as a frequency chart and a two-way crosstab — is off by default and appears once the project opts in by declaring an insights: block in siamang.yaml; the example project ships one, commented out and ready to uncomment (a frequency chart for life_satisfaction and a life_satisfaction × age_group crosstab).
8) **Run an analysis and open the report.** Open **Analysis**; the **Scripts** list shows the analysis steps declared in `siamang.yaml`. Click **Run** on a single script or Run all to run every step in order (clean → tabulate) and combine the results into one report. Each run appears in Run history with a status of success or failed (running while in progress), shown as a card with the pipeline steps, output chips linking to what the run produced, and expandable logs; reports are Markdown with an HTML download, and also appear under **Files** and in the Repository.

The example study. "Digital Life & Wellbeing 2026" is a full questionnaire — consent, screening, quotas, every question type, skip logic, and a custom thank-you page — plus two analysis scripts (cleaning.py for data cleaning, and tables.py for frequencies, a chi-square crosstab, and a Markdown report) and roughly 300 sample responses, so the data screens are alive immediately. Its repository follows the standard project layout:

TABLE 20.4

Path	What it is
`siamang.yaml`	Project configuration: survey entry point, deployment environments, analysis scripts to run.
`survey/questionnaire.py`	The survey itself, written as Python.
`scripts/`	Analysis scripts (in the example: cleaning.py and tables.py).
`outputs/`	Where analysis runs write tables and files.
`reports/`	Generated reports (Markdown / HTML).
`README.md`	Project notes, shown on the Dashboard.

Try it yourself: Create a project from the Example starting point, deploy the **pilot** environment, and submit the survey three times. Then open the Database and confirm your submissions sit alongside the ~300 seeded rows, then enable Data insights (uncomment the insights: block in siamang.yaml and commit) and open the Dashboard to see the frequency chart update. Finally, click **Run all** on the Analysis screen and open the combined report.

20.3.2 Beta notes: 30-day Pro trial, downgrade to Free when the trial ends

Because Siamang Cloud is in open beta, the subscription rules are unusual and worth understanding before you invest in a workspace. The platform has four plans — **Free**, **Plus**, **Pro**, and **Corporate** — and a plan applies to a whole organization and every project inside it. During the beta, the documented rules are:

- Every new workspace starts on a **30-day Pro trial** — full access to every Pro feature, with a countdown on the Billing screen and in the top bar.
- Plan switching is turned off, and other plans show “Available at the official release”; note that paid beta offers (for example, a discounted year of Pro as a one-time payment) may still appear before the official release.
- **When the trial ends**, the workspace is downgraded to Free — your data is preserved and remains exportable, and work continues within the Free limits.
- Paid plans (with Stripe) arrive at the official release. Billing is per organization, and only the organization **owner** manages it.

Warning: Do not treat the trial’s full Pro access as the permanent state of your workspace. After 30 days the workspace is downgraded to the Free plan at the official release

— plan any long fieldwork with that horizon in mind, and remember you can export your data at any time via **Database → Export**.

You now have an account, an organization, and a first project with a live survey and a generated report. The next chapter tours the web app systematically — Dashboard, Repository and editing workflows, Database, Deployments, Analysis, Files, and project settings — and shows how organizations, teams, and roles govern who can do what.

Chapter Summary

- Siamang Cloud is the hosted home for surveys written with the `siamang` library: your survey is code in a Git-backed project, and the platform publishes it, stores responses, validates every commit, runs analysis, and manages teams and plans.
- The Cloud adds one-click hosting, managed storage, automatic validation badges, hosted analysis with run history, and collaboration on top of everything the library does locally.
- Accounts sign in via GitHub or email/password; personal Profile settings (display name, password, theme, API keys, SSH keys, support) follow you across organizations.
- API keys (`sck_…`) are shown once at creation and are revocable; SSH public keys enable clone/push from your own machine.
- For respondent data you are the controller and Siamang Cloud the processor; you can export or delete your data at any time.
- The Quick Start loop: create an organization → create a project (Example / Template / Empty) → commit in the Repository → deploy to an environment → fill the survey → watch Database and the Dashboard (stats row always; live Data insights charts after enabling the insights: block in siamang.yaml) → run analysis to a report.
- The example study “Digital Life & Wellbeing 2026” ships a full questionnaire, two analysis scripts, ~300 sample responses, and pilot (50) and main (1200) environments.
- During the open beta every workspace gets a 30-day Pro trial; nothing is charged while no payment provider is connected (paid beta offers, such as a one-time year of Pro, may still appear before the official release), and the workspace is downgraded to Free (data preserved and exportable, work continues within the Free limits) after the trial.

Documentation References

- `wiki/Cloud-Overview.md` — positioning and the library-vs-Cloud comparison
- `wiki/Cloud-Quick-Start.md` — the end-to-end Quick Start loop
- `wiki/Cloud-Your-First-Project.md` — starting points and the standard repository layout
- `wiki/Cloud-Account-and-Security.md` — profile tabs, password reset, API keys, SSH keys
- `wiki/Cloud-Subscription-Tiers.md` — plans and the open-beta trial rules
- `docs/cloud/privacy-policy.md`, `docs/cloud/terms-of-use.md` — controller/processor split, ownership of content, beta terms

Chapter 21. Working in the Cloud Web App

Chapter 20 got you signed in, set up an account, and walked the Quick Start from a new project to a live survey with a first report. This chapter slows down and maps the web app properly. You will learn how the interface is organized, how to work with survey code in the Repository — editing in the browser or from your own machine, with every change versioned — and how organizations, roles, and permissions decide who can do what. Everything here is what the documentation exposes to you as a user; you never touch servers.

Learning Objectives

By the end of this chapter, you will be able to:

- Navigate the web app's top bar, sidebars, and organization- and project-level screens.
- Use each project screen — Dashboard, Repository, Database, Deployments, Analysis, Connectors, Files, Settings, and the Console — for its documented purpose.
- Edit survey files in the browser, commit changes, read validation results, browse history, and work with branches and pull requests.
- Clone and push a project from your own machine (HTTPS via Repository → Remotes; SSH with a public key registered under Profile → SSH keys), and mirror a repository to GitHub or GitLab.
- Distinguish personal and cooperative organizations, invite teammates, and explain what the owner, admin, and member roles can each do.

21.1 Web App Tour

The web app has two consistent landmarks: a **top bar** and a **left sidebar** whose contents change with context.

21.1.1 Navigation, dashboards, menus — the documented UI map

The top bar, visible after sign-in, shows the current organization (click its name to switch organizations and jump between their projects — everything on screen, including projects, team, and billing, re-scopes), your **Pro trial** countdown while it is active, a **Console** button when you are inside a project, and your **account** menu. The light/dark theme toggle lives in **Profile → Appearance**, not in the top bar.

The left sidebar changes with context. At the organization level it lists **Projects**, **Team**, and **Settings**. Inside a project it lists Dashboard · Repository · Database · Deployments · Analysis · Connectors · Files · Settings, with a status dot by the project name showing its latest validation state.

Organization screens. Three screens manage the workspace as a whole:

TABLE 21.1

Screen	What it shows
Organizations	Workspace overview: the organization, its type (personal or cooperative), your role, and the current plan. An owner can convert a personal workspace to cooperative here.
Projects	The project list and the **New project** button (Empty / Template / Example). Each row shows the project's status and response count; at your plan's project limit the button is disabled with an upgrade prompt.
Team	A read-only roster (email, role, join date). Member management happens in **Settings → Members**.

Project screens. Inside a project, eight screens cover the survey lifecycle:

TABLE 21.2

Screen	What you do there
Dashboard	The project's home: a stats row (status, branch, responses, commits, deployments) above Data insights — off by default; live charts over all collected responses (counts, partial rate, responses per day, frequency, crosstab) appear once an insights: block is declared in siamang.yaml — plus the README, recent commits, a language breakdown, and the latest deployment.
Repository	The survey as code: file tree, in-browser editor, commits, branches, pull requests, remotes, and rendered reports (covered in §21.2).
Database	Browse and export responses: pick a table, page and filter rows on the **Data** tab, inspect columns on the **Schema** tab, **Export** to CSV, Excel, SPSS, Parquet, or SQLite, and delete individual responses.
Deployments	Publish and watch fieldwork: **New deployment** to an environment, per-deployment status (Live / Building / Failed / Stopped), the public survey URL, a live monitor (responses against the cap, quota cells, the codebook), build **Logs**, and **Stop** / **Redeploy** actions.
Analysis	The analysis steps declared in `siamang.yaml`, each with **Run** and **View in Repo**; **Run all** runs every step in order and combines reports; Run history shows each run as a card with a status (success / failed / running), step progress, output chips, and expandable logs.
Connectors	Send project data to external stores — available from the Plus plan (first targets: Google Sheets, Excel 365, Supabase, Airtable, Dropbox); object storage and data warehouses (S3, GCS, Azure, BigQuery, Snowflake, your own Postgres, SFTP, REDCap, HTTP) require Pro; MCP is a Corporate feature.
Files	Everything generated or uploaded, in two groups: **Repository outputs** (reports and generated tables) and **Assets** (uploaded files and exports), with Upload / Download / Delete. Uploads are capped at 50 MB per file.
Settings	Project configuration tabs (see below).

Project **Settings** has six tabs: **General** (project name and default branch), **Runtime** (Python version and packages, read from `siamang.yaml` — edit the file and commit to change it), **Secrets** (encrypted, write-only values that connectors and Git mirrors reference by name), **Git** (repository settings such as protecting the `main` branch), **Activity** (this project's audit trail), and the **Danger Zone** (permanently delete the project and its data).

Finally, the **Console** is a command panel for the project, opened with the **Console** button in the top bar or the **backtick** (`) key. It is where you manage schedules: type `schedules` to list them, `schedule add --cron "<expr>" (--all | --script <name>)` to add one, and `schedule rm <id>` to remove one.

> **Tip:** Learn the two keyboard shortcuts early: **Ctrl/Cmd+S** saves and commits in the editor, and the **backtick (`)** opens the project Console from anywhere.

21.2 Repository and Editing

The **Repository** screen is where you read and change the survey code — in the browser or from your own machine — with validation on every commit.

21.2.1 Managing survey files, editing in the browser, versioning as exposed to users

The editor. A file tree sits on the left; clicking a file opens it in the in-browser code editor, with syntax highlighting for Python, YAML, and Markdown, line numbers, and a **modified** marker on the file until you commit.

Edit and commit (requires the **member** role or higher):

1) Open **Repository** and click a file — for example `survey/questionnaire.py`.
2) Make your changes.
3) Click **Save & commit** (or press **Ctrl/Cmd+S**).
4) Write a short commit message and confirm.

The change is committed and **validation runs automatically**: the file gets a green **valid** badge, or an error/warning count you click for details. A failing commit tells you exactly what to fix. The platform never saves without a commit, so your history stays clean and every change is reviewable.

History, branches, and pull requests. The **Commits** list shows branch history; clicking a commit reveals its **diff** and validation status. You can create or switch a branch in the Repository toolbar, commit changes there, open a **pull request**, review the diff, and **merge** into `main`. An owner or admin can **protect** the `main` branch (**Settings → Git**) so changes must pass validation before landing. Adding, renaming, moving, and deleting files — for example a new analysis script under `scripts/` via **New file** — are commits too.

Project secrets (member role or higher) hold credentials — API tokens, service-account keys — that connectors and Git mirrors need. (Analysis scripts run in a sandbox with no network access and no secrets.) Never store them in code:

1) Open **Settings → Secrets**.
2) Click **New secret**, enter a name and value, and save.
3) Reference the secret **by name** from `siamang.yaml` or a script.

The value is encrypted and **write-only** — afterwards only the name is visible.

Working from your own machine: Remotes. The **Remotes** button opens the **Repository connections** dialog, which does two things:

- **Clone and push locally.** Under Local clone, copy the HTTPS clone command with its username and access token. (The dialog shows HTTPS only; for SSH, add a public key under Profile → SSH keys first — the SSH address itself is not shown in the UI.) Clone, edit on your machine, and `git push`; pushed commits appear in the Repository and are validated exactly like browser commits.
- **Mirror to GitHub** (Plus plan and above) or GitLab (Pro plan and above). Under **Mirrors**, choose the provider, enter the remote path, and pick the project **secret** holding your access token. Add the mirror, then **Sync now** to push; you can pause, resume, or delete the mirror later. The platform keeps the external remote in sync with the project repository.

Reading reports. Markdown (`.md`) files — including generated reports — open with a rendered **Preview** and an **Edit** view, plus **MD** and **HTML** download buttons.

> **Try it yourself:** In a project created from the Example study, create a branch, change the questionnaire's title in `survey/questionnaire.py`, and commit with **Ctrl/Cmd+S**. Watch the validation badge, then open a pull request and read the diff before merging. Finally, open the **Commits** list and confirm every step — branch, edit, merge — is a reviewable commit.

21.3 Organizations and Teams

An **organization** is your workspace: it owns your projects, your team, and your plan. The subscription always belongs to the organization **owner**.

21.3.1 Organizations, member roles, permissions — what each role can do

Two types of organization:

TABLE 21.3

Type	What it is
Personal	A solo workspace with a single member — you, the owner.
Cooperative	A team workspace: you can invite people and give them roles.

You can start personal and convert later; projects and data stay exactly as they are. To convert (requires the **owner** role):

1) Open **Organization settings** (the **Settings** button, or **Manage** from the Organizations screen).
2) On the **General** tab, set the type to **cooperative** and give the workspace a team name.
3) Save.

Inviting a teammate requires a cooperative organization, the owner or admin role, and member room on your plan. The invite is sent by email; if the person has no account yet, they receive an invitation link and create the account while accepting (the account email must match the invited email):

1) Open **Organization settings → Members**.
2) Click **Invite member**.
3) Enter their email and pick a role — **admin** or **member**.
4) Send the invite.

The **Team** screen shows the full roster (email, role, join date) but is read-only; invites, role changes, and removals live in **Settings → Members** (a **Manage members** button on the Team screen takes you there).

Roles and permissions. Every member has exactly one of three roles, and the documentation draws a sharp line: your **role** decides what *you* can do; your **plan** decides what the *organization* can do — they are independent.

TABLE 21.4

Role	Can do
owner	Everything, including changing the plan and billing, organization settings, and SSO. There is exactly one owner.
admin	Manage the organization profile and members (invite, change roles, remove), create projects, set branch protection, and manage integrations.
member	Contribute to projects: edit code, run analysis, deploy, and manage project secrets.

Only the **owner** can buy, change, or cancel the subscription. To **change a role or remove a member** (owner or admin), open **Organization settings → Members**, find the person, change their role from the dropdown or click **Remove**; access updates immediately, and you cannot change or remove the owner. For day-to-day work, the documented role requirements for key actions are:

TABLE 21.5

Action	Minimum role
Create a project	admin (or owner)
Edit and commit code; deploy; manage project secrets	member
Invite and manage members	admin (or owner)
Change billing / plan	owner only
View the organization Activity feed	admin (or owner)

Activity and audit. Organization settings → Activity is an audit feed of what happened across the organization — deploys, runs, invites, deletions — with a time-range filter. It is visible to the owner and admins, and is useful both for compliance and for simply seeing who changed what. Each project also has its own audit trail under project **Settings → Activity**.

> **Note:** If an invite fails, the documentation's checklist is: the organization must be cooperative, you must be owner or admin, and your plan must have member room. An invitee without an account creates one from the emailed invitation link, using the invited email address.
>
> **Try it yourself:** Convert a personal organization to cooperative, invite a colleague as a **member**, and confirm on the Team screen that they appear. Then check **Organization settings → Activity** and find the invite event in the audit feed.

You can now find your way around every screen, edit survey code with confidence in its versioning, and structure a team with the right roles. The next chapter picks up where the tour stopped: deploying a survey from the web app, watching responses arrive in the Database, and running analysis to a report — the full fieldwork loop in the cloud.

Chapter Summary

- The web app has a constant top bar (organization switcher, trial countdown, Console button, account menu) and a left sidebar that changes between organization level (Projects, Team, Settings) and project level (Dashboard through Settings).
- The project Dashboard's Data insights charts live data — counts, partial rate, responses per day, frequencies, crosstabs — over all collected responses, once the project opts in with an insights: block in siamang.yaml (the section is off by default).

- The Repository provides a browser editor with syntax highlighting; every save is a commit (**Save & commit** / Ctrl/Cmd+S), and validation runs automatically on every commit with a badge or error count.
- History is fully explorable: commit diffs, branches, pull requests, merges, and optional protection of the `main` branch.
- The Remotes dialog provides local clone/push over HTTPS (SSH keys are registered under Profile → SSH keys, but the SSH address is not shown in the UI) and mirroring to GitHub (Plus plan and above) or GitLab (Pro plan and above); project secrets are encrypted, write-only, and referenced by name.
- Organizations are personal (solo) or cooperative (team, convertible later); the subscription always belongs to the owner.
- Roles are independent of plans: owner (everything, one per organization), admin (members, projects, integrations), member (edit, run, deploy, secrets).
- Organization and project Activity feeds audit deploys, runs, invites, and deletions for owners and admins.

Documentation References

- `wiki/Cloud-Web-App.md` — layout, organization and project screens, settings, Console
- `wiki/Cloud-Repository-and-Editing.md` — editor, commits, branches, secrets, remotes, reports
- `wiki/Cloud-Organizations-and-Team.md` — organization types, invites, roles, activity feed
- `wiki/Cloud-FAQ-and-Troubleshooting.md` — invite checklist and UI troubleshooting

Chapter 22. Cloud Deployment, Data, and Analysis

This chapter picks up where Chapter 21 left off. You have a project, you have edited and committed your questionnaire in the Repository, and every commit has passed automatic validation. Now it is time to put the survey in front of respondents, watch the data arrive, and turn that data into results — all without leaving the browser.

Learning Objectives

By the end of this chapter, you will be able to:

- Deploy a survey from the Siamang Cloud web app into a named environment and share its public URL with respondents.
- Preview a survey safely before it goes live, and monitor fieldwork — including quotas — from the deployment card.
- Explain how response limits are computed from your plan and your `siamang.yaml` environment caps.
- Browse responses in the Database screen, inspect table schemas, and export data in five formats.
- Run committed analysis scripts from the Analysis screen and open the reports they produce.
- Write your own analysis scripts with the Cloud Analysis SDK (`siamang_cloud`): reading tables, weighting, building tables, testing significance, and composing a report.

22.1 Deploying a Survey from the Cloud

Deploying compiles your latest commit and publishes the survey to a live, public URL that respondents open in any browser — no account needed on their side. Everything happens on the **Deployments** screen of your project.

22.1.1 The deploy workflow, links and sharing, and quota handling

Environments. You never deploy "the project" in the abstract; you deploy into a named, isolated run of the survey called an **environment**. Each environment is declared in `siamang.yaml` with its own response cap. The example study "Digital Life & Wellbeing 2026" ships with two environments, which reflects the pattern most projects follow: a small **pilot** first, then the **main** run.

```
environments:
      - { name: pilot, max_responses: 50 }     # a separate deployment + response
    cap
      - { name: main,  max_responses: 1200 }
```

Preview before you deploy. A preview builds a staging version of the survey that you can click through yourself. It shows you exactly what respondents will see, and it does not accept real responses — nothing is written to your data.

1) Open **Deployments**.
2) Click **Preview** (top right).
3) Open the preview link and walk through the survey from start to finish.

> **Tip:** Make the preview walkthrough a habit before every deployment. It is the fastest way to catch a wording mistake, a broken skip, or a missing quota cell before real respondents do.

Deploy. Deploying requires the **member** role or higher, and a commit that passes validation — a broken survey never goes live by accident.

1) On **Deployments**, click **New deployment**.
2) Pick an **environment** (for a first field test, `pilot`).
3) Confirm to build and publish the latest commit.
4) Watch the build **Logs** stream on the deployment card; the status moves to **Live**.

A deployment card always shows one of four statuses:

TABLE 22.1

Status	Meaning
Live	The survey is published and accepting responses.
Building	The platform is compiling and publishing the latest commit.
Failed	The build did not complete; read the Logs on the card, fix the issue, and redeploy.
Stopped	The survey is no longer accepting responses; collected data is preserved.

Share the link. Once the card shows **Live**, it displays the **public survey URL**. Copy it and distribute it however your study recruits: by email, through a panel provider, as a QR code on printed material — anywhere a link can live. Respondents open it in any browser and never sign in.

Monitor fieldwork. While a deployment is Live, its card is a live monitor of your fieldwork. It shows:

- responses collected so far, and that count against the environment's cap, with a progress bar;
- **quota** cells and how full each one is, so you can see at a glance which segments are closing;
- the survey's **codebook** — the list of variables and their response codes — for quick reference.

The same incoming responses also appear row by row in the **Database** screen and feed the live charts on the **Dashboard**, so you can watch a quota fill up on the deployment card and immediately inspect the underlying rows in the database.

Stop or redeploy. Two buttons on the deployment card control the lifecycle after launch. **Stop** takes a live survey offline so it stops accepting responses; everything already collected stays in the database. **Redeploy** builds and publishes again from a stopped or failed deployment — the normal path after you have committed a fix.

Response limits. Two caps govern how many responses a deployment can collect, and the effective limit is always the **smaller** of the two:

1) your **plan's** response cap (for example, the Free plan allows 500 responses per project), and
2) the **environment's** `max_responses` cap in `siamang.yaml`.

So on the Free plan, a `main` environment configured for 1200 responses still stops at 500. To raise the plan-side cap, you upgrade the plan; to raise the environment-side cap, you edit `siamang.yaml` and commit.

> **Warning:** If a deployment stops collecting earlier than you expected, check both caps before assuming a bug. The smaller cap wins, and the plan cap is the one that is easy to forget.

> **Try it yourself:** In a project created from the Example study, open **Deployments**, click **Preview**, and complete the survey once in staging. Then deploy the `pilot` environment, copy the public URL, and submit two real responses. Watch the deployment card's response counter and quota cells update, then find the same two rows in **Database → responses**.

22.2 Viewing and Exporting Data

The **Database** screen is where you browse everything your project has collected or computed. The left side lists the project's **tables**: always the `responses` table, plus any additional tables your analysis scripts write (Section 22.3). The data updates as new responses arrive, so this screen works just as well during live fieldwork as after it.

22.2.1 Response viewers, export formats, and steps

Browsing responses. To look at your data:

1) Open **Database**.
2) Pick a table on the left — start with `responses`.
3) On the **Data** tab, page through the rows; click a column to **sort**, and use the row **filter** to narrow down to the records you care about.

In the `responses` table, each answer variable (for example `gender`, `age_group`, `satisfaction`) appears as its own column, alongside identifiers such as `respondent_id` and submission timestamps.

The Schema tab. Before you analyze or export, the **Schema** tab tells you exactly what the table contains: its **columns**, their **types**, and whether each column is nullable. A quick schema check is the reliable way to confirm that a variable came through with the type you expected.

Exporting. Any table can be exported in a few clicks:

1) With the table open, click **Export**.
2) Choose a format.
3) Download the file.

Five formats are available, each suited to a different downstream tool:

TABLE 22.2

Format	File	Notes
CSV	`.csv`	Carries the data; variable and value labels are available via the survey's dictionary.
Excel	`.xlsx`	Like CSV, for spreadsheet workflows; labels via the dictionary.
SPSS	`.sav`	The format to hand a colleague who works in SPSS. This export writes the data without variable or value labels — a labeled .sav is produced by the engine's SPSS export (SurveyData) inside an analysis script, not by Database → Export or db.export_table.
Parquet	`.parquet`	A columnar format for data-science pipelines.
SQLite	`.sqlite`	A single-file snapshot of the whole table.

> **Tip:** If you need to send data somewhere other than a file — object storage, a data warehouse, Google Sheets, or your own database — that is what **Connectors** do (covered in Chapter 23). The simplest way to get data out, though, is always **Database → Export**.

Deleting a single response. Sometimes one record has to go — most commonly to honor a respondent's erasure request under the GDPR, where you act as the data controller for your respondents' data.

1) Open **Database** and select the `responses` table.
2) Find the response and click **Delete**.
3) Confirm.

The response is permanently removed, and the action is recorded in the organization's **Activity** log, so there is an audit trail of the deletion.

The Files screen. Everything your project generates or receives is collected under **Files**, in two groups. **Repository outputs** are the reports and generated tables that are tracked in the repository; **Assets** are files you have uploaded yourself, plus your exports. You can **Upload**, **Download**, or **Delete** from this screen; uploads are capped at 50 MB per file.

22.3 Cloud Analysis and Reporting

Siamang Cloud gives you two ways to make sense of your data: the live **Dashboard** for quick looks, and committed **analysis scripts** for full, reproducible analysis with shareable reports. The rule of thumb is: explore on the Dashboard, publish with scripts.

22.3.1 In-platform analysis and reports

The Dashboard (quick looks). The Dashboard's Data insights section is off by default: a project opts in by declaring an insights: block in siamang.yaml, which pins a set of widgets computed live over all collected responses. Four widget types are available: stats (a summary — total responses, unique respondents, how many are partial, and when the last response arrived); timeseries (a responses-per-day chart for tracking fieldwork progress); frequency (a bar chart of one variable's distribution); and crosstab (a two-way table of two variables, with row and column totals). The variables each widget uses are fixed in the configuration — there are no dropdown pickers on screen. The Dashboard is read-only and meant for exploring. For weighting, significance tests, custom tables, and anything you want to save or share, you use analysis scripts.

Scripts and runs. Analysis in Siamang Cloud is code: ordinary Python scripts that read the project's responses, compute results, and produce tables, files, and reports. You commit them to the repository (conventionally under `scripts/`) and declare each one in `siamang.yaml` as a `type: analysis` task:

```
tasks:
      cleaning:
        type: analysis
        entry: scripts/cleaning.py
        description: "Clean raw responses, write to clean_responses"

      final_tables:
        type: analysis
        entry: scripts/final_tables.py
        description: "Build frequency tables and crosstabs"   # becomes a report section
    title
        report: outputs/final_tables.md                        # the report the script
    saves
        outputs:
          - outputs/final_tables.xlsx                          # extra files to keep
```

The example project wires two steps this way — cleaning and tables. (The fragment above shows the general task shape; the Template starting point calls its second step final_tables, and a weighting step is a common addition you declare yourself.) The platform runs each script in an isolated environment with read access to your project's data, captures whatever it produces, and saves every run.

> **Note:** The example study "Digital Life & Wellbeing 2026" ships two analysis scripts (cleaning.py and tables.py) and declares two analysis tasks. Only the scripts declared as tasks in siamang.yaml appear on the Analysis screen — so declare every script you want to run in the cloud.

Running scripts. On the Analysis screen, the declared tasks appear as a Pipeline strip of chips — one chip per task, with a status dot showing how its last run finished; clicking a chip opens the Run dialog for that task. The Run script and Run all buttons live in the screen header. You can:

1) Click Run script (or a task's chip in the Pipeline strip) to run a single task — useful while iterating on one step.
2) Click **Run all** to execute every step in declaration order (for example, clean → tabulate) and combine their reports into one document. If any step fails, the run stops there.

The details of a run — when it ran, how long it took, whether it succeeded, and chips linking to what it produced — are shown on the run's card in Run history.

Run history. Every run lands in **Run history** and finishes either **completed** (it ran to the end) or **failed** (it raised an error). Each run is shown as a card, in the same style as a deployment: the status, a stepper of the steps it executed, an excerpt of the error if it failed, output chips linking to what the run produced — a report (opens in the Repository), new database tables (which jump you straight to them in the Database screen), and files to download — and a collapsible "View logs" block containing everything the script printed.

Where each kind of output ends up:

TABLE 22.3

Your script writes…	…and it appears in
`Report(...).add(...).save("outputs/report.md")`	The **report**, openable in the Repository and listed in the run's Outputs
`db.write_table("clean_responses", df)`	**Database → Tables**
A file under `outputs/` (for example an `.xlsx`)	**Files** and the run's Outputs (downloadable)
`print(...)`	The run's **Logs**

Reports. A **report** is the shareable document an analysis produces — a clean document built from tables, headings, and notes with the `Report` builder (Section 22.3.2). Reports are generated in **Markdown** and **HTML**; PDF output is planned. An individual script's report (the path in its `report:` declaration) opens in the Repository as a rendered document, with **MD** and **HTML** download buttons, and also appears under **Files**. **Run all** combines every step's report into a single document with a table of contents, each step titled by its `description`, written to the project's reports folder (by default `reports/report.md`).

> **Note:** On the Plus plan and above, you can also run a single script or a full run-all automatically on a cron schedule — for example, refreshing tables and reports nightly during fieldwork. Schedules are managed from the project **Console** and are covered in Chapter 23.

22.3.2 The Cloud Analysis SDK: installation, API, and worked examples

Analysis scripts are written against the **Cloud Analysis SDK**, the `siamang_cloud` package. There is nothing to install: the package is already available in your project's run environment, and the Python version (fixed by the sandbox image) and any additional packages your scripts need — chosen from a curated allowlist, with commit validation flagging anything outside it — are declared in the runtime: section of siamang.yaml. Everything is exposed through one import surface:

```
from siamang_cloud import db, analysis, respondents, Report
```

TABLE 22.4

Import	What you use it for
`db`	List, inspect, read and write your project's tables
`analysis`	Weighting, frequencies, crosstabs, chi-square, Cramér's V
`respondents`	Dedup resumed submissions, completion time, partial flags
`Report`	Build a Markdown / HTML report document

Tables load as pandas `DataFrame`s, so everything you know about pandas, numpy, and scipy applies directly.

The `db` module — project data. Collected answers live in a table called `responses`; your analysis steps add more tables (for example `clean_responses`, `weighted_responses`). The `db` module is the whole interface:

TABLE 22.5

Call	Returns	Notes
`db.list_tables()`	`list[str]`	Names of every table in your project
`db.schema("responses")`	`list[dict]`	Each column's `name`, `type`, `nullable`, `default`
`db.table("responses").to_pandas()`	`DataFrame`	Load a whole table. The `responses` table is flattened so each answer is its own column
`db.write_table("clean", df, if_exists="replace")`	—	Save a DataFrame as a table. `if_exists` is `"fail"` (default), `"replace"` or `"append"`
`db.as_survey_data("responses")`	survey-data object	Load a table wrapped for label-aware analysis
`db.export_table("responses", "out.csv")`	written path	Write a table to a file — format inferred from the extension

There is no free-form SQL: you read tables and work with them in pandas, so for a custom slice you load the table and filter in Python. `export_table` supports `csv`, `parquet`, `xlsx`, `sav` (SPSS), and `sqlite`, chosen by the file extension.

```
from siamang_cloud import db

    print(db.list_tables())                       # e.g. ['responses',
    'survey_meta']
    print(db.schema("responses"))                 # column names and types

    df = db.table("responses").to_pandas()        # one column per answer
    db.export_table("responses", "outputs/responses.sav")   # SPSS for a colleague
```

The `analysis` module — weights, tables, significance. These helpers take and return plain pandas objects, so results drop straight into a `Report`.

TABLE 22.6

Function	What it does
`frequencies(df, "col", weight=None)`	Frequency table with `value`, `count`, `percent`
`crosstab(df, "row", "col", weight=None, normalize=None)`	Two-way contingency table; `normalize="index"` / `"columns"` / `"all"` for percentages
`chi2(df, "a", "b")`	Pearson chi-square: returns `chi2`, `dof`, `p`, `cramers_v`, `n`
`cell_weights(df, "col", targets)`	Post-stratification weights so one variable matches target proportions
`rake_weights(df, targets)`	Raking (iterative proportional fitting) to several marginal targets

The `targets` argument is a mapping of category to share — proportions or counts, normalized for you. The weight functions return a Series scaled to mean 1.0 — store it as a column (df["w"] = …) and pass the column's name as the weight= argument to frequencies or crosstab.

```
from siamang_cloud import analysis, db

    df = db.table("responses").to_pandas()
    freq = analysis.frequencies(df, "satisfaction")
    table = analysis.crosstab(df, "work_mode", "satisfaction", normalize="index")
    test = analysis.chi2(df, "work_mode", "satisfaction")
    print(test["p"], test["cramers_v"])
```

The `respondents` module — response hygiene. Three helpers operate over the loaded `responses` DataFrame to clean up real-world fieldwork artifacts:

TABLE 22.7

Function	What it does
`dedup_responses(df, id_col="respondent_id")`	Collapse repeated submissions (a resume updates, not duplicates); keeps the most recent
`completion_time(df)`	Per-response duration in seconds (prefers a `duration_s` column, else timestamps)
`partial_flag(df, required=[...])`	Boolean `Series`: `True` where any required field is blank

A complete worked example. This script — an extended version of the example study's cleaning.py + tables.py pair, with raking added — reads responses, cleans them, rakes the sample to census margins, computes weighted estimates, and saves a report:

```
"""Key tables (type: analysis) — frequencies, a weighted crosstab, chi-square."""

    from siamang_cloud import Report, analysis, db, respondents

    # 1. Read collected responses (one column per answer).
    raw = db.table("responses").to_pandas()

    # 2. One row per respondent; drop partials and speeders (under 2 minutes).
    df = respondents.dedup_responses(raw, id_col="respondent_id")
    df["duration_s"] = respondents.completion_time(df)
    df["is_partial"] = respondents.partial_flag(df, required=["gender", "age_group", "satisfaction"])
    df = df[~df["is_partial"]]
    df = df[df["duration_s"].isna() | (df["duration_s"] >= 120)]

    # 3. The achieved sample skews young/male; rake it to census margins.
    CENSUS = {
        "gender": {1: 0.49, 2: 0.48, 3: 0.03},
        "age_group": {1: 0.32, 2: 0.36, 3: 0.32},
    }
    df["w"] = analysis.rake_weights(df, CENSUS)

    # 4. Weighted estimates.
    sat = analysis.frequencies(df, "satisfaction", weight="w")
    ct = analysis.crosstab(df, "work_mode", "satisfaction", weight="w", normalize="index")
    test = analysis.chi2(df, "work_mode", "satisfaction")

    # 5. Save a report — it appears under Analysis & Reports.
    (
        Report(
            title="Digital Life & Wellbeing — key tables",
            description="Weighted estimates (raked to census gender x age margins).",
        )
        .heading("Job satisfaction")
        .add(sat, caption="Table 1. Satisfaction, weighted %")
        .heading("Satisfaction by work mode")
        .add(ct, caption="Table 2. Row % within work mode")
        .note(f"chi2={test['chi2']:.1f}, dof={test['dof']}, p={test['p']:.4f}, "
              f"Cramér's V={test['cramers_v']:.2f}")
        .save("outputs/key_tables.md")
    )
    print("report written to outputs/key_tables.md")
```

If a later step needs the intermediate data, persist it as a table:

```
db.write_table("weighted_responses", df, if_exists="replace")
```

And declare the script as a task in `siamang.yaml` so it appears on the Analysis screen:

```
tasks:
      key_tables:
        type: analysis
        entry: scripts/key_tables.py
        description: "Weighted frequencies, crosstab + chi-square"
        report: outputs/key_tables.md
```

Building the report. Every `Report` builder method returns the report itself, so you chain calls and finish with `save()`:

- **Narrative** — `.heading(text)`, `.markdown(md)`, `.note(md)`, `.value(label, v)`, `.divider()`.
- **Inserts** — `.add(component, caption=...)` accepts a pandas `DataFrame` or a library table/chart; `.image(path, caption=...)` embeds a figure.
- **Save** — `.save("outputs/report.md")` writes Markdown; `.save("outputs/report.html")` writes HTML. (PDF output is planned.)

Try it yourself: In your Example project, open `scripts/tables.py` in the Repository, change one `Report` heading and one caption, and commit with **Save & commit**. Then go to **Analysis**, click Run on that script, and open the finished run's card — its report output chip opens your updated report in the Repository. Finally click **Run all** and compare the combined document in `reports/`.

Chapter Summary

- Deploying compiles the latest validated commit and publishes it to a public URL; you deploy into named **environments** (typically `pilot`, then `main`), each with its own response cap in `siamang.yaml`.
- **Preview** builds a staging version that collects no data; use it to walk the survey before every deployment.
- A live deployment card monitors responses against the cap, quota-cell fill, and the codebook; **Stop** preserves collected data, and **Redeploy** republishes after a fix.
- The effective response limit is the smaller of the plan cap and the environment cap.

- The **Database** screen browses tables with sort, filter, and paging; the **Schema** tab documents columns; **Export** produces CSV, Excel, SPSS (`.sav`), Parquet, or SQLite; single responses can be permanently deleted for GDPR erasure, with the action logged.
- The **Dashboard** answers quick questions live; **analysis scripts** — committed, declared in `siamang.yaml`, and run from the Analysis screen — produce reproducible tables, files, and reports, with every run kept in **Run history**.
- The Cloud Analysis SDK (`siamang_cloud`) is preinstalled in the run environment and provides `db` (tables in and out), `analysis` (frequencies, crosstabs, chi-square, weighting), `respondents` (dedup, completion time, partial flags), and `Report` (Markdown/HTML documents).
- **Run all** executes every analysis step in order and merges their reports into one document with a table of contents.

Documentation References

- `wiki/Cloud-Deploying-a-Survey.md` — environments, preview, deploy workflow, monitoring, stop/redeploy, response limits
- `wiki/Cloud-Viewing-and-Exporting-Data.md` — Database browsing, schema, export formats, response deletion, Files screen
- `wiki/Cloud-Analysis-and-Reporting.md` — Dashboard insights, analysis tasks in `siamang.yaml`, runs and run history, reports
- `wiki/Cloud-Analysis-SDK.md` — the `siamang_cloud` package: `db`, `analysis`, `respondents`, `Report`, and the worked examples
- `wiki/Cloud-Quick-Start.md` — the end-to-end loop from deploy to first report
- `wiki/Cloud-Web-App.md` — Deployments, Database, Analysis, and Files screens in the project sidebar
- `wiki/Cloud-siamang-yaml.md` — `environments`, `type: analysis` tasks, `runtime`, and `reports` configuration

Chapter 23. Connectors, Scheduling, and Webhooks

Chapter 22 showed you how to get data out manually (**Database → Export**) and how to run analysis on demand. This chapter covers the three ways Siamang Cloud connects your project to the outside world and to the clock: **connectors** push tables to external stores, **schedules** run your analysis automatically, and **webhooks** notify your own systems when a deploy or run finishes. All three are premium features, gated by plan — schedules and webhooks require Plus or above, and connectors open from Plus, with each target having its own minimum plan (everyday targets such as Google Sheets, Excel 365, and Supabase on Plus; storage, warehouse, and database targets on Pro; MCP servers on Corporate).

Learning Objectives

By the end of this chapter, you will be able to:

- Explain what connectors do, recognize the connector catalog, and declare a connector task in `siamang.yaml`.
- Supply the correct `config` keys and a project secret for each connector target — without ever committing credentials.
- Schedule an analysis script or a full run-all on a cron schedule using the project Console.
- Read and write standard five-field cron expressions in UTC.
- Register a webhook endpoint, choose the events it subscribes to, and verify the signature of incoming deliveries.

23.1 Connectors

A **connector** moves data between your project and an external store. You use one to export a project table — for example, your cleaned responses — to somewhere your team already works, or to pull an external table into your project. Connectors are declared as tasks in `siamang.yaml`, just like analysis scripts, and they open from the Plus plan: everyday targets (Google Sheets, Excel 365, Supabase) need Plus, storage and warehouse targets need Pro, and MCP servers need Corporate. Git mirrors (syncing the repository with GitHub or GitLab) are managed separately under Repository → Remotes and are gated per provider: a GitHub mirror is available from Plus, while GitLab and self-hosted remotes require Pro.

> **Note:** Twelve export targets transfer data today — s3, gcs, azure, sftp, sheets, excel365, supabase, database, bigquery, snowflake, redcap, and http — and two of them (database and supabase) can also import a table into your project; only airtable, dropbox, and mcp

are marked “coming soon”. Live transfers share a few limits: each run replaces the destination’s contents, every column is written as text, exports are capped at 100,000 rows (50,000 for Google Sheets, 10,000 for Excel 365), and connectors run only when you trigger them manually, with no automatic retries.

23.1.1 Connector catalog and configuration keys

The catalog of destinations:

TABLE 23.1

Destination	`target`	Typical use
Object storage — Amazon S3, Cloudflare R2, MinIO	`s3`	Drop an export file into a bucket
Google Cloud Storage	`gcs`	Drop an export file into a GCS bucket
Azure Blob Storage	`azure`	Drop an export file into a container
Your own Postgres database (BYO database)	`database`	Push a table out, or pull one in
Google Sheets	`sheets`	Export a table to a spreadsheet
Google BigQuery	`bigquery`	Sync a table into a dataset
Snowflake	`snowflake`	Sync a table into a warehouse
Excel 365 (OneDrive / SharePoint)	excel365	Export a table to an Excel workbook (from Plus)
Supabase	supabase	Push a table out, or pull one in (from Plus)
SFTP server	sftp	Drop an export file on an SFTP server (from Pro)
REDCap	redcap	Push a table to a REDCap project (from Pro)
Custom HTTP endpoint	http	POST a table (CSV or JSON) to your own endpoint (from Pro)
Airtable	airtable	Coming soon (Plus)
Dropbox	dropbox	Coming soon (Plus)
MCP server	mcp	Coming soon (Corporate)

Declaring a connector. A connector is a `type: connector` task in `siamang.yaml`. It names a `target`, the project `table` to move, a `direction` (`out` exports from your project, `in` imports into it), a project `secret` holding the credentials, and a nested `config:` block with destination-specific settings:

```
tasks:
      export_to_s3:
        type: connector
        target: s3
        direction: out                  # out = export from your project; in = import into
    it
        table: clean_responses          # the project table to move
        secret: aws_creds               # the name of a project secret (set in the web app)
        config:
          bucket: my-research-exports
          key: digital-life/responses.csv
```

The structural rule to remember: destination-specific settings always go **inside** `config:`, never at the top level of the task. Export connectors serialize the table as CSV (the http target can also send JSON), so give file destinations a .csv name. If a required key is missing — for example, an s3 connector without bucket or key — the connector fails when it runs with a "<target>: missing config: …" error, so check your keys against the table before triggering a run. The required keys per target:

TABLE 23.2

target	Required config keys	Secret
s3	bucket, key	Required — JSON with access_key and secret_key (endpoint and region for R2/MinIO)
gcs	bucket, key	Service-account JSON
azure	container, path	JSON credentials object: account + sas_token (optional endpoint)
database	None required (table and schema are optional)	Required — a postgres:// DSN (set it under Project → Secrets)
sheets	spreadsheet_id	Service-account JSON
bigquery	dataset, table	Service-account JSON
snowflake	database, schema, table, warehouse	JSON with account, user, and private_key (key-pair auth, no password)

Two more worked declarations — exporting a cleaned table to BigQuery and publishing a weighted table to a spreadsheet:

```
tasks:
      export_to_warehouse:
        type: connector
        target: bigquery
        direction: out
        table: clean_responses
        secret: bq_service_account
        config:
          dataset: research
          table: responses_export

      publish_to_sheet:
        type: connector
        target: sheets
        direction: out
        table: weighted_responses
        secret: sheets_service_account
        config:
          spreadsheet_id: 1A2b3C4d5E6f7G8h9I0jKlMnOpQrStUvWxYz
```

Credentials. Never put credentials in `siamang.yaml` — that file is committed to your project's Git repository, where every teammate can read it. Instead, each connector's `secret:` field names a **project secret** that you set separately in the web app under project **Settings → Secrets**, and the platform looks up that named secret when the connector runs. Project secrets are write-only: you can set or replace a value, but it is never shown back.

> **Warning:** Treat the `secret:` field as a *reference*, not a place for the value. If a token or service-account key ever lands in a committed file, rotate it immediately and move it into a project secret.
>
> Try it yourself: Practice the declaration pattern. In an Example project, open siamang.yaml in the Repository, add a type: connector task for a target of your choice using the tables above, and commit with Save & commit. Validation runs automatically on the commit and warns you if the target you picked requires a higher plan than yours; a missing config key, by contrast, is only reported when the connector runs — so compare your config: block against the tables above before you trigger a run.

23.2 Scheduling

A **schedule** runs your analysis without anyone clicking a button — for example, cleaning the data and rebuilding the report every night during fieldwork. You choose what runs, on which branch, and how often. Schedules are a **Plus and up** feature.

A schedule runs one of two things:

- **A single script** (`run_script`) — one analysis task, named by its task name from `siamang.yaml`.
- **Run all** (`run_all`) — every analysis task in declaration order, producing the combined report described in Chapter 22.

23.2.1 Console schedule commands and cron examples

Schedules are managed from the project **Console** — a command panel opened with the **Console** button in the project's top bar, or by pressing the **backtick (`)** key. Four commands cover the whole lifecycle:

TABLE 23.3

Command	What it does
`schedules`	List the project's scheduled runs
`schedule add --cron "<expr>" --all`	Schedule a **Run all**
`schedule add --cron "<expr>" --script <name>`	Schedule a single analysis task (named as in `siamang.yaml`)
`schedule rm <id>`	Remove a schedule

Cron expressions. The `<expr>` is a standard five-field cron expression — minute, hour, day-of-month, month, day-of-week — interpreted in **UTC**. A new schedule starts from when you create it; it does not back-fill runs that "would have" happened earlier. Common patterns:

```
0 2 * * *       Every day at 02:00 UTC
   30 7 * * 1-5    07:30 UTC on weekdays (Mon–Fri)
   0 */6 * * *     Every 6 hours
   0 9 * * 1       09:00 UTC every Monday
```

Two applications of the table: a nightly *Run all* is `0 2 * * *`; to re-run just a `tables` script on weekday mornings, schedule a `run_script` for that task with `30 7 * * 1-5`.

> **Tip:** Cron times are UTC, not your local time. If your fieldwork day ends at 18:00 in a UTC+2 timezone, a "nightly" rebuild at local midnight is `22:00 UTC` — that is `0 22 * * *`, not `0 0 * * *`.

To set up that nightly Run all, for example:

1) Open the project and press the backtick (`) key to open the **Console**.
2) Type `schedule add --cron "0 2 * * *" --all` and confirm.
3) Type `schedules` to verify the new entry is listed.
4) To remove it later, run `schedule rm <id>` with the id from the listing.

Scheduled runs appear in **Run history** exactly like manually triggered ones, finishing completed or failed with the same run card — status, output chips, and a collapsible log — so a nightly failure is visible the next morning, with the error excerpt ready to read.

23.3 Webhooks

A **webhook** lets the platform tell *your* systems when something finishes. You register an endpoint URL, and each time a subscribed event happens, Siamang Cloud sends an outgoing `POST` request with a JSON body to that URL. Webhooks are a **Plus and up** feature, configured per organization.

23.3.1 Events, headers, and the signature-verification code example

Four events can trigger a delivery:

TABLE 23.4

Event	Fires when
`deploy.live`	A deployment finished and the survey is live
`deploy.failed`	A deployment failed
`run.completed`	An analysis run finished successfully
`run.failed`	An analysis run failed

Registering an endpoint. Webhooks live in **Organization settings → Integrations**. For each endpoint you configure:

- **URL** — where to `POST`. Slack-compatible receivers that accept this JSON payload work directly; to reach a Slack channel's own incoming webhook, point the endpoint at a small intermediate handler that maps the payload to Slack's message format.
- **Events** — which events to subscribe to; leave the list **empty to receive all events**.
- **Secret** (optional) — a shared secret used to sign each request, so your receiver can verify the delivery genuinely came from Siamang Cloud.

The app keeps a delivery log per endpoint, recording what was sent and whether it succeeded. Failed deliveries are retried automatically with increasing delays before the platform gives up. An endpoint cannot be switched off temporarily — you delete it instead — and deleting it also removes its delivery log, so export anything you still need from the log first.

Delivery format. Each delivery is a JSON `POST` whose body includes the `event` name plus details — for example, `run.completed` carries the run and the script, and `deploy.live` carries the survey URL. The headers are:

TABLE 23.5

Header	Content
`Content-Type`	`application/json`
`X-Siamang-Event`	The event name (for example `run.completed`)
`X-Siamang-Signature`	`sha256=<hex>` — present only when you set a secret; an HMAC-SHA256 signature of the exact request body

Verifying the signature. When you set a secret, always verify deliveries before acting on them. The check is an HMAC-SHA256 over the raw request body:

```
import hashlib
    import hmac

    def verify(secret: str, body: bytes, header: str) -> bool:
        expected = "sha256=" + hmac.new(secret.encode(), body, hashlib.sha256).hexdigest()
        return hmac.compare_digest(expected, header)
```

Warning: Feed `verify` the raw body bytes exactly as received. Do not parse the JSON and re-serialize it first — re-serialization can change whitespace and key order, which changes the bytes and makes a valid signature fail the check.

Try it yourself: Register an endpoint in **Organization settings → Integrations** with the events list left empty, pointing at a Slack-compatible receiver (or a small relay that forwards to a Slack incoming webhook). Then go to a project, run any analysis script from the **Analysis** screen, and watch the run.completed notification arrive at your receiver. Finally, open the endpoint's delivery log to confirm the delivery was recorded as successful.

Chapter Summary

- Connectors move project tables to and from external stores — S3-compatible object storage, GCS, Azure Blob, SFTP, Google Sheets, Excel 365, Supabase, your own Postgres database, BigQuery, Snowflake, REDCap, and custom HTTP endpoints. The Connectors screen opens from Plus, and each target has its own minimum plan (everyday targets on Plus; storage, warehouse, and database targets on Pro; MCP servers on Corporate).
- A connector is a `type: connector` task in `siamang.yaml` with a `target`, `direction`, `table`, a named project `secret`, and destination-specific keys nested inside `config:`; missing config keys surface as an error when the connector runs, while commit validation warns when a target requires a higher plan.

- Credentials never go in `siamang.yaml`; they live in write-only project secrets referenced by name. Only the airtable, dropbox, and mcp targets are marked “coming soon”; the twelve live export targets transfer data on demand (capped at 100,000 rows per run), and database and supabase can also import.
- **Schedules** (Plus and up) run a single analysis task or a full Run all on a five-field cron expression in UTC, managed from the project Console with `schedules`, `schedule add`, and `schedule rm`.
- **Webhooks** (Plus and up) send a signed JSON `POST` to your URL on `deploy.live`, `deploy.failed`, `run.completed`, and `run.failed`; endpoints are registered per organization under **Settings → Integrations**, with a delivery log and automatic retries.
- When you set a webhook secret, verify each delivery’s `X-Siamang-Signature` header using HMAC-SHA256 over the raw request body.

Documentation References

- `wiki/Cloud-Connectors.md` — connector catalog, task declarations, required `config` keys, project secrets
- `wiki/Cloud-Scheduling-and-Webhooks.md` — Console schedule commands, cron expressions, webhook events, headers, delivery log, and the signature-verification example
- `wiki/Cloud-FAQ-and-Troubleshooting.md` — connector troubleshooting and plan gating of schedules and webhooks
- `wiki/Cloud-Web-App.md` — the Console, the Connectors screen, and organization **Settings → Integrations**
- `wiki/Cloud-Subscription-Tiers.md` — plan gating: webhooks and schedules on Plus and up; connectors from Plus with per-target minimum plans (GitHub mirror on Plus, GitLab/self-hosted mirrors on Pro, MCP on Corporate)

Chapter 24. Plans, Cloud Configuration, and FAQ

This closing chapter of Part V gathers the reference material you will return to in daily work: what each plan includes, the one file that wires a cloud project together, and answers to the questions that come up most often.

Learning Objectives

By the end of this chapter, you will be able to:

- Compare the four subscription tiers and choose the right plan for a study or team.
- Explain how the open beta affects trials, plan switching, and billing.
- Read and edit a complete `siamang.yaml` project configuration file, key by key.
- Resolve the most common account, deployment, data, and analysis problems using the documented fixes.
- State the user-relevant points of the Privacy Policy and Terms of Use, including the controller/processor split for respondent data.

24.1 Subscription Tiers

Siamang Cloud offers four plans — **Free**, **Plus**, **Pro**, and **Corporate**. The plan applies to a whole **organization** and every project inside it, and only the organization **owner** can buy, change, or cancel it. Recall from Chapter 21 that plans and roles are independent: the plan decides what the *organization* can do, while your role decides what *you* can do inside it.

24.1.1 The four plans and their limits

TABLE 24.1

	Free	Plus	Pro	Corporate
Price / month	$0	$25	$200	Custom
How to get it	Self-serve	Self-serve	Self-serve	Contact sales
Projects	2	10	Unlimited	Unlimited
Team members	2	15	Unlimited	Unlimited
Responses per project	500	Unlimited	Unlimited	Unlimited
Webhooks	—	Yes	Yes	Yes

	Free	Plus	Pro	Corporate
Schedules	—	Yes	Yes	Yes
Connectors & Git mirrors	—	Everyday targets + GitHub mirror	All targets + GitLab / self-hosted	All + MCP targets
SSO (SAML / OIDC)	—	—	Yes	Yes
Self-hosted deployment	—	—	—	Yes

Free is a genuinely useful, capped tier rather than a stripped-down trial. Every plan includes Git-backed projects with validation on every commit; deployments to a hosted survey URL; the response database with browsing and export to CSV, XLSX, SPSS, Parquet, or SQLite; analysis and reports in Markdown and HTML (PDF planned); the Files screen; and team invitations with roles.

The premium features, by plan:

TABLE 24.2

Feature	Plans	What it does
Webhooks	Plus, Pro, Corporate	Get a signed POST to your own URL when a deploy or run finishes (to reach Slack, point the endpoint at a small intermediate handler — see Chapter 23).
Schedules	Plus, Pro, Corporate	Run an analysis script or a full run-all on a cron schedule.
Connectors & Git mirrors	Plus, Pro, Corporate	Move tables to external stores (everyday targets such as Sheets, Excel 365, and Supabase from Plus; storage, warehouse, and database targets from Pro; MCP servers from Corporate) and mirror the repo to GitHub (from Plus) or GitLab / self-hosted (from Pro).
SSO (SAML / OIDC)	Pro, Corporate	Configure single sign-on for your organization.
Self-hosted deployment	Corporate	Run Siamang Cloud on your own infrastructure for data-residency or control.

Choosing a plan. Free suits individuals, evaluation, and small one-off studies (2 projects, 2 members, up to 500 responses per project). Plus fits small teams running recurring studies that need automation — unlimited responses, webhooks and schedules, everyday connectors (Google Sheets, Excel 365, Supabase), and a GitHub mirror. Pro is for teams needing scale and integrations: unlimited projects, members, and responses, the full connector catalog, GitLab and self-hosted Git mirrors, and SSO. Corporate serves organizations with data-residency or on-premises requirements: everything in Pro plus MCP-server connectors and self-hosted deployment, arranged with sales. When you reach a cap or use a feature your plan does not include, the app tells you and points you to **Billing** — you will not hit a dead end.

> **Note:** During the open beta, every new workspace starts on a **30-day Pro trial** with full access to every Pro feature, and a countdown is shown on the Billing screen and in the top bar. While no payment provider is connected, plan switching is turned off — other plans show "Available at the official release" — and nothing is ever charged; with Stripe connected, upgrades to Plus and Pro are self-serve (Corporate remains via sales). When the trial ends, the workspace is downgraded to Free: your data and projects are preserved, and work continues within the Free limits (2 projects, 2 members, 500 responses per project), with actions beyond those limits blocked behind an upgrade prompt. Billing is per organization, and only the owner manages it.

24.2 siamang.yaml

Every project has a `siamang.yaml` at the repository root. It is the one place where you wire the project together: where the survey lives, which deployment environments exist, which analysis scripts run, the runtime your scripts use, and where reports go. You edit it like any other file — in the web editor or over Git — and commit it. The platform reads it when it validates a commit, deploys a survey, and runs analysis.

24.2.1 Keys and a complete example

Here is a full, annotated configuration for the example study:

```yaml
name: "Digital Life & Wellbeing 2026"  # the project's display name
    org: "research-agency"               # the organization that owns it
    version: "1.0"                       # config format version

    tasks:
      # ── The survey itself ──────────────────────────────────────────
      survey:
        type: survey
        entry: survey/questionnaire.py   # the module that defines `survey = Questionnaire(...)`
        environments:
          - { name: pilot, max_responses: 50 }     # a separate deployment + response cap
          - { name: main,  max_responses: 1200 }

      # ── Analysis scripts (run with the Cloud Analysis SDK) ─────────
      cleaning:
        type: analysis
        entry: scripts/cleaning.py
        description: "Dedup respondents, drop speeders/partials -> clean_responses"

      tables:
        type: analysis
        entry: scripts/tables.py
        description: "Frequencies, a crosstab with a chi-square test, Markdown report"
        report: outputs/tables.md    # the report this script saves

    runtime:
      python: "3.11"                     # interpreter your scripts run on
      packages:
        - "siamang[charts]>=0.5"
        - "pandas>=2.0"
        - "numpy>=1.26"

    reports:
      dir: reports/                      # folder for the combined report
      combined: reports/report.md        # the "Run all" document
      formats: [md, html]                # render Markdown and HTML
```

The keys, section by section:

- **`name` / `org`** — `name` is the display name shown across the web app; `org` is the slug of the owning organization.
- **`tasks`** — a *mapping* of task name to specification, not a list. The key (`survey`, `cleaning`, `tables`, …) is the task's name, and it is how the task is referred to elsewhere — for example, as a section heading in a combined report or the `script_name` of a schedule. Each specification has a `type`.
- **`type: survey`** — one survey task drives deployments. `entry` (required) is the path to the questionnaire module that defines a module-level `survey` object (and optionally an `options` dict for compiler settings and quotas). `environments` is a list of `{ name, max_responses }`; each environment is a separate deployment with its own response cap.
- **`type: analysis`** — one Python script per task, written with the Cloud Analysis SDK (Chapter 22). `entry` (required) is the path to the script; `description` is the human-readable label that becomes the section title in the combined report; `report` is the path to the report the script saves via

`Report.save(...)`; and `outputs` lists extra generated files to keep, such as an Excel export. Markdown outputs are also folded into the combined report by **Run all**.

- **`type: connector`** — moves a table in or out of an external store, with fields `target`, `direction` (`out`/`in`), `table`, `secret`, and a nested `config:` block (Chapter 23; available from Plus, with per-target minimum plans).
- `runtime` — applies to analysis scripts (and survey compilation). python is declarative: every script runs in the same sandbox image, which fixes the interpreter version (3.11), so the key is informational. packages lists extra packages your analysis imports, chosen from a curated allowlist (siamang, pandas, and numpy are the usual entries); commit validation rejects anything outside the allowlist.
- `reports` — only combined (the document's path, for example reports/report.md) is acted on: it is where Run all writes the merged document. dir and formats appear in template configurations but are reserved keys that the platform does not read yet.
- insights — an optional block that pins Dashboard widgets (stats, timeseries, frequency, crosstab) with their variables fixed in the configuration; Data insights are off unless this block is declared (Section 22.3.1).
- database / storage — reserved sections carried by the example project's file (platform-managed database and output-storage settings); the current platform does not read them yet.

Tip: The project Settings → Runtime tab shows the Python version (fixed by the sandbox image, so it is displayed read-only) and the allowlisted packages, but it reads them from siamang.yaml. To change the packages, edit the file and commit — never hunt for a separate runtime settings panel.

24.3 FAQ and Troubleshooting

24.3.1 Frequently asked questions and documented fixes

Account and access

- *Can I use my own editor or scripts?* Yes. Create a personal token under **Profile → API keys**, or clone the repository over HTTPS or SSH from **Repository → Remotes**.
- *I forgot my password.* Click **Forgot password?** on the sign-in screen to receive a reset link by email. If you sign in with GitHub, you have no Siamang Cloud password to reset — manage it at GitHub.
- *Where are the Terms of Use and Privacy Policy?* Under **Profile → Support**, and online in the repository under `docs/cloud/`.

Organizations and team

- *Personal or cooperative?* Personal is a solo workspace; cooperative lets you invite teammates with roles. You can convert personal to cooperative at any time — projects and data stay exactly as they are.
- *Who invites, who pays?* In a cooperative organization, the **owner** or an **admin** invites members. Billing is per organization, and only the **owner** changes the plan.

Projects and deployment

- *Empty, Template, or Example?* Start from **Example** ("Digital Life & Wellbeing 2026") first — it includes sample data, so every screen works immediately. Template is a minimal survey plus two starter scripts; Empty is a bare scaffold.
- *Where is my survey's public link?* On the **Deployments** card once the deployment is **Live**. Respondents need no account.
- *Preview or Deploy?* **Preview** builds a staging version that does not collect data; **Deploy** publishes a live survey that does.

Data

- *How do I get my data out?* **Database → Export**, to CSV, Excel, SPSS (`.sav`), Parquet, or SQLite.
- *Can I delete a single response?* Yes — **Database → Delete** on the row, useful for GDPR erasure. The deletion is logged in the organization's **Activity** feed.

Plans

- *What does Free include, and what is gated?* Free: 2 projects, 2 members, 500 responses per project. Webhooks and schedules need Plus; everyday connectors (Sheets, Excel 365, Supabase) and GitHub mirroring come with Plus; the full connector catalog, GitLab/self-hosted mirrors, and SSO need Pro; MCP connectors and self-hosted deployment are Corporate.
- *Can I change plans during the open beta? While no payment provider is connected, no: every workspace starts on a 30-day Pro trial with full access, other plans show "Available at the official release", and nothing is charged (with Stripe connected, upgrades are self-serve). When the trial ends, the workspace is downgraded to Free — data and projects preserved, work continuing within the Free limits (2 projects, 2 members, 500 responses per project).*

Troubleshooting. The documented fixes for the most common problems:

TABLE 24.3

Symptom	Fix
"Upgrade required" or a disabled button	You reached a plan limit or used a gated feature. During the beta your Pro trial gives full access; afterwards, check **Organization settings → Billing**.
A commit failed validation	The file shows an error/warning count instead of the green **valid** badge. Click the badge, read what is wrong, fix it, and commit again — nothing goes live until it passes.
A deployment failed or is stuck	Open the build **Logs** on the deployment card, fix the issue in the repo, then **Redeploy**. A build cannot start while another is in progress for the same environment — wait for it to finish.
An analysis run failed	Open the run and read its **Logs**. If a step depends on an earlier one (tables may need cleaned and weighted data first), run the earlier step — or use **Run all** to run every step in order.
You cannot invite someone	Check: the organization is cooperative; you are owner or admin; your plan has member room. The invitee does not need an account yet — the invitation is emailed as a pending invite, which counts against your member limit and expires after a limited time.
Clone or push fails	For SSH, add your public key under **Profile → SSH keys** first. If a token stopped working, re-copy the command from **Repository → Remotes**.
Connectors are not moving data	Check that the project secret named in secret: is set (Project → Secrets); that the target is covered by your plan (a 402 error names the required plan); and that your config: keys are correct — a bad key fails at run time with "<target>: missing config: …". Only the airtable, dropbox, and mcp targets are still "coming soon" (they return 409).

Privacy and terms in brief. Two user-facing points from the legal documents matter most in daily work. First, for survey respondent data, you are the data controller and Siamang Cloud acts only as your processor: respondents should be pointed to *your* privacy notice, and you are responsible for lawful basis, notice, and consent. The platform supports your side of that duty — you can export tables at any time and delete individual responses, projects, or your whole account. Second, **you own what you create**: questionnaires, scripts, configuration, and the responses you collect, granting only the limited permission needed to host and run the service. The service itself is operated by Siamang Labs LLC, is currently an open beta offered free of charge and "as is" with no uptime guarantee, uses no advertising trackers and does not sell your data, and requires account holders to be at least 16 years old. Keep your own copies of important data, and contact **info@siamang-team.org** or the GitHub issue tracker with questions.

> **Try it yourself:** Open your project's `siamang.yaml` and locate each key from Section 24.2 in your own file. Then deliberately introduce a small error — for example, remove the `entry` line of an analysis task — and commit. Watch validation flag the problem, read the error detail from the badge, revert the change, and commit again until the badge is green.

Chapter Summary

- Four plans — Free ($0), Plus ($25), Pro ($200), Corporate (custom) — apply per organization; only the owner manages billing, and plans are independent of member roles.
- Free includes the full core workflow with caps (2 projects, 2 members, 500 responses per project); Plus adds webhooks, schedules, everyday connectors, and GitHub mirroring; Pro adds the full connector catalog, GitLab/self-hosted mirrors, and SSO; Corporate adds MCP connectors and self-hosted deployment.
- During the open beta, every workspace gets a 30-day Pro trial; while no payment provider is connected, plan switching is off and nothing is charged, and after the trial the workspace is downgraded to Free (data and projects preserved, work continues within the Free limits).
- `siamang.yaml` wires the project together: `name`/`org`, a `tasks` mapping (`survey` with `environments`, `analysis` scripts with `entry`/`description`/`report`/`outputs`, `connector` tasks), the `runtime` (Python version and packages), and the `reports` section for the combined Run all document.
- The documented FAQ and troubleshooting table covers validation failures, failed deployments and runs, invitation problems, clone/push issues, and connector setup problems.
- You are the data controller for respondent data; you own your content; the beta service is free and "as is" — export early and often.

Closing note. You have now walked the complete Siamang Cloud journey: accounts and organizations (Chapter 20), the web app and repository (Chapter 21), deployment, data, and analysis (Chapter 22), connectors, schedules, and webhooks (Chapter 23), and the plans, configuration, and answers collected here. The appendices that follow provide quick-reference material — including the documentation pages this part draws on (`wiki/Cloud-*.md`, `docs/cloud/privacy-policy.md`, and `docs/cloud/terms-of-use.md`) — and the **Profile → Support** screen inside the app is always the fastest route to help, documentation, and the legal texts.

Documentation References

- `wiki/Cloud-Subscription-Tiers.md` — plan table, premium features, plan positioning, and open-beta billing rules
- `wiki/Cloud-siamang-yaml.md` — the annotated configuration file and every key
- `wiki/Cloud-FAQ-and-Troubleshooting.md` — FAQ entries and documented fixes
- `docs/cloud/privacy-policy.md` — controller/processor split, data rights, retention, security, contact

- `docs/cloud/terms-of-use.md` — ownership of content, acceptable use, beta availability, contact
- `wiki/Cloud-Your-First-Project.md` — the standard project repository layout
- `wiki/Cloud-Web-App.md` — Billing during the beta and the Support screen

Appendices

Appendix A. CLI Reference

Siamang ships a single executable, `siamang`, with four subcommands: `validate`, `preview`, `deploy`, and `init`. The same commands are available through the module interpreter as `python -m siamang ...`. Help is built in at both levels:

```
siamang --help
    siamang <subcommand> --help
    python -m siamang validate my_survey.py
```

Global behavior: loading your questionnaire. Every subcommand that operates on a survey loads it from a `.py` file. By default the CLI reads a module-level attribute named `survey`; override the name with `--attribute ATTR`. If the attribute is missing, the load raises an `AttributeError` telling you to set `survey = sg.Questionnaire(...)` or to pass `--attribute NAME`.

A.1 Commands

A.1.1 `siamang validate`

Checks a survey file for structural errors and lint findings without deploying anything. It runs survey.validate(strict=...), validates the module-level options dict (quotas are checked only here), and then survey.lint(), printing each finding as [severity] [code] message (location). A failure of validate() or of the options check is reported as a validation error and yields exit code 2.

```
siamang validate PATH [--attribute ATTR] [--strict]
```

TABLE A.1

Flag	Default	Description
`PATH`	(required)	Path to a Python file exposing a `Questionnaire`.
`--attribute`	`survey`	Module-level attribute name to load.
`--strict`	off	Pass `strict=True` to `validate()` (also fails on strict-level lint errors).

Exit codes:

TABLE A.2

Code	Meaning
`0`	Valid; no `error`-severity lint warnings.
`1`	Reserved: error-severity lint rules fire only at the strict level, and under --strict they surface as exit code 2.
`2`	`validate()` or the options check raised a ValueError (structural problem).

Note: Exit code 1 is a reserved part of the contract. All error-severity lint rules fire only at the strict lint level, and under --strict the same findings first fail validate(strict=True), so in practice they surface as exit code 2. If you rely on exit codes in CI, test against your installed version.

```
$ siamang validate my_survey.py
    OK — no warnings.

    $ siamang validate draft_survey.py
    [warning] [EMPTY_PAGE] Page 'consent' has no items. (consent)

    $ siamang validate draft_survey.py --strict
    validation error: Strict questionnaire validation failed: EMPTY_PAGE,
    CATEGORICAL_WITHOUT_LABELS

    $ siamang validate broken_survey.py   # show_if references a typo'd variable
    validation error: Page 'demographics' show_if references unknown variables: regon
```

A.1.2 `siamang preview`

Serves the survey locally so you can click through it exactly as a respondent would. It spins up a local FastAPI server with the React frontend and the SQLite backend (`LocalBackend` + `LocalFrontend`). The survey is reachable at `http://127.0.0.1:<port>`; responses land in the `--db` file. Press Ctrl+C to stop. The command also prints a one-line diagnostic about the React compile path (sucrase + esbuild fast path versus in-browser `@babel/standalone`).

```
siamang preview PATH [--attribute ATTR] [--port PORT] [--open] [--db DB]
```

TABLE A.3

Flag	Default	Description
PATH	(required)	Path to the questionnaire `.py` file.
`--attribute`	`survey`	Module-level attribute name.
`--port`	`8000`	Bind port for the local server.
`--open`	off	Open the survey in the default browser on startup.
`--db`	`survey.db`	SQLite file used by the local backend.

```
$ siamang preview my_survey.py --port 8000 --open
    Preview ready at http://0.0.0.0:8000
      survey_id: 42a1c0e9d3f5
      dashboard: sqlite:///survey.db
      [react] sucrase + esbuild minify available — fast path
    Press Ctrl+C to stop.
```

Responses collected during preview can be read back from Python:

```
from siamang.deploy.backends.local import LocalBackend

    df = LocalBackend(path="survey.db").get_responses(survey_id="42a1c0e9d3f5")
```

A.1.3 `siamang deploy`

Runs the full deployment pipeline from the command line, driven by the configuration file (Section A.2).

```
siamang deploy PATH [--attribute ATTR]
                        [--backend NAME] [--frontend NAME]
                        [--profile PROFILE] [--config PATH]
```

TABLE A.4

Flag	Default	Description
PATH	(required)	Path to the questionnaire `.py` file.
`--attribute`	`survey`	Module-level attribute name.
`--backend`	from config	Backend name (see `list_backends()`).
`--frontend`	from config	Frontend name (see `list_frontends()`).
`--profile`	(current config)	Selects a named profile block.
`--config`	`~/.siamang.toml` (already loaded)	Override the config path.

Resolution order: load `--config` if given (otherwise reload `~/.siamang.toml`); apply `--profile` if given; pick the backend/frontend pair from `--backend`/`--frontend`, falling back to the profile's defaults (local/local if unset). Adapter keyword arguments come from the matching backend/frontend config blocks; for the name `local`, deploy does not forward the `[backends.local]`/`[frontends.local]` blocks (the adapters' own optional kwargs — `path`, `host`, `port`, `open_browser` — are used by `siamang preview`).

```
$ siamang deploy my_survey.py --profile production
    Deployed: https://political-trust-2026.vercel.app
      survey_id: 42a1c0e9d3f5
      backend:   supabase
      frontend:  vercel
      dashboard: https://abcdef.supabase.co/project/_/editor

    # CLI override of the configured pair:
    siamang deploy my_survey.py --backend supabase --frontend netlify
```

A.1.4 `siamang init`

Creates the configuration file. Run interactively, it walks you through choosing the default backend/frontend and, when `supabase`/`vercel` are picked, their credentials (secrets are read with `getpass`, so they are not echoed). The config file is written with `chmod 600` applied.

```
siamang init [--path PATH] [--non-interactive]
```

TABLE A.5

Flag	Default	Description
`--path`	`~/.siamang.toml`	Where to write the config.
`--non-interactive`	off	Write defaults (`backend="local"`, `frontend="local"`) and skip prompts.

```
$ siamang init
    siamang init — interactive setup
    Target: /home/you/.siamang.toml
    Default backend (local/supabase) [local]: supabase
    Default frontend (local/vercel) [local]: vercel
    Supabase URL: https://abcdef.supabase.co
    Supabase anon_key:
    Supabase service_key:
    Vercel token:
    Vercel team_id (optional):

    Wrote /home/you/.siamang.toml (chmod 600 applied).

    $ siamang init --non-interactive
    Wrote /home/you/.siamang.toml (defaults: local/local).
```

A.2 Configuration File

`siamang deploy` reads its defaults from a TOML config, conventionally `~/.siamang.toml`. The file has four kinds of tables: [defaults] picks the default backend/frontend pair, [backends.<name>] and [frontends.<name>] hold the constructor kwargs forwarded to the adapters, and [profiles.<name>] blocks override the defaults for a named deployment target.

```
# ~/.siamang.toml

    # Defaults — used when --profile isn't passed
    [defaults]
    backend  = "local"            # or "supabase"
    frontend = "local"            # or "vercel"

    # Adapter kwargs are forwarded to the adapter constructors and are
    # shared across all profiles.
    [backends.supabase]
    url         = "https://abcdef.supabase.co"
    anon_key    = "eyJ..."
    service_key = "eyJ..."

    [frontends.vercel]
    token        = "vercel-token-..."
    project_name = "political-trust-2026"

    # A named profile, selected with `siamang deploy --profile production`.
    # Keys here override [defaults].
    [profiles.production]
    backend  = "supabase"
    frontend = "vercel"
```

Environment variables with the SIAMANG_* prefix (legacy SURVLIB_* names are accepted as fallback) override the backend kwargs from the file:

TABLE A.6

Variable	Purpose
`SIAMANG_SUPABASE_URL`	Supabase project URL (canonical).
`SIAMANG_SUPABASE_ANON_KEY`	Supabase anon key (canonical).
`SIAMANG_SUPABASE_SERVICE_KEY`	Supabase service key (canonical).
`SURVLIB_SUPABASE_URL` / `SURVLIB_SUPABASE_ANON_KEY` / `SURVLIB_SUPABASE_SERVICE_KEY`	Legacy fallbacks for the three values above.

VERCEL_TOKEN is not part of this override mechanism: the Vercel frontend adapter reads it directly from the environment, and only when token is not set in the config — the config value wins. See Section B.1.1 for the full override and direct-read tables.

Documentation References

- `wiki/CLI-Reference.md` — subcommand synopses, flag tables, exit codes, resolution order, example sessions, preview response retrieval.
- `docs/reference/cli.md` — output examples, configuration file format, environment variable overrides, exit-code wording.
- `wiki/Deployment.md` — `siamang preview`/`siamang deploy` behavior and local backend details.

Appendix B. Configuration Reference

This appendix is the complete reference for Siamang's user-level configuration: the `~/.siamang.toml` file, the `siamang.config` Python module, environment-variable overrides, and permission handling. Chapter 2 introduces these topics from a practical, getting-started perspective; everything documented about the configuration system is collected here. All material is drawn from the configuration guide (`wiki/Configuration.md`) unless otherwise noted.

B.1 `~/.siamang.toml`

Siamang reads deployment settings from `~/.siamang.toml` in your home directory. The file holds the default backend/frontend, per-adapter credentials, and named profiles. Create it once with `siamang init`; `siamang deploy` reads it on every run. The `siamang.config` module loads, validates, and saves the file, applies environment overrides, and warns when file permissions would leak secrets.

B.1.1 All documented keys, profiles, and environment variables

File format — four top-level tables. The file is TOML with four kinds of tables:

TABLE B.1

Table	Contents
`[defaults]`	Default `backend` and `frontend` names.
`[backends.<name>]`	Keyword arguments forwarded to that backend adapter's constructor.
`[frontends.<name>]`	Keyword arguments forwarded to that frontend adapter's constructor.
`[profiles.<name>]`	Named overrides of `[defaults]`, selected with `siamang deploy --profile <name>`.

A complete annotated example:

```toml
# ~/.siamang.toml

    [defaults]
    backend  = "local"
    frontend = "local"

    # Adapter kwargs — forwarded to the backend/frontend constructor.
    [backends.supabase]
    url         = "https://abcdef.supabase.co"
    anon_key    = "eyJ..."
    service_key = "eyJ..."

    [frontends.vercel]
    token        = "vercel-token-..."
    project_name = "political-trust-2026"

    # A named profile, selected with `siamang deploy --profile production`.
    # Keys here override [defaults].
    [profiles.production]
    backend  = "supabase"
    frontend = "vercel"
```

Documented behavior of these keys:

- The local backend and local frontend need no configured kwargs — `siamang deploy` does not forward `[backends.local]`/`[frontends.local]` blocks, while the adapters themselves accept optional `path`, `host`, `port`, and `open_browser`, used by `siamang preview` — so a minimal config is just `[defaults]` with both `"local"` — exactly what `siamang init --non-interactive` writes.
- Each `[profiles.<name>]` block overrides `[defaults]`; adapter credential blocks (`[backends.*]`, `[frontends.*]`) are shared across all profiles.

The `Config` object. Loaded configuration is represented as an immutable dataclass:

```python
@dataclass(frozen=True, slots=True)
    class Config:
        defaults:  dict[str, Any]
        backends:  dict[str, dict[str, Any]]
        frontends: dict[str, dict[str, Any]]
        profiles:  dict[str, dict[str, Any]]
        path:      Path | None
```

Methods on `Config`:

TABLE B.2

Method	Returns	Description
`default_backend()`	`str`	`defaults["backend"]` or `"local"`.
`default_frontend()`	`str`	`defaults["frontend"]` or `"local"`.
`backend(name)`	`dict`	Kwargs for a backend; raises `ConfigError` if unconfigured.
`frontend(name)`	`dict`	Kwargs for a frontend; raises `ConfigError` if unconfigured.
`with_profile(name)`	`Config`	A copy with `[profiles.name]` merged over `defaults`; raises `ConfigError` if undefined.
`to_toml()`	`str`	Serialise back to TOML.

Module functions. Imports: `from siamang.config import Config, load, save, use_profile, current, ConfigError`.

TABLE B.3

Function	Description
`load(path="~/.siamang.toml")`	Read and parse the file, apply env overrides, set it as the active config. A missing file yields a Config with only the environment overrides applied (no error).
`current()`	The active `Config` (set by the last `load`/`use_profile`), or an empty one.
`use_profile(name)`	Switch the active config to `current().with_profile(name)`.
`save(config, path=None)`	Write `config` to disk as TOML, creating parent directories and applying `chmod 600`. Returns the `Path`.

Typical usage:

```
from siamang.config import load
    cfg = load()                          # ~/.siamang.toml
    cfg.default_backend()                 # "local"
    prod = cfg.with_profile("production") # merged defaults
    prod.default_backend()                # "supabase"
    cfg.backend("supabase")               # {"url": ..., "anon_key": ..., "service_key":
    ...}
```

Constructing and saving a configuration programmatically:

```
from siamang.config import Config, save
    cfg = Config(
        defaults={"backend": "supabase", "frontend": "vercel"},
        backends={"supabase": {"url": "...", "anon_key": "...", "service_key": "..."}},
        frontends={"vercel": {"token": "...", "project_name": "wave-4"}},
    )
    save(cfg, "~/.siamang.toml")          # chmod 600 applied
```

Environment overrides and precedence. load() overlays environment-variable values onto the matching adapter kwargs — environment wins over file, and the overlays apply even when no config file exists on disk. The prefix maps to an adapter; the remainder of the variable name (lowercased) becomes the kwarg key, so `SIAMANG_SUPABASE_URL` sets `backends.supabase["url"]`. Both canonical `SIAMANG_*` and legacy `SURVLIB_*` prefixes are recognized.

TABLE B.4

Prefix	Target
`SIAMANG_SUPABASE_*`	`backends.supabase`
`SIAMANG_GSHEETS_*`	`backends.gsheets`
`SIAMANG_VERCEL_*`	`frontends.vercel`
`SIAMANG_NETLIFY_*`	`frontends.netlify`
`SURVLIB_*` (same suffixes)	Legacy fallback for each of the above

Documented examples (wiki/Configuration.md; MANUAL.md; README.md):

```
export SIAMANG_SUPABASE_URL="https://abcdef.supabase.co"
    export SIAMANG_SUPABASE_ANON_KEY="eyJ..."
    export SIAMANG_SUPABASE_SERVICE_KEY="eyJ..."
    export SIAMANG_GSHEETS_CREDENTIALS_FILE="./service-account.json"
```

Direct adapter reads (independent of the config loader, used when kwargs are blank):

TABLE B.5

Variable	Read by
`VERCEL_TOKEN`	`VercelFrontend`
`NETLIFY_AUTH_TOKEN`	`NetlifyFrontend`

This keeps tokens out of the file entirely — ideal for CI. MANUAL.md additionally lists `SIAMANG_SUPABASE_URL`, `SIAMANG_SUPABASE_ANON_KEY`, `SIAMANG_SUPABASE_SERVICE_KEY`, and `VERCEL_TOKEN`, and notes that legacy `SURVLIB_*` names are accepted as fallback. README.md's

Google Sheets deploy example uses `SIAMANG_GSHEETS_CREDENTIALS_FILE` and `NETLIFY_AUTH_TOKEN`.

Secrets and permissions.

- `load()` calls `check_permissions(path)` on POSIX and logs a warning if the file is group- or world-accessible (any `0o077` bits set), suggesting `chmod 600`.
- `check_permissions(Path(...))` returns `True` if the mode is `0600`-clean; otherwise it logs a warning and returns `False`.
- `save()` and `siamang init` set `0o600` automatically.
- Guidance: keep secrets out of version control; prefer environment variables for CI; reserve the file for local machines with `chmod 600`.

Documentation References

- `wiki/Configuration.md` — file location and format, the four tables, `Config` dataclass and methods, module functions, environment overrides and precedence, profiles, permissions and security guidance.
- `MANUAL.md` — documented environment variable names and `SURVLIB_*` fallback note.
- `README.md` — project layout entry for `config/` (“User configuration (~/.siamang.toml), secrets”) and Google Sheets deploy environment examples.
- `CHANGELOG.md` — `SIAMANG_SUPABASE_*` environment names with legacy `SURVLIB_SUPABASE_*` fallback.

Appendix C. Conventions, Glossary, and Further Resources

This appendix collects the working vocabulary of the book, the project and naming conventions used throughout, and pointers to the official documentation, repository, and licensing terms.

C.1 Glossary

C.1.1 Key terms

Terms are listed alphabetically. Class and function names appear in code font; conceptual terms in plain text.

TABLE C.1

Term	Definition
Aggregate root	The top-level object owning a model tree. In Siamang the `Questionnaire` is the aggregate root: it owns pages, variables, and scripts, and provides `validate()`, `compile()`, `simulate()`, `deploy()`.
Answer	One respondent's value for a variable, stored in the collected `DataFrame` under `variable.name`; contrasts with the `Variable`, which is the definition.
Backend	The service that stores responses. Bundled: `local` (SQLite), `supabase`, `gsheets`.
Banner table (`BannerTable`)	Multi-cell cross-break table from `data.tables.banner(rows=..., columns=...)`; exports to CSV/Excel.
`Block`	Named container of questions and nested blocks inside a page; used for shared titles, bulk `show_if`/`hide_if`, and randomization.
Codebook	Machine-readable dataset documentation — labels, scales, missing-value conventions, provenance. In Siamang, the `VariableMap` *is* the codebook.
Connector	Siamang Cloud feature moving data between a project and external stores (object storage, SQL databases, Google Sheets, BigQuery, Snowflake); declared as a task in `siamang.yaml`.
`CrossTable`	Declarative two-way table (`data.report.crosstab(...)`) with automatic labels, percentages, and Chi-square/Cramér's V.
`DeployResult`	Object returned by `survey.deploy(...)`; carries the public `url` and lets you `collect()` accumulated responses.
Expression	Typed condition language for visibility and routing (`age.ge(18) & gender.eq(2)`); evaluable in Python, compiled to SurveyJS for the frontend.
`FilterRule`	Python-predicate escape hatch for conditions; evaluates in Python only — for analysis-time filtering and simulation, not frontend gating.

Term	Definition
`FreqTable`	Declarative one-way frequency table (`data.report.freq(...)`) with counts, percentages, and automatic value labels.
Frontend	The host that publishes the survey bundle. Bundled: `local`, `vercel`, `netlify`.
`FrontendBuilder`	Layer-2 orchestrator combining a runtime adapter, a `UIConfig`, and a backend client template into a `SurveyBundle`.
`GroupMeanTable`	Declarative table of group means (`data.report.means(..., by=...)`) with an automatically selected comparison test.
Kish effective sample size	ESS = $(\Sigma w)^2/\Sigma w^2$ under weighting, from `data.analysis.effective_sample_size()`; never exceeds the raw N.
`LikertScale`	Question type rendering a symmetric ordinal rating scale (`points >= 2`, anchor labels, optional "Not applicable").
`Matrix`	Question type rendering a grid of subquestions — one `Variable` per row sharing a column scale; one dataset column per row.
`Media`	Image, video, or audio attachment to a prompt or option; `kind` inferred from the URL extension.
`MissingValue`	Structured non-substantive response (code + label + kind such as `refusal` or `dont_know`).
`Option`	Enriched answer choice with its own `show_if`/`hide_if` and optional `Media`; used in a question's `choices=` list.
Organization	Siamang Cloud workspace owning projects, team, and plan; personal (solo) or cooperative (owner/admin/member roles).
`Page`	One screen in the survey; unique `name` used as a branching target, plus `items`, navigation, and visibility fields.
`Question`	Base class of the seven question types; binds a prompt (text) to Variables (var). Not used directly — instantiate one of the subclasses.
`Questionnaire`	The aggregate root of a survey — the module-level object named `survey` that the CLI discovers.
`Quota`	Respondent cap for one cell (`variable`, `target_value`, `limit`); attached at deploy time, enforced by the backend.
`Ranking`	Question type for drag-and-drop preference ordering, optionally limited with `max_ranked`.
`Report`	Composable narrative document embedding tables and charts into one shareable analysis output.
`Script`	Sandboxed JavaScript snippet hooked into one of seven lifecycle triggers (`onInit` … `onRandomize`).
`simulate()`	`Questionnaire.simulate(n=100, seed=42)` — generates synthetic, logically valid responses honoring visibility rules.
`SurveyBundle`	Self-contained static output of the frontend layer (`index.html`, `style.css`, `env.js`, manifest).

Term	Definition
`SurveyData`	Analysis object joining a pandas `DataFrame` of answers to a `VariableMap`, with `report`, `plot`, `analysis`, `processing`, `tables` accessors.
`SurveySchema`	Frozen, platform-agnostic intermediate representation compiled from a `Questionnaire`; consumed by runtimes, backends, and themes.
`UIConfig`	Frozen theming dataclass controlling palette, typography, layout, branding, UI strings, and access gate.
`Variable`	Atomic measurement unit: `name`, `scale` (nominal/ordinal/interval/ratio), `label`, value `labels`, missing-value conventions, codebook metadata.
`VariableMap`	`dict[str, Variable]` registry indexed by name — the survey's central schema, codebook, and bridge between model and data.
Webhook	Siamang Cloud integration sending an outgoing `POST` with a JSON body to your endpoint when a subscribed event (deploy, run) happens.
`siamang.yaml`	Siamang Cloud project configuration file: runtime settings, analysis script tasks, environment response caps, connector tasks.

C.2 Conventions and Resources

C.2.1 Naming and project conventions; documentation and license

Project layout. A survey is one plain Python module (e.g. `my_survey.py`) with a module-level variable named `survey`, discovered automatically by every CLI command (override with `--attribute`). Configuration and credentials live in `~/.siamang.toml` (created once with `siamang init`); local preview writes responses to `survey.db` in the working directory.

Naming. Use lowercase alphanumeric names with underscores for variables (`remote_freq`, `trust_govt`) — the name becomes the dataset column, the default question id, and the token used in expressions. Build each `Variable` once at module level and reuse that object everywhere; with an explicit `VariableMap`, validation requires questions to reference the registered instances.

Validate before deploy. Run `siamang validate my_survey.py` before every preview and deploy (exit code 0 = valid, 1 = reserved — error-severity lint findings only occur at the strict level and surface as exit 2 there — 2 = structural failure or a failed options check); add `--strict` before any public launch. In Siamang Cloud, validation runs automatically on every commit.

Resources.

- Source repository: `https://github.com/Siamang-Labs/siamang`. Issues: `https://github.com/Siamang-Labs/siamang/issues`.
- In-repo documentation: `README.md`, `MANUAL.md`, `docs/` (`getting-started`, `concepts`, `cookbook`, and the `docs/reference/` API pages), plus the `wiki/` pages mirrored to the GitHub Wiki. A complete worked example lives in `examples/full_pipeline/`.
- Siamang Cloud (hosted platform, open beta): `https://app.siamang.org/login`.

License. Siamang is source-available: free for noncommercial use — personal projects, research, education, charities, universities, public research bodies, and government institutions — under the PolyForm Noncommercial License 1.0.0. Commercial use is available only through Siamang Cloud or, for self-hosted engine use, under a separate corporate license (see `LICENSE-COMMERCIAL.md`; request by email to info@siamang-team.org). Versions up to and including 0.5.0 remain under the MIT license.

Documentation References

- `README.md` — project identity, license terms, deployment matrix, and project layout.
- `wiki/Home.md` and `wiki/_Sidebar.md` — feature matrix and the full map of Library and Cloud wiki pages.
- `docs/concepts.md` / `wiki/Core-Concepts.md` — the five-layer model and core terminology.
- `docs/reference/core.md` and the other `docs/reference/` pages — authoritative definitions of the glossary classes.
- `wiki/Cloud-Organizations-and-Team.md`, `wiki/Cloud-Connectors.md`, `wiki/Cloud-Scheduling-and-Webhooks.md`, `wiki/Cloud-siamang-yaml.md` — Cloud terms (Organization, Connector, Webhook, `siamang.yaml`).
- `wiki/Validation-and-Linting.md` — `siamang validate` exit codes and the validate-before-deploy habit.
- `LICENSE-COMMERCIAL.md` (referenced from `README.md`) — commercial licensing pointer.