# Mathematical Computing and Reproducible Experiments

**B80 - English-access edition**

O002 project contributors

2026-08-31

# Table of contents

# 1 Introduction

## 1.1 How to use this book

This book treats computation as part of doing mathematics: formulating questions, representing objects, running experiments, checking programs, preserving a record of the work, and explaining what the results actually establish.

The only formal prerequisite is A30-level competence: algebra, functions, and trigonometry at the precalculus level. Two compulsory primers introduce Python from the beginning, before the twelve main units. Calculus, linear algebra, probability, and differential equations appear only after the reader reaches the relevant courses; they are not hidden prerequisites for Unit 1.

Every experiment in this book is intended to run locally with open software. A computational result does not become a mathematical proof merely because a computer produced it. Computation can nevertheless reveal counterexamples, test implementations, guide conjectures, and produce a record that other people can inspect.

This is a separate English edition of the original Indonesian course *Komputasi Matematis dan Eksperimen yang Dapat Direproduksi*. The Indonesian 14-unit edition, version `2026.08.22.1`, remains available at https://doi.org/10.5281/zenodo.22053905, with editable source at https://github.com/KokunoYumeto/mathematical-computing-reproducible-experiments-id. Its historical 12-unit release, `2026.08.22`, remains available at https://doi.org/10.5281/zenodo.22052053. The English edition preserves the fourteen unit identifiers and all seventy-five exercises; it does not replace either Indonesian release.

The permanent DOI for this English edition is https://doi.org/10.5281/zenodo.22210474.

This English edition has its own source, build and validation record. Its source repository is https://github.com/KokunoYumeto/mathematical-computing-reproducible-experiments-en. The release assets provide the English PDF, EPUB and offline learner package. A successful Indonesian build is source evidence, not proof of English validation; consult the English release receipts for the checked boundary.

## 1.2 Local setup

Run these commands from the root of the source package. For the standard Python profile in Windows PowerShell, create a separate environment, activate it, install the frozen package versions, and make the project's kernelspec visible to Jupyter:

```
py -3.13 -m venv .venv
.\.venv\Scripts\Activate.ps1
python -m pip install -r environment/python-lock.txt
$env:JUPYTER_PATH = (Resolve-Path build-support/jupyter).Path
python -m jupyter kernelspec list
```

On Linux or macOS, use the equivalent commands:

```
python3.13 -m venv .venv
source .venv/bin/activate
python -m pip install -r requirements-build.txt
export
 ↪  JUPYTER_PATH="$(pwd)/build-support/jupyter${JUPYTER_PATH:+:$JUPYTER_PATH}"
python -m jupyter kernelspec list
```

The list should include `o002-frozen`. The file `requirements-build.txt` lists the cross-platform root dependencies; `environment/python-lock.txt` records the resolved set of 95 Windows packages used by the Indonesian source release. A POSIX environment resolved from the root dependencies must be recorded as a compatible profile, not as byte-identical to the Windows lock. On Windows, check the runtime against the lock with:

```
python scripts/python_environment.py verify --lock
 ↪  environment/python-lock.txt --receipt
 ↪  environment/PYTHON_ENVIRONMENT_RECEIPT.json
```

Quarto is required to build the reader, and a TeX distribution is required for the PDF. Check them with `quarto --version` and `lualatex --version`. Ordinary Python experiments do not require TeX.

The SageMath laboratory uses a separate profile so that Sage packages do not mix with the standard Python environment. On Windows, use WSL Ubuntu 22.04 and its SageMath 9.5 packages:

```
wsl.exe --install -d Ubuntu-22.04
wsl.exe -d Ubuntu-22.04 -- bash -lc "sudo apt-get update && sudo apt-get
 ↪  install -y sagemath=9.5-4 python3-sage=9.5-4"
wsl.exe -d Ubuntu-22.04 -- /usr/bin/sage --version
python scripts/sage_environment.py verify --lock
 ↪  environment/sage-ubuntu22.04-dpkg-lock.txt --receipt
 ↪  environment/SAGE_ENVIRONMENT_RECEIPT.json
```

If lock verification rejects the local profile, you may still use that profile for learning, but must not describe it as byte-identical to the release environment. The inherited source receipt records Ubuntu 22.04, SageMath 9.5, and the inventory of 1,064 packages actually tested for that source edition. The English edition must record its own execution checks. No paid service or remote runtime is required.

## 1.3 What every unit provides

Every unit contains:

1. learning objectives and local prerequisites;
2. learner-facing text with stable identifiers;
3. code that runs without a paid service;
4. automated tests for claims about programs;
5. exercises, hints, answers, and full solutions, with executable checks for the fifteen additional mastery exercises;
6. discussion of the boundary between experiment, proof, and conclusion; and
7. records of versions, environments, and artifact hashes at production checkpoints.

The standard environment is recorded in the complete Python lock and its verification receipt. The SageMath laboratory uses a separate local profile recorded in the Sage lock and Sage receipt. Package names and versions are part of the experiment, not merely installation suggestions. Check each receipt's edition and execution date: inherited source receipts do not establish English-edition completion.

## 1.4 Licensing, attribution, and source boundaries

All fourteen units were independently authored for the Indonesian course. This English edition translates that original course; it is not a translation of an English donor textbook. No suitable pre-existing English edition of the whole original course was available to reuse.

Patrick Walls, *Mathematical Python*; Irving and colleagues, *Research Software Engineering with Python*; *Scientific Python Lectures*; Hans Fangohr, *Introduction to Python for Computational Science and Engineering*; and the *Official Sage Tutorial 10.9* are references for comparing coverage, software semantics, and production practice. Their existing English material remains available in its original form. No prose, exercises, figures, data, or code from those works has been copied or adapted into this edition. Mentioning them does not imply endorsement.

Useful English references include the Python 3.13 tutorial, the SciPy reference for `scipy.special.exprel`, and the official Sage tutorial. The Python tutorial assumes basic programming knowledge, so it supplements rather than replaces this course's novice primers. A current web reference can describe a newer release than the frozen course environment; check the recorded software version before relying on an API detail.



Translation, production, and quality checks are assisted by **OpenAI Codex gpt-5.6-sol, Ultra**, at the user's direction. This disclosure does not replace credit to authors, sources, rights holders, or human contributors.

# 2 Primer P01 - Running Python Experiments

## 2.1 Learning objectives

After completing this compulsory primer, you will be able to:

- check that the Python interpreter in use belongs to the intended local environment and run commands from a terminal;
- write expressions, bind values to names, and recognize `int`, `float`, `str`, `bool`, and `None`;
- use operators, call functions, import modules, and find help;
- distinguish Jupyter cells, Quarto code blocks, and Python scripts;
- read the important parts of a traceback without guessing;
- distinguish a computed value from the text or display presented to the reader; and
- uncover hidden notebook state by restarting the kernel and running all cells from the beginning.

This primer assumes no previous programming experience. You need only basic arithmetic. All accompanying code for this primer is original, runs locally with the Python standard library, and is located in `source/code/primer01_execution.py`.

## 2.2 The local runtime and terminal

The **runtime** is the program that actually executes the code. In this unit, that runtime is a local Python interpreter. A terminal is a text window in which we ask the system to run programs. Jupyter and Quarto use the same runtime when both are configured to use the same environment.

From the project root, check the runtime with the following commands. Type the commands themselves, not any prompt symbol displayed to their left.

```
python --version
python -c "print(2 + 3)"
```

The first command should report Python 3. The second asks Python to calculate `2 + 3` and then write `5` to the terminal. If `python` is not found, do not install packages at random. First

follow the local setup guide, activate the course environment, and check the active version again.

Three preliminary questions prevent many irreproducible results:

1. What command was run?
2. From which working directory was it run?
3. Which runtime and environment received the command?

A relative path such as `source/code/primer01_execution.py` is interpreted relative to the working directory. For this reason, the examples in this book are run from the project root.

## 2.3 Expressions, names, and assignment

An expression asks Python to produce a value. The number `4`, the expression `2 + 3`, and the call `len("data")` each produce a value. The `=` operator performs **assignment**: Python evaluates the right-hand side and then binds the result to the name on the left-hand side.

```
length = 8
width = 5
area = length * width
area
```

**Listing 2.1**

```
40
```

The name `area` is not a permanent box. A later assignment can bind the same name to a new value. To check whether values are equal, use `==`, not `=`.

```
area == 40
```

**Listing 2.2**

```
True
```

Choose meaningful names, such as `sample_count`, rather than vague names such as `x1`, unless that short symbol is itself part of the mathematical model.

## 2.4 Your first five kinds of scalar values

Common scalar values you will encounter first are:

| Type | Example | Initial meaning |
|---|---|---|
| `int` | `8` | an integer |
| `float` | `0.01` | an approximation to a real number |
| `str` | `"test A"` | text |
| `bool` | `True` | a truth value: true or false |
| `NoneType` | `None` | no value yet, or no result |

```
count = 8
tolerance = 0.01
label = "test A"
passed = count > 5
note = None

type(count).__name__
```

**Listing 2.3**

```
'int'
```

Call `type(...)` with the other names to check their types one at a time. Lists, tuples, loops, and comprehensions are deliberately left until P02; this primer does not need them yet.

`None` is different from `0`, `False`, and the empty string `""`. It indicates that no value has been supplied. Do not use `None` as a number.

The basic arithmetic operators are `+`, `-`, `*`, `/`, `//`, `%`, and `**`. Division with `/` produces a `float`; `//` is floor division; `%` gives the remainder consistent with floor division; and `**` is exponentiation.

```
sum_result = 7 + 2
difference = 7 - 2
product = 7 * 2
quotient = 7 / 2
floor_quotient = 7 // 2
remainder = 7 % 2
power = 7**2

power
```

**Listing 2.4**

```
49
```

All seven names still hold scalar values. No collection has been created; lists and tuples are introduced in P02.

The comparisons `==`, `!=`, `<`, `<=`, `>`, and `>=` produce `bool` values. The operators `and`, `or`, and `not` combine truth values. Use parentheses when the order of operations needs to be clear to the reader.

## 2.5 Function calls, imports, and help

A function accepts arguments and may return a value. In `round(3.14159, ndigits=3)`, `3.14159` is a positional argument and `ndigits=3` is a keyword argument.

```
round(3.14159, ndigits=3)
```

**Listing 2.5**

```
3.142
```

A module groups reusable names. Import the `math` module, then include the module name so that the function's origin remains visible.

```python
import math

radius = 3
circle_area_value = math.pi * radius**2
circle_area_value
```

**Listing 2.6**

```
28.274333882308138
```

Use `help(math.sqrt)` in Python or `math.sqrt?` in a Jupyter cell to view local documentation. Help explains a function's contract; it does not guarantee that the function is suitable for your question. Check the meaning of the arguments, units, domain, return value, and possible errors.

## 2.6 Jupyter cells and Quarto blocks

A Jupyter notebook has code cells and Markdown cells. Code cells are sent to the Python kernel; Markdown cells contain explanations, equations, and the connections between questions and results. Quarto uses fenced code blocks in `.qmd` files and can execute them when building HTML or PDF.

The source of a Quarto code block looks like this:

````
~~~{python}
value = 6 * 7
value
~~~
````

Running one cell does not automatically run all the cells above it. The kernel retains names while its process is alive. Cell execution numbers show the order in which cells actually ran; a cell's position on the page shows only the source order. This distinction matters for reproducibility.

## 2.7 Computation is not display

Python can compute a value without displaying it. In a notebook, the value of the last expression is usually passed to the display mechanism. Assignment usually does not display a

representation of the value. `print(...)` writes text to standard output. In an ordinary script, a bare expression such as `2 + 3` is still evaluated but is not automatically printed.

**Listing 2.7**

```
computed_value = 2 + 3
display_text = f"2 + 3 = {computed_value}"
print(display_text)
```

```
2 + 3 = 5
```

`computed_value` is an `int` with value 5. `display_text` is a `str` chosen for communication. An attractive display does not prove that a computation is correct, and a value in memory is not yet a saved artifact. If a result needs to be audited, save the data and metadata in specified files.

Quarto can capture cell output for inclusion in the reader. That output still needs interpretation: is it a value, diagnostic text, a figure, or an error message?

## 2.8 Reading a traceback

When Python fails, it displays a **traceback**. Read the last line first: it gives the error type and a short message. Then find the nearest frame that points to your file and line of code. The frames above it explain the chain of calls that brought execution there.

```
Traceback (most recent call last):
  File "experiment.py", line 2, in <module>
    result = data * 2
NameError: name 'data' is not defined
```

`NameError` means that a name has not been bound in the namespace being used. `TypeError` usually means that an operation received an unsuitable type of value. `ZeroDivisionError` indicates division by zero. `SyntaxError` means that the source could not be parsed as a Python program. The message is initial evidence, not a reason to delete lines at random.

In P01, run a failing example in a console or practice cell, read the traceback, and then correct its cause. The `try/except` syntax is taught in P02; the book's supplied checking blocks may use it as scaffolding, but you are not yet expected to write that structure yourself. Do not hide failures throughout the program merely to make the output look green.

## 2.9 Your first script and local record

A `.py` file is script source that can be run again. From the project root, run the P01 companion:

```
python source/code/primer01_execution.py --output output/p01-results.json
```

The script computes the same example each time and writes a JSON record. It also prints one location message so that a person can tell which file was created. Open `output/p01-results.json` as text. The `computed_value` field stores the computed value, while `display_text` stores the text chosen for the reader. The runtime fields name the Python implementation and version that actually ran the script.

For now, simply understand that the terminal runs the file from beginning to end. Module structure and the main part of a script are discussed in P02.

## 2.10 Hidden state and restart-and-run-all

Consider the following two cells.

```
# Cell A
data = 7

# Cell B
result = data * 2
```

If Cell A has run before, Cell B may succeed even after Cell A is moved, changed, or deleted from the notebook. The old value of `data` remains alive in the kernel. The notebook appears correct, but the visible source is not sufficient to produce that result. This is **hidden state**.

The compulsory check before accepting a notebook is to:

1. save the source;
2. select **Restart Kernel and Run All Cells**;
3. run the cells in source order from a clean namespace;
4. make sure no unrecorded manual input is required; and
5. compare the generated artifacts with the expected results.

The companion notebook provides three state checks that can be run directly from the Jupyter interface. The required test is to run the last cell with old state, restart the kernel, show that the cell fails on its own, and then select **Restart Kernel and Run All Cells** to show that the complete source sequence succeeds. The code companion also provides `run_cells` for the test suite; tuples, dictionaries, and exception handling within that scaffolding are discussed in P02. The notebook deliberately retains the cross-edition fixture name `hasil` (Indonesian for `result`) so the hidden-state sequence and its versioned receipt remain directly comparable with the source edition.

A successful restart-and-run-all demonstrates that the recorded cell sequence can execute in the current environment. It does not yet prove that the mathematical model is appropriate or that another environment will produce identical bytes.

The same exercise is available in the P01 clean-kernel notebook.

The notebook runs locally and contains all three checks in complete source order; its kernelspec metadata points to the project's frozen `o002-frozen` environment.

## 2.11 Exercises

### 2.11.1 Exercise 1 - values, types, and operators

Create five names: `count=8`, `tolerance=0.01`, `label="test A"`, a boolean `passed` that checks whether `count > 5`, and `note=None`. Also calculate the floor quotient and remainder when dividing `count` by 3. Run the code.

> **Hint**
>
> Use `//` for the floor quotient, `%` for the remainder, and `type(...)` to check the types of the values.

> **Self-check**
>
> Add the following checks below your code. The cell should finish without an `AssertionError`.

```
assert type(count) is int
assert type(tolerance) is float
assert type(label) is str
assert type(passed) is bool and passed is True
assert note is None
assert count // 3 == 2
assert count % 3 == 2
```

Answer and solution

*Solution* 2.1.

```
count = 8
tolerance = 0.01
label = "test A"
passed = count > 5
note = None
floor_quotient = count // 3
remainder = count % 3

assert floor_quotient == 2
assert remainder == 2
```

Each name is bound to the value produced by its right-hand side. The comparison `count > 5` produces a `bool`; `None` is not the number zero.

### 2.11.2 Exercise 2 - imports, calls, and help

Import the `math` module, then calculate the area of a circle of radius 3 using `circle_area` from the P01 companion. Use local help to find the meaning of `math.isclose`, then compare the function result with `math.pi * 3**2`.

Hint

Import the companion with `from source.code.primer01_execution import circle_area`. Call `help(math.isclose)` in a separate console if the output is long.

Self-check

```
import math
from source.code.primer01_execution import circle_area

exercise_area = circle_area(3)
assert math.isclose(exercise_area, math.pi * 9, rel_tol=0.0,
 ↪  abs_tol=1e-15)
```

Answer and solution

*Solution* 2.2.

```
import math
from source.code.primer01_execution import circle_area

exercise_radius = 3
exercise_area = circle_area(exercise_radius)
assert math.isclose(
    exercise_area,
    math.pi * exercise_radius**2,
    rel_tol=0.0,
    abs_tol=1e-15,
)
```

The import provides a module or function name. The call supplies the argument 3 and returns a value. The `isclose` documentation explains the relative and absolute tolerances that form part of the check.

### 2.11.3 Exercise 3 - reading and fixing a traceback

Call `circle_area(-2)`, read the traceback, and state the error type and its final message. Then correct the input to 2 without changing the function merely to hide the error.

Hint

The last line names `ValueError`. Run the failure in a separate console. The following check is supplied scaffolding; its `try`/`except` syntax will be covered in P02.

Self-check

```
from source.code.primer01_execution import circle_area

try:
    circle_area(-2)
except ValueError as radius_error:
    assert str(radius_error) == "radius must not be negative"
else:
    raise AssertionError("negative input should have been rejected")

assert circle_area(2) > 0
```

Answer and solution

The type is `ValueError` and the message is `radius must not be negative`. The function rejects input outside its contract. The correct solution is to supply a nonnegative radius or correct the data source, not to remove the check.

*Solution* 2.3.

```
from source.code.primer01_execution import circle_area

valid_radius = 2
valid_area = circle_area(valid_radius)
assert valid_area > 0
```

### 2.11.4 Exercise 4 - computation, display, and scripts

Run the P01 script from the terminal. In Python, call `computation_and_display(8, 5)`. Identify the field that stores the computed result and the field that stores only display text. Explain why printing text is not the same as saving a JSON record.

Hint

The computed value has type `int`; the display has type `str`. `print` sends text to standard output, whereas the script writes bytes to the specified path. Dictionary indexing in the checking block is supplied scaffolding and will be explained in P02.

Self-check

```
from source.code.primer01_execution import computation_and_display

display_record = computation_and_display(8, 5)
assert display_record["computed_value"] == 13
assert display_record["display_text"] == "8 + 5 = 13"
assert type(display_record["computed_value"]) is int
assert type(display_record["display_text"]) is str
```

Answer and solution

*Solution* 2.4.

```
from source.code.primer01_execution import computation_and_display

display_record = computation_and_display(8, 5)
value_computed = display_record["computed_value"]
text_displayed = display_record["display_text"]
assert value_computed == 13
assert text_displayed.endswith("= 13")
```

Terminal text can disappear when the terminal closes, and it does not bind the result to a runtime version or parameters. The script's JSON file is an artifact that can be read again, checked, and hashed. Both still need to be evaluated against the actual mathematical question.

### 2.11.5 Exercise 5 - clean kernels and hidden state

Use the P01 notebook to demonstrate three facts: Cell B succeeds in an old kernel that has run Cell A; Cell B fails with `NameError` in a clean kernel; and the sequence A followed by B succeeds after **Restart Kernel and Run All Cells**. This is a compulsory mastery exercise.

Hint

Open `source/notebooks/o002-p01-clean-kernel.ipynb`. Follow the three marked states in the notebook and do not add unrecorded manual values. The following checking block is project-test scaffolding that you only need to run in P01; the lists, `Path`, attributes, keyword arguments, `subprocess`, and `sys` it uses are discussed in P02 or later project units.

❗ Self-check

```
from pathlib import Path
import subprocess
import sys

kernel_test = subprocess.run(
    [
        sys.executable,
        "-m",
        "unittest",
        "tests.test_primer01.Primer01NotebookKernelTests",
    ],
    cwd=Path.cwd(),
    capture_output=True,
    text=True,
)
assert kernel_test.returncode == 0, kernel_test.stdout +
 ↪  kernel_test.stderr
```

Answer and solution

```
assert "OK" in kernel_test.stderr
notebook_check_result = "actual kernel: fails out of order; clean run-all
 ↪  passes"
notebook_check_result
```

*Solution* 2.5.

```
'actual kernel: fails out of order; clean run-all passes'
```

The test starts a real Jupyter kernel. Cell B alone fails with `NameError`, whereas the complete notebook runs Cell A followed by Cell B in a new kernel and produces 14. The source order, not an old history of clicks, is the evidence that can be executed again.

## 2.12 Summary

- A terminal runs commands with a particular runtime and working directory.
- An expression produces a value; assignment binds that value to a name.

- `int`, `float`, `str`, `bool`, and `None` have different roles.
- Functions are called with arguments; modules are imported; local help explains contracts that need to be checked.
- Notebooks and Quarto execute cells, whereas a script executes a file from the beginning. Execution order is part of the experiment.
- A computed value, display text, and a saved artifact are not the same thing.
- A traceback gives the location, type, and message of an error; read it before fixing the code.
- Restart Kernel and Run All Cells detects hidden state, but does not replace mathematical checks or a frozen environment.

# 3 Control Flow, Collections, Functions, Modules, and Files

## 3.1 Learning objectives

After completing this primer, you will be able to:

- choose a `list`, `tuple`, `dict`, or `set` to suit the purpose of the data;
- read and modify collections using indices and slices;
- write explicit branches and `for` and `while` loops before expressing them more concisely as a *comprehension*;
- define functions with clear parameters, return values, and local scope;
- reject invalid input with informative exceptions;
- separate importable code from command-line actions through modules and `if __name__ == "__main__"`;
- read and write text, CSV, and JSON using `Path`; and
- carry out a real red-green testing cycle with `unittest`.

Local prerequisite: Primer P01. This primer assumes no previous programming experience and requires no calculus, linear algebra, or differential equations.

## 3.2 Four basic collections

A collection stores several values, but each kind of collection expresses a different contract.

| Type | Example | When to choose it | Important property |
|---|---|---|---|
| `list` | `[4, 1, 4]` | a sequence that can be changed | ordered; duplicates allowed |
| `tuple` | `(4, 1, 4)` | ordered values to be treated as fixed | ordered; immutable |
| `dict` | `{"S01": 4}` | a mapping from keys to values | each key is unique |
| `set` | `{1, 4}` | membership tests or removal of duplicates | has no indices |

Use square brackets to retrieve an element from a list or tuple. Python indices start at zero; negative indices count from the end.

```
values = [4, 1, 4, 2]
first = values[0]
last = values[-1]
middle = values[1:-1]
(first, last, middle)
```

**Listing 3.1**

```
(4, 2, [1, 4])
```

The slice `values[start:stop]` includes position `start` but stops before `stop`. Thus `values[1:3]` takes positions 1 and 2. The form `values[:]` creates a new list with the same values; it is not a second name for the old list.

Lists can be changed. The `append` method adds one element to the same list. Tuples do not support that modification.

```
measurements = [2, 5]
measurements.append(7)
frozen_measurements = tuple(measurements)
(measurements, frozen_measurements)
```

**Listing 3.2**

```
([2, 5, 7], (2, 5, 7))
```

A dictionary uses keys, not positions, to retrieve values. The `get` method can supply a default value when a key is absent. Use ordinary indexing if a missing key is instead an error that should remain visible.

```
by_sample = {"S01": 2, "S02": 5, "S03": 2}
sample_ids = set(by_sample)
unique_values = set(by_sample.values())
(by_sample["S02"], by_sample.get("S99"), sample_ids, unique_values)
```

Do not rely on the display order of a set. If a result must be stable for reading or hashing, convert it to a sorted list, for example `sorted(unique_values)`. Python dictionaries preserve

**Listing 3.3**

```
(5, None, {'S01', 'S02', 'S03'}, {2, 5})
```

insertion order, but this project's canonical JSON still sorts keys so that the incidental order of dictionary construction does not change the output bytes.

## 3.3 Branching with `if`

A branch selects an action based on a boolean value. Python executes only the first block whose condition is true.

```
residual = 0.0008
tolerance = 0.001

if residual < 0:
    status = "error: residual must not be negative"
elif residual <= tolerance:
    status = "passes at the specified tolerance"
else:
    status = "does not yet pass"

status
```

**Listing 3.4**

```
'passes at the specified tolerance'
```

Indentation determines the blocks. Four spaces are the convention in this project. The condition `residual <= tolerance` is a checkable claim about one result; it does not prove that the algorithm is correct for every input.

Comparison operators produce `True` or `False`. Combine conditions with `and`, `or`, and `not`, but do not make a line so dense that domain rules become difficult to review.

## 3.4 Explicit loops: `for`, then `while`

Use `for` when you want to visit every element of a collection. The following example selects the even numbers and then stores their squares.

```
values = [5, 2, -4, 3, 0]
even_squares = []

for value in values:
    if value % 2 == 0:
        even_squares.append(value * value)

even_squares
```

**Listing 3.5**

```
[4, 16, 0]
```

The loop's state can be read step by step: `value` is the element currently being examined and `even_squares` holds the results accepted so far. Master this explicit form before moving to the following compact version.

```
same_result = [value * value for value in values if value % 2 == 0]
same_result == even_squares
```

**Listing 3.6**

```
True
```

A *list comprehension* is not a new algorithm; here it expresses the same loop and branch more concisely. If a transformation needs several conditions, error handling, or intermediate logging, an explicit loop is usually clearer.

### 3.4.1 From comprehensions to generator expressions

Square brackets build the entire list immediately. The same form inside parentheses creates a *generator expression*: values are produced one at a time as they are consumed. The following call to `tuple(...)` exhausts the generator and stores its results as a tuple.

```
generated = (value * value for value in values if value % 2 == 0)
frozen_result = tuple(generated)
(frozen_result, tuple(generated))
```

**Listing 3.7**

```
((4, 16, 0), ())
```

The second result is an empty tuple because a generator can be traversed only once. Thus `[expression for ...]` produces a list immediately, whereas `tuple(expression for ...)` consumes a lazy stream and freezes it. Unit 2 uses the second form to accept any iterable without storing two intermediate copies.

Use `while` when repetition should continue as long as a condition remains true. There must be progress that eventually makes the condition false.

```
current = 3
countdown = []

while current > 0:
    countdown.append(current)
    current -= 1

countdown.append(0)
countdown
```

**Listing 3.8**

```
[3, 2, 1, 0]
```

On each iteration, `current` decreases by one and is bounded below by zero. This fact explains why the loop stops. Without the line `current -= 1`, the condition remains true and the loop never finishes.

## 3.5 Functions, parameters, `return`, and scope

A function names a rule that can be called repeatedly. Parameters are local names that receive values from the caller. `return` sends a result back to the caller.

```
def clipped_mean(values, lower=0, upper=10):
    selected = []
    for value in values:
        if lower <= value <= upper:
            selected.append(value)
    if not selected:
        raise ValueError("no values within the range")
    return sum(selected) / len(selected)

clipped_mean([-4, 2, 6, 20])
```

**Listing 3.9**

```
4.0
```

`values`, `lower`, `upper`, and `selected` are local to the call. The name `selected` is not available outside the function. The defaults `lower=0` and `upper=10` are used only if the caller does not override them.

Avoid making a mathematical function depend on global variables that can change out of sight. It is more transparent to accept decisions as parameters and return results. This makes inputs, outputs, and tests easier to read.

A function without an explicit `return` returns `None`. Printing is not the same as returning: `print` sends text to the screen, whereas `return` supplies a value that a subsequent calculation can use.

## 3.6 Informative exceptions

Input outside the contract should fail clearly. Use `TypeError` when the value's type is unsuitable and `ValueError` when its type is correct but its value is outside the domain.

```
def reciprocal(value):
    if isinstance(value, bool) or not isinstance(value, (int, float)):
        raise TypeError("value must be a number")
    if value == 0:
        raise ValueError("value must not be zero")
    return 1 / value

(reciprocal(4), reciprocal(-2))
```

**Listing 3.10**

```
(0.25, -0.5)
```

Catch an exception only if that layer knows how to handle it. The following parser adds a line number and then propagates the failure.

```
def parse_integer_lines(text):
    values = []
    for line_number, raw_line in enumerate(text.splitlines(), start=1):
        token = raw_line.strip()
        if not token:
            continue
        try:
            values.append(int(token))
        except ValueError as error:
            raise ValueError(
                f"line {line_number} is not an integer: {token!r}"
            ) from error
    return values

parse_integer_lines("2\n\n-3\n")
```

**Listing 3.11**

```
[2, -3]
```

Do not use `except Exception: pass`. That form erases evidence of failure and may let an experiment produce apparently complete output even though part of the processing never happened.

## 3.7 Modules and the `__main__` boundary

A Python file is a module. Suppose `positive_math.py` contains the following functions.

When run as `python positive_math.py`, Python gives the module the special name `"__main__"` and calls `main`. When a test runs `from positive_math import positive_total`, that block does not execute. Importing therefore obtains the function without silently printing, reading files, or overwriting output.

**Listing 3.12** `positive_math.py`

```
def positive_total(values):
    total = 0
    for value in values:
        if value > 0:
            total += value
    return total


def main():
    print(positive_total([3, -8, 2]))
    return 0


if __name__ == "__main__":
    raise SystemExit(main())
```

Standard-library and third-party imports go at the top of a file. Local imports refer to modules that are part of the project. A module name is not text to be executed; it marks a boundary that makes functions usable and testable from other files.

## 3.8 Paths and three file formats

`Path` constructs paths without manually joining Windows or POSIX separators.

```
from pathlib import Path

project_root = Path.cwd()
input_path = project_root / "data" / "measurements.csv"
input_path.as_posix().endswith("data/measurements.csv")
```

**Listing 3.13**

```
True
```

Store paths relative to the project root in configurations and manifests. Do not store personal account folder names or assume that the working directory is always the same.

### 3.8.1 Text

Text files suit simple lines of data. Specify UTF-8 encoding. When bytes must be deterministic across systems, specify LF line endings.

**Listing 3.14** `write_text_example.py`

```
from pathlib import Path

path = Path("output") / "values.txt"
path.parent.mkdir(parents=True, exist_ok=True)
path.write_text("2\n-1\n5\n", encoding="utf-8", newline="\n")
```

### 3.8.2 CSV

CSV stores tables. Use the `csv` module so that commas, quotation marks, and line endings are handled consistently. Open the file with `newline=""`.

First create the course's small sample table by running `python source/code/primer02_control_files.py --output-dir output/primer02-demo` from the project root. That command creates the CSV file used below; the example does not assume that an external data file already exists.

**Listing 3.15** `read_csv_example.py`

```
import csv
from pathlib import Path

rows = []
with Path("output/primer02-demo/measurements.csv").open(
    "r", encoding="utf-8", newline=""
) as stream:
    for row in csv.DictReader(stream):
        rows.append({
            "sample_id": row["sample_id"],
            "value": int(row["value"]),
        })
```

Values read from CSV start out as strings. Conversion with `int` or `float` is part of validation, not merely a cosmetic step.

### 3.8.3 JSON

JSON stores lists, mappings, strings, numbers, booleans, and `null`. It does not directly store `Path` objects, sets, or functions. Convert the data to a supported form and specify serialization rules.

The independent three-sample object below illustrates JSON syntax; it is not a summary computed from the four-row CSV example above.

```
import json

summary = {"count": 3, "sample_ids": ["S01", "S02", "S03"]}
json_text = json.dumps(
    summary, ensure_ascii=False, indent=2, sort_keys=True
) + "\n"
json_text
```

**Listing 3.16**

```
'{\n  "count": 3,\n  "sample_ids": [\n    "S01",\n    "S02",\n    "S03"\n  ]\n}\n'
```

CSV works well for flat tables; JSON works well for nested structures; plain text works well for very simple, explicitly specified formats. A file extension does not prove that its contents are valid. Always parse and validate the structure you need.

## 3.9 Red-green testing with `unittest`

Red-green testing means seeing a test fail because the behavior is not yet correct, fixing the code, and then seeing the same test pass. Do not write code and tests that are already green and claim that the red stage was checked.

Create two files in an empty practice directory. The first version of the module is deliberately wrong:

**Listing 3.17** `positive_math.py`

```
def positive_total(values):
    return sum(values)  # wrong: includes negative values in the sum
```

The test states the intended behavior:

**Listing 3.18** `test_positive_math.py`

```
import unittest

from positive_math import positive_total


class PositiveTotalTests(unittest.TestCase):
    def test_only_positive_values_are_added(self):
        self.assertEqual(positive_total([3, -8, 2, 0]), 5)


if __name__ == "__main__":
    unittest.main()
```

Run:

```
python -m unittest -v test_positive_math.py
```

The test should be **red**: the incorrect function produces `-3`, not `5`. Fix the module using the loop and `if` from the preceding section, then run the same command. The test should be **green**. Add a case with no positive values and a case with rejected input. Record the command and Python version; the interface color is not evidence, but the process status and test listing provide a checkable result.

A green test provides evidence of correct behavior for the cases executed. It does not prove a theorem about all programs or all inputs. Function contracts, choices of boundary cases, and mathematical reasoning remain necessary.

## 3.10 Deterministic companion artifacts

The original MIT-licensed companion code is located in `source/code/primer02_control_files.py`. From the project root, run:

```
python source/code/primer02_control_files.py --output-dir
 ↪  output/primer02-demo
```

The command creates `values.txt`, `measurements.csv`, `summary.json`, and `manifest.json`. All outputs use UTF-8, LF, a fixed field order, and canonical JSON serialization. The manifest records the relative paths, byte sizes, and SHA-256 hashes of the three data artifacts. It uses no network, current time, user folder, or random numbers.

Run the companion tests with:

```
python tests/test_primer02.py -v
```

These tests check contracts, failure types, imports without side effects, the four output files, hash bindings, and two byte-identical bundles.

## 3.11 Exercises

### 3.11.1 Exercise 1 - collections and slices

In a learner module named `latihan_p02.py`, write `collection_report(values)` for a nonempty list of integers. Return a dictionary containing the first element, the last element, a slice without either endpoint, the first two elements as a tuple, the sorted unique values, and a `positions` dictionary mapping string-form indices to values. Reject an empty list with `ValueError`.

> **Hint**
>
> Use indices `0` and `-1`, the slice `1:-1`, `tuple(values[:2])`, `sorted(set(values))`, and `enumerate` to construct `positions`.

> **Executable check**
>
> From the project root, run:
>
> ```
> python tests/test_primer02.py --learner-module latihan_p02.py --exercise
> ↪ 01
> ```
>
> The check loads the `latihan_p02.py` you wrote, not the project's answer module. It checks all fields, result ordering, and rejection of an empty list.

Answer and solution

```python
def collection_report(values):
    if not values:
        raise ValueError("values must not be empty")
    return {
        "first": values[0],
        "last": values[-1],
        "middle": values[1:-1],
        "first_pair": tuple(values[:2]),
        "unique_sorted": sorted(set(values)),
        "positions": {
            str(index): value for index, value in enumerate(values)
        },
    }

collection_report([4, 1, 4, 2])
```

**Listing 3.19**

```
{'first': 4,
 'last': 2,
 'middle': [1, 4],
 'first_pair': (4, 1),
 'unique_sorted': [1, 2, 4],
 'positions': {'0': 4, '1': 1, '2': 4, '3': 2}}
```

The set is used only for uniqueness; `sorted` restores deterministic order. Per-element type validation can be added before constructing the report.

### 3.11.2 Exercise 2 - control flow before comprehensions

Add `even_squares_explicit(values)` to `latihan_p02.py`. For `[5, 2, -4, 3, 0]`, it should build the list of squares of the even numbers using `for`, `if`, and `append`. Write an equivalent `even_squares_comprehension(values)`. Add `countdown(start)` using `while` to produce `[start, ..., 1, 0]` and reject `start < 0`.

Hint

An even number satisfies `value % 2 == 0`. In the `while` loop, decrease the control variable exactly once on every iteration.

Executable check

```
python tests/test_primer02.py --learner-module latihan_p02.py --exercise
↪ 02
```

The check imports all three functions from the learner module, compares the explicit form with the comprehension, and checks that the countdown stops at zero and rejects a negative starting value.

Answer and solution

```
def even_squares_explicit(values):
    result = []
    for value in values:
        if value % 2 == 0:
            result.append(value * value)
    return result

def even_squares_comprehension(values):
    return [value * value for value in values if value % 2 == 0]

def countdown(start):
    if start < 0:
        raise ValueError("start must be nonnegative")
    result = []
    while start > 0:
        result.append(start)
        start -= 1
    result.append(0)
    return result

values = [5, 2, -4, 3, 0]
explicit = even_squares_explicit(values)
compact = even_squares_comprehension(values)
(explicit, compact, countdown(3))
```

**Listing 3.20**

```
([4, 16, 0], [4, 16, 0], [3, 2, 1, 0])
```

Both lists of squares should equal `[4, 16, 0]`. The explicit form shows the sequence of operations that the comprehension then expresses more concisely.

### 3.11.3 Exercise 3 - functions, scope, and errors

Add `safe_mean(values)` to `latihan_p02.py`. The function should return the mean as a `float`, use a local variable `total`, reject an empty collection with `ValueError`, and reject nonnumeric elements with `TypeError`. Show that `total` cannot be accessed after the function finishes.

**Hint**

Freeze the iterable as a tuple, check for emptiness, and then sum through a loop. Reject `bool` even though it is technically a subclass of `int`.

**Executable check**

```
python tests/test_primer02.py --learner-module latihan_p02.py --exercise
↪ 03
```

The check calls `safe_mean` from the learner module and requires the correct mean for the example, with distinct exception types for an empty collection and text elements.

Answer and solution

```
def safe_mean(values):
    frozen = tuple(values)
    if not frozen:
        raise ValueError("values must not be empty")
    total = 0.0
    for value in frozen:
        if isinstance(value, bool) or not isinstance(value, (int,
        ↪  float)):
            raise TypeError("each element must be a number")
        total += value
    return total / len(frozen)

safe_mean([2, 4, 9])
```

**Listing 3.21**

```
5.0
```

The result is `5.0`. The name `total` exists only inside the function; trying to read `total` from the global scope raises `NameError`. That exception is evidence of scope, not an error to hide.

### 3.11.4 Exercise 4 - a learner module and red-green tests

Create your own module named `positive_math.py` and a test file named `test_positive_math.py`. Start with the incorrect implementation using `sum(values)`, run the test so that it fails for `[3, -8, 2, 0]`, and then fix the function using a loop and `if`. Add a test for `[-3, 0]`.

Hint

The first test should expect `5`; the second should expect `0`. Keep the demo call inside `if __name__ == "__main__"` so that importing the module in a test prints nothing.

Executable check

In the directory containing the two learner files, run `python -m unittest -v test_positive_math.py`. The project's reference check is also available:

```
python tests/test_primer02.py --learner-module positive_math.py
 ↪  --learner-test test_positive_math.py --exercise 04
```

Save the red-stage and green-stage outputs as evidence that the same test actually detected the repair. The project check above runs your test file twice in an isolated directory: the `sum(values)` implementation must be red and your final module must be green. The check also imports your `positive_math.py` and examines additional cases.

Answer and solution

Final contents of `positive_math.py`:

**Listing 3.22** `positive_math.py`

```
def positive_total(values):
    total = 0
    for value in values:
        if value > 0:
            total += value
    return total


def main():
    print(positive_total([3, -8, 2, 0]))
    return 0


if __name__ == "__main__":
    raise SystemExit(main())
```

Final contents of `test_positive_math.py`:

**Listing 3.23** `test_positive_math.py`

```
import unittest

from positive_math import positive_total


class PositiveTotalTests(unittest.TestCase):
    def test_mixed_values(self):
        self.assertEqual(positive_total([3, -8, 2, 0]), 5)

    def test_no_positive_values(self):
        self.assertEqual(positive_total([-3, 0]), 0)


if __name__ == "__main__":
    unittest.main()
```

The incorrect version is red because `sum([3, -8, 2, 0]) == -3`. The final version is green because it sums only values satisfying `value > 0`.

### 3.11.5 Exercise 5 - text, CSV, JSON, and a manifest

Add `write_demo_bundle(output_dir)` to `latihan_p02.py`. The function must create an output directory containing: text with one number per line; CSV with columns `sample_id,value`; a JSON summary consistent with both data files; and a manifest containing the relative paths, byte sizes, and SHA-256 hashes of the three artifacts. Add `core_sha256` over the manifest's core section. Run it twice into different directories and show that every pair of corresponding files is byte-identical.

> **Hint**
>
> Use `Path`, `encoding="utf-8"`, `newline=""` for CSV, `lineterminator="\n"`, and `json.dumps(..., sort_keys=True, indent=2)`. Do not include the current time or absolute paths in the output.

Executable check

```
python tests/test_primer02.py --learner-module latihan_p02.py --exercise
↪ 05
```

The check calls `write_demo_bundle` from the learner module in two temporary directories. It parses the numbers in the text, the CSV columns and rows, and the JSON summary; checks consistency of values and sample IDs; ensures that manifest paths are relative; binds sizes, SHA-256, and `core_sha256` to the actual bytes; and then compares each file from the two runs. This exercise check therefore evaluates the output you created, rather than merely running tests against the project's answer module.

Answer and solution

The companion code's `write_demo_bundle` function is the reference solution. It writes all three formats, reads the text and CSV back, creates the JSON summary, and then binds each artifact to the manifest. Run:

```
python source/code/primer02_control_files.py --output-dir
↪ output/primer02-demo
python tests/test_primer02.py -v
```

All four files must exist. `manifest.json` must use schema `o002.p02.bundle-manifest.v1`; its three artifact records must match the actual bytes. Matching hashes establish matching bytes for the runs checked, not the scientific correctness of the data or program.

## 3.12 Summary

- Lists and tuples store sequences; dictionaries map keys; sets express unique membership without indices.
- `if`, `for`, and `while` make control flow visible. Use comprehensions after understanding explicit loops.
- Functions receive inputs through parameters, use local state, and send results back through `return`.
- Exceptions should state contract violations and must not be swallowed without handling.
- The `__main__` boundary prevents importing a module from running its side effects.
- `Path`, UTF-8, CSV/JSON parsers, and canonical serialization make file work more portable and checkable.
- A red-green cycle shows that a test can detect one concrete defect; testing is still not a universal mathematical proof.

# 4 Computation, Representation, and Proof

## 4.1 Learning objectives

After completing this unit, you will be able to:

- distinguish mathematical objects, computer representations, and program output;
- explain the difference between exact arithmetic and floating-point arithmetic;
- compare floating-point numbers using stated tolerances;
- prepare an experiment record that allows the experiment to be rerun;
- write small tests of program behavior; and
- distinguish examples, counterexamples, bounded computational evidence, and general mathematical proofs.

Local prerequisites: arithmetic with rational numbers, functions, simple equations, and reasoning about "for every" versus "there exists." No calculus or linear algebra is required.

## 4.2 Three layers that must not be confused

Consider the rational-number identity

$$\frac{1}{10} + \frac{2}{10} = \frac{3}{10}.$$

There are three distinct layers:

1. **Mathematical objects.** Rational numbers and addition are defined mathematically. The equality above is exact.
2. **Representation.** A computer must store objects using particular bit patterns or data structures. Not every rational number has a finite binary floating-point representation.
3. **Program output.** The output depends on the representation, algorithm, environment, and code that actually ran.

A common mistake is to treat the output as the object itself. A program that prints `0.30000000000000004` does not refute the rational-number identity above. It reveals a property of the floating-point representation used by the program.

## 4.3 Exact and floating-point arithmetic

Python provides floating-point numbers for numerical computation and `Fraction` for exact rational numbers.

```
0.1 + 0.2
```

**Listing 4.1**

```
0.30000000000000004
```

```
from fractions import Fraction

Fraction(1, 10) + Fraction(2, 10)
```

**Listing 4.2**

```
Fraction(3, 10)
```

Different representations serve different needs. Exact arithmetic preserves algebraic equalities, but numerators and denominators can grow in size. Floating-point arithmetic is fast and underlies many numerical libraries, but introduces rounding error.

Numerical comparisons must therefore specify both tolerance and scale:

```
from math import isclose

x = 0.1 + 0.2
isclose(x, 0.3, rel_tol=1e-12, abs_tol=1e-15)
```

**Listing 4.3**

```
True
```

A tolerance is not a magic formula. It is part of the experiment specification. We should be able to explain why its magnitude is appropriate for the question being asked.

## 4.4 A reproducible experiment contract

A computational experiment needs at least the following seven elements.

| Element | Question to answer |
|---|---|
| Question | What claim or phenomenon are we investigating? |
| Environment | Which language and software versions are used? |
| Inputs | What are the data, parameters, and their units? |
| Method | Which algorithm and representation are used? |
| Checks | Which tests can detect implementation errors? |
| Outputs | What raw data and summaries are saved? |
| Conclusion | What do the results support, and what remains unproved? |

Run the Unit 1 experiment from the project root:

```
python source/code/unit01_experiment.py
```

The resulting JSON file records the Python version, parameters, exact result, floating-point result, and checks of two conjectures. Running the same command in the same environment should produce identical bytes.

## 4.5 Proof, evidence, and counterexamples

An experiment can check that

$$n^2 + n$$

is even for the first thousand integers tested. This check provides useful computational evidence: it can uncover code errors or counterexamples. However, a thousand cases do not prove the claim for every integer.

A short general proof is

$$n^2 + n = n(n + 1).$$

Of any two consecutive integers, one is even, so their product is even. This argument applies to every integer, not just the inputs we happened to run.

By contrast, a single counterexample is enough to refute a universal claim. The polynomial

$$p(n) = n^2 + n + 41$$

produces primes for many small values, but `p(40)=41²`. A computer can find this example; the factorization explains mathematically why the claim "prime for every nonnegative integer" is false.

## 4.6 Tests as bounded claims

A program test makes a bounded, checkable claim, for example:

- exact rational addition produces `3/10`;
- the floating-point result is close to `0.3` at the specified tolerance;
- the parity checker finds no violation in the test range;
- the experiment reports the counterexample `n=40`; and
- writing the JSON twice with the same environment and parameters produces identical files.

Tests increase confidence in an implementation. They do not, by themselves, prove the theorem being implemented.

## 4.7 Exercises

### 4.7.1 Exercise 1 - representation

Explain in your own words why the output `0.30000000000000004` does not make the equation $1/10 + 2/10 = 3/10$ false.

> **Hint**
>
> Separate the mathematical object from the way the computer stores it.

> **Answer and solution**
>
> The equality is an exact equality of rational numbers. The numbers `0.1` and `0.2` are not stored exactly in the binary floating-point representation Python uses. The small error in the output comes from the approximate representation, not from a change in the rules of rational addition.

### 4.7.2 Exercise 2 - specifying tolerances

Describe one situation where `abs_tol` is appropriate and one where `rel_tol` is more appropriate. Give reasons, not just numbers.

> **Hint**
>
> Compare quantities close to zero with quantities that have a large natural scale.

> **Answer and solution**
>
> For an equation residual that is theoretically zero, an absolute tolerance is appropriate because the error is compared directly with zero. When comparing two distances of roughly a million meters, a relative tolerance is more informative because the acceptable error scales with the distance. In practice, both are often used together and must be derived from the problem's requirements.

### 4.7.3 Exercise 3 - the limits of an experiment

Change the parity-check limit from 1,000 to 100,000. What new conclusion is justified, and what conclusion remains unjustified?

> **Hint**
>
> More cases enlarge the region that has been checked; they do not turn it into the set of all integers.

> **Answer and solution**
>
> Justified: no counterexample was found in the range 0 through 99,999, assuming the checker is implemented correctly. Unjustified: the claim has been proved for every integer. A general proof still requires an argument such as the factorization `n(n+1)`.

### 4.7.4 Exercise 4 - a counterexample

Use the program to find the smallest nonnegative integer for which $n^2 + n + 41$ is composite. Then explain the result without relying on the program.

**Hint**

Check 40 and factor the result.

**Answer and solution**

The smallest value is `n=40`. Exactly, $40^2 + 40 + 41 = 1681 = 41^2$, so the value is composite. This exact calculation provides a mathematical explanation of compositeness that can be checked independently of the program. Establishing that 40 is the smallest such value additionally requires exact primality checks for every integer from 0 through 39; the factorization alone does not establish minimality.

### 4.7.5 Exercise 5 - an experiment record

Read `output/unit01-results.json`. Name two fields needed to rerun the experiment and one additional piece of information that would be needed if the experiment used external libraries.

**Hint**

Look at the parameters and environment.

**Answer and solution**

One possible answer: `limit` and the Python version are needed. If the experiment uses external libraries, the record must also include their names and versions, and ideally an environment file or dependency lockfile from which they can be reinstalled.

## 4.8 Summary

- A mathematical object is not identical to its computer representation.
- Exact and floating-point arithmetic serve different purposes.
- A tolerance is part of the specification, not a repair added after the results look bad.
- A reproducible experiment records its question, environment, inputs, method, checks, outputs, and the limits of its conclusions.
- Many examples can support a conjecture; one counterexample can refute a universal claim; a general proof requires an argument that applies throughout the domain.

# 5 Objects, Values, and Functions

## 5.1 Learning objectives

After completing this unit, you will be able to:

- choose suitable Python representations for integers, rational numbers, and finite sequences;
- distinguish equality of values from identity of objects;
- write functions with stated preconditions, results, and error behavior;
- avoid unintended changes through mutable inputs;
- formulate invariants that tests can check; and
- produce structured output that does not depend on incidental ordering.

Local prerequisites: Unit 1, sets and functions at the A30 level, and basic operations on finite vectors. This unit does not yet require NumPy.

## 5.2 Mathematical kinds and program types

Mathematical statements specify sets of objects and valid operations. Programs need concrete data types. The two are related, but they are not identical.

| Mathematical intent | Initial representation | Important limitation |
|---|---|---|
| integer | `int` | practical size is limited by memory |
| rational number | `Fraction` | numerators and denominators can grow |
| approximation to a real number | `float` | rounding and finite range |
| fixed finite vector | `tuple` | length and component types must be checked |
| collection intended to change | `list` | aliases can observe the same changes |

```
from fractions import Fraction

values = (7, Fraction(7, 3), 7 / 3)
[(type(value).__name__, value) for value in values]
```

**Listing 5.1**

```
[('int', 7), ('Fraction', Fraction(7, 3)), ('float', 2.3333333333333335)]
```

A type name does not prove that a value meets its mathematical intent. A `tuple` of length three can represent a vector in three-dimensional space, color coordinates, or three unrelated pieces of data. The program contract supplies that additional meaning.

## 5.3 Values, identity, and aliases

The `==` operator compares values according to the type's rules. The `is` operator checks whether two names refer to the same Python object. For reproducibility, do not replace a question about values with a question about object identity.

```
a = [1, 2, 3]
b = a
c = list(a)

(a == c, a is c, a is b)
```

**Listing 5.2**

```
(True, False, True)
```

Because `b` is an alias for `a`, a change made through either name is visible through the other. By contrast, `c` initially has the same value but is a different list. Effects like this can make an experiment depend on the order in which cells are executed.

If a vector is intended to be an immutable value, `tuple` expresses that contract more faithfully than `list`. This is not a ban on lists; choose mutation only when it is part of the method.

## 5.4 Functions as contracts

A good computational function specifies three things:

1. **Preconditions:** which inputs are accepted?
2. **Result:** what value is returned when the preconditions hold?
3. **Failure:** what happens when a precondition is violated?

The following example computes the mean of rational numbers exactly.

```
from fractions import Fraction

def mean_fraction(values):
    """Return the exact mean of a nonempty sequence of rational numbers."""
    values = tuple(Fraction(value) for value in values)
    if not values:
        raise ValueError("values must not be empty")
    return sum(values, start=Fraction(0)) / len(values)

mean_fraction([Fraction(1, 3), Fraction(1, 2)])
```

**Listing 5.3**

```
Fraction(5, 12)
```

Converting the input to a `tuple` freezes the values the function actually uses. Once tuple construction is complete, changes to the original list do not affect these local values. This conversion does not promise an atomic snapshot if another thread mutates the input during construction.

## 5.5 Pure functions and changes of state

The following function multiplies a vector by a scalar without changing its argument.

```
def scale_vector(scalar, vector):
    return tuple(scalar * component for component in vector)

v = (2, -1, 4)
(scale_vector(3, v), v)
```

**Listing 5.4**

```
((6, -3, 12), (2, -1, 4))
```

Given the same inputs, a pure function produces the same output and does not change any state outside the function. This makes testing and tracing the origin of results easier. Real programs still need to read files or write output; place those effects at the program's boundaries rather than spreading them throughout every mathematical function.

## 5.6 Invariants and testing

An invariant is a property that must remain true during a particular operation. For scalar multiplication of vectors, testable invariants include:

- the length of the vector does not change;
- multiplying by 1 preserves the value;
- multiplying by 0 produces the zero vector; and
- `scale_vector(a*b, v)` equals `scale_vector(a, scale_vector(b, v))`.

Tests on many examples can detect violations in the implementation. An algebraic proof is still needed to establish the properties for all scalars and all vectors in the intended domain.

## 5.7 Canonical output

Two experiments can produce the same information but different bytes because of field order, spacing, or line endings. When output serves as evidence, specify a canonical form. The Unit 2 script:

```
python source/code/unit02_objects.py --output output/unit02-results.json
```

writes UTF-8 JSON with sorted field names, fixed indentation, and LF line endings. The file's hash then identifies a particular sequence of bytes that can be compared.

## 5.8 Exercises

### 5.8.1 Exercise 1 - choosing representations

Choose an initial representation for each of the following objects and explain its limitations: $2/7$, a numerical approximation to $\sqrt{2}$, and the fixed vector $(1, 0, -1)$.

> **Hint**
>
> Distinguish exact values from approximations, and fixed values from collections that will be changed.

> **Answer and solution**
>
> Use `Fraction(2, 7)` for the exact rational number, `float` for a numerical approximation to the square root of two with a stated tolerance, and `tuple` for the fixed vector. A `float` does not store the square root of two exactly, and a `tuple` does not by itself guarantee that all its components are numeric.

### 5.8.2 Exercise 2 - values and identity

Create two distinct lists with the value `[1, 2]`. Show one comparison that is true and one that is false. Explain the results.

> **Hint**
>
> Use `==` and `is`.

> **Answer and solution**
>
> If `a=[1,2]` and `b=[1,2]`, then `a == b` is true because their values are equal, whereas `a is b` is false because they are different objects. Mathematical questions about whether lists have the same elements in the same order normally use `==`.

### 5.8.3 Exercise 3 - a function contract

Write the preconditions, result, and failure behavior for a function that returns the least nonnegative remainder when $a$ is divided by modulus $m$.

Hint

The modulus must be positive.

Answer and solution

Preconditions: `a` and `m` are integers and `m > 0`. Result: an integer `r` such that `0 <= r < m` and `a-r` is divisible by `m`. If `m <= 0`, the function must reject the input, for example by raising `ValueError`.

### 5.8.4 Exercise 4 - a dangerous alias

Explain why a function that appends an element to its input list can make an experiment's results depend on notebook cell order. Give one remedy.

Hint

Another name may refer to the same list.

Answer and solution

The first call changes the list that a second call later uses. Running cells in a different order produces a different initial state. Possible remedies include returning a new list, using a `tuple` for a fixed value, or making the mutation explicit and resetting the state before the experiment.

### 5.8.5 Exercise 5 - tests and proof

Does testing the four `scale_vector` invariants on a million vectors prove that the implementation is correct for all inputs? Give a precise answer.

Hint

Separate the tested domain from the universal domain.

Answer and solution

No. These tests provide strong evidence on the sample and can find counterexamples, but they do not cover every input. General correctness of the function is established by

checking the implementation's definition against the distributive and associative laws in the stated domain of scalars and components.

## 5.9 Summary

- Program types represent mathematical objects but do not determine their full meaning.
- `==` compares values; `is` compares object identity.
- Functions should specify preconditions, results, and failure behavior.
- Pure functions and immutable values make reproduction and testing easier.
- Invariants connect mathematical contracts with program tests.
- Canonical output makes artifact hashes meaningful.

# 6 Arrays, Shapes, and Vectorization

## 6.1 Learning objectives

After completing this unit, you will be able to:

- create NumPy arrays with a stated shape and `dtype`;
- explain the relationship between indices, axes, and array shape;
- replace simple componentwise loops with vectorized operations;
- use broadcasting only after checking shape compatibility;
- distinguish a memory-sharing *view* from an independent copy;
- formulate invariants for shapes, values, and the state of inputs; and
- limit experimental conclusions so they are not mistaken for general proofs.

Local prerequisites: Units 1-2, simple linear functions, lists and tuples, and coordinates in the plane. This unit assumes neither calculus nor formal linear algebra.

## 6.2 Arrays as data with a contract

An array is more than a collection of numbers. Its minimal contract includes:

1. **shape** (`shape`): the number of positions along each axis;
2. **number of dimensions** (`ndim`);
3. **representation type** (`dtype`); and
4. **the meaning of each axis**, which we must document ourselves.

```
import numpy as np

temperature = np.array(
    [[28.0, 29.5, 30.0], [27.5, 29.0, 31.0]],
    dtype=np.float64,
)
(temperature.shape, temperature.ndim, temperature.dtype)
```

**Listing 6.1**

```
((2, 3), 2, dtype('float64'))
```

The shape `(2, 3)` says only that there are two rows and three columns. It does not say whether the rows represent cities, days, or repeated trials. That meaning is part of the data specification.

A one-dimensional array with shape `(3,)` differs from a single-row table with shape `(1, 3)` and a single-column table with shape `(3, 1)`. All three contain three numbers, but operations along their axes can behave differently.

## 6.3 `dtype` is a representation, not a mathematical set

NumPy stores the elements of an ordinary array in a single `dtype`. An explicit choice makes the contract easier to check:

```
integers = np.array([1, 2, 3], dtype=np.int64)
approximations = np.array([0.1, 0.2, 0.3], dtype=np.float64)
(integers.dtype, approximations.dtype, approximations.sum())
```

**Listing 6.2**

```
(dtype('int64'), dtype('float64'), np.float64(0.6000000000000001))
```

`int64` stores integers within a finite range. Operations that go outside that range can overflow; it is not the same as the set of all mathematical integers. `float64` stores binary approximations with finite precision; it is not the same as the real numbers.

NumPy can use `dtype=object` for Python objects such as integers of arbitrary precision, subject to practical limits, or `Fraction` values, but many benefits of numerical array computation are lost. If the question requires exact arithmetic, deliberately choose an exact tool (such as `Fraction`, SymPy, or Sage) instead of calling a `float64` result exact.

## 6.4 Vectorization and a scalar reference

Suppose each data value $x_i$ is transformed by the function

$$y_i = 3x_i - 2.$$

A clear scalar loop can serve as a reference:

```
x = np.array([1, 2, 3, 4], dtype=np.int64)
y_reference = np.array([3 * int(value) - 2 for value in x])
y_reference
```

**Listing 6.3**

```
array([ 1,  4,  7, 10])
```

NumPy lets us write the same rule for the entire array:

```
y = 3 * x - 2
y
```

**Listing 6.4**

```
array([ 1,  4,  7, 10])
```

Vectorization expresses an array operation without a Python loop that we write ourselves. It is often more concise and can be faster, but brevity is not a proof of correctness. Compare the result with a simple reference and check invariants derived from the formula:

- `y.shape == x.shape`;
- `x` is unchanged; and
- $\sum_i y_i = 3\sum_i x_i - 2n$, where $n$ is the number of elements.

The scalar reference must also be checked. Two implementations that copy the same conceptual error can agree and still be wrong.

## 6.5 Broadcasting with shape checks

Broadcasting extends operations between arrays without requiring us to manually create tiled copies. Suppose each column of a table receives a different offset:

```
table = np.array([[10, 20, 30], [40, 50, 60]], dtype=np.int64)
offset = np.array([1, -2, 3], dtype=np.int64)
table + offset
```

**Listing 6.5**

```
array([[11, 18, 33],
       [41, 48, 63]])
```

The shapes `(2, 3)` and `(3,)` are compatible because their last axes both have size three. The offsets are applied to every row. Do not rely on successful execution as the only check: an unintended shape may still broadcast and produce a plausible-looking answer.

The `add_column_offsets` function in the Unit 3 code specifies a stricter contract: the table must be two-dimensional, the offsets must be one-dimensional, and the number of offsets must equal the number of columns.

## 6.6 Copies and views

A NumPy array slice is usually a *view*: a new object that accesses the original array's memory. Changing the view can change the original data.

```
original = np.arange(6)
view = original[1:4]
view[0] = -10
original
```

**Listing 6.6**

```
array([  0, -10,   2,   3,   4,   5])
```

Use `.copy()` when an experiment needs an independent value:

```
original = np.arange(6)
independent_copy = original[1:4].copy()
independent_copy[0] = -10
original
```

**Listing 6.7**

```
array([0, 1, 2, 3, 4, 5])
```

Make this choice deliberately. Views are useful for avoiding large copies; copies are useful for preventing unexpected changes. `np.shares_memory(a, b)` can help check the relationship in tests.

## 6.7 The Unit 3 experiment

Run from the project root:

```
python source/code/unit03_arrays.py --output output/unit03-results.json
```

The script records the NumPy version, shapes, `dtype`, vectorized results, broadcasting results, view and copy traces, floating-point limitations, and the limits of what the conclusions establish. JSON is written as UTF-8 with sorted keys, fixed indentation, and LF line endings. In the same environment, the same invocation produces identical bytes.

## 6.8 Computation and proof

For the array `[1, 2, 3, 4]`, the program checks that the vectorized operation agrees with the scalar reference. This result supports a bounded claim about the case and implementation that were run. It does not prove equality for every array, every `dtype`, or every size.

For arithmetic without overflow, the general argument proceeds component by component. At index $i$, both implementations compute the same expression, $3x_i - 2$. Since the index $i$ is arbitrary, equality holds at every component; the shape is also preserved because exactly one output is created for each input.

The argument must be qualified by the representation domain. With `int64`, overflow can change the relationship to mathematical integers. With `float64`, rounding can cause algebraically equivalent arrangements of operations to produce different bits. Tests help expose these effects; the specification and argument determine what is actually being claimed.

## 6.9 Exercises

### 6.9.1 Exercise 1 - shape and axis meanings

An array stores measurements from three sensors at four times and has shape `(4, 3)`. Explain the meaning of both axes. What shapes do `data[2, :]` and `data[:, 1]` produce?

**Hint**

A single index removes the selected axis, whereas `:` retains all positions along that axis.

**Answer and solution**

Axis 0 represents the four times, and axis 1 represents the three sensors. `data[2, :]` selects all sensors at the third time and therefore has shape `(3,)`. `data[:, 1]` selects the second sensor at all times and therefore has shape `(4,)`. This meaning comes from the data contract, not the shape alone.

### 6.9.2 Exercise 2 - vectorization and invariants

Vectorize a loop that computes $z_i = 5x_i + 1$. State two testable invariants and derive the sum invariant.

**Hint**

Apply the operation directly to `x`, then sum the component equations.

**Answer and solution**

Write `z = 5*x + 1`. Two useful invariants are `z.shape == x.shape` and that `x` is unchanged. If there are $n$ elements, then

$$\sum_i z_i = \sum_i (5x_i + 1) = 5\sum_i x_i + n.$$

Tests can compare the vectorized result with a scalar reference and check all three properties on selected cases.

### 6.9.3 Exercise 3 - a broadcasting contract

Determine whether the following shape pairs are compatible for addition intended to mean "one offset per column": `(5, 3)` with `(3,)`, `(5, 3)` with `(5,)`, and `(5, 3)` with `(1, 3)`. Explain each answer.

> **Hint**
>
> Compare dimensions from the right. Their sizes must be equal, or one of them must be one.

> **Answer and solution**
>
> `(5, 3)` with `(3,)` is compatible and applies three offsets to every row. `(5, 3)` with `(5,)` is incompatible because the last sizes, 3 and 5, differ. `(5, 3)` with `(1, 3)` is compatible; the first axis, of size one, is expanded to five. Although two pairs are compatible under NumPy's rules, the function contract must still specify that the second input really represents column offsets.

### 6.9.4 Exercise 4 - tracing a view

For the following code, determine the final values of `a`, `b`, and `c` without running it.

```
a = np.array([0, 1, 2, 3])
b = a[1:3]
c = a[1:3].copy()
b[0] = 8
c[1] = 9
```

> **Hint**
>
> `b` shares memory with `a`, whereas `c` does not.

> **Answer and solution**
>
> The final arrays are `a: [0, 8, 2, 3]`, `b: [8, 2]`, and `c: [1, 9]`. The change through `b` affects the same element in `a`. The copy `c` was created before the change, and changes to `c` do not propagate back to `a`. To test an entire array for equality, use a check such as `np.array_equal(a, [0, 8, 2, 3])`. In contrast, `a == [0, 8, 2, 3]` returns an array of elementwise Boolean comparisons, not a single Boolean.

### 6.9.5 Exercise 5 - experiments are not general proofs

You generate a million arrays of small numbers and find that `3*x-2` always agrees with the loop reference. State the strongest justified conclusion, then outline a general proof and give one limitation of the representation.

> **Hint**
>
> Separate the sample's coverage, the componentwise argument, and the behavior of finite `dtype` representations.

> **Answer and solution**
>
> The strongest empirical conclusion is that no difference was found in the million cases and environment tested. For a general proof in the specified arithmetic domain, choose an arbitrary index $i$: both methods compute $3x_i - 2$, so all their components are equal and the shape is preserved. A limitation is that `int64` can overflow and `float64` rounds; therefore, any relationship to integer or real arithmetic must state the range and representation model.

## 6.10 Summary

- `shape`, `ndim`, `dtype`, and axis meanings together form the array contract.
- Vectorization expresses componentwise operations but still needs a reference and invariants.
- Broadcasting is safe to teach and use when both shape and axis intent are checked.
- Slices are usually views; `.copy()` breaks the memory-sharing relationship.
- Canonical output records experiments that can be compared byte for byte.
- Checking many cases provides computational evidence, not a replacement for a general argument; `dtype` limitations must be part of the claim.

# 7 Visualization with Integrity

## 7.1 Learning objectives

After completing this unit, you will be able to:

- begin a visualization with a clearly stated question;
- build a graph progressively using `fig` and `ax`, followed by `plot`, `scatter`, and `errorbar`;
- map variables to positions, markers, lines, and intervals without hiding their units;
- explain how axis truncation and aspect ratio change the visual impression;
- display uncertainty without assigning it an unspecified statistical meaning;
- provide data, figures, and textual descriptions that can be checked again; and
- distinguish patterns in a figure from general mathematical proof.

Local prerequisites: Unit 1, Cartesian coordinates, equations of straight lines, ratios, and intervals at the A30 level. This unit does not require calculus.

## 7.2 A figure begins with a question

A graph is not decoration added after a calculation is complete. It is a structured argument about data. Before choosing colors or line styles, state the question.

In this unit, the question is:

> Are seven distance measurements consistent with the model $d = 2t + 1$, if each measurement has an uncertainty of $\pm 0.4$ meters?

The dataset is deliberately small so that every mark on the graph can be matched to the table.

| Time $t$ (s) | Measured distance $d$ (m) | Uncertainty (m) | Model value (m) |
|---:|---:|---:|---:|
| 0 | 1.0 | 0.4 | 1.0 |
| 1 | 3.1 | 0.4 | 3.0 |
| 2 | 4.8 | 0.4 | 5.0 |
| 3 | 7.2 | 0.4 | 7.0 |

| Time $t$ (s) | Measured distance $d$ (m) | Uncertainty (m) | Model value (m) |
|---:|---:|---:|---:|
| 4 | 9.0 | 0.4 | 9.0 |
| 5 | 10.9 | 0.4 | 11.0 |
| 6 | 13.1 | 0.4 | 13.0 |

Because the question compares measurements with a model, the graph must show both. Showing only the model line would remove the measurement evidence; showing only the points would hide the object of comparison.

## 7.3 Encoding: what means what?

Every visual element must have a meaning that can be written down without looking at the figure.

| Visual element | Meaning |
|---|---|
| horizontal position | time $t$, in seconds |
| vertical position | distance $d$, in meters |
| circular points | measured values |
| vertical bars | measurement intervals $d \pm 0.4$ meters |
| dashed line | model values $d = 2t + 1$ |

Color helps distinguish elements, but it must not be the only encoding. Circular points and a dashed line remain distinguishable in grayscale print or for readers who cannot distinguish particular colors.

Choose an encoding that fits the question. Position on a scaled axis is useful for comparing quantities. A pie chart is unsuitable here: distances at different times are not parts of a single whole.

## 7.4 Progressive lab: from `fig` and `ax` to a complete graph

Matplotlib separates the **whole figure** (`fig`) from the **coordinate region** (`ax`). `fig` controls the canvas that will be saved. `ax` receives the data, scales, labels, and legend. This separation makes each decision visible and testable. The following lab begins with an empty canvas, then adds exactly one layer at each stage.

### 7.4.1 Stage 1 - create `fig` and `ax`

**Listing 7.1**

```
from matplotlib import pyplot as plt
from source.code.unit04_visualization import ALT_TEXT, OBSERVATIONS

fig_u04_lab, ax_u04_lab = plt.subplots(figsize=(8.0, 4.8))
```

Variable names may differ, but the pattern `fig, ax = plt.subplots(...)` is worth retaining: someone reading the code can immediately see which object will be saved and which receives the data encoding. The figure size is fixed so that it does not depend on the interactive window size.

Prepare four data sequences from the table. The lists follow the already validated observation order.

**Listing 7.2**

```
times_u04 = [observation.time_s for observation in OBSERVATIONS]
measured_distances_u04 = [
    observation.measured_distance_m for observation in OBSERVATIONS
]
uncertainties_u04 = [observation.uncertainty_m for observation in
 ↪  OBSERVATIONS]
model_u04 = [observation.model_distance_m for observation in OBSERVATIONS]
```

### 7.4.2 Stage 2 - `plot` for the model

`ax.plot` suits model values ordered by time. The dashed line provides a second encoding in addition to color.

The comma in `model_line_u04, = ...` unpacks the one-element list of lines returned by `plot`. At this stage, there are no measurement points or uncertainty bars yet.

### 7.4.3 Stage 3 - `scatter` for measurements

`ax.scatter` adds the seven measurements as separate markers. Their circular shape keeps the series recognizable without color.

**Listing 7.3**

```
model_line_u04, = ax_u04_lab.plot(
    times_u04,
    model_u04,
    color="#D55E00",
    linestyle="--",
    linewidth=2,
    label="Model d = 2t + 1",
)
assert model_line_u04.get_linestyle() == "--"
```

**Listing 7.4**

```
measurement_points_u04 = ax_u04_lab.scatter(
    times_u04,
    measured_distances_u04,
    marker="o",
    s=36,
    color="#0072B2",
    label="Measurements",
    zorder=3,
)
assert len(measurement_points_u04.get_offsets()) == 7
```

### 7.4.4 Stage 4 - `errorbar` for uncertainty

The uncertainty bars form a separate layer. `fmt="none"` prevents `errorbar` from drawing a second set of points over the `scatter` output.

The `yerr` values specify vertical deviations from each point. Their statistical meaning does not come from the method name; that meaning must still be stated by the measurement procedure.

### 7.4.5 Stage 5 - labels with units and a legend

The data are now visible, but the graph still cannot be interpreted without units, ranges, and an encoding key.

**Listing 7.5**

```
uncertainty_bars_u04 = ax_u04_lab.errorbar(
    times_u04,
    measured_distances_u04,
    yerr=uncertainties_u04,
    fmt="none",
    capsize=4,
    ecolor="#4D4D4D",
    label="Uncertainty (±0.4 m)",
    zorder=2,
)
assert uncertainty_bars_u04.has_yerr
```

### 7.4.6 Stage 6 - an accessible description and deterministic saving

A static figure needs a description that states the question, axes, markers, pattern, and limits of the conclusion. Quarto associates the following description with the lab figure. The same description is available in the `ALT_TEXT` constant and is written to a text artifact by the Unit 4 script.

```
fig_u04_lab
```

**Listing 7.6**

```
ax_u04_lab.set_title("Measured distance and a constant-velocity model")
ax_u04_lab.set_xlabel("Time, t (s)")
ax_u04_lab.set_ylabel("Distance, d (m)")
ax_u04_lab.set_xlim(0, 6.25)
ax_u04_lab.set_ylim(0, 14)
ax_u04_lab.set_xticks(times_u04)
ax_u04_lab.set_yticks(range(0, 15, 2))
ax_u04_lab.grid(True, color="#D9D9D9", linewidth=0.8)
ax_u04_lab.set_axisbelow(True)
legend_u04 = ax_u04_lab.legend(loc="upper left", frameon=True)
fig_u04_lab.subplots_adjust(left=0.11, right=0.97, bottom=0.15, top=0.87)

assert "(s)" in ax_u04_lab.get_xlabel()
assert "(m)" in ax_u04_lab.get_ylabel()
assert len(legend_u04.get_texts()) == 3
```

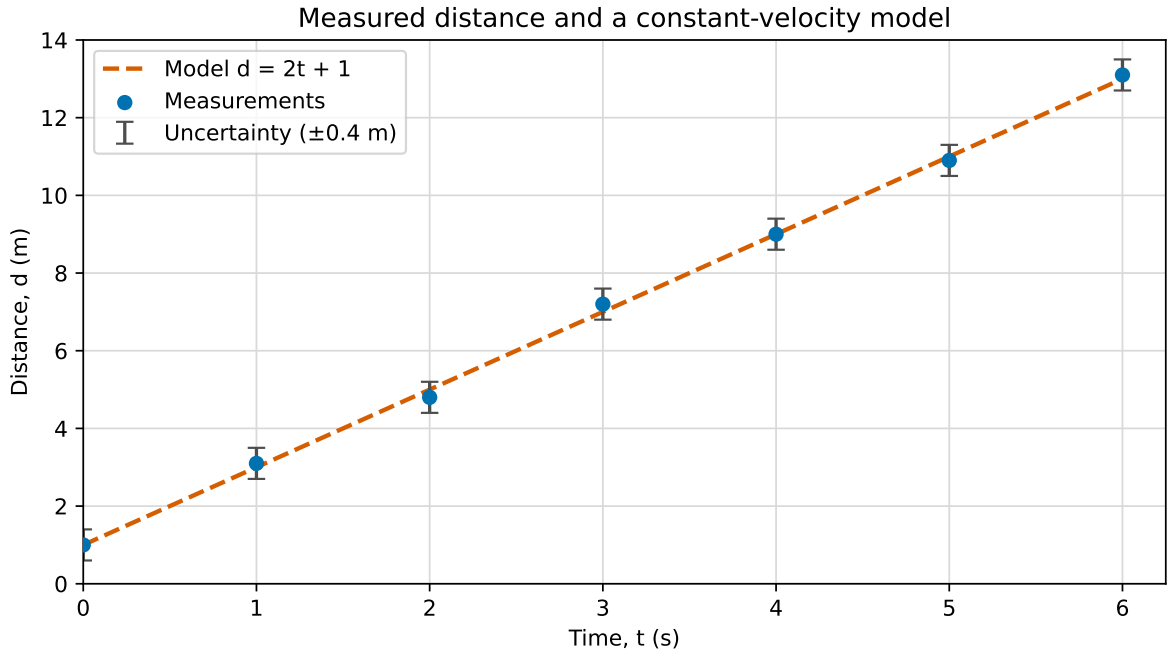


Figure 7.1: The completed construction after adding the model line, measurement points, uncertainty bars, labels with units, and a legend.

Calling `savefig` without fixing the size, metadata, font, and SVG hash salt can produce different bytes. The `write_artifacts` function fixes all of these, saves the data and description alongside the figure, and binds all three in a SHA-256 manifest. Close the lab canvas when finished so that a long-running process does not accumulate figures in memory.

**Listing 7.7**

```
plt.close(fig_u04_lab)
```

## 7.5 Axes, scales, and units

The label `Time, t (s)` states the variable name, symbol, and unit. The label `Distance, d (m)` does the same. Numbers without units are insufficient when comparing physical quantities.

The vertical axis in the Unit 4 figure begins at zero. This choice makes visual height proportional to the stated distance. Starting at zero is not an absolute rule for every line graph, but any truncation must be clearly visible and must not be used to exaggerate differences.

Suppose two bars represent 98 and 100. With a zero baseline, their height ratio is

$$\frac{100 - 0}{98 - 0} = \frac{100}{98} \approx 1.02.$$

If the baseline is hidden at 97, the ratio becomes

$$\frac{100 - 97}{98 - 97} = 3.$$

The difference in the data remains 2, but the second bar appears three times as tall as the first. This truncation may be useful for showing small changes, but the baseline and actual values must be stated clearly.

## 7.6 Aspect ratio and slope

The aspect ratio is the ratio of the plot region's width to its height. A very narrow graph can make the same line look steep; a very wide graph can make it look flat. The model's mathematical slope remains

$$m = \frac{\Delta d}{\Delta t} = 2 \text{ m/s},$$

regardless of the figure's size on screen. Read the axis values and their units, rather than relying only on the angle in pixels. The unit script fixes the figure size and axis limits explicitly so that the result does not change with the window size.

## 7.7 Uncertainty is not decoration

In this example, the error bars represent measurement tolerance intervals $[d - 0.4, d + 0.4]$ meters. They are **not** automatically standard deviations, confidence intervals, or the ranges of all possible values. Those meanings are justified only if a measurement procedure or statistical model establishes them.

All model values lie inside the measurement intervals. A justified statement is, "These data are consistent with the model at the seven times examined." An overstatement is, "The data prove that the model is always correct." Other models may also fit the same seven intervals.

When comparing two series, use the same definition of uncertainty or explain the difference. Bars with different meanings must not be compared as though they were identical.

## 7.8 Text descriptions and accessibility

Useful alternative text does not merely say "line graph." It includes:

1. the graph's purpose;
2. the variables and their units;
3. the encoding of markers and lines;
4. the main pattern and important deviations; and
5. the limits of the conclusion when the figure is easily misinterpreted.

The Unit 4 script embeds a title and description in the SVG metadata and writes the same description as `unit04-alt.txt`. The data remain available as CSV, so readers are not forced to obtain values from visual positions alone.

The legend, marker shapes, and line styles also repeat the information carried by color. This repetition is not wasteful: it makes the information accessible through more than one cue.

## 7.9 Deterministic artifacts

Run from the project root:

```
python source/code/unit04_visualization.py --output-dir output/unit04
```

The command produces four files:

- `unit04-data.csv`: tabular data with units in the column names;
- `unit04-figure.svg`: a vector figure with fixed scales and metadata;
- `unit04-alt.txt`: the visual description in plain text; and
- `unit04-manifest.json`: the question, axis policy, encoding meanings, claim limits, and SHA-256 hashes of the other three artifacts.

The script uses neither the network, the current time, nor random numbers. Data order, number formatting, font, figure size, SVG identifier salt, and metadata are fixed. In the same software environment, two runs must produce the same bytes.

Deterministic does not mean correct. An error written deterministically is still an error. Hashes help ensure that two people examine the same bytes; tests and reasoning check whether those bytes represent the intended meaning.

## 7.10 Visual evidence and mathematical proof

A figure can reveal patterns, anomalies, and possible counterexamples. It is very useful for framing the next question. However, seven points do not prove an equation for all times.

If motion is **defined** to have an initial position of 1 meter and a constant velocity of 2 meters per second, then the equation

$$d(t) = 1 + 2t$$

follows from the definition of constant-velocity motion. That is an argument about the model. The graph then checks whether the limited measurements are consistent with the model's consequences.

Conversely, an observation whose interval does not contain the model value can provide evidence that the model, uncertainty, data, or implementation needs examination. It does not automatically identify which part is wrong.

## 7.11 Exercises

### 7.11.1 Exercise 1 - question and encoding

A friend plots the seven measurements but does not show the model line. Identify the missing information and one improvement that does not rely only on color.

> **Hint**
>
> Return to the main question: which two objects are being compared?

> **Answer and solution**
>
> The graph lacks the reference values $d = 2t + 1$, so readers cannot directly assess whether the measurements are consistent with the model. Add a dashed model line with a legend entry or a direct label. A different line style keeps it distinguishable without relying on color.

### 7.11.2 Exercise 2 - a truncated axis

Two values are 98 and 100. Calculate their apparent height ratio for baselines of 0 and 97. Explain why the title "the second value is three times as large" is unjustified.

> **Hint**
>
> Apparent height is the value minus the baseline; the data magnitude remains the original value.

> **Answer and solution**
>
> With a baseline of 0, the ratio is $100/98 \approx 1.02$. With a baseline of 97, the apparent height ratio is $(100 - 97)/(98 - 97) = 3$. However, the second value is only $100/98$ times the first, not three times. The number 3 describes the geometry of the truncated figure, not the data ratio.

### 7.11.3 Exercise 3 - the meaning of error bars

Are the $\pm 0.4$ meter bars in this unit 95% confidence intervals? Write a justified statement about these bars.

Hint

Use only the definition of uncertainty that was actually given.

Answer and solution

There is no basis for calling them 95% confidence intervals. The unit defines them only as measurement tolerances from $d - 0.4$ to $d + 0.4$ meters. A justified statement is: at all seven times, the model value lies within the stated tolerance interval.

### 7.11.4 Exercise 4 - an accessible description

Write two or three sentences of alternative text for the Unit 4 graph. Do not merely repeat the title.

Hint

Include the axes, encoding, pattern, uncertainty, and limits of the claim.

Answer and solution

Example: "The graph compares measured distance in meters over times 0–6 seconds with the model $d = 2t + 1$. Circular points and $\pm 0.4$ meter bars represent measurements, while a dashed line represents the model; all model values lie inside the measurement intervals. The agreement of seven points supports consistency for these data, not proof that the model holds for all times."

### 7.11.5 Exercise 5 - figures and proof

A graph shows 1,000 points exactly on the curve $y = x^2$. Does this prove that the program always calculates squares correctly? Give one computational step and one mathematical step that strengthen the check.

Hint

The figure contains only finitely many inputs and has limited visual resolution.

Answer and solution

No. A thousand points check only the selected inputs, and small differences may be hidden by the figure's resolution. Computational step: test boundary values, negative values, and deterministic random inputs by comparing the numerical outputs with exact results. Mathematical step: examine the algorithm's definition and show that the operations applied to every input really produce $x \cdot x$ on the stated domain.

### 7.11.6 Mastery exercise - from data to plot to manifest

Run `write_artifacts` in a new directory. Use code to demonstrate that this workflow produces CSV data, a static SVG figure, a text description, and a JSON manifest; that the manifest contains units and an accessible description; and that the hashes of the three artifacts recorded in the manifest match their actual bytes.

Hint

Use `paths = write_artifacts(directory)`, read the JSON with `json.loads`, then calculate `hashlib.sha256(paths[key].read_bytes()).hexdigest()` for `data`, `figure`, and `alt_text`.

Self-check

The following code must finish without an `AssertionError`. A temporary directory keeps the check from changing release artifacts.

```
import hashlib
import json
from pathlib import Path
import tempfile

from source.code.unit04_visualization import ALT_TEXT, write_artifacts

with tempfile.TemporaryDirectory() as tmp_u04:
    paths_u04 = write_artifacts(Path(tmp_u04))
    manifest_u04 =
 ↪  json.loads(paths_u04["manifest"].read_text(encoding="utf-8"))

    assert set(paths_u04) == {"data", "figure", "alt_text", "manifest"}
    assert manifest_u04["units"] == {
        "distance": "m",
        "time": "s",
        "uncertainty": "m",
    }
    assert manifest_u04["accessibility"]["description"] == ALT_TEXT
    for artifact_key_u04 in ("data", "figure", "alt_text"):
        digest_u04 = hashlib.sha256(
            paths_u04[artifact_key_u04].read_bytes()
        ).hexdigest()
        assert (
            manifest_u04["artifacts"][paths_u04[artifact_key_u04].name]
            == digest_u04
        )
```

Answer and solution

*Solution* 7.1.

```
import hashlib
import json
from pathlib import Path
import tempfile

from source.code.unit04_visualization import ALT_TEXT, write_artifacts

with tempfile.TemporaryDirectory() as directory_u04:
    artifacts_u04 = write_artifacts(Path(directory_u04))
    manifest_u04 = json.loads(
        artifacts_u04["manifest"].read_text(encoding="utf-8")
    )

    assert artifacts_u04["data"].suffix == ".csv"
    assert artifacts_u04["figure"].suffix == ".svg"
    assert artifacts_u04["alt_text"].read_text(encoding="utf-8").strip()
     ↪  == ALT_TEXT
    assert manifest_u04["accessibility"]["svg_metadata"] is True
    assert manifest_u04["construction"][-1] == "deterministic_save"

    for artifact_name_u04 in ("data", "figure", "alt_text"):
        byte_u04 = artifacts_u04[artifact_name_u04].read_bytes()
        hash_u04 = hashlib.sha256(byte_u04).hexdigest()
        assert (

 ↪  manifest_u04["artifacts"][artifacts_u04[artifact_name_u04].name]
            == hash_u04
        )
```

CSV preserves values and units, SVG provides static output, alternative text offers a nonvisual route, and the manifest binds the bytes of all three. Matching hashes establish byte identity, not the correctness of the model.

## 7.12 Summary

- Visualization begins with a question, not a choice of colors.
- Positions, markers, lines, axes, scales, and units must have explicit meanings.
- Truncated axes and aspect ratios can change the impression without changing the data.
- The meaning of uncertainty must be stated; error bars do not automatically carry a statistical meaning.
- Tabular data, alternative text, figures, and a hash manifest make artifacts checkable through more than one route.
- Progressive construction using `fig`, `ax`, `plot`, `scatter`, and `errorbar` makes encoding, units, legends, and saving separately testable.
- Graphs provide limited visual evidence and can reveal anomalies; general proof still requires an argument covering the entire domain.

# 8 Exact and Symbolic Computation

## 8.1 Learning objectives

After completing this unit, you will be able to:

- distinguish a symbolic expression from a value obtained by substitution;
- use `Fraction` and SymPy to preserve exact results;
- state assumptions and domains before simplifying expressions;
- check algebraic equivalence without removing domain restrictions;
- factor and solve simple equations exactly;
- explain the boundary between computer algebra results and mathematical proof;
- produce canonical symbolic records that can be compared byte for byte; and
- use local SageMath for `ZZ`, `QQ`, `RR`, `SR`, parents and coercion, polynomial rings, factorization, exact solving, and explicitly stated approximate conversions.

Local prerequisites: Units 1 and 2, polynomial operations, quadratic equations, and the concept of a function's domain. The first route uses the Python standard library and SymPy; the local SageMath 9.5 lab is a required part of the unit. No calculus or linear algebra is required.

## 8.2 An expression is not a value

The expression

$$p(x) = x^2 - 1$$

contains the free symbol $x$. It does not yet have a single numerical value. Substituting $x = 3$ produces the exact value 8, but does not turn the original expression into a number.

```
import sympy as sp

x = sp.Symbol("x")
p = x**2 - 1
(p, p.free_symbols, p.subs(x, 3))
```

**Listing 8.1**

```
(x**2 - 1, {x}, 8)
```

This distinction matters for notebooks run out of order. The name `x` in code may refer to a Python object, whereas a SymPy symbol named `x` is part of an expression tree. Display and save the tree actually used, not just the last result on screen.

For small rational numbers, the standard library is sufficient:

```
from fractions import Fraction

Fraction(1, 10) + Fraction(2, 10)
```

**Listing 8.2**

```
Fraction(3, 10)
```

`Fraction(3, 10)` and `sp.Rational(3, 10)` both preserve the exact rational value. Choose boundaries between libraries explicitly; do not convert a value to `float` merely to obtain shorter-looking output.

## 8.3 Assumptions and domains are part of the problem

Without sign information, the formula $\sqrt{x^2} = x$ is not true for every real number. If $x$ is real, the correct result is $|x|$. If $x > 0$, the result is $x$.

```
x_real = sp.Symbol("x", real=True)
x_positive = sp.Symbol("x", positive=True)

(
    sp.simplify(sp.sqrt(x_real**2)),
    sp.simplify(sp.sqrt(x_positive**2)),
)
```

The two symbols have the same printed name, but their internal assumptions differ. A symbolic computation record must therefore preserve the symbol definitions or their accompanying domain statements.

**Listing 8.3**

```
(Abs(x), x)
```

Domains must also be preserved when factors are canceled. For

$$r(x) = \frac{x^2 - 1}{x - 1},$$

SymPy obtains $x + 1$ after cancellation. Both formulas give the same value on their common domain, namely $x \neq 1$. However, they are not the same function when the original domain is included in the definition: the original formula is undefined at $x = 1$, whereas $x + 1$ has the value 2 there.

## 8.4 Structural equality and equivalence

The `==` operator on two SymPy expressions primarily compares their canonical structures. It does not universally answer the question, "Do these formulas have the same value for every valid input?" For example, the expressions $(x + 1)^2$ and $x^2 + 2x + 1$ have different structures but represent the same polynomial.

```
left = (x + 1) ** 2
right = x**2 + 2*x + 1

structurally_equal = left == right
polynomial_identity = sp.expand(left - right) == 0
(structurally_equal, polynomial_identity)
```

**Listing 8.4**

```
(False, True)
```

The checking procedure must fit the expression class. Expanding the difference is appropriate for polynomial identities. `simplify(left-right) == 0` is useful as another check, but it does not replace recording domains, root branches, assumptions, or singular points.

## 8.5 Exact factorization and solving

Symbolic computation preserves roots and factors without decimal approximation.

```
x = sp.Symbol("x")
polynomial = x**4 - 1
factored = sp.factor(polynomial)
(factored, sp.expand(factored) == polynomial)
```

**Listing 8.5**

```
((x - 1)*(x + 1)*(x**2 + 1), True)
```

The expected factorization is

$$(x-1)(x+1)(x^2+1).$$

The `expand` check tests whether the result returns to the input polynomial. For the equation $x^2 = 2$, the solution domain must be stated:

```
x = sp.Symbol("x", real=True)
solutions = sp.solveset(sp.Eq(x**2, 2), x, domain=sp.S.Reals)
solutions
```

**Listing 8.6**

$$\left\{-\sqrt{2}, \sqrt{2}\right\}$$

The exact real solution set is $\{-\sqrt{2}, \sqrt{2}\}$. Substituting each candidate into $x^2 - 2$ must give an exactly zero residual. Changing the domain may change the form or number of solutions, so do not save a list of roots without its equation and domain.

## 8.6 Symbolic results and proof

Computer algebra output can serve several different roles:

- **candidate:** `solve` proposes values that can then be checked;
- **local certificate:** re-expansion or residual substitution checks a specific relation;
- **experiment:** many examples can test a conjecture; and
- **proof:** an argument explains why a claim holds throughout the domain.

The factorization of $x^4 - 1$ can be checked directly using distributivity. However, one successful command does not prove that every output of the algorithm is correct for every input. A broader claim also depends on the algorithm's specification and the correctness of its implementation. In mathematical writing, state exactly which relation the computer checked and which general step was proved mathematically.

## 8.7 Canonical output

Readable two-dimensional output can change with terminal width, printer settings, or library versions. This unit uses `sympy.srepr` to record expression trees and JSON with sorted keys, fixed indentation, UTF-8, and LF line endings. The SymPy version is also saved because the library's canonical forms may change between releases.

Run the script from the project root:

```
python source/code/unit05_symbolic.py --output output/unit05-results.json
```

The `core_sha256` field binds the core record before the hash field is added. Running the same command in the same environment must produce identical bytes. A hash is not proof of mathematical correctness; it establishes that two byte sequences are the same.

## 8.8 Required local SageMath lab

SageMath brings many mathematical structures together in a system that uses Python syntax. In this unit, Sage is neither a web service nor an optional snippet. The lab runs locally with SageMath 9.5, frozen in WSL Ubuntu 22.04. SageCell or a remote service does not meet the requirement because its runtime, network availability, and result bytes are not under the control of the local experiment.

If that profile is not yet available, follow the local setup guide and verify the lock before the lab. Version differences must be recorded; do not describe a different profile as identical to the release environment.

From the project root in PowerShell, run:

```
wsl.exe -d Ubuntu-22.04 -- /usr/bin/sage -python
 ↪  source/code/unit05_sage_lab.py --output output/unit05-sage-results.json
wsl.exe -d Ubuntu-22.04 -- /usr/bin/sage -python tests/test_unit05_sage.py
```

The first command writes deterministic JSON to the specified relative path. The second runs all lab tests in the same runtime. Sage blocks in this reader deliberately use ordinary `python` fences rather than Quarto `{python}` blocks. The Windows build therefore does not attempt to import Sage through the ordinary Python kernel.

### 8.8.1 Parents determine where objects live

Sage asks for an object's `parent()`, not just its Python implementation type. The four core parents in this lab are:

| Name | Mathematical parent | Role |
|---|---|---|
| `ZZ` | integer ring | exact integer arithmetic |
| `QQ` | rational field | exact rational division |
| `RR` | 53-bit real field | floating-point approximation |
| `SR` | symbolic ring | symbolic expressions and equations |

```
from sage.all import QQ, RR, SR, ZZ

integer_value = ZZ(6)
rational_value = QQ(1) / 3
approximation = RR(rational_value)
symbolic_value = SR(rational_value)

print(integer_value.parent())
print(rational_value.parent())
print(approximation.parent())
print(symbolic_value.parent())
```

Printed values alone are not enough. `1/3` in `QQ` is an exact value, whereas a value in `RR` is an approximation at its parent's precision.

### 8.8.2 Coercion must be explainable

**Coercion** is the canonical conversion that Sage chooses when an operation mixes parents. For example, `ZZ(6) + QQ(1)/3` lives in `QQ` because there is a canonical map from `ZZ` to `QQ`. The lab explicitly checks the maps from `ZZ` to `QQ`, `QQ` to `RR`, and `QQ` to `SR`.

```
from sage.all import QQ, RR, SR, ZZ

assert QQ.has_coerce_map_from(ZZ)
assert RR.has_coerce_map_from(QQ)
assert SR.has_coerce_map_from(QQ)
mixed_sum = ZZ(6) + QQ(1) / 3
assert mixed_sum.parent() is QQ
```

The availability of a coercion does not mean that every change of representation is lossless. Mapping `QQ` to `RR` turns an exact rational value into an approximation. Scientific code should therefore make conversion points explicit.

### 8.8.3 Polynomial rings and exact factorization

A polynomial's parent records its coefficient ring. `PolynomialRing(QQ, "x")` specifies univariate polynomials with rational coefficients. This is more specific than a symbolic expression that happens to look like a polynomial.

```
from sage.all import PolynomialRing, QQ

R = PolynomialRing(QQ, "x")
x = R.gen()
p = x**4 - 1
factorization = p.factor()

assert p.parent() is R
assert R.base_ring() is QQ
assert factorization.prod() == p
print(factorization)
```

The exact factorization is $(x-1)(x+1)(x^2+1)$. The `prod()` check multiplies the factors within the same parent and compares the result with the original polynomial.

### 8.8.4 Solving in SR and exact residuals

For symbolic equations, the domain and parent must still be recorded. The lab uses `SR`, requests solutions as dictionaries, then substitutes each candidate into the residual.

```
from sage.all import SR, solve

y = SR.var("y")
solutions = solve(y**2 == 2, y, solution_dict=True)
roots = [solution[y] for solution in solutions]
assert all((root**2 - 2).simplify_full() == 0 for root in roots)
```

Zero residuals check the two returned candidates. They do not by themselves prove that no other solutions exist on a different domain.

### 8.8.5 Approximate conversions must be explicit

Do not convert exact objects to decimals just to shorten their display. Preserve the original object and specify the target parent:

```
from sage.all import QQ, RR

exact_value = QQ(1) / 3
approximation = RR(exact_value)
assert exact_value.parent() is QQ
assert approximation.parent() is RR
print(exact_value, approximation, RR.precision())
```

`RR(exact_value)` is a representation decision that can be located in the source. The lab record preserves the exact value, conversion expression, approximate value, parent, and 53-bit precision.

### 8.8.6 An auditable Sage record

`unit05-sage-results.json` records SageMath 9.5, the local-execution requirement, parents, coercion maps, polynomial rings, factors, solutions, residuals, conversions to `RR`, and `core_sha256`. Every object is converted to a specified JSON-compatible form before serialization. Two runs using the same runtime must produce the same bytes.

The script rejects versions other than 9.5. This restriction does not claim that other versions are wrong; it binds the execution evidence to the runtime actually tested. SageMath results

are still no substitute for general proof. Parents and residuals explain what the computation checked; mathematical arguments explain why a claim holds on its domain.

## 8.9 Exercises

### 8.9.1 Exercise 1 - expressions and values

Construct the expression $q(t) = t^3 - 2t$ in SymPy. Identify its free symbol, then evaluate it exactly at $t = 1/2$.

> **Hint**
>
> Use `Symbol`, `Rational`, the `free_symbols` attribute, and `subs`.

> **Answer and solution**
>
> The code `t=sp.Symbol("t"); q=t**3-2*t` gives the free-symbol set `{t}`. The substitution `q.subs(t, sp.Rational(1,2))` gives $1/8 - 1 = -7/8$. The expression `q` still contains `t`; `-7/8` is its value at one input, not a replacement for the original expression.

### 8.9.2 Exercise 2 - a missing assumption

A reader simplifies $\sqrt{z^2}$ to $z$ for all real numbers. Give a counterexample and state the correct general result.

> **Hint**
>
> Try a negative real number.

> **Answer and solution**
>
> Take $z = -3$. Then $\sqrt{z^2} = \sqrt{9} = 3$, whereas $z = -3$, so the simplification fails. For real $z$, the correct general form is $\sqrt{z^2} = |z|$. The form $z$ is valid with the additional assumption $z \geq 0$; a specifically positive SymPy symbol can be declared using `positive=True`.

### 8.9.3 Exercise 3 - equivalence and domains

Compare $(u^2 - 4)/(u - 2)$ with $u + 2$. State the set on which they have equal values and explain why the two functions do not automatically have the same domain.

Hint

Factor the numerator and examine the denominator before canceling factors.

Answer and solution

Since $u^2 - 4 = (u - 2)(u + 2)$, both formulas give the same value for every $u \neq 2$. The fractional formula is undefined at $u = 2$, while $u + 2$ has the value 4. They are therefore equivalent on their common domain, but are not functions with the same domain unless the restriction $u \neq 2$ is retained.

### 8.9.4 Exercise 4 - factors, roots, and checks

Factor $v^3 - v$, find all its real roots exactly, then describe two checks that can be made without trusting the displayed output.

Hint

Take out a factor of `v`, then factor the difference of squares.

Answer and solution

$v^3 - v = v(v^2 - 1) = v(v - 1)(v + 1)$, so its real roots are $(-1, 0, 1)$. First check: expanding `v*(v-1)*(v+1)` must recover `v**3-v`. Second check: substituting each root into the polynomial must give an exactly zero residual. These checks support clearly defined local claims.

### 8.9.5 Exercise 5 - stable symbolic artifacts

Why is saving only a screenshot of `pretty print` output insufficient for a reproducible symbolic experiment? Name at least four elements of a better record.

Hint

Consider expression trees, assumptions, domains, versions, and output bytes.

Answer and solution

A screenshot may lose structure, changes easily with the layout, and cannot be evaluated again. A better record includes (1) the canonical expression tree, (2) symbol assumptions, (3) the problem domain, (4) the SymPy or SageMath version, (5) inputs and methods, and (6) JSON or a text format with fixed ordering and line-ending rules. Hashes can then compare the bytes, while residuals or expansion check specific mathematical claims.

### 8.9.6 Sage exercise 1 - parents, coercion, and factors

In local SageMath 9.5, construct `ZZ(5)`, `QQ(2)/3`, and their sum. Record each object's parent and explain the coercion that occurs. Then construct the ring `QQ[z]`, factor $z^4 - 1$, and check that multiplying the factors recovers the original polynomial.

Hint

Use `parent()`, `QQ.has_coerce_map_from(ZZ)`, `PolynomialRing(QQ, "z")`, and the `factor()` and `prod()` methods.

Self-check

Run this block only through `/usr/bin/sage -python`, not the Windows Quarto kernel.

```
from sage.all import PolynomialRing, QQ, ZZ

a = ZZ(5)
b = QQ(2) / 3
c = a + b
Rz = PolynomialRing(QQ, "z")
z = Rz.gen()
pz = z**4 - 1
fz = pz.factor()

assert a.parent() is ZZ
assert b.parent() is QQ
assert c.parent() is QQ
assert QQ.has_coerce_map_from(ZZ)
assert Rz.base_ring() is QQ
assert fz.prod() == pz
```

Answer and solution

```
from sage.all import PolynomialRing, QQ, ZZ

integer_value = ZZ(5)
rational_value = QQ(2) / 3
mixed_sum = integer_value + rational_value

assert str(integer_value.parent()) == "Integer Ring"
assert str(rational_value.parent()) == "Rational Field"
assert mixed_sum == QQ(17) / 3
assert mixed_sum.parent() is QQ

ring = PolynomialRing(QQ, "z")
z = ring.gen()
polynomial = z**4 - 1
factorization = polynomial.factor()
assert factorization.prod() == polynomial
assert {str(factor) for factor, _ in factorization} == {
    "z - 1",
    "z + 1",
    "z^2 + 1",
}
```

Sage uses the canonical map from `ZZ` to `QQ`, so the sum lives in `QQ` and remains exact. The polynomial lives in `QQ[z]`; multiplying its factors within that same parent provides a local certificate for the factorization.

### 8.9.7 Sage exercise 2 - exact solutions and approximations

Solve $w^2 = 5$ in `SR`, check each solution's residual exactly, then explicitly convert the positive root to `RR`. Save the exact value, original parent, approximate value, target parent, and precision. Explain what the residuals establish and what they do not.

Hint

Use `solve(..., solution_dict=True)`, `simplify_full()`, and `RR(root)`. Do not replace the `SR` object before checking the exact residual.

❗ Self-check

```
from sage.all import RR, SR, solve

w = SR.var("w")
solutions = solve(w**2 == 5, w, solution_dict=True)
roots = [solution[w] for solution in solutions]

assert {str(root) for root in roots} == {"-sqrt(5)", "sqrt(5)"}
assert all((root**2 - 5).simplify_full() == 0 for root in roots)

positive_root = next(root for root in roots if root > 0)
approximation = RR(positive_root)
assert positive_root.parent() is SR
assert approximation.parent() is RR
assert RR.precision() == 53
```

Answer and solution

```
from sage.all import RR, SR, solve

w = SR.var("w")
solutions = solve(w**2 == 5, w, solution_dict=True)
exact_roots = sorted((solution[w] for solution in solutions), key=str)
zero_residuals = [
    bool((root**2 - 5).simplify_full() == 0)
    for root in exact_roots
]

positive_root = next(root for root in exact_roots if root > 0)
approximate_value = RR(positive_root)
record = {
    "exact": str(positive_root),
    "exact_parent": str(positive_root.parent()),
    "approximate": str(approximate_value),
    "approximate_parent": str(approximate_value.parent()),
    "precision_bits": RR.precision(),
}

assert zero_residuals == [True, True]
assert record["exact"] == "sqrt(5)"
assert record["exact_parent"] == "Symbolic Ring"
assert record["approximate_parent"].startswith("Real Field")
assert record["precision_bits"] == 53
```

The exact residuals check that the two candidates satisfy the equation. The conversion `RR(positive_root)` produces a 53-bit approximation without erasing the record of the exact object. This check does not prove that the `solve` algorithm is complete for every equation or every domain.

## 8.10 Summary

- Symbolic expressions describe structure; substitution produces values at specific inputs.
- Exact arithmetic preserves rational and algebraic relationships without decimal rounding.
- Assumptions and domains determine which simplifications are valid.
- Structural equality, equivalence on a common domain, and equality of functions are three different questions.

- Factorization, re-expansion, exact solving, and residuals complement one another as local checks.
- Computer algebra output is execution evidence whose claims must be bounded, not an automatic substitute for general proof.
- Canonical representations, library versions, and hashes make artifacts comparable.
- The required local SageMath 9.5 lab uses `ZZ`, `QQ`, `RR`, `SR`, parents and coercion, polynomial rings, factorization, solving, residuals, and explicit approximate conversion; remote services are not a substitute.

# 9 Floating-Point Numbers, Error, and Stability

## 9.1 Learning objectives

After completing this unit, you will be able to:

- explain the basic model of binary floating-point numbers;
- interpret `ulp`, machine epsilon, and the overflow and underflow limits;
- recognize cancellation that destroys significant digits;
- distinguish forward error from backward error;
- distinguish problem conditioning from algorithmic stability;
- choose more stable reformulations and summation methods;
- compare naive evaluation with `scipy.special` on a stated domain;
- set tolerances as an error budget before inspecting the results; and
- limit experimental evidence to the cases and environments actually run.

Local prerequisites: Units 1, 2, and 3; integer powers, square roots, absolute values, rational functions, and relative error at A30 level. No calculus is required.

## 9.2 A finite binary model

In this project's environment, `float` uses a 64-bit binary format. In simplified form, a finite normal number is stored as

$$(-1)^s \, (1.b_1 b_2 \ldots b_{52})_2 \, 2^e.$$

There is one sign bit, a bounded exponent, and 53 bits of significand precision when the leading one before the binary point is counted. Because there are only finitely many bit patterns, only finitely many real numbers can be stored. Operation results are usually rounded to the nearest representable number.

The `hex()` method reveals the binary value of a `float` without losing information:

```
x = 0.1
(x, x.hex(), x.as_integer_ratio())
```

**Listing 9.1**

```
(0.1, '0x1.999999999999ap-4', (3602879701896397, 36028797018963968))
```

The fraction returned by `as_integer_ratio()` is the rational value **actually stored**. It is close to 1/10, but it is not 1/10.

## 9.3 `ulp`, epsilon, and changing spacing

One *unit in the last place* (`ulp`) is the local spacing between adjacent representable numbers. This spacing changes with the magnitude of the number.

```
import math
import sys

(
    sys.float_info.epsilon,
    math.ulp(1.0),
    math.ulp(float(2**53)),
)
```

**Listing 9.2**

```
(2.220446049250313e-16, 2.220446049250313e-16, 2.0)
```

Around 1, `ulp(1.0)` equals `sys.float_info.epsilon`: the difference between 1 and the next larger `float`. Some books use "machine epsilon" for this number. Others use the *unit roundoff* $u = \varepsilon/2$ in the round-to-nearest model. Because terminology varies, state the definition you use.

Around $2^{53}$, one `ulp` is 2. Consequently:

```
large_value = float(2**53)
(large_value + 1.0 == large_value, large_value + 2.0 == large_value + 2)
```

This is not a mistake in Python's integer addition. The issue is that the result is being forced back onto the `float64` grid, whose spacing at this scale is 2.

**Listing 9.3**

```
(True, True)
```

## 9.4 Overflow, subnormals, and underflow

Precision and range are two different limits. `sys.float_info.max` is the largest finite `float`. Multiplying it by 2 produces positive infinity in the tested environment. At the small end, `sys.float_info.min` is the smallest positive **normal** number, not the smallest positive number.

Subnormal numbers fill part of the gap toward zero with reduced precision. `math.ulp(0.0)` gives the smallest positive subnormal. Dividing it by two gives zero because there is no smaller positive representable number.

```
largest_float = sys.float_info.max
smallest_subnormal = math.ulp(0.0)
(
    largest_float * 2.0,
    sys.float_info.min,
    smallest_subnormal,
    smallest_subnormal / 2.0,
)
```

**Listing 9.4**

```
(inf, 2.2250738585072014e-308, 5e-324, 0.0)
```

Check values with `math.isfinite` before passing an infinite result to the next stage, and check separately for zero caused by underflow. Checks after the event do not replace an analysis of scale before computation.

## 9.5 Cancellation and stable reformulation

Subtracting two nearly equal numbers can eliminate most of their significant digits. For large $x$, consider

$$d(x) = \sqrt{x+1} - \sqrt{x}.$$

At $x = 10^{16}$, the addition `x + 1.0` already rounds back to `x`. Direct subtraction then produces zero. Before computing, rationalize algebraically:

$$d(x) = \frac{(x+1) - x}{\sqrt{x+1} + \sqrt{x}} = \frac{1}{\sqrt{x+1} + \sqrt{x}}.$$

```
x = 1e16
direct = math.sqrt(x + 1.0) - math.sqrt(x)
stable = 1.0 / (math.sqrt(x + 1.0) + math.sqrt(x))
(x + 1.0 == x, direct, stable)
```

**Listing 9.5**

```
(True, 0.0, 5e-09)
```

The second form avoids subtracting two nearly equal square roots. This reformulation follows from an algebraic identity over the real numbers; the Unit 6 program compares the two implementations with a high-precision `Decimal` reference. A high-precision reference remains a checking tool, not a proof in its own right.

## 9.6 SciPy: stable `exprel` near zero

SciPy provides a specialized implementation of the function

$$E(x) = \frac{e^x - 1}{x} \quad (x \neq 0), \qquad E(0) = 1.$$

Its mathematical domain is all real numbers once the value at zero is defined by this continuous extension. This unit's detailed lab uses the narrower computational domain $0 < |x| \leq 10^{-4}$ because the reported relative backward error divides by $|x|$. The `Decimal` oracle does not compute `exp(x) - 1`: it sums the power series for $E$ and $E'$ without cancellation. Its precision is at least 80 digits and increases with the exponent of $x$, retaining 50 guard digits beyond the order of $x$. The first correction $x/2$ therefore remains resolved even for the smallest binary64 subnormal. The upper bound and the exclusion of $x = 0$ delimit the lab's error report, not the mathematical domain of `exprel`.

Direct evaluation computes `exp(x)`, subtracts 1, and divides by `x`. When `x` is small, `exp(x)` and 1 are nearly equal, so subtraction can discard significant digits. For more extreme values, `exp(x)` may round to exactly 1, making the naive numerator zero.

```python
import math
import scipy
from scipy import special

x = 1e-8
naive = (math.exp(x) - 1.0) / x
stable = float(special.exprel(x))

extreme_x = 1e-16
extreme_naive = (math.exp(extreme_x) - 1.0) / extreme_x
extreme_stable = float(special.exprel(extreme_x))

(scipy.__version__, naive, stable, extreme_naive, extreme_stable)
```

**Listing 9.6**

```
('1.15.2', 0.999999993922529, 1.000000005, 0.0, 1.0)
```

This difference does not automatically mean that the mathematical problem is ill-conditioned. A measure of local relative conditioning is

$$\kappa_E(x) = \left| \frac{xE'(x)}{E(x)} \right|.$$

For $x \neq 0$,

$$E'(x) = \frac{(x-1)e^x + 1}{x^2}.$$

Near zero, $E(x)$ approaches 1, $E'(x)$ approaches $1/2$, and $\kappa_E(x)$ is approximately $|x|/2$. This derivative formula is supplied as a diagnostic tool; deriving it is not an A30 prerequisite. At $x = 10^{-8}$, the condition number is about $5 \times 10^{-9}$, so small relative perturbations in the input should not be amplified. The large error in naive evaluation is therefore algorithmic instability, not inherent sensitivity of the function.

The companion script compares both results with an adaptive-precision `Decimal` series reference on the stated lab domain. The reported forward error is

$$e_f = |\widehat{E} - E(x)|.$$

To interpret the result in terms of backward error, the script uses the first-order estimate

$$|\Delta x| \approx \frac{e_f}{|E'(x)|}.$$

This asks approximately how much the input would have to change for the computed output to be the exact output for that changed input. This estimate is **not** an exact inverse solution, and the JSON labels it accordingly. At $x = 10^{-8}$, the naive evaluation's estimated relative backward error is greater than 1, whereas SciPy's estimate is far below $10^{-6}$. SciPy's relative forward error is also below $10^{-15}$ in the tested run.

These results provide evidence for the recorded values, SciPy version, and environment. They give a concrete example of a specialized implementation being more accurate than the naive formula near zero. They do not prove that `scipy.special.exprel` is stable for every real number or every backend. A general claim still requires analysis of the algorithm, the rounding model, and range handling.

## 9.7 Forward error and backward error

Suppose a problem asks for $y = f(x)$ and a program returns $\hat{y}$.

- **Forward error** measures the distance from the desired answer: $|\hat{y} - f(x)|$.
- **Backward error** asks how far the input must change for the output to be an exact answer: find $\Delta x$ such that $\hat{y} = f(x + \Delta x)$.

For the problem $y = \sqrt{a}$, squaring the output gives an input for which that output would be exact. A readily checkable measure of relative backward error is therefore

$$\frac{|\hat{y}^2 - a|}{|a|}.$$

```
y_hat = math.sqrt(2.0)
residual = y_hat * y_hat - 2.0
(y_hat, residual)
```

**Listing 9.7**

```
(1.4142135623730951, 4.440892098500626e-16)
```

A small backward error means that the result is an exact solution to a slightly perturbed problem. Whether its forward error is also small depends on the conditioning of the problem.

## 9.8 Conditioning is not stability

**Conditioning** is a property of the mathematical question: how much does the output change when the input changes slightly? **Algorithmic stability** is a property of the method of computation: does internal rounding introduce errors far beyond the problem's sensitivity?

For

$$f(x) = \frac{1}{1-x},$$

inputs close to 1 are highly sensitive. At $x = 0.99999999$, a perturbation of about $10^{-12}$ can be amplified by roughly $10^8$ in the relative output error. This problem is ill-conditioned near the singular point $x = 1$. Changing programming languages cannot remove that mathematical sensitivity.

By contrast, the problem $d(x) = \sqrt{x+1} - \sqrt{x}$ for large $x$ does not require a zero answer. Direct subtraction loses information, whereas the rationalized form retains it. Here, the main difference is the stability of the evaluation method.

Four possibilities must be distinguished:

| Problem | Algorithm | Practical meaning |
|---|---|---|
| well-conditioned | stable | small errors are generally expected |
| well-conditioned | unstable | the algorithm wastes information |
| ill-conditioned | stable | results can still be sensitive to the data |
| ill-conditioned | unstable | problem sensitivity and algorithmic error compound |

## 9.9 Summation and order of operations

Floating-point addition is not associative. Over the real numbers,

$$10^{16} + 1 - 10^{16} = 1.$$

However, a naive algorithm that sums from left to right loses the 1 before the final subtraction.

```
values = [1e16, 1.0, -1e16]

def naive_sum(data):
    total = 0.0
    for value in data:
        total += value
    return total

(
    naive_sum(values),
    naive_sum([1e16, -1e16, 1.0]),
    math.fsum(values),
)
```

**Listing 9.8**

```
(0.0, 1.0, 1.0)
```

`math.fsum` tracks the small lost parts more carefully and returns 1 for this example. The built-in `sum` may use an improved strategy in some Python versions; the script therefore records its result too, but does not use it as the definition of the naive algorithm. `math.fsum` is generally more accurate for data of mixed magnitudes, but this is not a guarantee that every summation problem is error-free. Record the method, version, data order, and magnitudes.

## 9.10 Tolerances as an error budget

The closeness test used in this unit has the form

$$|\hat{y} - y_{ref}| \leq A + R|y_{ref}|,$$

where $A$ is the absolute budget and $R$ is the relative budget. The absolute part matters near zero; the relative part scales with the reference.

The budget must come from the problem before the result is inspected. Its sources may include measurement precision, model approximation, discretization, iteration stopping criteria, and rounding. Adding the bounds for individual components gives a conservative bound when no sharper analysis is available.

`isclose` or a similar function only checks the rule it is given. It does not choose a tolerance, repair an algorithm, or prove that the reference is correct. Save the values of $A$ and $R$, the observed error, and the reasons for choosing them.

## 9.11 Deterministic records and limits of evidence

Run from the project root:

```
python source/code/unit06_floating.py --output output/unit06-results.json
```

The script records the Python and SciPy versions, the binary64 profile, cancellation, the `scipy.special.exprel` comparison, forward and backward errors, conditioning, summation, and error-budget decisions. Numbers that must be distinguished bit by bit are stored using `repr`, `hex`, or `Decimal` text. The JSON uses UTF-8, sorted keys, fixed indentation, and LF line endings. The `core_sha256` field binds the record before the hash is added.

The experiment establishes the behavior of the cases actually run in the recorded environment. It can supply a counterexample to a claim such as "summation order never matters." It does not prove a theorem about all binary64 operations.

General statements require a model, for example: each normal operation is rounded to the nearest representable value with relative error at most approximately $u$, provided no overflow or underflow occurs. Error bounds can then be derived step by step from that model. Tests check whether the implementation and environment follow the recorded assumptions.

## 9.12 Exercises

### 9.12.1 Exercise 1 - `ulp` and the lost increment

Around $2^{53}$, `ulp` is 2. Explain why `float(2**53) + 1.0 == float(2**53)`, but adding 2 can produce the next number.

> **Hint**
>
> Imagine a grid of representable numbers whose points are spaced 2 apart.

> **Answer and solution**
>
> At this scale, adjacent finite `float` values are $2^{53}, 2^{53}+2, 2^{53}+4, \ldots$. The real value $2^{53}+1$ lies exactly between two grid points. The ties-to-even rule rounds it to the value with an even significand, namely $2^{53}$. The value $2^{53}+2$ is itself representable, so the increment of 2 is not lost.

### 9.12.2 Exercise 2 - eliminating cancellation

Derive a stable form of $\sqrt{x+4} - \sqrt{x}$ and explain why it is better for large $x$.

**Hint**

Multiply the numerator and denominator by the sum of the two square roots.

**Answer and solution**

$$\sqrt{x+4} - \sqrt{x} = \frac{(x+4) - x}{\sqrt{x+4} + \sqrt{x}} = \frac{4}{\sqrt{x+4} + \sqrt{x}}.$$

The direct form subtracts two large, nearly equal numbers, so significant digits can be lost. The rationalized form divides 4 by a positive sum, avoiding that subtraction. The real domain considered here is $x \geq 0$.

### 9.12.3 Exercise 3 - forward and backward error

For the approximation $\hat{y} = 1.414$ to $\sqrt{2}$, estimate the absolute forward error and the relative backward error based on the equation $y^2 = 2$.

**Hint**

Use $\sqrt{2} \approx 1.41421356$ and compute $1.414^2$.

**Answer and solution**

The absolute forward error is approximately $|1.414 - 1.41421356| = 0.00021356$. Since $1.414^2 = 1.999396$, the relative backward error with respect to the input 2 is

$$\frac{|1.999396 - 2|}{2} = 0.000302.$$

The first number compares the output with the desired square root; the second asks what relative change in the input would make that output an exact square root.

### 9.12.4 Exercise 4 - conditioning or algorithm?

Classify these two observations: (a) the value of $1/(1-x)$ changes greatly when $x$ near 1 is perturbed slightly; (b) subtracting two nearly equal square roots gives zero, but the rationalized form gives an accurate nonzero value.

Hint

Ask whether the sensitivity already belongs to the mathematical function or arises from the sequence of operations.

Answer and solution

Observation (a) is primarily ill-conditioning: the function has a singular point at 1 and is inherently sensitive nearby. Even a stable algorithm cannot remove that sensitivity to the data. Observation (b) is primarily instability of direct evaluation: the same problem can be computed much more accurately after algebraic reformulation. In practice, both can occur together.

### 9.12.5 Exercise 5 - summation, tolerance, and limits on claims

For the data `[1e16, 1.0, -1e16]`, the mathematical target is 1. Compare the results of `naive_sum` above and `math.fsum`. With an absolute budget of 0.5 and a relative budget of $10^{-12}$, determine which passes. Does one success prove that `math.fsum` is always exact?

Hint

Use the bound $A + R|y_{ref}|$ with $y_{ref} = 1$.

Answer and solution

`naive_sum` returns 0 because the 1 is lost when added to $10^{16}$. `math.fsum` returns 1 in this case. The allowed bound is $0.5 + 10^{-12}$; the naive algorithm's error is 1, so it fails, whereas the error of `math.fsum` is 0, so it passes. This success establishes only the result of this case in the tested environment; it does not prove that `math.fsum` is always exact for every list.

### 9.12.6 SciPy exercise - stability of `exprel`

Run `scipy_exprel_report(1e-8)` from the companion code. Record the mathematical and lab domains, the relative condition number, both implementations' relative forward errors, and their estimated relative backward errors. Determine whether the poor naive result primarily indicates an ill-conditioned problem or an unstable algorithm. Repeat the value comparison at `x=1e-16` and limit your claims to the evidence actually obtained.

**Hint**

If the condition number is far below 1 but the naive forward error is large, amplification of input perturbations is not the problem. Also check whether `math.exp(1e-16) == 1.0`.

**Executable check**

From the project root, run:

```
python tests/test_unit06.py
↪ Unit06FloatingTests.test_scipy_special_exprel_is_stable_near_zero -v
```

The check covers the lab domain, SciPy version and function, forward error, estimated backward error, conditioning, extreme cancellation, and limits of the evidence. A separate regression test also runs the smallest binary64 subnormal and checks that the oracle still captures the first correction $x/2$, rather than silently rounding it to zero.

Answer and solution

**Listing 9.9**

```
import runpy

unit06 = runpy.run_path("source/code/unit06_floating.py")
report = unit06["scipy_exprel_report"](1e-8)
print("domain:", report["mathematical_domain"])
print("lab:", report["laboratory_domain"])
print("condition:", report["conditioning"]["relative_condition_number"])
print("naive forward:", report["forward_error"]["naive_relative"])
print("SciPy forward:", report["forward_error"]["scipy_relative"])
print("naive backward:",
 ↪  report["backward_error"]["naive_relative_estimate"])
print("SciPy backward:",
 ↪  report["backward_error"]["scipy_relative_estimate"])
```

```
domain: all real x
lab: 0 < |x| <= 1e-4 for binary64 inputs
condition: 5.000000008333333343795E-9
naive forward: 1.1077470932201461685 1E-8
SciPy forward: 4.7054021432286778908 1E-17
naive backward: 2.2154941827478019860 9E+0
SciPy backward: 9.4108042707726818001 0E-9
```

---

The mathematical domain is all real numbers with $E(0) = 1$; the lab is restricted to $0 < |x| \leq 10^{-4}$ for binary64 inputs. At $x = 10^{-8}$, the condition number is about $5 \times 10^{-9}$, but the naive relative forward error exceeds $10^{-9}$ and its estimated relative backward error exceeds 1. By contrast, SciPy's relative forward error is below $10^{-15}$ and its estimated relative backward error is below $10^{-6}$. Since the problem is well-conditioned near zero, this difference reveals cancellation in the naive algorithm.
At $x = 10^{-16}$, `math.exp(x)` rounds to 1, so the naive formula gives 0, whereas `scipy.special.exprel` gives 1 in this run. These tests do not prove behavior for all inputs or backends; they check only the recorded cases and environment.

## 9.13 Summary

- `float` values form a finite binary grid; `ulp` changes with scale.
- Epsilon, *unit roundoff*, the normal range, and the subnormal range answer different questions.
- Cancellation can be avoided through algebraic reformulation before evaluation.
- `scipy.special.exprel` provides an executable example of a specialized function that avoids naive cancellation near zero on the stated lab domain.
- Forward error measures the answer; backward error measures the change in the problem that makes that answer exact.
- Conditioning belongs to the problem, whereas stability belongs to the algorithm.
- Summation methods and operation order affect error.
- A tolerance is a budget justified in advance, not a number adjusted to make a test pass.
- Experiments provide bounded evidence; general claims require a rounding model and a mathematical argument.

# 10 Designing Experiments That Can Be Checked

## 10.1 Learning objectives

After completing this unit, you will be able to:

- turn a broad question into a testable hypothesis and outcome measure;
- distinguish the variable being changed, the outcome being measured, and the conditions being controlled;
- set the parameter grid, number of replications, and stopping rule before seeing the results;
- explain what a pseudorandom seed can and cannot reproduce;
- separate development data from held-out test data (*holdout*);
- report effect sizes and negative results without changing the target afterward; and
- limit experimental conclusions so that they are not treated as general proofs.

Local prerequisites: Units 1–4, percentages, means, comparisons of fractions, and simple step functions at A30 level. No calculus or inferential statistics is required.

## 10.2 Questions first, results second

The Unit 7 experiment studies a simple decision rule. Each case has an integer score from 0 to 9. For a threshold $k$, the program predicts a positive outcome if

$$\text{score} \geq k.$$

We choose one threshold from the grid $\{3, 4, 5, 6, 7\}$. The question fixed before running the experiment is:

> Does the threshold selected on development data improve accuracy on holdout data by at least 0.10 compared with the reference threshold of 6?

The value 0.10 means ten percentage points, not "ten percent of the previous accuracy." The initial hypothesis answers "yes." The experiment may then support or fail to support that answer; a hypothesis is not a promise about the result.

This question states three things that are often missing:

1. the comparison rule: a threshold of 6;
2. the outcome measure: the proportion of correct predictions; and
3. the minimum effect considered meaningful: 0.10.

Without all three, it is easy to choose a story after seeing the numbers.

## 10.3 Variables, controls, and confounders

In this experiment:

| Role | Content |
|---|---|
| variable being changed | decision threshold $k$ |
| outcome being measured | accuracy: the number of correct predictions divided by the number of cases |
| controlled conditions | case generator, sample size, replications, and seed schedule |
| random factors | the synthetic score and outcome for each case |

A **confounder** is another factor that changes along with the main variable, making the cause of a difference unclear. If threshold 3 is tested on easy cases while threshold 7 is tested on difficult cases, the difference in accuracy may come from the cases rather than the threshold.

The script prevents this confounding through a paired design: within a replication, every threshold is evaluated on exactly the same cases. Only the threshold changes. This design does not remove every possible error; the case-generation rule itself may still be poor. It does, however, isolate the comparison being made.

## 10.4 Parameter grid and selection rule

The parameter grid is fixed as

$$K = (3, 4, 5, 6, 7).$$

The program does not add new thresholds after seeing the results. Every candidate completes 20 development replications, each containing 200 cases. The threshold with the largest total number of correct predictions is chosen. If two thresholds tie, the smaller threshold wins. This tie-breaking rule may seem trivial, but without a fixed rule, two people running the experiment may select different results.

Grid search compares only these five candidates. If threshold 5 wins, a valid conclusion is "5 is best on this grid and these development data according to the specified measure," not "5 is the best possible threshold for every population."

A very dense grid is not a free benefit, either. The more choices tried on the same data, the greater the chance of finding one that fits peculiarities of the development data by chance. This is one reason for using a holdout.

## 10.5 Seeds and nondeterminism

`random.Random(seed)` generates a **pseudorandom** sequence. The same seed in the same environment produces the same sequence, allowing a failure to be rerun. This unit uses base seed 1729 for development and 271828 for the holdout. Replication $i$ uses `base_seed + i`.

A seed does not turn an experiment into a proof, and it cannot capture every source of real-world nondeterminism. Examples include:

- physical sensor noise;
- changes in temperature or voltage;
- the order of parallel processes;
- data packets arriving over a network; and
- human actions during measurement.

For these sources, record the time, equipment, calibration, treatment order, environment, and raw data as relevant. Independent replication is still needed. Writing `seed=42` does not reproduce a laboratory.

## 10.6 Replication and stopping rules

A single run may favor one threshold by chance. Replication shows whether the result persists across several samples. The unit's plan specifies:

- development: exactly $20 \times 200 = 4{,}000$ cases; and
- holdout: exactly $12 \times 250 = 3{,}000$ cases.

The word “exactly” is part of the method. The program does not stop when the results first look good. If we keep inspecting results and stop at a pleasing replication, the data-collection rule depends on the outcome; the final summary is then more optimistic than the original plan warrants.

A stopping rule may specify a fixed number of cases, a fixed time limit, or another criterion. What matters is that it is written before results are inspected and reported together with any deviations.

## 10.7 Development data and holdout data

Development data are used to compare the grid and select a threshold. Holdout data are generated with a separate seed schedule, do not influence the selection, and are inspected once after the threshold has been fixed.

A simple analogy is practice and an examination. If examination answers are used to change how you answer, and the score on that same examination is then reported as the final assessment, the examination has become practice. Likewise, if holdout results are inspected repeatedly to change the grid, features, or rules, the holdout is no longer an untouched assessment. We need a new holdout, or we must honestly describe that stage as further development.

This separation does not make the results applicable to every real-world setting. The holdout data still come from a particular case generator. The separation merely reduces one kind of overfitting to the development data.

## 10.8 Effect size, not visual impression

In the reference record, threshold 5 is selected. On the holdout, threshold 5 is correct on 2,355 of 3,000 cases, whereas reference threshold 6 is correct on 2,204 cases. The effect size is

$$\frac{2355}{3000} - \frac{2204}{3000} = \frac{151}{3000} \approx 0.0503.$$

The improvement is therefore about 5.03 percentage points. A graph with a truncated axis can make that difference look large, but its numerical effect size does not change. Because the specified minimum is 10 percentage points, the initial hypothesis is **not supported**.

“Not supported” does not mean “the two rules are certainly identical.” The result shows a positive observed effect that is smaller than the target set in advance. This unit does not calculate a confidence interval or a p-value; do not attach statistical interpretations that have not been calculated.

## 10.9 Negative results are still results

After seeing 0.0503, we must not lower the effect threshold from 0.10 to 0.05 and claim that the original plan succeeded. Nor may we delete losing thresholds or display only favorable replications.

Negative results are useful because they:

- constrain plausible claims;
- prevent others from retracing a dead end without knowing its outcome;
- may reveal that a practical target is too ambitious; and
- help design the next experiment without rewriting the history of this one.

A new experiment may have a new hypothesis. What is dishonest is rewriting the old plan as though the new target had been specified from the start.

## 10.10 The canonical Unit 7 record

Run from the project root:

```
python source/code/unit07_experiment_design.py --output
 ↪  output/unit07-results.json
```

The JSON stores the question, hypothesis, grid, variables, controls, selection rule, stopping rule, seed schedule, each replication summary, development results, holdout results, exact effect size, negative result, and limits of the evidence. Fractions are stored as numerator and denominator and as decimals to four places. Keys are sorted, indentation and line endings are fixed, and `core_sha256` binds the record before the hash field is added.

In the same Python environment, the same command produces the same bytes. The Python version is also recorded because the details of the pseudorandom generator form part of the environment that must be fixed.

## 10.11 Experiments and proof

This experiment establishes something very limited about execution: for the recorded seeds and code, the numbers of correct predictions can be recalculated. It provides empirical evidence about five thresholds on finite synthetic samples.

It does not prove that threshold 5 is best:

- outside the grid from 3 to 7;
- for every sequence of data;
- for other case generators; or
- for real phenomena not modeled by the generator.

A general proof requires a precise mathematical claim. For example, if the score distribution and outcome probabilities are defined exactly, one can calculate the expected accuracy of each threshold by summing probabilities over the ten scores. That calculation is an argument about the defined model, not a guarantee that the model represents the real world.

## 10.12 Exercises

### 10.12.1 Exercise 1 - variables and confounders

An experiment tests threshold 4 on the first 100 cases and threshold 6 on the next 100 cases. Identify the main variable, the measured outcome, and a possible confounder. Suggest one improvement.

> **Hint**
>
> Ask whether the two thresholds encounter equally difficult cases.

> **Answer and solution**
>
> The main variable is the threshold, and the outcome may be the proportion of correct predictions. The group of cases is a confounder because the first and second sequences may differ in difficulty. The most direct improvement is to evaluate both thresholds on the same 200 cases. If that is not possible, randomize the assignment order and replicate according to rules fixed in advance.

### 10.12.2 Exercise 2 - a seed is not a time machine

Explain why saving a seed is sufficient to rerun the Unit 7 simulation but insufficient to repeat a temperature measurement in a laboratory.

Hint

Distinguish a pseudorandom sequence from physical disturbances and the state of the equipment.

Answer and solution

The simulation calculates the entire sequence from an algorithm and a seed, so the same pair can generate the same cases in a fixed environment. Laboratory temperature is influenced by sensors, calibration, time, location, and physical disturbances that do not come from `random.Random`. A seed must be supplemented with raw data, equipment state, time, procedure, and independent replications.

### 10.12.3 Exercise 3 - selection and holdout

On development data, thresholds 4, 5, and 6 make 760, 810, and 790 correct predictions out of 1,000 cases. Which threshold is selected? Why must you not change the selection after seeing that threshold 6 is slightly better on the holdout?

Hint

Apply the selection rule to the development data, then preserve the holdout's role as an untouched assessment.

Answer and solution

Threshold 5 is selected because 810 is the largest number of correct predictions. If the selection changes to 6 after inspecting the holdout, those data have influenced selection and no longer provide a separate assessment. Threshold 6 can become a hypothesis for a new experiment with a new holdout, but the old experiment's results must still be reported according to the original plan.

### 10.12.4 Exercise 4 - effect size and stopping rules

Calculate the improvement of threshold 5 over threshold 6 from 2,355 and 2,204 correct predictions in 3,000 cases. Is the target of 0.10 reached? May the experiment stop early at a replication showing 0.11?

Hint

Subtract the counts of correct predictions, then divide by the common number of cases.

Answer and solution

The improvement is $(2355 - 2204)/3000 = 151/3000 \approx 0.0503$, or about 5.03 percentage points. The target of 10 points is not reached. The experiment must not stop at a single replication showing 0.11 because the original rule requires every replication; stopping based on the results would change the design and make the summary overly favorable.

### 10.12.5 Exercise 5 - negative results and limits of evidence

Write an honest conclusion for the Unit 7 experiment and one claim that must not be made. Then state what would be needed to prove a general claim about the best threshold in a defined model.

Hint

Separate the holdout effect, the target of 0.10, and the full space of possibilities.

Answer and solution

An honest conclusion: threshold 5 was selected from the grid and then outperformed threshold 6 by about 5.03 percentage points on the holdout, but did not reach the specified minimum effect of 10 points. An invalid claim: threshold 5 has been proved best for all data or for the real world. For a general claim within an exact model, specify the score distribution and outcome probabilities, calculate the expected accuracy of every candidate (or of the entire threshold domain), and compare them using an argument that covers them all.

## 10.13 Summary

- The question, hypothesis, comparator, outcome measure, and minimum effect are specified before results are seen.
- Other variables are controlled so that differences are not confounded by sample difficulty or changes in procedure.
- The grid, tie-breaking rule, replications, seed schedule, and stopping rule are all part of the plan.
- A seed reproduces pseudorandomness, not unrecorded physical nondeterminism.
- Development data select parameters; the holdout assesses the selection once.
- Numerical effect size matters more than visual impression, and negative results must still be reported.
- Finite experiments provide limited empirical evidence; general proofs require arguments covering the domain of the claim.

# 11 Testing and Validating Computations

## 11.1 Learning objectives

After completing this unit, you will be able to:

- use assertions to state results or invariants that can be checked;
- distinguish example-based testing from property-based testing;
- distinguish unit, integration, and regression tests;
- choose an oracle without copying the algorithm under test;
- use exact equality and numerical tolerances where each is appropriate;
- test boundary values, preconditions, and error behavior;
- formulate metamorphic relations and invariants;
- validate an implementation against independently derived results; and
- limit claims based on testing so they are not mistaken for general mathematical proofs.

Local prerequisites: Units 1–5, functions, finite sums, quadratic equations, and the concept of a domain. This unit requires only the Python standard library.

## 11.2 Assertions and contracts

An assertion states a condition expected to hold at a particular point in execution. In `unittest` tests, expressions such as `self.assertEqual(actual, expected)` record a failure as a test result. Python's built-in `assert` statement is better suited to internal invariants whose failure indicates a programming error.

Do not use `assert n >= 0` as the sole validation of public input. Python can remove `assert` statements when run with optimization enabled. Preconditions on user input must be checked explicitly and produce documented error types.

## 11.3 Examples and properties

An example test pairs a particular input with a known output:

**Listing 11.1**

```
def triangular_iterative(n):
    if isinstance(n, bool) or not isinstance(n, int):
        raise TypeError("n must be an integer")
    if n < 0:
        raise ValueError("n must be nonnegative")

    total = 0
    for k in range(1, n + 1):
        previous = total
        total += k
        assert total >= previous
    return total
```

| $n$ | $1 + 2 + \cdots + n$ |
|---|---|
| 0 | 0 |
| 1 | 1 |
| 2 | 3 |
| 10 | 55 |
| 100 | 5,050 |

Examples are easy to read and especially useful for important cases or previously discovered errors. A few examples, however, do not explain the structure of the entire domain.

A property test checks a relation that should hold for many inputs. If $T(n) = 1 + 2 + \cdots + n$, then

$$T(n + 1) - T(n) = n + 1$$

and

$$T(2n) = 2T(n) + n^2.$$

The second relation does not determine one output from scratch; it relates several function calls. Such a relation is called **metamorphic** and is useful when the correct output for each input is difficult to calculate directly.

Testing a property at a thousand fixed values is still a finite check. It covers more ground than five examples, but it is not a universal proof.

## 11.4 Unit, integration, and regression tests

These three labels answer different questions.

| Level | Focus | Unit 8 example |
|---|---|---|
| unit | one function or small contract | `triangular_iterative(10) == 55` |
| integration | several parts working together | the oracle, properties, and JSON writer produce a valid report |
| regression | a particular error does not recur | the boundary value `n=0` still produces 0 |

A regression test is usually added after an error has been found and fixed. It should preserve the smallest input that triggers the error and the correct result. Calling every large test a "regression" obscures the failure history that the test is actually intended to guard against.

## 11.5 Oracles and reference values

An oracle is a source for deciding what the correct output is. Options include:

- small values calculated and reviewed in advance;
- a mathematical definition or identity;
- a simpler reference implementation;
- reference data with clearly documented origins; or
- invariants and metamorphic relations.

An oracle that repeats production code under a different name is not independent. The same error can exist in both places while every test passes.

The implementation tested in this unit adds the integers from `1` to `n` in a loop. Its main oracle is derived by pairing terms. Write the sum in forward and reverse order:

$$T(n) = 1 + 2 + \cdots + n,$$

$$T(n) = n + (n-1) + \cdots + 1.$$

Adding the two lines gives $n$ pairs, each with value $n + 1$. Thus

$$2T(n) = n(n+1), \qquad T(n) = \frac{n(n+1)}{2}.$$

The reference function uses this formula, not a second loop. Agreement between the two is therefore more informative than a comparison of two copies of the same loop.

## 11.6 Exact equality and tolerances

If the result is an exact integer or rational number, use exact equality. `triangular_iterative(10)` must equal `55`, not merely be close to it. A tolerance in such a test can hide an off-by-one error.

For a floating-point approximation, specify the quantity being checked and justify its tolerance. Newton iteration for $x^2 = 2$ produces an approximation to the square root of two. The Unit 8 test checks the residual

$$|x^2 - 2| \le 10^{-12}$$

after five iterations starting from $x_0 = 1$.

```
from math import isclose

x = 1.0
for _ in range(5):
    x = 0.5 * (x + 2.0 / x)

isclose(x*x, 2.0, rel_tol=0.0, abs_tol=1e-12)
```

**Listing 11.2**

```
True
```

The tolerance is applied to a residual with a clear meaning. Do not choose a tolerance after seeing a failure merely to make the test turn green.

## 11.7 Boundary values and domains

The summation function in this unit has the nonnegative integers as its domain. Tests should check at least:

- the boundary `n=0`;
- the first positive value `n=1`;
- an ordinary value such as `n=10`;
- negative values, which must be rejected;
- noninteger numbers, which must be rejected; and
- `True`, because `bool` is a subclass of `int` in Python but is not an input intended by this mathematical contract.

Error tests need to check the type of failure, not merely that "something failed." A `ValueError` for a negative integer and a `TypeError` for `2.5` express two different contract violations.

## 11.8 Invariants and metamorphic testing

A simple loop invariant is that the sum does not decrease after the next positive integer is added. Metamorphic relations compare different calls:

**Listing 11.3**

```
for n in range(201):
    value = triangular_iterative(n)
    assert triangular_iterative(n + 1) - value == n + 1
    assert triangular_iterative(2*n) == 2*value + n*n
```

Metamorphic relations do not require a table of 201 answers, but they still require a derivation. For the doubling relation,

$$T(2n) - 2T(n) = n(2n+1) - n(n+1) = n^2.$$

If a relation is formulated incorrectly, the test can reject a correct implementation. Test code needs review just as production code does.

## 11.9 Independent validation

Validation asks whether the program answers the intended question, not merely whether it is consistent with itself. In this unit:

1. the specification defines $T(n)$ as the sum of the integers from 1 to $n$;
2. the implementation uses repeated addition;
3. the oracle uses the independently derived pairing formula;
4. the table of small values provides an easily checked reference point; and
5. the increment and doubling properties provide additional cross-checks.

These different layers reduce the risk of a single error passing unnoticed. They do not make validation perfect: the specification may be wrong, the oracle may have been derived incorrectly, or the test domain may miss a defect.

## 11.10 A deterministic report

Run from the project root:

```
python source/code/unit08_validation.py
```

The report contains reference values, comparisons with the pairing formula, property violations, boundary and domain checks, numerical residuals, test levels, and limits on the conclusions. The JSON uses UTF-8, sorted keys, fixed indentation, and LF line endings. `core_sha256` binds the core contents before the hash field is added. This execution path uses no current time, network access, or random numbers.

Two byte-identical outputs show that two executions recorded the same results under the same conditions. This does not establish that the oracle or specification is necessarily correct.

## 11.11 Coverage, evidence, and proof

Coverage has several meanings. Line coverage says which parts of the code were executed. Branch coverage says which control-flow alternatives were taken. Domain coverage says which mathematical inputs were checked. Having 100% line coverage does not mean that every value or property has been tested.

If the program agrees with the pairing formula for `0 <= n <= 1000`, the justified conclusion is that no mismatch was found on that finite domain in the execution environment used. The proof of the pairing formula applies to all nonnegative integers because it treats $n$ generally

and pairs the terms. A proof that the loop implements the sum can use a loop invariant. Tests support the claim that the concrete implementation agrees with the argument in the cases checked; tests alone do not replace either proof.

## 11.12 Exercises

### 11.12.1 Exercise 1 — choosing an assertion

For each of the following claims, choose exact equality or a comparison with a tolerance and explain your choice: (a) the sum `1+...+100`, (b) a numerical approximation to $\sqrt{2}$, and (c) the length of an output list.

> **Hint**
>
> Ask whether the representation and the specified result are exact.

> **Answer and solution**
>
> Use exact equality for (a), namely `5050`, because Python integer operations in this range are exact. Use a comparison with a tolerance for (b), deriving the tolerance from a residual or the requirements of the problem. Use exact equality for (c), because a list length is an integer. A tolerance for the length could accept a list with the wrong number of elements.

### 11.12.2 Exercise 2 — constructing an oracle

You are testing a function that calculates $1 + 3 + 5 + \cdots + (2n - 1)$ with a loop. Derive an oracle that does not use the same loop and state its domain.

> **Hint**
>
> Draw a square that gains one L-shaped border at each step.

> **Answer and solution**
>
> The sum of the first $n$ odd numbers is $n^2$. A square of size $(n-1) \times (n-1)$ gains $2n-1$ new points to become an $n \times n$ square. The oracle can therefore calculate `n*n` for a nonnegative integer `n`. Compare the loop implementation with that formula, not with a copy of the loop.

### 11.12.3 Exercise 3 — metamorphic relations

For the function $S(n) = n^2$, write two metamorphic relations that can be tested without a complete table of values. Explain why the relations are valid.

> **Hint**
>
> Compare $S(n+1)$ with $S(n)$, then try the input $-n$.

> **Answer and solution**
>
> Two relations are $S(n+1) - S(n) = 2n + 1$ and $S(-n) = S(n)$. The first follows from $(n+1)^2 - n^2 = 2n + 1$; the second follows from $(-n)^2 = n^2$. They can detect different classes of error, but boundary examples and direct checks against some reference values are still needed.

### 11.12.4 Exercise 4 — a boundary regression test

A mean function once divided by zero when given an empty list, although its contract required it to reject that input with `ValueError`. Write the core of its regression test and explain why the test should be retained.

> **Hint**
>
> Use `with self.assertRaises(ValueError):`.

> **Answer and solution**
>
> The core is `with self.assertRaises(ValueError): mean([])`. Retain the test because it preserves the smallest input that once violated the contract and the behavior now required. If a later refactor returns `NaN` or allows `ZeroDivisionError` to escape, the test shows that the public behavior has changed.

### 11.12.5 Exercise 5 — the limits of testing evidence

All tests pass, every line of code is covered, and a million random inputs reveal no failure. State the strongest justified conclusion and give two reasons why this is not yet a general proof.

**Hint**

Distinguish execution coverage from domain coverage, and examine the oracle.

**Answer and solution**

The strongest conclusion is that the implementation passed the stated tests, coverage checks, and samples in that environment. This is not a general proof because a million samples do not exhaust an infinite domain, and the oracle or specification may share errors with the implementation. A general proof requires an argument covering the entire domain; further validation requires independent oracles, boundary cases, properties, and environment records.

## 11.13 Summary

- Appropriate assertions state local contracts that can be checked.
- Examples cover important cases; properties and metamorphic relations extend the checks without automatically providing a universal proof.
- Unit, integration, and regression tests protect against different layers of risk.
- Oracles must come from values, derivations, data, or implementations sufficiently independent of the code under test.
- Exact equality suits exact results; tolerances must reflect numerical meaning and scale.
- Boundary values, invalid domains, and error types are part of the contract.
- Cross-validation and canonical reports strengthen the evidence.
- Coverage and passing tests do not replace general mathematical proof.

# 12 Data, Configuration, and Provenance

## 12.1 Learning objectives

After completing this unit, you will be able to:

- distinguish raw data, derived data, and run reports;
- specify schemas, types, column meanings, and units;
- separate variable configuration from transformation code;
- record environment identity and the limits of lockfiles;
- use relative paths that cannot escape the project root;
- record the origins and rights status of each component;
- bind artifacts to byte counts and SHA-256 hashes without overstating what hashes establish;
- exclude private data from public artifacts; and
- distinguish semantic reproducibility from byte-for-byte reproducibility.

Local prerequisites: Units 1–4 and 7–8, tables, ratios, units, text files, and simple functions. The basic route uses only the Python standard library.

## 12.2 Three data layers

**Raw data** are the records received before this unit's analytical transformations. "Raw" does not mean without a history: a sensor, form, or export process may already have changed them. The term simply identifies the frozen entry point for this run.

Unit 9 uses five synthetic records:

| sample_id | length_cm | mass_g |
|---|---|---|
| S01 | 10.0 | 20.0 |
| S02 | 12.0 | 24.6 |
| S03 | 8.0 | 15.8 |
| S04 | 14.0 | 28.7 |
| S05 | 11.0 | 21.8 |

**Derived data** are produced by the rule

$$r = \frac{\text{mass in grams}}{\text{length in centimeters}},$$

then rounded to three decimal places using the *round half even* rule. The derived column is named `mass_per_length_g_per_cm`.

A **run manifest** is not new measurement data. It records which files were used, which transformation was performed, which configuration and environment applied, and which output bytes were produced.

Do not overwrite raw data with derived results. If a transformation needs to be corrected, retaining the raw data with recorded hashes allows the outputs to be regenerated and compared.

## 12.3 Schemas and units

A schema is a structural contract. This unit's raw-data schema specifies:

| Field | Type | Unit | Meaning |
|---|---|---|---|
| `sample_id` | text of the form `S` + two digits | none | synthetic identifier |
| `length_cm` | positive decimal text | cm | length |
| `mass_g` | positive decimal text | g | mass |

Storing decimals as text avoids an initial conversion to floating point. The code rejects numeric values that are not text, then converts the validated text to `Decimal` for calculation. An identifier has no unit; `None` is more accurate than an empty unit field that could mean "forgotten."

The name `length` without a unit is insufficient. A value of 10 could mean 10 mm, 10 cm, or 10 m, and these give different ratios. Units must stay with the data through to the output; the derived-data schema specifies `g/cm`.

Validation checks field names, identifier format, uniqueness, finiteness, and positivity. Passing schema validation does not guarantee correct measurements, but it prevents several structural misinterpretations.

## 12.4 Configuration separate from code

Transformation code states **how** the ratio is calculated. Configuration states the run settings that may legitimately vary, for example:

```
{
  "derived_path": "data/derived/unit09_mass-per-length.csv",
  "ratio_decimal_places": 3,
  "raw_path": "data/raw/unit09_measurements.csv",
  "rounding": "ROUND_HALF_EVEN",
  "schema": "o002.unit09.config.v1"
}
```

If the number of decimal places is embedded in many lines of code, a small change can easily become inconsistent. A single validated configuration makes that choice inspectable and allows it to be bound to a hash.

Separating configuration does not mean accepting arbitrary values. This unit accepts only 0 through 9 decimal places and one tested rounding rule. Configuration is input with a schema, not a back door for executing text as code.

## 12.5 Environment identity and locks

Outputs can change even when the data and code appear identical, because language, library, operating-system, or hardware versions differ. The Unit 9 manifest records the Python implementation and version and states that the basic route uses only the standard library.

Because there are no external dependencies, this unit's "environment lock" is small: the Python identity and the `Decimal` arithmetic context are serialized canonically and bound to a SHA-256 hash. The context fixes precision at 28 digits, round-half-even rounding, exponent limits, and operation traps; it does not inherit the calling process's precision or traps. Projects using NumPy, SymPy, SageMath, or other libraries need a lockfile that specifies exact versions and package sources. A list of names without versions is not a lock.

A lockfile does not freeze the entire world, either. Differences in operating system, locale, CPU, or system libraries may still matter. Record the layers that affect the question, and do not call environments "identical" merely because one version number matches.

## 12.6 Safe relative paths

The manifest stores logical paths such as `data/raw/unit09_measurements.csv`, not `user-profile/Name/Desktop/data.csv`. Relative paths can move with the project and do not disclose local account names.

The unit's validator accepts canonical relative POSIX paths. It rejects:

- `/etc/secret` or `C:/secret`, because they are absolute;
- `../secret` and `data/../secret`, because they can escape the root or obscure their destination;
- backslashes, so one form is used on every system; and
- redundant forms such as `data//raw/file.csv`.

Each artifact role must also have a unique path after Unicode normalization and case folding. Consequently, `data/raw.csv` and `DATA/RAW.CSV` are treated as a collision, so that the same manifest remains safe on case-insensitive filesystems.

Path validation is a security and reproducibility boundary. It is not, however, permission to read every file that happens to be inside the project. The list of permitted inputs must still be limited by the run plan.

## 12.7 Provenance and component-specific rights

Provenance answers "where did this component come from, and what happened to it?" Rights answer "on what basis may this component be used or distributed?" The two are related but distinct.

The manifest records separately:

- synthetic raw data defined in the code;
- derived data and their transformation formula;
- configuration;
- environment identity; and
- the Unit 9 code, which points to `LICENSE-CODE.md` and the MIT license.

The example data contain no third-party material. These records are not used to make a blanket licensing claim about every project file. If a CC BY-SA image, MIT code, and specially licensed data are combined, each component still needs its own creator/source, license, change record, and restrictions. Provenance must not be discarded merely because the derived results are numbers.

## 12.8 SHA-256 and byte counts

For each component, the manifest stores a logical path, media type, role, byte count, and SHA-256 hash. If one byte changes, the hash will almost certainly change. The size helps detect empty files or accidental truncation and provides a simple check before hashing.

A hash answers a question about byte identity. It does not prove that:

- the transformation formula is correct;
- the units are appropriate;
- the data were collected ethically;
- the recorded license is valid; or
- the manifest includes every component it should.

Those questions require other checks. Matching hashes of an incorrect file only establish that two people have the same incorrect file.

## 12.9 Privacy and exclusions

Reproducibility is not a reason to publish names, email addresses, phone numbers, precise locations, or free text that could identify someone. The unit's dataset uses `S01` through `S05` as synthetic identifiers and explicitly rejects private fields.

In real research, strategies may include:

- separating restricted data from public artifacts;
- publishing schemas, code, synthetic data, and safe summaries;
- recording how authorized researchers can apply for access;
- removing unnecessary fields before analysis; and
- checking whether combinations of fields can still identify people.

Writing a confidential dataset's hash into a public manifest can also leak information in some circumstances, especially when the space of possible values is small. Decide what may be published before creating a public manifest.

## 12.10 A canonical run manifest

Run from the project root:

```
python source/code/unit09_provenance.py --output output/unit09-manifest.json
```

The script does not access the network or the current time. The manifest contains raw and derived records, schemas, configuration, environment identity, artifact hash bindings, the input-code-configuration-output chain, privacy policy, component-specific rights, and limits on reproducibility. The JSON uses UTF-8, sorted keys, fixed indentation, and a single final LF newline. `core_sha256` binds the contents before that hash field is added.

The raw and derived CSV files in this example are logical artifacts whose canonical bytes are constructed in memory and bound to hashes in the manifest. The command therefore writes only the requested JSON, but you can reconstruct the exact CSV files from the stored records and column rules.

## 12.11 Semantics and bytes

Two outputs are **byte-for-byte reproducible** if every byte matches. They necessarily have the same SHA-256 hash. This criterion is useful for checking deterministic workflows, archives, and distribution.

Two outputs can be **semantically reproducible** even when their bytes differ. For example, the following JSON texts store the same mapping:

```
{"a":1,"b":2}
{
  "b": 2,
  "a": 1
}
```

Whitespace and key order differ, so the hashes differ, but a JSON parser produces the same value. In numerical computing, two algorithms can also produce equivalent values within a specified tolerance and with specified units without producing identical bytes.

A report must therefore state the level required. Unit 9's deterministic repetition targets identical bytes in a locked environment. Broader scientific validation must also compare schemas, units, formulas, and the meaning of the results.

## 12.12 Exercises

### 12.12.1 Exercise 1 — raw data, derived data, and lineage

If `mass_g` for S03 changes from `15.8` to `15.9`, which components must have new hashes, and which should remain unchanged?

Hint

Follow the chain raw data → transformation → derived data; the code and configuration are not edited.

Answer and solution

The raw-data hash changes, although its byte count may remain the same, and the derived-data hash must change because S03's ratio is calculated from the new mass. The manifest's `core_sha256` also changes because its artifact bindings change. The code, configuration, and environment-identity hashes remain the same if their bytes are unchanged. Provenance must record that the new outputs came from that revision of the raw data.

### 12.12.2 Exercise 2 — schemas and units

A table replaces `length_cm` with a column named `length` without updating the schema. Why is the value `10.0` insufficient for calculating an interpretable ratio? Give a correction.

Hint

Numbers do not automatically carry units.

Answer and solution

`10.0` could mean millimeters, centimeters, meters, or another unit. The ratio and its units change with that choice. Correct this by using an unambiguous field such as `length_cm`, stating `unit: cm` in the schema, validating the data against the schema, and changing the output formula if unit conversion is required.

### 12.12.3 Exercise 3 — configuration and safe paths

A configuration contains `raw_path: "../../private.csv"`, and the number of decimal places is written directly into three different functions. Identify two problems and give their corrections.

Hint

Check for directory traversal and for a single source of run settings.

Answer and solution

The path `../../private.csv` can escape the project root and must be rejected; use a canonical relative path from the permitted input list. Embedding the number of decimal places in three locations makes inconsistent changes easy. Store one `ratio_decimal_places` value in versioned configuration, validate its range, and have every call read the same value.

### 12.12.4 Exercise 4 — rights and privacy

You receive a third-party table containing measurements, names, and email addresses. State what must be separated or recorded before creating a public artifact.

Hint

Provenance, permission to redistribute, and analytical need are three different questions.

Answer and solution

Record the source, creator, version, license or basis for permission, and changes to the table component. Separate names and email addresses from the public data unless there is a valid basis, need, and consent; use safe identifiers or synthetic data where appropriate. Check whether combined measurements can still identify people. Do not assume that the project's code license automatically covers third-party data.

### 12.12.5 Exercise 5 — semantics, bytes, and hashes

Two JSON files have different key orders and whitespace but produce the same object when parsed. What can you say about their semantic reproducibility, byte-for-byte reproducibility, and SHA-256 hashes?

Hint

Compare the parsed value with the original sequence of characters.

Answer and solution

They are semantically reproducible for that JSON mapping because their parsed values match. They are not byte-for-byte reproducible because their character sequences differ, so their SHA-256 hashes almost certainly differ. Each hash binds only its file's bytes; to require the same hash, serialize both using the same canonical rules.

## 12.13 Summary

- Raw data, derived data, and run manifests have different roles and must not overwrite one another.
- Schemas include types, meanings, and units; structural validation is not proof of measurement quality.
- Configuration is separated from code but still has a schema and limits.
- Python identity and versioned dependencies form a minimum environment lock.
- Canonical relative paths improve portability and prevent directory traversal.
- Each component retains its own provenance and rights status.
- Sizes and SHA-256 hashes bind bytes, not correctness, ethics, or licensing.
- Privacy may require data to be excluded from public artifacts.
- Semantic reproducibility does not always require identical bytes; a manifest must state its target level.

# 13 Automation and Reproducible Workflows

## 13.1 Learning objectives

After completing this unit, you will be able to:

- describe an experiment as a graph of tasks and dependencies;
- distinguish sources, intermediate products, and final artifacts;
- rebuild only when recorded inputs, parameters, code, or environment change;
- detect missing dependencies and cycles before running tasks;
- write artifacts atomically so that failures do not leave apparently complete results;
- explain why timestamps alone are not evidence of identical contents; and
- audit workflow records without treating a successful process as mathematical proof.

Local prerequisites: Units 1–2 and 8–9. You should understand functions, files, hashes, configuration, and tests before automating them.

## 13.2 From a command list to a graph

A small experiment is often written as a list of commands:

```
generate-data
analyze-data
draw-figure
write-report
```

That sequence hides the reason for each step. A dependency graph states that the analysis needs the data, the figure needs the analysis, and the report needs both the analysis and the figure. With that information, a workflow manager can determine a valid order and which work needs to be repeated.

The graph must be acyclic. If `analysis` requires `report` while `report` requires `analysis`, there is no first task that can be completed.

## 13.3 Sources and artifacts

- **Sources** are written or obtained outside the workflow: code, configuration, raw data, and licenses.
- **Intermediate products** can be regenerated but support the next step.
- **Final artifacts** are delivered to readers or reviewers.

Do not make a final artifact the only copy of source data. An intermediate product that is expensive to regenerate may be cached, but its hash and the rules for creating it must still be known.

## 13.4 When a task is up to date

Timestamps are useful for local optimization, but they can change when files are copied or machine clocks disagree. A robust record binds together:

1. the hash of each input's bytes;
2. the hash or version of the task code;
3. canonicalized parameters;
4. the relevant environment identity; and
5. the hash of each output.

The Unit 10 script stores the complete, canonical fingerprint preimage: input names and their SHA-256 hashes, the task code's SHA-256 hash, parameters, and the identity of the runtime actually running the example. The runtime identity includes the Python implementation and version, cache tag, operating-system family, machine architecture, and byte order. It deliberately excludes the hostname and profile paths. `task_fingerprint` is the SHA-256 hash of the preimage's canonical JSON bytes, so it can be recalculated directly from the record.

If any input changes, the task and all its descendants become out of date. If everything matches and the outputs still have their recorded hashes, the task can safely be skipped. The checker also accepts a list of required output names. An empty output mapping, a missing required output, or an extra undeclared output makes the task out of date. This unit does not define a contract for tasks with no outputs.

## 13.5 Atomic writes and failure

Writing directly to `results.json` can leave a truncated file if the process stops. A safer pattern is:

1. write to a new temporary file in the same directory;
2. close it and synchronize it if crash durability is required;
3. verify its structure and hash;
4. atomically rename the temporary file to the destination name; and
5. do not remove the old result until its replacement is ready.

The temporary name must be unique and must not refer to an existing link. Cleaning up temporary files is local failure handling, not a replacement for a manifest.

## 13.6 Named targets and checks

The O002 workflow has at least these targets:

```
test     run tests without building the reader
results  produce experiment artifacts
html     build the web reader after test and results
pdf      build the print reader after test and results
qa       check both readers and the manifest
all      run the complete set of acceptance checks
```

The `all` target must not publish. Publication is a separate transaction that uses only artifacts that have passed `qa`.

## 13.7 An example workflow

This unit's script validates a small graph, calculates a topological order, and creates a canonical record containing the fingerprint preimage, actual runtime, and output contract:

```
python source/code/unit10_pipeline.py --output output/unit10-results.json
```

The example does not execute shell commands. It models workflow decisions so that ordering, cycle detection, and freshness logic can be tested without external effects.

## 13.8 Automation does not determine truth

A green workflow establishes that the programmed checks passed in that environment. It does not prove that the checks are sufficient, the model is appropriate, or the theorem is true. Automation reduces recurring oversights; mathematical responsibility remains with the designers and reviewers.

## 13.9 Exercises

### 13.9.1 Exercise 1 — task order

Task `data` has no dependencies; `fit` requires `data`; `plot` requires `data` and `fit`; `report` requires `plot`. Give one valid order.

> **Hint**
>
> Every task must appear after all its dependencies.

> **Answer and solution**
>
> The order `data, fit, plot, report` is valid. `plot` cannot precede `fit`, and `report` cannot precede `plot`. When there are several independent tasks, more than one order may be valid.

### 13.9.2 Exercise 2 — a cycle

Explain why the dependencies `a -> b`, `b -> c`, and `c -> a` must be rejected before any task runs.

> **Hint**
>
> Look for a task whose dependencies have all been completed.

> **Answer and solution**
>
> There is no initial task: `a` waits for `b`, `b` waits for `c`, and `c` waits for `a`. Running part of the workflow before detecting the cycle can leave apparently valid artifacts, so graph validation must precede any side effects.

### 13.9.3 Exercise 3 — timestamps

A data file is copied with the same contents but a new timestamp. Must the analysis always be repeated? Compare timestamp-based and hash-based policies.

> **Hint**
>
> A timestamp records a filesystem event, not contents.

> **Answer and solution**
>
> A timestamp-based policy may rebuild even when the contents are unchanged. A hash-based policy can skip the analysis if the data bytes, code, parameters, environment, and recorded outputs all still match. Hashing requires more data to be read but binds more directly to the contents.

### 13.9.4 Exercise 4 — a write failure

Why is writing new output directly over an old artifact dangerous? Set out a safe writing sequence.

> **Hint**
>
> Consider a process that stops after writing half the bytes.

> **Answer and solution**
>
> A failure can destroy the old artifact and leave truncated output under the official filename. Write to a uniquely named temporary file in the same directory, close and verify it, then replace the destination atomically. If a step fails, the old artifact remains available and the temporary file can be disposed of.

### 13.9.5 Exercise 5 — a green workflow

All workflow targets pass. Give two reasons why the mathematical conclusion could still be wrong.

> **Hint**
>
> Tests may be incomplete, and the model may not match the question.

Answer and solution

The tests may omit a case that exposes an incorrect implementation, and the model or mathematical assumptions may be wrong even when the code implements them accurately. A green workflow strengthens the reproducibility of the checks, not their logical completeness.

## 13.10 Summary

- A dependency graph explains ordering and the effects of changes.
- A complete fingerprint preimage makes the hash of inputs, code, parameters, and runtime reproducible.
- Content hashes and output contracts bind freshness more directly than timestamps; an empty output mapping is not considered up to date.
- Cycles and missing dependencies must be rejected before side effects occur.
- Atomic writes prevent half-finished outputs from using the official filename.
- Automation makes checks consistent but does not replace mathematical judgment.

# 14 Numerical Experiments and Their Prerequisites

## 14.1 Learning objectives

After completing this unit, you will be able to:

- choose numerical methods whose mathematical prerequisites you understand;
- record invariants, tolerances, iteration limits, and stopping criteria;
- distinguish problem-formulation error, discretization error, rounding error, and implementation error;
- check results through substitution, residuals, bounds, or comparison methods;
- run convergence experiments without treating them as general proofs; and
- reject library usage that hides assumptions about the domain.

This unit offers several routes. The root-finding route requires A30 and the concept of a continuous function. The quadrature route requires B30, the linear systems route requires B40, and the differential equations route requires B70. Defer any route whose prerequisites you have not yet met.

> **!** The B80 core boundary
>
> Only the sections on methods and theorems, A30 bisection, the SciPy comparison, and sources of error, together with Exercises 1 and 2 and SciPy exercise `s01`, form this unit's B80 core. The B30, B40, and B70 routes remain available as extensions, but must not count as evidence of B80 completion before their respective prerequisites have been met.

## 14.2 Methods come with theorems

A software command does not guarantee that a problem satisfies a method's assumptions. Bisection, for example, maintains an interval on which the function has opposite signs at the two endpoints. Guaranteeing a root in that interval requires continuity. A program can check the endpoint signs; it cannot infer continuity from a few samples. This unit's implementation also rejects nonfinite endpoints, midpoints, or function values.

Every numerical experiment states:

1. the problem and domain;
2. the mathematical assumptions;
3. the method and representation;
4. the parameters and stopping criteria;
5. the invariants being checked;
6. the error measure or residual; and
7. the limits of the conclusions.

## 14.3 A30 route: bisection

To find a root of $f(x) = 0$, start with $[a, b]$ such that $f(a)f(b) < 0$. Each step selects the midpoint and keeps the half-interval that still has a sign change.

If the initial width is $w_0 = b - a$, after $k$ steps the width is $w_k = w_0/2^k$. This relationship provides an interval-based error bound, rather than relying only on a small value of $f(x)$.

```
from source.code.unit11_numerical import bisection

result = bisection(lambda x: x*x - 2.0, 1.0, 2.0, width_tolerance=1e-12)
(result.midpoint, result.width, result.iterations)
```

**Listing 14.1**

```
(1.414213562372879, 9.094947017729282e-13, 40)
```

That value is an approximation. The exact statement $x = \sqrt{2}$ comes from the definition of the positive root of $x^2 = 2$, not from the printed digits.

## 14.4 A30 route: an executable SciPy comparison

The original implementation makes each bisection step readable. A second implementation, `scipy.optimize.root_scalar`, provides an independently developed comparison. Both take a specified function, initial interval, method, and iteration limit, but their tolerance parameters must not be treated as equivalent. `original_width_tolerance` bounds the **final interval width** in the original implementation; `scipy_xtol` and `scipy_rtol` specify absolute and relative closeness criteria for the root returned by SciPy. Their numerical values may coincide in this example, but their meanings remain distinct and are recorded separately.

```
from source.code.unit11_numerical import (
    SCIPY_VERSION,
    compare_bisection_with_scipy,
    square_minus_two,
)

comparison_u11 = compare_bisection_with_scipy(
    square_minus_two,
    1.0,
    2.0,
    original_width_tolerance=1e-12,
    scipy_xtol=1e-12,
)

assert comparison_u11["scipy_version"] == SCIPY_VERSION
assert comparison_u11["verification"]["strict_sign_change"]
assert comparison_u11["scipy"]["converged"]
assert comparison_u11["verification"]["roots_inside_initial_bracket"]
assert comparison_u11["verification"]["roots_agree"]

(
    comparison_u11["scipy_version"],
    comparison_u11["original"]["midpoint"],
    comparison_u11["scipy"]["root"],
    comparison_u11["verification"]["absolute_root_difference"],
)
```

**Listing 14.2**

```
('1.15.2', 1.414213562372879, 1.4142135623724243, 4.547473508864641e-13)
```

The record stores the version of SciPy actually imported, not just the library name. `strict_sign_change` checks $f(a)f(b) < 0$ before the call. `converged` and `flag` record SciPy's report. Both roots must remain inside the initial interval, and their difference must not exceed the recorded agreement tolerance.

Agreement between two implementations improves the chance of detecting local coding errors. It does not prove that the function is continuous or that the interval contains exactly one root, nor does it turn approximate digits into an exact value. Continuity and the existence guarantee rest on the assumptions and the Intermediate Value Theorem; the version and convergence checks provide computational evidence.

## 14.5 B30 route: quadrature and refinement

> ⚠ B30 prerequisite gate - deferred extension
>
> The quadrature section and Exercise 3 require B30. Both can be used for enrichment once that prerequisite has been met, but neither counts as B80 core content or evidence of B80 completion.

The composite trapezoidal rule replaces the curve on each subinterval with a line segment. A refinement experiment uses $n, 2n, 4n, \ldots$ subintervals and records changes in the result. Values that appear stable support a conjecture of convergence, but do not replace error analysis.

For a linear function, the trapezoidal rule is exact in exact arithmetic. This case provides a stronger implementation check than an arbitrarily chosen reference number.

## 14.6 B40 route: linear systems and residuals

> ⚠ B40 prerequisite gate - deferred extension
>
> The linear systems section and Exercise 4 require B40. Neither counts as B80 core content or evidence of B80 completion.

If a program proposes $\hat{x}$ as a solution of $Ax = b$, compute the residual $r = b - A\hat{x}$. A small residual means that the equation is nearly satisfied on the scale being used. A small residual does not always mean a small error in the solution; the relationship depends on the matrix's conditioning.

For a two-by-two system with rational coefficients, an exact solution using `Fraction` can provide an independent comparison for the floating-point route.

## 14.7 B70 route: Euler steps

> ⚠ B70 prerequisite gate - deferred extension
>
> The differential equations section and Exercise 5 require B70. Neither counts as B80 core content or evidence of B80 completion.

For the problem $y' = g(t, y)$, an Euler step uses

$$y_{k+1} = y_k + hg(t_k, y_k).$$

Reducing $h$ and comparing with a known solution can reveal an error pattern in the example. This tests the implementation and the behavior of that case; it does not prove an order of convergence for every function $g$.

## 14.8 Four sources of disagreement

When a result differs from expectations, distinguish:

- **problem-formulation error:** the model or data does not represent the question;
- **discretization error:** the method replaces a continuous problem with a finite one;
- **rounding error:** floating-point representation changes the operations; and
- **implementation error:** the code does not carry out the intended method.

Changing a tolerance addresses only some of these sources. A report must therefore not label a single number "numerical error" without defining it.

## 14.9 Experiment artifacts

Run the local examples whose prerequisites you have met:

```
python source/code/unit11_numerical.py --output output/unit11-results.json
```

The output records the final bisection interval, the quadrature refinement table, system solutions and residuals, and Euler-step errors. The function identities, domains or initial data, tolerances, refinement grids, step counts, and reference values used by the checks are stored alongside the results.

The record also includes the SciPy comparison, the SciPy version, interval checks, convergence, agreement between results, and the limits of the evidence. The quadrature, linear systems, and Euler tables are retained as extension regression checks; their presence in the JSON does not make them part of the B80 core. The `curriculum_routes` field makes this separation explicit.

## 14.10 Exercises

### 14.10.1 Exercise 1 - bisection invariants

Name two invariants that should be checked at each bisection step, and explain why the midpoint value alone is insufficient.

> **Hint**
>
> Consider the sign change and the shrinking interval.

> **Answer and solution**
>
> The interval endpoints must continue to bracket a sign change, and the width must halve at each step. The midpoint value alone does not establish that a root remains bracketed or that the implementation selected the correct half.

### 14.10.2 Exercise 2 - an iteration bound

What is the minimum number of steps guaranteeing an interval width of at most $10^{-6}$ if the initial width is 1?

> **Hint**
>
> Find $k$ such that $2^{-k} \leq 10^{-6}$.

> **Answer and solution**
>
> Since $2^{19} = 524288 < 10^6$ and $2^{20} = 1048576 > 10^6$, 20 steps are needed. After 20 steps, the width is $2^{-20} < 10^{-6}$.

### 14.10.3 SciPy exercise - two implementations for a root of a cubic

Use `compare_bisection_with_scipy` to find a root of $f(x) = x^3 - x - 2$ on $[1, 2]$ with a width tolerance of $10^{-10}$. Verify the sign change, SciPy version, convergence report, inclusion of both results in the interval, and agreement between the roots. Explain one matter that still requires a mathematical argument.

**Hint**

Define the Python function first. The comparison result has fields named `endpoint_values`, `scipy_version`, `scipy`, `original`, and `verification`.

**Self-check**

```
from source.code.unit11_numerical import SCIPY_VERSION,
 ↪  compare_bisection_with_scipy

def cubic_function_u11(x):
    return x**3 - x - 2

cubic_result_u11 = compare_bisection_with_scipy(
    cubic_function_u11,
    1.0,
    2.0,
    original_width_tolerance=1e-10,
    scipy_xtol=1e-10,
)

assert cubic_result_u11["endpoint_values"][0] < 0
assert cubic_result_u11["endpoint_values"][1] > 0
assert cubic_result_u11["scipy_version"] == SCIPY_VERSION
assert cubic_result_u11["scipy"]["converged"]
assert cubic_result_u11["verification"]["roots_inside_initial_bracket"]
assert cubic_result_u11["verification"]["roots_agree"]
assert (
    cubic_result_u11["verification"]["absolute_root_difference"]
    <= cubic_result_u11["verification"]["agreement_tolerance"]
)
```

Answer and solution

*Solution* 14.1.

```
from source.code.unit11_numerical import SCIPY_VERSION,
 ↪  compare_bisection_with_scipy

def solution_cubic_function_u11(x):
    return x**3 - x - 2

cubic_solution_u11 = compare_bisection_with_scipy(
    solution_cubic_function_u11,
    1.0,
    2.0,
    original_width_tolerance=1e-10,
    scipy_xtol=1e-10,
)

cubic_verification_u11 = cubic_solution_u11["verification"]
assert cubic_solution_u11["method"] == "bisect"
assert cubic_solution_u11["scipy_version"] == SCIPY_VERSION
assert cubic_solution_u11["scipy"]["flag"] == "converged"
assert cubic_verification_u11["strict_sign_change"]
assert cubic_verification_u11["roots_inside_initial_bracket"]
assert cubic_verification_u11["roots_agree"]
```

Since $f(1) = -2$ and $f(2) = 4$, the endpoint values have opposite signs. Assuming that $f$ is continuous, the Intermediate Value Theorem guarantees at least one root in the interval. The two implementations agree within the recorded tolerance, and SciPy reports convergence. However, the two programs prove neither continuity nor uniqueness of the root; both require separate mathematical arguments.

### 14.10.4 Exercise 3 - a quadrature check

**Curriculum status:** deferred B30 extension; not B80 core.

Apply the trapezoidal rule to $f(x) = 3x + 2$ on $[0, 4]$ with one subinterval. Compare with the exact integral.

Hint

The trapezoid's area is its width times the average endpoint height.

Answer and solution

The endpoint values are 2 and 14, so the trapezoid's area is $4(2+14)/2 = 32$. The exact integral is $[3x^2/2 + 2x]_0^4 = 24 + 8 = 32$. This exact agreement is expected because the graph of a linear function on the interval is itself a line segment.

### 14.10.5 Exercise 4 - a residual is not a solution error

**Curriculum status:** deferred B40 extension; not B80 core.

Explain why a small residual does not automatically guarantee that an approximate solution is close to the exact solution.

Hint

Consider a system that is highly sensitive to small changes in the right-hand side.

Answer and solution

The residual measures how well $\hat{x}$ satisfies the given equation. For an ill-conditioned matrix, a small change in the right-hand side can correspond to a large change in the solution. A residual must therefore be interpreted alongside scale and conditioning information, not directly as the error in the solution.

### 14.10.6 Exercise 5 - an Euler experiment

**Curriculum status:** deferred B70 extension; not B80 core.

For $y' = y$, $y(0) = 1$, compare the errors at $t = 1$ for $h = 1/10$ and $h = 1/20$. What conclusions are justified?

Hint

Use the reference solution $e^t$, and do not turn two cases into a theorem.

Answer and solution

Euler's method gives $(1 + h)^{1/h}$. For $h = 0.1$ the value is approximately 2.5937; for $h = 0.05$ it is approximately 2.6533; whereas $e \approx 2.7183$. The error decreases in these two cases. This supports the expected behavior and checks the implementation, but two step sizes do not prove convergence or an error order in general.

## 14.11 Summary

- Numerical methods come with assumptions and theorems.
- The original implementation and `scipy.optimize.root_scalar(method="bisect")` provide an A30 cross-check that must record the version, interval, tolerances, convergence, and agreement between results.
- Stopping criteria must be specified before inspecting the results.
- Residuals, interval bounds, and exact cases provide different checks.
- Problem-formulation, discretization, rounding, and implementation errors must not be conflated.
- Refinement experiments provide computational evidence; proving convergence requires a mathematical argument.
- B30 quadrature, B40 linear systems, and B70 Euler steps remain extensions and must not count as B80 core content.

# 15 A Verifiable Capstone Project

## 15.1 Learning objectives

After completing this unit, you will be able to:

- turn a mathematical question into an experiment with explicit limits on its claims;
- separate mathematical claims, implementation claims, and empirical observations;
- submit source, data, environments, commands, tests, and artifacts linked by hashes;
- ask someone else to reproduce the results without additional spoken instructions;
- respond to substantive review with verifiable changes or reasons; and
- explain which parts of a project constitute proof and which provide only computational evidence.

Local prerequisites: all O002 core units relevant to the project, and all mathematical prerequisites of the chosen topic. A project must not use an advanced mathematical topic merely because a library can compute with it.

## 15.2 The question comes before the tools

A project starts with a question, not a desire to use a particular library. A good question specifies the objects, domain, quantities to observe, and the form of an answer that would be meaningful. For example:

> How does the error in evaluating an algebraic expression change as the input approaches a point of cancellation, and which reformulation preserves more significant digits over the specified range?

This question does not promise a universal theorem. It defines an experimental scope, a comparison of methods, and an error measure that can be audited.

## 15.3 Three kinds of claim

Label each project conclusion with one of the following kinds.

| Kind | Example | Evidence required |
|---|---|---|
| mathematical | two formulas are equivalent on a specified domain | a symbolic argument stating the domain and exceptions |
| implementation | a function returns a residual within a specified bound | inspection, tests, and edge cases |
| empirical | method A is more accurate on a particular test grid | data, parameters, environment, error measure, and analysis |

The most serious error is to promote an empirical observation to a mathematical claim without an argument. Another error is to give a correct mathematical proof without checking whether the program actually implements the method it justifies.

## 15.4 The minimum reproducibility package

A minimal project package has the following structure.

```
project/
  README.md
  LICENSES/
  environment/
  source/
  tests/
  inputs/
  outputs/
  RUN_MANIFEST.json
```

`RUN_MANIFEST.json` records the schema version; `command` as an array of arguments; `environment` with the runtime, runtime version, and dependencies; the `parameters` object; lists of `inputs` and `outputs`; size and hash records in `artifacts`; and an array of `claims`. Every input and output path must refer to a registered artifact. Each claim object contains `id`, `kind`, `statement`, `evidence`, and `limitations`, and its `evidence` must refer to registered outputs. Artifact hashes thus also bind the evidence supporting each claim. The manifest must contain at least one artifact, one output, and one claim, and paths must not be repeated.

Paths must use canonical POSIX form and be relative to the package root; machine-specific user-profile paths must not be included in the public package.

This unit's verification script takes a manifest and a package root:

```
python source/code/unit12_capstone.py verify
```

This short command uses the default layout above: package `project`, manifest `project/RUN_MANIFEST.json`, and report `output/unit12-results.json`. The `--root`, `--manifest`, and `--output` options are available for other layouts.

The verifier does not determine whether a mathematical argument is correct. It checks that the named files exist, their sizes and hashes match, paths stay inside the package, the execution contract has the minimum required structure, claims contain the required fields, and each claim's evidence refers to outputs that have actually been verified. It does not execute `command`, assess the contents of the README or licenses, or establish that the minimum folder structure is sufficient; those matters are still checked through blind reproduction and the rubric.

## 15.5 A bounded blind reproduction

Give the package to a reviewer who did not help create it. The reviewer follows the README and records:

1. the environment actually installed;
2. the commands executed;
3. input and output hashes;
4. which tests passed or failed;
5. differences in results; and
6. ambiguous or incomplete instructions.

"It worked on the author's computer" is not a reproducibility standard. Conversely, differences between machines do not necessarily mean the research is wrong. Trace differences to versions, platforms, nondeterminism, tolerances, or defects in the package.

## 15.6 Review and revision

Substantive review challenges claims, methods, domains, tests, visualizations, or reproducibility. The project's response includes:

- a summary of the issue raised;
- a decision to accept, partly accept, or reject the criticism;

- the changes made to files or arguments;
- any new evidence added; and
- the limitations that remain.

Rejecting a comment is allowed, but "my computer gives the same result" is not a sufficient reason when the comment concerns the validity of a claim.

## 15.7 Completion rubric

A project passes only when all of the following conditions are met:

1. the question and domain are explicit;
2. sources, data, and component rights are clear;
3. the environment can be installed and the commands rerun;
4. tests cover relevant invariants and edge cases;
5. artifacts and the manifest agree;
6. visualizations include labels, units, scales, and descriptions;
7. mathematical, implementation, and empirical claims are kept distinct;
8. conclusions state their limitations;
9. a bounded blind reproduction is documented; and
10. one response to substantive review is included.

Failure to meet one condition does not necessarily invalidate the entire investigation, but the project does not meet the O002 standard until the shortcoming is corrected or its resolution is openly documented.

## 15.8 Exercises

### 15.8.1 Exercise 1 - classifying claims

Classify the following statement: "On Python 3.13 under Windows, algorithm A takes less time than B for all 100 sizes tested." Does the statement prove that A is always faster?

> **Hint**
>
> Consider the platform, version, and set of sizes.

Answer and solution

It is an empirical observation limited to that environment and the 100 test sizes. It does not prove that A is always faster for every size, dataset, platform, version, or implementation. The conclusion must retain those limits.

### 15.8.2 Exercise 2 - an incomplete manifest

A manifest records the script name and output image, but no parameters, library versions, or data hashes. Name three reproducibility failures that could arise.

Hint

Ask whether the reviewer can identify the same inputs, environment, and bytes.

Answer and solution

The reviewer might use different parameters, encounter different behavior because of library versions, or use data with the same filename but different bytes. All three situations can produce a different image without a reliable way to identify the source of the difference.

### 15.8.3 Exercise 3 - an unsafe path

Why must a public manifest not refer to a user-profile path such as `user-profile/Name/Desktop/data.csv`? Provide a replacement.

Hint

Consider privacy and portability.

Answer and solution

The path exposes the local profile structure and does not exist on the reviewer's machine. Store the data inside the package, for example at `inputs/data.csv`, then use a path relative to the project root and bind the file's bytes with its size and SHA-256 hash.

### 15.8.4 Exercise 4 - nonidentical reproduction

Two machines produce values that differ in the last digit, but both fall within the tolerance specified before the experiment. Has reproduction failed?

> **Hint**
>
> Distinguish semantic reproduction from byte identity.

> **Answer and solution**
>
> Not necessarily. If the contract permits floating-point variation and both values meet a justified tolerance, the semantic results can agree even when their numerical bytes differ. The manifest must still record the platform and versions, however, and artifacts claimed to be byte-identical must be checked with hashes.

### 15.8.5 Exercise 5 - responding to review

A reviewer points out that a graph truncates the vertical axis, making small differences appear very large. Write an adequate response.

> **Hint**
>
> Explain the decision, changes, and their effect on the conclusion.

> **Answer and solution**
>
> Acknowledge that the axis choice can exaggerate the visual impression. Add a graph showing the full range or a clear truncation marker, retain the truncated graph only if it helps reveal local variation, and report the effect size numerically. Revise the conclusion if the previous claim depended on a misleading visual impression, then record the new artifact's hash.

## 15.9 Summary

- A project starts with a question and a domain, not with tools.
- Mathematical, implementation, and empirical claims require different evidence.
- Manifests and hashes bind artifacts, but do not prove theorems.
- Blind reproduction tests whether the package and instructions are sufficient.
- Substantive review must receive a response containing verifiable changes or reasons.
- Completing a project requires mathematical, computational, documentary, and collaborative work, all of which must be open to inspection.