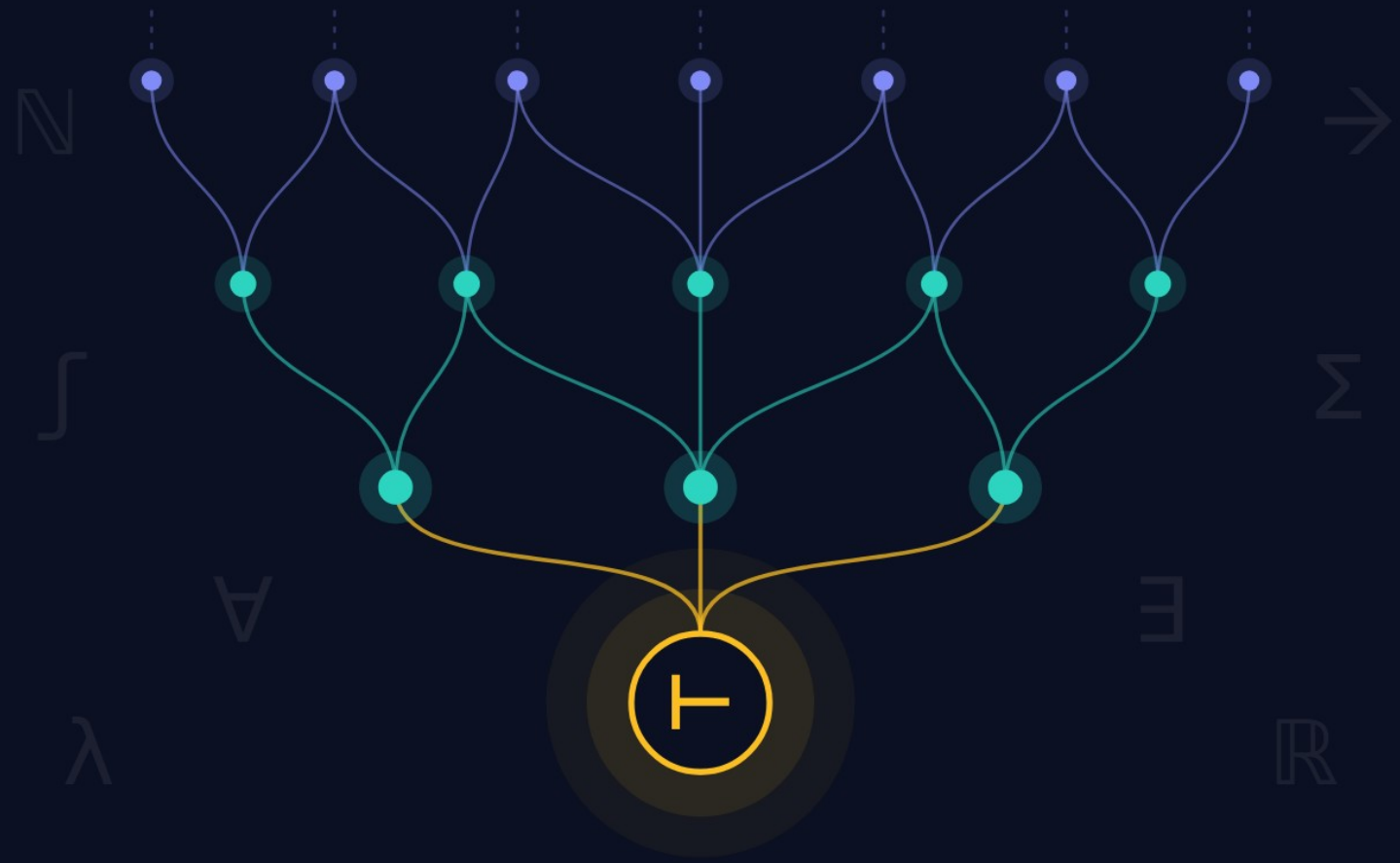
LEAN 4.35.0

MATHLIB v4.35.0



LEAN 4

From First Proof to Formalized Mathematics

EVERY EXAMPLE COMPILED

Preface

This book teaches Lean 4 from installation to formalized analysis, with every example written for Lean 4.35.0 and the matching Mathlib release.

It is written for a working mathematician or a strong student. It assumes no type theory and no functional programming. The mathematical examples lean toward analysis: inequalities, sums, limits, derivatives and integrals.

How to read it

Read Parts I and II in order, with the editor open, typing the examples rather than pasting them. Part III can be read by need once Chapters 12 and 13 are done. Parts IV and V are independent of each other.

Part	Chapters	After it you can
I. Getting Oriented	1–4	Run Lean in a well-configured editor and read the Infoview
II. The Language and Its Logic	5–11	Write definitions and prove statements by hand, in term style and tactic style
III. Mathematics with Mathlib	12–20	Find lemmas, drive the automation, and formalize analysis up to measure theory
IV. Lean as a Programming Language	21–23	Write and verify programs, and write your own tactics
V. Working Practice	24–27	Maintain a project, debug proofs, use AI tools, and contribute to Mathlib

Conventions

- **Code blocks are read in order.** A block that begins with an `import` line starts a new file. A block without one continues the file above it, and uses its imports, `open` lines, variables and definitions. In Chapters 7 to 20, the body of each chapter is one file that begins with `import Mathlib`, and its exercises and solutions are two more. The blocks of Chapters 4 to 6, 21 and 22 run in plain Lean.
- **Files that import Mathlib ask for a module docstring.** In a project made with the `math` template, Mathlib's linters want a comment `/-! … -/` just after the imports. The blocks leave it out, so each such file shows this one warning until you add the comment, or the line `set_option linter.style.header false` after the imports.
- **Worked examples show the proof state.** A comment containing ⊢ gives the goal at that point, exactly as the Infoview displays it. Read the comment, predict the next tactic, then read on.

- **Exercises are Lean statements ending in `sorry`.** Paste the exercise block, replace each `sorry`, and stop when the error and warning marks are gone. Lean tells you at once whether your answer is right.
- **Every chapter ends with solutions.** A solution is one working proof, not the only one. If Lean accepts yours, yours is correct.
- **The main names are linked.** A file name links to that file on GitHub, pinned to the version this book uses. The main lemmas and definitions link to their entry in the Mathlib documentation, whose search box finds any other name. A tool or site links to its home page.
- **Symbols are typed with a backslash.** `\to` gives →, `\R` gives ℝ. Each symbol's abbreviation is given at first use, and Appendix B lists them all.

Versions

The book uses Lean 4.35.0, released on 12 October 2026, and Mathlib at its `v4.35.0` tag. That Mathlib snapshot has 8,624 files and about 2.3 million lines.

Lean ships a new minor version roughly every month, and 4.36.0 was at release-candidate stage when this was written. The language changes slowly, but Mathlib renames lemmas and moves files all the time. When a name in this book no longer resolves, Chapter 12 shows how to find its replacement in under a minute. The release notes list what changed in each version.

How the examples were checked

Every Lean example, exercise and solution in this book was compiled with Lean 4.35.0 against Mathlib at the tag `v4.35.0`, and every message quoted, from Chapter 4 to Appendix C, is the one Lean printed for that example. The examples that are meant to fail were run too.

Five limits apply, and you should know them.

- **Targeted imports.** Mathlib was compiled from source in a machine that could not reach Mathlib's download cache. The examples were therefore checked with the specific files they need, such as `import Mathlib.Analysis.SpecialFunctions.Log.Basic`, rather than with `import Mathlib`, and with the options of the `math` project template. With the whole library imported, a bare `simp` may occasionally use a different lemma; the statements and the other tactics behave the same.
- **Network commands were not run.** `lake exe cache get`, `#loogle`, `#leansearch` and the search sites of Chapter 12 need the internet. Their behaviour is described from their documentation.
- **The editor was not driven by hand.** The VS Code steps of Chapters 2 and 3 follow the extension's manual and the official installation pages.

- **One framework is in transition.** The loop verification example of Chapter 22 uses `mvcgen`, which Lean 4.35.0 deprecates in favour of the experimental `vcgen`; it compiles with a warning.
- **Tools change.** Chapter 26 describes the AI tools as their own pages presented them in November 2026.

Mathlib changes every day. If an example fails in a newer version, the most likely cause is a renamed lemma, and Chapter 24 shows how to find the new name.

Authorship and licence

This book was written by Claude, an AI model made by Anthropic. Dr. Chadi SABA conceived the project, guided the writing through successive prompts, chose what to keep, and takes responsibility for the published text.

Dr. SABA is a mathematician whose research centres on harmonic and non-harmonic trigonometric polynomials. He received his PhD from the University of Bordeaux in 2024, with a thesis on the Littlewood problem and non-harmonic Fourier series. Since September 2025 he has been teaching in the Department of Engineering at the University of Limoges. He created this book to learn Lean and Mathlib himself, and he is its first reader.

This book is free. You may read, copy, print and share it, unchanged and with this notice, but you may not sell it or share a modified version (licence CC BY-NC-ND 4.0, creativecommons.org/licenses/by-nc-nd/4.0). The latest version is available on Dr. SABA's personal website. To report an error or suggest a change, you can contact Dr. SABA here.

How to cite this book

L∃∀N 4: From First Proof to Formalized Mathematics, by Claude, supervised by Chadi SABA, version of November 2026, https://csaba.perso.math.cnrs.fr/.

For LaTeX, the BibTeX entry is:

```
@misc{lean4firstproof,
  title  = {L$\exists\forall$N 4: From First Proof
            to Formalized Mathematics},
  author = {{Claude (Anthropic)}},
  note   = {Supervised by Chadi SABA. Version of November 2026},
  year   = {2026},
  howpublished = {\url{https://csaba.perso.math.cnrs.fr/}}
}
```

Contents

Preface **2**

Part I — Getting Oriented **7**

Chapter 1: What Lean Is 7

Chapter 2: Installing Lean and Starting a Project 11

Chapter 3: Setting Up VS Code Properly 16

Chapter 4: Your First Hour in the Editor 23

Part II — The Language and Its Logic **29**

Chapter 5: Terms, Types and Functions 29

Chapter 6: Propositions and Proofs 35

Chapter 7: Tactics, the Core Toolkit 41

Chapter 8: Equality, Rewriting and calc 50

Chapter 9: Inductive Types, Recursion and Induction 57

Chapter 10: Structures and Type Classes 65

Chapter 11: Dependent Type Theory, One Level Deeper 74

Part III — Mathematics with Mathlib **83**

Chapter 12: Finding Things in Mathlib 83

Chapter 13: Automation 91

Chapter 14: Numbers, Casts and Their Traps 103

Chapter 15: Sets, Functions, Finsets and Sums 112

Chapter 16: Algebraic Structures 122

Chapter 17: Filters, Limits and Topology 130

Chapter 18: Calculus 142

Chapter 19: Measure Theory and Integration 152

Chapter 20: A Worked Project in Analysis 162

Part IV — Lean as a Programming Language **169**

Chapter 21: Functional Programming and Effects 169

Chapter 22: Verified Programs 178

Chapter 23: Metaprogramming 185

Part V — Working Practice **195**

Chapter 24: Projects, Lake, and Keeping Up with Mathlib 195

Chapter 25: Proof Engineering 203

Chapter 26: Lean and AI Tools 212

Chapter 27: Community, Contributing, and Where to Go Next 219

Appendices.....224
Appendix A: Which Tactic When.....224
Appendix B: Unicode Abbreviations.....228
Appendix C: Common Error Messages.....234
Appendix D: Glossary.....250
Appendix E: Every Tactic, Alphabetically.....256

Part I — Getting Oriented

Chapter 1: What Lean Is

Lean is a programming language in which mathematical statements are types and proofs are programs, so the type checker that validates a program also validates a proof.

One system, two uses

Lean 4 is a functional programming language. You define functions, run them, and compile them to native code. Lean itself is written in Lean.

Lean 4 is also a proof assistant. You state a theorem and build a proof, and a small program called the kernel checks every step.

The two uses are one activity. Here are a function, a computation and a theorem about the function:

```
def double (n : Nat) : Nat := 2 * n

#eval double 21        -- 42

theorem double_eq (n : Nat) : double n = n + n := by
  unfold double
  omega
```

`def` introduces a definition and `theorem` introduces a proved statement. `#eval` runs code. The lines after `by` are a tactic proof: `unfold double` replaces `double n` by its definition, and `omega` closes the arithmetic goal `2 * n = n + n`.

Proofs are terms

A proposition in Lean is a type, and a proof of it is a term of that type. Proving a theorem means building a term. Checking a proof means type checking.

```
theorem swap (p q : Prop) (h : p ∧ q) : q ∧ p := ⟨h.2, h.1⟩
```

Here `h` is a proof of `p ∧ q`. It has two components, `h.1 : p` and `h.2 : q`. The proof of `q ∧ p` is the same pair in the other order. No tactic is involved: tactics are programs that write such terms for you.

Who checks what

Three components handle your file, in this order.

1. The **parser** turns text into syntax. Lean's syntax is extensible, which is how Mathlib adds notation such as `∑ i ∈ s, f i`.
2. The **elaborator** turns syntax into fully explicit terms. It infers types, fills in implicit arguments, finds type class instances and runs tactics.

3. The **kernel** type checks the resulting terms. It knows nothing about tactics or notation.

Only the kernel has to be trusted. A bug in a tactic can make a proof fail, but it cannot make a false statement pass, because the kernel rechecks the term the tactic produced.

The kernel is small, and independent reimplementations exist to recheck proofs. The 4.35.0 toolchain ships two of them in its `bin` folder, `lean4lean` and `nanoda_bin`. The community page Did you really prove it? explains how to audit a claimed proof.

Axioms

Lean's logic is a dependent type theory with three axioms added on top.

Axiom	What it says	Declared in
`propext`	Equivalent propositions are equal	`Init/Core.lean`
`Quot.sound`	Related elements become equal in a quotient	`Init/Prelude.lean`
`Classical.choice`	From the fact that a type is nonempty, you get an element	`Init/Prelude.lean`

The law of excluded middle is a theorem derived from these three. Mathlib uses all of them freely, so Mathlib mathematics is classical mathematics.

The command `#print axioms` reports which axioms a theorem depends on:

```
#print axioms double_eq   -- 'double_eq' depends on axioms: [propext,
Quot.sound]
#print axioms swap        -- 'swap' does not depend on any axioms
```

An unfinished proof can be closed with `sorry`. Lean accepts it with a warning and records the axiom `sorryAx`, so `#print axioms` exposes any gap left in a development.

Mathlib

Mathlib is the community's mathematics library: one repository, one set of conventions, one definition of each concept. It covers algebra, analysis, topology, measure theory, probability, number theory, geometry, combinatorics and category theory.

Three pages show what is in it: the overview by topic, the undergraduate curriculum list, and the live statistics.

Mathlib states each result in the greatest generality its authors could manage. Limits are defined with filters, the integral is the Bochner integral, and the derivative is the Fréchet derivative. Much of Part III is about reading that generality comfortably.

What working in Lean feels like

You state a goal, Lean shows what is left to prove, and you transform the goal step by step until nothing is left.

```
import Mathlib

example (a b : ℝ) : 2 * a * b ≤ a ^ 2 + b ^ 2 := by
  have h : 0 ≤ (a - b) ^ 2 := sq_nonneg (a - b)
  nlinarith
```

With the cursor at the end of the `have` line, the Infoview panel beside the editor shows the proof state:

```
a b : ℝ
h : 0 ≤ (a - b) ^ 2
⊢ 2 * a * b ≤ a ^ 2 + b ^ 2
```

The lines above `⊢` are what you know. The line after it is what you must prove. `sq_nonneg` is a Mathlib lemma, and `nlinarith` is a tactic for nonlinear arithmetic that finds the right combination of the hypotheses.

This loop of reading the state, choosing a step and reading the new state is the whole activity. The rest of the book is vocabulary for it.

What Lean is not

- **Not an automatic prover.** Automation closes routine goals. The structure of the proof comes from you.
- **Not a computer algebra system.** `ℝ` is the real numbers, not floating point, and there is no `solve` command.
- **Not a judge of your definitions.** Lean checks that the proof proves the statement. Whether the statement says what you meant is your responsibility, and most real mistakes in formalization are wrong statements.

The pieces and their names

Name	What it is
Lean 4	The language, compiler and kernel, developed by the Lean FRO
elan	The version manager. It installs whichever Lean version each project asks for
Lake	The build tool and package manager. It ships with Lean
`Std`	Lean's standard library of data structures. It ships with Lean
Batteries	A community extension of the standard library
Mathlib	The mathematics library

Name	What it is
Aesop	A proof search tactic used throughout Mathlib
ProofWidgets	Interactive displays inside the Infoview
Reservoir	The index of Lean packages
Zulip	The community chat, where questions get answered within hours

Exercises

All four can be done in the web editor with nothing installed.

1. Delete the editor's contents and type `#eval 2 ^ 100`. What appears on the right?
2. Paste the `double` block from this chapter, then delete the line `omega`. What does the error say is left to prove?
3. Replace the whole proof of `double_eq` after `by` with `sorry`, and add `#print axioms double_eq` below. What does Lean report?
4. Define `triple` and prove `triple n = n + n + n` by imitating `double`.

Solutions

1. `1267650600228229401496703205376`. Lean's `Nat` has arbitrary precision, so nothing overflows.
2. The error is `unsolved goals`, followed by the state `n : Nat ⊢ 2 * n = n + n`. `unfold` did its job and nothing closed the remaining goal.
3. A warning, `declaration uses `sorry``, and then `'double_eq' depends on axioms: [sorryAx]`.
4. The same two tactics work:

```
def triple (n : Nat) : Nat := 3 * n

theorem triple_eq (n : Nat) : triple n = n + n + n := by
  unfold triple
  omega
```

Chapter 2: Installing Lean and Starting a Project

Installation is three steps: install VS Code, install the Lean 4 extension, and let the extension install everything else.

What ends up on your machine

Component	Role	Where it lives
VS Code	The editor	Standard application folder
Lean 4 extension	Connects VS Code to Lean, shows the Infoview	Inside VS Code
Git	Downloads dependencies, which are Git repositories	System
elan	Picks and installs the right Lean version per project	`~/.elan` (Windows: `%USERPROFILE%\.elan`)
Lean toolchains	One full copy of Lean per version in use	`~/.elan/toolchains`
Mathlib	Source and compiled files, one copy per project	`.lake/packages/mathlib` in the project

You never install "Lean" as one global program. You install elan, and elan fetches the version each project asks for.

The recommended route

1. Install VS Code.
2. Open the Extensions view with `Ctrl+Shift+X` (`Cmd+Shift+X` on macOS), search for `lean4`, and install the extension published by `leanprover`.
3. A guide called **Lean 4 Setup** opens. If it does not, click the **∀** symbol at the top right of any editor tab and choose **Documentation... > Show Setup Guide**.
4. In the guide, open the step **Install Lean** and click the **Install Lean** button. This installs elan and Git. On macOS it installs Apple's Command Line Tools, which contain Git.
5. Open the step **Set Up Lean 4 Project** and click **Create a new project using Mathlib**. Choose a parent folder and a project name.

The extension then creates the project, downloads Mathlib with its precompiled files, and opens the folder. The download is several gigabytes, so the first project takes a few minutes.

The official install page describes the same route, and the extension's guide explains what each button runs.

Where to put the project

The setup guide warns against network drives and folders synchronized by cloud storage such as OneDrive. A project holds tens of thousands of small files, and synchronization tools slow Lean down or lock files in the middle of a build.

As a precaution, also use a path with no spaces and no accented characters, for example `C:\lean\MyProject` or `~/lean/MyProject`.

The command-line route

Use this if you prefer a terminal or the button fails. The commands come from the manual installation page.

Linux:

```
sudo apt install git curl
curl https://elan.lean-lang.org/elan-init.sh -sSf | sh
source $HOME/.elan/env
code --install-extension leanprover.lean4
```

macOS, where `git --version` triggers the installation of the Command Line Tools if they are missing:

```
git --version
curl https://elan.lean-lang.org/elan-init.sh -sSf | sh
source $HOME/.elan/env
code --install-extension leanprover.lean4
```

Windows, in a terminal, with Git for Windows already installed:

```
curl -O --location https://elan.lean-lang.org/elan-init.ps1
powershell -ExecutionPolicy Bypass -f elan-init.ps1
del elan-init.ps1
```

Then create a project that depends on Mathlib:

```
lake +leanprover-community/mathlib4:lean-toolchain new MyProject math
cd MyProject
lake exe cache get
lake build
```

The `+leanprover-community/mathlib4:lean-toolchain` part tells elan to use the Lean version that Mathlib currently requires. `lake exe cache get` downloads Mathlib's compiled files, so nothing in Mathlib is compiled on your machine.

Lake offers five project templates. `lake new --help` lists them.

Template	Creates
`std`	A library and an executable. The default
`exe`	An executable only
`lib`	A library only

Template	Creates
`math-lax`	A library with a Mathlib dependency
`math`	A library with a Mathlib dependency and Mathlib's linting and CI standards

What is in the project

```
MyProject/
├── .github/workflows/    continuous integration, used if you push to GitHub
├── .gitignore            ignores .lake
├── .lake/                downloaded packages and build output
├── MyProject.lean        root of your library: imports your modules
├── MyProject/
│   └── Basic.lean        your first module
├── README.md
├── lake-manifest.json    the exact version of every dependency
├── lakefile.toml         the project's configuration
└── lean-toolchain        the Lean version this project uses
```

The configuration file written by the `math` template looks like this:

```
name = "MyProject"
version = "0.1.0"
keywords = ["math"]
defaultTargets = ["MyProject"]

[leanOptions]
pp.unicode.fun = true # pretty-prints `fun a ↦ b`
relaxedAutoImplicit = false
weak.linter.mathlibStandardSet = true
maxSynthPendingDepth = 3

[[require]]
name = "mathlib"
scope = "leanprover-community"
rev = "v4.35.0"

[[lean_lib]]
name = "MyProject"
```

The `[[require]]` block is the dependency on Mathlib, and `[[lean_lib]]` declares your own library. The `[leanOptions]` block turns on Mathlib's conventions for your files too. Chapter 24 goes through this file line by line.

Your own files go in the `MyProject/` folder. A file `MyProject/Limits.lean` is the module `MyProject.Limits`, and other files load it with `import MyProject.Limits`.

How versions work

The file `lean-toolchain` holds one line, such as `leanprover/lean4:v4.35.0`. Whenever `lean` or `lake` runs inside the project, elan reads that line and uses exactly that version, downloading it first if needed.

Mathlib compiles with one Lean version only: the one named in its own `lean-toolchain`. Your project's file must name the same version. The project creation commands above arrange this, and Chapter 24 covers updating both together.

Command	Effect
`elan show`	Lists installed toolchains and the one active in the current folder
`elan default stable`	Sets the version used outside any project
`elan self update`	Updates elan itself
`elan toolchain uninstall <name>`	Removes an old toolchain to free disk space

Checking the installation

Open a terminal inside VS Code with **Terminal > New Terminal** and run:

```
lean --version
lake --version
```

The answers look like `Lean (version 4.35.0, …)` and `Lake version 5.0.0-src+eb6cafb (Lean version 4.35.0)`.

Then create the file `MyProject/Test.lean` with this content:

```
import Mathlib

#check TopologicalSpace

example : (2 : ℝ) + 2 = 4 := by norm_num
```

An orange bar in the scroll bar shows Lean loading Mathlib, which takes a few seconds. When it disappears, there should be no red underline, and placing the cursor on the `#check` line shows a type in the Infoview. The installation works.

Three first-day problems

`unknown module prefix 'Mathlib'`. Lean was started outside the project. Use **File > Open Folder…** and select the folder that contains `lakefile.toml`, not a single file and not a parent folder.

Lean starts compiling Mathlib. The terminal or the Infoview shows hundreds of `Building Mathlib.…` lines. The compiled files are missing. Stop the build, choose **∀ > Project Actions… > Fetch Mathlib Build Cache** or run `lake exe cache get`, then run **∀ > Server: Restart File**.

Downloads fail with a certificate error. The setup guide attributes this to antivirus software that scans encrypted connections. Disable that feature for the duration of the download.

If something else goes wrong, **∀ > Troubleshooting: Show Setup Information** produces a report to paste into the #new members channel on Zulip.

Working without installing anything

- The web editor runs current Lean with Mathlib in the browser. It is enough for every exercise in Parts I and II.
- The Lean Game Server hosts the Natural Number Game, a guided first contact with tactic proofs.
- The repository of Mathematics in Lean can be opened in a cloud workspace, which the setup guide recommends for locked-down computers.

Exercises

1. Which Lean version does your project use, and which file says so?
2. Create `MyProject/Scratch.lean` containing `import Mathlib` and `#eval (2 : ℚ) / 3 + 1 / 6`. What is displayed?
3. Mathlib's source is on your disk. Find and open the file that defines the real numbers. Can you edit it?
4. Close VS Code, reopen it without opening a folder, and open `Test.lean` alone with **File > Open File...**. What happens, and how do you recover?

Solutions

1. The file `lean-toolchain` at the project root. `elan show` or `lean --version` in the project's terminal confirms it.
2. `5 / 6`. Lean computes with exact rationals. Type ℚ as `\Q`.
3. It is `.lake/packages/mathlib/Mathlib/Basic/Real/Basic.lean`, also visible on GitHub. VS Code opens it read-only: the extension protects everything under `.lake` so that you cannot break a dependency by accident.
4. The first line is underlined in red with `unknown module prefix 'Mathlib'`. Recover with **∀ > Open Project... > Open Local Project...** and select the project folder.

Chapter 3: Setting Up VS Code Properly

Ten minutes of configuration removes most of the daily friction: one extra extension, a block of settings, a font, a window layout and a dozen shortcuts.

The checklist

- ☐ Trust the project folder when VS Code asks
- ☐ Install Error Lens
- ☐ Paste the settings block into your user settings
- ☐ Install a font with full mathematical coverage
- ☐ Arrange the window: code on the left, Infoview on the right
- ☐ Set the Infoview options and save them as defaults
- ☐ Check Unicode input, and move the leader key if backslash is awkward on your keyboard
- ☐ Learn the shortcuts in the table
- ☐ Add the theorem snippet

Each item is explained below, in this order.

Trust the folder

When a project is opened for the first time, VS Code asks whether you trust the authors of its files. Choose **Yes, I trust the authors**. Lean does not start in an untrusted folder.

What the Lean extension already configures

The extension sets these for Lean files, so there is nothing to do. The extension manual recommends leaving them alone.

Setting	Value in Lean files	Reason
Indentation	2 spaces, never tabs	Lean's layout is indentation sensitive
Encoding and line endings	UTF-8, `\n`	Same files on every operating system
Trailing whitespace	Trimmed on save, one final newline	Mathlib's style
Format on save, paste and type	Off	Lean has no automatic formatter
Ambiguous Unicode highlighting	Off	Otherwise VS Code boxes every `∧` and `|`

Setting	Value in Lean files	Reason
Files under `.lake` and `.elan`	Read-only	Protects dependencies from accidental edits

To see or change settings that apply only to Lean files, open the Settings page with `Ctrl+,` and type `@lang:lean4` in its search bar.

Extensions

Extension	Verdict	What it adds
Lean 4	Required	Everything: the Lean server, the Infoview, Unicode input, project commands
Error Lens	Recommended	Prints each error, warning and message at the end of its own line. The result of `#eval` and `#check` appears beside the command. Lean's own manual recommends it
Paperproof	Optional, later	Draws a tactic proof as a tree of hypotheses and goals. Worth trying once you reach Chapter 7

Nothing else is needed. Git support is built into VS Code.

If you use an AI completion extension, switch off its inline suggestions for Lean files while you work through Parts I and II. You learn the tactics by choosing them yourself. Chapter 26 covers AI tools.

Settings to paste

Open the command palette with `Ctrl+Shift+P` (`Cmd+Shift+P` on macOS), run **Preferences: Open User Settings (JSON)**, and merge these lines inside the outer braces.

```
{
  // Enter always means "new line". Accept a completion with Tab.
  "editor.acceptSuggestionOnEnter": "off",
  // No completions made of random words from open files.
  "editor.wordBasedSuggestions": "off",

  // Mathlib's line limit is 100 characters.
  "editor.rulers": [100],
  // The minimap takes width that the Infoview needs.
  "editor.minimap.enabled": false,
  // Colour each identifier by its role: variable, function, keyword.
  "editor.semanticHighlighting.enabled": true,

  // A font in which every mathematical symbol has the same width.
  "editor.fontFamily": "'JuliaMono', 'DejaVu Sans Mono', monospace",
  "editor.fontSize": 14,

  // Error Lens: wait until typing pauses, and keep inline messages short.
  "errorLens.delay": 1000,
```

```
  "errorLens.messageMaxChars": 200,

  // Save automatically, so that files you import are never stale on disk.
  "files.autoSave": "afterDelay"
}
```

The first two come straight from the extension manual, which calls the default `Enter` behaviour irritating in Lean. The ruler matches Mathlib's own editor settings.

Semantic highlighting depends on the colour theme. The Lean page on semantic highlighting notes that the default light theme colours variables almost like plain text, and shows how to choose your own colours.

These are user settings and apply to every project. To restrict one to a single project, put it in `.vscode/settings.json` inside that project instead. The general mechanism is described in the VS Code settings documentation.

Font

Lean code is full of symbols such as `ℝ ∀ ∃ ↦ ⟨ ⟩ ‖ ∑ ∫ ⁻¹ ₁ 𝒩`. When the editor font lacks a symbol, the system borrows it from another font with a different width, and columns stop lining up.

The Infoview uses the editor's font family and size, as its stylesheet shows. One font choice therefore fixes both panels.

- JuliaMono is free and was designed for broad coverage of mathematical Unicode.
- DejaVu Sans Mono is the long-standing recommendation in the Lean community.

Install the font in the operating system, restart VS Code, and paste this test line into a Lean file. Keep the font in which you can tell `ν`, `ν` and `ν` apart at a glance.

```
-- ν ν ν ∧ ⊢ ↦ ⟨x, y⟩ x⁻¹ x₁ 𝒩 ∑ ∫ ‖x‖ ∀ ε > 0, ∃ δ > 0
```

Window layout

- Keep the code on the left and the Infoview on the right. Drag the divider until the ruler at column 100 is just visible.
- Hide the file explorer with `Ctrl+B` when you are proving. It returns with the same key.
- With two monitors, right-click the Infoview tab and choose **Move into New Window**.
- If the Infoview disappears, `Ctrl+Shift+Enter` brings it back.
- The Problems panel, `Ctrl+Shift+M`, lists every error and warning in the open files.

The breadcrumbs bar and sticky scroll are on by default. Both show which namespace and which theorem the cursor is in, which matters in long files.

Infoview options

The cog icon at the top right of the **Tactic state** section opens the display options. Start with the defaults, then try these two.

- **Display target before assumptions** puts the goal on top. It helps once a proof has twenty hypotheses.
- **Hide instance assumptions** removes lines such as `inst✝ : TopologicalSpace X`. It helps from Chapter 16 on.

When the display suits you, choose **Save current settings to default settings** in the same menu.

Four Infoview features are worth knowing from day one.

- **Hover** over any name or subexpression in the Infoview to see its type and documentation. `Ctrl+click` jumps to its definition.
- **Pin** (top right) keeps the state at the current cursor position on screen while you move elsewhere.
- **Pause** freezes the display, which helps when a slow file keeps refreshing.
- **Colours** show what the last tactic changed: inserted parts in green, parts about to disappear in red.

Unicode input

Type a backslash and an abbreviation. The abbreviation turns into the symbol as soon as it is unambiguous, or when you press `Tab` or space.

Type	Get	Type	Get	Type	Get
`\R`	ℝ	`\to`	→	`\le`	≤
`\N`	ℕ	`\mapsto`	↦	`\ne`	≠
`\forall`	∀	`\iff`	↔	`\in`	∈
`\ex`	∃	`\and`	∧	`\<>`	⟨⟩
`\e`	ε	`\or`	∨	`\norm`	‖‖
`\de`	δ	`\not`	¬	`\\|-`	⊢

Three ways to find an abbreviation you do not know:

- Hover over the symbol in any Lean file. The tooltip lists every abbreviation that produces it.
- Press `Ctrl+Alt+\` and search the symbol picker by name.
- Choose **∀ > Documentation... > Show Unicode Input Abbreviations** for the full table of 1,865 entries. Appendix B has the hundred you will use.

On keyboards where the backslash needs a three-key combination, such as French AZERTY, change the leader. In Settings, search **Lean 4 > Input: Leader** and enter a character you never type in Lean.

You can add abbreviations of your own. Analysts miss one for the lower Lebesgue integral:

```
"lean4.input.customTranslations": { "lint": "∫⁻" }
```

Shortcuts

On macOS, read `Cmd` for `Ctrl` unless a second column entry says otherwise.

Keys	macOS	Action
`Ctrl+Shift+Enter`		Show or hide the Infoview
`Ctrl+Shift+X`		Restart File: reload imports and recheck this file
`Ctrl+Shift+Alt+Enter`		Fold or unfold **All Messages** in the Infoview
`Ctrl+K Ctrl+S`		Search Mathlib with Loogle (Chapter 12)
`Ctrl+Alt+\`		Find a Unicode symbol by name
`Ctrl+Space`	`Option+Esc`	Completion. After a dot, it lists everything that applies
`Ctrl+.`		Code actions: accept a `Try this` suggestion
`F12` or `Ctrl+click`		Go to definition
`Alt+←`	`Ctrl+-`	Go back to where you were
`Ctrl+P`		Open a file by name
`Ctrl+T`		Find a declaration by name in the whole project
`Ctrl+Shift+O`		Jump to a declaration in this file
`Ctrl+Shift+P`		Command palette: every command by name

Outside a Lean file, `Ctrl+Shift+X` opens the Extensions view. Inside one, the Lean extension takes the key for Restart File.

Every Lean command is also in the ∀ menu at the top right of the editor.

Searching Mathlib's text

The project search, `Ctrl+Shift+F`, skips Mathlib by default. Mathlib sits in `.lake`, which the project's `.gitignore` excludes, and the search respects that file.

To search Mathlib's source, click the **…** under the search box, enter `.lake/packages/mathlib/Mathlib` in **files to include**, and switch off the cog button **Use Exclude Settings and Ignore Files**. Chapter 12 presents faster tools for finding lemmas.

A snippet for theorem skeletons

A snippet expands a short prefix into a template. Create the file `.vscode/lean.code-snippets` in your project with this content. Mathlib does the same for its file headers in its own snippet files.

```
{
  "Theorem skeleton": {
    "scope": "lean4",
    "prefix": "thm",
    "body": [
      "theorem ${1:name} : ${2:statement} := by",
      "  ${0:sorry}"
    ]
  }
}
```

Typing `thm` and accepting the completion now inserts the skeleton. `Tab` moves from the name to the statement to the proof.

Exercises

1. Type this line by hand in `MyProject/Test.lean`, below the import. Which abbreviations did you use?

```
example : ∀ ε : ℝ, ε > 0 → ∃ δ > 0, δ ≤ ε := fun ε hε ↦ ⟨ε, hε, le_refl ε⟩
```

2. Hover over `↦` in that line. Which abbreviations produce it?
3. Put the cursor just after `↦`. What does the Infoview show, and under which heading?
4. Use the symbol picker to find how to type `∂` and `∫`.
5. Remove the final `⟩`. Where does the error appear with Error Lens installed, and where without it?

Solutions

1. `\forall`, `\e`, `\R`, `\to`, `\ex`, `\de`, `\le`, `\mapsto` and `\<>`. `fun` is typed as plain letters, and `hε` is `h` followed by `\e`.
2. `\mapsto`, `\ma`, `\|->` and `\r-|`.
3. Under **Expected type**, it shows `ε : ℝ` and `hε : ε > 0` above the line `⊢ ∃ δ > 0, δ ≤ ε`. Lean is telling you what the term after `↦` must prove.
4. `∂` is `\partial` or `\pd`. `∫` is `\integral`. Note that `\int` gives `ℤ`.

5. With Error Lens, the message is printed in red at the end of the line. Without it, the line only has a red underline, and the message is in the Infoview under **Messages** and in the Problems panel.

Chapter 4: Your First Hour in the Editor

Lean checks the file continuously while you type, and everything it has to say appears in the Infoview, for the position where the cursor is.

Everything in this chapter runs in plain Lean. Use a new file `MyProject/FirstHour.lean` with no import line.

The file is live

There is no run button. Lean reads the file from top to bottom and checks each command as soon as you stop typing. Changing a line makes Lean recheck everything below it.

What you see	Meaning
Orange bar in the scroll bar	Lean is still checking that part
Blue underline	A message: the output of `#eval` or `#check`
Orange underline	A warning, for example a proof containing `sorry`
Red underline	An error
Double check mark in the margin	That theorem is fully proved

Asking Lean questions

Three commands ask questions without defining anything. Put the cursor on the line to see the answer in the Infoview.

Command	Question it asks
`#eval e`	What is the value of `e`?
`#check e`	What is the type of `e`?
`#print c`	How is `c` defined?

```
#eval 1 + 1                        -- 2
#eval "Hello, " ++ "Lean"          -- "Hello, Lean"
#eval [1, 2, 3].map (· * 2)        -- [2, 4, 6]
#eval 2 ^ 10                       -- 1024
#eval 7 / 2                        -- 3
#eval 2 - 5                        -- 0
#eval (2 - 5 : Int)                -- -3
```

The last three lines show the first trap of the language. A bare numeral is a natural number, and on `Nat` division rounds down and subtraction stops at zero. Chapter 14 is devoted to this.

```
#check 1 + 1                       -- 1 + 1 : Nat
#check (1 + 1 : Int)               -- 1 + 1 : Int
#check 1 + 1 = 2                   -- 1 + 1 = 2 : Prop
```

```
#check Nat.succ                  -- Nat.succ (n : Nat) : Nat
#check Nat.add_comm              -- Nat.add_comm (n m : Nat) : n + m = m + n
#check fun (n : Nat) => n + 1    -- fun n ↦ n + 1 : Nat → Nat
```

Read `e : T` as "`e` has type `T`". The third line says that the statement `1 + 1 = 2` is a proposition. The fifth says that `Nat.add_comm` takes two numbers and returns a proof of an equation.

```
#print Nat
-- inductive Nat : Type
-- number of parameters: 0
-- constructors:
-- Nat.zero : Nat
-- Nat.succ : Nat → Nat
```

Statements that need a proof

```
example : 1 + 1 = 2 := rfl

theorem one_add_one : 1 + 1 = 2 := rfl
```

`example` states and proves something without naming it. Lean checks it and forgets it. `theorem` gives the result a name so that later proofs can use it. The term `rfl` proves any equation whose two sides compute to the same value.

A statement with hypotheses has this anatomy:

```
theorem my_first (a b : Nat) (h : a = b) : b + 0 = a := by ...
        ──┬──── ─────┬─────── ─────┬─────   ───┬───    ──┬──
          name     variables   hypothesis   statement    proof
```

Everything before the colon is given. What follows the colon is to be proved. What follows `:=` is the proof.

A first tactic proof, one step at a time

```
theorem my_first (a b : Nat) (h : a = b) : b + 0 = a := by
  rw [Nat.add_zero]
  rw [h]
```

Put the cursor at the end of the first line, just after `by`. The Infoview shows the starting state:

```
a b : Nat
h : a = b
⊢ b + 0 = a
```

The lemma `Nat.add_zero` states `n + 0 = n`. The tactic `rw [Nat.add_zero]` finds the left side of that equation in the goal and replaces it by the right side. With the cursor at the end of the second line, the state is:

```
a b : Nat
h : a = b
⊢ b = a
```

`rw [h]` uses the hypothesis `h : a = b` the same way, replacing `a` by `b`. The goal becomes `b = b`. `rw` closes such a goal by itself, and the Infoview reports that no goals remain.

Two habits to form now:

- **The cursor decides what you see.** At the end of a tactic line you see the state after that tactic. Move up and down a proof with the arrow keys and watch the goal change.
- **Read the state before choosing a tactic.** The hypotheses tell you what you may use. The line after `⊢` tells you what shape of tactic to reach for.

Several goals and bullets

```
example (p q : Prop) (hp : p) (hq : q) : p ∧ q := by
  constructor
  · exact hp
  · exact hq
```

To prove a conjunction you prove both parts. `constructor` splits the goal `p ∧ q` into two goals, and the Infoview shows both, each with a name:

```
case left
p q : Prop
hp : p
hq : q
⊢ p

case right
p q : Prop
hp : p
hq : q
⊢ q
```

Each bullet `·`, typed `\.`, focuses on the next goal. Inside the first bullet only `⊢ p` is visible, and `exact hp` closes it because `hp` is exactly a proof of `p`. A bullet that ends with its goal still open is an error at that bullet, which keeps mistakes local.

`sorry`, the placeholder

`sorry` proves anything. Lean accepts it with the warning `declaration uses `sorry`` .

```
theorem hard (a b : Nat) : a + b = b + a := by
  sorry
```

It is the main working tool. State the theorem, write `sorry`, check that the statement itself has no error, then replace the `sorry`. In a long proof, write the outline with a `sorry` at each step and fill them one by one.

Asking Lean for the lemma

Replace the `sorry` above by `exact?`. Lean searches the library for a lemma that closes the goal and answers in the Infoview:

```
Try this:
  [apply] exact Nat.add_comm a b
```

Click the suggestion and it replaces `exact?` in your file. Chapter 12 covers this family of search tactics.

Comments

```
-- A comment to the end of the line.

/- A block comment,
   over several lines. -/

/-- A documentation comment. It belongs to the declaration that follows
and appears when you hover over its name. -/
theorem documented : 1 + 1 = 2 := rfl
```

Reading your first errors

Fix the first error in the file first. Later errors are often its consequences.

Message begins	You wrote	What it means
`Unknown identifier `foo``	`#check foo`	No such name here: a typo, a missing import, or a name inside a namespace that is not open
`Type mismatch … has type String … but is expected to have type n = n + 0`	`example (n : Nat) : n = n + 0 := "hello"`	The term you gave has a different type from the one required
`unsolved goals` followed by a state	A proof that stops too early	The displayed goal is still open at the end of the proof or bullet
`Tactic `rfl` failed: The left-hand side a + b is not definitionally equal to the right-hand side b + a`	`example (a b : Nat) : a + b = b + a := by rfl`	`rfl` only proves what holds by pure computation. This needs a lemma
`Tactic `decide` proved that the proposition 2 + 2 = 5 is false`	`example : 2 + 2 = 5 := by decide`	The statement is wrong

Message begins	You wrote	What it means
`No goals to be solved`	A tactic after the proof is finished	Delete the extra tactic

Appendix C collects the common error messages with their causes and fixes.

Exercises

Paste this block into your file and replace each `sorry`. The double check mark in the margin tells you when a proof is complete.

```
-- Exercise 1. Predict each result before looking at the Infoview.
#eval 17 / 5
#eval 17 % 5
#eval 3 - 7
#eval (3 - 7 : Int)

-- Exercise 2. Replace each `sorry` by `rfl` or by `by decide`.
-- One of the three accepts only `by decide`. Which, and why?
example : 2 * 3 = 6 := sorry
example : "ab" ++ "c" = "abc" := sorry
example : 10 < 100 := sorry

-- Exercise 3. Two rewrites: the lemma `Nat.mul_one`, then the hypothesis.
example (a b : Nat) (h : a = b) : a * 1 = b := by
  sorry

-- Exercise 4. Two goals, two bullets.
example (p q : Prop) (hp : p) (hq : q) : q ∧ p := by
  sorry

-- Exercise 5. Let `exact?` find the lemma. Then look at it with `#check`.
example (a b c : Nat) : a * (b + c) = a * b + a * c := by
  sorry

-- Exercise 6. This statement is false. Try `decide` and read the error.
example : 2 + 2 = 5 := by
  sorry
```

Solutions

```
-- Exercise 1.
#eval 17 / 5            -- 3
#eval 17 % 5            -- 2
#eval 3 - 7             -- 0
#eval (3 - 7 : Int)     -- -4

-- Exercise 2. `10 < 100` is not an equation, so `rfl` does not apply.
-- `decide` evaluates the statement and finds it true.
example : 2 * 3 = 6 := rfl
example : "ab" ++ "c" = "abc" := rfl
example : 10 < 100 := by decide

-- Exercise 3. After the first rewrite the goal is `a = b`.
example (a b : Nat) (h : a = b) : a * 1 = b := by
  rw [Nat.mul_one]      -- ⊢ a = b
```

```
  rw [h]                -- ⊢ b = b, closed automatically

-- Several rewrites can share one pair of brackets.
example (a b : Nat) (h : a = b) : a * 1 = b := by
  rw [Nat.mul_one, h]

-- Exercise 4. The order of the bullets follows the order in `q ∧ p`.
example (p q : Prop) (hp : p) (hq : q) : q ∧ p := by
  constructor
  · exact hq            -- case left:  ⊢ q
  · exact hp            -- case right: ⊢ p

-- Exercise 5. `exact?` answers: exact Nat.mul_add a b c
example (a b c : Nat) : a * (b + c) = a * b + a * c := by
  exact Nat.mul_add a b c

#check Nat.mul_add      -- Nat.mul_add (n m k : Nat) : n * (m + k) = n * m +
n * k

-- Exercise 6. There is no proof. `decide` reports:
--   Tactic `decide` proved that the proposition 2 + 2 = 5 is false
```

Part II — The Language and Its Logic

Chapter 5: Terms, Types and Functions

Every expression in Lean has a type, a function is applied by writing its argument after it, and a function of two arguments is a function that returns a function.

This chapter needs no import. The comment after each command is what the Infoview shows.

Everything has a type

```
#check 3                -- 3 : Nat
#check (3 : Int)        -- 3 : Int
#check "abc"            -- "abc" : String
#check [1, 2, 3]        -- [1, 2, 3] : List Nat
#check (1, "one")       -- (1, "one") : Nat × String
#check Nat → Nat        -- Nat → Nat : Type
#check Nat              -- Nat : Type
#check Type             -- Type : Type 1
```

`Nat → Nat` is the type of functions from `Nat` to `Nat`. Type the arrow as `\to`. `Nat × String` is the type of pairs, with `×` typed as `\x`.

Types are themselves expressions. `Nat` has type `Type`, so a type can be passed to a function like any other value. That is how Lean expresses "for every type α".

Defining and applying functions

```
def square (n : Nat) : Nat := n * n

#eval square 7          -- 49
#check square           -- square (n : Nat) : Nat
```

Application is written `square 7`, with a space and no parentheses. It binds tighter than every operator, which is the most common source of early mistakes:

```
#eval square 3 + 1      -- 10, read as (square 3) + 1
#eval square (3 + 1)    -- 16
```

Functions of several arguments

```
def add (a b : Nat) : Nat := a + b

#check add              -- add (a b : Nat) : Nat
#check add 3            -- add 3 : Nat → Nat
#eval (add 3) 4         -- 7
```

The type of `add` is `Nat → Nat → Nat`, which means `Nat → (Nat → Nat)`. Given one number, `add` returns a function waiting for the second. `add 3 4` is `(add 3) 4`.

Supplying only some arguments is allowed and useful:

```
def addThree : Nat → Nat := add 3

#eval addThree 10       -- 13
```

Anonymous functions

```
#check fun (n : Nat) => n + 1      -- fun n ↦ n + 1 : Nat → Nat
#eval (fun n => n + 1) 5           -- 6
#eval [1, 2, 3].map (fun n => n * n)   -- [1, 4, 9]
```

`fun n => n + 1` is the function that mathematicians write n ↦ n + 1. Lean accepts that arrow too: `fun n ↦ n + 1`, with `↦` typed as `\mapsto`. Mathlib uses `↦` everywhere, and so does this book from Part III on.

Two shorthands appear constantly:

```
#eval [1, 2, 3].map (· + 10)       -- [11, 12, 13]   (· + 10) is fun x => x
+ 10
#eval (square ∘ addThree) 2        -- 25             ∘ is composition, typed
\o
```

A `def` with arguments is only a convenient way to write a `fun`. The two definitions below are the same object, and `rfl` proves it:

```
def square' : Nat → Nat := fun n => n * n

example : square = square' := rfl
```

Local names, conditions and cases

```
def hypot2 (a b : Nat) : Nat :=
  let a2 := a * a
  let b2 := b * b
  a2 + b2

#eval hypot2 3 4        -- 25

def absDiff (a b : Nat) : Nat :=
  if a ≤ b then b - a else a - b

#eval absDiff 3 10      -- 7
#eval absDiff 10 3      -- 7
```

`let` names an intermediate value. `if … then … else` is an expression with a value, not a statement.

A function can also be defined by cases on the shape of its argument. Every natural number is either `0` or `n + 1` for some `n`:

```
def fact : Nat → Nat
  | 0 => 1
  | n + 1 => (n + 1) * fact n

#eval fact 5            -- 120
#eval fact 30           -- 265252859812191058636308480000000
```

`fact` calls itself on a smaller argument. Lean accepts a recursive definition only when it can see that the recursion terminates. Chapter 9 explains the rules.

Implicit arguments

A function that works for every type takes the type as an argument:

```
def ident (α : Type) (a : α) : α := a

#eval ident Nat 5       -- 5
```

Writing `Nat` each time is tedious, since Lean can read it off from `5`. Curly braces make the argument implicit:

```
def ident' {α : Type} (a : α) : α := a

#eval ident' 5                  -- 5
#eval ident' "x"                -- "x"
#eval @ident' Nat 5             -- 5   @ makes every argument explicit
#eval ident' (α := Nat) 5       -- 5   or name the one you want to give
```

Bracket	Name	Who supplies the argument
`(x : α)`	Explicit	You
`{x : α}`	Implicit	Lean, by inferring it from the other arguments
`[inst : C α]`	Instance	Lean, by searching for a registered instance (Chapter 10)

With this you can read any signature in the library:

```
#check List.map
-- List.map.{u_1, u_2} {α : Type u_1} {β : Type u_2} (f : α → β) (l : List
α) : List β
```

`List.map` takes two types that you never write, a function and a list. The `.{u_1, u_2}` part lists universe levels and can be ignored for now.

Namespaces and dot notation

Library names are hierarchical: `Nat.succ`, `List.length`, `Nat.add_comm`. The first component is usually the type the function is about.

If `x` has type `T`, then `x.f` means `T.f x`:

```
#eval [1, 2, 3].length          -- 3   means List.length [1, 2, 3]
#eval (5 : Nat).succ            -- 6   means Nat.succ 5
#eval "abc".length              -- 3   means String.length "abc"
```

This is why typing a dot after an expression and pressing `Ctrl+Space` lists everything you can do with it.

You create a namespace with `namespace … end`, and `open` lets you drop the prefix:

```
namespace Geometry

def area (w h : Nat) : Nat := w * h
def perimeter (w h : Nat) : Nat := 2 * (w + h)

#eval area 3 4                  -- 12   inside the namespace, no prefix

end Geometry

#eval Geometry.area 3 4         -- 12   outside, the full name

open Geometry in
#eval perimeter 3 4             -- 14   open for one command only

open Geometry
#eval area 2 2                  -- 4    open for the rest of the file
```

Variables and sections

`variable` declares arguments once for several definitions. A `section` limits how far the declaration reaches.

```
section
variable (α : Type) (xs : List α)

def twice : List α := xs ++ xs

end

#check twice            -- twice (α : Type) (xs : List α) : List α
#eval twice Nat [1, 2]  -- [1, 2, 1, 2]
```

Lean added to `twice` exactly the variables its body mentions. Mathlib files open with long `variable` lines for this reason. When a lemma seems to have fewer hypotheses than it should, look up the file for a `variable` line.

Pairs

```
def p : Nat × String := (1, "one")

#eval p.1               -- 1
#eval p.2               -- "one"

def swap {α β : Type} (p : α × β) : β × α := (p.2, p.1)

#eval swap (1, "one")   -- ("one", 1)
```

Universes, in one paragraph

`Nat : Type`, `Type : Type 1`, `Type 1 : Type 2`, and so on. The hierarchy exists to avoid paradoxes of the "set of all sets" kind. A definition meant for types at every level is written with `{α : Type u}` after declaring `universe u`. Mathlib writes `{α : Type*}` for the same thing. Read both as "any type".

Errors you will meet here

Message	Typical cause
`Function expected at add 1 2 but this term has type Nat`	Too many arguments: `add 1 2 3`
`failed to synthesize instance of type class HAdd Nat String ?m`	An operator between types that have none: `n + "a"`
`failed to synthesize instance of type class Neg Nat`	A negative natural number: `square (-1)`. Write `(-1 : Int)` and use a function on `Int`
`failed to synthesize instance of type class OfScientific Nat`	A decimal where a natural number is expected: `square 2.5`

Exercises

Replace each `sorry`. The `example` lines are tests: each shows an error until the definition above it is right.

```
def square (n : Nat) : Nat := n * n

-- Exercise 1.
def cube (n : Nat) : Nat := sorry
example : cube 3 = 27 := rfl

-- Exercise 2. The larger of two numbers, with `if … then … else`.
def myMax (a b : Nat) : Nat := sorry
example : myMax 3 7 = 7 := rfl
example : myMax 7 3 = 7 := rfl

-- Exercise 3. Apply a function twice.
def applyTwice {α : Type} (f : α → α) (a : α) : α := sorry
example : applyTwice (· + 3) 10 = 16 := rfl
example : applyTwice (fun s => s ++ "!") "hi" = "hi!!" := rfl

-- Exercise 4. The sum 0 + 1 + … + n, by cases.
def sumTo : Nat → Nat
  | 0 => sorry
  | n + 1 => sorry
example : sumTo 10 = 55 := rfl

-- Exercise 5. Predict the type of each expression, then check.
#check applyTwice (· + 3)
#check @applyTwice Nat
#check applyTwice square

-- Exercise 6. The function n ↦ (n + 1)², built from `square` with `∘`.
def succSquare : Nat → Nat := sorry
example : succSquare 4 = 25 := rfl
```

Solutions

```
def square (n : Nat) : Nat := n * n

-- Exercise 1.
def cube (n : Nat) : Nat := n * n * n
example : cube 3 = 27 := rfl

-- Exercise 2.
def myMax (a b : Nat) : Nat := if a ≤ b then b else a
example : myMax 3 7 = 7 := rfl
example : myMax 7 3 = 7 := rfl

-- Exercise 3. The inner call runs first.
def applyTwice {α : Type} (f : α → α) (a : α) : α := f (f a)
example : applyTwice (· + 3) 10 = 16 := rfl
example : applyTwice (fun s => s ++ "!") "hi" = "hi!!" := rfl

-- Exercise 4. The sum up to n + 1 is (n + 1) plus the sum up to n.
def sumTo : Nat → Nat
  | 0 => 0
  | n + 1 => (n + 1) + sumTo n
example : sumTo 10 = 55 := rfl

-- Exercise 5.
#check applyTwice (· + 3)   -- applyTwice fun x ↦ x + 3 : Nat → Nat
#check @applyTwice Nat      -- applyTwice : (Nat → Nat) → Nat → Nat
#check applyTwice square    -- applyTwice square : Nat → Nat

-- Exercise 6. `(· + 1)` runs first, then `square`.
def succSquare : Nat → Nat := square ∘ (· + 1)
example : succSquare 4 = 25 := rfl
```

Chapter 6: Propositions and Proofs

A proposition is a type, a proof is a term of that type, and each logical connective comes with one way to build a proof of it and one way to use a proof of it.

This chapter writes proofs directly as terms, with no tactics. Proofs are rarely written this way in practice, but tactics only automate what is shown here, and error messages speak this language. No import is needed.

Propositions are types

```
variable (p q r : Prop)

#check p ∧ q                        -- p ∧ q : Prop
#check p → q                        -- p → q : Prop
#check ∀ n : Nat, n + 0 = n         -- ∀ (n : Nat), n + 0 = n : Prop
#check ∃ n : Nat, n * n = 49        -- ∃ n, n * n = 49 : Prop
```

`p : Prop` says that `p` is a proposition. `hp : p` says that `hp` is a proof of `p`. By convention, proofs and hypotheses get names starting with `h`.

The `variable` line makes `p`, `q` and `r` available to every example below.

Implication is a function type

A proof of `p → q` is a function that turns a proof of `p` into a proof of `q`. Using an implication is applying the function. Proving one is writing a `fun`.

```
-- Use: apply hpq to hp.
example (hp : p) (hpq : p → q) : q := hpq hp

-- Prove: assume hp, then produce a proof of r.
example (hpq : p → q) (hqr : q → r) : p → r :=
  fun hp => hqr (hpq hp)
```

Read the second proof from the inside out. `hpq hp` is a proof of `q`, and `hqr` turns it into a proof of `r`.

For all is a function type too

A proof of `∀ n, P n` is a function that takes a number `n` and returns a proof of `P n`.

```
-- Use: apply h to the value you care about.
example (P : Nat → Prop) (h : ∀ n, P n) : P 5 := h 5

-- A hypothesis "for all n, P n implies Q n" takes a number, then a proof.
example (P Q : Nat → Prop) (h : ∀ n, P n → Q n) (h5 : P 5) : Q 5 := h 5 h5

-- Prove: take an arbitrary n.
example : ∀ n : Nat, n + 0 = n := fun n => rfl
```

And is a pair

```
-- Prove: give both proofs between angle brackets, typed \< and \>.
example (hp : p) (hq : q) : p ∧ q := ⟨hp, hq⟩

-- Use: take the first or the second component.
example (h : p ∧ q) : p := h.1
example (h : p ∧ q) : q ∧ p := ⟨h.2, h.1⟩
```

`h.1` and `h.2` can also be written `h.left` and `h.right`.

Or is a choice

```
-- Prove: say which side holds.
example (hp : p) : p ∨ q := Or.inl hp
example (hq : q) : p ∨ q := Or.inr hq

-- Use: handle both cases.
example (h : p ∨ q) : q ∨ p :=
  match h with
  | Or.inl hp => Or.inr hp
  | Or.inr hq => Or.inl hq
```

`Or.inl` is "left injection" and `Or.inr` is "right injection". To use `h : p ∨ q` you do not know which side holds, so you supply a proof of the goal in each case.

Not is an implication to `False`

`¬p` is by definition `p → False`. A proof of `¬p` is therefore a function, and it is applied like one.

```
-- Use: a proof of p and a proof of ¬p give False.
example (hp : p) (hnp : ¬p) : False := hnp hp

-- From False, anything follows.
example (hp : p) (hnp : ¬p) : q := (hnp hp).elim
example (hp : p) (hnp : ¬p) : q := absurd hp hnp

-- Prove: assume p and derive False. This is the contrapositive.
example (hpq : p → q) (hnq : ¬q) : ¬p :=
  fun hp => hnq (hpq hp)
```

Iff is a pair of implications

```
-- Prove: give both directions.
example : p ∧ q ↔ q ∧ p :=
  ⟨fun h => ⟨h.2, h.1⟩, fun h => ⟨h.2, h.1⟩⟩

-- Use: .mp is the forward direction, .mpr the backward one.
example (h : p ↔ q) (hp : p) : q := h.mp hp
example (h : p ↔ q) (hq : q) : p := h.mpr hq
```

The names come from *modus ponens* and *modus ponens reversed*.

Exists is a witness with a proof

```
-- Prove: give the witness and a proof that it works.
example : ∃ n : Nat, n * n = 49 := ⟨7, rfl⟩

-- Use: unpack the witness and its property.
example (P : Nat → Prop) (h : ∃ n, P n ∧ n > 0) : ∃ n, P n :=
  match h with
  | ⟨n, hn, _⟩ => ⟨n, hn⟩
```

In the second example, `h` unpacks into a number `n`, a proof `hn : P n` and a proof of `n > 0`. The underscore discards the part that is not needed.

The whole logic in one table

Statement	To prove it, give	To use a proof `h` of it
`p → q`	A function: `fun hp => …`	Apply it: `h hp`
`∀ x, P x`	A function: `fun x => …`	Apply it to a value: `h a`
`p ∧ q`	A pair: `⟨hp, hq⟩`	`h.1` and `h.2`
`p ∨ q`	`Or.inl hp` or `Or.inr hq`	A `match` with two cases
`p ↔ q`	Two functions: `⟨forward, backward⟩`	`h.mp` and `h.mpr`
`∃ x, P x`	A witness and a proof: `⟨a, ha⟩`	A `match` on `⟨x, hx⟩`
`¬p`	A function to `False`: `fun hp => …`	`h hp` is a proof of `False`
`False`	Nothing proves it	`h.elim` proves any goal
`True`	`trivial`	Carries no information

Chapter 7 gives the same table again with tactics in the cells.

Intermediate steps and holes

`have` names an intermediate result inside a proof:

```
example (hpq : p → q) (hqr : q → r) (hp : p) : r :=
  have hq : q := hpq hp
  hqr hq
```

An underscore asks Lean to fill in a term. When it cannot, the error tells you what is expected there, which makes `_` a way to ask "what do I need here?":

```
example (hpq : p → q) (hqr : q → r) (hp : p) : r :=
  hqr _
-- error: don't know how to synthesize placeholder
-- context:
-- p q r : Prop
-- hpq : p → q
-- hqr : q → r
```

```
-- hp : p
-- ⊢ q
```

Classical reasoning

Everything above is constructive. Proof by contradiction and the excluded middle are available through the `Classical` namespace:

```
example : p ∨ ¬p := Classical.em p

example (h : ¬¬p) : p := Classical.byContradiction h
```

`Classical.byContradiction` has type `(¬p → False) → p`. Since `¬¬p` unfolds to `¬p → False`, the hypothesis `h` fits exactly. Mathlib uses classical logic without comment, and so will you.

A theorem is a function

```
theorem and_of (p q : Prop) (hp : p) (hq : q) : p ∧ q := ⟨hp, hq⟩

#check @and_of     -- and_of : ∀ (p q : Prop), p → q → p ∧ q

example (a b : Prop) (ha : a) (hb : b) : a ∧ b := and_of a b ha hb
```

Arguments before the colon and arrows after it are the same thing. `(hp : p) : q` and `: p → q` state the same theorem. Using a theorem means applying it to arguments, exactly like a function.

This is the single most useful fact for reading Mathlib. A lemma whose statement reads "`(h : a ≤ b) : a + c ≤ b + c`" is a function that you feed a proof of `a ≤ b`.

Errors you will meet here

Message	Typical cause
`don't know how to synthesize placeholder` with a context	You left `_` and Lean cannot guess the term. The line after `⊢` is what is missing
`Application type mismatch: The argument hq has type q but is expected to have type p`	Arguments in the wrong order, as in `⟨hq, hp⟩` for `p ∧ q`
`Insufficient number of fields for` `` `⟨...⟩` `` `constructor`	Too few components between the angle brackets
`Type mismatch` … `has type p but is expected to have type q`	The proof you gave proves something else

Exercises

Replace each `sorry` by a term. No tactics.

```
variable (p q r : Prop)

-- Exercise 1.
example (hp : p) (hq : q) : q ∧ p := sorry

-- Exercise 2. Extract p, then choose the correct side.
example (h : p ∧ q) : p ∨ r := sorry

-- Exercise 3.
example (hpq : p → q) (hqr : q → r) (hp : p) : r := sorry

-- Exercise 4. Nested pairs: h.2.1 is allowed.
example (h : p ∧ (q ∧ r)) : (p ∧ q) ∧ r := sorry

-- Exercise 5.
example (hnq : ¬q) (h : p ∧ q) : False := sorry

-- Exercise 6. Three assumptions to take: fun hpq hnq hp => …
example : (p → q) → (¬q → ¬p) := sorry

-- Exercise 7.
example (P : Nat → Prop) (h : ∀ n, P n) : P 0 ∧ P 1 := sorry

-- Exercise 8.
example : ∃ n : Nat, n + n = 10 := sorry

-- Exercise 9. Unpack h with a match, then build the new witness.
example (P Q : Nat → Prop) (h : ∃ n, P n) (hPQ : ∀ n, P n → Q n) : ∃ n, Q
n := sorry

-- Exercise 10. Two cases.
example (h : p ∨ q) (hpr : p → r) (hqr : q → r) : r := sorry

-- Exercise 11. This direction needs Classical.byContradiction.
example : (¬q → ¬p) → (p → q) := sorry
```

Solutions

```
variable (p q r : Prop)

-- Exercise 1.
example (hp : p) (hq : q) : q ∧ p := ⟨hq, hp⟩

-- Exercise 2. h.1 proves p, and p is the left side of p ∨ r.
example (h : p ∧ q) : p ∨ r := Or.inl h.1

-- Exercise 3.
example (hpq : p → q) (hqr : q → r) (hp : p) : r := hqr (hpq hp)

-- Exercise 4. h.1 : p,  h.2.1 : q,  h.2.2 : r.
example (h : p ∧ (q ∧ r)) : (p ∧ q) ∧ r := ⟨⟨h.1, h.2.1⟩, h.2.2⟩

-- Exercise 5. hnq is a function from q to False.
example (hnq : ¬q) (h : p ∧ q) : False := hnq h.2

-- Exercise 6. The goal ¬p is p → False, hence the third assumption hp.
example : (p → q) → (¬q → ¬p) := fun hpq hnq hp => hnq (hpq hp)

-- Exercise 7.
```

```
example (P : Nat → Prop) (h : ∀ n, P n) : P 0 ∧ P 1 := ⟨h 0, h 1⟩

-- Exercise 8. The witness is 5, and 5 + 5 = 10 holds by computation.
example : ∃ n : Nat, n + n = 10 := ⟨5, rfl⟩

-- Exercise 9. The same n works, with hPQ n hn as the proof.
example (P Q : Nat → Prop) (h : ∃ n, P n) (hPQ : ∀ n, P n → Q n) : ∃ n, Q
n :=
  match h with
  | ⟨n, hn⟩ => ⟨n, hPQ n hn⟩

-- Exercise 10.
example (h : p ∨ q) (hpr : p → r) (hqr : q → r) : r :=
  match h with
  | Or.inl hp => hpr hp
  | Or.inr hq => hqr hq

-- Exercise 11. Assume ¬q. Then h gives ¬p, which contradicts hp.
example : (¬q → ¬p) → (p → q) :=
  fun h hp => Classical.byContradiction (fun hnq => h hnq hp)
```

Chapter 7: Tactics, the Core Toolkit

A tactic proof is a list of commands that transform the goal, and about twenty tactics handle the logical structure of almost every proof.

From this chapter on, every file starts with `import Mathlib`. In the code, a comment after a tactic shows the goal after that tactic has run.

The table again, with tactics

Look at the outermost symbol of the goal, or of the hypothesis you want to use, and read off the tactic.

Statement	When it is the goal	When it is a hypothesis h
`p → q`	`intro hp`	`apply h`, or `h hp` as a term
`∀ x, P x`	`intro x`	`specialize h a`, or `h a` as a term
`p ∧ q`	`constructor`	`obtain ⟨hp, hq⟩ := h`
`p ∨ q`	`left` or `right`	`rcases h with hp | hq`
`p ↔ q`	`constructor`	`h.mp`, `h.mpr`, or `rw [h]`
`∃ x, P x`	`use a`	`obtain ⟨x, hx⟩ := h`
`¬p`	`intro hp`	`h hp` proves `False`
`a ≠ b` (= `¬(a = b)`)	`intro hab`	`h hab` proves `False` (with `hab : a = b`)
`False`		`contradiction`
`True`	`trivial`	nothing to use

Here `a` is a value you choose: in `specialize h a` it is the value you put in place of `x`, and in `use a` it is the witness. Appendix A extends this table to equalities, inequalities, sets, and general moves such as `by_contra` and `by_cases`.

Three basic moves: `intro`, `apply`, `exact`

```
import Mathlib

example (p q r : Prop) (hpq : p → q) (hqr : q → r) : p → r := by
  intro hp        -- hp : p        ⊢ r
  apply hqr       --               ⊢ q
  apply hpq       --               ⊢ p
  exact hp        -- no goals
```

- `intro hp` handles a goal of the form `p → r`. It moves `p` into the hypotheses under the name `hp` and leaves `r` to prove.

- `apply hqr` reasons backwards. Since `hqr : q → r` ends with the goal `r`, it suffices to prove `q`.
- `exact hp` closes a goal with a term that has exactly the goal's type.

The same proof can go forwards, from what is known:

```
example (p q r : Prop) (hpq : p → q) (hqr : q → r) : p → r := by
  intro hp
  have hq : q := hpq hp     -- new hypothesis hq : q
  exact hqr hq
```

Forward steps: `have` and `specialize`

`have name : statement := proof` adds a proved fact to the hypotheses. It is the tactic you will type most.

```
example (f : ℕ → ℕ) (h : ∀ n, f n < f (n + 1)) : f 0 < f 2 := by
  have h0 : f 0 < f 1 := h 0
  have h1 : f 1 < f 2 := h 1
  omega
```

`h 0` is the hypothesis `h` applied to `0`. `omega` then combines the two inequalities.

State the type of a `have` explicitly. Without it, Lean records the fact exactly as produced, here `h0 : f 0 < f (0 + 1)`. `omega` treats `f (0 + 1)` and `f 1` as unrelated quantities and fails.

If the proof of a `have` needs several steps, write `by` and continue with an indented block:

```
  have h0 : f 0 < f 1 := by
    apply h
```

`specialize h 5` replaces a universal hypothesis by one of its instances. Use it when you need `h` for a single value only.

```
example (f : ℕ → ℕ) (h : ∀ n, f n < f (n + 1)) : f 5 < f 6 := by
  specialize h 5        -- h : f 5 < f (5 + 1)
  exact h
```

And: `constructor`, `obtain`

```
example (a b : ℝ) (h : 0 < a ∧ a < b) : 0 < b ∧ a ≠ b := by
  obtain ⟨ha, hab⟩ := h     -- ha : 0 < a,  hab : a < b
  constructor
  · linarith                -- case left:  ⊢ 0 < b
  · exact ne_of_lt hab      -- case right: ⊢ a ≠ b
```

`obtain ⟨ha, hab⟩ := h` splits the hypothesis into its two parts. After it, the state is:

```
a b : ℝ
ha : 0 < a
hab : a < b
⊢ 0 < b ∧ a ≠ b
```

`linarith` proves `0 < b` from `ha` and `hab`. `ne_of_lt` is the lemma `a < b → a ≠ b`.

Three related forms:

- `obtain ⟨hp, -, hr⟩ := h` discards a component you do not need.
- `rintro ⟨hp, hq⟩` is `intro` followed by `obtain`, in one step.
- `exact ⟨hq, hp⟩` proves a conjunction directly when both parts are at hand.

Or: `left`, `right`, `rcases`

```
example (x : ℝ) (h : x = 1 ∨ x = -1) : x ^ 2 = 1 := by
  rcases h with h1 | h2
  · rw [h1]          -- case inl: h1 : x = 1    ⊢ 1 ^ 2 = 1
    norm_num
  · rw [h2]          -- case inr: h2 : x = -1   ⊢ (-1) ^ 2 = 1
    norm_num
```

`rcases h with h1 | h2` creates one goal per case. The vertical bar separates the cases, as the comma separates the parts of a conjunction. Patterns nest: for `h : p ∧ (q ∨ r)`, write `rcases h with ⟨hp, hq | hr⟩`.

To prove a disjunction, choose a side:

```
example (n : ℕ) : n = 0 ∨ n ≥ 1 := by
  by_cases h : n = 0
  · left             -- case pos: h : n = 0     ⊢ n = 0
    exact h
  · right            -- case neg: h : ¬n = 0    ⊢ n ≥ 1
    omega
```

`by_cases h : n = 0` splits the proof in two: one case where `h : n = 0` holds and one where `h : ¬n = 0` holds.

Exists: `use` and `obtain`

```
example : ∀ n : ℕ, ∃ m : ℕ, n < m := by
  intro n           -- n : ℕ    ⊢ ∃ m, n < m
  use n + 1         --          ⊢ n < n + 1
  omega
```

`use` supplies the witness. The goal becomes the property for that witness.

```
example (a : ℝ) (h : ∃ x : ℝ, x > a) : ∃ y : ℝ, y > a - 1 := by
  obtain ⟨x, hx⟩ := h     -- x : ℝ,  hx : x > a
  use x                    -- ⊢ x > a - 1
  linarith
```

`obtain ⟨x, hx⟩ := h` gives you a witness and its property. When the property is an equation such as `n = 3`, write `rfl` in its place, as in `obtain ⟨n, hn, rfl⟩ := h`. Lean then substitutes `3` for `n` everywhere.

For a goal with several parts, `refine` takes a term with holes `?_`, and each hole becomes a goal:

```
example : ∃ n : ℕ, 5 < n ∧ n < 8 := by
  refine ⟨6, ?_, ?_⟩ <;> norm_num
```

A complete ε–δ proof

Every ε–δ argument has the same skeleton, and it uses only the tactics above.

```
example (f : ℝ → ℝ) (h : ∀ ε > 0, ∃ δ > 0, ∀ x, |x| < δ → |f x| < ε) :
    ∀ ε > 0, ∃ δ > 0, ∀ x, |x| < δ → |2 * f x| < ε := by
  intro ε hε
  obtain ⟨δ, hδ, hf⟩ := h (ε / 2) (by linarith)
  use δ, hδ
  intro x hx
  have h1 : |f x| < ε / 2 := hf x hx
  rw [abs_mul, abs_two]
  linarith
```

Line 1, `intro ε hε`. `∀ ε > 0, P ε` abbreviates `∀ ε, ε > 0 → P ε`. So there are two things to introduce: the number and the proof that it is positive.

```
ε : ℝ
hε : ε > 0
⊢ ∃ δ > 0, ∀ (x : ℝ), |x| < δ → |2 * f x| < ε
```

Line 2, `obtain … := h (ε / 2) (by linarith)`. The hypothesis is used with `ε / 2`. It needs two arguments for the same reason: the number, and a proof of `ε / 2 > 0`. That proof is produced on the spot by `(by linarith)`.

```
δ : ℝ
hδ : δ > 0
hf : ∀ (x : ℝ), |x| < δ → |f x| < ε / 2
⊢ ∃ δ > 0, ∀ (x : ℝ), |x| < δ → |2 * f x| < ε
```

Line 3, `use δ, hδ`. The same `δ` works. `∃ δ > 0, …` needs the witness and the proof that it is positive, so `use` receives both.

Lines 4 to 7. After `intro x hx` and the `have`, the rewrite with `abs_mul` and `abs_two` turns `|2 * f x|` into `2 * |f x|`:

```
hx : |x| < δ
h1 : |f x| < ε / 2
⊢ 2 * |f x| < ε
```

`linarith` finishes, since the goal is twice `h1`.

Negation

A goal `¬p` is an implication to `False`, so `intro` works:

```
example (a b : ℝ) (h : a ≤ b) : ¬ b < a := by
  intro hba         -- hba : b < a    ⊢ False
  linarith
```

`push Not` rewrites a negated statement by moving the negation inwards, the way you would on paper.

```
example (f : ℕ → ℕ) (h : ¬ ∀ n, f n ≤ 10) : ∃ n, 10 < f n := by
  push Not at h     -- h : ∃ n, 10 < f n
  exact h
```

Before	After `push Not`
`¬ ∀ x, P x`	`∃ x, ¬ P x`
`¬ ∃ x, P x`	`∀ x, ¬ P x`
`¬ a ≤ b`	`b < a`
`¬ a < b`	`b ≤ a`
`¬ (p ∧ q)`	`p → ¬ q`
`¬ (p → q)`	`p ∧ ¬ q`

Older texts call this tactic `push_neg`. That name still works in the Mathlib version used here but prints a deprecation warning.

Proof by contradiction is `by_contra`. It adds the negation of the goal as a hypothesis and changes the goal to `False`:

```
example (a b : ℝ) (h : ∀ ε > 0, a ≤ b + ε) : a ≤ b := by
  by_contra hab                              -- hab : ¬a ≤ b    ⊢ False
  push Not at hab                            -- hab : b < a
  have h' := h ((a - b) / 2) (by linarith)   -- h' : a ≤ b + (a - b) / 2
  linarith
```

`by_contra! hab` does the first two lines at once. `contrapose!` replaces "hypothesis implies goal" by its contrapositive:

```
example (x : ℝ) (h : x ^ 2 ≠ 0) : x ≠ 0 := by
  contrapose! h     -- h : x = 0    ⊢ x ^ 2 = 0
  rw [h]            --              ⊢ 0 ^ 2 = 0
  norm_num
```

When the hypotheses contradict each other, `contradiction` closes any goal, and `exfalso` changes any goal to `False`.

Iff

```
example (a b : ℝ) : a - b = 0 ↔ a = b := by
  constructor
  · intro h         -- h : a - b = 0    ⊢ a = b
    linarith
  · intro h         -- h : a = b        ⊢ a - b = 0
    linarith
```

A hypothesis `h : p ↔ q` is used as `h.mp : p → q` or `h.mpr : q → p`. It can also rewrite `p` into `q` anywhere, which is the subject of Chapter 8.

Tactics that finish a goal

Tactic	Closes goals that
`exact e`	Are proved by the term `e`

Tactic	Closes goals that
`assumption`	Are identical to a hypothesis
`rfl`	State that two things are equal by computation
`contradiction`	Have contradictory hypotheses
`omega`	Are linear arithmetic over `ℕ` or `ℤ`
`linarith`	Follow linearly from the hypotheses in an ordered field such as `ℝ`
`norm_num`	Are numerical computations
`tauto`	Are propositional tautologies
`simp`	Simplify to `True` with the library's simplification lemmas
`exact?`	Are proved by one library lemma, which it finds for you

Chapter 13 treats the automation in detail.

Combining tactics

```
example (p q : Prop) (hp : p) (hq : q) : p ∧ q := by
  constructor <;> assumption
```

Form	Meaning
`t <;> s`	Run `t`, then run `s` on every goal it produced
`all_goals t`	Run `t` on every open goal
`repeat t`	Run `t` until it fails
`first | t | s`	Try `t`, and if it fails try `s`
`try t`	Run `t`, and do nothing if it fails
`case left => t`	Work on the goal named `left`

In a project created with the `math` template, Lean warns when a tactic runs while several goals are open and leaves some untouched. The fix is always to focus with a bullet.

Terms inside tactics

Anything from Chapter 6 can be given to `exact`, `have`, `apply` or `use`. Lemmas are functions, so a short proof is often one application:

```
example (a b c : ℝ) (hab : a ≤ b) (hbc : b ≤ c) : a ≤ c := by
  exact le_trans hab hbc

-- The same, using dot notation on the proof hab : a ≤ b.
example (a b c : ℝ) (hab : a ≤ b) (hbc : b ≤ c) : a ≤ c := hab.trans hbc
```

`le_trans` has type `a ≤ b → b ≤ c → a ≤ c`. Giving it only its first argument is common: `apply le_trans hab` changes the goal `a ≤ c` into `b ≤ c`.

Choosing the next tactic

1. Read the goal. If its outermost symbol is in the table, use the tactic from the middle column.
2. If a hypothesis is an `∧` or an `∃`, unpack it at once with `obtain`. It costs nothing and shows you what you have.
3. If the goal is an atomic statement, look for a hypothesis or a lemma whose conclusion matches it, and `apply` it.
4. If you know an intermediate fact that would help, state it with `have` and prove it separately.
5. If the goal looks routine, try the finishing tactics, then `exact?`.

Exercises

```
import Mathlib

-- Exercise 1. intro, constructor, exact.
example (p q r : Prop) (hpq : p → q) (hpr : p → r) : p → q ∧ r := by
  sorry

-- Exercise 2. Unpack three parts with one obtain.
example (p q r : Prop) (h : p ∧ q ∧ r) : r ∧ p := by
  sorry

-- Exercise 3. Two cases. One ends with `left`, the other with `right`.
example (p q r : Prop) (h : p ∨ q) (hqr : q → r) : p ∨ r := by
  sorry

-- Exercise 4. Give the witness, then let norm_num check it.
example : ∃ x : ℝ, 2 * x + 1 = 7 := by
  sorry

-- Exercise 5. Unpack the hypothesis, reuse its witness.
example (a : ℝ) (h : ∃ x : ℝ, x > a) : ∃ y : ℝ, y > a - 1 := by
  sorry

-- Exercise 6. Two `have` lines with explicit types, then linarith.
example (f : ℕ → ℝ) (h : ∀ n, f n ≤ f (n + 1)) : f 0 ≤ f 2 := by
  sorry

-- Exercise 7. The goal is a negation.
example (a b : ℝ) (h : a < b) : ¬ b ≤ a := by
  sorry

-- Exercise 8. By contradiction. Use h with a / 2.
example (a : ℝ) (h : ∀ ε > 0, a ≤ ε) : a ≤ 0 := by
  sorry

-- Exercise 9. "f is not bounded above", unfolded.
example (f : ℝ → ℝ) (h : ¬ ∃ M, ∀ x, f x ≤ M) : ∀ M, ∃ x, M < f x := by
  sorry
```

```
-- Exercise 10.
example (a b : ℝ) : a ≤ b ↔ 0 ≤ b - a := by
  sorry

-- Exercise 11. by_cases on `0 ≤ x`.
example (x : ℝ) : 0 ≤ x ∨ x < 0 := by
  sorry

-- Exercise 12. The sum of two functions that tend to 0 at 0 tends to 0.
-- Useful lemmas: lt_min, min_le_left, min_le_right, lt_of_lt_of_le,
abs_add_le.
example (f g : ℝ → ℝ)
    (hf : ∀ ε > 0, ∃ δ > 0, ∀ x, |x| < δ → |f x| < ε)
    (hg : ∀ ε > 0, ∃ δ > 0, ∀ x, |x| < δ → |g x| < ε) :
    ∀ ε > 0, ∃ δ > 0, ∀ x, |x| < δ → |f x + g x| < ε := by
  sorry
```

Solutions

```
import Mathlib

-- Exercise 1.
example (p q r : Prop) (hpq : p → q) (hpr : p → r) : p → q ∧ r := by
  intro hp
  constructor
  · exact hpq hp
  · exact hpr hp

-- Exercise 2. `p ∧ q ∧ r` means `p ∧ (q ∧ r)`, and the pattern flattens it.
example (p q r : Prop) (h : p ∧ q ∧ r) : r ∧ p := by
  obtain ⟨hp, -, hr⟩ := h
  exact ⟨hr, hp⟩

-- Exercise 3.
example (p q r : Prop) (h : p ∨ q) (hqr : q → r) : p ∨ r := by
  rcases h with hp | hq
  · left
    exact hp
  · right
    exact hqr hq

-- Exercise 4.
example : ∃ x : ℝ, 2 * x + 1 = 7 := by
  use 3
  norm_num

-- Exercise 5.
example (a : ℝ) (h : ∃ x : ℝ, x > a) : ∃ y : ℝ, y > a - 1 := by
  obtain ⟨x, hx⟩ := h
  use x
  linarith

-- Exercise 6.
example (f : ℕ → ℝ) (h : ∀ n, f n ≤ f (n + 1)) : f 0 ≤ f 2 := by
  have h0 : f 0 ≤ f 1 := h 0
  have h1 : f 1 ≤ f 2 := h 1
  linarith
```

```
-- Exercise 7.
example (a b : ℝ) (h : a < b) : ¬ b ≤ a := by
  intro hba
  linarith

-- Exercise 8. If a > 0, then a ≤ a / 2, which is absurd.
example (a : ℝ) (h : ∀ ε > 0, a ≤ ε) : a ≤ 0 := by
  by_contra! ha                              -- ha : 0 < a
  have h' := h (a / 2) (by linarith)         -- h' : a ≤ a / 2
  linarith

-- Exercise 9.
example (f : ℝ → ℝ) (h : ¬ ∃ M, ∀ x, f x ≤ M) : ∀ M, ∃ x, M < f x := by
  push Not at h
  exact h

-- Exercise 10.
example (a b : ℝ) : a ≤ b ↔ 0 ≤ b - a := by
  constructor
  · intro h
    linarith
  · intro h
    linarith

-- Exercise 11.
example (x : ℝ) : 0 ≤ x ∨ x < 0 := by
  by_cases h : 0 ≤ x
  · left
    exact h
  · right
    push Not at h
    exact h

-- Exercise 12. Take the smaller of the two δ, then use the triangle
inequality.
example (f g : ℝ → ℝ)
    (hf : ∀ ε > 0, ∃ δ > 0, ∀ x, |x| < δ → |f x| < ε)
    (hg : ∀ ε > 0, ∃ δ > 0, ∀ x, |x| < δ → |g x| < ε) :
    ∀ ε > 0, ∃ δ > 0, ∀ x, |x| < δ → |f x + g x| < ε := by
  intro ε hε
  obtain ⟨δ₁, hδ₁, h₁⟩ := hf (ε / 2) (by linarith)
  obtain ⟨δ₂, hδ₂, h₂⟩ := hg (ε / 2) (by linarith)
  use min δ₁ δ₂, lt_min hδ₁ hδ₂
  intro x hx
  have hx₁ : |x| < δ₁ := lt_of_lt_of_le hx (min_le_left δ₁ δ₂)
  have hx₂ : |x| < δ₂ := lt_of_lt_of_le hx (min_le_right δ₁ δ₂)
  have hf' : |f x| < ε / 2 := h₁ x hx₁
  have hg' : |g x| < ε / 2 := h₂ x hx₂
  have htri : |f x + g x| ≤ |f x| + |g x| := abs_add_le (f x) (g x)
  linarith
```

In Exercise 12, `lt_min` proves `0 < min δ₁ δ₂`, `min_le_left` and `min_le_right` compare the minimum with each `δ`, `lt_of_lt_of_le` chains a strict and a weak inequality, and `abs_add_le` is the triangle inequality. Subscripts are typed `\1` and `\2`.

Chapter 8: Equality, Rewriting and `calc`

An equation is used by rewriting with it, chains of equalities and inequalities are written with `calc`, and `simp` rewrites with the library's whole collection of simplification lemmas at once.

Two kinds of equal

Some equations hold by computation. Lean checks them by evaluating both sides, and `rfl` proves them:

```
import Mathlib

example : (2 : ℕ) + 2 = 4 := rfl
example (n : ℕ) : n + 0 = n := rfl
```

Others are true but need a proof:

```
example (n : ℕ) : 0 + n = n := rfl
-- error: Type mismatch … rfl … is expected to have type 0 + n = n

example (n : ℕ) : 0 + n = n := zero_add n     -- a lemma, proved by
induction
example (x : ℝ) : x + 0 = x := add_zero x     -- on ℝ, even this needs a
lemma
```

Addition on `ℕ` is defined by recursion on its second argument. So `n + 0` computes to `n`, but `0 + n` is stuck until `n` is known. The first kind of equality is called *definitional*, the second *propositional*. Chapter 11 returns to this. In practice you rarely think about it: you rewrite.

`rw`

`rw [h]`, where `h : a = b`, replaces every `a` in the goal by `b`.

```
example (a b c : ℝ) (h₁ : a = b) (h₂ : b = c) : a = c := by
  rw [h₁]          -- ⊢ b = c
  exact h₂
```

Form	Effect
`rw [h]`	Replace the left side of `h` by its right side, in the goal
`rw [← h]`	Replace the right side by the left side. Type `←` as `\l`
`rw [h] at h'`	Rewrite inside the hypothesis `h'`
`rw [h] at h' ⊢`	Rewrite in `h'` and in the goal
`rw [h₁, h₂]`	Rewrite with `h₁`, then with `h₂`
`nth_rw 2 [h]`	Rewrite only the second occurrence

After rewriting, `rw` tries `rfl`. That is why many proofs end with a `rw` and nothing else.

```
example (a b : ℝ) (h : a = b + 1) : b + 1 = a := by
  rw [← h]       -- ⊢ a = a, closed by rfl

example (a b : ℝ) (h : a + 0 = b) : a = b := by
  rw [add_zero] at h      -- h : a = b
  exact h
```

Rewriting with lemmas

A lemma such as `mul_comm : a * b = b * a` is an equation with variables. `rw` uses the first place where the pattern fits, which may not be the one you want:

```
example (a b c : ℝ) : a * b * c = b * (a * c) := by
  rw [mul_comm]           -- ⊢ c * (a * b) = b * (a * c)
  sorry
```

`a * b * c` means `(a * b) * c`, and the first product Lean finds is the outer one. Give the lemma its arguments to say which product you mean:

```
example (a b c : ℝ) : a * b * c = b * (a * c) := by
  rw [mul_comm a b]       -- ⊢ b * a * c = b * (a * c)
  rw [mul_assoc]          -- ⊢ b * (a * c) = b * (a * c), closed by rfl
```

An equivalence can be used to rewrite as well. `sub_eq_zero` states `a - b = 0 ↔ a = b`:

```
example (a b : ℝ) (h : a - b = 0) : a = b := by
  rw [sub_eq_zero] at h   -- h : a = b
  exact h
```

When `rw` fails

```
example (a b : ℝ) : a + b = b + a := by
  rw [mul_comm]
-- error: Tactic `rewrite` failed: Did not find an occurrence of the pattern
--   ?a * ?b
-- in the target expression
--   a + b = b + a
```

The message shows the pattern `rw` searched for. The `?a` and `?b` are the lemma's variables. Here there is simply no product in the goal.

`rw` also cannot rewrite under a `fun`, a `∀` or a `∃` when the pattern involves the bound variable:

```
example (f : ℕ → ℕ) (h : ∀ x, f x = x) : (fun x => f x + 0) = fun x => x :=
by
  rw [h]
-- error: Did not find an occurrence of the pattern  f ?x

example (f : ℕ → ℕ) (h : ∀ x, f x = x) : (fun x => f x + 0) = fun x => x :=
by
  simp only [h, add_zero]       -- works: simp rewrites under binders
```

When `rw` fails under a binder, switch to `simp only […]` or `simp_rw […]` with the same lemmas.

`calc`

A `calc` block is the Lean form of a displayed chain. Each line states the next expression and justifies the step.

```
example (a b c d : ℝ) (h₁ : a = b + 1) (h₂ : b = c * 2) (h₃ : c = d) :
    a = d * 2 + 1 := by
  calc a = b + 1     := h₁
    _ = c * 2 + 1    := by rw [h₂]
    _ = d * 2 + 1    := by rw [h₃]
```

The underscore stands for the right side of the previous line. A justification is either a term, like `h₁`, or `by` followed by tactics.

Relations can be mixed. Lean combines `≤`, `<` and `=` correctly:

```
example (x y : ℝ) (h₁ : x ≤ y) (h₂ : y < 3) : 2 * x + 1 < 7 := by
  calc 2 * x + 1 ≤ 2 * y + 1 := by linarith
    _ < 2 * 3 + 1            := by linarith
    _ = 7                    := by norm_num
```

This is the form analysts use most. The triangle inequality between three points is a two-line `calc`:

```
example (a b c : ℝ) : |a - c| ≤ |a - b| + |b - c| := by
  calc |a - c| = |(a - b) + (b - c)| := by rw [sub_add_sub_cancel]
    _ ≤ |a - b| + |b - c|           := abs_add_le (a - b) (b - c)
```

The first step uses `sub_add_sub_cancel : a - b + (b - c) = a - c`, and `rw` closes the resulting `|a - c| = |a - c|`. The second is the triangle inequality `abs_add_le`. The library already has the result as `abs_sub_le`.

Write the `calc` skeleton first, with `sorry` as every justification. When Lean accepts the skeleton, the plan is sound, and each `sorry` is a small independent problem.

`simp`

Thousands of library lemmas are tagged `@[simp]`: equations whose right side is simpler than the left, such as `x + 0 = x` or `(xs ++ []) = xs`. The tactic `simp` rewrites with all of them, repeatedly, until nothing changes.

```
example (xs : List ℕ) : (xs ++ []).length = xs.length := by simp
```

`simp` does not use your hypotheses unless you list them:

```
example (a b : ℝ) (h : a = b) : a + 0 = b * 1 := by simp
-- error: unsolved goals  … ⊢ a = b

example (a b : ℝ) (h : a = b) : a + 0 = b * 1 := by simp [h]
```

Form	Effect
`simp`	Simplify the goal with the simp lemmas
`simp [h, foo]`	The same, also using `h` and the lemma `foo`

Form	Effect
`simp at h`	Simplify the hypothesis `h`
`simp only [foo, bar]`	Rewrite with exactly these lemmas, nothing else
`simp?`	Simplify, and report which lemmas were used
`simpa using h`	Simplify the goal and `h`, then close the goal with `h`
`simp_all`	Simplify the goal and all hypotheses, using each on the others

`simp? [h]`, run on the second example above, answers with a suggestion to click:

```
Try this:
  [apply] simp only [h, add_zero, mul_one]
```

The `simp only` form is stable: it does the same thing when Mathlib adds new simp lemmas next month. The convention in Mathlib is that a bare `simp` may end a proof, and a `simp` in the middle of a proof should be replaced by its `simp only` form.

The pattern "simplify a hypothesis, then use it" is written `simpa`:

```
example (x : ℝ) (h : x + 0 = 2 * 1) : x = 2 := by
  simpa using h
```

Writing `simp at h` followed by `exact h` also works, but the `math` template's linter flags it and suggests `simp only [...]` from `simp?`, or `suffices`; `simpa` avoids the warning too.

Definitions: `unfold`

Lean does not look inside your definitions unless asked.

```
def double (x : ℝ) : ℝ := 2 * x

example (x : ℝ) : double x = x + x := by
  unfold double       -- ⊢ 2 * x = x + x
  ring
```

`rw [double]` and `simp [double]` unfold a definition too. `ring` proves identities in commutative rings and is covered in Chapter 13.

Equality of functions: `ext`

Two functions are equal when they agree at every point. `ext x` reduces the goal to that:

```
example : (fun x : ℝ ↦ x + 0) = (fun x ↦ x) := by
  ext x               -- x : ℝ  ⊢ x + 0 = x
  simp
```

`ext` applies the same idea to sets (equal when they have the same elements) and to many structures. `funext x` is the version for functions only.

Other tools for equalities

Tool	Use
`h.symm`	From `h : a = b`, a proof of `b = a`
`h₁.trans h₂`	From `a = b` and `b = c`, a proof of `a = c`
`congrArg f h`	From `h : a = b`, a proof of `f a = f b`
`subst h`	With `h : a = b` and `b` a variable, replace `b` by `a` everywhere and remove `h`
`congr 1`	Reduce `f a = f b` to `a = b`
`ac_rfl`	Prove equalities that hold by associativity and commutativity
`conv`	Rewrite at one precise position. See the conversion mode guide

```
example (f : ℝ → ℝ) (a b : ℝ) (h : a = b) : f (a + 1) = f (b + 1) := by
  congr 1             -- ⊢ a + 1 = b + 1
  rw [h]

example (a b : ℝ) (h : a = b) (h' : a ^ 2 = 4) : b ^ 2 = 4 := by
  subst h             -- b is gone:  h' : a ^ 2 = 4  ⊢ a ^ 2 = 4
  exact h'
```

The community guides on `calc` and on the simplifier go further.

Exercises

```
import Mathlib

-- Exercise 1. Two rewrites in one rw.
example (a b c : ℝ) (h₁ : a = 2 * b) (h₂ : b = c + 1) : a = 2 * (c + 1) :=
by
  sorry

-- Exercise 2. Rewrite from right to left, then finish with ring.
example (x y : ℝ) (h : x + 1 = y) : y - 1 = x := by
  sorry

-- Exercise 3. Rewrite in the hypothesis with mul_one and add_zero.
example (a b : ℝ) (h : a * 1 = b + 0) : a = b := by
  sorry

-- Exercise 4. Use mul_assoc backwards, then mul_comm a b, then mul_assoc.
example (a b c : ℝ) : a * (b * c) = b * (a * c) := by
  sorry

-- Exercise 5. A calc with three lines.
example (a b : ℝ) (h₁ : a = 3 * b) (h₂ : b = 2) : a ^ 2 = 36 := by
  sorry

-- Exercise 6. A calc mixing ≤ and =.
example (x y : ℝ) (hx : x ≤ 2) (hy : y ≤ 5) : 3 * x + y ≤ 11 := by
  sorry
```

```
-- Exercise 7. Two uses of abs_sub_le.
example (a b c d : ℝ) : |a - d| ≤ |a - b| + |b - c| + |c - d| := by
  sorry

-- Exercise 8.
example : (fun x : ℝ ↦ 2 * x) = (fun x ↦ x + x) := by
  sorry

-- Exercise 9. simp needs to be told about h.
example (xs ys : List ℕ) (h : ys = []) : (xs ++ ys).length = xs.length := by
  sorry

-- Exercise 10. Unfold, then conclude.
noncomputable def avg (a b : ℝ) : ℝ := (a + b) / 2

example (a : ℝ) : avg a a = a := by
  sorry

example (a b : ℝ) (h : a ≤ b) : avg a b ≤ b := by
  sorry
```

Solutions

```
import Mathlib

-- Exercise 1.
example (a b c : ℝ) (h₁ : a = 2 * b) (h₂ : b = c + 1) : a = 2 * (c + 1) :=
by
  rw [h₁, h₂]

-- Exercise 2.
example (x y : ℝ) (h : x + 1 = y) : y - 1 = x := by
  rw [← h]            -- ⊢ x + 1 - 1 = x
  ring

-- Exercise 3.
example (a b : ℝ) (h : a * 1 = b + 0) : a = b := by
  rw [mul_one, add_zero] at h
  exact h

-- Exercise 4.
example (a b c : ℝ) : a * (b * c) = b * (a * c) := by
  rw [← mul_assoc]        -- ⊢ a * b * c = b * (a * c)
  rw [mul_comm a b]       -- ⊢ b * a * c = b * (a * c)
  rw [mul_assoc]

-- Exercise 5.
example (a b : ℝ) (h₁ : a = 3 * b) (h₂ : b = 2) : a ^ 2 = 36 := by
  calc a ^ 2 = (3 * b) ^ 2 := by rw [h₁]
    _ = (3 * 2) ^ 2        := by rw [h₂]
    _ = 36                 := by norm_num

-- Exercise 6.
example (x y : ℝ) (hx : x ≤ 2) (hy : y ≤ 5) : 3 * x + y ≤ 11 := by
  calc 3 * x + y ≤ 3 * 2 + 5 := by linarith
    _ = 11                   := by norm_num

-- Exercise 7. First go from a to d through c, then from a to c through b.
example (a b c d : ℝ) : |a - d| ≤ |a - b| + |b - c| + |c - d| := by
```

```
  calc |a - d| ≤ |a - c| + |c - d|      := abs_sub_le a c d
    _ ≤ |a - b| + |b - c| + |c - d|     := by linarith [abs_sub_le a b c]

-- Exercise 8.
example : (fun x : ℝ ↦ 2 * x) = (fun x ↦ x + x) := by
  ext x
  ring

-- Exercise 9.
example (xs ys : List ℕ) (h : ys = []) : (xs ++ ys).length = xs.length := by
  simp [h]

-- Exercise 10.
noncomputable def avg (a b : ℝ) : ℝ := (a + b) / 2

example (a : ℝ) : avg a a = a := by
  unfold avg          -- ⊢ (a + a) / 2 = a
  ring

example (a b : ℝ) (h : a ≤ b) : avg a b ≤ b := by
  unfold avg          -- ⊢ (a + b) / 2 ≤ b
  linarith
```

Two remarks on these solutions. In Exercise 7, `linarith [abs_sub_le a b c]` means "use linear arithmetic, with this extra fact added to the hypotheses". In Exercise 10, the word `noncomputable` is required: division of real numbers cannot be executed by a program, and Lean refuses the definition without it. The error it gives otherwise is `failed to compile definition, consider marking it as 'noncomputable'`.

Chapter 9: Inductive Types, Recursion and Induction

Every type in Lean is built by listing its constructors, a function out of such a type is defined by one case per constructor, and a proof about it is an induction with one case per constructor.

Types defined by their constructors

The simplest inductive type is a finite list of values:

```
import Mathlib

inductive Weekday where
  | monday | tuesday | wednesday
  deriving Repr, DecidableEq
```

`deriving Repr` lets `#eval` print the values. `deriving DecidableEq` lets Lean test two of them for equality.

Constructors can take arguments, including arguments of the type being defined. That is how the natural numbers and lists are made:

```
#print Nat
-- inductive Nat : Type
-- number of parameters: 0
-- constructors:
-- Nat.zero : ℕ
-- Nat.succ : ℕ → ℕ

#print List
-- inductive List.{u} : Type u → Type u
-- number of parameters: 1
-- constructors:
-- List.nil : {α : Type u} → List α
-- List.cons : {α : Type u} → α → List α → List α
```

A natural number is `zero` or the successor of a natural number. A list is empty, written `[]`, or an element in front of a list, written `x :: xs`. Nothing else is a natural number or a list, and this is what makes induction valid.

Functions by cases

```
def Weekday.next : Weekday → Weekday
  | .monday => .tuesday
  | .tuesday => .wednesday
  | .wednesday => .monday

#eval Weekday.monday.next       -- Weekday.tuesday
```

The leading dot in `.monday` is short for `Weekday.monday`. Lean knows which type is expected. Naming the function `Weekday.next` is what allows the notation `d.next`.

Proofs by cases

```
theorem Weekday.next3 (d : Weekday) : d.next.next.next = d := by
  cases d <;> rfl
```

`cases d` creates one goal per constructor:

```
case monday
⊢ monday.next.next.next = monday

case tuesday
⊢ tuesday.next.next.next = tuesday

case wednesday
⊢ wednesday.next.next.next = wednesday
```

Each is true by computation, so `<;> rfl` closes all three.

Recursive functions

```
def sumList : List ℕ → ℕ
  | [] => 0
  | x :: xs => x + sumList xs

#eval sumList [1, 2, 3]         -- 6

def sumTo : ℕ → ℕ
  | 0 => 0
  | n + 1 => (n + 1) + sumTo n

def fib : ℕ → ℕ
  | 0 => 0
  | 1 => 1
  | n + 2 => fib n + fib (n + 1)

#eval (List.range 10).map fib   -- [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

In each definition the recursive call is on a part of the argument: `xs` inside `x :: xs`, `n` inside `n + 1`. This is *structural recursion*. Lean accepts it without further question because a part is always smaller than the whole.

Induction

The statement to prove is $2 \cdot (0 + 1 + \ldots + n) = n(n + 1)$.

```
theorem sumTo_eq (n : ℕ) : 2 * sumTo n = n * (n + 1) := by
  induction n with
  | zero => rfl
  | succ k ih =>
    rw [sumTo]
    rw [mul_add, ih]
    ring
```

`induction n with` opens one case per constructor of `ℕ`. In the `succ` case you name the predecessor `k` and the induction hypothesis `ih`. The state at the start of that case is:

```
case succ
k : ℕ
ih : 2 * sumTo k = k * (k + 1)
⊢ 2 * sumTo (k + 1) = (k + 1) * (k + 1 + 1)
```

The three tactics then do what you would do on paper.

Tactic	Goal afterwards
`rw [sumTo]` unfolds one step of the definition	`2 * (k + 1 + sumTo k) = (k + 1) * (k + 1 + 1)`
`rw [mul_add, ih]` distributes, then uses the hypothesis	`2 * (k + 1) + k * (k + 1) = (k + 1) * (k + 1 + 1)`
`ring`	No goals: both sides are the same polynomial

Lists work the same way, with cases `nil` and `cons`:

```
theorem sumList_append (xs ys : List ℕ) :
    sumList (xs ++ ys) = sumList xs + sumList ys := by
  induction xs with
  | nil => simp [sumList]
  | cons x xs ih => simp [sumList, ih, add_assoc]
```

In the `cons` case the hypothesis is `ih : sumList (xs ++ ys) = sumList xs + sumList ys`, and the goal is the same statement for `x :: xs`. `simp` unfolds `sumList`, rewrites with `ih`, and reassociates the sum.

Other induction principles

Starting from a number other than zero. `Nat.le_induction` starts at the bound given by a hypothesis:

```
theorem two_mul_add_one_le (n : ℕ) (hn : 3 ≤ n) : 2 * n + 1 ≤ 2 ^ n := by
  induction n, hn using Nat.le_induction with
  | base => norm_num
  | succ k hk ih =>
    -- hk : 3 ≤ k,  ih : 2 * k + 1 ≤ 2 ^ k   ⊢ 2 * (k + 1) + 1 ≤ 2 ^ (k + 1)
    rw [pow_succ]
    omega
```

After `rw [pow_succ]` the right side is `2 ^ k * 2`. `omega` treats `2 ^ k` as an unknown number and concludes from `ih` and `hk`.

Strong induction. When the step needs the statement for several smaller values, the cleanest form is a recursive proof. A theorem is a function, so it may call itself on smaller arguments, exactly like `fib`:

```
theorem fib_le : ∀ n, fib n ≤ 2 ^ n
  | 0 => by simp [fib]
  | 1 => by simp [fib]
  | n + 2 => by
    have h1 := fib_le n             -- h1 : fib n ≤ 2 ^ n
    have h2 := fib_le (n + 1)       -- h2 : fib (n + 1) ≤ 2 ^ (n + 1)
```

```
    rw [fib, pow_succ, pow_succ]    -- ⊢ fib n + fib (n + 1) ≤ 2 ^ n * 2 * 2
    rw [pow_succ] at h2             -- h2 : fib (n + 1) ≤ 2 ^ n * 2
    omega
```

Lean checks that the recursive calls are on smaller numbers, which is the justification of strong induction. The tactic form is `induction n using Nat.strong_induction_on with | _ n ih => …`, where `ih` gives the statement for every `m < n`.

Cases without induction. When no induction hypothesis is needed, `cases` or `obtain` splits a number into zero and successor:

```
example (n : ℕ) (h : n ≠ 0) : ∃ k, n = k + 1 := by
  obtain _ | k := n
  · contradiction           -- h : 0 ≠ 0
  · exact ⟨k, rfl⟩          -- ⊢ ∃ k_1, k + 1 = k_1 + 1
```

Inductive predicates

A property can be defined by its rules of proof. "0 is even, and if n is even then n + 2 is even":

```
inductive IsEven : ℕ → Prop where
  | zero : IsEven 0
  | add_two (n : ℕ) (h : IsEven n) : IsEven (n + 2)
```

The constructors are the only ways to prove `IsEven n`. To prove it, apply them:

```
example : IsEven 4 := by
  apply IsEven.add_two      -- ⊢ IsEven 2
  apply IsEven.add_two      -- ⊢ IsEven 0
  exact IsEven.zero
```

To refute it, ask which constructor could have produced the proof:

```
theorem not_isEven_one : ¬ IsEven 1 := by
  intro h
  cases h       -- no constructor can prove IsEven 1: no cases remain
```

And you can do induction on a proof of `IsEven n`, with one case per rule:

```
theorem IsEven.exists_two_mul {n : ℕ} (h : IsEven n) : ∃ k, n = 2 * k := by
  induction h with
  | zero => exact ⟨0, rfl⟩
  | add_two m hm ih =>
    -- ih : ∃ k, m = 2 * k    ⊢ ∃ k, m + 2 = 2 * k
    obtain ⟨k, hk⟩ := ih
    exact ⟨k + 1, by omega⟩
```

Mathlib defines many notions this way, among them the σ-algebra generated by a family of sets and the transitive closure of a relation.

Logic is made of inductive types

The connectives of Chapter 6 are ordinary inductive types, which is why the same tactics work on them.

```
#print Or
-- inductive Or : Prop → Prop → Prop
-- number of parameters: 2
-- constructors:
-- Or.inl : ∀ {a b : Prop}, a → a ∨ b
-- Or.inr : ∀ {a b : Prop}, b → a ∨ b

#print False
-- inductive False : Prop
-- number of parameters: 0
-- constructors:

#print Eq
-- inductive Eq.{u_1} : {α : Sort u_1} → α → α → Prop
-- number of parameters: 2
-- constructors:
-- Eq.refl : ∀ {α : Sort u_1} (a : α), a = a
```

`Or` has two constructors, so `rcases` on a disjunction gives two cases. `False` has none, so `cases` on a proof of `False` leaves nothing to prove. Equality has one constructor, `Eq.refl a : a = a`, and `rfl` is that constructor.

Recursion that is not structural

Euclid's algorithm calls itself on `a % b`, which is not a part of `b`. Lean then needs a quantity that decreases, and a proof that it does:

```
def gcdA : ℕ → ℕ → ℕ
  | a, 0 => a
  | a, b + 1 => gcdA (b + 1) (a % (b + 1))
termination_by _ b => b
decreasing_by exact Nat.mod_lt a (Nat.succ_pos b)

#eval gcdA 12 18        -- 6
```

`termination_by _ b => b` says that the second argument decreases. `decreasing_by` proves it, here with `Nat.mod_lt`, the lemma `x % y < y` for positive `y`.

Lean often finds both by itself. When it does not, the error says so:

```
def bad (n : ℕ) : ℕ := bad (n + 1)
-- error: fail to show termination for bad …
-- failed to prove termination, possible solutions:
--   - Use `have`-expressions to prove the remaining goals
--   - Use `termination_by` to specify a different well-founded relation
--   - Use `decreasing_by` to specify your own tactic …
```

This requirement protects consistency. If `bad` were accepted with body `bad n + 1`, one could prove `0 = 1`. The guide on well-founded recursion covers harder cases.

Exercises

```
import Mathlib

-- Exercise 1. The length of a list.
```

```
def len {α : Type} : List α → ℕ
  | [] => sorry
  | _ :: xs => sorry
example : len [7, 8, 9] = 3 := rfl

-- Exercise 2. A proof by cases.
inductive Color where
  | red | green | blue

def Color.swap : Color → Color
  | .red => .green
  | .green => .red
  | .blue => .blue

example (c : Color) : c.swap.swap = c := by
  sorry

-- Exercise 3. The sum of the first n odd numbers is n².
def sumOdd : ℕ → ℕ
  | 0 => 0
  | n + 1 => sumOdd n + (2 * n + 1)

theorem sumOdd_eq (n : ℕ) : sumOdd n = n ^ 2 := by
  sorry

-- Exercise 4. Induction on xs. Needs your `len` from Exercise 1.
theorem len_append {α : Type} (xs ys : List α) :
    len (xs ++ ys) = len xs + len ys := by
  sorry

-- Exercise 5. Use pow_succ in the inductive step, then omega.
theorem succ_le_two_pow (n : ℕ) : n + 1 ≤ 2 ^ n := by
  sorry

-- Exercise 6. Apply a constructor twice.
inductive IsEven : ℕ → Prop where
  | zero : IsEven 0
  | add_two (n : ℕ) (h : IsEven n) : IsEven (n + 2)

example (n : ℕ) (h : IsEven n) : IsEven (n + 4) := by
  sorry

-- Exercise 7. Two rounds of `cases`.
example : ¬ IsEven 3 := by
  sorry

-- Exercise 8. Induction on the proof hn.
theorem IsEven.add {m n : ℕ} (hm : IsEven m) (hn : IsEven n) : IsEven (m +
n) := by
  sorry
```

Solutions

```
import Mathlib

-- Exercise 1.
def len {α : Type} : List α → ℕ
  | [] => 0
  | _ :: xs => 1 + len xs
```

```
example : len [7, 8, 9] = 3 := rfl

-- Exercise 2.
inductive Color where
  | red | green | blue

def Color.swap : Color → Color
  | .red => .green
  | .green => .red
  | .blue => .blue

example (c : Color) : c.swap.swap = c := by
  cases c <;> rfl

-- Exercise 3.
def sumOdd : ℕ → ℕ
  | 0 => 0
  | n + 1 => sumOdd n + (2 * n + 1)

theorem sumOdd_eq (n : ℕ) : sumOdd n = n ^ 2 := by
  induction n with
  | zero => rfl
  | succ k ih =>
    rw [sumOdd, ih]     -- ⊢ k ^ 2 + (2 * k + 1) = (k + 1) ^ 2
    ring

-- Exercise 4.
theorem len_append {α : Type} (xs ys : List α) :
    len (xs ++ ys) = len xs + len ys := by
  induction xs with
  | nil => simp [len]
  | cons x xs ih => simp [len, ih, add_assoc]

-- Exercise 5.
theorem succ_le_two_pow (n : ℕ) : n + 1 ≤ 2 ^ n := by
  induction n with
  | zero => norm_num
  | succ k ih =>
    rw [pow_succ]       -- ih : k + 1 ≤ 2 ^ k   ⊢ k + 1 + 1 ≤ 2 ^ k * 2
    omega

-- Exercise 6. `n + 4` computes to `n + 2 + 2`, so the constructor applies.
inductive IsEven : ℕ → Prop where
  | zero : IsEven 0
  | add_two (n : ℕ) (h : IsEven n) : IsEven (n + 2)

example (n : ℕ) (h : IsEven n) : IsEven (n + 4) := by
  apply IsEven.add_two
  apply IsEven.add_two
  exact h

-- Exercise 7. A proof of IsEven 3 must come from a proof of IsEven 1,
-- and nothing proves IsEven 1.
example : ¬ IsEven 3 := by
  intro h
  cases h with
  | add_two n h' => cases h'

-- Exercise 8. m + (k + 2) computes to (m + k) + 2.
```

```
theorem IsEven.add {m n : ℕ} (hm : IsEven m) (hn : IsEven n) : IsEven (m +
n) := by
  induction hn with
  | zero => exact hm
  | add_two k hk ih => exact IsEven.add_two (m + k) ih
```

Chapter 10: Structures and Type Classes

A structure bundles several pieces of data and facts into one object, and a type class is a structure that Lean finds for you, which is how `+`, `≤` and every algebraic law in Mathlib work on all types at once.

Structures

```
import Mathlib

structure Point where
  x : ℝ
  y : ℝ
```

This one command creates a type `Point`, a constructor `Point.mk`, and a projection for each field.

```
#check Point.mk     -- Point.mk (x y : ℝ) : Point
#check Point.x      -- Point.x (self : Point) : ℝ
```

There are three ways to build a value, and they mean the same thing:

```
def origin : Point := { x := 0, y := 0 }    -- named fields
def p1 : Point := ⟨1, 2⟩                    -- anonymous constructor
def p2 : Point := Point.mk 3 4              -- explicit constructor

#check p1.x         -- p1.x : ℝ
```

`p1.x` is dot notation for `Point.x p1`, as in Chapter 5. Functions defined in the `Point` namespace get the same notation:

```
def Point.add (p q : Point) : Point :=
  { x := p.x + q.x, y := p.y + q.y }

example : (p1.add p2).x = 1 + 3 := rfl
```

To copy a value while changing some fields, use `with`:

```
def p3 : Point := { p1 with y := 10 }

example : p3.x = 1 := rfl
```

When are two structures equal?

Two values of a structure are equal when all their fields are equal. The attribute `@[ext]` generates that lemma and registers it for the `ext` tactic of Chapter 8.

```
@[ext] structure Vec2 where
  x : ℝ
  y : ℝ

#check @Vec2.ext    -- ∀ {x y : Vec2}, x.x = y.x → x.y = y.y → x = y

example (v w : Vec2) (hx : v.x = w.x) (hy : v.y = w.y) : v = w := by
  ext
  -- two goals:  ⊢ v.x = w.x   and   ⊢ v.y = w.y
```

```
  · exact hx
  · exact hy
```

Fields can be proofs

A structure can carry a fact about its data. Building a value then requires proving the fact.

```
structure PosReal where
  val : ℝ
  pos : 0 < val

def PosReal.one : PosReal := { val := 1, pos := by norm_num }

def PosReal.double (a : PosReal) : PosReal where
  val := 2 * a.val
  pos := by
    have h := a.pos
    -- h : 0 < a.val   ⊢ 0 < 2 * a.val
    linarith
```

The field `pos` has type `0 < val`: its type mentions the earlier field. This is a dependent type at work, and it is how Mathlib bundles an object with its properties.

The form `def … : PosReal where` followed by one line per field is the usual layout for anything longer than one line.

Structures of hypotheses

A structure whose fields are all proofs is a proposition. Declare it with `: Prop`. It is a named conjunction with named parts.

```
structure IsLinear (f : ℝ → ℝ) : Prop where
  map_add : ∀ x y, f (x + y) = f x + f y
  map_smul : ∀ c x, f (c * x) = c * f x
```

To prove it, give each field:

```
theorem isLinear_three : IsLinear (fun x ↦ 3 * x) where
  map_add := by
    intro x y
    -- ⊢ 3 * (x + y) = 3 * x + 3 * y
    ring
  map_smul := by
    intro c x
    ring
```

To use it, project a field out of the hypothesis:

```
theorem IsLinear.map_zero {f : ℝ → ℝ} (hf : IsLinear f) : f 0 = 0 := by
  have h := hf.map_smul 0 0
  -- h : f (0 * 0) = 0 * f 0
  rw [zero_mul, zero_mul] at h
  -- h : f 0 = 0
  exact h
```

Mathlib states most compound properties this way. `Continuous`, `IsCompact` and `MeasureTheory.Integrable` are propositions with named consequences reached by dot notation, such as `hf.comp hg`.

Type classes: one name, many types

Suppose a function should work for every type that has an area. A **class** declares what is required, and an **instance** provides it for one type.

```
class HasArea (α : Type) where
  area : α → ℝ

structure Rect where
  w : ℝ
  h : ℝ

structure Square where
  side : ℝ

instance : HasArea Rect where
  area r := r.w * r.h

instance : HasArea Square where
  area s := s.side ^ 2
```

A function that needs an area asks for the instance in square brackets:

```
def twiceArea {α : Type} [HasArea α] (a : α) : ℝ := 2 * HasArea.area a

#check @twiceArea   -- @twiceArea : {α : Type} → [HasArea α] → α → ℝ

example : twiceArea (Rect.mk 2 3) = 12 := by
  norm_num [twiceArea, HasArea.area]

example : twiceArea (Square.mk 3) = 18 := by
  norm_num [twiceArea, HasArea.area]
```

There are now three kinds of argument brackets.

Brackets	Name	Who supplies the argument
`(a : α)`	Explicit	You
`{α : Type}`	Implicit	Lean, by unification with the other arguments
`[HasArea α]`	Instance implicit	Lean, by searching the declared instances

The search is called **instance synthesis**. When it fails, the error names the class it could not find. Calling `twiceArea` on a type with no instance gives:

```
error: failed to synthesize instance of type class
  HasArea Circle
```

Notation is type classes

`a + b` is notation for `HAdd.hAdd a b`, and for two values of the same type it goes through the class `Add`. Giving your type an `Add` instance gives it `+`.

```
instance : Add Point where
  add p q := { x := p.x + q.x, y := p.y + q.y }

#check p1 + p2      -- p1 + p2 : Point

example : (p1 + p2).x = 1 + 3 := rfl
```

The same holds for `*` (`Mul`), `≤` (`LE`), `0` (`Zero`), `1` (`One`), `⁻¹` (`Inv`), `‖x‖` (`Norm`) and `∈` (`Membership`).

The algebraic hierarchy

Mathlib defines algebraic structures as classes that bundle operations with their laws: `Group`, `CommRing`, `Field`, `LinearOrder` and several hundred more. The command `#synth` shows which instance Lean finds.

```
#synth Field ℝ            -- Real.instField
#synth LinearOrder ℝ      -- Real.linearOrder
#synth Ring ℤ             -- Int.instRing
#synth Field ℤ            -- error: failed to synthesize  Field ℤ
```

A lemma is stated once, for every type with the needed structure. Look at what `mul_comm` and `le_trans` assume:

```
#check @mul_comm
-- ∀ {G : Type u_1} [inst : CommMagma G] (a b : G), a * b = b * a

#check @mul_inv_cancel
-- ∀ {G : Type u_1} [inst : Group G] (a : G), a * a⁻¹ = 1

#check @le_trans
-- ∀ {α : Type u_1} [inst : Preorder α] {a b c : α}, a ≤ b → b ≤ c → a ≤ c
```

`mul_comm` holds in any `CommMagma`, the weakest structure with a commutative multiplication. It applies to `ℝ` because Lean can derive `CommMagma ℝ` from `Field ℝ` by walking up the hierarchy.

You state your own results the same way. Here is a computation valid in every group:

```
example {G : Type*} [Group G] (a b : G) : a * b * b⁻¹ = a := by
  -- ⊢ a * b * b⁻¹ = a
  rw [mul_assoc]
  -- ⊢ a * (b * b⁻¹) = a
  rw [mul_inv_cancel]
  -- ⊢ a * 1 = a
  rw [mul_one]
```

The automation is generic too. `ring` works in any commutative ring, not only in `ℝ`:

```
example {R : Type*} [CommRing R] (a b : R) :
    (a + b) ^ 2 = a ^ 2 + 2 * a * b + b ^ 2 := by
  ring
```

```
example {α : Type*} [LinearOrder α] (a b : α) : min a b ≤ max a b :=
  le_trans (min_le_left a b) (le_max_left a b)
```

In the Infoview, an instance argument appears as a line such as `inst✝ : Group G`. The dagger marks a name you cannot refer to. You never need to: Lean passes the instance along by itself.

The group and ring classes are defined in `Mathlib/Algebra/Group/Defs.lean` and `Mathlib/Algebra/Ring/Defs.lean`. The design is explained in the chapters *Structures* and *Hierarchies* of Mathematics in Lean.

Building a class from other classes

`extends` includes the fields of other classes. This is how a monoid is "a type with a multiplication and a one, such that…":

```
class MyMonoid (α : Type) extends Mul α, One α where
  mul_assoc : ∀ a b c : α, a * b * c = a * (b * c)
  one_mul : ∀ a : α, 1 * a = a
  mul_one : ∀ a : α, a * 1 = a

instance : MyMonoid ℕ where
  mul_assoc := Nat.mul_assoc
  one_mul := Nat.one_mul
  mul_one := Nat.mul_one
```

Because `MyMonoid` extends `Mul` and `One`, the notation `*` and `1` is available as soon as `[MyMonoid α]` is assumed. A theorem proved from the axioms then holds in every instance:

```
theorem MyMonoid.one_unique {α : Type} [MyMonoid α] (e : α)
    (h : ∀ a : α, e * a = a) : e = 1 := by
  have h1 := h 1
  -- h1 : e * 1 = 1
  rw [MyMonoid.mul_one] at h1
  -- h1 : e = 1
  exact h1
```

Coercions

A natural number is not a real number in Lean: `ℕ` and `ℝ` are different types. When a term of one type is used where another is expected, Lean inserts a **coercion**, displayed as an up arrow `↑`.

```
#check ((3 : ℕ) : ℝ)        -- ↑3 : ℝ

example (n : ℕ) (x : ℝ) : x + n = n + x := by
  -- ⊢ x + ↑n = ↑n + x
  ring
```

You typed `x + n`. Lean elaborated it to `x + ↑n`, where `↑n` is the real number corresponding to `n`. Coercions are instances of the class `Coe`, and you can declare your own:

```
structure EvenNat where
  val : ℕ
  even : Even val

instance : Coe EvenNat ℕ := ⟨EvenNat.val⟩

def four : EvenNat := ⟨4, by decide⟩

#eval (four : ℕ) + 1        -- 5
```

Facts move across a coercion with `exact_mod_cast`:

```
example (n : ℕ) (h : 2 ≤ n) : (2 : ℝ) ≤ n := by
  exact_mod_cast h
```

The goal is `2 ≤ ↑n` in `ℝ`, and `h` is `2 ≤ n` in `ℕ`. They differ only by casts, which the tactic removes. Chapter 14 is devoted to this topic.

Errors you will meet here

Message	Cause	Fix
`failed to synthesize instance of type class HasArea Circle`	No instance of that class exists for that type	Declare the instance, or add the hypothesis `[HasArea α]` to your statement
`failed to synthesize Field ℤ`	The structure does not exist: `ℤ` is not a field	Work in `ℚ` or `ℝ`, or use a lemma that needs less
`No applicable extensionality theorem found for type Point`	`ext` was used on a structure without `@[ext]`	Add `@[ext]` to the structure
`Fields missing: `pos``	A structure value was built without one of its fields	Supply the field, here a proof

Exercises

```
import Mathlib

-- Exercise 1. A closed interval, and the fact that its length is
nonnegative.
structure ClosedInterval where
  lo : ℝ
  hi : ℝ
  le : lo ≤ hi

def ClosedInterval.length (I : ClosedInterval) : ℝ := I.hi - I.lo

def ClosedInterval.unit : ClosedInterval := { lo := 0, hi := 1, le :=
sorry }

theorem ClosedInterval.length_nonneg (I : ClosedInterval) : 0 ≤ I.length :=
```

```
by
  sorry

-- Exercise 2. Give Point a negation, so that the example below holds by
rfl.
structure Point where
  x : ℝ
  y : ℝ

instance : Neg Point where
  neg p := sorry

example (p : Point) : (-p).x = -p.x := rfl

-- Exercise 3. The composition of two linear maps is linear.
structure IsLinear (f : ℝ → ℝ) : Prop where
  map_add : ∀ x y, f (x + y) = f x + f y
  map_smul : ∀ c x, f (c * x) = c * f x

theorem IsLinear.comp {f g : ℝ → ℝ} (hf : IsLinear f) (hg : IsLinear g) :
    IsLinear (fun x ↦ f (g x)) where
  map_add := by
    sorry
  map_smul := by
    sorry

-- Exercise 4. Valid in every group. Use ← mul_assoc, inv_mul_cancel,
one_mul.
example {G : Type*} [Group G] (a b : G) : a⁻¹ * (a * b) = b := by
  sorry

-- Exercise 5. Declare the instance: the area of a triangle is base * height
/ 2.
class HasArea (α : Type) where
  area : α → ℝ

structure Triangle where
  base : ℝ
  height : ℝ

-- instance : HasArea Triangle where …

-- Exercise 6. Move a fact from ℕ to ℝ.
example (n : ℕ) (h : n < 5) : (n : ℝ) < 5 := by
  sorry

-- Exercise 7. Valid in every linear order. Use min_le_right and le_trans.
example {α : Type*} [LinearOrder α] (a b c : α) (h : b ≤ c) : min a b ≤ c :=
by
  sorry
```

Solutions

```
import Mathlib

-- Exercise 1.
structure ClosedInterval where
  lo : ℝ
  hi : ℝ
```

```
  le : lo ≤ hi

def ClosedInterval.length (I : ClosedInterval) : ℝ := I.hi - I.lo

def ClosedInterval.unit : ClosedInterval := { lo := 0, hi := 1, le := by
norm_num }

theorem ClosedInterval.length_nonneg (I : ClosedInterval) : 0 ≤ I.length :=
by
  unfold ClosedInterval.length
  -- ⊢ 0 ≤ I.hi - I.lo
  have h := I.le
  -- h : I.lo ≤ I.hi
  linarith

-- Exercise 2.
structure Point where
  x : ℝ
  y : ℝ

instance : Neg Point where
  neg p := { x := -p.x, y := -p.y }

example (p : Point) : (-p).x = -p.x := rfl

-- Exercise 3.
structure IsLinear (f : ℝ → ℝ) : Prop where
  map_add : ∀ x y, f (x + y) = f x + f y
  map_smul : ∀ c x, f (c * x) = c * f x

theorem IsLinear.comp {f g : ℝ → ℝ} (hf : IsLinear f) (hg : IsLinear g) :
    IsLinear (fun x ↦ f (g x)) where
  map_add := by
    intro x y
    -- ⊢ f (g (x + y)) = f (g x) + f (g y)
    rw [hg.map_add, hf.map_add]
  map_smul := by
    intro c x
    rw [hg.map_smul, hf.map_smul]

-- Exercise 4.
example {G : Type*} [Group G] (a b : G) : a⁻¹ * (a * b) = b := by
  rw [← mul_assoc, inv_mul_cancel, one_mul]

-- Exercise 5.
class HasArea (α : Type) where
  area : α → ℝ

structure Triangle where
  base : ℝ
  height : ℝ

noncomputable instance : HasArea Triangle where
  area t := t.base * t.height / 2

-- Exercise 6.
example (n : ℕ) (h : n < 5) : (n : ℝ) < 5 := by
  exact_mod_cast h
```

```
-- Exercise 7.
example {α : Type*} [LinearOrder α] (a b c : α) (h : b ≤ c) : min a b ≤ c :=
  le_trans (min_le_right a b) h
```

In Exercise 5 the word `noncomputable` is required. Division of real numbers cannot be executed by a program, and Lean asks you to acknowledge that. Chapter 14 explains why.

Chapter 11: Dependent Type Theory, One Level Deeper

Four ideas explain most of Lean's surprising behaviour: types have types, two expressions can be equal by computation or only by proof, propositions carry no data, and some definitions cannot be executed.

You can prove a great deal without this chapter. Read it now for the vocabulary, and again when an error message uses one of its words.

Types have types

```
import Mathlib

#check ℕ               -- ℕ : Type
#check 2 + 2 = 4       -- 2 + 2 = 4 : Prop
#check Prop            -- Prop : Type
#check Type            -- Type : Type 1
#check Type 1          -- Type 1 : Type 2
#check ℕ → Type        -- ℕ → Type : Type 1
```

The levels are called **universes**. `Prop` holds statements, `Type` holds ordinary types such as `ℕ` and `ℝ`, and `Type 1` holds things that quantify over all of `Type`. The ladder stops the theory from containing a type of all types.

Written	Means	You meet it in
`Prop`	The universe of propositions	Every statement
`Type`	The universe of data types	`ℕ`, `ℝ`, `List ℕ`
`Type u`	A universe at level `u`, after `universe u`	Lean core and older Mathlib files
`Type*`	"Some universe, I do not care which"	Almost every Mathlib statement
`Sort u`	Either `Prop` or a `Type`, at level `u`	Very general definitions such as `id`

When you write a general statement, write `{α : Type*}`. That is the whole practical content of universes for an analyst.

Functions whose result type depends on the argument

In `f : ℕ → ℝ` every value has the same type. In a **dependent function** the type of the result varies with the argument.

```
#check Fin 3           -- Fin 3 : Type

def lastOf (n : ℕ) : Fin (n + 1) := Fin.last n

#check @lastOf         -- lastOf : (n : ℕ) → Fin (n + 1)
#eval lastOf 4         -- 4
```

`Fin n` is the type of natural numbers below `n`. The value `lastOf 4` lives in `Fin 5`, and `lastOf 9` lives in `Fin 10`.

You have used dependent functions since Chapter 6. A proof of `∀ n : ℕ, n = n` is a function taking `n` to a proof of `n = n`, and the type `n = n` depends on `n`.

```
#check ∀ n : ℕ, n = n              -- ∀ (n : ℕ), n = n : Prop
#check (n : ℕ) → Fin (n + 1)       -- (n : ℕ) → Fin (n + 1) : Type
```

`∀` is the notation for a dependent function type that lands in `Prop`.

Equal by computation, equal by proof

Lean has two notions of sameness.

- **Definitional equality.** Two expressions are definitionally equal when unfolding definitions and computing turns one into the other. Lean checks this by itself, and `rfl` proves it.
- **Propositional equality.** The statement `a = b`, which may need a real proof.

The difference shows on the first try:

```
example (n : ℕ) : n + 0 = n := rfl     -- works
example (n : ℕ) : 0 + n = n := rfl     -- error
```

```
error: Type mismatch
  rfl
has type
  ?m.9 = ?m.9
but is expected to have type
  0 + n = n
```

Addition on `ℕ` is defined by recursion on its second argument. So `n + 0` computes to `n`, but `0 + n` is stuck until `n` is known. The second statement is true, and its proof is an induction, available as `Nat.zero_add` or through `simp`.

Your own definitions unfold the same way:

```
def square (x : ℝ) : ℝ := x * x

example (x : ℝ) : square x = x * x := rfl

example (x : ℝ) (h : 0 ≤ x * x) : 0 ≤ square x := h
```

In the last line, `h` proves `0 ≤ x * x` and the goal is `0 ≤ square x`. Lean accepts `h` because the two statements are definitionally equal.

The tactic `change` restates the goal as something definitionally equal, which makes such a step visible:

```
example (x : ℝ) (h : 0 ≤ x * x) : 0 ≤ square x := by
  change 0 ≤ x * x
  exact h
```

Automation does not look inside your definitions

Tactics such as `ring`, `linarith` and `positivity` work on the expression as written. They do not unfold `square`.

```
example (x : ℝ) : square x = x ^ 2 := by
  ring                -- error: unsolved goals ⊢ square x = x ^ 2

example (x : ℝ) : square x = x ^ 2 := by
  unfold square
  -- ⊢ x * x = x ^ 2
  ring
```

This is the most common reason a proof that "should be obvious" fails. Unfold your definition first, with `unfold`, or with `simp only [square]`.

Better still, prove the basic facts about a new definition once, as named lemmas, and use those afterwards. That is how Mathlib is organized: a definition, followed by its API.

Propositions carry no data

Any two proofs of the same proposition are equal, by definition:

```
example (p : Prop) (h₁ h₂ : p) : h₁ = h₂ := rfl
```

This is **proof irrelevance**. It is why a structure with proof fields, such as `PosReal` in Chapter 10, is determined by its data fields alone.

The price is that you cannot extract data from a proof when building data. Lean rejects this definition:

```
example (h : ∃ n : ℕ, n > 3) : ℕ := by
  obtain ⟨n, hn⟩ := h
  exact n
```

```
error: Tactic `cases` failed with a nested error:
Tactic `induction` failed: recursor `Exists.casesOn` can only eliminate into
`Prop`
```

Inside a proof, `obtain ⟨n, hn⟩ := h` is always allowed, because the goal is then a proposition. To define a number from an existence statement, you need the axiom of choice, described below.

Subtypes

The type of elements of `α` satisfying `p` is written `{ x : α // p x }`. A term is a pair: a value and a proof.

```
def two : { x : ℝ // 0 < x } := ⟨2, by norm_num⟩

#check two.val            -- ↑two : ℝ
#check two.property       -- two.property : 0 < ↑two
```

The value is `.val` or `.1`, displayed with the coercion arrow. The proof is `.property` or `.2`.

```
example (a : { x : ℝ // 0 < x }) : 0 < (a : ℝ) := a.property

def addPos (a b : { x : ℝ // 0 < x }) : { x : ℝ // 0 < x } :=
  ⟨a.val + b.val, add_pos a.property b.property⟩

example (a b : { x : ℝ // 0 < x }) (h : (a : ℝ) = b) : a = b := by
  ext
  exact h
```

Two terms of a subtype are equal when their values are equal. Proof irrelevance takes care of the second component.

Subtypes are everywhere in Mathlib:

- `Fin n` is a structure with a value `val : ℕ` and a proof `isLt : val < n`.
- The nonnegative reals `ℝ≥0` are `{ r : ℝ // 0 ≤ r }`, defined in `Mathlib/Basic/NNReal/Defs.lean`.
- A set `s : Set ℝ` used as a type means the subtype of its elements. The Infoview writes it `↑s`.

```
example (s : Set ℝ) (a : s) : (a : ℝ) ∈ s := a.property
```

Quotients

A quotient identifies elements related by an equivalence relation. The relation is packaged as a `Setoid`. Here, two natural numbers are identified when they have the same parity.

```
def parityRel : Setoid ℕ where
  r a b := a % 2 = b % 2
  iseqv := {
    refl := fun _ ↦ rfl
    symm := fun h ↦ h.symm
    trans := fun h₁ h₂ ↦ h₁.trans h₂ }

def Parity : Type := Quotient parityRel

def Parity.mk (n : ℕ) : Parity := Quotient.mk parityRel n
```

Three operations do all the work.

Operation	Mathematical meaning
`Quotient.sound`	Related elements have equal classes
`Quotient.lift f h`	A function `f` that respects the relation, as shown by `h`, descends to the quotient
`Quotient.inductionOn`	To prove a property of every class, prove it for every representative

```
example : Parity.mk 3 = Parity.mk 7 := by
  apply Quotient.sound
```

```
  -- ⊢ 3 ≈ 7
  rfl

def Parity.toBit : Parity → ℕ :=
  Quotient.lift (fun n ↦ n % 2) (fun _ _ h ↦ h)

#eval (Parity.mk 7).toBit       -- 1

theorem Parity.toBit_lt (p : Parity) : p.toBit < 2 := by
  induction p using Quotient.inductionOn with
  | _ n =>
    -- ⊢ toBit ⟦n⟧ < 2
    exact Nat.mod_lt n (by norm_num)
```

The second argument of `Quotient.lift` is the proof that the function is well defined. Try to lift `fun n ↦ n` instead: the goal becomes `a ≈ b ⊢ a = b`, which is false, so the ill-defined function cannot be written.

The notation `⟦n⟧` is the class of `n`, typed `\[[` and `\]]`.

Mathlib's quotients include the real numbers, built from Cauchy sequences of rationals in `Mathlib/Basic/Real/Basic.lean`, quotient groups and rings, and functions modulo almost-everywhere equality in `Mathlib/MeasureTheory/Function/AEEqFun.lean`. The last one is what `Lᵖ` spaces are made of. The post What is a quotient? compares this with the set-theoretic construction.

Choice and `noncomputable`

The axiom `Classical.choice` produces an element of any nonempty type. Its most useful form turns an existence statement into a value and a fact about it:

```
#check @Classical.choose        -- {p : α → Prop} → (∃ x, p x) → α
#check @Classical.choose_spec   -- (h : ∃ x, p x), p (Classical.choose h)

noncomputable def pick (h : ∃ x : ℝ, x ^ 2 = 2) : ℝ := Classical.choose h

theorem pick_spec (h : ∃ x : ℝ, x ^ 2 = 2) : pick h ^ 2 = 2 :=
  Classical.choose_spec h
```

A definition that uses choice has no program behind it, and must be marked `noncomputable`. Without the mark, Lean says:

```
error: failed to compile definition, consider marking it as 'noncomputable'
because it depends on 'Classical.choose', which is 'noncomputable'
```

The same happens with real numbers. Division on `ℝ` is noncomputable, so `def inv (x : ℝ) : ℝ := 1 / x` needs the mark too. It costs nothing: `noncomputable` only means `#eval` will not run it. Theorems never need it.

Many Mathlib files begin with `noncomputable section`, which marks everything in the section.

Inside a proof, the tactic `choose` applies choice to a hypothesis of the form `∀ x, ∃ y, …`:

```
example (f : ℕ → ℕ) (h : ∀ n, ∃ m, f m > n) :
    ∃ g : ℕ → ℕ, ∀ n, f (g n) > n := by
  choose g hg using h
  -- g : ℕ → ℕ
  -- hg : ∀ (n : ℕ), f (g n) > n
  exact ⟨g, hg⟩
```

Analysts do this constantly: "for each `n` pick x_n such that…".

Decidability and `if`

An `if p then a else b` needs a way to decide `p`. For natural numbers, comparison is computed. For real numbers it cannot be, and Mathlib supplies a classical instance.

```
#eval if 3 < 5 then 1 else 0         -- 1

noncomputable def absR (x : ℝ) : ℝ := if 0 ≤ x then x else -x
```

To prove something about a definition by cases, `split_ifs` creates one goal per branch and adds the condition as a hypothesis:

```
example (x : ℝ) : 0 ≤ absR x := by
  unfold absR
  -- ⊢ 0 ≤ if 0 ≤ x then x else -x
  split_ifs with h
  · -- h : 0 ≤ x      ⊢ 0 ≤ x
    exact h
  · -- h : ¬0 ≤ x     ⊢ 0 ≤ -x
    linarith
```

When the condition is known, `simp` chooses the branch:

```
example (x : ℝ) (h : 0 ≤ x) : absR x = x := by
  simp [absR, h]
```

The tactic `decide` proves a statement by running its decision procedure. It works on `ℕ` and fails on `ℝ`, where there is nothing to run:

```
example : (3 : ℕ) < 5 := by decide       -- works
example : (3 : ℝ) < 5 := by decide       -- error: reduction got stuck at
Classical.choice
example : (3 : ℝ) < 5 := by norm_num     -- works: norm_num proves it, it
does not compute it
```

The axioms, again

```
#print axioms funext          -- 'funext' depends on axioms: [Quot.sound]
#print axioms Classical.em
-- 'Classical.em' depends on axioms: [propext, Classical.choice, Quot.sound]
```

Function extensionality is a theorem, derived from quotients. The excluded middle is a theorem, derived from choice. Seeing these three axioms under your own theorem is normal. Seeing `sorryAx` means a `sorry` is hiding somewhere upstream.

The chapters *Dependent Type Theory* and *Axioms and Computation* of Theorem Proving in Lean give the full account, and the Language Reference is the specification.

Errors you will meet here

Message	Cause	Fix
`Type mismatch: rfl has type ?m = ?m but is expected to have type 0 + n = n`	True, but not by computation	Use the lemma, or `simp`, `omega`, `ring`
`unsolved goals` after `ring` or `linarith failed` on a goal mentioning your definition	Automation does not unfold definitions	`unfold` it first
`failed to compile definition, consider marking it as 'noncomputable'`	The definition uses choice or real arithmetic	Add `noncomputable`
`recursor 'Exists.casesOn' can only eliminate into Prop`	Data extracted from an `∃` outside a proof	Use `Classical.choose`, or state the hypothesis as a subtype
`Tactic 'decide' failed … reduction got stuck at Classical.choice`	Nothing to compute, typically on `ℝ`	Use `norm_num`
`motive is not type correct`	`rw` would produce an ill-typed term, because another part of the goal depends on what is rewritten	Use `subst`, `simp only`, or `conv`

The last message deserves an example. The type `Fin n` of `v` depends on `n`, so rewriting `n` alone breaks the typing:

```
example (n m : ℕ) (h : n = m) (v : Fin n) (w : Fin m) (hw : w.val = v.val) :
    (h ▸ v : Fin m) = w := by
  rw [h]              -- error: motive is not type correct

example (n m : ℕ) (h : n = m) (v : Fin n) (w : Fin m) (hw : w.val = v.val) :
    (h ▸ v : Fin m) = w := by
  subst h             -- replaces m by n everywhere, in types as well
  ext
  exact hw.symm
```

Exercises

```
import Mathlib

-- Exercise 1. Predict which of these five lines are accepted. Then check.
example (n : ℕ) : n + 0 = n := rfl
example (n : ℕ) : n * 0 = 0 := rfl
```

```
example (n : ℕ) : n * 1 = n := rfl
example (n : ℕ) : n + 1 = Nat.succ n := rfl
example : 2 * 3 = 6 := rfl

-- Exercise 2. Prove the one that failed.
example (n : ℕ) : n * 1 = n := by
  sorry

-- Exercise 3. Half of a positive real is a positive real.
noncomputable def half (a : { x : ℝ // 0 < x }) : { x : ℝ // 0 < x } :=
  ⟨a.val / 2, sorry⟩

-- Exercise 4. Automation does not unfold definitions.
def cube (x : ℝ) : ℝ := x * x * x

example (x : ℝ) : cube (2 * x) = 8 * cube x := by
  sorry

-- Exercise 5. Two numbers with the same parity have the same class.
def parityRel : Setoid ℕ where
  r a b := a % 2 = b % 2
  iseqv := {
    refl := fun _ ↦ rfl
    symm := fun h ↦ h.symm
    trans := fun h₁ h₂ ↦ h₁.trans h₂ }

def Parity : Type := Quotient parityRel

def Parity.mk (n : ℕ) : Parity := Quotient.mk parityRel n

example : Parity.mk 4 = Parity.mk 10 := by
  sorry

-- Exercise 6. Turn "for every x there is a y" into a function.
example (h : ∀ x : ℝ, ∃ y : ℝ, x < y) : ∃ f : ℝ → ℝ, ∀ x, x < f x := by
  sorry

-- Exercise 7. A definition by cases. Four cases appear after split_ifs.
noncomputable def absR (x : ℝ) : ℝ := if 0 ≤ x then x else -x

example (x : ℝ) : absR x = absR (-x) := by
  sorry
```

Solutions

```
import Mathlib

-- Exercise 1. The third line fails. Multiplication recurses on its second
-- argument: n * 1 unfolds to n * 0 + n, then to 0 + n, which is stuck.

-- Exercise 2.
example (n : ℕ) : n * 1 = n := by
  simp

example (n : ℕ) : n * 1 = n := Nat.mul_one n

-- Exercise 3.
noncomputable def half (a : { x : ℝ // 0 < x }) : { x : ℝ // 0 < x } :=
  ⟨a.val / 2, by
```

```
    have h := a.property
    -- h : 0 < ↑a    ⊢ 0 < ↑a / 2
    positivity⟩

-- Exercise 4.
def cube (x : ℝ) : ℝ := x * x * x

example (x : ℝ) : cube (2 * x) = 8 * cube x := by
  unfold cube
  -- ⊢ 2 * x * (2 * x) * (2 * x) = 8 * (x * x * x)
  ring

-- Exercise 5.
def parityRel : Setoid ℕ where
  r a b := a % 2 = b % 2
  iseqv := {
    refl := fun _ ↦ rfl
    symm := fun h ↦ h.symm
    trans := fun h₁ h₂ ↦ h₁.trans h₂ }

def Parity : Type := Quotient parityRel

def Parity.mk (n : ℕ) : Parity := Quotient.mk parityRel n

example : Parity.mk 4 = Parity.mk 10 := by
  apply Quotient.sound
  -- ⊢ 4 ≈ 10
  rfl

-- Exercise 6.
example (h : ∀ x : ℝ, ∃ y : ℝ, x < y) : ∃ f : ℝ → ℝ, ∀ x, x < f x := by
  choose f hf using h
  -- f : ℝ → ℝ     hf : ∀ (x : ℝ), x < f x
  exact ⟨f, hf⟩

-- Exercise 7.
noncomputable def absR (x : ℝ) : ℝ := if 0 ≤ x then x else -x

example (x : ℝ) : absR x = absR (-x) := by
  unfold absR
  split_ifs with h₁ h₂ h₃
  · -- h₁ : 0 ≤ x,  h₂ : 0 ≤ -x     ⊢ x = -x
    linarith
  · -- h₁ : 0 ≤ x,  h₂ : ¬0 ≤ -x    ⊢ x = - -x
    ring
  · -- h₁ : ¬0 ≤ x, h₃ : 0 ≤ -x     ⊢ -x = -x
    rfl
  · -- h₁ : ¬0 ≤ x, h₃ : ¬0 ≤ -x    ⊢ -x = - -x
    linarith
```

In Exercise 7 the first and last cases are contradictory or degenerate, and `linarith` settles both: it treats a hypothesis `¬0 ≤ x` as `x < 0`. The one-line version is `split_ifs <;> linarith`.

Part III — Mathematics with Mathlib

Chapter 12: Finding Things in Mathlib

The lemma you need almost always exists, and four tools find it: `exact?` from the goal, the naming convention from the statement, Loogle from a pattern, and LeanSearch from plain English.

This is the skill that separates a frustrating week from a productive one. Mathlib has hundreds of thousands of lemmas, nobody remembers them, and everybody searches.

Which tool for which situation

You have	Use	Time
A goal that one lemma should close	`exact?`	Seconds
A goal that one lemma should reduce	`apply?`	Seconds
An equation one rewrite should transform	`rw?`	Seconds
A goal `simp` closes, and a wish to know how	`simp?`	Instant
A guess at the statement's shape	Guess the name, then `Ctrl+Space`	Seconds
A pattern such as `_ * _ ≤ _ * _`	Loogle	Seconds
A sentence: "the triangle inequality for integrals"	LeanSearch	Seconds
A topic to explore	The Mathlib documentation and the source file	Minutes

Let Lean search: `exact?`, `apply?`, `rw?`

`exact?` searches the library for a lemma that closes the goal using the hypotheses in context. The answer appears as a suggestion: click it, or press `Ctrl+.`, and it replaces the `exact?`.

```
import Mathlib

example (a b : ℝ) : |a + b| ≤ |a| + |b| := by
  exact?      -- Try this: exact abs_add_le a b

example (a b c : ℝ) (h : a ≤ b) : a + c ≤ b + c := by
  exact?      -- Try this: exact add_le_add_left h c

example (a b : ℝ) (ha : 0 ≤ a) (hb : 0 ≤ b) : 0 ≤ a * b := by
  exact?      -- Try this: exact Left.mul_nonneg ha hb
```

`apply?` also accepts lemmas that leave new goals. `rw?` looks for a rewrite.

```
example (a b c : ℝ) (hc : 0 < c) (h : a ≤ b) : a * c ≤ b * c := by
  apply?      -- Try this: exact (mul_le_mul_iff_of_pos_right hc).mpr h

example (a b : ℝ) : |a * b| = |a| * |b| := by
  rw?         -- Try this: rw [abs_mul]
```

`simp?` runs `simp` and reports which lemmas it used. This turns the simplifier into a search engine.

```
example (x : ℝ) (h : 0 < x) : x / x = 1 := by
  simp? [h.ne']
  -- Try this: simp only [ne_eq, h.ne', not_false_eq_true, div_self]
```

The output teaches you that the lemma is `div_self`, and that it needs `x ≠ 0`.

Three practical rules:

- **State exactly one step.** `exact?` finds a single lemma. If the gap is two lemmas wide, it fails. Isolate the step with `have h : statement := by exact?`.
- **Put the needed facts in context.** `exact?` uses hypotheses, so prove `0 < c` first if the lemma will need it.
- **Replace the call with its answer.** A file full of `exact?` is slow to check. Always accept the suggestion.

Guess the name

Mathlib names describe statements, by a fixed convention. Once you can read names, you can write them. The full rules are on the naming conventions page.

Rule 1. The name spells the statement, left to right, with a dictionary.

Symbol	Name	Symbol	Name	Symbol	Name
`+`	`add`	`≤`	`le`	`∧`	`and`
`-`	`sub`, or `neg` when unary	`<`	`lt`	`∨`	`or`
`*`	`mul`	`=`	`eq`, often omitted	`¬`	`not`
`/`	`div`	`≠`	`ne`	`↔`	`iff`
`^`	`pow`	`∈`	`mem`	`→`	`of`, or `imp`
`⁻¹`	`inv`	`∉`	`notMem`	`∃`	`exists`
`∣`	`dvd`	`⊆`	`subset`	`∀`	`forall`
`•`	`smul`	`∪`, `∩`	`union`, `inter`	`∘`	`comp`
`0`, `1`	`zero`, `one`	`ᶜ`	`compl`	`∑`, `∏`	`sum`, `prod`

Symbol	Name	Symbol	Name	Symbol	Name
`|x|`	`abs`	`‖x‖`	`norm`	`∫`	`integral`

```
#check abs_mul             -- |a * b| = |a| * |b|
#check add_sq              -- (a + b) ^ 2 = a ^ 2 + 2 * a * b + b ^ 2
#check neg_neg             -- - -a = a
#check sub_eq_add_neg      -- a - b = a + -b
#check two_mul             -- 2 * n = n + n
```

Rule 2. Sign conditions have short names.

Statement	Name	Example
`0 < a`	`pos`	`mul_pos : 0 < a → 0 < b → 0 < a * b`
`a < 0`	`neg`	`mul_neg_of_pos_of_neg`
`0 ≤ a`	`nonneg`	`abs_nonneg : 0 ≤ |a|`
`a ≤ 0`	`nonpos`	`mul_nonpos_iff`

Rule 3. The conclusion comes first, and hypotheses follow `of`.

```
#check @lt_of_lt_of_le     -- a < b → b ≤ c → a < c
#check @le_of_lt           -- a < b → a ≤ b
#check @ne_of_gt           -- b < a → a ≠ b
```

Read `lt_of_lt_of_le` as "`lt`, from `lt` and `le`".

Rule 4. Standard properties have standard suffixes.

Suffix	Meaning	Example
`_comm`, `_assoc`	Commutativity, associativity	`add_comm`, `mul_assoc`
`_refl`, `_symm`, `_trans`	Reflexivity, symmetry, transitivity	`le_refl`, `le_trans`
`_antisymm`	Antisymmetry	`le_antisymm : a ≤ b → b ≤ a → a = b`
`_iff`	The statement is an equivalence	`sub_nonneg : 0 ≤ a - b ↔ b ≤ a`
`_left`, `_right`	Which argument is concerned	`le_max_left : a ≤ max a b`
`_self`	An argument is repeated	`mul_self_nonneg : 0 ≤ a * a`
`_cancel`	Something cancels	`sub_add_cancel : a - b + b = a`
`₀`	Version for fields and other structures with zero	`div_le_iff₀`

Suffix	Meaning	Example
`'`	A variant of the unprimed lemma	`h.ne'`

Rule 5. Capitalization tells you what kind of thing it is.

Style	Used for	Examples
`snake_case`	Theorems and proofs	`add_comm`, `abs_nonneg`
`UpperCamelCase`	Types, propositions, structures, classes	`Continuous`, `Finset`, `IsCompact`
`lowerCamelCase`	Functions and other data	`Finset.range`, `Real.sqrt`, `deriv`

A theorem about a camel-case notion lowers its first letter inside the snake-case name: `Continuous.add`, `continuous_id`, `isCompact_Icc`.

Rule 6. The namespace is the main subject. Lemmas about sums over finite sets start with `Finset.sum_`, lemmas about real square roots with `Real.sqrt_`, and lemmas about natural numbers with `Nat.`.

```
#check Finset.sum_range_succ
-- ∑ x ∈ Finset.range (n + 1), f x = ∑ x ∈ Finset.range n, f x + f n

#check @Finset.sum_le_sum
-- (∀ i ∈ s, f i ≤ g i) → ∑ i ∈ s, f i ≤ ∑ i ∈ s, g i
```

Use completion to confirm a guess

Type the start of a guessed name and press `Ctrl+Space`. The completion list shows every name with that prefix, each with its statement.

Typing `abs_sub` lists `abs_sub_comm`, `abs_sub_le`, `abs_sub_lt_iff`, `abs_sub_abs_le_abs_sub` and more. Arrow down the list and read the statements. Half of all searches end here.

Dot notation finds lemmas for you

Many lemmas are meant to be called on a hypothesis. If `h : a < b`, then typing `h.` and pressing `Ctrl+Space` lists what follows from it.

```
example (a b : ℝ) (h : a < b) : a ≤ b := h.le
example (a b : ℝ) (h : a < b) : a ≠ b := h.ne
example (a b : ℝ) (h : a < b) : b ≠ a := h.ne'
example (a b c : ℝ) (h₁ : a ≤ b) (h₂ : b ≤ c) : a ≤ c := h₁.trans h₂
example (a b c : ℝ) (h₁ : a < b) (h₂ : b ≤ c) : a < c := h₁.trans_le h₂
example (a b : ℝ) (h : a ≤ b) (h' : b ≤ a) : a = b := h.antisymm h'
```

An `iff` lemma is used in one direction with `.mp` or `.mpr`, or as a rewrite rule:

```
example (a b : ℝ) (h : 0 ≤ a - b) : b ≤ a := sub_nonneg.mp h
example (a b : ℝ) (h : b ≤ a) : 0 ≤ a - b := sub_nonneg.mpr h
example (a b : ℝ) (h : b ≤ a) : 0 ≤ a - b := by rwa [sub_nonneg]
```

Loogle: search by shape

Loogle searches Mathlib by the form of the statement. Open it in the browser, or inside VS Code with `Ctrl+K Ctrl+S`. With text selected in the editor, the shortcut searches for the selection.

Query	Finds
`Real.sin`	Lemmas whose statement mentions the constant `Real.sin`
`"differ"`	Lemmas with `differ` in their name
`_ * (_ ^ _)`	Lemmas containing a subexpression of that shape
`Real.sqrt ?a * Real.sqrt ?a`	The same, where both `?a` must be the same term
`|- tsum _ = _ * tsum _`	Lemmas whose conclusion has that shape
`|- _ < _ → tsum _ < tsum _`	Lemmas with that conclusion and that hypothesis
`Real.sin, "two", _ * _`	Lemmas satisfying all three filters

The underscore `_` matches anything. The turnstile `|-`, or `⊢`, restricts the pattern to the conclusion. Filters are separated by commas, and all must match.

The same search is available as a command inside a Lean file, through the package LeanSearchClient that Mathlib imports:

```
#loogle |- |_ + _| ≤ |_| + |_|
```

A good habit: copy the goal from the Infoview, replace the variables by `_`, put `|-` in front, and search.

LeanSearch: search in words

LeanSearch takes a sentence in natural language and returns Mathlib statements ranked by relevance. It is the right tool when you know the mathematics and not the Lean vocabulary.

Queries such as "a continuous function on a compact set is bounded" or "derivative of a composition" return the lemma names with their statements. LeanExplore is a second engine of the same kind.

Inside a Lean file, the query must end with a full stop or a question mark:

```
#leansearch "If a natural number n is less than m, then the successor of n
is less than the successor of m."
```

The documentation

The Mathlib documentation has one page per file, generated from the source. Its search box matches declaration names.

A documentation page shows each declaration with its full statement, its docstring, and a link to the source line. Every identifier in a statement is a link to its own definition.

Two curated entry points are faster than search when you explore a topic:

- The overview and the undergraduate list map mathematical topics to the declarations that formalize them.
- The Mathlib Phrasebook shows how to state common mathematical objects.

Read the source

When you find one relevant lemma, its neighbours are usually what you need next. `Ctrl+click` on a name, or `F12`, opens the file at that lemma.

A Mathlib file follows a pattern: a header comment listing the main definitions and results, then a definition, then dozens of short lemmas about it. Reading twenty lines around a lemma teaches you the local vocabulary.

Inside the editor, three commands inspect a name:

```
#check abs_le           -- the statement, all arguments before the colon
#check @abs_le          -- the same statement as one ∀ type, with instance
names
#print abs_le           -- the statement and the proof term
```

Names change

Mathlib renames lemmas to keep the conventions uniform. The old name is kept for a few months with a deprecation warning that gives the new one, and is then deleted.

All three stages are visible in the version this book uses:

```
#check (Set.mem_setOf_eq)   -- the parentheses matter: a bare #check name
skips the warning
-- warning: `Set.mem_setOf_eq` has been deprecated: Use `Set.mem_ofPred_eq`
instead

#check Set.not_mem_empty
-- error: Unknown constant `Set.not_mem_empty`     (it is now
Set.notMem_empty)

#check abs_add
-- error: Unknown identifier `abs_add`             (it is now abs_add_le)
```

When a name from a book, a paper or an AI assistant does not resolve, do not guess variants. State what the lemma should say and run `exact?`, or search its shape with Loogle. Both find the current name in seconds.

The meaning of a name can change too. In this version `add_le_add_left h c` proves `a + c ≤ b + c`, and in older versions it proved `c + a ≤ c + b`. Trust `#check`, not memory.

A one-minute procedure

1. Write the step as a `have` with the exact statement you want, and try `exact?`.
2. If that fails, try the automation of Chapter 13: `simp`, `linarith`, `positivity`, `gcongr`.
3. If that fails, guess the name and browse with `Ctrl+Space`.
4. If that fails, search the shape in Loogle, then the idea in LeanSearch.
5. If that fails, the statement may need a hypothesis you forgot, or may be false. Test it on an example.
6. If it is true and absent, ask in the Is there code for X? channel on Zulip. Then prove it yourself.

Exercises

```
import Mathlib

-- Exercises 1 to 7. Each is closed by one lemma. First guess its name from
the
-- convention. Then check your guess with exact?.

example (a b : ℝ) : |a * b| = |a| * |b| := by
  sorry

example (a b : ℝ) (h : a ≤ b) : -b ≤ -a := by
  sorry

example (a b : ℝ) (ha : 0 < a) (hb : 0 < b) : 0 < a * b := by
  sorry

example (a b c : ℝ) (h₁ : a < b) (h₂ : b ≤ c) : a < c := by
  sorry

example (x : ℝ) : 0 ≤ x ^ 2 := by
  sorry

example (a b : ℝ) : |a| - |b| ≤ |a - b| := by
  sorry

example (n : ℕ) :
    ∑ i ∈ Finset.range (n + 1), (i : ℝ) = ∑ i ∈ Finset.range n, (i : ℝ) +
n := by
  sorry

-- Exercise 8. Without running Lean, write the statement each name
describes.
-- Then run #check.
--   sub_nonneg      mul_self_nonneg      le_div_iff₀      abs_sub_comm

-- Exercise 9. Write a Loogle query for each.
```

```
--   (a) lemmas whose conclusion is  |_ - _| ≤ |_ - _| + |_ - _|
--   (b) lemmas about Finset.sum with "comm" in the name
--   (c) lemmas stating that the square root of a square is something
```

Solutions

```
import Mathlib

example (a b : ℝ) : |a * b| = |a| * |b| := abs_mul a b

example (a b : ℝ) (h : a ≤ b) : -b ≤ -a := neg_le_neg h

example (a b : ℝ) (ha : 0 < a) (hb : 0 < b) : 0 < a * b := mul_pos ha hb

example (a b c : ℝ) (h₁ : a < b) (h₂ : b ≤ c) : a < c := lt_of_lt_of_le h₁
h₂

example (x : ℝ) : 0 ≤ x ^ 2 := sq_nonneg x

example (a b : ℝ) : |a| - |b| ≤ |a - b| := abs_sub_abs_le_abs_sub a b

example (n : ℕ) :
    ∑ i ∈ Finset.range (n + 1), (i : ℝ) = ∑ i ∈ Finset.range n, (i : ℝ) +
n :=
  Finset.sum_range_succ _ n

-- Exercise 8.
#check @sub_nonneg          -- 0 ≤ a - b ↔ b ≤ a
#check @mul_self_nonneg     -- 0 ≤ a * a
#check @le_div_iff₀         -- 0 < c → (a ≤ b / c ↔ a * c ≤ b)
#check @abs_sub_comm        -- |a - b| = |b - a|
```

`exact?` may answer with a different lemma that proves the same thing. For the second exercise it suggests `neg_le_neg_iff.mpr h`, and for the fourth `Std.lt_of_lt_of_le h₁ h₂`. Any accepted answer is correct.

For Exercise 9:

- (a) `|- |_ - _| ≤ |_ - _| + |_ - _|`, which finds `abs_sub_le`.
- (b) `Finset.sum, "comm"`, which finds `Finset.sum_comm` among others.
- (c) `Real.sqrt (?a ^ 2)`, or `Real.sqrt (?a * ?a)` for the other spelling of a square.

Chapter 13: Automation

Each automation tactic solves one well-defined class of goals, and knowing the class tells you which tactic to call and why it failed.

None of them is a general prover. They are specialists, and a proof is you directing specialists.

The map

Tactic	Proves	Works in	Does not
`ring`	Equalities that follow from the ring axioms	Commutative rings and semirings	Use hypotheses
`field_simp`	Nothing by itself: clears denominators	Fields	Know that a denominator is nonzero unless told
`linear_combination`	Equalities that follow from other equalities	Commutative rings	Find the coefficients
`abel`, `group`	Equalities in abelian groups, in groups	Groups	Handle ring multiplication
`linarith`	Linear inequalities that follow from the hypotheses	Ordered fields	Multiply two unknowns
`nlinarith`	The same, after multiplying pairs of hypotheses	Ordered fields	Guess the right products every time
`positivity`	`0 < e`, `0 ≤ e`, `e ≠ 0`, by the form of `e`	Ordered rings	Handle most subtractions
`gcongr`	`f a ≤ f b` from `a ≤ b`, for a visible pattern `f`	Orders	Rearrange the two sides
`norm_num`	Statements about explicit numbers	Number types	Reason about variables
`omega`	Linear arithmetic over `ℕ` and `ℤ`, with `%` and `/` by constants	`ℕ`, `ℤ` only	Multiply two unknowns, or touch `ℝ`

Tactic	Proves	Works in	Does not
`decide`	Closed statements, by running a decision procedure	Finite, computable cases	Scale, or work on $\mathbb{R}$
`simp`	Goals that rewrite to `True` using tagged lemmas	Everything	Prove what no simp lemma covers
`tauto`	Propositional tautologies	Logic	Use quantifiers cleverly
`aesop`	Goals reachable by intro, cases, constructor and simp	Logic, sets	Arithmetic
`grind`	Goals reachable by congruence, linear arithmetic and ring reasoning	Mixed	Nonlinear inequalities

The sections below take them by kind of goal.

Equalities: `ring`

`ring` proves an equality when both sides are the same polynomial. It works in every commutative ring, and in $\mathbb{N}$.

```
import Mathlib

example (a b : ℝ) : (a + b) ^ 2 = a ^ 2 + 2 * a * b + b ^ 2 := by ring
example (a b : ℝ) : (a - b) * (a + b) = a ^ 2 - b ^ 2 := by ring
example (x : ℝ) : (x + 1) / 2 = x / 2 + 1 / 2 := by ring
example (a b : ℕ) : (a + b) ^ 2 = a ^ 2 + 2 * a * b + b ^ 2 := by ring
```

`ring` ignores hypotheses. This fails:

```
example (a b : ℝ) (h : a = b + 1) : a ^ 2 = b ^ 2 + 2 * b + 1 := by
  ring
-- error: unsolved goals  ⊢ a ^ 2 = 1 + b * 2 + b ^ 2
```

There are three fixes, in increasing power.

```
-- 1. Rewrite with the hypothesis first.
example (a b : ℝ) (h : a = b + 1) : a ^ 2 = b ^ 2 + 2 * b + 1 := by
  rw [h]
  ring

-- 2. Eliminate the variable.
example (a b : ℝ) (h : a = b + 1) : a ^ 2 = b ^ 2 + 2 * b + 1 := by
  subst h
  ring
```

```
-- 3. Say which multiple of the hypothesis gives the goal.
example (a b : ℝ) (h : a = b + 1) : a ^ 2 = b ^ 2 + 2 * b + 1 := by
  linear_combination (a + b + 1) * h
```

`linear_combination e₁ * h₁ + e₂ * h₂` checks, with `ring`, that the goal equals that combination of the hypotheses. You supply the coefficients, and the tactic verifies them.

When `ring` fails, it suggests `ring_nf`, which puts both sides in normal form and shows what is left. The leftover tells you what is wrong:

```
example (a b c : ℝ) : a * (b + c) + 0 = a * b + a * c + 1 := by
  ring_nf
  -- ⊢ a * b + a * c = 1 + a * b + a * c       the statement is false
  sorry
```

Equalities with division: `field_simp`

`ring` treats `x⁻¹` as an opaque symbol, so it cannot prove `x / x = 1`. Indeed that is false at `x = 0` under Lean's convention `0 / 0 = 0`.

`field_simp` multiplies through by the denominators. It needs to know they are nonzero, and looks for hypotheses of the form `x ≠ 0` in the context.

```
example (x : ℝ) (hx : x ≠ 0) : (x ^ 2 + x) / x = x + 1 := by
  field_simp

example (x y : ℝ) (hx : x ≠ 0) (hy : y ≠ 0) :
    1 / x + 1 / y = (x + y) / (x * y) := by
  field_simp
  -- ⊢ y + x = x + y
  ring
```

The standard pattern for identities with fractions is `field_simp` followed by `ring`.

Inequalities: `linarith`

`linarith` proves a linear inequality or equality that follows from the hypotheses by adding them with nonnegative coefficients. It is complete for that problem over ordered fields.

```
example (a b c : ℝ) (h₁ : a < b) (h₂ : b ≤ c) : a < c := by linarith

example (x y : ℝ) (h₁ : 2 * x + y ≤ 5) (h₂ : x - y ≤ 1) : x ≤ 2 := by
linarith

example (ε : ℝ) (hε : 0 < ε) : ε / 2 < ε := by linarith

example (x y : ℝ) (h : x < y) (h' : y < x) : False := by linarith
```

It treats every nonlinear term as an unknown. So it cannot use `0 < x` and `0 < y` to get `0 < x * y`, and it knows nothing about `|x|`, `max` or functions.

The remedy is to hand it the missing facts in brackets. Each one is a proof term, added to the hypotheses for this call only.

```
example (a b : ℝ) : 2 * a * b ≤ a ^ 2 + b ^ 2 := by
  linarith [sq_nonneg (a - b)]

example (a b : ℝ) (h : |a - b| < 1) (hb : b = 2) : a < 3 := by
  linarith [abs_lt.mp h]

example (a b : ℝ) (f : ℝ → ℝ) (hf : Monotone f) (h : a ≤ b) : f a ≤ f b +
1 := by
  linarith [hf h]
```

In the first example, `sq_nonneg (a - b)` is the fact `0 ≤ (a - b) ^ 2`. After expansion it is linear in the unknowns `a ^ 2`, `a * b` and `b ^ 2`, and so is the goal.

This is the working method for inequalities: find the one or two nonlinear facts the proof rests on, and let `linarith` do the bookkeeping.

Inequalities: `nlinarith`

`nlinarith` runs `linarith` after adding products of pairs of hypotheses and the fact that each square appearing in the problem is nonnegative. It often finds simple nonlinear arguments by itself:

```
example (x y : ℝ) (hx : 0 < x) (hy : 0 < y) : 0 < x * y := by nlinarith

example (a b : ℝ) (ha : 0 ≤ a) (hb : 0 ≤ b) (h : a ≤ b) : a ^ 2 ≤ b ^ 2 :=
by nlinarith

example (x : ℝ) (h : 2 ≤ x) : 4 ≤ x ^ 2 := by nlinarith
```

When it fails, give it the squares the proof needs. Proofs of polynomial inequalities are usually "a sum of squares is nonnegative", and you name the squares.

```
example (a b : ℝ) : a * b ≤ ((a + b) / 2) ^ 2 := by
  nlinarith [sq_nonneg (a - b)]

example (a b c : ℝ) : a * b + b * c + c * a ≤ a ^ 2 + b ^ 2 + c ^ 2 := by
  nlinarith [sq_nonneg (a - b), sq_nonneg (b - c), sq_nonneg (c - a)]
```

Signs: `positivity`

`positivity` proves goals of the form `0 < e`, `0 ≤ e` or `e ≠ 0` by looking at how `e` is built: sums and products of positive things, even powers, absolute values, and so on.

```
example (x y : ℝ) : 0 ≤ x ^ 2 + y ^ 2 := by positivity
example (x : ℝ) : 0 < x ^ 2 + 1 := by positivity
example (x : ℝ) (hx : 0 < x) : 0 < x / (x + 1) := by positivity
example (ε : ℝ) (hε : 0 < ε) : 0 < ε / 3 := by positivity
example (n : ℕ) : 0 < (n : ℝ) + 1 := by positivity
```

It uses hypotheses such as `0 < x` for the variables. It gives up when positivity is not visible in the form of the expression:

```
example (x : ℝ) : 0 ≤ x ^ 2 - x + 1 := by positivity
-- error: failed to prove positivity/nonnegativity/nonzeroness

example (x : ℝ) : 0 ≤ x ^ 2 - x + 1 := by nlinarith [sq_nonneg (x - 1)]
```

In ε-arguments, `positivity` is what proves `0 < ε / 2`, `0 < δ * ε` and `0 < min δ₁ δ₂`.

Monotonicity: `gcongr`

`gcongr` proves an inequality between two expressions of the same shape by reducing it to inequalities between the parts that differ.

```
example (a b c d : ℝ) (h₁ : a ≤ b) (h₂ : c ≤ d) : a + c ≤ b + d := by gcongr

example (a b c : ℝ) (hc : 0 ≤ c) (h : a ≤ b) : a * c ≤ b * c := by gcongr

example (a b : ℝ) (ha : 0 ≤ a) (h : a ≤ b) : a ^ 3 ≤ b ^ 3 := by gcongr

example (a b : ℝ) (ha : 0 < a) (h : a < b) : 1 / b < 1 / a := by gcongr
```

The two sides must match except at the places where `a` became `b`. Side conditions such as `0 ≤ c` are discharged by `positivity` and by hypotheses. If one is missing, it is returned as a goal.

Under a sum, `gcongr with i hi` names the bound variable and its membership hypothesis:

```
example (f g : ℕ → ℝ) (n : ℕ) (h : ∀ i, f i ≤ g i) :
    ∑ i ∈ Finset.range n, f i ≤ ∑ i ∈ Finset.range n, g i := by
  gcongr with i hi
  -- hi : i ∈ Finset.range n    ⊢ f i ≤ g i
  exact h i
```

`gcongr` is the natural companion of `calc`. Each line of an estimate changes one thing, and `gcongr` justifies the line.

Explicit numbers: `norm_num`

`norm_num` evaluates arithmetic on numerals and proves equalities, inequalities and divisibility between them, in `ℕ`, `ℤ`, `ℚ`, `ℝ` and `ℂ`.

```
example : (2 : ℝ) ^ 10 = 1024 := by norm_num
example : (1 : ℝ) / 3 + 1 / 6 = 1 / 2 := by norm_num
example : (7 : ℝ) / 2 < 4 := by norm_num
example : (3 : ℝ) ≠ 0 := by norm_num
example : |(-3 : ℝ)| = 3 := by norm_num
example : (12345 : ℤ) % 7 = 4 := by norm_num
example : Nat.Prime 101 := by norm_num
```

It also runs `simp`, so it accepts extra lemmas and hypotheses in brackets, and it simplifies before computing:

```
example (x : ℝ) (h : x = 2) : x ^ 2 + 1 = 5 := by norm_num [h]

example : ∑ i ∈ Finset.range 5, (i : ℝ) = 10 := by
  norm_num [Finset.sum_range_succ]
```

Integers and naturals: `omega`

`omega` decides linear arithmetic over $\mathbb{N}$ and $\mathbb{Z}$. It understands truncated subtraction on $\mathbb{N}$, and division and remainder by numerals.

```
example (n m : ℕ) (h : n < m) : n + 1 ≤ m := by omega
example (n : ℕ) (h : n % 2 = 1) : (n + 1) % 2 = 0 := by omega
example (n : ℕ) (h : 0 < n) : n - 1 + 1 = n := by omega
example (x y : ℤ) (h : 2 * x + 1 = 2 * y) : False := by omega
```

Use it for every index computation: bounds of `Finset.range`, lengths of lists, off-by-one steps in inductions. It does nothing on $\mathbb{R}$, where `linarith` is the tool, and nothing with a product of two variables.

For small finite checks, `decide` runs the computation, and `interval_cases` splits a bounded variable into its possible values:

```
example : 7 | 91 := by decide

example (n : ℕ) (h : n < 4) : n ^ 2 < 10 := by
  interval_cases n <;> norm_num
```

`simp`, in depth

`simp` rewrites the goal repeatedly, left to right, with the lemmas tagged `@[simp]` in the library, until nothing applies. Mathlib has tens of thousands of such lemmas. If the goal becomes `True`, it is closed.

Form	Effect
`simp`	Simplify the goal with the default simp set
`simp [h, foo]`	Also rewrite with the hypothesis `h` and the lemma `foo`
`simp [← foo]`	Use `foo` from right to left
`simp [-foo]`	Do not use the simp lemma `foo`
`simp [myDef]`	Unfold the definition `myDef`
`simp at h`	Simplify the hypothesis `h`
`simp at *`	Simplify the goal and every hypothesis
`simp only [a, b]`	Rewrite with exactly these lemmas and nothing else
`simp?`	Run `simp` and print the equivalent `simp only [...]`
`simpa using e`	Simplify the goal and `e`, then close the goal with `e`
`simp_all`	Simplify everything, using hypotheses to simplify each other
`dsimp`	Simplify by definitional unfolding only

```
example (a b : ℝ) (h : a = 0) : a * b + b = b := by simp [h]

example (f : ℕ → ℕ) (h : ∀ n, f n = n + 1) : f 2 = 3 := by simp [h]

example (a b : ℕ) (h : a = b) : a + 0 = b * 1 := by
  simpa using h
```

In the last example, `simp` turns the goal into `a = b`, which is `h`. That is what `simpa using h` does in one step.

Use `simp` to finish, not in the middle. A bare `simp` in the middle of a proof produces a goal that depends on the current simp set. When Mathlib adds a simp lemma, the goal changes and the next tactic breaks.

Projects created with the `math` template enable Mathlib's linters, which flag this pattern:

```
warning: `simp at h'` is a flexible tactic modifying `h'`. Try `simp?` and
use the
suggested `simp only [...]`. Alternatively, use `suffices` to explicitly
state the simplified form.
```

The fix is the one the warning names. Run `simp?`, and accept its `simp only [...]`:

```
example (a b : ℕ) (h : a = b) : a + 0 = b * 1 := by
  simp only [add_zero, mul_one]      -- produced by simp?
  -- ⊢ a = b
  exact h
```

You can teach `simp` your own rules by tagging a lemma. Choose lemmas whose right side is simpler than the left:

```
def triple (x : ℝ) : ℝ := 3 * x

@[simp] theorem triple_zero : triple 0 = 0 := by simp [triple]

example (y : ℝ) : (triple 0 + y) * 1 = y := by simp   -- fails without the
@[simp] tag
```

The guide on the simplifier covers its options and pitfalls.

Logic and search: `tauto`, `aesop`, `grind`

`tauto` proves propositional tautologies. `aesop` is a general search: it introduces, splits hypotheses, applies constructors and simplifies, with backtracking. It is good at goals about sets and logic.

```
example (p q : Prop) : ¬(p ∧ q) ↔ ¬p ∨ ¬q := by tauto

example (p q r : Prop) (h : p ∧ (q ∨ r)) : (p ∧ q) ∨ (p ∧ r) := by aesop

example (α : Type) (s t : Set α) : s ∩ t ⊆ s := by aesop
```

`grind` is the newest general tactic, built into Lean. It combines congruence closure, linear arithmetic, ring normalization and instantiation of library lemmas, in the style of an SMT solver.

```
example (f : ℕ → ℕ) (a b : ℕ) (h₁ : f a = b) (h₂ : a = b) : f (f a) = b :=
by grind

example (a b c : ℤ) (h₁ : a < b) (h₂ : b < c) : a + 2 ≤ c := by grind

example (a b : ℝ) (h : a < b) : a < (a + b) / 2 := by grind

example (xs ys : List ℕ) : (xs ++ ys).length = ys.length + xs.length := by
grind
```

It does not replace the specialists. On the nonlinear inequality `2 * a * b ≤ a ^ 2 + b ^ 2` it fails, and prints the model it could not refute, where `nlinarith [sq_nonneg (a - b)]` succeeds.

Two commands try several tactics for you and report which ones work. Use them when you have no idea, then keep the suggestion.

```
example (a b : ℝ) (h : a ≤ b) : a ≤ b + 1 := by
  try?        -- Try these:  grind    grind only
```

The Mathlib tactic `hint` does the same with `simp_all`, `norm_num`, `linarith`, `positivity` and others.

A worked estimate

The statement: if `a` is within `ε / 2` of `b`, and `b` within `ε / 2` of `c`, then `a` is within `ε` of `c`.

```
example (a b c ε : ℝ) (h₁ : |a - b| < ε / 2) (h₂ : |b - c| < ε / 2) :
    |a - c| < ε := by
  calc |a - c| = |(a - b) + (b - c)| := by ring_nf
    _ ≤ |a - b| + |b - c| := abs_add_le _ _
    _ < ε / 2 + ε / 2 := by gcongr
    _ = ε := by ring
```

Each line uses one tool for one reason. `ring_nf` rewrites inside the absolute value. `abs_add_le` is the triangle inequality, found with `exact?`. `gcongr` adds the two strict inequalities. `ring` simplifies the bound.

The same proof with a fact handed to `linarith` is shorter and less readable:

```
example (a b c ε : ℝ) (h₁ : |a - b| < ε / 2) (h₂ : |b - c| < ε / 2) :
    |a - c| < ε := by
  have h := abs_sub_le a b c      -- h : |a - c| ≤ |a - b| + |b - c|
  linarith
```

Both styles are legitimate. Use `calc` when the reader should follow the estimate, and `linarith [...]` when the step is routine.

When automation fails

A failure message says what the tactic was missing.

Message	What it tells you	Next move
`linarith failed to find a contradiction`, with the goal negated as `a+`	No linear combination of the listed hypotheses works	Find the missing nonlinear fact and pass it in brackets
`unsolved goals` after a `Try this: ring_nf` suggestion	The two sides are different polynomials	Run `ring_nf`, read what is left: a hypothesis is needed, or the statement is wrong
`failed to prove positivity/nonnegativity/nonzeroness`	Some subterm has no visible sign	Prove that subterm's sign with `have`, or switch to `nlinarith`
`not a positivity goal`	The goal is not of the form `0 < e`, `0 ≤ e`, `e ≠ 0`	Reshape the goal, for example with `sub_nonneg`
`omega could not prove the goal: No usable constraints found`	The goal is not about `ℕ` or `ℤ`	Use `linarith`, or cast with Chapter 14
`omega could not prove the goal: a possible counterexample…`	A nonlinear term is treated as an unknown, or the statement is false	Check the statement on the counterexample
`simp made no progress`	No simp lemma applies	Add the lemma or hypothesis in brackets, or unfold a definition
`gcongr did not make progress`	The two sides do not have the same shape	Rewrite one side first, or split with `calc`

When everything fails on a goal you believe, split it. Introduce one intermediate `have` and try again on each half. Automation that fails on a two-step gap usually succeeds on each step.

An order of attack

1. Is it an equality of polynomial expressions? `ring`. With fractions, `field_simp` then `ring`.
2. Is it about explicit numbers? `norm_num`.
3. Is it about $\mathbb{N}$ or $\mathbb{Z}$, and linear? `omega`.
4. Is it `0 < e` or `0 ≤ e`? `positivity`.
5. Do both sides have the same shape? `gcongr`.
6. Is it a linear consequence of the hypotheses? `linarith`, with extra facts in brackets.
7. Is it a polynomial inequality? `nlinarith`, with squares in brackets.
8. Is it logic or sets? `tauto`, `aesop`.
9. Otherwise `simp`, then `grind`, then `exact?`.

Exercises

```
import Mathlib

-- Each goal is closed by one tactic from this chapter, sometimes with a
fact in
-- brackets or one preparatory step. Choose the tactic before you type it.

example (a b : ℝ) : (a + b) ^ 3 = a ^ 3 + 3 * a ^ 2 * b + 3 * a * b ^ 2 + b
^ 3 := by
  sorry

example (x y : ℝ) (h₁ : x + y = 10) (h₂ : x - y = 2) : x = 6 := by
  sorry

example : (3 : ℝ) / 4 + 1 / 4 = 1 := by
  sorry

example (n : ℕ) (h : 3 ≤ n) : n - 2 + 2 = n := by
  sorry

example (x : ℝ) : 0 < x ^ 4 + x ^ 2 + 1 := by
  sorry

example (a b c : ℝ) (h : a ≤ b) (hc : 0 ≤ c) : a * c + 1 ≤ b * c + 1 := by
  sorry

example (x : ℝ) (hx : x ≠ 0) : x ^ 2 / x = x := by
  sorry

example (p q : Prop) : p ∧ q → q ∧ p := by
  sorry

example (x y : ℝ) : x * y ≤ (x ^ 2 + y ^ 2) / 2 := by
```

```
  sorry

example (a b : ℝ) (h : a = 2 * b) (h' : b = 3) : a ^ 2 = 36 := by
  sorry

-- Two tactics, one per component.
example (ε : ℝ) (hε : 0 < ε) : ∃ δ > 0, δ < ε := by
  refine ⟨ε / 2, ?_, ?_⟩
  · sorry
  · sorry

-- Three steps: a sign fact, a rewrite with div_lt_one, a linear conclusion.
example (x y : ℝ) (hx : 0 < x) (hy : 0 < y) : x / (x + y) < 1 := by
  sorry

-- Factor first: (x - 1) * (x - 2) = 0. Then use mul_eq_zero.
example (x : ℝ) (h : x ^ 2 - 3 * x + 2 = 0) : x = 1 ∨ x = 2 := by
  sorry
```

Solutions

```
import Mathlib

example (a b : ℝ) : (a + b) ^ 3 = a ^ 3 + 3 * a ^ 2 * b + 3 * a * b ^ 2 + b
^ 3 := by
  ring

example (x y : ℝ) (h₁ : x + y = 10) (h₂ : x - y = 2) : x = 6 := by
  linarith

example : (3 : ℝ) / 4 + 1 / 4 = 1 := by
  norm_num

example (n : ℕ) (h : 3 ≤ n) : n - 2 + 2 = n := by
  omega

example (x : ℝ) : 0 < x ^ 4 + x ^ 2 + 1 := by
  positivity

example (a b c : ℝ) (h : a ≤ b) (hc : 0 ≤ c) : a * c + 1 ≤ b * c + 1 := by
  gcongr

example (x : ℝ) (hx : x ≠ 0) : x ^ 2 / x = x := by
  field_simp

example (p q : Prop) : p ∧ q → q ∧ p := by
  tauto

example (x y : ℝ) : x * y ≤ (x ^ 2 + y ^ 2) / 2 := by
  nlinarith [sq_nonneg (x - y)]

example (a b : ℝ) (h : a = 2 * b) (h' : b = 3) : a ^ 2 = 36 := by
  rw [h, h']
  -- ⊢ (2 * 3) ^ 2 = 36
  norm_num

example (ε : ℝ) (hε : 0 < ε) : ∃ δ > 0, δ < ε := by
  refine ⟨ε / 2, ?_, ?_⟩
  · positivity
```

```
  · linarith

example (x y : ℝ) (hx : 0 < x) (hy : 0 < y) : x / (x + y) < 1 := by
  have h : 0 < x + y := by positivity
  rw [div_lt_one h]
  -- ⊢ x < x + y
  linarith

example (x : ℝ) (h : x ^ 2 - 3 * x + 2 = 0) : x = 1 ∨ x = 2 := by
  have h' : (x - 1) * (x - 2) = 0 := by linarith
  -- mul_eq_zero : a * b = 0 ↔ a = 0 ∨ b = 0
  rcases mul_eq_zero.mp h' with h1 | h2
  · left
    linarith
  · right
    linarith
```

In the last solution, `linarith` proves `(x - 1) * (x - 2) = 0` from `h` because, once expanded, both are the same linear relation between the unknowns `x ^ 2` and `x`.

Chapter 14: Numbers, Casts and Their Traps

`ℕ`, `ℤ`, `ℚ`, `ℝ` and `ℂ` are five separate types joined by explicit maps, and most beginner errors about numbers come from an operation performed in the wrong type.

The number types

Type	Typed	What to know
`ℕ`	`\N`	Subtraction stops at 0. Division rounds down. Computable
`ℤ`	`\Z`	Division rounds toward minus infinity for a positive divisor. Computable
`ℚ`	`\Q`	Exact fractions. Computable
`ℝ`	`\R`	A complete ordered field. Not computable: proofs only
`ℂ`	`\C`	A pair of reals `re`, `im`. No order
`ℝ≥0`	`\R\ge0`, after `open scoped NNReal`	Nonnegative reals, as a subtype of `ℝ`
`ℝ≥0∞`	after `open scoped ENNReal`	`[0, ∞]`. The values of measures
`Fin n`		The natural numbers below `n`

The inclusions `ℕ → ℤ → ℚ → ℝ → ℂ` are functions, inserted automatically and displayed as `↑`.

A numeral takes its type from its surroundings

```
import Mathlib

#check 2                  -- 2 : ℕ
#check (2 : ℝ)            -- 2 : ℝ
#check (2 : ℝ) + 2        -- 2 + 2 : ℝ
#check (1 / 2 : ℝ)        -- 1 / 2 : ℝ
```

With no information, a numeral is a natural number. An ascription on a whole expression reaches inside it: in `(1 / 2 : ℝ)`, both `1` and `2` are real, and the division is real division.

The trap is the reverse order, a computation in `ℕ` followed by a cast:

```
#eval (1 / 2 : ℕ)                 -- 0
#eval (1 / 2 : ℚ)                 -- 1 / 2

example : (1 / 2 : ℝ) = 0.5 := by norm_num
example : ((1 / 2 : ℕ) : ℝ) = 0 := by norm_num
```

Both statements are true. The second one divides in `ℕ`, gets `0`, and then casts `0` to `ℝ`.

Where you put the type decides where the cast goes. `#check` shows the result:

```
#check fun (n : ℕ) ↦ (n / 2 : ℝ)            -- fun n ↦ ↑n / 2        real
division
#check fun (n : ℕ) ↦ ((n / 2 : ℕ) : ℝ)      -- fun n ↦ ↑(n / 2)      floor,
then cast
#check fun (n m : ℕ) ↦ (n + m : ℝ)          -- fun n m ↦ ↑n + ↑m
#check fun (n m : ℕ) ↦ ((n + m : ℕ) : ℝ)    -- fun n m ↦ ↑(n + m)
```

Rule. When a statement mixes `ℕ` with `ℝ`, check it with `#check` or read it in the Infoview before proving it. Look at where each `↑` sits.

Subtraction on `ℕ`

```
#eval (2 - 5 : ℕ)                  -- 0

example (n : ℕ) (h : 1 ≤ n) : n - 1 + 1 = n := Nat.sub_add_cancel h
example (n : ℕ) (h : 1 ≤ n) : n - 1 + 1 = n := by omega
```

Subtraction on `ℕ` is truncated: `a - b = 0` whenever `a ≤ b`. So `n - 1 + 1 = n` is false for `n = 0`, and every lemma about natural subtraction carries a hypothesis such as `m ≤ n`.

Three ways to live with it:

- **Avoid it.** State `a = b + c` instead of `a - c = b`. State `∀ n, P (n + 1)` instead of `∀ n ≥ 1, P n` with `n - 1` inside.
- **Let `omega` handle it.** It knows truncated subtraction exactly.
- **Move to `ℤ` or `ℝ`**, where subtraction is honest, with the cast tactics below.

Division

```
#eval (7 / 2 : ℕ)                  -- 3
#eval (7 % 2 : ℕ)                  -- 1
#eval (-7 / 2 : ℤ)                 -- -4

example (x : ℝ) : x / 0 = 0 := div_zero x
example : (0 : ℝ)⁻¹ = 0 := inv_zero
```

Division by zero is defined, and equals zero, in every field in Mathlib. This is a convention, not a contradiction. A function must return something on every input, and `0` is the choice that makes the most lemmas true without hypotheses.

Nothing false follows, because the lemmas that matter still require a nonzero denominator:

```
#check @div_self                   -- a ≠ 0 → a / a = 1
#check @div_add_div                -- b ≠ 0 → d ≠ 0 → a / b + c / d = (a * d
+ b * c) / (b * d)

example (x : ℝ) (hx : x ≠ 0) : x * x⁻¹ = 1 := mul_inv_cancel₀ hx
```

The same idea gives other functions a value outside their natural domain. These are called junk values:

Expression	Value in Mathlib
`x / 0`, `0⁻¹`	`0`
`√x` for `x < 0`	`0`
`Real.log 0`	`0`
`Real.log x` for `x < 0`	`Real.log (-x)`
`deriv f x` when `f` is not differentiable at `x`	`0`
`∫ x, f x` when `f` is not integrable	`0`

```
example : √(-1 : ℝ) = 0 := Real.sqrt_eq_zero_of_nonpos (by norm_num)
example (x : ℝ) (hx : 0 ≤ x) : √x ^ 2 = x := Real.sq_sqrt hx
example (x : ℝ) : √(x ^ 2) = |x| := Real.sqrt_sq_eq_abs x
```

A theorem you state is only as meaningful as its hypotheses. If you prove something about `f x / g x` and forget `g x ≠ 0`, Lean will not complain, and your theorem will say something about `0` at those points. The FAQ Division by zero in type theory discusses the design.

Casts

A fact about natural numbers is not a fact about real numbers, even when it looks the same:

```
example (m n : ℕ) (h : m ≤ n) : (m : ℝ) ≤ n := by
  exact h
-- error: Type mismatch
--   h
-- has type
--   m ≤ n
-- but is expected to have type
--   ↑m ≤ ↑n
```

The bridge is a family of lemmas named `Nat.cast_…`, `Int.cast_…` and `Rat.cast_…`:

```
#check @Nat.cast_add      -- ↑(m + n) = ↑m + ↑n
#check @Nat.cast_mul      -- ↑(m * n) = ↑m * ↑n
#check @Nat.cast_pow      -- ↑(m ^ n) = ↑m ^ n
#check @Nat.cast_le       -- ↑m ≤ ↑n ↔ m ≤ n
#check @Nat.cast_lt       -- ↑m < ↑n ↔ m < n
#check @Nat.cast_inj      -- ↑m = ↑n ↔ m = n
#check @Nat.cast_sub      -- m ≤ n → ↑(n - m) = ↑n - ↑m
```

Note the hypothesis on `Nat.cast_sub`. The cast of a truncated subtraction is the real subtraction only when nothing was truncated.

You rarely apply these by hand. Four tactics do it.

Tactic	What it does	Use it when
`exact_mod_cast h`	Closes the goal with `h`, up to moving casts	The goal and `h` are the same statement in two types
`push_cast`	Pushes casts toward the variables: `↑(m + n)` becomes `↑m + ↑n`	You want to compute in the larger type
`norm_cast`	Pulls casts outward and removes them when possible	The goal is really a statement about the smaller type
`zify`, `qify`, `rify`	Turn a statement about `ℕ` into one about `ℤ`, `ℚ`, `ℝ`	Subtraction or division gets in the way

```
example (m n : ℕ) (h : m ≤ n) : (m : ℝ) ≤ n := by exact_mod_cast h

example (m n : ℕ) (h : (m : ℝ) = n) : m = n := by exact_mod_cast h

example (n m : ℕ) : ((n * m + 3 : ℕ) : ℝ) = n * m + 3 := by
  push_cast
  -- ⊢ ↑n * ↑m + 3 = ↑n * ↑m + 3
  ring

example (n : ℕ) (h : 3 ≤ n) : ((n - 3 : ℕ) : ℝ) = n - 3 := by
  push_cast [h]         -- the hypothesis lets it use Nat.cast_sub
  rfl
```

`linarith` and `nlinarith` do not move facts across a cast by themselves. Give them the cast version first:

```
example (n : ℕ) (h : 2 ≤ n) : (4 : ℝ) ≤ n ^ 2 := by
  have h' : (2 : ℝ) ≤ n := by exact_mod_cast h
  nlinarith
```

`zify` moves a whole statement to the integers. Given `h : b ≤ a`, it converts the truncated subtraction correctly:

```
example (a b : ℕ) (h : b ≤ a) (h' : a - b = 3) : a = b + 3 := by
  zify [h] at h' ⊢
  -- h' : ↑a - ↑b = 3     ⊢ ↑a = ↑b + 3       both in ℤ
  linarith
```

In the other direction, `lift` replaces a nonnegative integer by a natural number:

```
example (n : ℤ) (hn : 0 ≤ n) : ∃ k : ℕ, (k : ℤ) = n := by
  lift n to ℕ using hn
  -- n : ℕ     ⊢ ∃ k, ↑k = ↑n
  exact ⟨n, rfl⟩
```

Powers

The notation `x ^ y` means three different functions, depending on the type of the exponent.

Exponent type	Function	Defined for
`ℕ`	Repeated multiplication, `Monoid.npow`	Any monoid
`ℤ`	`zpow`, with `x ^ (-n) = (x ^ n)⁻¹`	Groups and fields
`ℝ`	`Real.rpow`, defined through `exp` and `log`	Real bases

```
example (x : ℝ) : x ^ 2 = x * x := sq x
example (x : ℝ) (n : ℕ) : x ^ (n + 1) = x ^ n * x := pow_succ x n
example (x : ℝ) (h : 0 < x) (n : ℕ) : 0 < x ^ n := pow_pos h n
example (x y : ℝ) (h : 0 ≤ x) (hxy : x ≤ y) : x ^ 5 ≤ y ^ 5 :=
pow_le_pow_left₀ h hxy 5
```

The trap is a fractional exponent written without a type:

```
example (x : ℝ) : x ^ (1 / 2) = 1 := by norm_num
```

This is true, and it is not what you meant. The exponent `1 / 2` is a natural number, equal to `0`, so the left side is `x ^ 0`. A square root as a power must be written with a real exponent:

```
example (x : ℝ) : x ^ (1 / 2 : ℝ) = √x := (Real.sqrt_eq_rpow x).symm
```

The lemmas for the three powers have different names: `pow_add`, `zpow_add₀`, `Real.rpow_add`. When a lemma about powers refuses to apply, check which power your expression uses.

Absolute value, `min`, `max`

`|x|` is typed with the ordinary bar. The lemmas you need most turn an absolute value into two inequalities:

```
example (x ε : ℝ) (h : |x| < ε) : -ε < x ∧ x < ε := abs_lt.mp h

example (x : ℝ) : |x| ≤ 1 ↔ -1 ≤ x ∧ x ≤ 1 := abs_le

example (x a : ℝ) (h : |x - a| < 1) : x < a + 1 := by
  have h' := abs_lt.mp h
  -- h' : -1 < x - a ∧ x - a < 1
  linarith
```

Lemma	Statement
`abs_nonneg x`	`0 ≤ |x|`
`abs_of_nonneg`, `abs_of_neg`	`0 ≤ x → |x| = x`, and `x < 0 → |x| = -x`
`le_abs_self x`, `neg_le_abs x`	`x ≤ |x|`, and `-x ≤ |x|`
`abs_lt`, `abs_le`	`|a| < b ↔ -b < a ∧ a < b`, and the same with `≤`
`abs_add_le a b`	`|a + b| ≤ |a| + |b|`

Lemma	Statement
`abs_sub_comm a b`	`|a - b| = |b - a|`
`abs_sub_le a b c`	`|a - c| ≤ |a - b| + |b - c|`
`abs_mul a b`	`|a * b| = |a| * |b|`
`abs_cases x`	`|x| = x ∧ 0 ≤ x`, or `|x| = -x ∧ x < 0`

`min` and `max` come with `min_le_left`, `min_le_right`, `le_min`, `le_max_left`, `le_max_right` and `max_le`. In ε-δ proofs, the standard choice `δ = min δ₁ δ₂` is positive by `lt_min` or by `positivity`:

```
example (δ₁ δ₂ : ℝ) (h₁ : 0 < δ₁) (h₂ : 0 < δ₂) : 0 < min δ₁ δ₂ := lt_min h₁ h₂
```

Floor and the Archimedean property

```
example (x : ℝ) : (⌊x⌋ : ℝ) ≤ x := Int.floor_le x
example (x : ℝ) : x < ⌊x⌋ + 1 := Int.lt_floor_add_one x
example : ⌊(2.5 : ℝ)⌋ = 2 := by norm_num

example (x : ℝ) : ∃ n : ℕ, x < n := exists_nat_gt x

example (ε : ℝ) (hε : 0 < ε) : ∃ n : ℕ, 1 / (n + 1 : ℝ) < ε :=
exists_nat_one_div_lt hε
```

`⌊x⌋` is an integer, typed `\lfloor` and `\rfloor`, and `⌊x⌋₊` is the natural-number floor. The last two lemmas are the Archimedean property in the two forms analysis uses.

Real numbers and decimals

A decimal literal in `ℝ` is an exact rational, and `norm_num` computes with it exactly. `Float` is a different type, with rounding, used only for programs.

```
example : (0.1 : ℝ) + 0.2 = 0.3 := by norm_num

#eval (0.1 : Float) + 0.2         -- 0.300000
#eval (0.1 : ℚ) + 0.2             -- 3 / 10
```

The elementary functions live in the `Real` namespace: `Real.exp`, `Real.log`, `Real.sin`, `Real.cos`, `Real.sqrt` with notation `√`, and `Real.pi` with notation `π` after `open Real`.

```
example (x : ℝ) : 0 < Real.exp x := Real.exp_pos x
example (x y : ℝ) : Real.exp (x + y) = Real.exp x * Real.exp y :=
Real.exp_add x y
example (x y : ℝ) (h : x ≤ y) : Real.exp x ≤ Real.exp y := by gcongr
example (x : ℝ) : x + 1 ≤ Real.exp x := Real.add_one_le_exp x
```

Complex numbers

A complex number is a structure with fields `re` and `im`. Two are equal when both parts are equal, and `Complex.I` is the imaginary unit.

```
example : Complex.I ^ 2 = -1 := Complex.I_sq

example (z w : ℂ) (h₁ : z.re = w.re) (h₂ : z.im = w.im) : z = w :=
Complex.ext h₁ h₂

example (x : ℝ) : (x : ℂ).re = x := Complex.ofReal_re x

example (z : ℂ) (h : z.im = 0) : z = z.re := by
  apply Complex.ext
  · simp
  · simp [h]
```

`ring` works in `ℂ`, and `simp` knows the real and imaginary parts of sums, products and casts. The modulus is the norm `‖z‖`.

Extended nonnegative reals

Measures take values in `ℝ≥0∞`, the interval `[0, ∞]`. Its arithmetic is total and has its own conventions.

```
open scoped ENNReal

example : (2 : ℝ≥0∞) + ∞ = ∞ := by simp
example : (0 : ℝ≥0∞) * ∞ = 0 := by simp
example : (∞ : ℝ≥0∞) - ∞ = 0 := by simp
example (a : ℝ≥0∞) : a ≤ ∞ := le_top
```

Two functions connect it to `ℝ`. `ENNReal.ofReal` sends negative numbers to `0`, and `ENNReal.toReal` sends `∞` to `0`.

```
example (x : ℝ) (h : 0 ≤ x) : (ENNReal.ofReal x).toReal = x :=
ENNReal.toReal_ofReal h

example (a : ℝ≥0∞) (h : a ≠ ∞) : ENNReal.ofReal a.toReal = a :=
ENNReal.ofReal_toReal h
```

Subtraction is truncated, as in `ℕ`. `ring` still proves identities that use only `+` and `*`, since the type is a commutative semiring, but `linarith` does not apply. Proofs about `ℝ≥0∞` use `simp`, `gcongr` and the lemmas of the `ENNReal` namespace. Chapter 19 returns to this type.

Errors you will meet here

Message or symptom	Cause	Fix
`Type mismatch: h has type m ≤ n but is expected to have type ↑m ≤ ↑n`	A fact in `ℕ` used for a goal in `ℝ`	`exact_mod_cast h`
`linarith failed`, with `↑n` in the goal and a hypothesis about `n`	`linarith` does not cross casts	Restate the hypothesis in the target type with `exact_mod_cast`

Message or symptom	Cause	Fix
A goal that is obviously true will not close, and contains `↑(n - 1)` or `↑(n / 2)`	The subtraction or division was done in `ℕ`	Check the statement. Use `push_cast [h]` with `h : 1 ≤ n`, or restate
`omega could not prove the goal` on a statement with `/`	Division by a variable, or a real division	`omega` only handles division by numerals in `ℕ` and `ℤ`
`unsolved goals` after `ring` on an identity with `x⁻¹` or `/`	`ring` cannot cancel `x / x`	`field_simp` first, with `x ≠ 0` in context
A power lemma does not apply	Wrong kind of power: `ℕ`, `ℤ` or `ℝ` exponent	Hover over `^` to see which, and use the matching lemma

Exercises

```
import Mathlib

-- Exercise 1. Before running anything, say which of these are true. Then
prove
-- the true ones and correct the false ones.
example : (7 / 2 : ℕ) = 3 := by
  sorry
example : (7 / 2 : ℝ) = 3.5 := by
  sorry
example : ((1 / 3 : ℕ) : ℝ) * 3 = 1 := by
  sorry

-- Exercise 2. A fact crosses from ℝ to ℕ.
example (a b : ℕ) (h : (a : ℝ) + 1 ≤ b) : a < b := by
  sorry

-- Exercise 3. The cast of a truncated subtraction.
example (n : ℕ) (h : 5 ≤ n) : ((n - 5 : ℕ) : ℝ) = n - 5 := by
  sorry

-- Exercise 4. Unfold the absolute value into two inequalities.
example (x a : ℝ) (h : |x - a| < 2) : a - 2 < x := by
  sorry

-- Exercise 5. First obtain 1 ≤ n in ℝ. Then use div_le_one.
example (n : ℕ) (hn : 0 < n) : (1 : ℝ) / n ≤ 1 := by
  sorry

-- Exercise 6. Both bounds on the floor.
example (x : ℝ) : (⌊x⌋ : ℝ) ≤ x ∧ x < ⌊x⌋ + 1 := by
  sorry

-- Exercise 7. Move to ℤ, where subtraction is honest.
example (a b : ℕ) (h : b ≤ a) (h' : a - b = 7) : a = b + 7 := by
  sorry
```

```
-- Exercise 8. A nonlinear fact across a cast.
example (n : ℕ) (h : 3 ≤ n) : (9 : ℝ) ≤ n ^ 2 := by
  sorry
```

Solutions

```
import Mathlib

-- Exercise 1. The first two are true. The third is false: 1 / 3 is 0 in ℕ.
example : (7 / 2 : ℕ) = 3 := by
  norm_num
example : (7 / 2 : ℝ) = 3.5 := by
  norm_num
example : ((1 / 3 : ℕ) : ℝ) * 3 = 0 := by
  norm_num
example : (1 / 3 : ℝ) * 3 = 1 := by
  norm_num

-- Exercise 2.
example (a b : ℕ) (h : (a : ℝ) + 1 ≤ b) : a < b := by
  have h' : a + 1 ≤ b := by exact_mod_cast h
  omega

-- Exercise 3.
example (n : ℕ) (h : 5 ≤ n) : ((n - 5 : ℕ) : ℝ) = n - 5 := by
  push_cast [h]
  rfl

-- Exercise 4.
example (x a : ℝ) (h : |x - a| < 2) : a - 2 < x := by
  have h' := abs_lt.mp h
  -- h' : -2 < x - a ∧ x - a < 2
  linarith

-- Exercise 5.
example (n : ℕ) (hn : 0 < n) : (1 : ℝ) / n ≤ 1 := by
  have h : (1 : ℝ) ≤ n := by exact_mod_cast hn
  rw [div_le_one (by positivity)]
  -- ⊢ 1 ≤ ↑n
  exact h

-- Exercise 6.
example (x : ℝ) : (⌊x⌋ : ℝ) ≤ x ∧ x < ⌊x⌋ + 1 :=
  ⟨Int.floor_le x, Int.lt_floor_add_one x⟩

-- Exercise 7.
example (a b : ℕ) (h : b ≤ a) (h' : a - b = 7) : a = b + 7 := by
  zify [h] at h' ⊢
  linarith

-- Exercise 8.
example (n : ℕ) (h : 3 ≤ n) : (9 : ℝ) ≤ n ^ 2 := by
  have h' : (3 : ℝ) ≤ n := by exact_mod_cast h
  nlinarith
```

In Exercise 2, `exact_mod_cast` turns `↑a + 1 ≤ ↑b` in `ℝ` into `a + 1 ≤ b` in `ℕ`. Exercise 7 is also closed by `omega` alone.

Chapter 15: Sets, Functions, Finsets and Sums

A set in Lean is a predicate, a finite sum runs over a `Finset`, and both are handled by unfolding membership and then using the logic of Chapter 7.

Sets are predicates

A set of elements of `α` has type `Set α`, and `Set α` is by definition `α → Prop`. The statement `x ∈ s` means that the predicate `s` holds at `x`.

```
import Mathlib

open Set

#check ({x : ℝ | 0 < x})          -- {x | 0 < x} : Set ℝ
#check (univ : Set ℝ)             -- univ : Set ℝ      the whole type
#check (∅ : Set ℝ)                -- ∅ : Set ℝ
#check ({1, 2, 3} : Set ℕ)        -- {1, 2, 3} : Set ℕ
#check Icc (0 : ℝ) 1              -- Icc 0 1 : Set ℝ   the closed interval
[0, 1]
```

Every set is a subset of a fixed type. There is no set of "everything", and `x ∈ s` only makes sense when `x` has the type the set is about.

Intervals are named by their endpoints: `c` for closed, `o` for open, `i` for infinite.

Name	Set	Name	Set
`Icc a b`	`[a, b]`	`Ici a`	`[a, ∞)`
`Ioo a b`	`(a, b)`	`Ioi a`	`(a, ∞)`
`Ico a b`	`[a, b)`	`Iic b`	`(-∞, b]`
`Ioc a b`	`(a, b]`	`Iio b`	`(-∞, b)`

Membership unfolds to the predicate

`x ∈ {y | p y}` is definitionally `p x`. You can use a membership hypothesis directly as the property, or unfold it explicitly.

```
example : {x : ℝ | 1 < x} ⊆ {x : ℝ | 0 < x} := by
  intro x hx
  -- hx : x ∈ {x | 1 < x}     ⊢ x ∈ {x | 0 < x}
  rw [mem_ofPred_eq] at hx ⊢
  -- hx : 1 < x               ⊢ 0 < x
  linarith
```

`Set.mem_ofPred_eq` is the lemma `(x ∈ {y | p y}) = p x`. Until recently it was called `Set.mem_setOf_eq`, which still works with a deprecation warning.

Since the two forms are definitionally equal, you can also restate the hypothesis with its real meaning:

```
example (x : ℝ) (hx : x ∈ {y : ℝ | 1 < y}) : 0 < x := by
  have h : 1 < x := hx
  linarith
```

Membership in an interval is a conjunction, so `obtain` splits it:

```
example (x : ℝ) (hx : x ∈ Icc (0 : ℝ) 1) : x ^ 2 ≤ 1 := by
  obtain ⟨h0, h1⟩ := hx
  -- h0 : 0 ≤ x     h1 : x ≤ 1
  nlinarith

example : (1 / 2 : ℝ) ∈ Icc (0 : ℝ) 1 := by
  constructor <;> norm_num
```

The operations are the connectives

Set statement	Unfolds to	Lemma
`s ⊆ t`	`∀ x, x ∈ s → x ∈ t`	`Set.subset_def`
`x ∈ s ∩ t`	`x ∈ s ∧ x ∈ t`	`Set.mem_inter_iff`
`x ∈ s ∪ t`	`x ∈ s ∨ x ∈ t`	`Set.mem_union`
`x ∈ sᶜ`	`x ∉ s`	`Set.mem_compl_iff`
`x ∈ s \ t`	`x ∈ s ∧ x ∉ t`	`Set.mem_sdiff`
`x ∈ ∅`	`False`	`Set.notMem_empty`
`x ∈ univ`	`True`	`Set.mem_univ`
`s = t`	`∀ x, x ∈ s ↔ x ∈ t`	The `ext` tactic

So a subset proof starts with `intro x hx`, an element of an intersection is a pair, and an element of a union is a case split.

```
variable {α β : Type*} (s t u : Set α)

example : s ∩ t ⊆ s := by
  intro x hx
  -- hx : x ∈ s ∩ t    ⊢ x ∈ s
  exact hx.1

example (h : s ⊆ t) : s ∩ u ⊆ t ∩ u := by
  rintro x ⟨hs, hu⟩
  exact ⟨h hs, hu⟩
```

A subset hypothesis `h : s ⊆ t` is a function. Applied to `hs : x ∈ s`, it gives `h hs : x ∈ t`.

Equality of sets: `ext`

Two sets are equal when they have the same elements. The tactic `ext x` reduces `s = t` to `x ∈ s ↔ x ∈ t`.

```
example : s ∩ t = t ∩ s := by
  ext x
  -- ⊢ x ∈ s ∩ t ↔ x ∈ t ∩ s
  constructor
  · rintro ⟨hs, ht⟩
    exact ⟨ht, hs⟩
  · rintro ⟨ht, hs⟩
    exact ⟨hs, ht⟩
```

For identities of this kind, the fast route is to unfold the memberships and call `tauto`:

```
example : s ∩ (t ∪ u) = (s ∩ t) ∪ (s ∩ u) := by
  ext x
  simp only [mem_inter_iff, mem_union]
  -- ⊢ x ∈ s ∧ (x ∈ t ∨ x ∈ u) ↔ x ∈ s ∧ x ∈ t ∨ x ∈ s ∧ x ∈ u
  tauto
```

The library already has these facts, here as `Set.inter_union_distrib_left`. Write the manual proof once to learn the pattern, then use `exact?`.

Images and preimages

Notation	Name	Membership
`f '' s`	Image of `s`	`y ∈ f '' s ↔ ∃ x ∈ s, f x = y`
`f ⁻¹' v`	Preimage of `v`	`x ∈ f ⁻¹' v ↔ f x ∈ v`
`range f`	Image of the whole type	`y ∈ range f ↔ ∃ x, f x = y`

The preimage unfolds by `rfl`, so preimage proofs are easy. An element of an image is a triple: a point, a proof it is in `s`, and an equation.

```
variable (f : α → β) (v : Set β)

example : s ⊆ f ⁻¹' (f '' s) := by
  intro x hx
  -- ⊢ x ∈ f ⁻¹' f '' s,  which means  f x ∈ f '' s
  exact ⟨x, hx, rfl⟩

example : f '' (f ⁻¹' v) ⊆ v := by
  rintro y ⟨x, hx, rfl⟩
  -- hx : x ∈ f ⁻¹' v     ⊢ f x ∈ v
  exact hx
```

The pattern `⟨x, hx, rfl⟩` inside `rintro` deserves attention. The third component is the equation `f x = y`. Writing `rfl` there substitutes it at once, so `y` disappears and the goal speaks of `f x`.

```
example : f '' (s ∩ t) ⊆ f '' s ∩ f '' t := by
  rintro y ⟨x, ⟨hs, ht⟩, rfl⟩
  -- hs : x ∈ s,  ht : x ∈ t     ⊢ f x ∈ f '' s ∩ f '' t
  exact ⟨⟨x, hs, rfl⟩, ⟨x, ht, rfl⟩⟩
```

The fundamental link between the two is `Set.image_subset_iff : f '' s ⊆ v ↔ s ⊆ f ⁻¹' v`.

Unions and intersections of families

For a family `A : ℕ → Set ℝ`, the union is `⋃ i, A i` and the intersection is `⋂ i, A i`, typed `\bigcup` and `\bigcap`. Membership is an existential or a universal statement.

```
example (A : ℕ → Set ℝ) (x : ℝ) : x ∈ ⋃ i, A i ↔ ∃ i, x ∈ A i := mem_iUnion
example (A : ℕ → Set ℝ) (x : ℝ) : x ∈ ⋂ i, A i ↔ ∀ i, x ∈ A i := mem_iInter
```

Here is a complete example, using the Archimedean property from Chapter 14:

```
example : ⋃ n : ℕ, Icc (0 : ℝ) n = Ici 0 := by
  ext x
  simp only [mem_iUnion, mem_Icc, mem_Ici]
  -- ⊢ (∃ i, 0 ≤ x ∧ x ≤ ↑i) ↔ 0 ≤ x
  constructor
  · rintro ⟨n, h0, -⟩
    exact h0
  · intro h0
    obtain ⟨n, hn⟩ := exists_nat_gt x      -- hn : x < ↑n
    exact ⟨n, h0, hn.le⟩
```

The dash in `⟨n, h0, -⟩` discards a component that is not needed. The names contain a small `i` for "indexed": `iUnion`, `iInter`, `iSup`, `iInf`.

Functions

Injectivity and surjectivity are ordinary definitions, and `intro` unfolds them.

```
open Function

#print Function.Injective     -- ∀ ⦃a₁ a₂ : α⦄, f a₁ = f a₂ → a₁ = a₂
#print Function.Surjective    -- ∀ (b : β), ∃ a, f a = b

example : Injective (fun x : ℝ ↦ 2 * x + 1) := by
  intro a b h
  -- h : (fun x ↦ 2 * x + 1) a = (fun x ↦ 2 * x + 1) b     ⊢ a = b
  dsimp only at h
  -- h : 2 * a + 1 = 2 * b + 1
  linarith

example : Surjective (fun x : ℝ ↦ 2 * x + 1) := by
  intro y
  use (y - 1) / 2
  -- ⊢ (fun x ↦ 2 * x + 1) ((y - 1) / 2) = y
  ring
```

After `intro`, a hypothesis often contains a function applied to an argument, `(fun x ↦ …) a`. The tactic `dsimp only` performs the substitution. Many tactics, `ring` among them, see through it without help.

Composition is `g ∘ f`, typed `\circ`, and `(g ∘ f) x` is `g (f x)` by `rfl`.

```
example {γ : Type*} (g : β → γ) (hf : Injective f) (hg : Injective g) :
    Injective (g ∘ f) := by
  intro a b h
  -- h : (g ∘ f) a = (g ∘ f) b     ⊢ a = b
  exact hf (hg h)
```

Monotonicity follows the same pattern, with `Monotone`, `StrictMono`, `Antitone` and `StrictAnti`:

```
example : StrictMono (fun x : ℝ ↦ 2 * x + 1) := by
  intro a b h
  dsimp only
  linarith
```

Finite sets and sums

A `Finset α` is a finite set given as data, so it can be summed over and counted. The one you use most is `Finset.range n`, the set `{0, 1, …, n - 1}`.

```
open Finset

#eval range 5                                  -- {0, 1, 2, 3, 4}
#eval Finset.Icc 1 4                           -- {1, 2, 3, 4}
#eval (range 10).filter (fun n ↦ n % 3 = 0) -- {0, 3, 6, 9}
#eval ∑ i ∈ range 5, i ^ 2                     -- 30
#eval ∏ i ∈ range 5, (i + 1)                   -- 120
```

The notation is `∑ i ∈ s, f i`, typed `\sum` and `\in`. Older texts write `∑ i in s, f i`, which Lean no longer accepts. Over a finite type, `∑ i, f i` sums over all of it.

`i ∈ range n` means `i < n`, by `Finset.mem_range`. The sum over `range (n + 1)` therefore ends at `n`.

Proving a sum formula

The key lemma peels off the last term:

```
#check @sum_range_succ
-- ∑ x ∈ Finset.range (n + 1), f x = ∑ x ∈ Finset.range n, f x + f n
```

An induction then has a fixed shape: `simp` for the base case, and `rw [sum_range_succ, ih]` followed by algebra for the step.

```
theorem sum_id (n : ℕ) : ∑ i ∈ range (n + 1), (i : ℝ) = n * (n + 1) / 2 :=
by
  induction n with
  | zero => simp
  | succ k ih =>
    rw [sum_range_succ, ih]
    -- ⊢ ↑k * (↑k + 1) / 2 + ↑(k + 1) = ↑(k + 1) * (↑(k + 1) + 1) / 2
    push_cast
    ring
```

The goal contains `↑(k + 1)`, the cast of a natural-number sum. `push_cast` turns it into `↑k + 1`, and `ring` finishes.

The sum is taken in `ℝ`, by casting each term, so that the division by 2 is real. Stated in `ℕ`, the right side would use floor division and the induction step would fail. A statement in `ℕ` should avoid division:

```
example (n : ℕ) : 2 * ∑ i ∈ range (n + 1), i = n * (n + 1) := by
  induction n with
  | zero => rfl
  | succ k ih =>
    rw [sum_range_succ, mul_add, ih]
    ring
```

A geometric sum needs a nonzero denominator, and `field_simp` uses it:

```
example (x : ℝ) (hx : x ≠ 1) (n : ℕ) :
    ∑ i ∈ range n, x ^ i = (x ^ n - 1) / (x - 1) := by
  have h : x - 1 ≠ 0 := sub_ne_zero.mpr hx
  induction n with
  | zero => simp
  | succ k ih =>
    rw [sum_range_succ, ih]
    -- ⊢ (x ^ k - 1) / (x - 1) + x ^ k = (x ^ (k + 1) - 1) / (x - 1)
    field_simp
    -- ⊢ x ^ k - 1 + x ^ k * (x - 1) = x ^ (k + 1) - 1
    ring
```

Mathlib has this as `geom_sum_eq`.

The algebra of sums

Lemma	Statement
`sum_range_zero`	`∑ i ∈ range 0, f i = 0`
`sum_range_succ`	Peel off the last term
`sum_range_succ'`	Peel off the first term
`sum_add_distrib`	`∑ (f i + g i) = ∑ f i + ∑ g i`
`sum_sub_distrib`	The same for subtraction
`mul_sum`, `sum_mul`	`a * ∑ f i = ∑ a * f i`, and on the right
`sum_const`	`∑ _i ∈ s, b = #s • b`
`sum_comm`	Exchange two finite sums
`sum_congr`	Equal sets and equal terms give equal sums
`sum_range_sub`	Telescoping: `∑ i ∈ range n, (f (i + 1) - f i) = f n - f 0`
`sum_le_sum`	Termwise `≤` gives `≤` of the sums
`sum_nonneg`	Nonnegative terms give a nonnegative sum
`abs_sum_le_sum_abs`	The triangle inequality for sums
`sum_mul_sq_le_sq_mul_sq`	The Cauchy–Schwarz inequality

All are in the `Finset` namespace. The notation `#s` is the number of elements of `s`, also written `s.card`. These lemmas are spread over `Mathlib/Algebra/BigOperators` and `Mathlib/Algebra/Order/BigOperators`.

`simp` knows the simple ones:

```
example (n : ℕ) : ∑ _i ∈ range n, (1 : ℝ) = n := by simp
```

Estimating a sum

`gcongr` works under the sum sign, as in Chapter 13. A typical estimate is a `calc` with one idea per line:

```
example (f : ℕ → ℝ) (n : ℕ) (M : ℝ) (h : ∀ i, |f i| ≤ M) :
    |∑ i ∈ range n, f i| ≤ n * M := by
  calc |∑ i ∈ range n, f i|
      ≤ ∑ i ∈ range n, |f i| := abs_sum_le_sum_abs f (range n)
    _ ≤ ∑ _i ∈ range n, M := by
        gcongr with i hi
        exact h i
    _ = n * M := by simp
```

Line one is the triangle inequality. Line two bounds each term, and `gcongr with i hi` leaves exactly the termwise goal `|f i| ≤ M`. Line three evaluates a constant sum.

Errors you will meet here

Message	Cause	Fix
`Ambiguous term mem_Icc`, with interpretations `Set.mem_Icc` and `Finset.mem_Icc`	Both `Set` and `Finset` are open, and both define the name	Write the full name, or open only one namespace
`Set.mem_setOf_eq has been deprecated`	The lemma was renamed	Use `Set.mem_ofPred_eq`
`omega could not prove the goal` in a sum formula over `ℕ`	The statement divides in `ℕ`	State it in `ℝ`, or multiply the division away
`unexpected token 'in'` in a sum	Old notation `∑ i in s`	Write `∑ i ∈ s`
The goal has `(fun x ↦ …) a` and a tactic does not recognize it	The function is not applied yet	`dsimp only`, or `simp only []`, or `show` the reduced goal

Exercises

```
import Mathlib

open Set Function
```

```
variable {α β γ : Type*} (s t u : Set α) (f : α → β) (v w : Set β)

-- Exercise 1.
example : s ⊆ s ∪ t := by
  sorry

-- Exercise 2. Unfold the memberships, then tauto.
example : (s ∪ t)ᶜ = sᶜ ∩ tᶜ := by
  sorry

-- Exercise 3. A subset of the real line.
example : Icc (0 : ℝ) 1 ⊆ Icc (-1) 2 := by
  sorry

-- Exercise 4. Preimages unfold by rfl.
example : f ⁻¹' (v ∩ w) = f ⁻¹' v ∩ f ⁻¹' w := by
  sorry

-- Exercise 5.
example : Injective (fun x : ℝ ↦ 3 * x - 2) := by
  sorry

-- Exercise 6. Choose preimages one at a time with obtain ⟨y, rfl⟩.
example (g : β → γ) (hf : Surjective f) (hg : Surjective g) : Surjective (g
∘ f) := by
  sorry

-- Exercise 7. The sum of the first n odd numbers.
example (n : ℕ) : ∑ i ∈ Finset.range n, (2 * (i : ℝ) + 1) = n ^ 2 := by
  sorry

-- Exercise 8. The sum of the first cubes.
example (n : ℕ) : ∑ i ∈ Finset.range (n + 1), (i : ℝ) ^ 3 = (n * (n + 1) /
2) ^ 2 := by
  sorry

-- Exercise 9. Bound each term by 1.
example (f : ℕ → ℝ) (n : ℕ) (h : ∀ i, f i ≤ 1) : ∑ i ∈ Finset.range n, f i ≤
n := by
  sorry

-- Exercise 10. Telescoping, by induction.
example (f : ℕ → ℝ) (n : ℕ) : ∑ i ∈ Finset.range n, (f (i + 1) - f i) = f n
- f 0 := by
  sorry
```

Solutions

```
import Mathlib

open Set Function

variable {α β γ : Type*} (s t u : Set α) (f : α → β) (v w : Set β)

-- Exercise 1.
example : s ⊆ s ∪ t := by
  intro x hx
  left
```

```
  exact hx

-- Exercise 2.
example : (s ∪ t)ᶜ = sᶜ ∩ tᶜ := by
  ext x
  simp only [mem_compl_iff, mem_union, mem_inter_iff]
  -- ⊢ ¬(x ∈ s ∨ x ∈ t) ↔ x ∉ s ∧ x ∉ t
  tauto

-- Exercise 3.
example : Icc (0 : ℝ) 1 ⊆ Icc (-1) 2 := by
  intro x ⟨h0, h1⟩
  constructor <;> linarith

-- Exercise 4.
example : f ⁻¹' (v ∩ w) = f ⁻¹' v ∩ f ⁻¹' w := by
  ext x
  rfl

-- Exercise 5.
example : Injective (fun x : ℝ ↦ 3 * x - 2) := by
  intro a b h
  dsimp only at h
  -- h : 3 * a - 2 = 3 * b - 2
  linarith

-- Exercise 6.
example (g : β → γ) (hf : Surjective f) (hg : Surjective g) : Surjective (g
∘ f) := by
  intro z
  obtain ⟨y, rfl⟩ := hg z
  obtain ⟨x, rfl⟩ := hf y
  exact ⟨x, rfl⟩

-- Exercise 7.
example (n : ℕ) : ∑ i ∈ Finset.range n, (2 * (i : ℝ) + 1) = n ^ 2 := by
  induction n with
  | zero => simp
  | succ k ih =>
    rw [Finset.sum_range_succ, ih]
    push_cast
    ring

-- Exercise 8.
example (n : ℕ) : ∑ i ∈ Finset.range (n + 1), (i : ℝ) ^ 3 = (n * (n + 1) /
2) ^ 2 := by
  induction n with
  | zero => simp
  | succ k ih =>
    rw [Finset.sum_range_succ, ih]
    push_cast
    ring

-- Exercise 9.
example (f : ℕ → ℝ) (n : ℕ) (h : ∀ i, f i ≤ 1) : ∑ i ∈ Finset.range n, f i ≤
n := by
  calc ∑ i ∈ Finset.range n, f i
      ≤ ∑ _i ∈ Finset.range n, (1 : ℝ) := by
        gcongr with i hi
```

```
        exact h i
    _ = n := by simp

-- Exercise 10.
example (f : ℕ → ℝ) (n : ℕ) : ∑ i ∈ Finset.range n, (f (i + 1) - f i) = f n
- f 0 := by
  induction n with
  | zero => simp
  | succ k ih =>
    rw [Finset.sum_range_succ, ih]
    ring
```

The solutions write `Finset.range` in full because the file opens `Set` and not `Finset`. Opening both would make names such as `mem_Icc` ambiguous.

Chapter 16: Algebraic Structures

In Mathlib a group is a type with a `Group` instance, a homomorphism is a function bundled with its proofs, and a subgroup is a set bundled with its closure properties.

Analysis in Mathlib is built on this algebra. A normed space is a module with a norm, and the integral is a linear map. This chapter gives the vocabulary.

Working in an arbitrary group

Declare a type and assume the structure on it. Every lemma about groups is then available.

```
import Mathlib

section
variable {G : Type*} [Group G] (a b c : G)

#check mul_assoc a b c        -- a * b * c = a * (b * c)
#check one_mul a              -- 1 * a = a
#check mul_one a              -- a * 1 = a
#check inv_mul_cancel a       -- a⁻¹ * a = 1
#check mul_inv_cancel a       -- a * a⁻¹ = 1
#check inv_inv a              -- a⁻¹⁻¹ = a
#check mul_inv_rev a b        -- (a * b)⁻¹ = b⁻¹ * a⁻¹
```

A proof from the axioms is a `calc`, each line one axiom:

```
example (h : a * b = 1) : b = a⁻¹ := by
  calc b = 1 * b := (one_mul b).symm
    _ = a⁻¹ * a * b := by rw [inv_mul_cancel]
    _ = a⁻¹ * (a * b) := mul_assoc _ _ _
    _ = a⁻¹ * 1 := by rw [h]
    _ = a⁻¹ := mul_one _
```

The library has it as `eq_inv_of_mul_eq_one_right`. For identities that hold in every group, the tactic `group` decides them, and `simp` knows the standard simplifications:

```
example : a * b * b⁻¹ * a⁻¹ = 1 := by group
example : (a * b)⁻¹ = b⁻¹ * a⁻¹ := by simp
example (h : a * b = a * c) : b = c := mul_left_cancel h
```

A slightly longer argument: a group in which every element squares to the identity is commutative.

```
example (h : ∀ x : G, x * x = 1) : a * b = b * a := by
  have h1 : a * b * (a * b) = 1 := h (a * b)
  have ha : a⁻¹ = a := inv_eq_of_mul_eq_one_right (h a)
  have hb : b⁻¹ = b := inv_eq_of_mul_eq_one_right (h b)
  calc a * b = (a * b)⁻¹ := (inv_eq_of_mul_eq_one_right h1).symm
    _ = b⁻¹ * a⁻¹ := mul_inv_rev a b
    _ = b * a := by rw [ha, hb]
end
```

Multiplicative and additive

Mathlib has two parallel hierarchies. `Group` uses `*`, `1` and `⁻¹`. `AddGroup` uses `+`, `0` and `-`. Each multiplicative lemma has an additive twin, generated automatically, with a name translated word for word.

Multiplicative	Additive
`Group`, `CommGroup`, `Monoid`	`AddGroup`, `AddCommGroup`, `AddMonoid`
`mul_assoc`, `mul_comm`	`add_assoc`, `add_comm`
`one_mul`, `mul_one`	`zero_add`, `add_zero`
`inv_mul_cancel`, `mul_inv_cancel`	`neg_add_cancel`, `add_neg_cancel`
`mul_left_cancel`	`add_left_cancel`
`pow_succ : a ^ (n + 1) = a ^ n * a`	`succ_nsmul : (n + 1) • a = n • a + a`
`G →* H`, `Subgroup G`	`G →+ H`, `AddSubgroup G`
Tactic `group`	Tactic `abel`, for commutative additive groups

```
example {A : Type*} [AddCommGroup A] (x y z : A) : x - (y - z) = x - y +
z := by abel
```

Rings, fields, modules

Class	You get	Automation
`Ring R`	`+`, `*`, `-`, `0`, `1`, distributivity	`noncomm_ring`
`CommRing R`	Commutative multiplication as well	`ring`, `linear_combination`
`Field K`	Inverses of nonzero elements, `/`	`field_simp` then `ring`
`Module R M`	Scalar multiplication `r • x`. A vector space is a module over a field	`module`, `simp`
`Algebra R A`	A ring `A` that is also an `R`-module, compatibly	`ring`, `simp`

```
section
variable {R : Type*} [CommRing R] (a b : R)

example : (a + b) * (a - b) = a ^ 2 - b ^ 2 := by ring

example (h : a + b = 0) : b = -a := by
  linear_combination h
end

example {K : Type*} [Field K] (x y : K) (hx : x ≠ 0) (hy : y ≠ 0) :
    1 / x + 1 / y = (x + y) / (x * y) := by
```

```
  field_simp
  ring
```

There is no separate class for vector spaces. To say that `E` is a real vector space, write `[AddCommGroup E] [Module ℝ E]`. Scalar multiplication is `•`, typed `\smul`.

Concrete structures come with their instances, and small finite ones can be checked by computation:

```
#synth CommRing ℤ              -- Int.instCommRing
#synth Field ℚ                 -- Rat.instField
#synth CommRing (ZMod 6)       -- ZMod.commRing 6

#eval (3 : ZMod 7) * 5         -- 1

example : ∀ x : ZMod 4, x ^ 2 = 0 ∨ x ^ 2 = 1 := by decide
```

Homomorphisms are bundled

A group homomorphism from `G` to `H` is not a function with a side property. It is a term of the type `G →* H`, a structure that contains the function and the proofs that it respects `1` and `*`.

Notation	Meaning	Typed
`G →* H`	Monoid or group homomorphism	`\to*`
`G →+ H`	Additive homomorphism	`\to+`
`R →+* S`	Ring homomorphism	`\to+*`
`E →ₗ[R] F`	R-linear map	`\to\_l`
`E →L[R] F`	Continuous linear map	`\toL`
`G ≃* H`, `R ≃+* S`, `E ≃ₗ[R] F`	Isomorphisms	`\equiv*`
`α ≃ β`	Bijection with a chosen inverse	`\equiv`

A bundled map is applied like a function. Its properties are stated by lemmas that work for every kind of morphism:

```
section
variable {G H K : Type*} [Group G] [Group H] [Group K]

example (f : G →* H) (a b : G) : f (a * b) = f a * f b := map_mul f a b
example (f : G →* H) : f 1 = 1 := map_one f
example (f : G →* H) (a : G) : f a⁻¹ = (f a)⁻¹ := map_inv f a
example (f : G →* H) (a : G) (n : ℕ) : f (a ^ n) = f a ^ n := map_pow f a n

example (f : G →* H) (g : H →* K) : G →* K := g.comp f
end
```

The same names `map_add`, `map_mul`, `map_one`, `map_zero`, `map_smul` and `map_sum` serve ring homomorphisms and linear maps. `simp` applies them all:

```
example {R S : Type*} [Ring R] [Ring S] (f : R →+* S) (a b : R) :
    f (a * b + 1) = f a * f b + 1 := by
  rw [map_add, map_mul, map_one]

example {E F : Type*} [AddCommGroup E] [AddCommGroup F] [Module ℝ E] [Module
ℝ F]
    (f : E →ₗ[ℝ] F) (c : ℝ) (x y : E) : f (c • x + y) = c • f x + f y := by
  simp
```

To define a homomorphism, give the function and prove the fields. The field names end with a prime.

```
def doubleHom : ℤ →+ ℤ where
  toFun n := 2 * n
  map_zero' := by simp
  map_add' := by
    intro a b
    ring

#eval doubleHom 21        -- 42

example (a b : ℤ) : doubleHom (a + b) = doubleHom a + doubleHom b :=
  map_add doubleHom a b
```

Bundling is why you can write `g.comp f` and get a homomorphism without proving anything: the composition carries its proofs with it. The predicate style of Chapter 10, `IsLinear f`, is used in Mathlib mostly for properties of functions written as formulas, such as `Continuous f`; even these have a bundled form when a type of maps is useful, such as `C(X, Y)` for continuous maps.

Subobjects

A subgroup is a structure: a set, called the carrier, with proofs that it contains `1` and is closed under `*` and `⁻¹`. Membership `a ∈ K` works as for sets.

```
section
variable {G H : Type*} [Group G] [Group H]

example (K : Subgroup G) (a b : G) (ha : a ∈ K) (hb : b ∈ K) : a * b ∈ K :=
K.mul_mem ha hb
example (K : Subgroup G) : (1 : G) ∈ K := K.one_mem
example (K : Subgroup G) (a : G) (ha : a ∈ K) : a⁻¹ ∈ K := K.inv_mem ha
```

Subgroups of `G` form a lattice. The intersection is `K ⊓ L`, typed `\inf`. The whole group is `⊤` and the trivial subgroup is `⊥`.

```
example (K L : Subgroup G) (a : G) : a ∈ K ⊓ L ↔ a ∈ K ∧ a ∈ L :=
Subgroup.mem_inf
example (a : G) : a ∈ (⊥ : Subgroup G) ↔ a = 1 := Subgroup.mem_bot
```

To define a subgroup, give the carrier and the three proofs:

```
def evenInts : AddSubgroup ℤ where
  carrier := {n | Even n}
  zero_mem' := Even.zero
  add_mem' := by
    intro a b ha hb
```

```
    exact Even.add ha hb
  neg_mem' := by
    intro a ha
    exact Even.neg ha

example : (4 : ℤ) ∈ evenInts := ⟨2, rfl⟩
```

`Even n` means `∃ r, n = r + r`, so the pair `⟨2, rfl⟩` proves that `4` is even.

A second example, with real proofs: the centre of a group.

```
def myCenter : Subgroup G where
  carrier := {z | ∀ g : G, g * z = z * g}
  one_mem' := by
    intro g
    rw [mul_one, one_mul]
  mul_mem' := by
    intro a b ha hb g
    -- ha : a ∈ {z | ∀ (g : G), g * z = z * g}, that is, ∀ g, g * a = a * g;
hb alike
    -- ⊢ g * (a * b) = a * b * g
    rw [← mul_assoc, ha g, mul_assoc, hb g, ← mul_assoc]
  inv_mem' := by
    intro a ha g
    -- ⊢ g * a⁻¹ = a⁻¹ * g
    have h := congrArg (·⁻¹) (ha g⁻¹)
    -- h : (g⁻¹ * a)⁻¹ = (a * g⁻¹)⁻¹
    simp only [mul_inv_rev, inv_inv] at h
    -- h : a⁻¹ * g = g * a⁻¹
    exact h.symm
```

The step `congrArg (·⁻¹) h` applies the function `x ↦ x⁻¹` to both sides of an equation. It is the Lean form of "take inverses on both sides".

The same design gives `Submonoid`, `Subring`, `Subfield`, `Submodule R M` and `Ideal R`. A subobject can be used as a type: its elements are pairs `⟨x, hx⟩`, as for subtypes in Chapter 11, and it inherits the structure.

```
example (K : Subgroup G) : Group K := inferInstance
example (K : Subgroup G) (x : K) : (x : G) ∈ K := x.property
```

Kernels, images, quotients

A homomorphism has a kernel and a range, both subgroups:

```
example (f : G →* H) (a : G) : a ∈ f.ker ↔ f a = 1 := MonoidHom.mem_ker
example (f : G →* H) (y : H) : y ∈ f.range ↔ ∃ x, f x = y :=
MonoidHom.mem_range

example (f : G →* H) (a b : G) (ha : a ∈ f.ker) (hb : b ∈ f.ker) : a * b ∈
f.ker := by
  rw [MonoidHom.mem_ker] at *
  -- ha : f a = 1     hb : f b = 1     ⊢ f (a * b) = 1
  rw [map_mul, ha, hb, mul_one]

example (f : G →* H) : f.ker = ⊥ ↔ Function.Injective f :=
MonoidHom.ker_eq_bot_iff f
end
```

The quotient of `G` by a normal subgroup `N` is `G ⧸ N`, typed `\quot`. It is a group, the projection is `QuotientGroup.mk' N`, and a homomorphism that kills `N` descends through `QuotientGroup.lift`.

```
section
variable {G H : Type*} [Group G] [Group H] (N : Subgroup G) [N.Normal]

example : Group (G ⧸ N) := inferInstance

example (a : G) : (a : G ⧸ N) = 1 ↔ a ∈ N := QuotientGroup.eq_one_iff a

example (f : G →* H) (h : N ≤ f.ker) : G ⧸ N →* H := QuotientGroup.lift N f
h

-- The first isomorphism theorem.
noncomputable example (f : G →* H) : G ⧸ f.ker ≃* f.range :=
  QuotientGroup.quotientKerEquivRange f
end
```

These are the three operations of Chapter 11, `mk`, `lift` and `sound`, with the algebra already attached. Rings have the same interface: `R ⧸ I` for an ideal `I`, with `Ideal.Quotient.mk` and `Ideal.Quotient.lift`.

Where to go from here

This chapter is a map, not a course. For algebra in depth:

- The chapters *Groups and Rings* and *Linear Algebra* of Mathematics in Lean build these structures from scratch and explain the design.
- The Mathlib overview lists what exists: Sylow theorems, polynomial rings, field extensions, Galois theory, tensor products, matrices, determinants, eigenvalues.
- The page on linear algebra in Mathlib is a guided tour of `Module`, `LinearMap`, `Submodule` and `Basis`.
- The source starts in `Mathlib/Algebra/Group/Defs.lean` and `Mathlib/Algebra/Group/Subgroup/Basic.lean`.

Errors you will meet here

Message	Cause	Fix
`failed to synthesize HMul G G ?m` or `Mul G`	The type has no multiplication: a class assumption is missing	Add `[Group G]`, or the class your statement needs
`failed to synthesize Module ℝ E`	`E` was declared without its module structure	Add `[AddCommGroup E] [Module ℝ E]`

Message	Cause	Fix
`ring_nf made no progress on the goal` in a group or a noncommutative ring	`ring` needs commutativity	`group`, `abel` or `noncomm_ring`
`Fields missing: map_add'`	A bundled map was defined without all its proofs	Supply the missing field
`function expected` when applying a morphism	The term is not a bundled map, or a coercion is missing	Check its type with `#check`
`failed to synthesize N.Normal`	A quotient by a subgroup not known to be normal	Add `[N.Normal]`, or prove normality

Exercises

```
import Mathlib

variable {G H : Type*} [Group G] [Group H]

-- Exercise 1. If a and b commute, so do a and b⁻¹. A calc with three lines:
-- the first and last are closed by `group`.
example (a b : G) (h : a * b = b * a) : a * b⁻¹ = b⁻¹ * a := by
  sorry

-- Exercise 2. The kernel is stable under conjugation.
example (f : G →* H) (a : G) (ha : a ∈ f.ker) (g : G) : g * a * g⁻¹ ∈
f.ker := by
  sorry

-- Exercise 3. The intersection of two subgroups is closed under
multiplication.
example (K L : Subgroup G) (a b : G) (ha : a ∈ K ⊓ L) (hb : b ∈ K ⊓ L) : a *
b ∈ K ⊓ L := by
  sorry

-- Exercise 4. Define multiplication by 3 as an additive homomorphism.
def tripleHom : ℤ →+ ℤ where
  toFun n := 3 * n
  map_zero' := sorry
  map_add' := sorry

-- Exercise 5. A homomorphism with trivial kernel is injective.
-- First show f (a * b⁻¹) = 1.
example (f : G →* H) (hf : ∀ a, f a = 1 → a = 1) : Function.Injective f :=
by
  sorry

-- Exercise 6. In any ring, an element equal to its double is zero.
example {R : Type*} [Ring R] (a : R) (h : a + a = a) : a = 0 := by
  sorry
```

Solutions

```
import Mathlib

variable {G H : Type*} [Group G] [Group H]

-- Exercise 1.
example (a b : G) (h : a * b = b * a) : a * b⁻¹ = b⁻¹ * a := by
  calc a * b⁻¹ = b⁻¹ * (b * a) * b⁻¹ := by group
    _ = b⁻¹ * (a * b) * b⁻¹ := by rw [h]
    _ = b⁻¹ * a := by group

-- Exercise 2.
example (f : G →* H) (a : G) (ha : a ∈ f.ker) (g : G) : g * a * g⁻¹ ∈
f.ker := by
  rw [MonoidHom.mem_ker] at *
  -- ha : f a = 1     ⊢ f (g * a * g⁻¹) = 1
  rw [map_mul, map_mul, ha, mul_one, ← map_mul, mul_inv_cancel, map_one]

-- Exercise 3.
example (K L : Subgroup G) (a b : G) (ha : a ∈ K ⊓ L) (hb : b ∈ K ⊓ L) : a *
b ∈ K ⊓ L := by
  rw [Subgroup.mem_inf] at *
  exact ⟨K.mul_mem ha.1 hb.1, L.mul_mem ha.2 hb.2⟩

-- Exercise 4.
def tripleHom : ℤ →+ ℤ where
  toFun n := 3 * n
  map_zero' := by simp
  map_add' := by
    intro a b
    ring

-- Exercise 5.
example (f : G →* H) (hf : ∀ a, f a = 1 → a = 1) : Function.Injective f :=
by
  intro a b h
  have h1 : f (a * b⁻¹) = 1 := by
    rw [map_mul, map_inv, h, mul_inv_cancel]
  have h2 := hf _ h1
  -- h2 : a * b⁻¹ = 1
  calc a = a * b⁻¹ * b := by group
    _ = 1 * b := by rw [h2]
    _ = b := one_mul b

-- Exercise 6.
example {R : Type*} [Ring R] (a : R) (h : a + a = a) : a = 0 := by
  have h' : a + a = a + 0 := by rw [h, add_zero]
  exact add_left_cancel h'
```

Exercise 3 has a one-line solution: `K ⊓ L` is itself a subgroup, so `(K ⊓ L).mul_mem ha hb` proves the goal. Exercise 2 is closed by `simp [ha]` after the first `rw`.

Chapter 17: Filters, Limits and Topology

Mathlib expresses every kind of limit with one definition, `Tendsto f l₁ l₂`, in which the filters `l₁` and `l₂` say how the input and the output are approached.

This is the chapter where Mathlib's analysis stops looking like a textbook. The ε-N definitions are still there, as theorems you can rewrite with. What changes is that you rarely need them.

One definition for all limits

A first course defines the limit of a sequence, the limit of a function at a point, limits at infinity, one-sided limits and infinite limits. That is a dozen definitions, each with its own algebra of limits.

Mathlib has one statement, read "`f` tends to `l₂` along `l₁`":

```
import Mathlib

open Filter Topology

#check @Tendsto       -- (α → β) → Filter α → Filter β → Prop
```

Mathematical statement	In Lean
`uₙ → a` as `n → ∞`	`Tendsto u atTop (𝓝 a)`
`f(x) → b` as `x → a`	`Tendsto f (𝓝 a) (𝓝 b)`
`f(x) → b` as `x → a`, `x ≠ a`	`Tendsto f (𝓝[≠] a) (𝓝 b)`
`f(x) → b` as `x → a⁺`	`Tendsto f (𝓝[>] a) (𝓝 b)`
`f(x) → b` as `x → +∞`	`Tendsto f atTop (𝓝 b)`
`f(x) → +∞` as `x → +∞`	`Tendsto f atTop atTop`
`f(x) → b` as `x → a` within `s`	`Tendsto f (𝓝[s] a) (𝓝 b)`

The symbol `𝓝` is typed `\nhds`, and needs `open Topology`. The filters `atTop` and `atBot` need `open Filter`.

What a filter is

A filter on `α` is a collection of subsets of `α`, thought of as the "large" sets. It contains the whole type, is closed under finite intersection, and contains every superset of its members.

Filter	Its large sets	Meaning
`atTop` on `ℕ`	Sets containing all `n ≥ N` for some `N`	"For `n` large enough"

Filter	Its large sets	Meaning
`𝓝 a`	Sets containing a ball around `a`	"For `x` close enough to `a`"
`𝓝[s] a`	Sets containing `s ∩ ball`	"Close to `a`, and in `s`"
`𝓟 s`	Sets containing `s`	"On `s`"
`ae μ`	Sets whose complement has measure zero	"Almost everywhere"

```
#check @Filter.mem_atTop_sets -- s ∈ atTop ↔ ∃ a, ∀ (b : α), a ≤ b → b ∈ s
#check @Metric.mem_nhds_iff   -- s ∈ 𝓝 x ↔ ∃ ε > 0, Metric.ball x ε ⊆ s
```

`Tendsto f l₁ l₂` says that the preimage under `f` of every large set of `l₂` is a large set of `l₁`:

```
#check @tendsto_def      -- Tendsto f l₁ l₂ ↔ ∀ s ∈ l₂, f ⁻¹' s ∈ l₁
```

For `l₁ = atTop` and `l₂ = 𝓝 a`, this reads: for every neighbourhood of `a`, the sequence is in it from some rank on. That is the usual definition.

The file that defines filters is `Mathlib/Order/Filter/Defs.lean`. The chapter *Topology* of Mathematics in Lean develops the idea at length.

Getting the ε back

In a metric space, each instance of `Tendsto` has an ε characterization, stated as an `iff` that you rewrite with.

```
#check @Metric.tendsto_atTop
-- Tendsto u atTop (𝓝 a) ↔ ∀ ε > 0, ∃ N, ∀ n ≥ N, dist (u n) a < ε

#check @Metric.tendsto_nhds_nhds
-- Tendsto f (𝓝 a) (𝓝 b) ↔
--   ∀ ε > 0, ∃ δ > 0, ∀ ⦃x : α⦄, dist x a < δ → dist (f x) b < ε

#check Real.dist_eq      -- dist x y = |x - y|
```

Here is a limit proved by hand, the way an analysis course does it:

```
example : Tendsto (fun n : ℕ ↦ 1 / (n + 1 : ℝ)) atTop (𝓝 0) := by
  rw [Metric.tendsto_atTop]
  intro ε hε
  -- ⊢ ∃ N, ∀ n ≥ N, dist (1 / (↑n + 1)) 0 < ε
  obtain ⟨N, hN⟩ := exists_nat_one_div_lt hε       -- hN : 1 / (↑N + 1) < ε
  use N
  intro n hn
  rw [Real.dist_eq, sub_zero]
  -- ⊢ |1 / (↑n + 1)| < ε
  have h1 : (0 : ℝ) < 1 / (n + 1) := by positivity
  rw [abs_of_pos h1]
  have h2 : (1 : ℝ) / (n + 1) ≤ 1 / (N + 1) := by gcongr
  linarith
```

Every step is one you know: choose `N` by the Archimedean property, remove the absolute value, compare `1 / (n + 1)` with `1 / (N + 1)`. The `gcongr` step uses `hn : n ≥ N` from the context.

The library has this limit, and the proof is one name:

```
example : Tendsto (fun n : ℕ ↦ 1 / (n + 1 : ℝ)) atTop (𝓝 0) :=
  tendsto_one_div_add_atTop_nhds_zero_nat
```

The algebra of limits

The usual rules are lemmas applied with dot notation to a `Tendsto` hypothesis. They hold for every filter, so you learn them once.

```
example (u v : ℕ → ℝ) (a b : ℝ) (hu : Tendsto u atTop (𝓝 a)) (hv : Tendsto
v atTop (𝓝 b)) :
    Tendsto (fun n ↦ u n + v n) atTop (𝓝 (a + b)) := hu.add hv

example (u v : ℕ → ℝ) (a b : ℝ) (hu : Tendsto u atTop (𝓝 a)) (hv : Tendsto
v atTop (𝓝 b)) :
    Tendsto (fun n ↦ u n * v n) atTop (𝓝 (a * b)) := hu.mul hv

example (u : ℕ → ℝ) (a : ℝ) (hu : Tendsto u atTop (𝓝 a)) :
    Tendsto (fun n ↦ u n ^ 2 + 3) atTop (𝓝 (a ^ 2 + 3)) := (hu.pow
2).add_const 3
```

Lemma	Conclusion
`hu.add hv`, `hu.sub hv`, `hu.mul hv`	Limit of a sum, difference, product
`hu.div hv hb`	Limit of a quotient, given `hb : b ≠ 0`
`hu.const_mul c`, `hu.mul_const c`	Multiplication by a constant
`hu.const_add c`, `hu.add_const c`	Addition of a constant
`hu.pow n`, `hu.neg`, `hu.abs`	Powers, opposite, absolute value
`tendsto_const_nhds`	A constant tends to itself
`hf.comp hu`	Composition of limits
`(hf.tendsto a).comp hu`	A continuous `f` applied to a convergent sequence
`tendsto_nhds_unique ha hb`	The limit is unique
`le_of_tendsto' ha h`, `ge_of_tendsto' ha h`	A bound on the terms passes to the limit

Lemma	Conclusion
`squeeze_zero h₁ h₂ hw`	`0 ≤ vₙ ≤ wₙ` and `wₙ → 0` give `vₙ → 0`
`tendsto_of_tendsto_of_tendsto_of_le_of_le`	The squeeze theorem
`hu.congr h`, `hu.congr' h`	Replace the function by an equal, or eventually equal, one

The value of the limit in the conclusion is what the lemma computes, not what you want. After `h.const_add 2`, the limit reads `2 + 0`. The standard finish is `simpa using`:

```
example : Tendsto (fun n : ℕ ↦ 2 + 1 / (n + 1 : ℝ)) atTop (𝓝 2) := by
  have h := tendsto_one_div_add_atTop_nhds_zero_nat (𝕜 := ℝ)
  have h2 := h.const_add 2
  -- h2 : Tendsto (fun x ↦ 2 + 1 / (↑x + 1)) atTop (𝓝 (2 + 0))
  simpa using h2
```

The argument `(𝕜 := ℝ)` tells Lean in which field the general lemma is used. Without it, Lean cannot guess before seeing the goal.

Some standard limits, to be found with `exact?` or by guessing `tendsto_…`:

Name	Statement
`tendsto_one_div_atTop_nhds_zero_nat`	`1 / n → 0`
`tendsto_const_div_atTop_nhds_zero_nat C`	`C / n → 0`
`tendsto_pow_atTop_nhds_zero_of_lt_one`	`rⁿ → 0` for `0 ≤ r < 1`
`tendsto_natCast_atTop_atTop`	`(n : ℝ) → +∞`
`tendsto_inv_atTop_zero`	`x⁻¹ → 0` as `x → +∞`
`Real.tendsto_exp_atTop`	`exp x → +∞` as `x → +∞`
`Real.tendsto_exp_neg_atTop_nhds_zero`	`exp (-x) → 0` as `x → +∞`

Eventually

"For `n` large enough, `p n`" is written `∀ᶠ n in atTop, p n`, typed `\allf`. It means that the set of `n` where `p n` holds belongs to the filter. The same notation with `𝓝 a` means "for `x` close enough to `a`".

```
example : ∀ᶠ n : ℕ in atTop, 100 < n := eventually_gt_atTop 100

example (p : ℕ → Prop) (h : ∀ᶠ n in atTop, p n) : ∃ N, ∀ n ≥ N, p n :=
  eventually_atTop.mp h
```

Its strength is that eventual properties combine without choosing a maximum of ranks. The tactic `filter_upwards` does it: list the eventual facts you have, and prove the goal at a point where all of them hold.

```
example (u v : ℕ → ℝ) (hu : ∀ᶠ n in atTop, 0 < u n) (hv : ∀ᶠ n in atTop, u n
≤ v n) :
    ∀ᶠ n in atTop, 0 < v n := by
  filter_upwards [hu, hv] with n h1 h2
  -- h1 : 0 < u n     h2 : u n ≤ v n     ⊢ 0 < v n
  linarith
```

Compare with the ε-N proof of the same fact, which would name `N₁`, `N₂`, `max N₁ N₂` and two inequalities.

A limit gives eventual inequalities directly:

```
example (u : ℕ → ℝ) (a : ℝ) (hu : Tendsto u atTop (𝓝 a)) (ha : 0 < a) :
    ∃ N, ∀ n ≥ N, a / 2 < u n := by
  have h : ∀ᶠ n in atTop, a / 2 < u n := hu.eventually_const_lt (by
linarith)
  exact eventually_atTop.mp h
```

Lemma	Use
`Filter.Eventually.of_forall`	A property true everywhere is eventually true
`h.mono`	Weaken an eventual property
`h₁.and h₂`	Combine two eventual properties
`eventually_atTop`	Translate `∀ᶠ n in atTop` to `∃ N, ∀ n ≥ N`
`Metric.eventually_nhds_iff`	Translate `∀ᶠ y in 𝓝 x` to `∃ ε > 0, ∀ y, dist y x < ε →` …
`hu.eventually_const_lt`, `hu.eventually_lt_const`	From `u → a` and `c < a`, eventually `c < u n`

An ε proof, when you want one

The sum of two convergent sequences, by hand. This is `hu.add hv`, written out.

```
example (u v : ℕ → ℝ) (a b : ℝ) (hu : Tendsto u atTop (𝓝 a)) (hv : Tendsto
v atTop (𝓝 b)) :
    Tendsto (fun n ↦ u n + v n) atTop (𝓝 (a + b)) := by
  rw [Metric.tendsto_atTop] at *
  intro ε hε
  obtain ⟨N₁, h₁⟩ := hu (ε / 2) (by positivity)
  obtain ⟨N₂, h₂⟩ := hv (ε / 2) (by positivity)
  refine ⟨max N₁ N₂, ?_⟩
  intro n hn
  have hn₁ : n ≥ N₁ := le_trans (le_max_left _ _) hn
  have hn₂ : n ≥ N₂ := le_trans (le_max_right _ _) hn
  have e₁ := h₁ n hn₁
  have e₂ := h₂ n hn₂
```

```
  rw [Real.dist_eq] at *
  -- e₁ : |u n - a| < ε / 2     e₂ : |v n - b| < ε / 2     ⊢ |u n + v n - (a
+ b)| < ε
  calc |u n + v n - (a + b)| = |(u n - a) + (v n - b)| := by ring_nf
    _ ≤ |u n - a| + |v n - b| := abs_add_le _ _
    _ < ε / 2 + ε / 2 := by gcongr
    _ = ε := by ring
```

Every ingredient comes from earlier chapters: `obtain` on the hypotheses with `ε / 2`, `max` for the rank, and the `calc` of Chapter 13.

Continuity

`Continuous f` is defined topologically. `ContinuousAt f a` is by definition `Tendsto f (𝓝 a) (𝓝 (f a))`.

Statement	Meaning
`Continuous f`	Continuous everywhere
`ContinuousAt f a`	Continuous at the point `a`
`ContinuousOn f s`	The restriction of `f` to `s` is continuous
`ContinuousWithinAt f s a`	Continuous at `a` along `s`
`UniformContinuous f`	Uniformly continuous

You almost never prove continuity by hand. The tactic `fun_prop` proves that a function built from continuous pieces is continuous:

```
example : Continuous (fun x : ℝ ↦ x ^ 2 + 3 * x + 1) := by fun_prop

example : Continuous (fun x : ℝ ↦ Real.exp (x ^ 2) * |x|) := by fun_prop

example (f g : ℝ → ℝ) (hf : Continuous f) (hg : Continuous g) :
    Continuous (fun x ↦ f x * g (x + 1)) := by fun_prop
```

It uses the hypotheses `hf` and `hg` from the context. When a side condition is needed, such as a nonzero denominator, give it a tactic to discharge it:

```
example (x : ℝ) (hx : x ≠ 0) : ContinuousAt (fun x : ℝ ↦ 1 / x) x := by
  fun_prop (disch := assumption)
```

The term-mode equivalent composes lemmas such as `Continuous.add`, `Continuous.mul`, `Continuous.comp`, `continuous_id` and `continuous_const`. An older tactic, `continuity`, does the same job more slowly.

The ε-δ characterization is `Metric.continuous_iff`:

```
example : Continuous (fun x : ℝ ↦ 2 * x + 1) := by
  rw [Metric.continuous_iff]
  intro a ε hε
  -- ⊢ ∃ δ > 0, ∀ (a_1 : ℝ), dist a_1 a < δ → dist (2 * a_1 + 1) (2 * a + 1)
< ε
  refine ⟨ε / 2, by positivity, ?_⟩
```

```
  intro x hx
  rw [Real.dist_eq] at hx ⊢
  -- hx : |x - a| < ε / 2     ⊢ |2 * x + 1 - (2 * a + 1)| < ε
  have h : 2 * x + 1 - (2 * a + 1) = 2 * (x - a) := by ring
  rw [h, abs_mul, abs_two]
  linarith
```

Open, closed, compact

```
open Set

example : IsOpen (Ioo (0 : ℝ) 1) := isOpen_Ioo
example : IsClosed (Icc (0 : ℝ) 1) := isClosed_Icc
example : IsCompact (Icc (0 : ℝ) 1) := isCompact_Icc
```

Sets defined by continuous functions and inequalities are open or closed by one lemma:

```
example (f : ℝ → ℝ) (hf : Continuous f) : IsOpen {x | 0 < f x} :=
  isOpen_lt continuous_const hf

example (f : ℝ → ℝ) (hf : Continuous f) : IsClosed {x | f x ≤ 3} :=
  isClosed_le hf continuous_const

example (f : ℝ → ℝ) (hf : Continuous f) : IsOpen (f ⁻¹' Ioo 0 1) :=
  isOpen_Ioo.preimage hf
```

In a metric space, openness unfolds to balls with `Metric.isOpen_iff`:

```
example : IsOpen {x : ℝ | 0 < x} := by
  rw [Metric.isOpen_iff]
  intro x hx
  -- hx : x ∈ {x | 0 < x}     ⊢ ∃ ε > 0, Metric.ball x ε ⊆ {x | 0 < x}
  refine ⟨x, hx, ?_⟩
  intro y hy
  rw [Metric.mem_ball, Real.dist_eq] at hy
  have h := abs_lt.mp hy
  -- h : -x < y - x ∧ y - x < x     ⊢ y ∈ {x | 0 < x}
  change 0 < y
  linarith
```

Notion	Lean	Key lemmas
Closure, interior	`closure s`, `interior s`	`Metric.mem_closure_iff`, `mem_closure_iff_seq_limit`
Closed sets and limits	`IsClosed s`	`IsClosed.mem_of_tendsto`: a limit of points of `s` is in `s`
Dense	`Dense s`	`Rat.denseRange_cast`
Compact	`IsCompact s`	`isCompact_Icc`, `Metric.isCompact_iff_isClosed_bounded`
Sequential compactness		`IsCompact.tendsto_subseq`: a sequence in a compact set has a convergent subsequence

Notion	Lean	Key lemmas
Cauchy, complete	`CauchySeq u`, `CompleteSpace X`	`Metric.cauchySeq_iff`, `cauchySeq_tendsto_of_complete`

The big theorems of one-variable analysis

A continuous function on a nonempty compact set attains its maximum:

```
example (f : ℝ → ℝ) (hf : ContinuousOn f (Icc 0 1)) :
    ∃ x ∈ Icc (0 : ℝ) 1, ∀ y ∈ Icc (0 : ℝ) 1, f y ≤ f x := by
  have hne : (Icc (0 : ℝ) 1).Nonempty := ⟨0, by simp⟩
  obtain ⟨x, hx, hmax⟩ := isCompact_Icc.exists_isMaxOn hne hf
  exact ⟨x, hx, fun y hy ↦ hmax hy⟩
```

The intermediate value theorem says that the image of an interval contains the interval between the end values:

```
#check @intermediate_value_Icc
-- a ≤ b → ∀ {f : α → δ},
--   ContinuousOn f (Icc a b) → Icc (f a) (f b) ⊆ f '' Icc a b

example : ∃ x ∈ Icc (0 : ℝ) 1, x ^ 3 + x = 1 := by
  have hf : ContinuousOn (fun x : ℝ ↦ x ^ 3 + x) (Icc 0 1) := by fun_prop
  have h := intermediate_value_Icc (show (0 : ℝ) ≤ 1 by norm_num) hf
  -- h : Icc (0 ^ 3 + 0) (1 ^ 3 + 1) ⊆ (fun x ↦ x ^ 3 + x) '' Icc 0 1
  apply h
  -- ⊢ 1 ∈ Icc (0 ^ 3 + 0) (1 ^ 3 + 1)
  constructor <;> norm_num
```

The goal `∃ x ∈ Icc 0 1, x ^ 3 + x = 1` is, by definition, membership of `1` in the image. So `apply h` reduces it to checking that `1` lies between `f 0 = 0` and `f 1 = 2`.

The other classics are there under predictable names: `IsCompact.exists_isMinOn`, `IsCompact.uniformContinuousOn_of_continuous` for the Heine–Cantor theorem, and `cauchySeq_tendsto_of_complete` for completeness of `ℝ`.

Metric and normed spaces

The results above are stated for metric or topological spaces, and `ℝ` is one instance. To work in a general space, assume the structure:

```
section
variable {X : Type*} [MetricSpace X] (x y z : X)

example : dist x z ≤ dist x y + dist y z := dist_triangle x y z
example : dist x y = dist y x := dist_comm x y
example : dist x y = 0 ↔ x = y := dist_eq_zero
end

section
variable {E : Type*} [NormedAddCommGroup E] [NormedSpace ℝ E] (x y : E) (c :
ℝ)
```

```
example : ‖x + y‖ ≤ ‖x‖ + ‖y‖ := norm_add_le x y
example : ‖c • x‖ = |c| * ‖x‖ := by rw [norm_smul, Real.norm_eq_abs]
example : dist x y = ‖x - y‖ := dist_eq_norm x y
example : |‖x‖ - ‖y‖| ≤ ‖x - y‖ := abs_norm_sub_norm_le x y
end
```

A real normed space is `[NormedAddCommGroup E] [NormedSpace ℝ E]`. A Banach space adds `[CompleteSpace E]`. A Hilbert space is `[NormedAddCommGroup E] [InnerProductSpace ℝ E] [CompleteSpace E]`. The norm is typed `\norm`. On `ℝ`, `‖x‖ = |x|` by `Real.norm_eq_abs`.

Series

An infinite sum is `∑' n, f n`, typed `\sum'`. Like division, it is total: it equals the sum when the series is summable, and `0` otherwise. The honest statements are `HasSum f a` and `Summable f`.

```
example : ∑' n : ℕ, (1 / 2 : ℝ) ^ n = 2 := tsum_geometric_two

example : Summable (fun n : ℕ ↦ (1 / 3 : ℝ) ^ n) :=
  summable_geometric_of_lt_one (by norm_num) (by norm_num)

example : ∑' n : ℕ, (1 / 3 : ℝ) ^ n = 3 / 2 := by
  rw [tsum_geometric_of_lt_one (by norm_num) (by norm_num)]
  -- ⊢ (1 - 1 / 3)⁻¹ = 3 / 2
  norm_num
```

Lemma	Statement
`hf.hasSum`	A summable `f` has sum `∑' n, f n`
`hf.tsum_eq`	`HasSum f a` gives `∑' n, f n = a`
`hf.tendsto_sum_nat`	`HasSum f a` gives convergence of the partial sums to `a`
`hf.tendsto_atTop_zero`	The terms of a summable series tend to `0`
`Summable.of_nonneg_of_le`	Comparison test
`summable_geometric_of_lt_one`	Geometric series
`Real.summable_one_div_nat_pow`	`∑ 1 / n^p` converges iff `1 < p`

`HasSum` means unconditional convergence: the sum does not depend on the order of the terms. For real series this is absolute convergence.

Errors you will meet here

Message or symptom	Cause	Fix
`Function expected at` $\mathcal{N}$ (the hint says $\mathcal{N}$ is unknown), or `Unknown identifier `atTop``	The notation is not open	`open Filter Topology`
`Type mismatch` with $\mathcal{N}$ `(2 + 0)` against $\mathcal{N}$ `2`	A limit lemma computed the limit in unsimplified form	`simpa using h`
`typeclass instance problem is stuck`, when using a general limit lemma with `have`	Lean does not know the field	Add `(𝕜 := ℝ)`
`fun_prop` fails with a missing side goal	A condition such as `x ≠ 0` is needed	`fun_prop (disch := assumption)`, or `(disch := simp)`
The goal has `dist x y` and `linarith` fails	`dist` is opaque to arithmetic	`rw [Real.dist_eq]`
`Tendsto` goal about `fun n ↦ …` does not match a lemma about `u`	The lemma's function is written differently	`hu.congr` with a pointwise equality, or `simpa using`

Exercises

```
import Mathlib

open Filter Topology Set

-- Exercise 1. Combine the standard limit with const_mul and const_add.
example : Tendsto (fun n : ℕ ↦ 3 + 2 * (1 / (n + 1 : ℝ))) atTop (𝓝 3) := by
  sorry

-- Exercise 2. Squeeze between 0 and 1 / (n + 1).
example : Tendsto (fun n : ℕ ↦ 1 / (n + 1 : ℝ) ^ 2) atTop (𝓝 0) := by
  sorry

-- Exercise 3. A bound passes to the limit.
example (u : ℕ → ℝ) (a : ℝ) (hu : Tendsto u atTop (𝓝 a)) (h : ∀ n, u n ≤ 5)
: a ≤ 5 := by
  sorry

-- Exercise 4. Combine two eventual facts.
example (u v : ℕ → ℝ) (hu : ∀ᶠ n in atTop, 1 < u n) (hv : ∀ᶠ n in atTop, u n
< v n) :
    ∀ᶠ n in atTop, 1 < v n := by
  sorry
```

```
-- Exercise 5. Once with fun_prop, once with ε and δ.
example : Continuous (fun x : ℝ ↦ 3 * x - 2) := by
  sorry

-- Exercise 6. A convergent sequence is eventually bounded. Use ε = 1.
example (u : ℕ → ℝ) (a : ℝ) (hu : Tendsto u atTop (𝓝 a)) :
    ∃ N, ∀ n ≥ N, |u n| < |a| + 1 := by
  sorry

-- Exercise 7. A closed set defined by an inequality.
example : IsClosed {x : ℝ | x ^ 2 ≤ 4} := by
  sorry

-- Exercise 8. The intermediate value theorem.
example : ∃ x ∈ Icc (0 : ℝ) 2, x ^ 2 + x = 3 := by
  sorry

-- Exercise 9. A geometric series.
example : ∑' n : ℕ, (1 / 4 : ℝ) ^ n = 4 / 3 := by
  sorry
```

Solutions

```
import Mathlib

open Filter Topology Set

-- Exercise 1.
example : Tendsto (fun n : ℕ ↦ 3 + 2 * (1 / (n + 1 : ℝ))) atTop (𝓝 3) := by
  have h := tendsto_one_div_add_atTop_nhds_zero_nat (𝕜 := ℝ)
  have h2 := (h.const_mul 2).const_add 3
  -- h2 : Tendsto (fun x ↦ 3 + 2 * (1 / (↑x + 1))) atTop (𝓝 (3 + 2 * 0))
  simpa using h2

-- Exercise 2.
example : Tendsto (fun n : ℕ ↦ 1 / (n + 1 : ℝ) ^ 2) atTop (𝓝 0) := by
  apply squeeze_zero (g := fun n : ℕ ↦ 1 / (n + 1 : ℝ))
  · intro n
    positivity
  · intro n
    -- ⊢ 1 / (↑n + 1) ^ 2 ≤ 1 / (↑n + 1)
    have h : (1 : ℝ) ≤ n + 1 := by linarith [Nat.cast_nonneg (α := ℝ) n]
    have h2 : (n + 1 : ℝ) ≤ (n + 1) ^ 2 := by nlinarith
    gcongr
  · exact tendsto_one_div_add_atTop_nhds_zero_nat

-- Exercise 3.
example (u : ℕ → ℝ) (a : ℝ) (hu : Tendsto u atTop (𝓝 a)) (h : ∀ n, u n ≤ 5)
: a ≤ 5 :=
  le_of_tendsto' hu h

-- Exercise 4.
example (u v : ℕ → ℝ) (hu : ∀ᶠ n in atTop, 1 < u n) (hv : ∀ᶠ n in atTop, u n
< v n) :
    ∀ᶠ n in atTop, 1 < v n := by
  filter_upwards [hu, hv] with n h1 h2
  linarith

-- Exercise 5.
```

```
example : Continuous (fun x : ℝ ↦ 3 * x - 2) := by fun_prop

example : Continuous (fun x : ℝ ↦ 3 * x - 2) := by
  rw [Metric.continuous_iff]
  intro a ε hε
  refine ⟨ε / 3, by positivity, ?_⟩
  intro x hx
  rw [Real.dist_eq] at hx ⊢
  have h : 3 * x - 2 - (3 * a - 2) = 3 * (x - a) := by ring
  rw [h, abs_mul, abs_of_pos (by norm_num : (0 : ℝ) < 3)]
  -- hx : |x - a| < ε / 3     ⊢ 3 * |x - a| < ε
  linarith

-- Exercise 6.
example (u : ℕ → ℝ) (a : ℝ) (hu : Tendsto u atTop (𝓝 a)) :
    ∃ N, ∀ n ≥ N, |u n| < |a| + 1 := by
  rw [Metric.tendsto_atTop] at hu
  obtain ⟨N, hN⟩ := hu 1 (by norm_num)
  refine ⟨N, fun n hn ↦ ?_⟩
  have h := hN n hn
  rw [Real.dist_eq] at h
  -- h : |u n - a| < 1
  have h2 : |u n| - |a| ≤ |u n - a| := abs_sub_abs_le_abs_sub _ _
  linarith

-- Exercise 7.
example : IsClosed {x : ℝ | x ^ 2 ≤ 4} :=
  isClosed_le (by fun_prop) continuous_const

-- Exercise 8.
example : ∃ x ∈ Icc (0 : ℝ) 2, x ^ 2 + x = 3 := by
  have hf : ContinuousOn (fun x : ℝ ↦ x ^ 2 + x) (Icc 0 2) := by fun_prop
  have h := intermediate_value_Icc (show (0 : ℝ) ≤ 2 by norm_num) hf
  apply h
  constructor <;> norm_num

-- Exercise 9.
example : ∑' n : ℕ, (1 / 4 : ℝ) ^ n = 4 / 3 := by
  rw [tsum_geometric_of_lt_one (by norm_num) (by norm_num)]
  norm_num
```

Chapter 18: Calculus

A derivative in Mathlib is stated with `HasDerivAt f f' x`, proved by composing the rules of differentiation as lemmas, and connected to integrals by the fundamental theorem of calculus.

Three ways to speak of a derivative

Statement	Meaning	Use it for
`HasDerivAt f f' x`	`f` is differentiable at `x`, and its derivative there is `f'`	Proving and computing. It carries all the information
`DifferentiableAt ℝ f x`	`f` has some derivative at `x`	Hypotheses of theorems
`deriv f x`	The derivative as a number. Equal to `0` when `f` is not differentiable at `x`	Writing formulas

```
import Mathlib

open Real Set

example (f : ℝ → ℝ) (f' x : ℝ) (h : HasDerivAt f f' x) : deriv f x = f' :=
h.deriv

example (f : ℝ → ℝ) (f' x : ℝ) (h : HasDerivAt f f' x) : DifferentiableAt ℝ
f x :=
  h.differentiableAt

example (f : ℝ → ℝ) (x : ℝ) (h : DifferentiableAt ℝ f x) : HasDerivAt f
(deriv f x) x :=
  h.hasDerivAt
```

`deriv f x` is a junk value when `f` is not differentiable, in the sense of Chapter 14. The function `x ↦ |x|` has `deriv` equal to `0` at `0`. A formula involving `deriv` means something only next to a differentiability hypothesis.

The global versions are `Differentiable ℝ f` and `DifferentiableOn ℝ f s`. The first argument `ℝ` is the field over which you differentiate, since a function on `ℂ` can be real or complex differentiable.

Behind these stands the Fréchet derivative, for maps between normed spaces: `HasFDerivAt f f' x`, where `f'` is a continuous linear map `E →L[ℝ] F`, and `fderiv ℝ f x`. `HasDerivAt` is the special case of a function of one real variable. The definitions are in `Mathlib/Analysis/Calculus/Deriv/Basic.lean` and `Mathlib/Analysis/Calculus/FDeriv/Basic.lean`.

Computing a derivative

Each rule of differentiation is a lemma that builds a `HasDerivAt` from smaller ones.

Rule	Lemma	Conclusion
Identity	`hasDerivAt_id x`	`HasDerivAt id 1 x`
Constant	`hasDerivAt_const x c`	`HasDerivAt (fun _ ↦ c) 0 x`
Power	`hasDerivAt_pow n x`	`HasDerivAt (fun x ↦ x ^ n) (↑n * x ^ (n - 1)) x`
Sum, difference	`hf.add hg`, `hf.sub hg`	Derivative `f' + g'`, `f' - g'`
Product	`hf.mul hg`	Derivative `f' * g x + f x * g'`
Constant factor	`hf.const_mul c`	Derivative `c * f'`
Quotient	`hf.div hg hne`	Derivative `(f' * g x - f x * g') / g x ^ 2`
Power of a function	`hf.pow n`	Derivative `↑n * f x ^ (n - 1) * f'`
Chain rule	`hg.comp x hf`	Derivative `g' * f'`
`exp`, `sin`, `cos`	`Real.hasDerivAt_exp x`, `Real.hasDerivAt_sin x`, `Real.hasDerivAt_cos x`	Derivatives `exp x`, `cos x`, `-sin x`
`exp`, `sin`, `cos`, `log`, `sqrt` of a function	`hf.exp`, `hf.sin`, `hf.cos`, `hf.log hne`, `hf.sqrt hne`	The chain rule already applied

A product needs no adjustment:

```
example (x : ℝ) : HasDerivAt (fun x : ℝ ↦ Real.sin x * Real.exp x)
    (Real.cos x * Real.exp x + Real.sin x * Real.exp x) x :=
  (Real.hasDerivAt_sin x).mul (Real.hasDerivAt_exp x)
```

The one difficulty is that the derivative a lemma produces must match the derivative you wrote. `hasDerivAt_pow 2 x` produces `↑2 * x ^ (2 - 1)`, and you want `2 * x`.

```
example (x : ℝ) : HasDerivAt (fun x : ℝ ↦ x ^ 2) (2 * x) x := by
  have h := hasDerivAt_pow 2 x
  -- h : HasDerivAt (fun x ↦ x ^ 2) (↑2 * x ^ (2 - 1)) x
  simpa using h
```

There are two ways to close the gap.

- `simpa using h` simplifies both sides until they agree. It handles casts and exponents such as `2 - 1`.
- `convert h using 1` accepts `h` and leaves the mismatch as a goal, usually an equality between two expressions for the derivative, which `ring` proves.

```
example (x : ℝ) : HasDerivAt (fun x : ℝ ↦ (x ^ 2 + 1) ^ 5) (10 * x * (x ^ 2
+ 1) ^ 4) x := by
  have h1 : HasDerivAt (fun x : ℝ ↦ x ^ 2 + 1) (2 * x) x := by
    simpa using (hasDerivAt_pow 2 x).add_const 1
  have h2 := h1.pow 5
  -- h2 : HasDerivAt ((fun x ↦ x ^ 2 + 1) ^ 5) (↑5 * (x ^ 2 + 1) ^ (5 - 1) *
(2 * x)) x
  convert h2 using 1
  -- ⊢ 10 * x * (x ^ 2 + 1) ^ 4 = ↑5 * (x ^ 2 + 1) ^ (5 - 1) * (2 * x)
  ring
```

The method is always the same: build the derivative from the inside out with `have`, one rule per line, and fix the form at the end.

```
example (x : ℝ) : HasDerivAt (fun x : ℝ ↦ x ^ 2 + 3 * x) (2 * x + 3) x := by
  have h1 : HasDerivAt (fun x : ℝ ↦ x ^ 2) (2 * x) x := by simpa using
hasDerivAt_pow 2 x
  have h2 : HasDerivAt (fun x : ℝ ↦ 3 * x) 3 x := by simpa using
(hasDerivAt_id x).const_mul 3
  exact h1.add h2

example (x : ℝ) : HasDerivAt (fun x : ℝ ↦ Real.exp (x ^ 2)) (Real.exp (x ^
2) * (2 * x)) x := by
  have h1 : HasDerivAt (fun x : ℝ ↦ x ^ 2) (2 * x) x := by simpa using
hasDerivAt_pow 2 x
  exact h1.exp
```

Statements with `deriv`

To prove a formula for `deriv`, prove the `HasDerivAt` statement and take `.deriv`:

```
example (x : ℝ) : deriv (fun x : ℝ ↦ Real.sin (x ^ 2)) x = 2 * x * Real.cos
(x ^ 2) := by
  have h : HasDerivAt (fun x : ℝ ↦ Real.sin (x ^ 2)) (Real.cos (x ^ 2) * (2
* x)) x := by
    have h1 : HasDerivAt (fun x : ℝ ↦ x ^ 2) (2 * x) x := by simpa using
hasDerivAt_pow 2 x
    exact h1.sin
  rw [h.deriv]
  ring
```

`simp` computes the derivative of a single standard function. It often stops on sums and products, because each rule for `deriv` needs differentiability hypotheses:

```
example (x : ℝ) : deriv (fun x : ℝ ↦ x ^ 2) x = 2 * x := by simp
example (x : ℝ) : deriv Real.exp x = Real.exp x := by simp

#check @deriv_fun_add
-- DifferentiableAt 𝕜 f x → DifferentiableAt 𝕜 g x →
--   deriv (fun y ↦ f y + g y) x = deriv f x + deriv g x

example (x : ℝ) : deriv (fun x : ℝ ↦ x ^ 2 + 3 * x) x = 2 * x + 3 := by
  rw [deriv_fun_add (by fun_prop) (by fun_prop)]
  simp
```

The `HasDerivAt` route has no side conditions, which is why it is the one to learn first.

Differentiability

`fun_prop` proves differentiability the way it proves continuity:

```
example : Differentiable ℝ (fun x : ℝ ↦ Real.exp (x ^ 2) * Real.sin x) := by
fun_prop

example (x : ℝ) (hx : x ≠ 0) : DifferentiableAt ℝ (fun x : ℝ ↦ 1 / x) x :=
by
  fun_prop (disch := assumption)

example (f : ℝ → ℝ) (hf : Differentiable ℝ f) : Continuous f :=
hf.continuous
```

Higher regularity is `ContDiff ℝ n f`, for `n` a natural number, `∞` for smooth or `ω` for analytic (notations from `open scoped ContDiff`), and `iteratedDeriv n f` is the `n`-th derivative.

The theorems

Theorem	Name
Rolle	`exists_deriv_eq_zero`
Mean value theorem	`exists_deriv_eq_slope`
Zero derivative means constant	`is_const_of_deriv_eq_zero`
Nonnegative derivative means monotone	`monotone_of_deriv_nonneg`, `monotoneOn_of_deriv_nonneg`
Positive derivative means strictly monotone	`strictMono_of_deriv_pos`, `strictMonoOn_of_deriv_pos`
The derivative vanishes at a local extremum	`IsLocalMin.deriv_eq_zero`, `IsLocalMax.deriv_eq_zero`
Mean value inequality	`Convex.norm_image_sub_le_of_norm_deriv_le`
Convexity from the second derivative	`convexOn_of_deriv2_nonneg`
Taylor's theorem with Lagrange remainder	`taylor_mean_remainder_lagrange`

```
example (f : ℝ → ℝ) (a b : ℝ) (hab : a < b) (hfc : ContinuousOn f (Icc a b))
    (hfd : DifferentiableOn ℝ f (Ioo a b)) :
    ∃ c ∈ Ioo a b, deriv f c = (f b - f a) / (b - a) :=
  exists_deriv_eq_slope f hab hfc hfd

example (f : ℝ → ℝ) (hf : Differentiable ℝ f) (h : ∀ x, deriv f x = 0) (x
y : ℝ) :
    f x = f y :=
  is_const_of_deriv_eq_zero hf h x y
```

They are stated in `Mathlib/Analysis/Calculus/MeanValue.lean` and `Mathlib/Analysis/Calculus/LocalExtr/Rolle.lean`.

Monotonicity from the sign of the derivative

```
example : StrictMono (fun x : ℝ ↦ x ^ 3 + x) := by
  apply strictMono_of_deriv_pos
  intro x
  -- ⊢ 0 < deriv (fun x ↦ x ^ 3 + x) x
  have h : HasDerivAt (fun x : ℝ ↦ x ^ 3 + x) (3 * x ^ 2 + 1) x := by
    have h1 : HasDerivAt (fun x : ℝ ↦ x ^ 3) (3 * x ^ 2) x := by simpa using
hasDerivAt_pow 3 x
    exact h1.add (hasDerivAt_id x)
  rw [h.deriv]
  -- ⊢ 0 < 3 * x ^ 2 + 1
  positivity
```

The proof has the three steps of the paper argument: reduce to the sign of the derivative, compute the derivative, check the sign.

A classical inequality proved the same way, on a half-line. The function `t ↦ exp t - t` is nondecreasing on `[0, ∞)`, so its value at `x` is at least its value at `0`.

```
example (x : ℝ) (hx : 0 ≤ x) : 1 + x ≤ Real.exp x := by
  have hmono : MonotoneOn (fun t : ℝ ↦ Real.exp t - t) (Ici 0) := by
    apply monotoneOn_of_deriv_nonneg (convex_Ici 0)
    · fun_prop          -- continuity on [0, ∞)
    · fun_prop          -- differentiability on the interior
    · intro t ht
      rw [interior_Ici] at ht
      -- ht : t ∈ Ioi 0     ⊢ 0 ≤ deriv (fun t ↦ rexp t - t) t
      have h : HasDerivAt (fun t : ℝ ↦ Real.exp t - t) (Real.exp t - 1) t :=
        (Real.hasDerivAt_exp t).sub (hasDerivAt_id t)
      rw [h.deriv]
      have h1 : (1 : ℝ) ≤ Real.exp t := Real.one_le_exp (le_of_lt ht)
      linarith
  have h0 := hmono (mem_Ici.mpr le_rfl) hx hx
  -- h0 : (fun t ↦ rexp t - t) 0 ≤ (fun t ↦ rexp t - t) x
  dsimp only at h0
  rw [Real.exp_zero] at h0
  -- h0 : 1 - 0 ≤ rexp x - x
  linarith
```

`rexp` in the goal display is `Real.exp`. The library has the inequality for all `x` as `Real.add_one_le_exp`.

Integrals over an interval

The integral of `f` from `a` to `b` is written `∫ x in a..b, f x`, typed `\integral`. It is defined for any `a` and `b`, with `∫ x in b..a, f x = -∫ x in a..b, f x`.

```
example : ∫ x in (0:ℝ)..1, x ^ 2 = 1 / 3 := by
  rw [integral_pow]
  -- ⊢ (1 ^ (2 + 1) - 0 ^ (2 + 1)) / (↑2 + 1) = 1 / 3
  norm_num
```

```
example : ∫ x in (0:ℝ)..π, Real.sin x = 2 := by
  rw [integral_sin]
  -- ⊢ cos 0 - cos π = 2
  norm_num [Real.cos_zero, Real.cos_pi]

example (a b : ℝ) : ∫ x in a..b, Real.exp x = Real.exp b - Real.exp a :=
integral_exp
```

Lemma	Statement
`integral_pow n`	`∫ x in a..b, x ^ n = (b ^ (n + 1) - a ^ (n + 1)) / (n + 1)`
`integral_id`	`∫ x in a..b, x = (b ^ 2 - a ^ 2) / 2`
`integral_sin`, `integral_cos`, `integral_exp`	The usual primitives
`integral_one_div_of_pos`	`∫ x in a..b, 1 / x = log (b / a)` for `0 < a`, `0 < b`
`intervalIntegral.integral_const`	`∫ x in a..b, c = (b - a) • c`
`intervalIntegral.integral_add`, `integral_sub`	Linearity, given integrability
`intervalIntegral.integral_const_mul`	A constant comes out
`intervalIntegral.integral_symm`	Reversing the bounds changes the sign
`intervalIntegral.integral_add_adjacent_intervals`	Chasles' relation
`intervalIntegral.integral_mono_on`	Monotonicity
`intervalIntegral.integral_nonneg`	A nonnegative function has a nonnegative integral
`intervalIntegral.integral_comp_mul_deriv`	Change of variables
`intervalIntegral.integral_mul_deriv_eq_deriv_mul`	Integration by parts

The explicit formulas are in `Mathlib/Analysis/SpecialFunctions/Integrals/Basic.lean`.

Linearity needs integrability, because the integral of a non-integrable function is the junk value `0`. The hypothesis is `IntervalIntegrable f volume a b`, and a continuous function satisfies it:

```
example : ∫ x in (0:ℝ)..2, (3 * x ^ 2 + 1) = 10 := by
  have h1 : IntervalIntegrable (fun x : ℝ ↦ 3 * x ^ 2) MeasureTheory.volume
0 2 :=
    (by fun_prop : Continuous (fun x : ℝ ↦ 3 * x ^ 2)).intervalIntegrable 0
2
  have h2 : IntervalIntegrable (fun _ : ℝ ↦ (1 : ℝ)) MeasureTheory.volume 0
2 :=
    continuous_const.intervalIntegrable 0 2
  rw [intervalIntegral.integral_add h1 h2,
intervalIntegral.integral_const_mul, integral_pow,
    intervalIntegral.integral_const]
  -- ⊢ 3 * ((2 ^ (2 + 1) - 0 ^ (2 + 1)) / (↑2 + 1)) + (2 - 0) • 1 = 10
  norm_num
```

`volume` is the Lebesgue measure, the subject of Chapter 19.

The fundamental theorem of calculus

Both halves are in `Mathlib/MeasureTheory/Integral/IntervalIntegral/FundThmCalculus.lean`.

```
-- The integral of a derivative.
example (f f' : ℝ → ℝ) (a b : ℝ) (hf : ∀ x, HasDerivAt f (f' x) x) (hf' :
Continuous f') :
    ∫ x in a..b, f' x = f b - f a :=
  intervalIntegral.integral_eq_sub_of_hasDerivAt (fun x _ ↦ hf x)
(hf'.intervalIntegrable a b)

-- The derivative of an integral.
example (f : ℝ → ℝ) (hf : Continuous f) (a x : ℝ) :
    deriv (fun u ↦ ∫ t in a..u, f t) x = f x :=
  hf.deriv_integral f a x
```

To compute an integral that is not in the table, find a primitive, prove its derivative with the rules above, and apply the theorem:

```
example : ∫ x in (0:ℝ)..1, 2 * x * Real.exp (x ^ 2) = Real.exp 1 - 1 := by
  have hderiv : ∀ x : ℝ,
      HasDerivAt (fun x : ℝ ↦ Real.exp (x ^ 2)) (2 * x * Real.exp (x ^ 2)) x
:= by
    intro x
    have h1 : HasDerivAt (fun x : ℝ ↦ x ^ 2) (2 * x) x := by simpa using
hasDerivAt_pow 2 x
    have h2 := h1.exp
    -- h2 : HasDerivAt (fun x ↦ rexp (x ^ 2)) (rexp (x ^ 2) * (2 * x)) x
    convert h2 using 1
    ring
  have hcont : Continuous (fun x : ℝ ↦ 2 * x * Real.exp (x ^ 2)) := by
fun_prop
  rw [intervalIntegral.integral_eq_sub_of_hasDerivAt (fun x _ ↦ hderiv x)
    (hcont.intervalIntegrable 0 1)]
```

```
  -- ⊢ rexp (1 ^ 2) - rexp (0 ^ 2) = rexp 1 - 1
  norm_num
```

The proof has four parts: the derivative of the primitive, the continuity of the integrand, the theorem, and the evaluation at the bounds.

Errors you will meet here

Message or symptom	Cause	Fix
`Type mismatch` between `HasDerivAt f (↑2 * x ^ (2 - 1)) x` and `HasDerivAt f (2 * x) x`	The lemma's derivative is not in your form	`simpa using h`, or `convert h using 1` then `ring`
`simp made no progress` on a `deriv` of a sum or product	The `deriv` rules need differentiability	Prove `HasDerivAt` and use `.deriv`, or `rw [deriv_fun_add (by fun_prop) (by fun_prop)]`
`Ambiguous term integral_add`	`MeasureTheory` and `intervalIntegral` are both open	Write `intervalIntegral.integral_add` in full
`IntervalIntegrable … is not a fun_prop goal`	`fun_prop` does not prove integrability	`hcont.intervalIntegrable a b` from a continuity proof
`HasDerivAt (f + g)` where you expected `fun x ↦ f x + g x`	The lemma is stated for the sum of functions	Nothing: the two are definitionally equal, and `exact` accepts either
A true statement about `deriv f x` cannot be proved	`f` may not be differentiable at `x`, where `deriv` is `0`	Add the differentiability hypothesis

Exercises

```
import Mathlib

open Real Set

-- Exercise 1. Two rules, then add.
example (x : ℝ) : HasDerivAt (fun x : ℝ ↦ x ^ 4 + 2 * x) (4 * x ^ 3 + 2)
x := by
  sorry

-- Exercise 2. The product rule, with no adjustment needed.
example (x : ℝ) : HasDerivAt (fun x : ℝ ↦ Real.exp x * Real.cos x)
    (Real.exp x * Real.cos x + Real.exp x * (-Real.sin x)) x := by
```

```
  sorry

-- Exercise 3. The chain rule through sin.
example (x : ℝ) : HasDerivAt (fun x : ℝ ↦ Real.sin (x ^ 2)) (Real.cos (x ^
2) * (2 * x)) x := by
  sorry

-- Exercise 4.
example : Differentiable ℝ (fun x : ℝ ↦ Real.sin (x ^ 2) + Real.exp (3 * x))
:= by
  sorry

-- Exercise 5. Reduce to the sign of the derivative.
example : StrictMono (fun x : ℝ ↦ x ^ 5 + 2 * x) := by
  sorry

-- Exercise 6.
example (f : ℝ → ℝ) (hf : Differentiable ℝ f) (h : ∀ x, deriv f x = 0) : f 1
= f 0 := by
  sorry

-- Exercise 7.
example : ∫ x in (0:ℝ)..2, x ^ 3 = 4 := by
  sorry

-- Exercise 8. A constant factor, then the integral of the identity.
example : ∫ x in (1:ℝ)..2, (2 * x) = 3 := by
  sorry

-- Exercise 9. Find the primitive, then use the fundamental theorem.
example : ∫ x in (0:ℝ)..1, 3 * x ^ 2 * Real.exp (x ^ 3) = Real.exp 1 - 1 :=
by
  sorry
```

Solutions

```
import Mathlib

open Real Set

-- Exercise 1.
example (x : ℝ) : HasDerivAt (fun x : ℝ ↦ x ^ 4 + 2 * x) (4 * x ^ 3 + 2)
x := by
  have h1 : HasDerivAt (fun x : ℝ ↦ x ^ 4) (4 * x ^ 3) x := by simpa using
hasDerivAt_pow 4 x
  have h2 : HasDerivAt (fun x : ℝ ↦ 2 * x) 2 x := by simpa using
(hasDerivAt_id x).const_mul 2
  exact h1.add h2

-- Exercise 2.
example (x : ℝ) : HasDerivAt (fun x : ℝ ↦ Real.exp x * Real.cos x)
    (Real.exp x * Real.cos x + Real.exp x * (-Real.sin x)) x :=
  (Real.hasDerivAt_exp x).mul (Real.hasDerivAt_cos x)

-- Exercise 3.
example (x : ℝ) : HasDerivAt (fun x : ℝ ↦ Real.sin (x ^ 2)) (Real.cos (x ^
2) * (2 * x)) x := by
  have h1 : HasDerivAt (fun x : ℝ ↦ x ^ 2) (2 * x) x := by simpa using
hasDerivAt_pow 2 x
```

```
  exact h1.sin

-- Exercise 4.
example : Differentiable ℝ (fun x : ℝ ↦ Real.sin (x ^ 2) + Real.exp (3 * x))
:= by
  fun_prop

-- Exercise 5.
example : StrictMono (fun x : ℝ ↦ x ^ 5 + 2 * x) := by
  apply strictMono_of_deriv_pos
  intro x
  have h : HasDerivAt (fun x : ℝ ↦ x ^ 5 + 2 * x) (5 * x ^ 4 + 2) x := by
    have h1 : HasDerivAt (fun x : ℝ ↦ x ^ 5) (5 * x ^ 4) x := by simpa using
hasDerivAt_pow 5 x
    have h2 : HasDerivAt (fun x : ℝ ↦ 2 * x) 2 x := by simpa using
(hasDerivAt_id x).const_mul 2
    exact h1.add h2
  rw [h.deriv]
  -- ⊢ 0 < 5 * x ^ 4 + 2
  positivity

-- Exercise 6.
example (f : ℝ → ℝ) (hf : Differentiable ℝ f) (h : ∀ x, deriv f x = 0) : f 1
= f 0 :=
  is_const_of_deriv_eq_zero hf h 1 0

-- Exercise 7.
example : ∫ x in (0:ℝ)..2, x ^ 3 = 4 := by
  rw [integral_pow]
  norm_num

-- Exercise 8.
example : ∫ x in (1:ℝ)..2, (2 * x) = 3 := by
  rw [intervalIntegral.integral_const_mul, integral_id]
  norm_num

-- Exercise 9. The primitive is exp (x ^ 3).
example : ∫ x in (0:ℝ)..1, 3 * x ^ 2 * Real.exp (x ^ 3) = Real.exp 1 - 1 :=
by
  have hderiv : ∀ x : ℝ,
      HasDerivAt (fun x : ℝ ↦ Real.exp (x ^ 3)) (3 * x ^ 2 * Real.exp (x ^
3)) x := by
    intro x
    have h1 : HasDerivAt (fun x : ℝ ↦ x ^ 3) (3 * x ^ 2) x := by simpa using
hasDerivAt_pow 3 x
    convert h1.exp using 1
    ring
  have hcont : Continuous (fun x : ℝ ↦ 3 * x ^ 2 * Real.exp (x ^ 3)) := by
fun_prop
  rw [intervalIntegral.integral_eq_sub_of_hasDerivAt (fun x _ ↦ hderiv x)
    (hcont.intervalIntegrable 0 1)]
  norm_num
```

Chapter 19: Measure Theory and Integration

Mathlib integrates against an arbitrary measure with the Bochner integral, states "almost everywhere" with the filter `ae μ`, and provides each convergence theorem as one lemma.

The objects

Notion	In Lean	Notes
σ-algebra on `α`	`[MeasurableSpace α]`	A type class. `ℝ` carries its Borel σ-algebra
Measurable set	`MeasurableSet s`	
Measurable function	`Measurable f`	
Measure	`μ : Measure α`	A function from sets to `ℝ≥0∞`
Measure of a set	`μ s : ℝ≥0∞`, or `μ.real s : ℝ`	Defined on every set, as an outer measure
Lebesgue measure	`volume`	The default measure on `ℝ` and `ℝⁿ`
Almost everywhere	`∀ᵐ x ∂μ, p x`	Typed `\allm` and `\partial`
Equal almost everywhere	`f =ᵐ[μ] g`	Typed `=`, then `\^m`

```
import Mathlib

open MeasureTheory Set Filter Topology ENNReal

example : volume (Icc (0 : ℝ) 1) = 1 := by simp

example : volume (Icc (2 : ℝ) 5) = 3 := by
  rw [Real.volume_Icc]
  -- ⊢ ENNReal.ofReal (5 - 2) = 3
  norm_num

example : volume ({3} : Set ℝ) = 0 := by simp
```

A measure takes values in `ℝ≥0∞`, the type of Chapter 14. So `volume (Icc 2 5)` is `ENNReal.ofReal (5 - 2)`, and `norm_num` computes it.

The foundations are in `Mathlib/MeasureTheory/MeasurableSpace/Defs.lean` and `Mathlib/MeasureTheory/Measure/MeasureSpace.lean`. The chapter *Integration and Measure Theory* of Mathematics in Lean is the companion reading.

Measurability

Measurability is proved by closure properties, by the tactic `measurability`, or by `fun_prop` for functions.

```
example : MeasurableSet (Icc (0 : ℝ) 1) := measurableSet_Icc

example (s t : Set ℝ) (hs : MeasurableSet s) (ht : MeasurableSet t) :
MeasurableSet (s ∪ tᶜ) :=
  hs.union ht.compl

example (s : Set ℝ) (hs : IsOpen s) : MeasurableSet s := hs.measurableSet

example (f : ℝ → ℝ) (hf : Continuous f) : Measurable f := hf.measurable

example : Measurable (fun x : ℝ ↦ x ^ 2 + Real.exp x) := by fun_prop

example : MeasurableSet {x : ℝ | 0 < x ∧ x ^ 2 < 2} := by measurability
```

Open sets, closed sets and intervals are measurable. Continuous functions are measurable. In practice measurability goals are side conditions that one of the two tactics discharges.

The integral uses two weaker notions, which you will see in hypotheses: `AEMeasurable f μ`, measurable after modification on a null set, and `AEStronglyMeasurable f μ`, the version suited to functions with values in a normed space. A measurable or continuous real function satisfies both.

Measures

```
section
variable {α : Type*} [MeasurableSpace α] (μ : Measure α)

example : μ ∅ = 0 := measure_empty

example (s t : Set α) (h : s ⊆ t) : μ s ≤ μ t := measure_mono h

example (s t : Set α) : μ (s ∪ t) ≤ μ s + μ t := measure_union_le s t

example (s t : Set α) (hd : Disjoint s t) (ht : MeasurableSet t) : μ (s ∪ t)
= μ s + μ t :=
  measure_union hd ht

example (s : ℕ → Set α) : μ (⋃ n, s n) ≤ ∑' n, μ (s n) := measure_iUnion_le
s
```

Monotonicity and subadditivity hold for all sets. Additivity needs measurability. A short proof from these rules:

```
example (s t : Set α) (hs : μ s = 0) (ht : μ t = 0) : μ (s ∪ t) = 0 := by
  have h := measure_union_le (μ := μ) s t
  rw [hs, ht, add_zero] at h
  -- h : μ (s ∪ t) ≤ 0
  exact nonpos_iff_eq_zero.mp h
```

Classes of measures are type classes: `IsFiniteMeasure μ`, `IsProbabilityMeasure μ`, `SigmaFinite μ`. The restriction of `μ` to a set is `μ.restrict s`, the Dirac mass is `Measure.dirac a`, and the product is `μ.prod ν`.

Almost everywhere

`∀ᵐ x ∂μ, p x` means that the set where `p` fails has measure zero:

```
example (p : α → Prop) : (∀ᵐ x ∂μ, p x) ↔ μ {x | ¬p x} = 0 := ae_iff
```

It is an instance of `∀ᶠ` from Chapter 17, for the filter `ae μ`. Everything about eventual properties applies, and `filter_upwards` is the tool:

```
example (f g : α → ℝ) (hf : ∀ᵐ x ∂μ, 0 ≤ f x) (hg : ∀ᵐ x ∂μ, f x ≤ g x) :
    ∀ᵐ x ∂μ, 0 ≤ g x := by
  filter_upwards [hf, hg] with x h1 h2
  -- h1 : 0 ≤ f x     h2 : f x ≤ g x     ⊢ 0 ≤ g x
  linarith

example (f g : α → ℝ) (h : ∀ x, f x = g x) : f =ᵐ[μ] g :=
Eventually.of_forall h

example (f g h : α → ℝ) (h₁ : f =ᵐ[μ] g) (h₂ : g =ᵐ[μ] h) : f =ᵐ[μ] h :=
h₁.trans h₂
end
```

On paper you write "outside a null set `N₁ ∪ N₂`". In Lean you list the almost-everywhere hypotheses in `filter_upwards` and reason at a point where all of them hold. The null sets are never named.

Three integrals

Notation	Name	Defined for	When undefined
`∫⁻ x, f x ∂μ`	Lower Lebesgue integral, `lintegral`	`f : α → ℝ≥0∞`	Never: the value may be `∞`
`∫ x, f x ∂μ`	Bochner integral, `integral`	`f : α → E`, a Banach space	Equal to `0` when `f` is not integrable
`∫ x in s, f x ∂μ`	Integral over a set	The same	The same
`∫ x in a..b, f x`	Interval integral, Chapter 18	`f : ℝ → E`	The same

The measure is written after `∂`. Leaving it out means `volume`. The integral over a set is the integral for the restricted measure `μ.restrict s`, and the interval integral is the integral over `Ioc a b` when `a ≤ b`:

```
example (f : ℝ → ℝ) (a b : ℝ) (h : a ≤ b) : ∫ x in a..b, f x = ∫ x in Ioc a
b, f x :=
  intervalIntegral.integral_of_le h
```

`∫⁻` is typed `\integral` then `\^-`. It is the integral of measure theory textbooks for nonnegative functions, and the convergence theorems are cleanest for it.

The Bochner integral is what analysis uses. `Integrable f μ` means that `f` is almost everywhere strongly measurable and that `∫⁻ x, ‖f x‖ₑ ∂μ` is finite. The definitions are in `Mathlib/MeasureTheory/Integral/Lebesgue/Basic.lean` and `Mathlib/MeasureTheory/Integral/Bochner/Basic.lean`.

Proving integrability

```
section
variable {α : Type*} [MeasurableSpace α] (μ : Measure α)

-- Sums and constant multiples.
example (f g : α → ℝ) (hf : Integrable f μ) (hg : Integrable g μ) :
    Integrable (fun x ↦ f x + 2 * g x) μ := by
  fun_prop

-- Constants, on a finite measure space.
example [IsFiniteMeasure μ] (c : ℝ) : Integrable (fun _ : α ↦ c) μ :=
integrable_const c

-- Domination by an integrable function.
example (f g : α → ℝ) (hg : Integrable g μ) (hf : AEStronglyMeasurable f μ)
    (h : ∀ x, |f x| ≤ g x) : Integrable f μ :=
  hg.mono' hf (Eventually.of_forall h)

-- A continuous function on a compact interval.
example (f : ℝ → ℝ) (hf : Continuous f) : IntegrableOn f (Icc 0 1) :=
hf.integrableOn_Icc
```

`IntegrableOn f s μ` is integrability for the restricted measure. In the domination example, `|f x|` is accepted where the lemma expects `‖f x‖`, because on `ℝ` the norm is the absolute value by definition.

The rules of the integral

Lemma	Statement	Needs integrability
`integral_add hf hg`	`∫ (f + g) = ∫ f + ∫ g`	Yes
`integral_sub hf hg`	`∫ (f - g) = ∫ f - ∫ g`	Yes
`integral_const_mul c f`	`∫ c * f = c * ∫ f`	No
`integral_neg f`	`∫ (-f) = -∫ f`	No
`integral_mono hf hg h`	`f ≤ g` gives `∫ f ≤ ∫ g`	Yes
`integral_nonneg h`	`0 ≤ f` gives `0 ≤ ∫ f`	No

Lemma	Statement	Needs integrability
`integral_congr_ae h`	`f =ᵐ[μ] g` gives `∫ f = ∫ g`	No
`norm_integral_le_integral_norm f`	`‖∫ f‖ ≤ ∫ ‖f‖`	No
`abs_integral_le_integral_abs`	`|∫ f| ≤ ∫ |f|`	No
`integral_eq_zero_iff_of_nonneg`	For `0 ≤ f` integrable: `∫ f = 0 ↔ f =ᵐ[μ] 0`	Yes
`integral_undef h`	`∫ f = 0` when `f` is not integrable	

The rules that hold without integrability do so because both sides are then `0`. Additivity is the one that fails: if `f` is not integrable on [0, 1], neither is `g = 1 - f`, so `∫ f + ∫ g = 0`, while `f + g = 1` has integral 1.

```
example (f g : α → ℝ) (hf : Integrable f μ) (hg : Integrable g μ) :
    ∫ x, (2 * f x - g x) ∂μ = 2 * ∫ x, f x ∂μ - ∫ x, g x ∂μ := by
  rw [integral_sub (hf.const_mul 2) hg, integral_const_mul]

example (f : α → ℝ) (h : f =ᵐ[μ] 0) : ∫ x, f x ∂μ = 0 := by
  rw [integral_congr_ae h]
  -- ⊢ ∫ (a : α), 0 a ∂μ = 0
  simp
```

The convergence theorems

Theorem	Name
Monotone convergence, for `∫⁻`	`lintegral_iSup`, `lintegral_tendsto_of_tendsto_of_monotone`
Monotone convergence, for `∫`	`integral_tendsto_of_tendsto_of_monotone`
Fatou's lemma	`lintegral_liminf_le`
Dominated convergence	`tendsto_integral_of_dominated_convergence`
Dominated convergence for series	`hasSum_integral_of_dominated_convergence`
Fubini	`integral_prod`, `integral_integral_swap`
Tonelli	`lintegral_prod`
Markov's inequality	`mul_meas_ge_le_lintegral`, `mul_meas_ge_le_integral_of_nonneg`

The dominated convergence theorem reads exactly as in a textbook:

```
#check @tendsto_integral_of_dominated_convergence
-- (bound : α → ℝ),
-- (∀ (n : ℕ), AEStronglyMeasurable (F n) μ) →
-- Integrable bound μ →
-- (∀ (n : ℕ), ∀ᵐ (a : α) ∂μ, ‖F n a‖ ≤ bound a) →
-- (∀ᵐ (a : α) ∂μ, Tendsto (fun n ↦ F n a) atTop (𝓝 (f a))) →
-- Tendsto (fun n ↦ ∫ (a : α), F n a ∂μ) atTop (𝓝 (∫ (a : α), f a ∂μ))
```

Applying it is a matter of supplying the four hypotheses in order. When your hypotheses hold everywhere, `filter_upwards with x` turns each almost-everywhere goal into a pointwise one:

```
example (F : ℕ → α → ℝ) (f g : α → ℝ) (hF : ∀ n, AEStronglyMeasurable (F n)
μ)
    (hg : Integrable g μ) (hbound : ∀ n x, |F n x| ≤ g x)
    (hlim : ∀ x, Tendsto (fun n ↦ F n x) atTop (𝓝 (f x))) :
    Tendsto (fun n ↦ ∫ x, F n x ∂μ) atTop (𝓝 (∫ x, f x ∂μ)) := by
  apply tendsto_integral_of_dominated_convergence g hF hg
  · intro n
    filter_upwards with x
    -- ⊢ ‖F n x‖ ≤ g x
    rw [Real.norm_eq_abs]
    exact hbound n x
  · filter_upwards with x
    exact hlim x
end
```

Not every limit of integrals needs a convergence theorem. When the integrals can be computed, compute them:

```
example : Tendsto (fun n : ℕ ↦ ∫ x in (0:ℝ)..1, x ^ n) atTop (𝓝 0) := by
  have h : ∀ n : ℕ, ∫ x in (0:ℝ)..1, x ^ n = 1 / (n + 1 : ℝ) := by
    intro n
    rw [integral_pow]
    simp
  simp only [h]
  -- ⊢ Tendsto (fun n ↦ 1 / (↑n + 1)) atTop (𝓝 0)
  exact tendsto_one_div_add_atTop_nhds_zero_nat
```

L^p spaces

Notion	In Lean
`f` is in $\mathcal{L}^p$: measurable with finite `p`-norm	`MemLp f p μ`, with `p : ℝ≥0∞`
The `p`-norm, with values in `ℝ≥0∞`	`eLpNorm f p μ`
The space L^p of classes of functions	`Lp E p μ`
$\mathcal{L}^1$ is integrability	`memLp_one_iff_integrable`
Minkowski's inequality	`eLpNorm_add_le`
Hölder's inequality	`integral_mul_norm_le_Lp_mul_Lq`

```
example {α : Type*} [MeasurableSpace α] (μ : Measure α) (f g : α → ℝ)
    (hf : MemLp f 2 μ) (hg : MemLp g 2 μ) : MemLp (f + g) 2 μ :=
  hf.add hg
```

There are two levels. `MemLp f p μ` is a property of an honest function, and is what you use for estimates. `Lp E p μ` is the Banach space of equivalence classes, built as a quotient as in Chapter 11, and is what you use for completeness, duality and Hilbert space arguments. The space is defined in `Mathlib/MeasureTheory/Function/LpSpace/Basic.lean`.

The predicate was renamed from `Mem𝓛p` to `MemLp`, and the norm from `snorm` to `eLpNorm`. Older texts use the old names.

Fourier analysis and probability

Two areas built on this chapter, with their entry points.

Topic	File	Main names
Fourier series	`Mathlib/Analysis/Fourier/AddCircle.lean`	`fourier n`, `fourierCoeff f n`, `hasSum_fourier_series_L2`, `tsum_sq_fourierCoeff` for Parseval
Fourier transform	`Mathlib/Analysis/Fourier/FourierTransform.lean`	Notation $\mathscr{F}$ `f`, typed `\MCF`
Riemann–Lebesgue lemma	`Mathlib/Analysis/Fourier/RiemannLebesgueLemma.lean`	`Real.zero_at_infty_fourier`
Gaussian integral	`Mathlib/Analysis/SpecialFunctions/Gaussian/GaussianIntegral.lean`	`integral_gaussian`
Probability	`Mathlib/Probability`	Independence, conditional expectation, martingales, the strong law of large numbers, the central limit theorem

A probability space is a measure space with `[IsProbabilityMeasure μ]`. A random variable is a measurable function, and its expectation is its integral.

Errors you will meet here

Message or symptom	Cause	Fix
`failed to synthesize MeasurableSpace α`	The type has no σ-algebra	Add `[MeasurableSpace α]`
A goal `Integrable …` appears after `rw [integral_add]`	Linearity needs integrability	Supply `integral_add hf hg` with the two proofs
`Ambiguous term integral_add`	Two namespaces are open	Write `MeasureTheory.integral_add` or `intervalIntegral.integral_add`
`linarith` or `ring` fails on a goal about `μ s`	Measures live in `ℝ≥0∞`	Use the lemmas of `ENNReal`, or move to `μ.real s` with finiteness
A goal `∀ᵐ x ∂μ, …` when you have a pointwise fact	The lemma is stated almost everywhere	`filter_upwards with x`, or `Eventually.of_forall`
`‖f x‖` in a goal where you have `|f x|`	Lemmas are stated with norms	`rw [Real.norm_eq_abs]`
Unknown `Memℒp` or `snorm`	Renamed	`MemLp`, `eLpNorm`

Exercises

```
import Mathlib

open MeasureTheory Set Filter Topology ENNReal

variable {α : Type*} [MeasurableSpace α] (μ : Measure α)

-- Exercise 1.
example : volume (Ioo (1 : ℝ) 4) = 3 := by
  sorry

-- Exercise 2.
example : MeasurableSet (Icc (0 : ℝ) 1 ∪ {5}) := by
  sorry

-- Exercise 3.
example (s t : Set α) : μ (s ∩ t) ≤ μ s := by
  sorry

-- Exercise 4. Almost-everywhere inequalities are transitive.
example (f g h : α → ℝ) (h₁ : f ≤ᵐ[μ] g) (h₂ : g ≤ᵐ[μ] h) : f ≤ᵐ[μ] h := by
  sorry
```

```
-- Exercise 5. Linearity, with the integrability it needs.
example (f g : α → ℝ) (hf : Integrable f μ) (hg : Integrable g μ) :
    ∫ x, (f x + 3 * g x) ∂μ = ∫ x, f x ∂μ + 3 * ∫ x, g x ∂μ := by
  sorry

-- Exercise 6. A nonnegative function with zero integral vanishes almost
everywhere.
example (f : α → ℝ) (hf : Integrable f μ) (h0 : ∀ x, 0 ≤ f x) (h : ∫ x, f x
∂μ = 0) :
    f =ᵐ[μ] 0 := by
  sorry

-- Exercise 7. A bounded measurable function is integrable on a finite
measure space.
example [IsFiniteMeasure μ] (f : α → ℝ) (hf : AEStronglyMeasurable f μ) (C :
ℝ)
    (h : ∀ x, |f x| ≤ C) : Integrable f μ := by
  sorry

-- Exercise 8. Compute the integrals, then take the limit.
example : Tendsto (fun n : ℕ ↦ ∫ x in (0:ℝ)..1, x ^ n) atTop (𝓝 0) := by
  sorry
```

Solutions

```
import Mathlib

open MeasureTheory Set Filter Topology ENNReal

variable {α : Type*} [MeasurableSpace α] (μ : Measure α)

-- Exercise 1.
example : volume (Ioo (1 : ℝ) 4) = 3 := by
  rw [Real.volume_Ioo]
  norm_num

-- Exercise 2.
example : MeasurableSet (Icc (0 : ℝ) 1 ∪ {5}) :=
  measurableSet_Icc.union (measurableSet_singleton 5)

-- Exercise 3.
example (s t : Set α) : μ (s ∩ t) ≤ μ s := measure_mono inter_subset_left

-- Exercise 4.
example (f g h : α → ℝ) (h₁ : f ≤ᵐ[μ] g) (h₂ : g ≤ᵐ[μ] h) : f ≤ᵐ[μ] h := by
  filter_upwards [h₁, h₂] with x hx₁ hx₂
  -- hx₁ : f x ≤ g x     hx₂ : g x ≤ h x     ⊢ f x ≤ h x
  exact le_trans hx₁ hx₂

-- Exercise 5.
example (f g : α → ℝ) (hf : Integrable f μ) (hg : Integrable g μ) :
    ∫ x, (f x + 3 * g x) ∂μ = ∫ x, f x ∂μ + 3 * ∫ x, g x ∂μ := by
  rw [integral_add hf (hg.const_mul 3), integral_const_mul]

-- Exercise 6.
example (f : α → ℝ) (hf : Integrable f μ) (h0 : ∀ x, 0 ≤ f x) (h : ∫ x, f x
∂μ = 0) :
    f =ᵐ[μ] 0 :=
  (integral_eq_zero_iff_of_nonneg h0 hf).mp h
```

```
-- Exercise 7.
example [IsFiniteMeasure μ] (f : α → ℝ) (hf : AEStronglyMeasurable f μ) (C :
ℝ)
    (h : ∀ x, |f x| ≤ C) : Integrable f μ := by
  apply (integrable_const C).mono' hf
  filter_upwards with x
  rw [Real.norm_eq_abs]
  exact h x

-- Exercise 8.
example : Tendsto (fun n : ℕ ↦ ∫ x in (0:ℝ)..1, x ^ n) atTop (𝓝 0) := by
  have h : ∀ n : ℕ, ∫ x in (0:ℝ)..1, x ^ n = 1 / (n + 1 : ℝ) := by
    intro n
    rw [integral_pow]
    simp
  simp only [h]
  exact tendsto_one_div_add_atTop_nhds_zero_nat
```

Exercise 2 is also closed by the tactic `measurability`.

Chapter 20: A Worked Project in Analysis

This chapter formalizes one theorem from a blank file, in the order the work is really done: paper proof, statement, tests of the statement, skeleton with `sorry`, filling, checking, and comparison with Mathlib.

The theorem: **a uniform limit of continuous functions is continuous.**

Step 0: the paper proof

Write the proof you intend to formalize, with every quantity named.

Let `Fₙ → f` uniformly on `ℝ`, each `Fₙ` continuous. Fix `a` and `ε > 0`.

1. By uniform convergence, choose `N` with `|F_N(x) - f(x)| < ε/3` for all `x`.
2. By continuity of `F_N` at `a`, choose `δ > 0` with `|F_N(x) - F_N(a)| < ε/3` whenever `|x - a| < δ`.
3. For `|x - a| < δ`: `|f(x) - f(a)| ≤ |f(x) - F_N(x)| + |F_N(x) - F_N(a)| + |F_N(a) - f(a)| < ε`.

Three choices, one inequality. The Lean proof will have exactly this shape. If the paper proof is vague, the formalization will stall at the vague place.

Step 1: the statement

Mathlib has uniform convergence, in a general form. For a first project, define the notion yourself, in the form the paper proof uses.

```
import Mathlib

open Filter Topology

/-- `F n` converges uniformly to `f` on `ℝ`. -/
def UnifConv (F : ℕ → ℝ → ℝ) (f : ℝ → ℝ) : Prop :=
  ∀ ε > 0, ∃ N, ∀ n ≥ N, ∀ x, |F n x - f x| < ε
```

A sequence of functions is a function of two arguments, `F : ℕ → ℝ → ℝ`, so `F n` is the `n`-th function and `F n x` its value at `x`.

The order of the quantifiers is the whole content of the definition. `∃ N` comes before `∀ x`: one `N` serves every `x`.

Step 2: test the definition

A wrong definition makes every later theorem worthless, and Lean will not notice. Before proving anything about `UnifConv`, check that it holds where it should.

```
-- A constant sequence converges uniformly to its value.
example (f : ℝ → ℝ) : UnifConv (fun _ ↦ f) f := by
  intro ε hε
  use 0
```

```
  intro n _ x
  simpa using hε

-- x + 1/(n+1) converges uniformly to x.
example : UnifConv (fun n x ↦ x + 1 / (n + 1 : ℝ)) (fun x ↦ x) := by
  intro ε hε
  obtain ⟨N, hN⟩ := exists_nat_one_div_lt hε
  use N
  intro n hn x
  have h1 : (0 : ℝ) < 1 / (n + 1) := by positivity
  have h2 : (1 : ℝ) / (n + 1) ≤ 1 / (N + 1) := by gcongr
  have h3 : x + 1 / (n + 1 : ℝ) - x = 1 / (n + 1) := by ring
  rw [h3, abs_of_pos h1]
  linarith
```

Two examples that should be true are true. A test that should fail is just as useful: `fun n x ↦ x / (n + 1)` tends to `0` at every `x` but not uniformly, so it must not satisfy `UnifConv`. With the quantifiers in the wrong order, it would.

Step 3: the skeleton

Write the structure of the proof with `sorry` at every leaf. The goal of this step is a file that compiles, with warnings and no errors.

```
theorem continuous_of_unifConv_skeleton (F : ℕ → ℝ → ℝ) (f : ℝ → ℝ)
    (hF : ∀ n, Continuous (F n)) (hunif : UnifConv F f) : Continuous f := by
  rw [Metric.continuous_iff]
  intro a ε hε
  -- ⊢ ∃ δ > 0, ∀ (a_1 : ℝ), dist a_1 a < δ → dist (f a_1) (f a) < ε
  obtain ⟨N, hN⟩ := hunif (ε / 3) (by positivity)
-- choice 1
  obtain ⟨δ, hδ, hδ'⟩ := Metric.continuous_iff.mp (hF N) a (ε / 3) (by
positivity)  -- choice 2
  refine ⟨δ, hδ, ?_⟩
  intro x hx
  rw [Real.dist_eq]
  have h1 : |f x - F N x| < ε / 3 := sorry
  have h2 : |F N x - F N a| < ε / 3 := sorry
  have h3 : |F N a - f a| < ε / 3 := sorry
  sorry
```

Read the state after the two `obtain` lines. It contains everything the paper proof chose:

```
hN : ∀ n ≥ N, ∀ (x : ℝ), |F n x - f x| < ε / 3
δ : ℝ
hδ : δ > 0
hδ' : ∀ (a_1 : ℝ), dist a_1 a < δ → dist (F N a_1) (F N a) < ε / 3
```

The skeleton already proves something: the three estimates suffice, provided they can be combined. Each `sorry` is now a small, independent problem. This is the main technique for long proofs, and Chapter 25 returns to it.

Step 4: isolate the algebra

The last `sorry` needs a triangle inequality with three terms. It does not depend on functions or on ε, so it becomes a lemma of its own.

```
theorem abs_sub_three (a b c d : ℝ) : |a - d| ≤ |a - b| + |b - c| + |c -
d| := by
  calc |a - d| = |(a - b) + (b - c) + (c - d)| := by ring_nf
    _ ≤ |(a - b) + (b - c)| + |c - d| := abs_add_le _ _
    _ ≤ |a - b| + |b - c| + |c - d| := by gcongr; exact abs_add_le _ _
```

In the last line, `gcongr` reduces the goal to `|(a - b) + (b - c)| ≤ |a - b| + |b - c|`, which is the triangle inequality again.

Before writing such a lemma, look for it, for instance in Loogle with the pattern `|- |_ - _| ≤ |_ - _| + |_ - _| + |_ - _|`. For distances the library has `dist_triangle4`. When nothing turns up quickly, a helper lemma of three lines is cheaper than ten minutes of search.

Step 5: fill the gaps

```
theorem continuous_of_unifConv (F : ℕ → ℝ → ℝ) (f : ℝ → ℝ)
    (hF : ∀ n, Continuous (F n)) (hunif : UnifConv F f) : Continuous f := by
  rw [Metric.continuous_iff]
  intro a ε hε
  obtain ⟨N, hN⟩ := hunif (ε / 3) (by positivity)
  obtain ⟨δ, hδ, hδ'⟩ := Metric.continuous_iff.mp (hF N) a (ε / 3) (by
positivity)
  refine ⟨δ, hδ, ?_⟩
  intro x hx
  rw [Real.dist_eq]
  have h1 : |f x - F N x| < ε / 3 := by
    rw [abs_sub_comm]
    exact hN N le_rfl x
  have h2 : |F N x - F N a| < ε / 3 := by
    have h := hδ' x hx
    rwa [Real.dist_eq] at h
  have h3 : |F N a - f a| < ε / 3 := hN N le_rfl a
  calc |f x - f a|
      ≤ |f x - F N x| + |F N x - F N a| + |F N a - f a| := abs_sub_three _ _
_ _
    _ < ε / 3 + ε / 3 + ε / 3 := by gcongr
    _ = ε := by ring
```

The three small proofs each took one obstacle.

- `h1`: the hypothesis has `|F N x - f x|` and the goal has `|f x - F N x|`. `abs_sub_comm` swaps them. Then `hN N le_rfl x` applies `hN` to `n = N`, with `le_rfl : N ≤ N`.
- `h2`: `hδ'` speaks of `dist`, the goal of absolute values. `rwa [Real.dist_eq] at h` rewrites and closes the goal with the result.
- `h3`: exactly `hN` at `n = N` and `x = a`.

The final `calc` is the inequality of the paper proof, line for line.

Step 6: check the result

```
#print axioms continuous_of_unifConv
-- 'continuous_of_unifConv' depends on axioms: [propext, Classical.choice,
Quot.sound]
```

No `sorryAx`: the proof is complete. Then read the statement once more, alone, as a stranger would. Does `continuous_of_unifConv` say what the title of this chapter says? The hypotheses are continuity of each `F n` and uniform convergence, and the conclusion is continuity of `f`. Yes.

A finished project also has no warnings. In a project created with the `math` template, the linters will have pointed out unused variables, long lines and non-terminal `simp` calls along the way.

Step 7: compare with Mathlib

Mathlib's notion is `TendstoUniformly F f p`, for functions into a uniform space and convergence along any filter `p`. The theorem is one line:

```
example (F : ℕ → ℝ → ℝ) (f : ℝ → ℝ) (hF : ∀ n, Continuous (F n))
    (hunif : TendstoUniformly F f atTop) : Continuous f :=
  hunif.continuous (Frequently.of_forall hF)
```

The library version asks only that `F n` be continuous for infinitely many `n`, written `∃ᶠ n in atTop`. That is weaker than our hypothesis, and `Frequently.of_forall` bridges the two.

Our definition and Mathlib's are equivalent. Proving it connects the project to the library, and from then on every Mathlib theorem about uniform convergence applies to `UnifConv`.

```
theorem unifConv_iff (F : ℕ → ℝ → ℝ) (f : ℝ → ℝ) :
    UnifConv F f ↔ TendstoUniformly F f atTop := by
  rw [Metric.tendstoUniformly_iff]
  -- ⊢ UnifConv F f ↔ ∀ ε > 0, ∀ᶠ (n : ℕ) in atTop, ∀ (x : ℝ), dist (f x) (F
n x) < ε
  constructor
  · intro h ε hε
    obtain ⟨N, hN⟩ := h ε hε
    rw [eventually_atTop]
    refine ⟨N, fun n hn x ↦ ?_⟩
    rw [Real.dist_eq, abs_sub_comm]
    exact hN n hn x
  · intro h ε hε
    obtain ⟨N, hN⟩ := eventually_atTop.mp (h ε hε)
    refine ⟨N, fun n hn x ↦ ?_⟩
    have h' := hN n hn x
    rwa [Real.dist_eq, abs_sub_comm] at h'

theorem continuous_of_unifConv' (F : ℕ → ℝ → ℝ) (f : ℝ → ℝ)
    (hF : ∀ n, Continuous (F n)) (hunif : UnifConv F f) : Continuous f :=
  ((unifConv_iff F f).mp hunif).continuous (Frequently.of_forall hF)
```

In real work this is the right order of operations: look for Mathlib's notion first, and write your own definition only when there is none, or with a bridge lemma like `unifConv_iff`.

What the project teaches

1. **The paper proof comes first.** Name every object. The formal proof inherits its structure.
2. **Test definitions** on examples that should hold, and think of one that should not.
3. **Write the skeleton before the details.** `sorry` lets you verify the architecture early.
4. **Extract lemmas** that do not depend on the context. They are easier to prove and reusable.
5. **Make hypotheses match goals by small rewrites**: `abs_sub_comm`, `Real.dist_eq`, `rwa … at`.
6. **Finish with `#print axioms`** and a reading of the bare statement.
7. **Connect to the library.** A definition of your own needs a bridge to Mathlib's.

Exercises

```
import Mathlib

open Filter Topology

def UnifConv (F : ℕ → ℝ → ℝ) (f : ℝ → ℝ) : Prop :=
  ∀ ε > 0, ∃ N, ∀ n ≥ N, ∀ x, |F n x - f x| < ε

-- Exercise 1. A uniform limit of bounded functions is bounded.
-- Paper proof: take ε = 1, get N, and let M bound F N. Then M + 1 bounds f.
theorem bounded_of_unifConv (F : ℕ → ℝ → ℝ) (f : ℝ → ℝ)
    (hF : ∀ n, ∃ M, ∀ x, |F n x| ≤ M) (hunif : UnifConv F f) : ∃ M, ∀ x, |f
x| ≤ M := by
  sorry

-- Exercise 2. The uniform limit is unique.
-- Start with `ext x` and `apply eq_of_abs_sub_le_all`.
theorem unifConv_unique (F : ℕ → ℝ → ℝ) (f g : ℝ → ℝ)
    (hf : UnifConv F f) (hg : UnifConv F g) : f = g := by
  sorry

-- Exercise 3. The main theorem between metric spaces. Rewrite the proof of
this
-- chapter with `dist` in place of absolute values, and `dist_triangle4`.
theorem continuous_of_unifConv_metric {X Y : Type*} [MetricSpace X]
[MetricSpace Y]
    (F : ℕ → X → Y) (f : X → Y) (hF : ∀ n, Continuous (F n))
    (hunif : ∀ ε > 0, ∃ N, ∀ n ≥ N, ∀ x, dist (F n x) (f x) < ε) :
Continuous f := by
  sorry
```

Solutions

```
import Mathlib

open Filter Topology
```

```
def UnifConv (F : ℕ → ℝ → ℝ) (f : ℝ → ℝ) : Prop :=
  ∀ ε > 0, ∃ N, ∀ n ≥ N, ∀ x, |F n x - f x| < ε

-- Exercise 1.
theorem bounded_of_unifConv (F : ℕ → ℝ → ℝ) (f : ℝ → ℝ)
    (hF : ∀ n, ∃ M, ∀ x, |F n x| ≤ M) (hunif : UnifConv F f) : ∃ M, ∀ x, |f
x| ≤ M := by
  obtain ⟨N, hN⟩ := hunif 1 (by norm_num)
  obtain ⟨M, hM⟩ := hF N
  refine ⟨M + 1, fun x ↦ ?_⟩
  have h1 := hN N le_rfl x
  have h2 := hM x
  have h3 : |f x| ≤ |f x - F N x| + |F N x| := by
    calc |f x| = |(f x - F N x) + F N x| := by ring_nf
      _ ≤ |f x - F N x| + |F N x| := abs_add_le _ _
  rw [abs_sub_comm] at h1
  -- h1 : |f x - F N x| < 1     h2 : |F N x| ≤ M     ⊢ |f x| ≤ M + 1
  linarith

-- Exercise 2.
theorem unifConv_unique (F : ℕ → ℝ → ℝ) (f g : ℝ → ℝ)
    (hf : UnifConv F f) (hg : UnifConv F g) : f = g := by
  ext x
  apply eq_of_abs_sub_le_all
  intro ε hε
  -- ⊢ |f x - g x| ≤ ε
  obtain ⟨N₁, h₁⟩ := hf (ε / 2) (by positivity)
  obtain ⟨N₂, h₂⟩ := hg (ε / 2) (by positivity)
  have e₁ := h₁ (max N₁ N₂) (le_max_left _ _) x
  have e₂ := h₂ (max N₁ N₂) (le_max_right _ _) x
  have h := abs_sub_le (f x) (F (max N₁ N₂) x) (g x)
  rw [abs_sub_comm] at e₁
  linarith

-- Exercise 3.
theorem continuous_of_unifConv_metric {X Y : Type*} [MetricSpace X]
[MetricSpace Y]
    (F : ℕ → X → Y) (f : X → Y) (hF : ∀ n, Continuous (F n))
    (hunif : ∀ ε > 0, ∃ N, ∀ n ≥ N, ∀ x, dist (F n x) (f x) < ε) :
Continuous f := by
  rw [Metric.continuous_iff]
  intro a ε hε
  obtain ⟨N, hN⟩ := hunif (ε / 3) (by positivity)
  obtain ⟨δ, hδ, hδ'⟩ := Metric.continuous_iff.mp (hF N) a (ε / 3) (by
positivity)
  refine ⟨δ, hδ, fun x hx ↦ ?_⟩
  have h1 : dist (f x) (F N x) < ε / 3 := by
    rw [dist_comm]
    exact hN N le_rfl x
  have h2 : dist (F N x) (F N a) < ε / 3 := hδ' x hx
  have h3 : dist (F N a) (f a) < ε / 3 := hN N le_rfl a
  calc dist (f x) (f a)
      ≤ dist (f x) (F N x) + dist (F N x) (F N a) + dist (F N a) (f a) :=
dist_triangle4 _ _ _ _
    _ < ε / 3 + ε / 3 + ε / 3 := by gcongr
    _ = ε := by ring
```

The metric version is shorter than the real one. No conversion between `dist` and absolute values is needed, and the three-term inequality is already in the library. Generality often simplifies a formal proof.

Part IV — Lean as a Programming Language

Chapter 21: Functional Programming and Effects

Lean is a compiled, general-purpose functional language: pure functions for computation, `do` blocks for loops and input-output, and Lake to build an executable.

You can use Lean for mathematics without this chapter. It matters when you want to compute examples, test conjectures numerically, or write tactics, which are Lean programs.

Nothing here needs Mathlib. The code runs in a file with no import, except where `Std` is imported.

Data you get for free

```
#eval "Lean" ++ " " ++ "4"                  -- "Lean 4"
#eval s!"2 + 2 = {2 + 2}"                   -- "2 + 2 = 4"
#eval "a,b,c".splitOn ","                   -- ["a", "b", "c"]
#eval "42".toNat?                           -- some 42

#eval [1, 2, 3] ++ [4]                      -- [1, 2, 3, 4]
#eval 0 :: [1, 2]                           -- [0, 1, 2]
#eval #[1, 2, 3].push 4                     -- #[1, 2, 3, 4]
#eval #[10, 20, 30][1]!                     -- 20
#eval #[10, 20, 30][5]?                     -- none

#eval (3, "three").1                        -- 3
#eval Float.sqrt 2                          -- 1.414214
#eval (10 : Nat).toFloat / 3                -- 3.333333
```

Type	Literal	What it is
`String`	`"abc"`, `s!"x = {x}"`	Text. The `s!` form interpolates expressions in braces
`List α`	`[1, 2, 3]`	A linked list. Good for pattern matching
`Array α`	`#[1, 2, 3]`	A contiguous array. Good for indexing and loops
`Option α`	`some x`, `none`	A value that may be absent
`α × β`	`(a, b)`	A pair, with components `.1` and `.2`
`Nat`, `Int`	`42`, `-3`	Arbitrary precision
`Float`	`2.5`	64-bit floating point
`Bool`	`true`, `false`	A computed truth value

Indexing comes in three forms. `xs[i]!` panics when out of bounds, `xs[i]?` returns an `Option`, and `xs[i]` demands a proof that `i` is in bounds.

`Bool` and `Prop`

```
#check (5 == 5)        -- 5 == 5 : Bool
#check (5 = 5)         -- 5 = 5 : Prop
```

`==` computes a `Bool`. `=` states a proposition. An `if` accepts either, provided the proposition is decidable, as in Chapter 11. In programs, `==`, `!=`, `&&`, `||` and `!` are the usual operators.

Lists, the functional way

Most loops over a list are one of three functions: `map` transforms each element, `filter` keeps some, `foldl` accumulates.

```
#eval List.range 5                                  -- [0, 1, 2, 3, 4]
#eval (List.range 5).map (· ^ 2)                    -- [0, 1, 4, 9, 16]
#eval (List.range 10).filter (· % 2 == 0)           -- [0, 2, 4, 6, 8]
#eval (List.range 5).foldl (· + ·) 0                -- 10
#eval [1, 2, 3].sum                                 -- 6
#eval [1, 2, 3].zip ["a", "b", "c"]                 -- [(1, "a"), (2, "b"), (3,
"c")]
#eval [1, 2, 3] |>.map (· * 2) |>.filter (· > 2) -- [4, 6]
```

`(· ^ 2)` is short for `fun x => x ^ 2`. The pipeline `|>.` chains method calls from left to right.

Defining functions

Pattern matching, conditionals and local definitions:

```
def fact : Nat → Nat
  | 0 => 1
  | n + 1 => (n + 1) * fact n

#eval fact 20          -- 2432902008176640000

def describe (n : Nat) : String :=
  if n == 0 then "zero"
  else if n % 2 == 0 then "even"
  else "odd"

def hypot (a b : Float) : Float :=
  let a2 := a * a
  let b2 := b * b
  Float.sqrt (a2 + b2)

#eval hypot 3 4        -- 5.000000

def mean (xs : List Float) : Float :=
  total / n
where
  total := xs.foldl (· + ·) 0
```

```
  n := xs.length.toFloat

#eval mean [1, 2, 3, 4]    -- 2.500000
```

Your own data types work as in Chapters 9 and 10. Add `deriving Repr` so that `#eval` can print them:

```
inductive Shape where
  | circle (r : Float)
  | rect (w h : Float)
deriving Repr

def Shape.area : Shape → Float
  | .circle r => 3.14159 * r * r
  | .rect w h => w * h

#eval (Shape.rect 2 3).area    -- 6.000000
#eval Shape.circle 1           -- Shape.circle 1.000000
```

Values that may be missing

A function that can fail returns an `Option`. The caller must handle both cases, by `match` or with a default.

```
def safeDiv (a b : Nat) : Option Nat :=
  if b == 0 then none else some (a / b)

#eval safeDiv 10 2               -- some 5
#eval safeDiv 10 0               -- none
#eval (safeDiv 10 0).getD 0      -- 0
```

Chaining several fallible steps by `match` gets heavy. A `do` block does it: each `←` extracts a value, and the whole block is `none` as soon as one step is.

```
def addDivs (a b c : Nat) : Option Nat := do
  let x ← safeDiv a c
  let y ← safeDiv b c
  return x + y

#eval addDivs 10 20 5            -- some 6
#eval addDivs 10 20 0            -- none
```

`Except ε α` is the same idea with an error message:

```
def parseAge (s : String) : Except String Nat :=
  match s.toNat? with
  | none => .error s!"not a number: {s}"
  | some n => if n > 150 then .error s!"too old: {n}" else .ok n

#eval parseAge "42"              -- Except.ok 42
#eval parseAge "abc"             -- Except.error "not a number: abc"
```

Loops and mutable variables

Inside a `do` block you can write imperative code: `let mut` declares a variable that can be reassigned, and `for`, `while`, `break`, `continue` and early `return` work as in other languages.

For a pure function, wrap the block in `Id.run`:

```
def sumTo (n : Nat) : Nat := Id.run do
  let mut total := 0
  for i in [0:n+1] do
    total := total + i
  return total

#eval sumTo 100                  -- 5050

def fib (n : Nat) : Nat := Id.run do
  let mut a := 0
  let mut b := 1
  for _ in [0:n] do
    (a, b) := (b, a + b)
  return a

#eval fib 90                     -- 2880067194370816120
```

`[0:n]` is the range `0, 1, …, n - 1`. The function is still pure: `sumTo 100` always returns `5050`, and the mutation is invisible from outside. Lean compiles the block to an ordinary recursive function.

Arrays make this style efficient. When an array has a single owner, `push` and `set!` modify it in place.

```
def primesUpTo (n : Nat) : Array Nat := Id.run do
  let mut isPrime := Array.replicate (n + 1) true
  let mut primes := #[]
  for i in [2:n+1] do
    if isPrime[i]! then
      primes := primes.push i
      let mut j := i * i
      while j ≤ n do
        isPrime := isPrime.set! j false
        j := j + i
  return primes

#eval primesUpTo 50              -- #[2, 3, 5, 7, 11, 13, 17, 19, 23, 29,
31, 37, 41, 43, 47]
#eval (primesUpTo 100000).size   -- 9592
```

For key-value data, the standard library has hash maps:

```
import Std.Data.HashMap
open Std

def wordCount (text : String) : HashMap String Nat := Id.run do
  let mut counts : HashMap String Nat := {}
  for w in text.splitOn " " do
    if w != "" then
      counts := counts.insert w (counts.getD w 0 + 1)
  return counts

#eval (wordCount "the cat and the hat and the bat")["the"]?     -- some 3
```

The `import` line must be the first line of the file.

Recursion that Lean cannot check

Every `def` must terminate, as Chapter 9 explained. Lean proves it automatically for structural recursion and for simple arithmetic decrease. Otherwise it reports the inequality it could not prove:

```
def bad (n : Nat) : Nat :=
  if n = 0 then 0 else bad (n / 2 + 1)
-- error: failed to prove termination …
--   h✝ : ¬n = 0
--   ⊢ n / 2 + 1 < n
```

The goal is false for `n = 1`, and indeed `bad 1` loops forever. When the recursion does terminate, say on what, and prove the decrease:

```
def gcd1 (a b : Nat) : Nat :=
  if b = 0 then a else gcd1 b (a % b)
termination_by b
decreasing_by
  apply Nat.mod_lt
  omega

#eval gcd1 48 18                  -- 6
```

For a program whose termination you cannot or need not prove, `partial def` skips the check. The function runs normally and cannot be reasoned about in proofs.

```
partial def collatzSteps (n : Nat) (steps : Nat := 0) : Nat :=
  if n ≤ 1 then steps
  else if n % 2 == 0 then collatzSteps (n / 2) (steps + 1)
  else collatzSteps (3 * n + 1) (steps + 1)

#eval collatzSteps 27             -- 111
```

`(steps : Nat := 0)` is an argument with a default value.

Input and output

A value of type `IO α` is a program that may interact with the world and returns an `α`. `IO Unit` returns nothing of interest. Actions are sequenced in a `do` block.

```
def greet (name : String) : IO Unit := do
  IO.println s!"Hello, {name}!"

#eval greet "Lean"                -- Hello, Lean!

def countdown : IO Unit := do
  for i in [0:3] do
    IO.println s!"{3 - i}..."
  IO.println "Go!"
```

Action	Type	Does
`IO.println s`	`IO Unit`	Prints a line
`IO.eprintln s`	`IO Unit`	Prints to standard error

Action	Type	Does
`IO.FS.readFile path`	`IO String`	Reads a whole file
`IO.FS.writeFile path s`	`IO Unit`	Writes a file
`(← IO.getStdin).getLine`	`IO String`	Reads a line from the keyboard
`IO.rand lo hi`	`BaseIO Nat`, usable in `IO`	A random number in a range
`IO.monoMsNow`	`BaseIO Nat`, usable in `IO`	A clock, in milliseconds

The separation is strict. A function whose type has no `IO` cannot print, read or use randomness. This is what makes pure functions safe to reason about, and it is why a `theorem` can mention `fib` and not `greet`.

A complete program

The entry point of an executable is `main`. It may take the command-line arguments and return an exit code.

```
def main (args : List String) : IO UInt32 := do
  match args with
  | [] =>
    IO.eprintln "usage: stats NUMBER..."
    return 1
  | _ =>
    let nums := args.filterMap String.toNat?
    let total := nums.sum
    IO.println s!"count = {nums.length}"
    IO.println s!"sum   = {total}"
    IO.println s!"mean  = {total.toFloat / nums.length.toFloat}"
    return 0
```

To build it, create a project with the default template, which contains a library and a `Main.lean`:

```
lake new stats
cd stats
# replace the contents of Main.lean with the program above
lake build
lake exe stats 3 5 10
```

```
count = 3
sum   = 18
mean  = 6.000000
```

The compiled binary is `.lake/build/bin/stats`. During development you can also test `main` in the editor with `#eval main ["3", "5", "10"]`.

Monads, briefly

`Option`, `Except ε`, `IO` and `Id` are all monads: types of computations that can be chained with `←`. The `do` notation is the same for each, and what `←` means depends on the monad.

Monad	A computation that	`←` means
`Option`	May have no result	Stop with `none` if the step has none
`Except ε`	May fail with an error	Stop with the error
`IO`	May act on the world	Run the action, get its result
`StateM σ`	Reads and writes a state	Thread the state through
`Id`	Does nothing special	Plain evaluation
`MetaM`, `TacticM`	Manipulates Lean proofs	Chapter 23

That table is enough to read and write `do` blocks. The theory, and the type classes `Functor`, `Applicative` and `Monad`, are the subject of Functional Programming in Lean. The standard library is documented in the Language Reference.

Errors you will meet here

Message	Cause	Fix
`failed to prove termination`	Lean cannot see that the recursion decreases	Add `termination_by` and `decreasing_by`, or use `partial def`
`could not synthesize a `Repr` or `ToString` instance for type Nat → Nat`	`#eval` cannot print a value of that type, here a function	Apply the function to an argument, or add `deriving Repr` to your type
`failed to prove index is valid`	`xs[i]` needs a proof that `i` is in bounds	Use `xs[i]!` or `xs[i]?`, or supply the bound
`Type mismatch: Option Nat` where `Nat` is expected	A fallible result used as a value	`match` on it, use `.getD`, or use `←` inside a `do` block
`invalid `do` notation, expected type is not a monad application`	`do` used where a plain value is expected	Wrap with `Id.run do`, or give the function an `IO` type

Message	Cause	Fix
`Variable `x` cannot be mutated. Only variables declared using `let mut` can be mutated.`	Reassignment of an immutable variable	Declare it with `let mut`

Exercises

```
-- Exercise 1. The sum 1² + 2² + … + n², once with a loop and once with map
and sum.
def sumSquares (n : Nat) : Nat := sorry

-- Exercise 2. Is the string equal to its reverse? Use s.toList.
def isPalindrome (s : String) : Bool := sorry

-- Exercise 3. The largest element, if there is one.
def maxOf : List Int → Option Int := sorry

-- Exercise 4. The decimal digits of n, most significant first.
def digits (n : Nat) : List Nat := sorry

-- Exercise 5. The number of primes up to n, by trial division.
def countPrimes (n : Nat) : Nat := sorry

-- Exercise 6. Print the multiplication table of 1, 2, 3 by 1, 2, 3.
def table : IO Unit := sorry

-- Expected:
-- #eval sumSquares 10          -- 385
-- #eval isPalindrome "kayak"   -- true
-- #eval maxOf [3, -1, 7, 2]    -- some 7
-- #eval digits 2026            -- [2, 0, 2, 6]
-- #eval countPrimes 100        -- 25
```

Solutions

```
-- Exercise 1.
def sumSquares (n : Nat) : Nat := Id.run do
  let mut s := 0
  for i in [1:n+1] do
    s := s + i * i
  return s

def sumSquares' (n : Nat) : Nat := ((List.range (n + 1)).map (· ^ 2)).sum

-- Exercise 2.
def isPalindrome (s : String) : Bool := s.toList == s.toList.reverse

-- Exercise 3.
def maxOf : List Int → Option Int
  | [] => none
  | x :: xs =>
    match maxOf xs with
    | none => some x
    | some m => some (max x m)
```

```
-- Exercise 4.
def digits (n : Nat) : List Nat :=
  if n < 10 then [n] else digits (n / 10) ++ [n % 10]

-- Exercise 5.
def countPrimes (n : Nat) : Nat := Id.run do
  let mut count := 0
  for k in [2:n+1] do
    let mut isP := true
    for d in [2:k] do
      if d * d > k then break
      if k % d == 0 then
        isP := false
        break
    if isP then count := count + 1
  return count

-- Exercise 6.
def table : IO Unit := do
  for i in [1:4] do
    let row := (List.range 3).map (fun j => toString (i * (j + 1)))
    IO.println (String.intercalate " " row)
```

In Exercise 4, Lean proves termination by itself: it sees that `n / 10 < n` under the condition `¬n < 10`.

Chapter 22: Verified Programs

A verified program is a function together with a theorem stating what it computes, both checked by the same kernel, so the program cannot disagree with its specification.

Everything in this chapter runs in plain Lean, with no import.

A function and its specification

```
def sumTo : Nat → Nat
  | 0 => 0
  | n + 1 => (n + 1) + sumTo n

#eval sumTo 100                                  -- 5050
example : sumTo 100 = 5050 := by decide      -- a test, checked by the
kernel
```

`#eval` and `decide` test one input. A theorem covers all of them:

```
theorem sumTo_eq (n : Nat) : 2 * sumTo n = n * (n + 1) := by
  induction n with
  | zero => rfl
  | succ k ih =>
    simp only [sumTo]
    -- ih : 2 * sumTo k = k * (k + 1)
    -- ⊢ 2 * (k + 1 + sumTo k) = (k + 1) * (k + 1 + 1)
    grind
```

The proof follows the recursion of the function. Unfold one step with `simp only [sumTo]`, then use the induction hypothesis. Here `grind` finishes: it uses `ih` and normalizes the polynomial arithmetic. In a file with Mathlib, `linarith` or `nlinarith` would do the same.

This is the pattern of the whole chapter: **the proof has the shape of the program.** Structural recursion is proved by structural induction, a `match` by cases, an `if` by `split`.

Two implementations, one specification

The function above is clear and slow to run on large inputs, since it builds a deep stack. A second version carries an accumulator:

```
def sumAcc : Nat → Nat → Nat
  | 0, acc => acc
  | n + 1, acc => sumAcc n (acc + (n + 1))

def sumFast (n : Nat) : Nat := sumAcc n 0
```

The claim to prove is `sumFast n = sumTo n`. A direct induction fails:

```
theorem sumAcc_zero_bad (n : Nat) : sumAcc n 0 = sumTo n := by
  induction n with
  | zero => rfl
  | succ k ih =>
    simp only [sumAcc, sumTo]
    -- ih : sumAcc k 0 = sumTo k
```

```
    -- ⊢ sumAcc k (0 + (k + 1)) = k + 1 + sumTo k
    sorry
```

The hypothesis speaks of accumulator `0`, and the goal needs accumulator `0 + (k + 1)`. The statement is too weak to be its own induction hypothesis.

The remedy is to state what the function does for every accumulator, and to tell `induction` that the accumulator varies:

```
theorem sumAcc_eq (n acc : Nat) : sumAcc n acc = acc + sumTo n := by
  induction n generalizing acc with
  | zero => rfl
  | succ k ih =>
    simp only [sumAcc, sumTo]
    -- ih : ∀ (acc : Nat), sumAcc k acc = acc + sumTo k
    -- ⊢ sumAcc k (acc + (k + 1)) = acc + (k + 1 + sumTo k)
    rw [ih]
    omega

theorem sumFast_eq (n : Nat) : sumFast n = sumTo n := by
  simp [sumFast, sumAcc_eq]
```

`sumAcc_eq` is the loop invariant of the accumulator version. Finding it is the mathematical content of the verification. `generalizing acc` is how you say that the invariant must hold for all values of the accumulator.

Once `sumFast_eq` is proved, every theorem about `sumTo` transfers to `sumFast`. You reason about the simple definition and run the fast one.

A sorting algorithm

Insertion sort, with the two properties that define a sorting function: the output is sorted, and it is a rearrangement of the input.

```
def insertSorted (x : Nat) : List Nat → List Nat
  | [] => [x]
  | y :: ys => if x ≤ y then x :: y :: ys else y :: insertSorted x ys

def isort : List Nat → List Nat
  | [] => []
  | x :: xs => insertSorted x (isort xs)

#eval isort [3, 1, 4, 1, 5, 9, 2, 6]         -- [1, 1, 2, 3, 4, 5, 6, 9]
example : isort [3, 1, 2] = [1, 2, 3] := rfl
```

Sortedness is an inductive predicate, as in Chapter 9:

```
inductive Sorted : List Nat → Prop
  | nil : Sorted []
  | single (x : Nat) : Sorted [x]
  | cons (x y : Nat) (ys : List Nat) : x ≤ y → Sorted (y :: ys) → Sorted
(x :: y :: ys)
```

The key lemma says that insertion preserves sortedness. The induction is on the proof that the list is sorted, which gives exactly the three shapes of a sorted list.

```
theorem sorted_insertSorted (x : Nat) (xs : List Nat) (h : Sorted xs) :
    Sorted (insertSorted x xs) := by
  induction h with
  | nil => exact Sorted.single x
  | single y =>
    simp only [insertSorted]
    split
    · exact Sorted.cons x y [] (by assumption) (Sorted.single y)
    · exact Sorted.cons y x [] (by omega) (Sorted.single x)
  | cons y z zs hyz hzs ih =>
    simp only [insertSorted] at ih ⊢
    -- ih : Sorted (if x ≤ z then x :: z :: zs else z :: insertSorted x zs)
    -- ⊢ Sorted (if x ≤ y then x :: y :: z :: zs
    --           else y :: if x ≤ z then x :: z :: zs else z :: insertSorted
x zs)
    split
    · exact Sorted.cons x y (z :: zs) (by assumption) (Sorted.cons y z zs
hyz hzs)
    · split
      · exact Sorted.cons y x (z :: zs) (by omega) (by simp_all)
      · exact Sorted.cons y z _ hyz (by simp_all)

theorem sorted_isort (xs : List Nat) : Sorted (isort xs) := by
  induction xs with
  | nil => exact Sorted.nil
  | cons x xs ih => exact sorted_insertSorted x (isort xs) ih
```

Each `split` handles one `if` of the program, and adds its condition, or the negation, to the context. `by assumption` then finds `x ≤ y`, and `omega` turns `¬x ≤ y` into `y ≤ x`.

The second property uses `List.Perm`, the relation "is a permutation of", from Lean's core library:

```
theorem perm_insertSorted (x : Nat) (xs : List Nat) : (insertSorted x
xs).Perm (x :: xs) := by
  induction xs with
  | nil => exact List.Perm.refl _
  | cons y ys ih =>
    simp only [insertSorted]
    split
    · exact List.Perm.refl _
    · exact (List.Perm.cons y ih).trans (List.Perm.swap x y ys)

theorem perm_isort (xs : List Nat) : (isort xs).Perm xs := by
  induction xs with
  | nil => exact List.Perm.refl _
  | cons x xs ih => exact (perm_insertSorted x (isort xs)).trans
(List.Perm.cons x ih)
```

Together, `sorted_isort` and `perm_isort` say that `isort` sorts. Neither alone does: the function returning `[]` satisfies the first, and the identity satisfies the second. Writing a complete specification is as much work as proving it.

Let automation follow the program

For routine facts, `grind` can do the case analysis and the arithmetic once it is told which definition to unfold:

```
theorem length_insertSorted (x : Nat) (xs : List Nat) :
    (insertSorted x xs).length = xs.length + 1 := by
  induction xs <;> grind [insertSorted]
```

When a function is not structurally recursive, its own recursion scheme is available through `fun_induction`. It produces one case per branch of the definition, with the induction hypotheses for the recursive calls.

```
def digits (n : Nat) : List Nat :=
  if n < 10 then [n] else digits (n / 10) ++ [n % 10]

theorem digits_lt (n : Nat) : ∀ d ∈ digits n, d < 10 := by
  fun_induction digits n with
  | case1 n h =>
    -- h : n < 10      ⊢ ∀ (d : Nat), d ∈ [n] → d < 10
    simp
    omega
  | case2 n h ih =>
    -- h : ¬n < 10     ih : ∀ (d : Nat), d ∈ digits (n / 10) → d < 10
    -- ⊢ ∀ (d : Nat), d ∈ digits (n / 10) ++ [n % 10] → d < 10
    intro d hd
    simp at hd
    rcases hd with hd | hd
    · exact ih d hd
    · omega
```

The short form is `fun_induction digits n <;> grind`.

Specifications in the type

A function can return its result together with the proof that it is correct, as a subtype. The caller then cannot obtain the value without the guarantee.

```
def isqrt (n : Nat) : { r : Nat // r * r ≤ n ∧ n < (r + 1) * (r + 1) } :=
  ⟨Nat.sqrt n, Nat.sqrt_le n, Nat.lt_succ_sqrt n⟩
```

The same idea makes array indexing safe: `xs[i]` compiles only with a proof that `i < xs.size`, and `Fin n` and `Vector α n` carry their bounds in their types.

Loops

Programs written with `for` and `let mut` are harder to reason about than recursive ones. The loop is compiled to a library function, and `rfl` and `decide` often cannot see through it, as with a loop over a range such as `[0:n+1]`.

The practical method is the one of this chapter: write a simple recursive specification, prove the theorems about it, and prove or test that the loop computes the same thing.

Lean also has a verification condition generator for `do` blocks, in `Std.Tactic.Do`. You state the loop invariant, and the tactic reduces the program to arithmetic goals. In Lean 4.35.0 the tactic is being replaced: `mvcgen` works and prints a deprecation warning, and its successor `vcgen` is marked experimental. The example below compiles with that warning.

```
import Std.Tactic.Do
open Std.Do

def mySum (l : Array Nat) : Nat := Id.run do
  let mut out := 0
  for i in l do
    out := out + i
  return out

theorem mySum_correct (l : Array Nat) : mySum l = l.sum := by
  generalize h : mySum l = x
  apply Id.of_wp_run_eq h
  mvcgen
  case inv1 => exact ⇓(xs, out) => ⌜xs.prefix.sum = out⌝
  all_goals mleave; grind
```

The invariant on the `case inv1` line says: after processing the prefix `xs.prefix` of the array, `out` is its sum. Treat this interface as provisional and check the Language Reference for its current form.

What is trusted

A theorem about a Lean function is checked by the kernel against the function's definition. Three things lie outside that check.

- **The specification.** The theorem says what you wrote, which may not be what you meant.
- **The compiler.** The executable is produced by Lean's compiler, which is not verified. The kernel reasons about the definition, not about the machine code.
- **`native_decide`.** This tactic proves a statement by compiling and running it, so it adds the compiler to what must be trusted. `#print axioms` then shows an extra axiom whose name contains `native_decide`. Mathlib does not allow it.

For mathematics the first point is the one that matters. It is the reason for the advice in Chapter 20: test your definitions.

Before proving a property, you can also search for counterexamples. With Mathlib, the tactic `plausible` tests a statement on random inputs and reports a failing one.

Exercises

```
-- Exercise 1.
def double : Nat → Nat
  | 0 => 0
  | n + 1 => double n + 2

theorem double_eq (n : Nat) : double n = 2 * n := by
```

```
  sorry

-- Exercise 2. Use Nat.pow_succ : b ^ (n + 1) = b ^ n * b.
def pow2 : Nat → Nat
  | 0 => 1
  | n + 1 => 2 * pow2 n

theorem pow2_eq (n : Nat) : pow2 n = 2 ^ n := by
  sorry

-- Exercise 3. An accumulator: find the invariant, and generalize.
def revAcc {α : Type} : List α → List α → List α
  | [], acc => acc
  | x :: xs, acc => revAcc xs (x :: acc)

theorem revAcc_eq {α : Type} (xs acc : List α) : revAcc xs acc = xs.reverse
++ acc := by
  sorry

-- Exercise 4. Insertion adds exactly one element.
def insertSorted (x : Nat) : List Nat → List Nat
  | [] => [x]
  | y :: ys => if x ≤ y then x :: y :: ys else y :: insertSorted x ys

theorem mem_insertSorted (x y : Nat) (xs : List Nat) :
    y ∈ insertSorted x xs ↔ y = x ∨ y ∈ xs := by
  sorry

-- Exercise 5. Use fun_induction. In the second case, apply the induction
-- hypothesis, then Nat.pow_add and Nat.pow_one, then omega.
def log2 (n : Nat) : Nat :=
  if n < 2 then 0 else 1 + log2 (n / 2)

theorem pow_log2_le (n : Nat) (h : 1 ≤ n) : 2 ^ log2 n ≤ n := by
  sorry
```

Solutions

```
-- Exercise 1.
def double : Nat → Nat
  | 0 => 0
  | n + 1 => double n + 2

theorem double_eq (n : Nat) : double n = 2 * n := by
  induction n with
  | zero => rfl
  | succ k ih =>
    simp only [double, ih]
    -- ⊢ 2 * k + 2 = 2 * (k + 1)
    omega

-- Exercise 2.
def pow2 : Nat → Nat
  | 0 => 1
  | n + 1 => 2 * pow2 n

theorem pow2_eq (n : Nat) : pow2 n = 2 ^ n := by
  induction n with
  | zero => rfl
```

```
  | succ k ih =>
    rw [pow2, ih, Nat.pow_succ, Nat.mul_comm]

-- Exercise 3.
def revAcc {α : Type} : List α → List α → List α
  | [], acc => acc
  | x :: xs, acc => revAcc xs (x :: acc)

theorem revAcc_eq {α : Type} (xs acc : List α) : revAcc xs acc = xs.reverse
++ acc := by
  induction xs generalizing acc with
  | nil => rfl
  | cons x xs ih =>
    simp only [revAcc, ih]
    -- ⊢ xs.reverse ++ x :: acc = (x :: xs).reverse ++ acc
    simp

-- Exercise 4.
def insertSorted (x : Nat) : List Nat → List Nat
  | [] => [x]
  | y :: ys => if x ≤ y then x :: y :: ys else y :: insertSorted x ys

theorem mem_insertSorted (x y : Nat) (xs : List Nat) :
    y ∈ insertSorted x xs ↔ y = x ∨ y ∈ xs := by
  induction xs with
  | nil => simp [insertSorted]
  | cons z zs ih =>
    simp only [insertSorted]
    split
    · simp
    · simp only [List.mem_cons, ih]
      -- ⊢ y = z ∨ y = x ∨ y ∈ zs ↔ y = x ∨ y = z ∨ y ∈ zs
      grind

-- Exercise 5.
def log2 (n : Nat) : Nat :=
  if n < 2 then 0 else 1 + log2 (n / 2)

theorem pow_log2_le (n : Nat) (h : 1 ≤ n) : 2 ^ log2 n ≤ n := by
  fun_induction log2 n with
  | case1 n hn =>
    -- hn : n < 2     ⊢ 2 ^ 0 ≤ n
    simp
    omega
  | case2 n hn ih =>
    -- ih : 1 ≤ n / 2 → 2 ^ log2 (n / 2) ≤ n / 2     ⊢ 2 ^ (1 + log2 (n /
2)) ≤ n
    have ih' := ih (by omega)
    rw [Nat.pow_add, Nat.pow_one]
    omega
```

Exercise 4 is also closed by `induction xs <;> grind [insertSorted]`.

Chapter 23: Metaprogramming

Lean's notation and tactics are written in Lean, and you can add your own to a project with a few lines.

There are four levels, in increasing power. Use the lowest one that does the job.

Level	Command	What it can do
1	`notation`, `infixl`, `prefix`	Give a symbol to an existing expression
2	`macro`	Translate new syntax into existing syntax
3	`syntax` + `macro_rules`	The same, with several rules, recursion, and later extension
4	`elab`	Run a program that inspects the goal and builds the proof term

Most mathematical projects need levels 1 and 2 only.

Level 1: notation

```
import Mathlib
open Filter Topology

notation:50 f " ⇝ " l => Tendsto f atTop (𝓝 l)

example : (fun n : ℕ => (1 : ℝ) / ((n : ℝ) + 1)) ⇝ 0 :=
  tendsto_one_div_add_atTop_nhds_zero_nat

#check (fun n : ℕ => (1 : ℝ) / ((n : ℝ) + 1)) ⇝ 0
-- (fun n ↦ 1 / (↑n + 1)) ⇝ 0 : Prop
```

The string pieces are the symbols, the identifiers `f` and `l` are the holes, and `50` is the precedence of the whole expression, the same as `=`. Lean uses the notation for input and for display.

Three variants cover most needs.

- `infixl:65 " ⊕ " => myAdd` declares a left-associative binary operator. `infixr`, `prefix` and `postfix` work the same way.
- `local notation` lasts until the end of the current section or file.
- `scoped notation`, inside a namespace, is active only after `open` of that namespace. This is how `𝓝` works: it is declared in `Mathlib/Topology/Defs/Filter.lean` as `scoped[Topology] notation "𝓝" => nhds`, and `open Topology` turns it on.

Choose symbols that Mathlib does not use. With the arrow `→` in place of `↠`, the example above fails with `Ambiguous term`, since `→` is already the notation for morphisms in a category.

Level 2: macros

A macro replaces one piece of syntax by another before elaboration. The most useful kind is a tactic that tries a fixed list of tactics.

```
import Mathlib

macro "analysis_trivial" : tactic =>
  `(tactic| first | positivity | (norm_num; done) | linarith | nlinarith |
fun_prop)

example (x : ℝ) : 0 ≤ x ^ 2 + 1 := by analysis_trivial
example (x y : ℝ) (h : x < y) : x < y + 1 := by analysis_trivial
example : Continuous fun x : ℝ => x ^ 2 + Real.exp x := by analysis_trivial
```

The backquote expression `` `(tactic| …) `` is a syntax quotation: it is the syntax tree of the tactic written inside. `first | a | b | c` tries each alternative and keeps the first that succeeds.

A macro can take arguments. An argument is written `name:category`, and used in the quotation as `$name`.

```
macro "exists_and_done " x:term : tactic => `(tactic| (refine ⟨$x, ?_⟩;
simp))

example : ∃ n : Nat, n + 2 = 5 := by exists_and_done 3
```

The common categories are `term`, `tactic`, `ident` (a name), `num` (a numeral) and `str` (a string).

Hygiene

A name introduced inside a macro is not visible to the code around it.

```
macro "intro_h" : tactic => `(tactic| intro h)

example (p : Prop) : p → p := by
  intro_h
  exact h          -- error: Unknown identifier `h`
```

The hypothesis exists, and the goal view shows it as `h✝`. This protects the user: a macro can never capture or shadow a name by accident. To introduce a name the user can refer to, take it as an argument.

```
macro "intro_as " x:ident : tactic => `(tactic| intro $x:ident)

example (p : Prop) : p → p := by
  intro_as h
  exact h
```

Level 3: syntax and macro_rules

`macro` is short for two commands: `syntax` declares the new form, and `macro_rules` gives its translation. Written separately, a form can have several rules, and anyone can add more later.

```
syntax "close" : tactic

macro_rules
  | `(tactic| close) => `(tactic| rfl)
macro_rules
  | `(tactic| close) => `(tactic| assumption)
macro_rules
  | `(tactic| close) => `(tactic| omega)

example : 3 = 3 := by close
example (p : Prop) (h : p) : p := by close
example (a : Nat) : a ≤ a + 1 := by close
```

Lean tries the rules from the most recent to the oldest, and keeps the first whose result succeeds. Core Lean's `trivial` tactic is defined this way, in `Init/Tactics.lean`, so a library can teach `trivial` new cases.

Rules can be recursive.

```
syntax "my_split_ands" : tactic   -- Mathlib already has a split_ands

macro_rules
  | `(tactic| my_split_ands) => `(tactic| first | (constructor <;>
my_split_ands) | skip)

example (p q r : Prop) (hp : p) (hq : q) (hr : r) : p ∧ q ∧ r := by
  my_split_ands
  -- three goals: ⊢ p, ⊢ q, ⊢ r
  · exact hp
  · exact hq
  · exact hr
```

The same commands define new terms. Here `term,+` means one or more terms separated by commas, and `$xs,*` matches the rest of the list.

```
syntax "mysum! " term,+ : term

macro_rules
  | `(mysum! $x) => `($x)
  | `(mysum! $x, $xs,*) => `($x + mysum! $xs,*)

#eval mysum! 1, 2, 3, 4      -- 10
```

Level 4: elaborators

A macro cannot look at the goal. An elaborator can: it is a program, run by Lean while it checks your file, with access to the goal, the hypotheses and the whole library.

```
import Lean
open Lean Elab Tactic Meta
```

```
elab "show_goal" : tactic => do
  let g ← getMainGoal
  logInfo m!"The goal is: {← g.getType}"

example : 1 + 1 = 2 := by
  show_goal        -- The goal is: 1 + 1 = 2
  rfl
```

The body is a `do` block, as in Chapter 21, in the monad `TacticM`. The string `m!"…"` builds a message in which expressions are pretty-printed.

Metaprograms live in a stack of monads. Each one adds something to the one before, and a function from a lower monad can be called in a higher one without ceremony.

Monad	What it gives access to	Typical functions
`CoreM`	The environment: all declarations	`getEnv`, `logInfo`, `throwError`
`MetaM`	Metavariables and local contexts	`inferType`, `isDefEq`, `whnf`, `mkAppM`
`TermElabM`	Turning syntax into terms	`Term.elabTerm`
`TacticM`	The list of goals	`getMainGoal`, `getGoals`, `evalTactic`
`CommandElabM`	Top-level commands	`Command.liftTermElabM`

Expressions

A term of Lean is a value of the type `Expr`. A goal is a metavariable, a hole whose type is the statement to prove. A tactic is a program that fills the hole, possibly creating new holes.

```
import Lean
open Lean Meta

#eval show MetaM Unit from do
  let e ← mkAppM ``Nat.add #[mkNatLit 2, mkNatLit 3]
  logInfo m!"{e}"                  -- Nat.add 2 3
  logInfo m!"{← inferType e}"      -- Nat
  logInfo m!"{← whnf e}"           -- 5
```

`mkAppM` builds an application and fills in the implicit arguments. `inferType` computes a type. `whnf` reduces to weak head normal form, which is what `rfl` does on both sides.

The double backquote in ``` ``Nat.add ``` is a name that Lean checks at once. With a misspelt name, ``` ``Nat.ad ```, the file fails immediately with `Unknown constant`. With a single backquote the error would appear only when the tactic runs.

Writing a tactic

Here is `assumption`, written from scratch.

```
import Lean
open Lean Elab Tactic Meta

elab "my_assumption" : tactic => withMainContext do
  let goal ← getMainGoal
  let target ← goal.getType
  for d in ← getLCtx do
    if d.isImplementationDetail then continue
    if ← isDefEq d.type target then
      goal.assign d.toExpr
      return
  throwError "my_assumption: no hypothesis matches {target}"

example (p q : Prop) (hp : p) (hq : q) : q := by my_assumption
```

Line by line:

- `withMainContext` makes the hypotheses of the main goal the current local context. Every tactic that reads hypotheses starts with it.
- `getLCtx` returns the local context. Each entry `d` has a name `d.userName`, a type `d.type`, and a term `d.toExpr` that refers to it.
- `isImplementationDetail` skips internal entries, such as the theorem being defined.
- `isDefEq` asks whether two expressions are definitionally equal, and assigns metavariables to make them so.
- `goal.assign` fills the hole. The goal is then closed.
- `throwError` makes the tactic fail with a message.

A second tactic applies the first hypothesis that fits, and leaves the premises as new goals.

```
elab "apply_any" : tactic => withMainContext do
  let goal ← getMainGoal
  for d in ← getLCtx do
    if d.isImplementationDetail then continue
    try
      let newGoals ← goal.apply d.toExpr
      replaceMainGoal newGoals
      return
    catch _ => continue
  throwError "apply_any: no hypothesis applies to {← goal.getType}"

example (p q r : Prop) (hpq : p → q) (hqr : q → r) (hp : p) : r := by
  apply_any
  -- ⊢ q
  apply_any
  -- ⊢ p
  apply_any
```

`try … catch` restores the proof state when an attempt fails, so a failed `apply` leaves no trace.

An elaborator can also inspect the shape of the goal, and run existing tactics.

```
elab "sides" : tactic => withMainContext do
  let t ← getMainTarget
  match t.eq? with
  | some (_, lhs, rhs) => logInfo m!"left: {lhs}, right: {rhs}"
  | none => throwError "the goal is not an equality"

example (a b : Nat) : a + b = b + a := by
  sides                              -- left: a + b, right: b + a
  exact Nat.add_comm a b

elab "finish" : tactic => do
  evalTactic (← `(tactic| first | rfl | assumption | omega | simp_all))

example (a : Nat) (h : 0 < a) : a ≠ 0 := by finish
```

Tactics are not trusted

A tactic can contain any bug without making Lean unsound. The proof term it builds is checked by the kernel afterwards.

```
elab "wrong_assign" : tactic => do
  let goal ← getMainGoal
  goal.assign (mkNatLit 3)         -- claims that 3 is a proof

example : True := by wrong_assign
-- error: (kernel) declaration type mismatch, '_example' has type
--   Nat
-- but it is expected to have type
--   True
```

This is the reason Mathlib can accept large automation such as `grind`, `simp` and `linarith`: none of it has to be believed.

Commands

The same mechanism defines commands, the `#check` family.

```
import Lean
open Lean Elab Meta

elab "#type_of " t:term : command => do
  Command.liftTermElabM do
    let e ← Term.elabTerm t none
    Term.synthesizeSyntheticMVarsNoPostponing
    let e ← instantiateMVars e
    logInfo m!"{e} : {← inferType e}"

#type_of 1 + 1                     -- 1 + 1 : Nat
#type_of fun (x : Nat) => x        -- fun x ↦ x : Nat → Nat
```

Debugging a metaprogram

- `logInfo m!"…"` prints any value in the Infoview. Use it freely.

- `set_option trace.Elab.step true in` before a declaration prints each elaboration step.
- `set_option trace.Meta.synthInstance true in` prints the instance search, the most useful trace for mathematics (Chapter 10).
- `set_option pp.all true in` shows every implicit argument and universe.
- Hover over any tactic in VS Code and use Go to Definition to read its source. Short ones to start with: `Mathlib/Tactic/Use.lean`, `Mathlib/Tactic/Set.lean` and `Mathlib/Tactic/Contrapose.lean`. The functions used above are in `Lean/Elab/Tactic/Basic.lean`.

Where to go next

The reference for this subject is the free book Metaprogramming in Lean 4. The Language Reference documents notations and macros in its chapter *Notations and Macros*.

Errors you will meet here

Message	Cause	What to do
`Ambiguous term … Possible interpretations`	Your notation uses a symbol that already has a meaning	Choose another symbol, or make the notation `local`
`Unknown identifier` `` `h` `` after a macro	Hygiene: the macro introduced `h`, and you cannot refer to it	Pass the name as an `ident` argument
`unknown tactic`, then `unsolved goals`	The macro is declared after its use, or its file is not imported	Move the declaration up, or import the file
`Unknown identifier` `` `getMainGoal` ``	The namespaces are not open	`open Lean Elab Tactic Meta`
`failed to synthesize instance of type class OfNat MessageData 0`	`let mut n := 0` followed by `m!"{n}"`: Lean guessed the wrong type for `n`	Write `let mut n : Nat := 0`
`Tactic` `` `…` `` `has not been implemented`	The `elab` above did not compile, so the syntax has no meaning	Fix the first error in the `elab`
`no hypothesis matches …` from your own tactic, although the hypothesis is there	`withMainContext` is missing, so `getLCtx` returned an old context	Start the body with `withMainContext do`
`Unknown constant` `` `Nat.ad` ``	A misspelt name in `` ``… ``	Correct the name

Message	Cause	What to do
`(kernel) declaration type mismatch`	Your tactic assigned a term of the wrong type	Check the term with `inferType` before `assign`, or build it with `goal.apply`
`Function expected at f 1 … <missing>`	Two `term` arguments in a row: the first one swallowed the second	Separate them with a keyword, or write `x:term:max`

Exercises

```
import Mathlib
open Lean Elab Tactic Meta

-- Exercise 1. Declare a notation  a ≈[ε] b  for  near ε a b,
-- then state and prove that 2 is within 1 of 5/2.
def near (ε a b : ℝ) : Prop := |a - b| < ε

-- notation ...

example : near 1 2 (5 / 2) := by
  sorry

-- Exercise 2. Write a macro `my_field` that tries `ring1`, and if that
fails
-- runs `field_simp`, then `ring1` on the goal that remains, if any.
-- (`ring1` fails when it cannot close the goal. `ring` does not always
fail.)

example (x y : ℝ) (hx : x ≠ 0) (hy : y ≠ 0) : 1 / x + 1 / y = (x + y) / (x *
y) := by
  sorry   -- by my_field

-- Exercise 3. Write a macro `use_and_norm x` that runs `use x`, then
-- `norm_num` on the goal that remains, if any.

example : ∃ x : ℝ, x ^ 2 = 4 := by
  sorry   -- by use_and_norm 2

-- Exercise 4. Write an elaborator `count_hyps` that prints the number of
-- hypotheses of the main goal.

example (a b : ℕ) (h : a = b) : b = a := by
  -- count_hyps      -- should print: 3 hypotheses
  exact h.symm

-- Exercise 5. Write an elaborator `list_hyps` that prints every hypothesis
-- as  name : type.

example (a b : ℕ) (h : a = b) : b = a := by
  -- list_hyps
  exact h.symm
```

Solutions

```
import Mathlib
open Lean Elab Tactic Meta

-- Exercise 1.
def near (ε a b : ℝ) : Prop := |a - b| < ε

notation:50 a " ≈[" ε "] " b => near ε a b

example : (2 : ℝ) ≈[1] (5 / 2) := by
  unfold near
  -- ⊢ |2 - 5 / 2| < 1
  rw [abs_lt]
  constructor <;> norm_num

-- Exercise 2. `a <;> b` runs `b` on every goal that `a` leaves, so it also
-- works when `field_simp` closes the goal by itself.
macro "my_field" : tactic => `(tactic| first | ring1 | (field_simp <;>
ring1))

example (x y : ℝ) : (x + y) ^ 2 = x ^ 2 + 2 * x * y + y ^ 2 := by my_field
example (x y : ℝ) (hx : x ≠ 0) : (x + y) / x = 1 + y / x := by my_field
example (x y : ℝ) (hx : x ≠ 0) (hy : y ≠ 0) : 1 / x + 1 / y = (x + y) / (x *
y) := by
  my_field

-- Exercise 3. `$x:term` tells the parser which kind of argument `use`
receives.
macro "use_and_norm " x:term : tactic => `(tactic| (use $x:term <;>
norm_num))

example : ∃ x : ℝ, x ^ 2 = 4 := by use_and_norm 2
example : ∃ n : ℕ, n * 2 = 10 := by use_and_norm 5
example : ∃ x : ℝ, 0 < x ∧ x < 1 := by use_and_norm 1 / 2

-- Exercise 4.
elab "count_hyps" : tactic => withMainContext do
  let mut n : Nat := 0
  for d in ← getLCtx do
    if !d.isImplementationDetail then n := n + 1
  logInfo m!"{n} hypotheses"

example (a b : ℕ) (h : a = b) : b = a := by
  count_hyps        -- 3 hypotheses
  exact h.symm

-- Exercise 5.
elab "list_hyps" : tactic => withMainContext do
  for d in ← getLCtx do
    if d.isImplementationDetail then continue
    logInfo m!"{d.userName} : {d.type}"

example (a b : ℕ) (h : a = b) : b = a := by
  list_hyps         -- a : ℕ,  b : ℕ,  h : a = b
  exact h.symm
```

In a project made with the `math` template, Mathlib's linter reports that `count_hyps` and `list_hyps` do nothing. That is correct: they only print, and you remove them once you have read their output.

Two traps appear in these solutions. In Exercise 2, `first | ring | …` would stop at `ring` even when `ring` leaves the goal open, since `ring` falls back to `ring_nf` and counts that as success. In Exercise 3, `use x; norm_num` fails with `No goals to be solved` whenever `use` closes the goal by itself.

Part V — Working Practice

Chapter 24: Projects, Lake, and Keeping Up with Mathlib

A Lean project is a folder in which three small files fix the Lean version, the dependencies and what to build; Lake, the build tool installed with Lean, reads them.

The files of a project

A project created with `lake new MyProject math` (Chapter 2) contains the following.

File or folder	Role	Who writes it
`lean-toolchain`	The Lean version, one line: `leanprover/lean4:v4.35.0`	Copied from Mathlib
`lakefile.toml`	Name, options, dependencies, build targets	You
`lake-manifest.json`	The exact commit of every dependency	`lake update`
`MyProject.lean`	The root of your library: a list of imports	You
`MyProject/`	Your Lean files	You
`.lake/`	Downloaded packages and compiled files	Lake
`.github/workflows/`	Automatic builds on GitHub	The template
`.gitignore`	Tells git to ignore `.lake/`	The template

Commit everything to git except `.lake/`. The manifest in particular must be committed: it is what makes the project build the same way on another machine.

You can delete `.lake/` at any time. `lake exe cache get` and `lake build` recreate it.

lakefile.toml, line by line

This is the file the `math` template writes.

```
name = "MyProject"
version = "0.1.0"
keywords = ["math"]
defaultTargets = ["MyProject"]

[leanOptions]
pp.unicode.fun = true # pretty-prints `fun a ↦ b`
relaxedAutoImplicit = false
weak.linter.mathlibStandardSet = true
```

```
maxSynthPendingDepth = 3

[[require]]
name = "mathlib"
scope = "leanprover-community"
rev = "v4.35.0"

[[lean_lib]]
name = "MyProject"
```

Line	Meaning
`name`	The package name. It is also the prefix of your modules.
`version`, `keywords`	Metadata, used if you publish the package on Reservoir, the Lean package index.
`defaultTargets`	What a bare `lake build` compiles.
`[leanOptions]`	Options applied to every file of the project, as if each began with `set_option`.
`pp.unicode.fun`	Display `fun x ↦ …` and not `fun x => …`.
`relaxedAutoImplicit = false`	An undeclared name such as `foo` in a statement is an error, and not a silently created variable. A single undeclared letter, such as α, is still bound automatically.
`weak.linter.mathlibStandardSet`	Turn on Mathlib's style linters (Chapter 25). `weak.` means: ignore the option if it does not exist.
`maxSynthPendingDepth = 3`	A setting of instance search that Mathlib uses. Keep it, so that your files behave as Mathlib's do.
`[[require]]`	One dependency. Double brackets mean the block may be repeated.
`scope`	The owner of the package on Reservoir.
`rev`	A branch, a tag or a commit. The template writes the tag of the Lean version that created the project, here `v4.35.0`, so Mathlib stays at that release. `master` would mean the newest Mathlib at the time of `lake update`.

Line	Meaning
`[[lean_lib]]`	A library to build: the file `MyProject.lean` and everything it imports.

A project with a program adds a block for the executable, as in the `std` template:

```
[[lean_exe]]
name = "demo"
root = "Main"
```

Older projects have a `lakefile.lean`, the same information written in Lean. Mathlib itself uses one: `lakefile.lean`. The Lake chapter of the Language Reference documents every field of both forms.

Files, modules and imports

A file is a module, and its path is its name. The file `MyProject/Analysis/Lemmas.lean` is the module `MyProject.Analysis.Lemmas`.

```
-- MyProject/Analysis/Lemmas.lean
import Mathlib

theorem my_lemma (x : ℝ) : 0 ≤ x ^ 2 := sq_nonneg x
```

```
-- MyProject/Analysis/Main.lean
import MyProject.Analysis.Lemmas

example : (0 : ℝ) ≤ 3 ^ 2 := my_lemma 3
```

```
-- MyProject.lean, the root file
import MyProject.Analysis.Lemmas
import MyProject.Analysis.Main
```

Three rules:

- Imports come first in a file, before anything else.
- An import brings everything that the imported file imports. `Main.lean` above has Mathlib, through `Lemmas.lean`.
- `lake build` compiles the root file and what it imports. A file that the root does not reach is not built, and its errors go unnoticed. Add each new file to the root.

Imports cannot form a cycle. If two files need each other, move the shared part to a third file that both import.

When you edit `Lemmas.lean`, a file that imports it does not see the change at once. VS Code shows `Imports are out of date and must be rebuilt`. Run **∀ > Server: Restart File** in the importing file.

Mathlib's own files begin with the keyword `module` and use `public import`. That is Lean's module system, which controls what a file exports. Your files do not need it, and a plain `import` of a Mathlib module works as shown.

The commands you need

Command	What it does
`lake build`	Compile the project. Only files that changed, and files that depend on them, are recompiled.
`lake build MyProject.Analysis.Lemmas`	Compile one module and what it imports.
`lake exe cache get`	Download Mathlib's compiled files. Run it after creating, cloning or updating a project.
`lake update`	Move the dependencies to the newest versions the lakefile allows, and rewrite the manifest.
`lake exe demo`	Build and run the executable `demo`.
`lake env lean MyProject/Test.lean`	Check one file from the terminal, with the project's dependencies.
`lake clean`	Delete the compiled files of your project.
`lake new Name math`	Create a project in a new folder. `lake init` does the same in the current folder.
`lake --version`	Print the versions of Lake and Lean in use.

In VS Code, the same actions are in the ∀ menu under **Project Actions...** (Chapter 3).

To work on a project from GitHub: `git clone` it, `cd` into it, run `lake exe cache get`, and open the folder in VS Code. Do not run `lake update` on someone else's project. The manifest already names the versions it was written for.

Updating Mathlib

Mathlib changes every day, and each version works with one Lean version only. An update therefore changes three things together: `lean-toolchain`, `lake-manifest.json`, and the compiled files.

You never have to update. A project that builds today builds in five years with the same manifest. Update when you need something new from Mathlib, and do it between two pieces of work, not in the middle of a proof.

The procedure below is the one in Mathlib's wiki page Using mathlib4 as a dependency.

1. Commit your work, so that you can come back.
2. In `lakefile.toml`, change `rev` to `"master"` if it names a tag, as the file of the `math` template does.
3. In VS Code, choose **∀ > Project Actions... > Project: Update Dependency...**, and click **Update Lean Version** if prompted. Or, in a terminal, run the two commands below.
4. Run `lake build` and repair what broke.
5. Commit `lakefile.toml`, `lean-toolchain` and `lake-manifest.json` together with your repairs.

```
curl
https://raw.githubusercontent.com/leanprover-community/mathlib4/master/lean-
toolchain -o lean-toolchain
lake update
```

The first line copies Mathlib's Lean version into your project. The second moves Mathlib to its newest commit. Mathlib's `lakefile.lean` contains a hook that then runs `lake exe cache get` for you.

To cancel an update that goes badly:

```
git checkout lean-toolchain lake-manifest.json
lake exe cache get
```

Repairing after an update

Most breakage is renamed lemmas, and Lean tells you the new name. A renamed lemma keeps working for some months and prints a warning of the form `` `old_name` has been deprecated: Use `new_name` instead ``, with a click-to-fix link in VS Code.

These renames were met while the examples of this book were checked against Mathlib `v4.35.0`. They show what an update looks like in practice.

Older name	Name in `v4.35.0`
`abs_add`	`abs_add_le`
`Set.mem_setOf_eq`	`Set.mem_ofPred_eq`
`Set.mem_diff`	`Set.mem_sdiff`
`Set.not_mem_empty`	`Set.notMem_empty`
`Filter.eventually_of_forall`	`Filter.Eventually.of_forall`
`if_pos`	`ite_eq_left`, or `simp [h]`
`Memℒp`, `snorm`	`MemLp`, `eLpNorm`
module `Mathlib.Data.Real.Sqrt`	module `Mathlib.Analysis.Real.Sqrt`

Four remedies cover the other cases.

- **A declaration has disappeared.** Its deprecation period ended, or it moved to another file. Search its statement on Loogle (Chapter 12). If your file has specific imports, replace them by `import Mathlib`, and run `#min_imports` afterwards to get the short list again.
- **A proof by `simp` no longer closes the goal.** The simp set changed. This is the reason for the rule of Chapter 13: use `simp only […]` anywhere except on the last line of a proof. Run `simp?` to get the new list.
- **A statement no longer typechecks.** A definition changed its arguments. Hover over the name to read the new signature.
- **Many things fail.** The step is too large. Update in stages, as below.

Staying on a fixed version

Mathlib tags a version for each Lean release, such as `v4.35.0`. The `math` template already names the one that matches the Lean version you created the project with. To choose another, name it in the lakefile:

```
[[require]]
name = "mathlib"
scope = "leanprover-community"
rev = "v4.35.0"
```

Then put the same version in `lean-toolchain` (`leanprover/lean4:v4.35.0`) and run `lake update mathlib`.

This is also the way to cross a long gap. Go from tag to tag, `v4.33.0`, then `v4.34.0`, then `v4.35.0`, and repair at each step. At each step the deprecation warnings still name the replacements.

A paper or a thesis should cite the commit of its formalization. That commit contains the manifest, so the result can be checked again later.

Other dependencies

Any Lean package on GitHub can be required by its address:

```
[[require]]
name = "somePackage"
git = "https://github.com/someone/somePackage"
rev = "main"
```

The package must be compatible with your Lean version, and with your Mathlib if it depends on Mathlib. Reservoir lists packages with the versions they build on.

Mathlib already brings eight packages: `batteries`, `aesop`, `Qq`, `proofwidgets`, `plausible`, `importGraph`, `LeanSearchClient` and `Cli`. Do not require them yourself. Mathlib's `lake-manifest.json` fixes their versions.

Automatic builds on GitHub

The `math` template adds three workflow files, which run on GitHub's servers once the project is pushed there.

File	What it does
`lean_action_ci.yml`	Builds the project at every push and pull request, with `lean-action`, and publishes its documentation
`update.yml`	Tries a Mathlib update when you start it by hand, or on a schedule if you uncomment the `schedule` lines. It opens a pull request if the build passes, and an issue if it fails.
`create-release.yml`	Tags a release of your project each time `lean-toolchain` changes

The project's generated `README.md` lists the two GitHub settings to enable for them. A green check on GitHub then means that every file reached by the root compiles. It does not mean that the project is free of `sorry`, which is only a warning.

Errors you will meet here

Message	Cause	What to do
`unknown module prefix 'Mathlib'`	The file was opened by itself, or the folder opened in VS Code is not the project root	**File > Open Folder** on the folder that contains `lakefile.toml`
`unknown module prefix 'MyProject'`	`lean` was run directly	Use `lake env lean File.lean`
`error: MyProject/Extra.lean: bad import 'MyProject.Missing'`	The imported file does not exist	Check the path and the spelling
Hundreds of `Building Mathlib.…` lines	The compiled files are missing	Stop, run `lake exe cache get`, restart the file
`Imports are out of date and must be rebuilt`	An imported file changed	**∀ > Server: Restart File**
`Not running `lake exe cache get` yet, as the `lake` version (…) does not match the toolchain version (…)`	`lake update` changed `lean-toolchain` during the update	Run `lake exe cache get` yourself

Message	Cause	What to do
`error: no test driver configured`	`lake test` needs a `testDriver` in the lakefile	Use `lake build`, or configure one
`` `foo` has been deprecated: Use `bar` instead ``	A lemma was renamed	Click the fix, or rename by hand
`object file '…olean' of module … does not exist`	A dependency was not compiled	`lake exe cache get`, then `lake build`

Exercises

These are done in a terminal and in VS Code.

1. In your project, create `MyProject/Sandbox.lean` containing `import Mathlib` and one `example`. Run `lake build`. Is the file compiled? Add what is needed so that it is.
2. Find the exact commit of Mathlib that your project uses.
3. Put a `sorry` in `Sandbox.lean` and run `lake build`. What does Lake print, and does the build fail?
4. Pin your project to Mathlib `v4.35.0`. Which files change?
5. A colleague clones your project and sees Lean start to compile Mathlib. What did they forget?

Solutions

1. No. `lake build` compiles `MyProject.lean` and its imports only. Add the line `import MyProject.Sandbox` to `MyProject.lean`, and run `lake build` again.
2. Open `lake-manifest.json` and read the `rev` field of the entry whose `name` is `mathlib`. It is a 40-character commit hash. The address `https://github.com/leanprover-community/mathlib4/tree/` followed by that hash shows Mathlib exactly as your project sees it.
3. Lake prints `warning: MyProject/Sandbox.lean:…: declaration uses` `` `sorry` ``, and the build succeeds. A `sorry` is a warning, not an error. To find every `sorry` in a project, search for the word in VS Code with Ctrl+Shift+F, or read the warnings of a full build.
4. Set `rev = "v4.35.0"` in `lakefile.toml`, write `leanprover/lean4:v4.35.0` in `lean-toolchain`, and run `lake update mathlib`. Unless the project was created with Lean v4.35.0, whose template already wrote that tag, three files change: `lakefile.toml`, `lean-toolchain` and `lake-manifest.json`.
5. They did not run `lake exe cache get` before opening the project.

Chapter 25: Proof Engineering

A long proof is built like a program: plan it on paper, write every step as a statement with `sorry`, make that skeleton compile, then fill the holes one at a time.

From the paper proof to a plan

Write the informal proof as a numbered list of steps before you open Lean. Each step should be one sentence with a precise statement.

The example of this chapter is the fact that a convergent real sequence is bounded. The paper proof has three steps.

1. From some rank N on, the terms are within 1 of the limit l.
2. The first N terms are finitely many, so they are bounded by some B.
3. The larger of B and |l| + 1 bounds every term.

Two rules decide the shape of the Lean code.

- A step used once, with a short proof, becomes a `have` inside the main proof.
- A step that is a statement in its own right, or whose proof exceeds about ten lines, becomes a separate lemma. Step 2 is one: it says nothing about limits.

Sorry-driven development

First write the skeleton: all the steps, no proofs.

```
import Mathlib
open Filter Topology

theorem bounded_of_tendsto (u : ℕ → ℝ) (l : ℝ) (h : Tendsto u atTop (𝓝
l)) :
    ∃ M, ∀ n, |u n| ≤ M := by
  -- Step 1: from some rank N on, u n is within 1 of l.
  have h1 : ∃ N, ∀ n ≥ N, |u n - l| < 1 := by
    sorry
  obtain ⟨N, hN⟩ := h1
  -- Step 2: the first N terms are bounded.
  have h2 : ∃ B, ∀ n < N, |u n| ≤ B := by
    sorry
  obtain ⟨B, hB⟩ := h2
  -- Step 3: the larger of the two bounds works.
  refine ⟨max B (|l| + 1), fun n => ?_⟩
  sorry
```

This compiles, with no error and the warning `declaration uses `sorry`` (besides the request for a module docstring described in the Preface). That already proves something: the three statements are well formed, and they fit together. A wrong plan fails here, before any work is spent on details.

Now fill the holes in any order. Each one is a small, separate problem, and the file compiles after each.

Step 2 becomes a lemma of its own, by induction on N.

```
theorem bounded_initial (u : ℕ → ℝ) (N : ℕ) : ∃ B, ∀ n < N, |u n| ≤ B := by
  induction N with
  | zero => exact ⟨0, fun n hn => absurd hn (Nat.not_lt_zero n)⟩
  | succ k ih =>
    obtain ⟨B, hB⟩ := ih
    refine ⟨max B |u k|, fun n hn => ?_⟩
    -- hn : n < k + 1     ⊢ |u n| ≤ max B |u k|
    rcases Nat.lt_succ_iff_lt_or_eq.mp hn with h | h
    · exact (hB n h).trans (le_max_left _ _)
    · rw [h]
      exact le_max_right _ _
```

The finished theorem keeps the comments of the skeleton. They are now its documentation.

```
theorem bounded_of_tendsto (u : ℕ → ℝ) (l : ℝ) (h : Tendsto u atTop (𝓝
l)) :
    ∃ M, ∀ n, |u n| ≤ M := by
  -- Step 1: from some rank N on, u n is within 1 of l.
  have h1 : ∃ N, ∀ n ≥ N, |u n - l| < 1 := by
    have := Metric.tendsto_atTop.mp h 1 one_pos
    -- this : ∃ N, ∀ n ≥ N, dist (u n) l < 1
    simpa only [Real.dist_eq] using this
  obtain ⟨N, hN⟩ := h1
  -- Step 2: the first N terms are bounded.
  obtain ⟨B, hB⟩ := bounded_initial u N
  -- Step 3: the larger of the two bounds works.
  refine ⟨max B (|l| + 1), fun n => ?_⟩
  rcases lt_or_ge n N with hn | hn
  · exact (hB n hn).trans (le_max_left _ _)
  · have h3 : |u n| ≤ |u n - l| + |l| := by
      have := abs_add_le (u n - l) l
      simpa using this
    have h4 := hN n hn
    exact le_trans (by linarith) (le_max_right _ _)
```

The lemmas used are `Metric.tendsto_atTop`, `Real.dist_eq` and `abs_add_le`.

Turning a step into a lemma

The tactic `extract_goal` prints the current goal as a standalone statement, with exactly the hypotheses it needs.

```
example (u : ℕ → ℝ) (l : ℝ) (h : Tendsto u atTop (𝓝 l)) : ∃ M, ∀ n, |u n| ≤
M := by
  have h1 : ∃ N, ∀ n ≥ N, |u n - l| < 1 := by
    extract_goal
    sorry
  sorry
```

The Infoview shows:

```
theorem _example.extracted_1 (u : ℕ → ℝ) (l : ℝ) (h : Tendsto u atTop (𝓝
l)) :
    ∃ N, ∀ n ≥ N, |u n - l| < 1 := sorry
```

Copy it above the theorem, give it a name that follows the naming conventions, and prove it there. This is also the way to prepare a minimal example for a question on Zulip (Chapter 27).

For a project of several thousand lines, the plan itself is a document. The tool leanblueprint turns a LaTeX file of statements into a web page with a dependency graph, where each node is coloured by its state: stated, proved, or not yet started.

Proofs that survive

A proof is read and recompiled many times, after each Mathlib update. Six habits keep it working.

Habit	Fragile	Robust
Name what you introduce	`intro` then `assumption`; `cases h`	`intro x hx`; `obtain ⟨ha, hb⟩ := h`
Never refer to generated names	`rename_i h1 h2`, names shown as `h✝`	Names chosen at the point of introduction
Fix the simp set in mid-proof	`simp` followed by other tactics	`simp only […]`, from `simp?`
State the intermediate goal	A long chain of `rw`	`have h : statement := by …`, or `calc`
One goal at a time	Two goals handled by consecutive lines	A `·` bullet for each goal
Close with a decisive tactic	`simp` that happens to finish	`exact`, `linarith`, `ring`, `omega`, `positivity`

A bare `simp` is fine as the last tactic of a goal: if the simp set changes, it still proves the goal or fails clearly. In the middle of a proof, the next tactic depends on the exact form `simp` produced, and that form changes between versions.

Finding the cause of a failure

Work through these in order.

1. **Read the whole message.** It names the tactic, the term, the type found and the type expected. Appendix C lists the common ones.
2. **Put the cursor just before the failing line** and read the goal. Most failures come from a goal that is not what you believed.

3. **Cut the proof.** Replace everything after a line by `sorry`. If the error disappears, the cause is after the cut. Move the cut until one line is isolated.
4. **Print what a tactic did.** `show_term` before a tactic prints the term it built. `trace_state` prints the goals at that point, including in a place the cursor cannot reach, such as inside a `<;>` chain.
5. **Check the types.** Hover over each `↑` and each numeral. A statement about `ℕ` and the same statement about `ℝ` are displayed alike.

The last point deserves an example. Both sides display as `↑n < ↑m`:

```
example (n m : ℕ) (h : (n : ℤ) < m) : (n : ℝ) < m := by
  exact h
-- error: Type mismatch
--   h
-- has type
--   @Nat.cast ℤ instNatCastInt n < ↑m
-- but is expected to have type
--   @Nat.cast ℝ Real.instNatCast n < ↑m
```

The message expands the part that differs: one cast goes to `ℤ`, the other to `ℝ`. The remedy is `exact_mod_cast h` (Chapter 14).

When two expressions look identical and the message does not expand them, add a display option before the declaration:

Option	Shows
`set_option pp.numericTypes true in`	The type of every numeral: `(2 : ℝ)`
`set_option pp.coercions.types true in`	The target type of every cast: `(↑n : ℝ)`
`set_option pp.explicit true in`	Every implicit argument
`set_option pp.all true in`	Everything, including universes. Long, and a last resort.

Slow proofs

Lean limits the work spent on one declaration, counted in heartbeats. The default limit is 200000. Beyond it, the proof fails:

```
(deterministic) timeout at `whnf`, maximum number of heartbeats (200000) has been reached

Note: Use `set_option maxHeartbeats <num>` to set the limit.
```

The word after `timeout at` varies: `whnf`, `isDefEq`, `typeclass`, `elaborator`.

Raising the limit hides the problem. Measure first. `#count_heartbeats in` before a declaration reports its cost.

```
import Mathlib

#count_heartbeats in
theorem expand_ring (a b c : ℝ) :
    (a + b + c) ^ 2 = a ^ 2 + b ^ 2 + c ^ 2 + 2 * (a * b + b * c + c * a) :=
by
  ring
-- Used 369 heartbeats, which is less than the current maximum of 200000.

#count_heartbeats in
theorem expand_nlinarith (a b c : ℝ) :
    (a + b + c) ^ 2 = a ^ 2 + b ^ 2 + c ^ 2 + 2 * (a * b + b * c + c * a) :=
by
  nlinarith [sq_nonneg (a + b + c), sq_nonneg (a - b), sq_nonneg (b - c),
sq_nonneg (c - a)]
-- Used 3624 heartbeats, which is less than the current maximum of 200000.
```

The same statement costs ten times more with the wrong tool. Your numbers will differ slightly.

`set_option profiler true in` before a declaration lists where the time goes: `simp`, `ring`, `linarith`, typeclass inference, type checking.

Cause	Remedy
`simp` with the full simp set on a large goal	`simp only […]`, from `simp?`
`nlinarith`, `polyrith` or `grind` searching	Give the key product as a hint, or prove the step by `calc`
`exact?`, `apply?`, `hint` or `aesop?` left in the file	Replace by the proof they printed
`decide` on a large case	`norm_num`, `omega`, `simp`, or `decide +kernel`
One proof of 200 lines	Split into lemmas: each has its own budget, and Lean rechecks only the one you edit
Lean unfolding definitions to compare two terms	`rw` or `simp only` with the definition's lemma, in place of `show`, `change` or `exact` of a term that matches only after unfolding
Instance search	`set_option trace.Meta.synthInstance true in`, then give the instance by `haveI` or a type ascription

Editing also slows down when a file is long, since Lean rechecks from the point of the edit to the end of the file. Keep files under about a thousand lines, and put finished material in a file that the working file imports.

If a declaration does need more, raise the limit for it alone, with a comment, as Mathlib's linter requires:

```
set_option maxHeartbeats 400000 in
-- The `simp` call unfolds a large definition.
theorem my_big_theorem : True := trivial
```

The linters

A project made with the `math` template runs Mathlib's style linters on every file. Their messages are warnings: the file compiles, and the proof is correct. Each warning names the option that turns it off.

Warning	Meaning	What to do
`` `simp` is a flexible tactic modifying `⊢`. Try `simp?` and use the suggested `simp only [...]` ``	A bare `simp` is followed by a tactic that depends on its result	Use the `simp only` it suggests
`` The `show` tactic should only be used to indicate intermediate goal states for readability. However, this tactic invocation changed the goal. Please use `change` instead ``	`show` rewrote the goal up to definitional equality	Write `change`
`The following tactic starts with 2 goals and ends with 1 goal, 1 of which is not operated on`	A tactic ran while several goals were open	Give each goal a `·` bullet
`This line exceeds the 100 character limit, please shorten it!`	The line is too long	Break it. Continuation lines are indented by 4 spaces.
`` Variable name `h` is not explicitly referenced ``	A hypothesis is never used	Delete it. The theorem becomes more general.
`` Unused tactic linter: `skip` does nothing. ``	A tactic changed nothing	Delete it
`Please, use '·' (typed as \.) instead of '.' as 'cdot'`	A bullet was typed as a plain dot	Type `\.`

Warning	Meaning	What to do
`Please, add a comment explaining the need for modifying the maxHeartbeat limit`	`set_option maxHeartbeats` without a reason	Add the comment line after it

A second family runs on demand. `#lint` at the end of a file checks its declarations, as Mathlib does before accepting a contribution.

```
import Mathlib

def myDef (n : ℕ) : ℕ := n + 1

theorem unusedArg (a b : ℕ) (h : a = b) : a = a := rfl

#lint
```

It reports three things here. `myDef` has no documentation string (`docBlame`). `unusedArg` states `a = a`, a syntactic tautology (`synTaut`). And its hypothesis `h` is never used (`unusedArguments`).

Style in ten lines

Mathlib's style guide and naming conventions are worth following in your own files. Code that looks like Mathlib is easier to compare with Mathlib.

- Lines of at most 100 characters.
- Two spaces of indentation in a proof, four for the continuation of a statement.
- `by` at the end of the line that precedes the proof, never alone on a line.
- One tactic per line. Short closers may share a line: `constructor <;> norm_num`.
- A `·` bullet for each goal when a tactic creates several.
- `fun x ↦ …` in Mathlib. `fun x => …` means the same and is accepted everywhere.
- Hypotheses to the left of the colon: `(h : 0 < x) : P x`, not `: 0 < x → P x`.
- Theorem names in `snake_case` that describe the statement: `abs_sub_lt_of_tendsto`.
- A documentation string `/-- … -/` on every definition.
- Shared variables declared once with `variable`, inside a `section`.

Errors you will meet here

Message	Cause	What to do
`declaration uses `sorry``	A hole remains	Expected during development. Search the project for `sorry` before you call it finished.

Message	Cause	What to do
`(deterministic) timeout at …, maximum number of heartbeats (200000) has been reached`	The declaration is too expensive	Measure with `#count_heartbeats in`, then see the table above
`Type mismatch`, with `@Nat.cast ℤ …` against `@Nat.cast ℝ …`	The same statement in two number systems	`exact_mod_cast`, `push_cast`, `norm_cast`
`'show' tactic failed, pattern … is not definitionally equal to target`	The goal you wrote is equal to the real one only by a lemma, not by definition	Use `rw`, `simp only`, or `suffices … by …`
`maximum recursion depth has been reached`	A term or a `simp` loop is too deep	Find the looping lemma with `simp?`. For large numerals, use `norm_num`.
`unsolved goals` at the theorem's `by`	A case is missing after a `·` block or a `constructor`	Put the cursor at the end of the proof and read which goal is left

Exercises

```
import Mathlib

-- Exercise 1. Fill the three holes of this skeleton.
def TendsToZero (u : ℕ → ℝ) : Prop := ∀ ε > 0, ∃ N, ∀ n ≥ N, |u n| < ε

theorem bounded_mul_tendsToZero (u v : ℕ → ℝ) (M : ℝ) (hM : 0 < M)
    (hu : ∀ n, |u n| ≤ M) (hv : TendsToZero v) : TendsToZero (fun n => u n *
v n) := by
  intro ε hε
  -- Step 1: use hv with ε / M.
  have h1 : ∃ N, ∀ n ≥ N, |v n| < ε / M := by
    sorry
  obtain ⟨N, hN⟩ := h1
  refine ⟨N, fun n hn => ?_⟩
  -- Step 2: the key inequality.
  have h2 : |u n * v n| ≤ M * |v n| := by
    sorry
  -- Step 3: conclude.
  have h3 : M * |v n| < ε := by
    sorry
  exact lt_of_le_of_lt h2 h3

-- Exercise 2. This proof works and is fragile. Rewrite it in two lines,
-- with names you choose and a decisive closing tactic.
example (a b : ℝ) (h : 0 ≤ a ∧ 0 ≤ b) : 0 ≤ a * b + a := by
  cases h
```

```
  rename_i h1 h2
  nlinarith [mul_nonneg h1 h2]

-- Exercise 3. Read the error, find the cause, and repair the proof.
example (n : ℕ) (h : 0 < n) : (0 : ℝ) < n := by
  exact h
```

Solutions

```
import Mathlib

-- Exercise 1.
def TendsToZero (u : ℕ → ℝ) : Prop := ∀ ε > 0, ∃ N, ∀ n ≥ N, |u n| < ε

theorem bounded_mul_tendsToZero (u v : ℕ → ℝ) (M : ℝ) (hM : 0 < M)
    (hu : ∀ n, |u n| ≤ M) (hv : TendsToZero v) : TendsToZero (fun n => u n *
v n) := by
  intro ε hε
  -- Step 1: use hv with ε / M.
  have h1 : ∃ N, ∀ n ≥ N, |v n| < ε / M := hv (ε / M) (by positivity)
  obtain ⟨N, hN⟩ := h1
  refine ⟨N, fun n hn => ?_⟩
  -- Step 2: the key inequality.
  have h2 : |u n * v n| ≤ M * |v n| := by
    rw [abs_mul]
    -- ⊢ |u n| * |v n| ≤ M * |v n|
    exact mul_le_mul_of_nonneg_right (hu n) (abs_nonneg _)
  -- Step 3: conclude.
  have h3 : M * |v n| < ε := by
    calc M * |v n| < M * (ε / M) := by gcongr; exact hN n hn
      _ = ε := by field_simp
  exact lt_of_le_of_lt h2 h3

-- Exercise 2.
example (a b : ℝ) (h : 0 ≤ a ∧ 0 ≤ b) : 0 ≤ a * b + a := by
  obtain ⟨ha, hb⟩ := h
  positivity

-- Exercise 3. The error says: h has type (0 : ℕ) < n, and the expected
-- type is (0 : ℝ) < ↑n. The hypothesis is about natural numbers and the
-- goal about real numbers.
example (n : ℕ) (h : 0 < n) : (0 : ℝ) < n := by
  exact_mod_cast h
```

Chapter 26: Lean and AI Tools

An AI system can write a Lean proof and Lean will check it, so the proof can be trusted; what Lean does not check is that the statement is the one you meant.

This chapter describes the tools as they stood in November 2026. The field changes monthly, so read the names as a snapshot and the method as the lasting part.

What Lean checks and what it does not

A language model produces text that looks right. In ordinary mathematics that is a danger. In Lean the text is compiled, and a wrong proof is rejected. This is why the two fit together: the model proposes, the kernel decides.

Lean checks	Lean does not check
Every step of the proof	That the statement says what the English sentence says
That no gap remains: a `sorry` is reported	That the definitions in the file are the standard ones
Which axioms the proof uses: `#print axioms`	That the hypotheses can be satisfied at all
That every name used exists, with the type used	That the result is new, or interesting

The right-hand column is your work. It is the subject of the section *Checking a proof you did not write* below.

Four kinds of tools

Kind	Examples	Where it runs
Search	Loogle (by pattern), LeanSearch and LeanExplore (by meaning)	Web, and inside Lean (Chapter 12)
Automation inside Lean	`exact?`, `grind`, `aesop`, `hint`, `try?` (Chapter 13); `hammer`	Your machine, as tactics
General assistants and coding agents	Claude, ChatGPT and Codex, Gemini	Chat, or an agent working in your project folder
Dedicated provers	AlphaProof, Aristotle, Seed-Prover, Gauss, and open models	Company servers, or your own GPU

Search is covered in Chapter 12. It remains the tool you will use most.

Automation. The tactics of Chapter 13 are not language models. One addition uses machine learning to choose lemmas: LeanHammer provides a `hammer` tactic that selects premises from Mathlib and calls Aesop, Duper and the external prover Zipperposition. Its README lists support for Lean `v4.20.0` through `v4.35.0`. It is a separate package that brings its own Mathlib, so its README puts this block in place of the Mathlib requirement:

```
[[require]]
name = "Hammer"
git = "https://github.com/JOSHCLUNE/LeanHammer"
rev = "main"
```

Assistants and agents. A chat model answers from memory and cannot compile. An agent runs inside your project, compiles its attempts, reads the errors and tries again. Two community packages connect agents to Lean.

- lean-lsp-mcp gives an agent the same information you get from the Infoview: the goal at a position, the error messages, and searches on LeanSearch and Loogle. With Claude Code the setup is one command, `claude mcp add lean-lsp uvx lean-lsp-mcp`. Version 0.33.0 is dated 3 November 2026.
- lean4-skills is a set of instructions for agents, with commands such as `draft`, `prove`, `review` and `golf`, and an axiom check at each save point. It supports Claude Code, Codex, Gemini CLI and Cursor. Version 4.12.0 is dated 8 November 2026.

Dedicated provers are systems trained specifically to produce Lean proofs. These are the main ones, with the dates of their announcements, taken from the AI for Mathematics: 2025–2026 Digest.

System	Who	What was reported
AlphaProof	Google DeepMind	A reinforcement-learning system working in Lean, with strong results on olympiad problems (paper in 2025)
Aristotle	Harmonic	Formally verified solutions at gold-medal level on the 2025 International Mathematical Olympiad (28 July 2025)
Seed-Prover	ByteDance	Four of the six problems of the same olympiad solved in Lean, and one partially (23 July 2025)
Gauss	Math Inc.	A Lean formalization of the strong Prime Number Theorem, produced with human guidance (September 2025)

System	Who	What was reported
Goedel-Prover, Kimina-Prover, DeepSeek-Prover-V2	Academic and open-source groups	Open models for Lean 4 that anyone can run (February to May 2025)

Aristotle is offered as a paid service. Its page describes the principle of all these systems in one sentence: "AI writes the proofs and Lean checks every step of them, so a result never rests on the AI being right."

Why assistants fail on Lean, and what to give them

A general assistant makes five typical errors.

Error	Why	Sign
Lean 3 syntax	Much of its training text predates Lean 4	`begin … end`, `λ x,`, `nat.succ`, `assume`, a comma after each tactic
Old lemma names	Mathlib renames lemmas every week (Chapter 24)	`Unknown identifier`, or a deprecation warning
Invented lemma names	It follows the naming conventions and guesses	`Unknown constant`, for a name that looks perfectly plausible
A proof that almost works	It cannot see the goal after each tactic	`unsolved goals`, `linarith failed`
A changed statement	It is asked to succeed, and a weaker statement is easier	An added hypothesis, a changed type, a new `def`

The first four cost time only, since Lean rejects the proof. The fifth is the dangerous one.

Give the assistant what a colleague would need:

- the Lean and Mathlib versions, from `lean-toolchain` and `lake-manifest.json`;
- the whole file, with its `import` and `open` lines;
- the goal, copied from the Infoview, not retyped;
- the full error message, when there is one.

And ask for small things: a plan as a skeleton with `sorry` (Chapter 25), then one hole at a time.

```
I use Lean 4.35.0 with Mathlib v4.35.0. Here is my file:

  [the whole file]
```

```
The goal at the `sorry` on line 14 is:

  u : ℕ → ℝ
  l : ℝ
  h : Tendsto u atTop (𝓝 l)
  ⊢ ∃ N, ∀ n ≥ N, |u n - l| < 1

Fill this one `sorry`. Do not change the statement of the theorem,
and do not add definitions or hypotheses. If you are unsure that a
lemma name exists, say so, and tell me what to search for on Loogle.
```

Then compile the answer, and paste the error back if there is one. Three rounds usually settle a short goal. An agent with `lean-lsp-mcp` performs this loop by itself.

Checking a proof you did not write

Apply this list to any Lean proof you receive, from a person or from a machine.

1. **It compiles in your project.** Your Lean version, your Mathlib, no error.
2. **It contains no `sorry`.** The build prints no `declaration uses `sorry`` warning.
3. **It uses only the standard axioms.** `#print axioms name` shows nothing beyond `propext`, `Classical.choice` and `Quot.sound`.
4. **The statement is yours.** Write the statement yourself, and prove it by `:= their_theorem`. If that fails, the two statements differ.
5. **Nothing was added around it.** Read every new `def`, `structure`, `instance`, `axiom`, `notation` and `macro` in the file.
6. **The hypotheses can hold.** Check them on one concrete example.
7. **The statement survives Lean's conventions.** See the table below.

Each of these theorems compiles. None proves the sentence in its comment.

```
import Mathlib
open Filter Topology

-- "The square of a real number is positive."
axiom helper (x : ℝ) : 0 < x ^ 2
theorem thmA (x : ℝ) : 0 < x ^ 2 := helper x

#print axioms thmA
-- 'thmA' depends on axioms: [helper, propext, Classical.choice, Quot.sound]

-- "Every real sequence converges."
theorem thmC (u : ℕ → ℝ) (h : ∀ n, u (n + 1) < u n ∧ u n < u (n + 1)) :
    ∃ l, Tendsto u atTop (𝓝 l) := by
  have := h 0
  linarith [this.1, this.2]

-- "Every real number can be divided by zero."
theorem thmD (x : ℝ) : x / 0 = 0 := div_zero x

-- "2 to the power 20 is 1048576."  True, and proved outside the kernel.
theorem thmE : (2 : ℕ) ^ 20 = 1048576 := by native_decide
```

```
#print axioms thmE
-- 'thmE' depends on axioms: [propext, thmE._native.native_decide.ax_1_1]
```

`thmA` rests on an invented axiom, which is false at `x = 0`. `thmC` has a hypothesis that no sequence satisfies, so it says nothing. `thmD` is true in Lean and means something other than the sentence. `thmE` is true, and its proof trusts the compiler in place of the kernel.

Conventions that change the meaning of a statement

Lean's functions are total: they return a value on every input. Where mathematics leaves an expression undefined, Mathlib assigns a default value. A statement can be true for that reason alone.

Expression	Value in Mathlib	Lemma
`x / 0`, 0^{-1}	`0`	`div_zero`, `inv_zero`
`a - b` in $\mathbb{N}$, when `a ≤ b`	`0`	`Nat.sub_eq_zero_of_le`
`7 / 2` in $\mathbb{N}$	`3`	Division rounds down
$\sqrt{x}$ for `x ≤ 0`	`0`	`Real.sqrt_eq_zero_of_nonpos`
`Real.log 0`	`0`	`Real.log_zero`
`Real.log (-x)`	`Real.log x`	`Real.log_neg_eq_log`
`deriv f x`, when `f` is not differentiable at `x`	`0`	`deriv_zero_of_not_differentiableAt`
`∫ x, f x ∂μ`, when `f` is not integrable	`0`	`MeasureTheory.integral_undef`
`∑' n, f n`, when the series is not summable	`0`	`tsum_eq_zero_of_not_summable`
`sSup s` in $\mathbb{R}$, when `s` is empty or unbounded	`0`	`Real.sSup_empty`, `Real.sSup_of_not_bddAbove`

A statement such as `deriv f x = 0` is therefore weaker than "f has derivative 0 at x". The strong form is `HasDerivAt f 0 x` (Chapter 18). When you read a formal statement, look for the predicate that carries the existence: `HasDerivAt`, `Integrable`, `Summable`, `BddAbove`, `x ≠ 0`.

Mathlib's rules on AI

Mathlib's page Contributing to mathlib has a section *Use of AI*. Its main rules, as of mid-2026:

- If you used an AI tool for a contribution, say so in the description, with the tool and how you used it. A contribution with a substantial amount of generated code gets the label `LLM-generated`.
- You must understand everything in the contribution, and be able to justify each design decision to reviewers without the help of an AI.
- Comments on GitHub and on Zulip must be in your own words. Writing them with a language model is not allowed.
- Low-quality generated contributions are closed without comment, and repeated ones lead to suspension.

The page gives the reason: "As of mid-2026, code written by an AI without the supervision of a Lean subject expert fails to meet that bar by a large margin." Compiling is the minimum. Mathlib also asks for the right generality, the right place in the library, and proofs that can be maintained.

Exercises

```
import Mathlib

-- Exercise 1. An assistant returns this as a proof of:
-- "for natural numbers, (a - b) + b = a".
-- Does it prove that sentence? Find a and b for which the sentence is
false.
theorem sub_add_or (a b : ℕ) : a - b + b = a ∨ a < b := by omega

-- Exercise 2. Which of these two statements says "f has a critical point at
0"?
-- Explain what the other one allows.
def crit1 (f : ℝ → ℝ) : Prop := deriv f 0 = 0
def crit2 (f : ℝ → ℝ) : Prop := HasDerivAt f 0 0

-- Exercise 3. This is the statement you wanted:
--   theorem mine (x : ℝ) (hx : 0 < x) : Real.log (x ^ 2) = 2 * Real.log x
-- and this is what you received. Use point 4 of the checklist: prove `mine`
-- from `theirs`. Does it work? What does that tell you?
theorem theirs (x : ℝ) (hx : 0 < x) (hx1 : x ≠ 1) : Real.log (x ^ 2) = 2 *
Real.log x := by
  rw [Real.log_pow]
  norm_num

theorem mine (x : ℝ) (hx : 0 < x) : Real.log (x ^ 2) = 2 * Real.log x := by
  sorry
```

Exercise 4, on paper. The answer below came from an assistant. Name three signs that it was not written for current Lean.

```
theorem foo (a b : ℕ) : a + b = b + a :=
begin
  induction a with n ih,
  { simp },
  { rw [nat.succ_add, ih] }
end
```

Solutions

```
import Mathlib

-- Exercise 1. No. The theorem has an escape clause, `∨ a < b`. The sentence
-- is false for a = 2, b = 3, since subtraction in ℕ stops at 0:
example : (2 : ℕ) - 3 + 3 = 3 := by norm_num
-- The honest statement carries the hypothesis:
example (a b : ℕ) (h : b ≤ a) : a - b + b = a := Nat.sub_add_cancel h

-- Exercise 2. `crit2`. `crit1` also holds for every function that is not
-- differentiable at 0, such as the absolute value:
example : deriv (fun x : ℝ => |x|) 0 = 0 :=
  deriv_zero_of_not_differentiableAt not_differentiableAt_abs_zero

-- Exercise 3. It does not work directly: `theirs` needs `x ≠ 1`, which
`mine`
-- does not have. The received theorem is weaker than the one requested.
-- Here the extra hypothesis was never needed:
theorem mine (x : ℝ) (hx : 0 < x) : Real.log (x ^ 2) = 2 * Real.log x := by
  rw [Real.log_pow]
  norm_num
-- Lean's linter adds a remark of its own: `hx` is not used either.
-- With the convention Real.log (-x) = Real.log x, the identity holds for
all x.
```

Exercise 4. Any three of these: `begin … end` in place of `by`; a comma after each tactic; braces `{ }` for the cases, in place of `·` bullets or `| zero => …`; `induction a with n ih` without case names; `nat.succ_add` in lower case, where Lean 4 writes `Nat.succ_add`. This is Lean 3, which Mathlib left in 2023.

Chapter 27: Community, Contributing, and Where to Go Next

Lean is learned in a community: questions go to the Lean Zulip chat, and Mathlib grows through reviewed contributions from its users.

The Zulip chat

The Lean Zulip is the meeting place of users, Mathlib maintainers and Lean developers. It is free, and a question there is usually answered within hours.

Channel	Use it for
`#new members`	Any beginner question. Start here.
`#Is there code for X?`	"Does Mathlib have this theorem?"
`#mathlib4`	Design and development of Mathlib
`#lean4`	The language itself, Lake, the editor
`#PR reviews`	Asking for a review of a contribution
`#maths`	Mathematical questions around a formalization

Use your real name as display name, as the community asks. Write in your own words: Mathlib's rules do not allow comments written by a language model (Chapter 26).

Asking a question that gets answered

A good question contains a minimal working example: a complete file that anyone can paste into the web editor and see your problem. The community page How to write a minimal working example describes it.

A minimal working example has four properties.

- It starts with its imports, usually `import Mathlib`, and its `open` lines.
- It contains every definition it uses.
- It contains nothing else. Remove hypotheses, steps and definitions until the problem disappears, then put the last one back.
- It is posted as text between triple backticks, never as a screenshot.

Add the error message, and one sentence saying what you expected.

`extract_goal` (Chapter 25) does most of the reduction for you. Reducing often solves the problem by itself: the cause becomes visible once the surroundings are gone.

Here is a problem as it first appears, inside a longer file:

```
import Mathlib
open Real Filter Topology
```

```
noncomputable def myF (x : ℝ) : ℝ := x ^ 2 + 3 * x

lemma myF_nonneg (x : ℝ) (hx : 0 ≤ x) : 0 ≤ myF x := by
  unfold myF
  positivity

theorem stuck (u : ℕ → ℝ) (hu : ∀ n, 0 ≤ u n) (n : ℕ) :
    myF (u n) / (1 + myF (u n)) ≤ 1 := by
  rw [div_le_one]
  linarith
-- error: unsolved goals
-- ⊢ 0 < 1 + myF (u n)
```

And here is the same problem, reduced. The sequence and the function `myF` played no role.

```
import Mathlib

example (a : ℝ) (ha : 0 ≤ a) : a / (1 + a) ≤ 1 := by
  rw [div_le_one]
  linarith
-- error: unsolved goals
-- ⊢ 0 < 1 + a
```

Now the cause can be read off. `div_le_one` has a hypothesis `0 < b`, and `rw` left it as a goal that nothing closed. In this reduced example the repair is `rw [div_le_one (by positivity)]`. In `stuck` itself, `positivity` cannot see inside `myF`, so give it the fact: `rw [div_le_one (by linarith [myF_nonneg (u n) (hu n)])]`.

Contributing to Mathlib

Read Contributing to mathlib first. It explains what is in scope, and it warns about the queue: in mid-2026 Mathlib had more than 2600 open pull requests, so a review takes time.

A good first contribution is small: one missing lemma that you needed in your own work, next to the lemmas it resembles. Before writing it:

1. Check that it is missing, with Loogle, and by asking in `#Is there code for X?`.
2. State it in the generality of its neighbours. A lemma about `ℝ` often belongs to every ordered field.
3. Name it by the naming conventions, and format it by the style guide.

The mechanics follow the page How to contribute to mathlib.

1. On GitHub, fork `leanprover-community/mathlib4`.
2. Clone your fork and fetch the compiled files. This is done once.
3. Create a branch for each change.
4. Edit, commit, and push to your fork.
5. Open a pull request against the `master` branch of the main repository.
6. Wait for the automatic build. A green tick means that Mathlib compiles with your change.

7. Answer the reviewers' comments with new commits. When you have addressed them, write `-awaiting-author` in a comment to return the pull request to the review queue.
8. A maintainer approves with `bors merge`, and a bot merges your change.

```
git clone https://github.com/YOUR_USERNAME/mathlib4.git
cd mathlib4
lake exe cache get
git switch -c my_new_branch
# edit, then:
git commit -a
git push --set-upstream origin my_new_branch
```

If you add a file, run `lake exe mk_all`, which adds it to `Mathlib.lean`.

The title of a pull request has a fixed form, described in the title conventions: `type(scope): subject`.

Part	Rule	Example
`type`	One of `feat`, `fix`, `doc`, `style`, `refactor`, `test`, `chore`, `perf`, `ci`	`feat` for a new lemma
`scope`	Optional. The file or folder, without the `Mathlib/` prefix	`Analysis/SpecialFunctions/Sqrt`
`subject`	Imperative, no capital letter, no final dot	`add a bound on the square root`

Keep each pull request small. Several pull requests of thirty lines are reviewed faster than one of three hundred, and the first review teaches you what to change in the others.

Reviewing is also contributing. Anyone may review, and the review queue lists the pull requests that wait.

Your own project, or Mathlib?

Not everything belongs in Mathlib. Mathlib's page recommends a separate repository, with Mathlib as a dependency (Chapter 24), for work outside the maintainers' areas and for projects that want to move fast.

Large formalizations work this way. They live in their own repository, and move their general lemmas into Mathlib over time. The page How to contribute to mathlib names several of them, with a label for the pull requests they send, among them `carleson` (Carleson's theorem on the convergence of Fourier series), `brownian` (Brownian motion), `sphere-packing` and `FLT` (Fermat's Last Theorem). For an analyst, the Carleson project is a good one to read: harmonic analysis at research level, with a blueprint.

Reservoir indexes the public Lean packages.

Where to go next

The list follows the community's page Learning Lean 4.

If you want	Read or do
More exercises on mathematics, with solutions	Mathematics in Lean. Its chapters on topology, differential calculus, and integration and measure theory continue Part III of this book.
A slower start on proof itself	The Mechanics of Proof
A game	The Natural Number Game, and the others on the Lean Game Server
Two afternoons of guided files, including elementary analysis	Glimpse of Lean
The logic and the type theory	Theorem Proving in Lean 4
Programming	Functional Programming in Lean
Verification of programs	The Hitchhiker's Guide to Logical Verification
To write tactics	The Tactic Programming Guide, then Metaprogramming in Lean 4
Every detail of the language	The Language Reference
What Mathlib contains	The Mathlib overview and the API documentation
Talks and courses	The community's events page and its YouTube channel

The most effective next step is a project of your own. Take a theorem you know well, from a course you teach or from your own papers, of about one page. State it, plan it as in Chapter 25, and prove it. You will meet every topic of this book again, in the order in which you need it.

Exercises

```
import Mathlib
open Real Filter Topology

-- Exercise 1. Reduce this to a minimal working example, then repair it.
noncomputable def g (x : ℝ) : ℝ := Real.exp x + x ^ 2

lemma g_pos (x : ℝ) : 0 < g x := by
  unfold g
  positivity
```

```
theorem bound (v : ℕ → ℝ) (n : ℕ) (h : g (v n) ≤ 5) : g (v n) / 5 ≤ 1 := by
  rw [div_le_one]
  exact h
```

Exercise 2. You add the lemma `Real.sqrt_le_add_one` to the file `Mathlib/Analysis/SpecialFunctions/Sqrt.lean`. Write the title of the pull request.

Exercise 3. Your pull request received three comments from a reviewer, and the label `awaiting-author`. You have pushed commits that address all three. What do you do so that reviewers see it again?

Solutions

```
import Mathlib

-- Exercise 1. The function g, the sequence v and the lemma g_pos are not
-- needed to show the problem:
example (a : ℝ) (h : a ≤ 5) : a / 5 ≤ 1 := by
  rw [div_le_one]
  exact h
-- error: unsolved goals
-- ⊢ 0 < 5

-- The hypothesis `0 < b` of div_le_one was left as a goal. Give it
directly:
example (a : ℝ) (h : a ≤ 5) : a / 5 ≤ 1 := by
  rw [div_le_one (by norm_num)]
  exact h
```

Exercise 2. `feat(Analysis/SpecialFunctions/Sqrt): add Real.sqrt_le_add_one`. The type is `feat`, the scope omits `Mathlib/` and `.lean`, and the subject is imperative, without a capital letter or a final dot.

Exercise 3. Write a comment on the pull request containing the line `-awaiting-author`. A bot removes the label, and the pull request returns to the review queue.

Appendices

Appendix A: Which Tactic When

Find the shape of your goal, or of a hypothesis, in the left column, and try the tactics on the right in the order given. This appendix gives names only; each one is defined in Appendix E.

By the shape of the goal

The goal is	Try, in this order
`P → Q`, `∀ x, P x` or `¬P`	`intro`, `rintro`
`P ∧ Q`	`constructor`, `refine ⟨?_, ?_⟩`, `exact ⟨hp, hq⟩`
`P ∨ Q`	`left`, `right`
`P ↔ Q`	`constructor`, `rw`, `simp`
`∃ x, P x`	`use`, `exists`, `refine ⟨a, ?_⟩`
`False`	`contradiction`, `linarith`, `omega`, `simp_all`
`¬∀ …` or `¬∃ …`	`push Not` (formerly `push_neg`)
The same as a hypothesis	`assumption`, `exact`
A hypothesis, after simplification	`simpa using h`
True by definition	`rfl`, `unfold` then `rfl`, `decide`
An equation between numbers	`norm_num`, `rfl`, `decide`
An equation in a commutative ring	`ring`, `linear_combination`
An equation with divisions	`field_simp` then `ring`, `field`
An equation in an additive group, or a module	`abel`, `module`
An equation between functions	`funext`, `ext`
An equation between sets, structures or matrices	`ext`
`f a = f b`	`congr`, `congr!`, `congrm`
Almost a hypothesis, or almost a lemma	`convert`, `convert_to`

The goal is	Try, in this order
A linear inequality	`linarith`, `bound`, `nlinarith`
An inequality between natural numbers or integers	`omega`, `lia`
`0 ≤ e`, `0 < e` or `e ≠ 0`	`positivity`
`f a ≤ f b`, with `a ≤ b` known	`gcongr`, `rel`, `grw`
A polynomial inequality	`nlinarith [sq_nonneg (a - b)]`, `linear_combination`
An inequality in a lattice or a partial order	`order`
`x < ∞` or `x ≠ ∞` in `ℝ≥0∞`	`finiteness`
Divisibility or remainders of integers	`omega`, `decide`, `norm_num`
`Nat.Prime 101`, and other facts about numerals	`norm_num`, `decide`
`Continuous f`, `Differentiable ℝ f`, `ContDiff ℝ n f`	`fun_prop`, `continuity`
`Measurable f`, `AEMeasurable f`	`measurability`, `fun_prop`
`∀ᶠ x in l, P x`, or `s ∈ l` for a filter `l`	`filter_upwards`
`s ⊆ t`, set identities	`intro x hx`, `ext x`, `tauto_set`
A statement of propositional logic	`tauto`, `itauto`
A statement about a few concrete values	`decide`, `fin_cases`, `interval_cases`
A goal containing `if … then … else`	`split_ifs`, `split`, `simp [h]`
A goal containing a `match`	`split`, `cases`
A goal with casts `↑n`	`push_cast`, `norm_cast`, `exact_mod_cast`
A goal in `ℕ` that would be easier in `ℤ`, `ℚ` or `ℝ`	`zify`, `qify`, `rify`
`TFAE [P₁, P₂, P₃]`	`tfae_have`, `tfae_finish`
A class instance	`infer_instance`, `exact inferInstance`
Anything, and you have no idea	`exact?`, `apply?`, `hint`, `try?`, `aesop`, `grind`

By the shape of a hypothesis

The hypothesis h is	Try
`P ∧ Q`	`obtain ⟨hp, hq⟩ := h`
`P ∨ Q`	`rcases h with hp | hq`, `obtain hp | hq := h`
`∃ x, P x`	`obtain ⟨x, hx⟩ := h`
`P ↔ Q`	`rw [h]`, `h.mp`, `h.mpr`
`∀ x, P x`	`specialize h a`, `have h' := h a`
`P → Q`, with `hp : P` known	`specialize h hp`, `apply h`
`a = b`	`rw [h]`, `rw [← h]`, `subst`, `simp [h]`
`False`	`contradiction`, `exact h.elim`
¬P, with `hp : P` known	`contradiction`, `exact absurd hp h`
`¬∀ …` or `¬∃ …`	`push Not at h`
An inequality to combine with others	`linarith`, `nlinarith [h]`
`f x = f y`, to cancel `f`	`apply_fun`, `injection`
A hypothesis to simplify	`simp at h`, `norm_num at h`, `simp_all`
`n ∈ Finset.range 5`, `n < 5`	`interval_cases`, `fin_cases`
A hypothesis about an element of an inductive type	`cases`, `rcases`, `induction`
A hypothesis with casts	`norm_cast at h`, `push_cast at h`
`∀ x, ∃ y, P x y`, to get a function	`choose`
`h : x = a` with `x` a variable	`subst x`, `subst_vars`

By what you want to do

You want to	Use
State an intermediate fact	`have`, `suffices`, `obtain`, `observe`
Chain equalities and inequalities	`calc`, `trans`
Argue by contradiction	`by_contra`, `by_contra!`, `exfalso`
Prove the contrapositive	`contrapose`, `contrapose!`
Split on a proposition	`by_cases`, `by_cases!`

You want to	Use
Split on a natural number	`rcases n with _ | n`, `cases`, `mod_cases`, `interval_cases`
Argue by induction	`induction`, `fun_induction`
Use a symmetry of the statement	`wlog`
Give a name to an expression	`set`, `generalize`, `let`
Rewrite under ∀, ∃, `fun` or ∑	`simp_rw`, `simp only`, `conv`
Rewrite one occurrence only	`nth_rw`, `conv`
Unfold a definition	`unfold`, `delta`, `simp only [f]`, `dsimp only [f]`
Replace the goal by an equal one, by definition	`change`, `show`
Apply a function to both sides	`apply_fun`
Move between ℕ, ℤ, ℚ and ℝ	`lift`, `zify`, `qify`, `rify`, `push_cast`
Remove a quantifier on both sides	`peel`
Run one tactic on several goals	`<;>`, `all_goals`, `any_goals`
Work on one goal	`·`, `case`, `next`, `on_goal`, `pick_goal`, `swap`
Leave a hole	`sorry`
Search for a lemma	`exact?`, `apply?`, `rw?`, `#loogle`, `#leansearch`
Replace a slow or bare tactic by an explicit one	`simp?`, `aesop?`, `grind?`, `says`
See what is happening	`trace_state`, `show_term`, `extract_goal`
Test a statement for counterexamples	`plausible`, `decide`

Appendix B: Unicode Abbreviations

In a Lean file, type a backslash followed by an abbreviation, and VS Code replaces it with the symbol; these are the hundred or so symbols that mathematics in Lean uses most.

Typing and looking up a symbol

- The replacement happens as soon as the abbreviation is complete. Press **Tab** to replace an abbreviation that is a prefix of a longer one.
- Hover over any symbol in a Lean file: the tooltip lists every abbreviation that produces it.
- **Lean 4: Docs: Show Unicode Input Abbreviations**, in the command palette, lists all 1,865 abbreviations.
- **Lean 4: Input: Insert Unicode Symbol…**, bound to Ctrl+Alt+\ (Cmd+Alt+\ on macOS), searches by abbreviation or by symbol.
- The full list is the extension's file `abbreviations.json`.

Where a symbol has several abbreviations, the table gives the shortest and the most memorable.

Numbers

Symbol	Type	Meaning in Lean
ℕ	`\N` or `\nat`	Natural numbers, `Nat`
ℤ	`\Z`	Integers, `Int`
ℚ	`\Q`	Rational numbers
ℝ	`\R`	Real numbers
ℂ	`\C`	Complex numbers
ℝ≥0	`\nnreal`	`NNReal`, the real numbers ≥ 0
ℝ≥0∞	`\ennreal`	`ENNReal`, the interval [0, ∞]; the values of measures
𝕜	`\bbk`	The name Mathlib gives to a field of scalars, usually ℝ or ℂ
∞	`\infty`	The top element of `ℝ≥0∞` and similar types
π	`\pi`	`Real.pi`, after `open Real`
√	`\sqrt`	`Real.sqrt`
⌊ ⌋	`\lfloor`, `\rfloor`	Floor

Symbol	Type	Meaning in Lean
⌈ ⌉	`\lceil`, `\rceil`	Ceiling
∣	`\\|` or `\dvd`	Divides: `a | b`
≡	`\==`	Congruence: `a ≡ b [MOD n]`
↑	`\u`	A cast, as Lean displays it: `(↑n : ℝ)`

Logic and functions

Symbol	Type	Meaning in Lean
∀	`\forall` or `\all`	For all
∃	`\ex`	There exists. `∃!` is `\ex` followed by `!`.
¬	`\not` or `\neg`	Not
∧	`\and`	And
∨	`\or`	Or
→	`\to` or `\r`	Implication, and the type of functions
↔	`\iff`	If and only if
≠ ≤ ≥	`\ne`, `\le`, `\ge`	Not equal, less or equal, greater or equal
⊢	`\\|-`	Separates the hypotheses from the goal in the Infoview
⟨ ⟩	`\<` and `\>`, or `\<>` for both	Anonymous constructor: `⟨x, hx⟩`
·	`\.`	Focusing bullet; also the hole in `(· + 1)`
▸	`\t`	`h ▸ e`: transport `e` along the equality `h`
↦	`\mapsto`	`fun x ↦ x ^ 2`
λ	`\la` or `\fun`	The same as `fun`
∘	`\o` or `\circ`	Composition: `g ∘ f`
×	`\x` or `\times`	Product type, pairs
≃	`\equiv` or `\~-`	Equivalence of types: a bijection with its inverse
≅	`\iso`	Isomorphism in a category
⟶	`\hom`	Morphism in a category

Sets

Symbol	Type	Meaning in Lean
∈ ∉	`\in`, `\notin`	Membership
⊆	`\sub` or `\ss`	Subset
⊂	`\ssub`	Strict subset
⊇	`\supset`	Superset
∩ ∪	`\i` or `\cap`, `\un` or `\cup`	Intersection, union
⋃ ⋂	`\U` or `\bigcup`, `\I` or `\bigcap`	Indexed union and intersection: `⋃ i, s i`
⋃₀	`\U0`	Union of a set of sets
ᶜ	`\compl` or `\^c`	Complement: `sᶜ`
∅	`\empty`	Empty set
\	`\\`	Set difference: `s \ t`
Δ	`\symmdiff`	Symmetric difference
×ˢ	`\xs`	Product of sets: `s ×ˢ t`
⁻¹'	`\preim`	Preimage: `f ⁻¹' s`
''	Two apostrophes	Image: `f '' s`
𝒫	`\powerset`	Power set

Order

Symbol	Type	Meaning in Lean
⊤ ⊥	`\top`, `\bot`	Greatest and least element
⊓ ⊔	`\inf`, `\sup`	Binary infimum and supremum: `min` and `max` in a linear order
⨅ ⨆	`\Sup`, `\Inf`	Indexed supremum and infimum: `⨆ i, f i`

Analysis, topology and measure

Symbol	Type	Meaning in Lean
𝓝	`\nhds`	Neighbourhood filter: `𝓝 x`, `𝓝[s] x` (after `open Topology`)

Symbol	Type	Meaning in Lean
𝒫	`\MCP`	Principal filter: `𝒫 s`
𝒰	`\MCU`	Uniform structure
∀ᶠ	`\allf`	Eventually: `∀ᶠ x in l, P x`
∃ᶠ	`\exf`	Frequently
∀ᵐ	`\allm`	Almost everywhere: `∀ᵐ x ∂μ, P x`
∂	`\pd`	The measure in `∂μ`
∫	`\integral`	Integral: `∫ x in a..b, f x`
⨍	`\average`	Average value: `⨍ x in s, f x ∂μ`
∑ ∏	`\sum`, `\prod`	Finite sum and product: `∑ i ∈ s, f i`
∑'	`\sum` then `'`	Sum of a series: `∑' n, f n`
‖ ‖	`\norm` for both, `\\||` for one	Norm: `‖x‖`
‖ ‖₊	`\nnnorm`	Norm as an element of `ℝ≥0`
⟪ ⟫	`\<<`, `\>>`	Inner product: `⟪x, y⟫_ℝ`
≪	`\ll`	Absolute continuity of measures: `μ ≪ ν`
𝔼 ℙ	`\bbE`, `\bbP`	Expectation, probability (in `ProbabilityTheory`)

Algebra

Symbol	Type	Meaning in Lean
⁻¹	`\inv` or `\-1`	Inverse: `x⁻¹`
•	`\smul`	Scalar multiplication: `c • v`
/	`\quot` or `\/`	Quotient: `G ⧸ H`
⊗ ⊕	`\ox`, `\o+`	Tensor product, direct sum of two
⨁	`\O+`	Direct sum of a family
→ₗ	`\tol`	Linear map: `M →ₗ[R] N`
→L	`\to` then `L`	Continuous linear map: `E →L[𝕜] F`
→+*	`\to` then `+*`	Ring homomorphism

Symbol	Type	Meaning in Lean
ᵀ ᴴ	`\^T`, `\^H`	Transpose, conjugate transpose of a matrix
ᵒᵖ	`\op`	Opposite: `Rᵐᵒᵖ`, `Cᵒᵖ`

Brackets

Symbol	Type	Meaning in Lean
⟨⟩	`\<>`	Anonymous constructor; inserts both, cursor inside
‖‖	`\norm`	Norm; inserts both
⟦⟧	`\[[]]`	Class in a quotient: `⟦x⟧`
{⦃⦄}	`\{{}}`	Strict implicit argument: `⦃x⦄`
«»	`\f<<>>`	A name that contains spaces or symbols
⁅⁆	`\[-`, `\-]`	Lie bracket

Greek letters

The full names work too: `\alpha`, `\epsilon`, `\delta`.

α `\a`	β `\b`	γ `\g`	δ `\de`
ε `\e`	ζ `\ze`	η `\et`	θ `\th`
ι `\io`	κ `\ka`	λ `\la`	μ `\mu`
ν `\nu`	ξ `\xi`	π `\pi`	ρ `\rh`
σ `\si`	τ `\ta`	φ `\ph`	χ `\chi`
ψ `\ps`	ω `\om`	Γ `\G`	Δ `\D`
Σ `\S`	Π `\P`	Λ `\L`	Ω `\GO`

Subscripts and superscripts

₀ … ₉ `\0` … `\9`	ₙ `\_n`	ᵢ `\_i`	ⱼ `\_j`
ₖ `\_k`	ₘ `\_m`	² `\^2`	ⁿ `\^n`
ᶜ `\^c`	ᵐ `\^m`	ᶠ `\^f`	ᵀ `\^T`

Subscripts are ordinary letters of a name: `x₀`, `hε₁` and `δ₂` are three identifiers.

Traps

- `\int` gives ℤ. The integral sign is `\integral`.
- `\subset` gives ⊆, unlike in LaTeX. The strict inclusion ⊂ is `\ssub`.
- `∣` (`\|`, divides) and `|` (absolute value `|x|`, and the bar in patterns) are different characters.
- `∕` (`\/`, quotient) and `/` (division) are different characters.
- ² is not notation for a square: write `x ^ 2`.
- `λ` is reserved for functions, so it cannot name a variable.
- → (`\to`) builds functions; ⟶ (`\hom`) is a morphism in a category. Chapter 23 shows the clash this causes.
- ⁻¹' (`\preim`, preimage) and ⁻¹ (`\inv`, inverse) differ by one apostrophe.
- The bullet is · (`\.`), not the ASCII full stop. Mathlib's linter warns about the full stop.

Appendix C: Common Error Messages

This appendix lists more than a hundred messages you will meet, each with its usual cause and the remedy; every one was produced by Lean 4.35.0 and Mathlib v4.35.0 from a small failing example.

Names, numbers and expressions inside a message change with your code. The fixed words are the part to search for, here or on the Zulip chat.

How to read an error

- **Fix the first error first.** Later errors are often consequences: a syntax error makes Lean skip the rest of a declaration, and any failure inside a proof also produces `unsolved goals` at its `by`.
- **Read the goal printed with the message.** Most tactic errors end with the hypotheses and the goal at the point of failure. Compare that goal with the one you had in mind.
- **Look where the red line is.** Under a tactic, that tactic failed. Under `by` or under the name of the theorem, the proof as a whole is incomplete.
- **Follow the hints.** Many messages end with `Hint:` or `Try this:`. In VS Code, clicking the suggestion applies it.
- **Error codes.** On the command line a message may carry a code, as in `error(lean.unknownIdentifier)`. The code names the kind of error and is useful in a search.

Syntax: Lean cannot read the text

Message	Usual cause	What to do
`unexpected token ','; expected command`	A comma after a tactic, as in Lean 3: `intro h, exact h`	One tactic per line, or `;` between tactics
`unexpected token ','; expected '↦', '=>'`	`λ x, e` or `fun x, e`, as in Lean 3	`fun x => e` or `fun x ↦ e`
`Unknown identifier` `` `begin` `` , then `Invalid` `` `end`: There is no current scope to end ``	A Lean 3 proof between `begin` and `end`	`:= by`, then the tactics, indented
`unexpected token 'by'; expected ':=', 'where' or '|'`	`:=` missing before `by`	`theorem t : P := by`

Message	Usual cause	What to do
`unexpected identifier; expected command`	A line of the proof is not indented, so Lean reads it as a new command	Indent every tactic of a proof by the same amount, usually two spaces
`unexpected token 'in'; expected ','`	`∑ i in s, f i`, the old notation for finite sums	`∑ i ∈ s, f i`
`unexpected identifier; expected '['`	`rw h`: the brackets are missing	`rw [h]`
`Invalid syntax for induction tactic: The `with` keyword must be followed by a tactic or by an alternative` (or `for case analysis tactic`)	The Lean 3 forms `induction n with n ih` and `cases h with x hx`	`induction n with`, then one line per case (Chapter 9); `obtain ⟨x, hx⟩ := h`
`unexpected token 'λ'; expected '_' or identifier`	`λ` used as the name of a variable	Another name: `l`, `c` or `μ`
`expected token`	A character that Lean cannot read at that place, such as `ℝ≥0∞` before `open scoped ENNReal`. A square written `x²` gives `unexpected token '²'` instead	Add the missing `open`; write `x ^ 2` for a square
`unterminated comment`	A comment opened with `/-` and never closed with `-/`	Close it
`unexpected end of input; expected '}'`	A bracket opened and never closed	Close it; VS Code highlights the matching bracket
`unexpected token 'example'; expected term`	The previous declaration is unfinished, as in `example : ℕ := 1 +`	Finish the line above the error
`invalid 'import' command, it must be used in the beginning of the file`	An `import` after another command	Move all imports to the top of the file

Message	Usual cause	What to do
`Invalid name after `end`: Expected `Foo`, but found `Bar``	`section Foo` closed by `end Bar`	Use the same name; the hint offers the fix

Names, imports and options

Message	Usual cause	What to do
`Unknown identifier `foo``	A misspelled name, a lemma renamed in Mathlib, a namespace not opened, or a file not imported	Check the spelling; search the statement on Loogle; add the `open` or the `import`
`Unknown identifier `foo`` followed by `Note: It is not possible to treat `foo` as an implicitly bound variable here because it has multiple characters while the `relaxedAutoImplicit` option is set to `false`.`	A variable used in a statement and never declared	Declare it, `(foo : ℝ)`, or with `variable`
`Unknown constant `Real.sqrt_nonneg'``	A dotted name, or the argument of `#print axioms`, that names no declaration	As for `Unknown identifier`
`unknown namespace `Foo``	`open Foo`, with a misspelled namespace or one whose file is not imported	Check the name and the imports
`Invalid field `foo`: The environment does not contain `Nat.foo`, so it is not possible to project the field `foo` from an expression`	`x.foo` looks for a lemma `T.foo`, where `T` is the type of `x`, and there is none	Use the full name of the lemma, or check the type of `x`
`Invalid projection: Projections extract constructor fields for one-constructor inductive types`	`h.1` where `h` is not a conjunction, an `↔` or a structure	Hover over `h` to see its type

Message	Usual cause	What to do
`Ambiguous term` followed by `Possible interpretations:`	Two open namespaces define the same name, such as `Set.Icc` and `Finset.Icc`	Write the full name
`` `foo` has already been declared ``	A second declaration with an existing name	Rename it
`Unknown attribute` `` `[simpp]` ``	A misspelled attribute	`@[simp]`
`Unknown option` `` `maxHeartbeat` ``	A misspelled option	`maxHeartbeats`
`unknown tactic`	A misspelled tactic, a Mathlib tactic in a file that does not import Mathlib, or a Lean 3 tactic	Check the name in Appendix E; add `import Mathlib`
`unknown module prefix 'Mathlib'`	The file is open on its own, or VS Code is open on the wrong folder	**File > Open Folder** on the folder that contains `lakefile.toml` (Chapter 24)
`object file '…/Mathlib/Tactics.olean' of module Mathlib.Tactics does not exist`; from Lake, `bad import 'Mathlib.Tactics'`	A misspelled module name, or a module that is not compiled	Correct the name; for Mathlib, run `lake exe cache get`

Types and terms

Message	Usual cause	What to do
`Type mismatch h has type a = b but is expected to have type b = a`	The term proves a statement close to the goal, but not the goal	`h.symm`; `exact_mod_cast h` when casts differ; `convert h` to see the differences as goals

Message	Usual cause	What to do
`Type mismatch Real.log_pos has type 1 < ?m.10 → 0 < Real.log ?m.10 but is expected to have type 0 < Real.log x`	A lemma used without its hypotheses: each `?m` is an argument not supplied	Supply them: `Real.log_pos h`
`Application type mismatch: The argument h has type a ≤ b but is expected to have type … in the application …`	One argument of a function or lemma has the wrong type	Hover over the lemma to see its arguments and their order
`Function expected at f 1 2 but this term has type ℕ`	Too many arguments, or a hypothesis applied to a value when it is not a `∀` or an `→`	Count the arguments; check the type
`failed to synthesize instance of type class HAdd Nat (Option Nat) ?m.3`	An operation between two types that do not combine	Convert one side, or `match` on the `Option`
`failed to synthesize instance of type class OfNat (Sort ?u.1) 3 numerals are polymorphic in Lean, but the numeral `3` cannot be used in a context where the expected type is Sort ?u.1`	A numeral where Lean expects a type, a proposition, or a type without numbers	Check the statement; add a type ascription such as `(3 : ℝ)`
`failed to synthesize Fintype ℕ` (or another class and type)	The type has no instance of the class; here `fin_cases` on an infinite type	Use a tactic suited to the type, such as `interval_cases`; or add the class as an argument `[Fintype α]`
`typeclass instance problem is stuck` with `Note: Lean will not try to resolve this typeclass instance problem because the first and second type arguments to `HAdd` are metavariables.`	Lean does not know the types yet, as in `fun x => x + x`	Give the type: `fun (x : ℝ) => x + x`

Message	Usual cause	What to do
`don't know how to synthesize placeholder` (or `… for argument w`)	A `_` that Lean cannot fill	Give the term; in a tactic proof, write `?_` with `refine` to make it a new goal
`don't know how to synthesize implicit argument \`α\``	An implicit argument that nothing determines, as in `List.nil` alone	A type ascription: `([] : List ℕ)`
`type of theorem \`two\` is not a proposition`	`theorem` used for a piece of data	`def`
`failed to compile definition, consider marking it as 'noncomputable' because it depends on 'Real.sqrt', which is 'noncomputable'`	A definition uses something that cannot run: real numbers, choice, or an `if` on a real inequality (`Real.decidableLT`)	`noncomputable def`, or `noncomputable section` at the top of the file
`Insufficient number of fields for \`⟨...⟩\` constructor: Constructor \`And.intro\` has 2 explicit field, but only 1 was provided`	Fewer components than the structure has	`⟨hp, hq⟩`
`Invalid \`⟨...⟩\` notation: The expected type \`p\` is not an inductive type`	`⟨…⟩` where the goal is not a structure, or where Lean does not know the type yet	Check the goal; add a type ascription
`Fields missing: \`pos\``	A structure value without all its fields	Add them; the hint lists them
`Tactic \`cases\` failed with a nested error` and `recursor \`Exists.casesOn\` can only eliminate into \`Prop\``	Taking the witness out of an `∃` while building data, not a proof	`h.choose` and `h.choose_spec`, or the tactic `choose` (Chapter 11)

Definitions, patterns and recursion

Message	Usual cause	What to do
`Missing cases: (Nat.succ _)`	A `match`, or a definition by patterns, that does not cover every constructor	Add the missing case
`Redundant alternative: Any expression matching 1 will match one of the preceding alternatives`	A specific case placed after a case that catches everything	Put the specific cases first
`Invalid pattern variable: Variable name `n` was already used`	A pattern such as `n + n`: patterns contain constructors and new variables only	Match on `n`, and test the condition with `if`
`fail to show termination for r3 with errors failed to infer structural recursion`	Lean cannot see that the recursive argument decreases	Add `termination_by` with a measure, and `decreasing_by` if needed (Chapter 21)

Tactics that cannot start

Message	Usual cause	What to do
`Tactic `rfl` failed: The left-hand side a + b is not definitionally equal to the right-hand side b + a`	The equality holds by a theorem, not by computation. Real numerals behave the same way: `(2 : ℝ) + 2 = 4`	`ring`, `norm_num`, `omega` or `simp`
`Tactic `assumption` failed`	No hypothesis is exactly the goal. Lean may add `Try this:` with a short proof	Click the suggestion, or use `exact` with a lemma
`Tactic `apply` failed: could not unify the type of `le_of_lt h` a ≤ b with the goal a ≤ b + 1`	The conclusion of the lemma does not match the goal	`exact?`; another lemma; a `calc`

Message	Usual cause	What to do
`Tactic `introN` failed: There are no additional binders or `let` bindings in the goal to introduce`	`intro` when the goal is not a `∀`, an `→` or a `¬`	Check the goal; `unfold` the definition that hides the `∀`
`Tactic `constructor` failed: target is not an inductive datatype`	`constructor` on a goal such as `0 ≤ x`	A lemma, or `linarith`, `positivity`
`Tactic `cases` failed: major premise type is not an inductive type`; the same with `induction`	`cases` on a real inequality, or `induction` on a function	`rcases lt_or_ge a b with h | h`, `by_cases`, or an induction principle with `using`
`Tactic `rcases` failed: `h : Real.le↑ a b` is not an inductive datatype`	`obtain ⟨…⟩ := h` where `h` is an inequality, which has no components	Use `h` as it is
`'specialize' requires a term of the form `h x_1 .. x_n` where `h` appears in the local context`	`specialize h 1` where `h` is not a `∀` or an `→`	Check the type of `h`; `have h' := …`
`Tactic `subst` failed: invalid equality proof, it is not of the form (x = t) or (t = x)`	Neither side of the equation is a variable	`rw [h]`
`'change' tactic failed, pattern a ≤ b is not definitionally equal to target a + 0 ≤ b`; the same with `'show' tactic failed`	The two statements are equal by a lemma, not by definition	`rw`, `simp only [add_zero]`, or `suffices`
`Tactic `unfold` failed to unfold `myF` in x = x`	The definition does not occur in the goal	Check the goal; `unfold myF at h` for a hypothesis
`Tactic `split_ifs` failed: no if-then-else conditions to split`	No `if` in the goal	`split` for a `match`; unfold the definition first so the `if` appears

Message	Usual cause	What to do
`Tactic `contrapose` failed: the goal `0 < x` is not of the form `_ → _` or `_ ↔ _``	`contrapose` needs an implication	`contrapose! h`, which uses the hypothesis `h`
`Tactic `contradiction` failed`	No two hypotheses contradict each other directly	`linarith`, `omega`, `simp_all`, or `exact absurd h1 h2`
`invalid 'calc' step, left-hand side is b : ℝ but is expected to be a : ℝ`	The chain does not start with the left side of the goal	Start with the left side of the goal
`Tactic `rewrite` failed: Did not find an occurrence of the pattern f ?n in the target expression ∑ i ∈ Finset.range 5, f i = 0`	The left side of the equation does not occur as written: it is under `∑`, `∀` or `fun`, or `nth_rw` asks for an occurrence that does not exist	`simp only [h]`, `simp_rw [h]` or `conv`; check the exact form of the goal
`Invalid rewrite argument: Expected an equality or iff proof or definition name, but `h` is a proof of a < b`	`rw` with an inequality	`calc`, `gcongr`, `grw` or `linarith`
`Tactic `rewrite` failed: motive is not type correct`	Another part of the goal depends on the term being rewritten, such as `v : Fin n` when `n` is rewritten	`subst`, `simp` or `conv`; or rewrite before the dependent term appears
`` `exact?` could not close the goal. Try `apply?` to see partial suggestions. ``	No single lemma closes the goal	`apply?`; split the goal with `have`

Automation that gives up

Message	Usual cause	What to do
`linarith failed to find a contradiction`	The goal needs a product of variables, subtraction in ℕ, a cast, or a fact not in the context	`nlinarith [sq_nonneg (x - y)]`; `zify` or `push_cast` first; state the missing fact with `have`
`ring failed, ring expressions not equal`, from `linear_combination`	Wrong coefficients: the goal printed is what remains to be proved	Correct the coefficients until the printed goal is `0 = 0`
`The ``ring`` tactic failed to close the goal. Use ``ring_nf`` to obtain a normal form.`, then `unsolved goals`	Not an identity of commutative rings, or a hypothesis is needed	`linear_combination h`; `field_simp` first for divisions
`` `ring_nf` made no progress on the goal ``	`ring` in a structure that is not a commutative ring, such as a group	`group`, `abel` or `noncomm_ring`
`` `field_simp` made no progress on the goal ``	There is no division to clear	`ring`
`failed to prove positivity/nonnegativity/nonzeroness`	One part has no known sign, such as `x - 1`	Prove the sign of that part with `have`, then `positivity`; or `nlinarith`
`not a positivity goal`	The goal is not of the form `0 ≤ e`, `0 < e` or `e ≠ 0`	Move everything to one side; or `nlinarith`

Message	Usual cause	What to do
`` `gcongr` did not make progress. ``	The two sides do not have the same shape	Rewrite one side so that the terms match pairwise
`omega could not prove the goal: a possible counterexample may satisfy the constraints`	The goal is false, needs a product of variables, or needs a fact about a function	Read the counterexample; add facts with `have`
`omega could not prove the goal: No usable constraints found.`	The goal is about $\mathbb{R}$, or a definition hides the arithmetic	`linarith` for $\mathbb{R}$; `unfold` first
`Tactic `decide` proved that the proposition 2 + 2 = 5 is false`	The statement is false	Correct it
`Tactic `decide` failed to reduce … to `true` or `false`.` with `reduction got stuck at Classical.choice`	`decide` on real numbers, or on a property defined with classical logic	`norm_num`
`Expected type must not contain free variables` with `Hint: Use the `+revert` option`	`decide` on a goal with variables	`decide +revert`; or `interval_cases n`, then `decide`
`Tactic `tauto` failed`	The goal is not a tautology of propositional logic	Supply the missing facts first
`Tactic `aesop` failed, made no progress`, or `… failed to prove the goal after exhaustive search.` (also printed by `bound`)	Its rules do not reach the goal	A more specific tactic; `aesop?` shows the attempt

Message	Usual cause	What to do
`` `grind` failed `` followed by `[grind] Goal diagnostics`	`grind` cannot close the goal; the diagnostics list the facts it derived	Read the facts; give lemmas: `grind [foo]`
`` `fun_prop` was unable to prove `Continuous fun x ↦ 1 / x` `` with `Failed to prove necessary assumption` `` `∀ (x : ℝ), x ≠ 0` ``	A side condition fails, or the property is false	Prove the side condition; or state `ContinuousOn` on a set
`No applicable extensionality theorem found for type ℝ`	`ext` on an equality of numbers	`ext` is for functions, sets and structures
`interval_cases failed: could not find upper bound on n`	No hypothesis bounds the variable	Put the bound in the context
`mod_cast has type n < 3 but is expected to have type n < 2`	Once the casts are removed, the hypothesis does not prove the goal	Check the arithmetic
`` `simp` made no progress ``	No simp lemma applies; often a hypothesis is missing	`simp [hx]`; `simp?`
`unsolved goals`, with the goal unchanged by `norm_num` (if a later tactic closes it, you get the warning `Unused tactic linter: `norm_num` does nothing.` instead)	`norm_num` uses hypotheses only when they are given	`norm_num [h]`

Message	Usual cause	What to do
`unsolved goals ⊢ False` after `norm_num` or `simp`	The statement is false. Here it was `Nat.Prime 1001`, and $1001 = 7 \cdot 11 \cdot 13$	Correct the statement
`` `push Not` made no progress on the goal ``	There is no negation to push	Remove the tactic

Goals left over

Message	Usual cause	What to do
`unsolved goals` followed by the goals	The proof ended with goals remaining. The error sits at `by`, or at the start of a `·` block	Put the cursor at the end of the proof and read the goals
`No goals to be solved`	A tactic after the proof is complete	Delete it
`unsolved goals ⊢ x ≠ 0` after `rw [div_self]`	The hypotheses of a rewriting lemma become new goals	`rw [div_self hx]`, or prove the new goal
`unsolved goals case calc.step ⊢ c + 1 < c`	The `calc` chain ends at another term than the goal, and Lean asks for the missing step	End the chain at the right side of the goal
`unsolved goals case ha ⊢ 0 ≤ c` after `gcongr`	A side condition that `gcongr` could not prove	Prove it after `gcongr`, or put `hc : 0 ≤ c` in the context
`Type mismatch 1 has type ℝ but is expected to have type ℕ`, then `unsolved goals case h ⊢ sorry = 1`	`use (1 : ℝ)` where a natural number is required	`use 1`

Time and memory

Message	Usual cause	What to do
`(deterministic) timeout at `whnf`, maximum number of heartbeats (200000) has been reached`	One declaration needs too much work; the word after `timeout at` varies	Measure with `#count_heartbeats in`, then see Chapter 25
`maximum recursion depth has been reached` with `use `set_option maxRecDepth <num>` to increase limit`	`decide` or `rfl` on a large computation, or a `simp` set that loops	`norm_num`, `simp` or `decide +kernel`; find the loop with `simp?`

Programs, projects and metaprograms

These are explained in the chapters cited.

Message	Usual cause	What to do
`failed to prove index is valid, possible solutions:`	`xs[i]` without a proof that `i < xs.size` (Chapter 21)	`xs[i]!`, `xs[i]?`, or prove the bound
`could not synthesize a `Repr` or `ToString` instance for type Nat → Nat`	`#eval` of a value that cannot be printed, here a function	Apply it to an argument; add `deriving Repr` to your type
`invalid `do` notation, expected type is not a monad application`	`do` in a function that returns a plain value	`Id.run do`
`Variable `s` cannot be mutated. Only variables declared using `let mut` can be mutated.`	Assignment to a variable declared with `let`	`let mut`
`PANIC at List.get!Internal … invalid index`	`xs[i]!` out of bounds when the program runs	Check the bound, or use `xs[i]?`
`Imports are out of date and must be rebuilt`	A file imported by this one has changed (Chapter 24)	**∀ > Server: Restart File**
`Not running `lake exe cache get` yet, as the `lake` version (…) does not match the toolchain version (…)`	`lake update` changed `lean-toolchain`	Run `lake exe cache get` yourself

Message	Usual cause	What to do
`Unknown identifier `getMainGoal``	A metaprogram without the namespaces open (Chapter 23)	`open Lean Elab Tactic Meta`
`Tactic `…` has not been implemented`	The `elab` that defines the tactic did not compile	Fix the first error in the `elab`
`(kernel) declaration type mismatch`	A tactic of your own built a term of the wrong type	Check the term with `inferType` before assigning it

Warnings

A warning does not stop the file from compiling. The first two concern correctness; the others concern style, and come from Mathlib's linters in a `math` project (Chapter 25).

Warning	Meaning	What to do
`declaration uses `sorry``	The proof has a hole	Fill it before you rely on the theorem
`` `Set.mem_setOf_eq` has been deprecated: Use `Set.mem_ofPred_eq` instead ``	The lemma was renamed	Click **Replace the deprecated name**
`` `push_neg` has been deprecated. Prefer using `push Not` instead. ``	The tactic was renamed	`push Not`
`Variable name `h` is not explicitly referenced.`	A hypothesis is never used	Delete it, or name it `_h`
`Unused tactic linter: `skip` does nothing.`	A tactic changed nothing	Delete it
`Tactic `constructor` applied constructor `Or.inl`, but `Or.inr` also matches the goal.`	`constructor` on an `∨`	`left` or `right`
`` `simp` is a flexible tactic modifying `⊢`. Try `simp?` and use the suggested `simp only [...]`. ``	A bare `simp` followed by other tactics	Use the `simp only` that `simp?` prints

Warning	Meaning	What to do
`The `show` tactic should only be used to indicate intermediate goal states for readability. However, this tactic invocation changed the goal. Please use `change` instead for these purposes.`	`show` changed the goal up to definition	`change`
`The following tactic starts with 2 goals and ends with 1 goal, 1 of which is not operated on.`	A tactic ran while several goals were open	A `·` bullet for each goal
`This line exceeds the 100 character limit, please shorten it!`	A long line	Break it; continuation lines are indented by 4 spaces
`Please, use '·' (typed as `\.`) instead of '.' as 'cdot'.`	A bullet typed as a full stop	Type `\.`
`Please, add a comment explaining the need for modifying the maxHeartbeat limit`	`set_option maxHeartbeats` without a reason	Add a comment line below it
`Using `native_decide` is not allowed in mathlib`	The proof trusts the compiler (Chapter 22)	`decide`, `norm_num` or `decide +kernel`
`unclosed sections or namespaces; expected: 'end Foo'`	A `section` or `namespace` without its `end`	Add `end Foo`

Appendix D: Glossary

The words of Lean and Mathlib used in this book, in alphabetical order, each with the chapter that treats it when there is one.

Term	Meaning
Anonymous constructor	The notation `⟨a, b⟩`, which builds a value of a structure without naming its constructor: a pair, a proof of `p ∧ q`, or a proof of `∃ x, p x` from a witness and a proof
Auto-bound implicit	A single letter used in a statement without being declared becomes an implicit argument. With `relaxedAutoImplicit = false`, as in a `math` project, longer names are errors (Chapter 24)
Axiom	A statement accepted without proof. Mathlib uses three: `propext`, `Classical.choice` and `Quot.sound`. `#print axioms` lists those a theorem depends on (Chapter 22)
Binder	The part of `∀ x,`, `∃ x,`, `fun x =>` or `(x : α)` that introduces a variable. The brackets give its kind: `( )` explicit, `{ }` implicit, `[ ]` instance, `⦃ ⦄` strict implicit
Blueprint	A document listing the statements of a formalization project with their dependencies and their state (Chapter 25)
Cast	A coercion between number types, displayed `↑`: the natural number `n` seen as a real number is `(↑n : ℝ)` (Chapter 14)
Class	A structure whose values Lean finds automatically, such as `Field ℝ` or `TopologicalSpace X` (Chapter 10)
Classical	Said of reasoning that uses the excluded middle or the axiom of choice. Mathlib uses it freely; a definition that depends on it is `noncomputable`
Coercion	A conversion of a term to another type, inserted automatically by Lean and displayed `↑`
Constructor	One of the ways of building an element of an inductive type: `Nat.zero` and `Nat.succ`; `Or.inl` and `Or.inr` (Chapter 9)

Term	Meaning
Decidable	The class of propositions whose truth value Lean can compute. `decide` and `if` use it (Chapters 11 and 13)
Declaration	A named object: a definition, a theorem, a structure, an instance or an axiom
Definitional equality	Equality that Lean recognizes by unfolding definitions and computing, as `2 + 2 = 4` in ℕ. It is proved by `rfl`. Often shortened to *defeq*
Dependent type	A type that depends on a value, such as `Fin n`. A dependent function returns, for each input, a value in a type that may depend on that input (Chapter 11)
Deprecated	Said of a declaration that was renamed: the old name works for some months, with a warning that names the new one (Chapter 24)
Elaboration	The process that turns what you type into a complete term, by filling in types, implicit arguments, instances and coercions (Chapter 23)
elan	The program that installs versions of Lean and selects the right one for each project (Chapter 2)
Environment	All the declarations available at a point of a file: those of the imported files, and those above the point
Eventually	`∀ᶠ x in l, P x`: the set where `P` holds belongs to the filter `l`. With `l = atTop` on ℕ, it means for all `n` large enough (Chapter 17)
Extensionality	The principle that two functions, sets or structures are equal when their values, elements or fields are equal. The tactic is `ext`
Field notation	`x.foo`, which means `T.foo x` where `T` is the type of `x`: `h.le`, `hf.continuous`. Also called dot notation (Chapters 5 and 12)
Filter	A collection of sets regarded as large, used to state every kind of limit at once: `atTop`, `𝓝 x`, `𝓝[≠] x` (Chapter 17)
Goal	A statement that remains to be proved. The Infoview shows it after `⊢`, with the hypotheses above it

Term	Meaning
Heartbeat	Lean's unit of work. A declaration may use 200000 by default (Chapter 25)
Hypothesis	A named fact available in the current proof: an assumption of the theorem, or a fact proved on the way
Implicit argument	An argument in braces `{ }`, which Lean deduces from the other arguments. `@f` makes all the arguments of `f` explicit
Import	A line `import M` at the top of a file, which makes the declarations of the module `M` available (Chapter 24)
Inductive type	A type defined by its constructors, such as `ℕ`, `List α`, `Or` or `Exists`. Proofs about it proceed by cases and by induction (Chapter 9)
Infoview	The VS Code panel that shows the goals at the cursor and the messages (Chapter 3)
Instance	A value of a class that Lean uses automatically: a declaration `instance : Field ℝ`, or an argument `[Group G]` (Chapter 10)
Junk value	The value Mathlib gives to an expression that mathematics leaves undefined, such as `x / 0 = 0`. Every function in Lean is defined everywhere (Chapters 14 and 26)
Kernel	The small program at the core of Lean that checks every proof. Tactics, automation and AI tools work outside it and need not be trusted (Chapters 22 and 23)
Lake	Lean's build tool and package manager (Chapter 24)
Lemma	A synonym of `theorem`, defined by Mathlib
Linter	An automatic check that warns about style or likely mistakes, such as an unused hypothesis (Chapter 25)
Local context	The variables and hypotheses available at a point of a proof
Manifest	The file `lake-manifest.json`, which records the exact commit of every dependency (Chapter 24)

Term	Meaning
Mathlib	The library of mathematics for Lean, written and maintained by its community
Metavariable	A placeholder for a term not yet known, displayed `?m.12` or `?a`. A goal is a metavariable that the proof must fill (Chapter 23)
Module	A Lean file. Its name is its path: `Mathlib/Analysis/Real/Sqrt.lean` is the module `Mathlib.Analysis.Real.Sqrt` (Chapter 24)
Monad	A type such as `IO` or `Option` whose values are computations that can be chained in a `do` block (Chapter 21)
Motive	The statement `fun x => P x` that an induction proves for every `x`, or that a rewrite carries along an equation. It appears in the error `motive is not type correct`
Namespace	A prefix that groups names: `Real.sqrt` is `sqrt` in the namespace `Real`. After `open Real` you may write `sqrt`
Noncomputable	Said of a definition that Lean cannot run, because it uses real numbers or the axiom of choice. Proofs about it are not affected
Notation	A symbol that stands for an expression, such as ∑ or ‖x‖ (Chapter 23)
Pattern matching	Defining a function, or proving a statement, by giving one case for each constructor, with `match` or with `|` lines (Chapter 9)
Prop	The type of propositions. A proof of `p` is a term of type `p`
Proof irrelevance	The rule that two proofs of the same proposition are equal by definition
Proof term	The term that a proof builds, and that the kernel checks. `show_term` displays the term built by a tactic
Propositional equality	The proposition `a = b` when it is proved by a theorem rather than by definition: `a + b = b + a` is proved by `add_comm`

Term	Meaning
Quotient	`Quotient s` (or `Quot r`), the type of classes of an equivalence relation; `⟦x⟧` is the class of `x` (Chapter 11). Algebraic quotients are written `G ⧸ H` (Chapter 16)
Recursor	The induction principle generated for an inductive type, such as `Nat.rec`. `induction` and `match` are built on it
Section	The part of a file between `section` and `end`, which limits the scope of `variable`, `open` and options
Simp lemma	A lemma tagged `@[simp]`, which `simp` uses to rewrite from left to right (Chapter 13)
sorry	A placeholder that proves anything, accepted with a warning. A theorem that uses it depends on the axiom `sorryAx` (Chapter 25)
Structure	An inductive type with a single constructor and named fields, such as a pair or the data of a group (Chapter 10)
Subtype	`{ x : α // p x }`: the elements of `α` that satisfy `p`, each with its proof (Chapters 11 and 22)
Tactic	An instruction that transforms the goal, in a proof written after `by`. The script of tactics builds a proof term (Appendix E)
Term	Any expression: a number, a function, a proof, a type. Every term has a type, which `#check` displays
Term mode	Writing a proof as a term, without `by`: `fun h => h.1`
Termination	A recursive definition must end on every input. Lean checks it by structural recursion, or by a measure given with `termination_by` (Chapter 21)
Toolchain	A version of Lean. Each project names its toolchain in the file `lean-toolchain` (Chapters 2 and 24)
Transparency	How far Lean unfolds definitions when it compares two terms. `rw` unfolds only reducible definitions; `erw` and `rfl` unfold more
Type	What classifies a term. Types are terms too: `ℕ : Type`, `Type : Type 1`, and `p : Prop` for a proposition `p`

Term	Meaning
Type ascription	`(e : T)`, which tells Lean the type intended for `e`: `(2 : ℝ)`
Type class	Another name for a class
Unification	Finding values for metavariables that make two expressions equal. `apply`, `rw` and `exact` unify a lemma with the goal
Universe	A level in the hierarchy `Type : Type 1 : Type 2 : ...`, which avoids the paradox of a type of all types. `Sort u` and `Type u` name them (Chapter 11)
`variable`	A command that declares arguments shared by the declarations that follow it, up to the end of the section
Weak head normal form	The form of an expression after computing just enough to expose its outermost constructor or function. `whnf` computes it (Chapter 23)
Witness	The element given to prove `∃ x, P x`
Zulip	The chat where the Lean community meets (Chapter 27)

Appendix E: Every Tactic, Alphabetically

This is the complete list of the 408 tactics available in a file that begins with `import Mathlib`, for Lean 4.35.0 and Mathlib v4.35.0, taken from Lean's own index of tactics; the 60 tactics used only internally are named at the end.

- **The name of each tactic links to its definition** in the source code, at the exact version used in this book. The documentation of the tactic is the comment that starts at that line, or the one just above it.
- **Name means** gives what the name abbreviates or refers to. For a plain English word, it gives the sense in which the word is meant.
- **★ marks the tactics to learn first.** Appendix A says which tactic suits which goal.
- In VS Code, hovering over a tactic shows its full documentation. The documentation site has a page of all tactics, for the newest Mathlib.
- A name ending in `?` usually runs the tactic and prints a more explicit call that you can paste in its place. A name ending in `!` is usually a stronger variant.

Symbols and commands

Tactic	Name means	What it does
`(tacs)`	Parentheses	Runs a sequence of tactics as a single tactic: `(rw [h]; simp)`
`·` ★	The focusing dot, typed `\.`	Works on the first goal only, and fails unless the tactics below it close that goal
`<;>` ★	Then, on every goal	`t <;> s` runs `t`, then `s` on every goal that `t` produced
`_`	A hole	Like `done`: fails and lists the goals that remain
∎	QED, typed `\qed`	Runs `try?`
`#adaptation_note`	Adaptation note	Records a comment about a workaround needed after a Lean update
`#check`	Check	Prints a term and its type, from inside a proof
`#check'`	Check, explicit arguments only	Like `#check`, but omits implicit and instance arguments

Tactic	Name means	What it does
`#count_heartbeats`	Count heartbeats	Reports the work, in heartbeats, used by the tactics that follow it
`#defeq_abuse`	Definitional-equality abuse	Experimental: finds proofs that depend on unfolding definitions too far
`#find`	Find	Lists declarations whose conclusion matches a pattern such as `_ ∧ _`
`#leansearch`	LeanSearch	Sends a question in English to LeanSearch and prints the lemmas found
`#loogle`	Loogle	Sends a pattern to Loogle and prints the lemmas found
`#search`	Search	Searches with the engine chosen by the option `leansearchclient.backend`
`#statesearch`	State search	Sends the current goal to LeanStateSearch and prints lemmas that may apply

A

Tactic	Name means	What it does
`abel`	Abelian, after Niels Henrik Abel	Proves equations in commutative additive monoids and groups
`absurd`	Absurdity	With `h : p`, `absurd h` changes the goal to `¬p`
`ac_change`	Change, up to associativity and commutativity	Replaces the goal by one equal to it after reordering sums and products
`ac_nf`	Associative–commutative normal form	Normalizes expressions built from associative and commutative operations

Tactic	Name means	What it does
`ac_rfl`	Reflexivity up to associativity and commutativity	Proves equations such as `a + b + c = c + (b + a)`
`add_group`	Additive group	Proves equations in additive groups, without using commutativity
`admit`	Admit the goal without proof	A synonym of `sorry`
`aesop` ★	Automated Extensible Search for Obvious Proofs	Searches for a proof with a set of rules, by best-first search
`aesop?`	Aesop, and show the proof	Like `aesop`, then prints the script it found
`aesop_cat`	Aesop for categories	`aesop` with the rules of category theory
`aesop_cat?`	Aesop for categories, and show the proof	Like `aesop_cat`, then prints the script
`aesop_mat`	Aesop for matroids	Proves that a set lies in the ground set of a matroid
`algebra`	Algebra over a ring	Proves equations in a commutative algebra, with ring operations and scalar multiplication
`algebraize`	Make into an algebra	Turns ring homomorphisms in the context into `Algebra` instances
`algebraize_only`	Algebraize, instances only	Like `algebraize`, without the extra properties
`all_goals`	On all goals	Runs a tactic on every goal, and fails if it fails on one
`and_intros`	`And.intro`, repeated	Splits a goal `p ∧ q ∧ …` into one goal per part

Tactic	Name means	What it does
`any_goals`	On any goal	Runs a tactic on every goal, and succeeds if it succeeds on one
`apply` ★	Apply a lemma	Proves the goal with a lemma whose conclusion matches it; the hypotheses of the lemma become new goals
`apply … at`	Apply, forwards	`apply f at h` replaces `h : p` by `f h : q`
`apply?` ★	Apply, with a search	Lists the lemmas that `apply` could use on the goal
`apply_assumption`	Apply an assumption	Applies a hypothesis whose conclusion matches the goal
`apply_fun`	Apply a function to both sides	Turns `h : a = b` into `f a = f b`; on a goal `a = b`, reduces it to `f a = f b` for an injective `f`
`apply_mod_cast`	Apply, modulo casts	`apply` after normalizing the casts
`apply_rewrite`	Rewrite by applying implications	Uses implications `p → q` as rewriting rules, replacing `p` by `q`
`apply_rfl`	Apply a reflexivity lemma	Like `rfl`, without its final attempt with `Eq.refl`
`apply_rules`	Apply the rules, repeatedly	Applies the given lemmas again and again until the goals are closed
`apply_rw`	Apply, as a rewrite	The `grw` form of `apply_rewrite`
`arith_mult`	Arithmetic function, multiplicative	Proves that an arithmetic function is multiplicative
`arith_mult?`	The same, and show the proof	Like `arith_mult`, then prints the proof

Tactic	Name means	What it does
`assume`	Assume	`assume p` introduces an unnamed hypothesis of type `p`
`assumption` ★	Use an assumption	Closes the goal with a hypothesis of the same type
`assumption'`	Assumption, on all goals	Runs `assumption` on every goal, and succeeds if it closes one
`assumption_mod_cast`	Assumption, modulo casts	`assumption` after normalizing the casts

B

Tactic	Name means	What it does
`basify`	Move to the base type	Turns a goal about `ℕ∞`, `ℕ+`, `ℝ≥0` or `ℝ≥0∞` into one about `ℕ` or `ℝ`
`bdsimp`	Backward-compatible `dsimp`	A variant of `dsimp` kept for older proofs
`beta_reduce`	β-reduction	Replaces `(fun x => e) a` by `e` with `a` in place of `x`
`bicategory`	Bicategory	Proves equations in a bicategory with the coherence theorem
`bicategory_coherence`	Coherence in bicategories	Proves equations between structural 2-isomorphisms
`bicategory_nf`	Bicategory normal form	Normalizes both sides of an equation in a bicategory
`borelize`	Make Borel	Equips a topological space with its Borel σ-algebra
`bound`	Bound	Proves inequalities by working through the structure of both sides
`bv_check`	Bit-vector check	`bv_decide` with a proof stored in a file, without calling the SAT solver

Tactic	Name means	What it does
`bv_decide`	Bit-vector decide	Proves goals about fixed-width `BitVec` and `Bool` with a SAT solver
`bv_decide?`	The same, and show the proof	Saves the SAT proof and suggests the corresponding `bv_check`
`bv_normalize`	Bit-vector normalize	Runs only the simplification step of `bv_decide`
`bv_omega`	Bit-vector omega	`omega`, after translating statements about `BitVec`
`by_cases` ★	By cases	`by_cases h : p` splits into a case with `h : p` and a case with `h : ¬p`
`by_cases!`	By cases, negation pushed	`by_cases`, then `push Not` in the second case
`by_contra` ★	By contradiction	Proves `p` by assuming `¬p` and deriving `False`
`by_contra!`	By contradiction, negation pushed	`by_contra`, then `push Not` on the new hypothesis

C

Tactic	Name means	What it does
`calc` ★	Calculation	Proves a relation through a chain of steps `a = b`, `b ≤ c`, …, each with its own proof
`calc?`	Calculation, with a widget	Builds a `calc` proof from terms selected in the goal
`cancel_denoms`	Cancel the denominators	Multiplies out numerical denominators in equations and inequalities
`case`	Case	`case tag => tac` works on the goal named `tag`, and must close it
`case'`	Case, variant	Like `case`, without requiring that the goal be closed

Tactic	Name means	What it does
`cases` ★	Cases	Splits a value of an inductive type into one goal per constructor
`cases'`	Cases, older syntax	`cases` with the new names given as a list, as in Lean 3
`cases_type`	Cases on a type	Splits the hypotheses whose type is the given inductive type
`casesm`	Cases, on a match	Splits the hypotheses whose type matches a pattern such as `_ ∧ _`
`cbv`	Call by value	Evaluates the goal as a program would, unfolding definitions by their equations
`change` ★	Change	Replaces the goal by a statement equal to it by definition
`change?`	Change, and show the call	Prints a `change` that states the current goal explicitly
`check_compositions`	Check the compositions	Checks that the objects agree in each composition `f ≫ g` of the goal
`choose`	Choose, as in the axiom of choice	From `h : ∀ x, ∃ y, P x y`, obtains a function `f` and `hf : ∀ x, P x (f x)`
`classical`	Classical logic	Runs tactics with every proposition treated as decidable
`clean`	Clean up a term	Deprecated: `exact` with a term cleaned of auxiliary annotations
`clear`	Clear	Removes hypotheses from the context
`clear * -`	Clear all, except	`clear * - h₁ h₂` removes every hypothesis except those listed
`clear!`	Clear, with what depends on it	Removes hypotheses together with those that depend on them
`clear_value`	Clear the value	Forgets the value of a local definition `x := v`, and keeps `x`
`closedness`	Closedness	Proves that a set is closed, with `grind` and a list of lemmas

Tactic	Name means	What it does
`closedness?`	The same, and show the call	Like `closedness`, then prints an explicit `grind` call
`coherence`	The coherence theorem	Proves equations in a monoidal category or a bicategory
`compactness`	Compactness	Proves that a set is compact, with `grind` and a list of lemmas
`compactness?`	The same, and show the call	Like `compactness`, then prints an explicit `grind` call
`compute_degree`	Compute the degree	Proves `natDegree f ≤ d` or `degree f = d` for an explicit polynomial
`congr`	Congruence	Reduces `f a = f b` to `a = b`, recursively
`congr!`	Congruence, stronger	`congr` that also handles implicit arguments, instances and casts, and closes easy goals
`congrm`	Congruence, by matching a pattern	`congrm f ?_ (g ?_)` reduces the goal to equations at the holes of the pattern
`congrm?`	The same, with a widget	Builds a `congrm` pattern from subterms selected in the goal
`constructor` ★	Constructor	Proves the goal with a constructor of its type; the arguments become new goals
`constructor!`	Constructor, first match	`constructor` with the first constructor that fits, without a warning
`constructorm`	Constructor, on a match	Applies a constructor when the goal matches one of the patterns given
`continuity`	Continuity	Proves `Continuous f` with the lemmas tagged `@[continuity]`
`contradiction` ★	Contradiction	Closes the goal when two hypotheses contradict each other directly, as `h : p` and `h' : ¬p`

Tactic	Name means	What it does
`contrapose` ★	Contrapositive	Replaces `p → q` by `¬q → ¬p`; `contrapose!` also pushes the negations inside
`conv`	Conversion mode	Enters a mode that rewrites one chosen subterm of the goal
`conv'`	Conversion, variant	Runs conversion tactics without first turning the goal into a conversion goal
`conv?`	Conversion, with a widget	Builds the `conv` path to a subterm selected in the goal
`conv_lhs`	Conversion on the left-hand side	`conv` on the left side of the goal
`conv_rhs`	Conversion on the right-hand side	`conv` on the right side of the goal
`convert` ★	Convert	Closes the goal with a term whose type is close to it, and leaves the differences as goals
`convert_to`	Convert to	Replaces the goal by a given statement, and leaves the differences as goals
`cutsat`	The CUTSAT procedure: cuts and SAT	Deprecated synonym of `lia`

D

Tactic	Name means	What it does
`dbg_trace`	Debug trace	Prints a message when the tactic is elaborated
`decide` ★	Decide	Proves a decidable statement by evaluating it, such as `2 ^ 10 = 1024`
`decide_cbv`	Decide, by call-by-value	`decide`, evaluating by call-by-value rewriting
`decreasing_tactic`	Decreasing	The default tactic that proves that recursive calls decrease

Tactic	Name means	What it does
`decreasing_trivial`	Decreasing, easy cases	The extensible part of `decreasing_tactic` for simple cases
`delta`	δ-reduction	Unfolds definitions to their raw bodies, without equation lemmas
`discrete_cases`	Cases on `Discrete`	Runs `cases` on the hypotheses of a discrete category type
`done`	Done	Succeeds only if no goal remains
`dsimp`	Definitional `simp`	`simp` with lemmas that hold by definition only; the result is definitionally equal
`dsimp!`	`dsimp`, unfolding more	`dsimp` with automatic unfolding of definitions
`dsimp?`	`dsimp`, and show the call	Prints the equivalent `dsimp only […]`
`dsimp?!`	`dsimp!`, and show the call	Prints the equivalent `dsimp only […]`
`dyadic_interval`	Dyadic intervals	Proves numerical facts about real numbers by interval arithmetic with dyadic rationals
`dyadic_interval?`	The same, as a quick check	Checks quickly whether `dyadic_interval` would succeed

E

Tactic	Name means	What it does
`eapply`	Apply; *e* for existential variables, as in Coq	`apply`, without new goals for the variables that other goals determine
`econstructor`	Constructor; *e* for existential variables	`constructor`, with goals for the non-dependent arguments only
`enat_to_nat`	`ℕ∞` to `ℕ`	Moves the extended natural numbers in the context to `ℕ`
`eq_refl`	`Eq.refl`	The same as `exact rfl`, with some optimizations

Tactic	Name means	What it does
`erw`	Rewrite, *e* variant	`rw` that unfolds ordinary definitions to find the pattern
`erw?`	`erw`, and explain	Runs `erw`, then reports the subterm that prevents a plain `rw`
`eta_expand`	η-expansion	Replaces `f` by `fun x => f x`
`eta_reduce`	η-reduction	Replaces `fun x => f x` by `f`
`eta_struct`	η-reduction for structures	Replaces `⟨x.1, x.2⟩` by `x`
`eval_det`	Evaluate the determinant	Computes the determinant of a matrix written with `!![…]`
`eval_rank`	Evaluate the rank	Computes the rank of a matrix with numerical entries
`exact` ★	Exact	Closes the goal with a term of exactly its type
`exact?` ★	Exact, with a search	Searches the library for a lemma that closes the goal, and prints it
`exact_mod_cast` ★	Exact, modulo casts	`exact` after normalizing the casts in the goal and in the term
`exacts`	Exact, several	Closes the goals one after the other with the terms given
`exfalso` ★	*Ex falso quodlibet*: from a falsehood, anything	Replaces the goal by `False`
`exists`	There exists	`exists a` proves `∃ x, P x` with the witness `a`, then tries `trivial`
`existsi`	Exists, introduction	Gives the witnesses of an `∃`, and leaves the property as the goal
`expose_names`	Expose the names	Gives usable names to the inaccessible hypotheses shown as `h✝`
`ext` ★	Extensionality	Reduces an equation between functions, sets or structures to one for each value
`ext1`	Extensionality, one step	Applies a single extensionality lemma

Tactic	Name means	What it does
`extract_goal`	Extract the goal	Prints the goal as a standalone theorem
`extract_lets`	Extract the lets	Turns `let` expressions in the goal into local definitions

F

Tactic	Name means	What it does
`fail`	Fail	Always fails, with the message given
`fail_if_no_progress`	Fail if no progress	Runs tactics, and fails if they changed nothing
`fail_if_success`	Fail if success	Fails if the tactic succeeds; used in tests
`false_or_by_contra`	`False`, or by contradiction	Changes the goal to `False`, introducing the negation of the goal when needed
`fapply`	Apply, *f* variant	`apply`, with the new goals in the order of the arguments
`fconstructor`	Constructor, *f* variant	`constructor`, with the new goals in the order of the arguments
`field`	Field	Proves equations that hold in every field in which the denominators are nonzero
`field_simp` ★	Field simplification	Clears the denominators of equations and inequalities, using hypotheses that they are nonzero
`filter_upwards` ★	Filters are closed upwards	Reduces `∀ᶠ x in l, P x` to a statement about each `x`, using hypotheses that hold eventually
`fin_cases`	Finite cases	Splits into one goal per element of a finite type or a finite set
`fin_omega`	`omega` for `Fin`	Translates statements about `Fin n` so that `omega` can prove them
`find`	Find	Lists the declarations whose conclusion matches the goal exactly
`finiteness`	Finiteness	Proves `x < ∞` or `x ≠ ∞` in `ℝ≥0∞`

Tactic	Name means	What it does
`first`	First	`first | t₁ | t₂` runs the first tactic that succeeds
`focus`	Focus	Works on the first goal, with the other goals hidden
`fun_cases`	Cases, following a function	Splits into the cases of the definition of a function
`fun_induction` ★	Functional induction	Induction that follows the recursion of a function (Chapter 22)
`fun_prop` ★	Function property	Proves `Continuous f`, `Differentiable ℝ f`, `Measurable f` and similar, following the structure of `f`
`funext` ★	Function extensionality	Reduces `f = g` to `f x = g x` for an arbitrary `x`

G

Tactic	Name means	What it does
`gcongr` ★	Generalized congruence	Reduces `f a ≤ f b` to `a ≤ b`, following the common shape of both sides
`gcongr?`	The same, with a widget	Builds a `gcongr` pattern from subterms selected in the goal
`gconvert`	Generalized convert	Like `convert`, for relations such as `≤`
`generalize`	Generalize	Replaces an expression of the goal by a new variable
`generalize'`	Generalize, older form	Kept for older proofs; use `generalize`
`generalize_proofs`	Generalize the proofs	Replaces the proofs inside the goal by new hypotheses
`get_elem_tactic`	The tactic for getting an element	Proves that the index in `xs[i]` is in bounds; Lean calls it automatically

Tactic	Name means	What it does
`grewrite`	Generalized rewrite	Rewrites with a relation such as `≤` or `→`, where monotonicity allows it
`grind` ★	Grind, in the manner of SMT solvers	Combines congruence closure, arithmetic and lemma instantiation to close the goal
`grind?`	`grind`, and show the call	Prints an equivalent `grind only […]`
`grind_linarith`	`grind`, linear arithmetic only	Simple linear arithmetic, through `grind`
`grind_order`	`grind`, orders only	Simple goals about partial and linear orders, through `grind`
`grobner`	Gröbner bases	Proves polynomial equations from polynomial hypotheses, with Gröbner bases
`group`	Group	Proves equations in multiplicative groups, without using commutativity
`grw`	Generalized rewrite	`grewrite`, then tries `rfl`; rewrites with `a ≤ b` where monotonicity allows it
`guard_expr`	Guard an expression	Checks that two expressions are equal; used in tests
`guard_goal_nums`	Guard the number of goals	Checks the number of goals; used in tests
`guard_hyp`	Guard a hypothesis	Checks the type or value of a hypothesis; used in tests
`guard_hyp_nums`	Guard the number of hypotheses	Checks the number of hypotheses; used in tests
`guard_target`	Guard the target	Checks that the goal is a given expression; used in tests

H

Tactic	Name means	What it does
`have` ★	Have	States and proves an intermediate fact, then adds it as a hypothesis
`have'`	Have, in the style of `refine'`	`have` in which the holes `_` become new goals
`haveI`	Have, inlined	`have` whose value is substituted directly into the proof term; used for instances
`hint`	Hint	Tries a list of tactics and reports those that close the goal or make progress

I

Tactic	Name means	What it does
`if … then … else`	If, then, else	`if h : p then t₁ else t₂` splits into the cases `p` and `¬p`, like `by_cases`
`impossible`	Impossible	`impossible by t` proves with `t` that the goal is false, then closes it as `sorry` would
`inclusion`	Inclusion	Proves membership goals by reasoning with families of set inclusions
`inclusion?`	The same, as a quick check	Checks quickly whether `inclusion` would succeed
`induction` ★	Induction	Proves a statement for every element of an inductive type, with one case per constructor
`induction'`	Induction, older syntax	`induction` with the new names given as a list, as in Lean 3
`infer_instance`	Infer the instance	Closes a goal that is a class instance, by instance search
`inhabit`	Inhabited	Turns a hypothesis `Nonempty α` into an instance `Inhabited α`
`injection`	Injectivity of constructors	From `C a = C b` for a constructor `C`, derives `a = b`
`injections`	Injectivity, repeated	Applies `injection` to every hypothesis, recursively

Tactic	Name means	What it does
`interval_cases` ★	Cases over an interval	Splits into one goal per value of a bounded integer variable
`intro` ★	Introduce	Moves the premise of an `→`, a `∀` or a `¬` into the context, under the name given
`intros`	Introduce all	Introduces as many hypotheses as possible
`introv`	Introduce, keeping the variables' names	Introduces the variables under their names in the statement, and names the hypotheses as given
`itauto`	Intuitionistic tautology	Proves tautologies of intuitionistic propositional logic
`iterate`	Iterate	`iterate n t` runs `t` exactly `n` times

L

Tactic	Name means	What it does
`left` ★	Left	Proves `p ∨ q` by proving `p`; in general, uses the first of two constructors
`let`	Let	Adds a local definition `x := e`, whose value remains visible
`let rec`	Let, recursive	Adds a local recursive definition
`let'`	Let, in the style of `refine'`	`let` in which the holes `_` become new goals
`let_to_have`	Let, to have	Turns `let` expressions into `have` expressions where possible
`letI`	Let, inlined	`let` whose value is substituted directly; used for instances
`lia`	Linear integer arithmetic	Proves linear arithmetic over `ℕ` and `ℤ`, through `grind`
`lift`	Lift	`lift x to ℕ using hx` replaces an integer `x` with `hx : 0 ≤ x` by a natural number
`lift_lets`	Lift the lets	Moves the `let` expressions of the goal as far out as possible

Tactic	Name means	What it does
`linarith` ★	Linear arithmetic	Proves a linear equation or inequality from linear hypotheses, over an ordered field
`linarith?`	`linarith`, and show the hypotheses used	Like `linarith`, then prints a call with only the hypotheses it needed
`linear_combination` ★	Linear combination	Proves an equation or an inequality as a linear combination of hypotheses, checked by `ring`

M

The tactics whose names begin with `m` followed by the name of an ordinary tactic, such as `mintro` or `mexact`, are the versions of those tactics for the stateful goals of `Std.Do`, the framework that verifies programs with loops (Chapter 22).

Tactic	Name means	What it does
`map_tacs`	Map the tactics onto the goals	`map_tacs [t₁; t₂]` runs `t₁` on the first goal, `t₂` on the second, and so on
`massumption`	Assumption, in the stateful proof mode	`assumption` for a `Std.Do` goal
`match`	Match	Case analysis on expressions, with one branch per pattern
`match_scalars`	Match the scalars	Reduces an equation between linear combinations to equations between their coefficients
`mcases`	Cases, in the stateful proof mode	`rcases` for a `Std.Do` goal
`mclear`	Clear, in the stateful proof mode	`clear` for a `Std.Do` goal
`mconstructor`	Constructor, in the stateful proof mode	`constructor` for a `Std.Do` goal
`mdup`	Duplicate	Duplicates a hypothesis of a `Std.Do` goal

Tactic	Name means	What it does
`measurability`	Measurability	Proves `Measurable f`, `AEMeasurable f` or `MeasurableSet s` with the lemmas tagged `@[measurability]`
`measurability?`	The same, and show the proof	Like `measurability`, then prints the proof
`mexact`	Exact, in the stateful proof mode	`exact` for a `Std.Do` goal
`mexfalso`	Ex falso, in the stateful proof mode	`exfalso` for a `Std.Do` goal
`mexists`	Exists, in the stateful proof mode	`exists` for a `Std.Do` goal
`mframe`	Frame	Moves the pure hypotheses out of the stateful context of a `Std.Do` goal
`mhave`	Have, in the stateful proof mode	`have` for a `Std.Do` goal
`mintro`	Introduce, in the stateful proof mode	`intro` for a `Std.Do` goal
`mleave`	Leave the proof mode	Leaves the stateful proof mode and simplifies, usually right after `mvcgen`
`mleft`	Left, in the stateful proof mode	`left` for a `Std.Do` goal
`mod_cases`	Cases modulo n	`mod_cases h : e % 3` splits into the cases `e ≡ 0`, `1` and `2 [ZMOD 3]`
`module`	Module	Proves equations between linear combinations in a module, with the coefficients checked by `ring`
`module_nf`	Module normal form	Normalizes the linear combinations `a • x + … + b • y` of the goal
`monicity`	Monic	Proves that a polynomial is monic
`mono`	Monotonicity	Proves inequalities by applying monotonicity lemmas and hypotheses repeatedly

Tactic	Name means	What it does
`monoidal`	Monoidal	Proves equations in a monoidal category with the coherence theorem
`monoidal_coherence`	Coherence in monoidal categories	Proves equations between structural isomorphisms
`monoidal_nf`	Monoidal normal form	Normalizes both sides of an equation in a monoidal category
`move_oper`	Move an operand	Moves the given terms to the right end of a sum or a product; its usual forms are `move_add` and `move_mul`
`mpure`	Pure	Moves a pure hypothesis out of the stateful context
`mpure_intro`	Pure, introduction	Proves a stateful goal `⌜φ⌝` by proving `φ`
`mrefine`	Refine, in the stateful proof mode	`refine` for a `Std.Do` goal
`mrename_i`	Rename inaccessible names, in the stateful proof mode	`rename_i` for a `Std.Do` goal
`mreplace`	Replace, in the stateful proof mode	`replace` for a `Std.Do` goal
`mrevert`	Revert, in the stateful proof mode	`revert` for a `Std.Do` goal
`mright`	Right, in the stateful proof mode	`right` for a `Std.Do` goal
`mspec`	Specification	Applies a Hoare-triple specification to the program in the goal
`mspecialize`	Specialize, in the stateful proof mode	`specialize` for a `Std.Do` goal
`mspecialize_pure`	Specialize, from the pure context	`mspecialize` with a hypothesis from the pure context
`mstart`	Start the proof mode	Enters the stateful proof mode of `Std.Do`
`mstop`	Stop the proof mode	Leaves the stateful proof mode, without simplifying

Tactic	Name means	What it does
`mvcgen`	Verification condition generator, for monadic programs	Reduces a Hoare triple about a `do` program to verification conditions; deprecated in favour of `vcgen` (Chapter 22)
`mvcgen?`	The same, and suggest invariants	Expands to `mvcgen invariants?`, which suggests loop invariants

N

Tactic	Name means	What it does
`native_decide`	Decide, by native code	`decide` by compiling and running the check, which adds the compiler to what is trusted (Chapter 22)
`next`	Next	`next => tac` works on the next goal, and must close it
`nlinarith` ★	Nonlinear arithmetic	`linarith` after adding products of the hypotheses; give it squares as hints
`nofun`	No function cases	Closes a goal whose premise has no possible case, such as `False → p`
`nomatch`	No match	`nomatch h` closes the goal when `h` has no possible case
`noncomm_ring`	Noncommutative ring	Proves equations in rings that need not be commutative
`nontriviality`	Nontriviality	Adds the hypothesis `Nontrivial α`, when the goal is easy otherwise
`norm_cast` ★	Normalize the casts	Moves casts outward and removes them, stating the goal in the smallest type
`norm_num` ★	Normalize numerals	Proves and simplifies statements about numbers: arithmetic, `<`, `∣`, primality
`norm_num1`	`norm_num`, basic version	`norm_num` without the simp set
`nth_grewrite`	Generalized rewrite, n-th occurrence	`grewrite` at the chosen occurrences only

Tactic	Name means	What it does
`nth_grw`	Generalized rewrite, n-th occurrence	`grw` at the chosen occurrences only
`nth_rewrite`	Rewrite the n-th occurrence	`rewrite` at the chosen occurrences only
`nth_rw` ★	Rewrite the n-th occurrence	`nth_rw 2 [h]` rewrites only the second occurrence

O

Tactic	Name means	What it does
`observe`	Observe	`observe h : p` proves `p` with `exact?` and adds it as `h`
`obtain` ★	Obtain	`obtain ⟨x, hx⟩ := h` takes a hypothesis apart according to a pattern
`omega` ★	The Omega test, an algorithm of W. Pugh (1991)	Proves linear arithmetic over ℕ and ℤ, including `/` and `%` by constants
`on_goal`	On goal n	`on_goal 2 => tac` runs `tac` on the second goal
`open … in`	Open a namespace	Opens a namespace for the tactics that follow `in`
`order`	Order	Proves goals that follow from the hypotheses by the axioms of partial orders, linear orders and lattices

P

Tactic	Name means	What it does
`peel`	Peel off the quantifiers	Removes the matching quantifiers from a hypothesis and from the goal
`pick_goal`	Pick a goal	Moves the n-th goal to the front
`plausible`	Plausible	Tests the goal on random examples, and reports a counterexample if it finds one
`pnat_to_nat`	ℕ+ to ℕ	Moves the positive natural numbers in the context to ℕ

Tactic	Name means	What it does
`polynomial`	Polynomial	Proves equations between polynomials by expanding both sides
`polyrith`	Polynomial arithmetic	No longer supported: it relied on an external service that has closed
`positivity` ★	Positivity	Proves `0 ≤ e`, `0 < e` or `e ≠ 0` from the structure of `e`
`pull`	Pull	`pull c` moves a constant `c` outward, the reverse of `push`
`pure_coherence`	Pure coherence	Proves equations between structural isomorphisms of a monoidal category
`push` ★	Push	`push Not` moves negations inward, so that `¬∀` becomes `∃¬`; `push c` does the same for another constant `c`
`push_cast` ★	Push the casts	Moves casts inward, toward the variables: `↑(a + b)` becomes `↑a + ↑b`
`push_neg`	Push the negations	Deprecated: use `push Not`

Q

Tactic	Name means	What it does
`qify`	Make it about ℚ	Moves statements about `ℕ` or `ℤ` to `ℚ`

R

Tactic	Name means	What it does
`rcases` ★	Recursive cases	Takes a hypothesis apart by a nested pattern: `rcases h with ⟨x, hx⟩ | h'`
`rcongr`	Repeated congruence	Applies `congr` and `ext` repeatedly
`recover`	Recover	Restores goals that a faulty tactic dropped; for debugging tactics
`reduce`	Reduce	Computes the goal completely, unfolding every definition

Tactic	Name means	What it does
`reduce_mod_char`	Reduce modulo the characteristic	Reduces the numerals of a ring of characteristic p, such as `ZMod p`, modulo p
`refine` ★	Refine	Like `exact`, but the holes `?_` of the term become new goals
`refine'`	Refine, older style	`refine` in which the holes `_` also become goals
`refold_let`	Refold a let	Replaces the value of a local definition by its name
`rel`	Relation	`rel [h₁, h₂]` proves `f a ≤ f b` from exactly the hypotheses given, as `gcongr` does
`rename`	Rename	`rename t => h` names the latest hypothesis whose type matches `t`
`rename'`	Rename, by name	`rename' h => h'` renames the hypothesis `h`
`rename_bvar`	Rename a bound variable	Renames a variable bound by `∀`, `∃` or `fun`
`rename_i`	Rename the inaccessible names	Names the most recent hypotheses shown as `h✝`
`repeat`	Repeat	Runs a tactic again and again until it fails
`repeat'`	Repeat, on all goals	Runs a tactic repeatedly on every goal it produces
`repeat1`	Repeat, at least once	Like `repeat`, but fails if the first run fails
`repeat1'`	Repeat on all goals, at least once	Like `repeat'`, but fails if the first run fails
`replace`	Replace	Like `have`, but removes the old hypothesis of the same name
`revert`	Revert	Moves hypotheses back into the goal, the reverse of `intro`
`rewrite`	Rewrite	Replaces the left side of an equation by its right side, without trying `rfl` afterwards

Tactic	Name means	What it does
`rewrite!`	Rewrite, inserting casts	`rewrite` that inserts casts where types depend on the term being rewritten
`rfl` ★	Reflexivity	Proves `a = a`, `a ≤ a` and the like, when the two sides agree by definition
`rfl'`	Reflexivity, unfolding everything	`rfl` that also unfolds theorems and definitions by well-founded recursion
`rify`	Make it about ℝ	Moves statements about ℕ, ℤ, ℚ or `ℝ≥0` to ℝ
`right` ★	Right	Proves `p ∨ q` by proving `q`; in general, uses the second of two constructors
`ring` ★	Commutative ring	Proves equations that hold in every commutative ring, such as `(a + b) ^ 2 = a ^ 2 + 2 * a * b + b ^ 2`
`ring1`	Ring, basic version	`ring` without the fallback to `ring_nf`: fails if it cannot close the goal
`ring_nf`	Ring normal form	Puts the ring expressions of the goal in a normal form, without closing it
`rintro` ★	Recursive introduction	`intro` with patterns: `rintro x ⟨y, hy⟩`
`rotate_left`	Rotate left	Moves the first goal to the end of the list
`rotate_right`	Rotate right	Moves the last goal to the front
`rsuffices`	Suffices, with a pattern	`suffices` whose new hypothesis is taken apart by an `obtain` pattern
`run_tac`	Run a tactic program	Runs a program in the `TacticM` monad (Chapter 23)
`rw` ★	Rewrite	`rw [h]` replaces the left side of `h` by its right side, then tries `rfl`
`rw!`	Rewrite, inserting casts	`rewrite!`, then tidies up the casts it inserted
`rw?` ★	Rewrite, with a search	Lists the lemmas that can rewrite the goal, with the result of each

Tactic	Name means	What it does
`rw??`	Rewrite, interactive search	Suggests rewrites of a subterm that you shift-click in the goal
`rw_mod_cast`	Rewrite, modulo casts	`rw` that normalizes the casts at each step
`rwa`	Rewrite, then assumption	`rw`, then closes the goal with a hypothesis

S

Tactic	Name means	What it does
`says`	Says	`simp? says simp only [h]` keeps both the search and the explicit call it printed
`set` ★	Set	`set x := e with hx` names an expression and replaces it everywhere by `x`
`set_option … in`	Set an option	Sets an option for the tactics that follow `in`
`setm`	Set, by matching	Names the parts of the goal that match the holes `?a` of a pattern
`show` ★	Show	States the goal explicitly; with several goals, selects the one that matches
`show_term`	Show the term	Runs a tactic and prints the proof term it built
`simp` ★	Simplify	Rewrites repeatedly with the `@[simp]` lemmas and the lemmas given
`simp!`	Simplify, unfolding more	`simp` with automatic unfolding of definitions
`simp?` ★	Simplify, and show the lemmas	Prints the equivalent `simp only […]`

Tactic	Name means	What it does
`simp?!`	`simp!`, and show the lemmas	Prints the equivalent `simp only […]`
`simp_all`	Simplify all	Simplifies every hypothesis and the goal, each with the others
`simp_all!`	Simplify all, unfolding more	`simp_all` with automatic unfolding of definitions
`simp_all?`	Simplify all, and show the lemmas	Prints the equivalent `simp_all only […]`
`simp_all?!`	`simp_all!`, and show the lemmas	Prints the equivalent `simp_all only […]`
`simp_all_arith`	Simplify all, with arithmetic	Deprecated: write `simp_all +arith +decide`
`simp_all_arith!`	The same, unfolding more	Deprecated: write `simp_all! +arith +decide`
`simp_arith`	Simplify, with arithmetic	Deprecated: write `simp +arith +decide`
`simp_arith!`	The same, unfolding more	Deprecated: write `simp! +arith +decide`
`simp_intro`	Simplify and introduce	Introduces hypotheses and simplifies with them as it goes
`simp_rw`	Simp-rewrite	Rewrites with each lemma in turn, also under binders such as `∀` and `∑`
`simpa` ★	Simp, then close with an assumption	Simplifies the goal and a hypothesis, then closes one with the other: `simpa using h`

Tactic	Name means	What it does
`simpa!`	`simpa`, unfolding more	`simpa` with automatic unfolding of definitions
`simpa?`	`simpa`, and show the lemmas	Prints the equivalent `simpa only […]`
`simpa?!`	`simpa!`, and show the lemmas	Prints the equivalent `simpa only […]`
`skip`	Skip	Does nothing
`sleep`	Sleep	Waits a given number of milliseconds; used in tests
`slice_lhs`	Slice of the left-hand side	Works on a part of a composition `f ≫ g ≫ h` on the left side
`slice_rhs`	Slice of the right-hand side	Works on a part of a composition on the right side
`solve`	Solve	Like `first`, but accepts only a tactic that closes the goal
`solve_by_elim`	Solve by elimination	Closes the goal by applying hypotheses and the lemmas given, with backtracking
`sorry` ★	Sorry: an apology for a missing proof	Closes any goal without a proof, with a warning
`specialize` ★	Specialize	`specialize h a` replaces `h : ∀ x, P x` by `h : P a`
`specialize_all`	Specialize all	Runs `specialize h x` on every hypothesis `h` where it succeeds
`split`	Split	Splits a goal containing an `if` or a `match` into one goal per branch
`split_ands`	Split the conjunctions	Splits `p ∧ q ∧ …` in every goal

Tactic	Name means	What it does
`split_ifs` ★	Split the if-then-else expressions	Splits into one goal per branch of each `if` in the goal or a hypothesis
`stop`	Stop	Ignores the rest of the proof, as if it were `sorry`
`subsingleton`	Subsingleton	Proves `x = y` in a type with at most one element
`subst`	Substitute	With `h : x = e` for a variable `x`, replaces `x` by `e` everywhere
`subst_eqs`	Substitute the equations	Applies `subst` with the equations of the context, repeatedly
`subst_vars`	Substitute the variables	Applies `subst` to every hypothesis `x = e` or `e = x`
`substs`	Substitute, several	Deprecated: `subst` accepts several hypotheses
`success_if_fail_with_msg`	Success if failure with a message	Succeeds only if the tactic fails with the given message; used in tests
`suffices` ★	It suffices	`suffices h : p by …` reduces the goal to proving `p`
`suggestions`	Suggestions	`#suggestions` prints theorems of the library that may help with the goal
`swap`	Swap	Exchanges the first two goals
`swap_var`	Swap the variables	Exchanges the names of two variables everywhere
`sym`	Symbolic simulation	An interactive mode, built on `grind`, for step-by-step symbolic reasoning
`symm` ★	Symmetry	Turns a goal `a = b` into `b = a`, and the same for any symmetric relation

T

Tactic	Name means	What it does
`tauto` ★	Tautology	Proves tautologies of classical propositional logic, using the hypotheses
`tauto_set`	Tautology about sets	Proves inclusions and equalities between sets built with `∪`, `∩`, `\` and `ᶜ`
`tfae_finish`	TFAE, "the following are equivalent", finish	Closes `TFAE [P₁, P₂, …]` once enough implications are known
`tfae_have`	TFAE, have	`tfae_have 1 → 2 := …` adds one implication between the statements of a `TFAE` goal
`trace`	Trace	Prints a message in the Infoview
`trace_state`	Trace the state	Prints the current goals in the Infoview
`trans`	Transitivity	`trans b` splits `a ≤ c` into `a ≤ b` and `b ≤ c`, for any transitive relation
`transitivity`	Transitivity	A synonym of `trans`
`triv`	Trivial	Deprecated variant of `trivial`
`trivial`	Trivial	Tries `rfl`, `assumption`, `decide` and other simple tactics
`try`	Try	Runs a tactic, and succeeds even if it fails
`try?`	Try, and suggest	Tries many tactics and prints those that close the goal
`try_this`	Try this	Runs a tactic and prints it as a `Try this:` suggestion

U

Tactic	Name means	What it does
`unfold` ★	Unfold	Replaces a defined name by its definition
`unfold_projs`	Unfold the projections	Unfolds the projections of class instances in the goal
`unhygienic`	Without hygiene	Runs tactics so that the names they create can be referred to afterwards

Tactic	Name means	What it does
`unreachable!`	Unreachable	Raises an error if it is ever run; marks code that should never execute
`use` ★	Use this witness	`use a` proves `∃ x, P x` with the witness `a`, leaving `P a` after trying easy closing tactics

V

Tactic	Name means	What it does
`vcgen`	Verification condition generator	The successor of `mvcgen`, still experimental in Lean 4.35.0

W

Tactic	Name means	What it does
`whnf`	Weak head normal form	Computes the goal just until its outermost form appears
`with_implicit`	With implicit transparency	Runs tactics that unfold only reducible and implicit-reducible definitions
`with_panel_widgets`	With panel widgets	Shows the given widgets in the Infoview for the tactics inside
`with_reducible`	With reducible transparency	Runs tactics that unfold only definitions marked `@[reducible]`
`with_reducible_and_instances`	With reducible and instance transparency	Runs tactics that unfold only reducible definitions and instances
`with_unfolding_all`	With full transparency	Runs tactics that may unfold every definition
`with_unfolding_none`	With no transparency	Runs tactics that unfold no definition at all
`wlog`	Without loss of generality	`wlog h : a ≤ b` assumes `h`, and asks you to reduce the other case to this one

Z

Tactic	Name means	What it does
`zify`	Make it about ℤ	Moves statements about ℕ to ℤ, where subtraction behaves as expected

Internal tactics

Lean and Mathlib also define the following 60 tactics, which other tactics call or which serve as default arguments. You do not type them, but you may see their names in error messages: `ac_nf0`, `aesop_cat_nonterminal`, `aesop_unfold`, `apply_ext_theorem`, `array_get_dec`, `array_mem_dec`, `as_aux_lemma`, `attempt_all`, `attempt_all_par`, `bddDefault`, `cases_first_enat`, `cat_disch`, `cfc_cont_tac`, `cfc_tac`, `cfc_zero_tac`, `clean_wf`, `clear_`, `clear_aux_decl`, `compareOfLessAndEq_rfl`, `decreasing_trivial_pre_omega`, `decreasing_with`, `deriving_LawfulEq_tactic`, `deriving_LawfulEq_tactic_step`, `deriving_ReflEq_tactic`, `field_simp_discharge`, `first_par`, `forward`, `forward?`, `gcongr_discharger`, `get_elem_tactic_extensible`, `get_elem_tactic_trivial`, `infer_param`, `isBoundedDefault`, `liftable_prefixes`, `mfld_set_tac`, `monoidal_simps`, `mspec_no_bind`, `mspec_no_simp`, `mvcgen_trivial`, `mvcgen_trivial_extensible`, `norm_cast0`, `pnat_positivity`, `pure_coherence_internal`, `refine_lift`, `refine_lift'`, `rfl_cat`, `saturate`, `saturate?`, `simp_wf`, `sizeOf_list_dec`, `sorry_if_sorry`, `symm_saturate`, `toFinite_tac`, `to_encard_tac`, `try_suggestions`, `unit_interval`, `use_discharger`, `use_finite_instance`, `volume_tac` and `whisker_simps`, together with two entries of the parser itself.

Conversion mode, entered with `conv`, has its own small set of commands, such as `lhs`, `rhs`, `arg`, `enter` and `ext`, which select the part of the goal to rewrite. The community's conversion mode guide explains them with examples.