The Skode Skript Tutorial

Release 0.2.0

The Skode developer team

Automations that read like English.

```
when a new lead arrives
  score it with ai
  if score is hot then
    assign to "Maya" and reply on whatsapp
      "Thanks! We'll call you in 10 minutes"
```

Skode · developer.skode.ai
Help & support: skode.ai/support

CONTENTS

CONTENTS 2
About This Book 4
Whetting Your Appetite 5
Using the Skript Interpreter 6
- Installing it 6
- Your first run 6
- The commands 7
- The interpreter and its environment 7

An Informal Introduction to Skode Skript 8
- Script mode 8
- Strings 8
- Numbers 9
- First steps towards programming 9

Events: when Something Happens 11
- The built-in events 11
- Custom events 11
- Several rules in one file 12
- Shaping the test data 12
- A note on it 12

Variables 13
- One namespace, on purpose 13
- Growing and shrinking 13
- The unset-name rule 14
- Two names with reserved meanings 14

Data Types 15
- Numbers 15
- Strings 15
- Booleans 16
- What Skript deliberately does not have 16

Lists 17
- Doing something with each item 17
- Growing a list 17
- What lists are for — and not for 18

More Control Flow Tools 19
- The shape of a decision 19
- The complete condition catalogue 19
- Chaining steps with and 20
- Stopping early 20

Loops 21
- repeat N times 21
- for each, revisited 21
- while — and the thousand-pass promise 22
- Choosing the right loop 22

Functions 24
- Where definitions live, and when 24
- Shared state, not parameters 25
- Two guardrails 25
- The naming craft 25

Intermezzo: Skript Style 26
- Layout 26
- Naming 26
- Shape 26
- Restraint 26

Talking to the Business: Built-in Actions 27
- reply — answer the person in front of you 27
- send — message anyone, on any channel 27

assign, add tag — route and label 27
create task — leave work for a human 28
wait — patience as a step 28
log and say — notes to the trace 28
The whole toolbox, one table 29
Scoring with AI 30
The simulator's stand-in 30
Pinning the score 30
Guardrails, as always 31
Score as a number 31
Errors and How to Read Them 32
Sentences the language does not know 32
Structural mistakes 32
Value mistakes 33
check versus run 33
Testing Your Skripts 34
The three dials 34
Branch coverage, by hand and by eye 34
Machine-readable runs 34
What the simulator guarantees — and what it cannot 35
The Marketplace 36
Installing a package 36
Publishing your own 36
A worked example 36
From Simulation to Production 38
Deploying 38
What stays the same 38
What becomes real 38
The sandbox promises 38
A deployment ritual that works 38
What Now? 40
Appendix A: Language Reference 41
The grammar 41
Events 41
Actions 41
Conditions 42
Data types 42
Limits and guarantees 42
The CLI 42
Appendix B: Cookbook 43
1 · The classic: score-and-route triage 43
2 · Form follow-up with a task trail 43
3 · A drip sequence 43
4 · Invoice receipts, with a VIP touch 44
5 · Message keyword router 44
6 · Deal-won celebration and handover 45
7 · Multi-channel escalation sweep 45
8 · The self-auditing rule 46
9 · Budget-band pricing assistant 46
10 · Weekly-source hygiene check 47
11 · The polite decliner 47
Appendix C: Glossary 49
Appendix D: Skript for Python and JavaScript Programmers 50

About This Book

This is the official tutorial for **Skode Skript**, release 0.2.0 — the English-like automation language of the Skode platform. It is written in the tradition of the great language tutorials: informal, complete, and built on examples you can run.

Every example is real. This book is compiled by a build system that extracts each program from the manuscript and executes it against the actual Skript interpreter; the output printed beneath an example is the output it produced during that build, byte for byte (with two cosmetic courtesies: run timings are normalised, and a few symbols the print fonts lack are substituted). Error examples work the same way — each was genuinely rejected by the interpreter, and the message shown is its real reply. If an example had failed, this PDF would not have compiled. There are 73 verified executions behind these pages.

Whom it is for. Readers with no programming background can start at Chapter 1 and read straight through; the language was designed for you, and the book assumes nothing. Experienced programmers may prefer to skim Chapters 3–10 and lean on Appendix D, the Python/JavaScript translation table — the interesting parts for you are what the language *refuses* to have, and why.

How to read it. With a terminal open. Installation is one command (Chapter 2), each example is a file away from running, and the interpreter's traces are half the teaching. The book is short because the language is small: Chapters 1–12 are the whole language, 13–16 are craft (errors, testing, packaging, production), and the appendices are the reference you will come back to.

Conventions. Programs appear in dark boxes; their real output in light boxes beneath. `Inline code` marks anything you would type. Command lines beginning with `$` are typed at a shell. The names Maya, Sam and Ada stand for your teammates and customers, and bear no relation to persons living, dead, or fictional — except Ada, obviously.

The language, the CLI, this book's build system, and the book itself live at **developer.skode.ai/skript**. Corrections and questions are welcome on the community hub; the examples being executable means your bug report is reproducible by definition.

Now, to the appetite.

Whetting Your Appetite

If you run a business on Skode — or build for people who do — you have almost certainly wished a computer would just *do the obvious thing*. A new lead arrives; somebody should look at it, decide whether it is promising, hand it to the right person, and send a friendly reply before the lead loses interest. None of that is hard. It is just relentless, and it happens fifty times a day.

You could wire it together with a general-purpose programming language. Python, JavaScript and their friends are wonderful tools, but they ask a lot in exchange: you must learn what a callback is, how to authenticate an HTTP request, why your string had the wrong kind of quotes in it, and what a stack trace is trying to tell you at eleven o'clock at night. For a five-line business rule, that is a heavy toll.

Skode Skript is a different bargain. It is a small, real programming language — with variables, data types, loops, conditionals and functions — whose grammar is deliberately shaped like the English sentence you would have said out loud anyway. The program below is the complete, runnable answer to the situation described in the first paragraph:

```
when a new lead arrives
  score it with ai
  if score is hot then
    assign to "Maya" and reply on whatsapp
      "Thanks! We'll call you in 10 minutes"
```

Read it once and you already know what it does. That is the entire design philosophy of the language**if you can say it, you can probably write it** — and, just as important, anyone on your team can read it back six months later without archaeology.

Do not let the friendliness fool you into thinking Skript is a toy. Over the course of this tutorial you will use named variables and arithmetic, four kinds of data (numbers, strings, booleans and lists), three kinds of loop, user-defined functions that call each other, and an error system that answers back in plain English. Skript is also *sandboxed*: it runs inside guardrails — capped loops, capped recursion, no raw network access — so a typo in an automation can never take down anything else in your Skode workspace.

The language comes with a small command-line tool, `skript`, which runs your programs locally against simulated business events. You can write and test an automation on your laptop in any editor you like, watch exactly what it would do, and only then deploy the very same file to Skode, where the events, the AI scoring, and the WhatsApp messages become real.

This tutorial introduces the language informally, the way you would show it to a colleague: by typing things in and looking at what happens. Every example in this book was executed against the real interpreter while the book was being built, and the output you see printed is the output it genuinely produced. Nothing is mocked.

You do not need any prior programming experience. If you have some, you will move faster, and you may enjoy noticing which conventional ceremonies Skript simply refuses to have.

In the next chapter we install the interpreter and learn to drive it. After that, the language itself: values and variables, control flow, functions, and the built-in vocabulary of business actions. The later chapters cover testing, error messages, the package marketplace, and what changes when your skript leaves the simulator and goes to production. Two appendices — a formal language reference and a cookbook of complete programs — close the book.

It is a short language. It will not take long to learn all of it.

Using the Skript Interpreter

Installing it

The interpreter ships as a small command-line tool named `skript`. It needs Node.js 18 or newer, and nothing else. The quickest installation is one line:

```
curl -fsSL https://developer.skode.ai/skript/install.sh | bash
```

That installs the CLI and — if you use VS Code, Cursor, Windsurf or VSCodium — drops in syntax highlighting and snippets for `.skript` files. If you prefer plain npm:

```
npm install -g https://developer.skode.ai/skript/skode-skript-latest.tgz
```

Either way you end up with two commands on your PATH, `skript` and `skode`, which are the same program under two names. The second exists so that marketplace instructions such as `skode add zoho-crm-connector` read naturally; use whichever you like.

Verify the installation:

```
$ skript version
0.2.0
```

Your first run

The interpreter has a scaffolding command that writes a starter program for you:

```
$ skript new
✔ created lead-autopilot.skript
  next: skript run lead-autopilot.skript
```

Open the file in any editor. Any editor really is fine — a skript is an ordinary UTF-8 text file — though editors in the VS Code family will highlight it after `skript install-editor`. Now run it:

```
$ skript run lead-autopilot.skript
```

Two things about what happens next. First, no lead actually arrived: the interpreter *simulated* one, a demo lead named Ada Lovelace with a phone number, an email address and a budget. Second, everything your program did in response was printed as a **trace** — one line per step, with an arrow showing each step's result. The trace is the interpreter's answer to the question you will ask most often while learning: *what exactly did my program just do?*

You will see hundreds of these traces in this book. They look like this (this one is real, like every output shown in this tutorial):

```
when a new lead arrives
  score it with ai
  if score is hot then
    assign to "Maya" and reply on whatsapp
      "Thanks! We'll call you in 10 minutes"
```

```
▶ example.skript — Skode Skript · local simulation

  ● event: lead.created              → name: Ada Lovelace · phone: +1 555 0100 · email: ada@example.com
  ⚡ score it with ai                 → score 91 · HOT
  ✓ if score is hot                  → yes
  → assign to "Maya"                 → lead assigned
  ✉ reply on whatsapp                → "Thanks! We'll call you in 10 minutes"

  ✔ done in 2.1ms — 1 message · assigned to Maya
```

The `●` line describes the simulated event. Each following line is one step: the AI scored the lead, the condition was checked (`✓` means it passed), the lead was assigned, the reply was sent. The fina`✔` line is a summary. Nothing

here left your machine — messages in the simulator are printed, not sent.

The commands

The CLI has five commands. This is all of them:

Command	What it does
`skript run file.skript`	run the program against a simulated event
`skript check file.skript`	parse only; report errors with line numbers; exit 1 on failure
`skript new [name]`	scaffold a starter program
`skript add <name>`	install a package from the Skode marketplace (Chapter 15)
`skript install-editor`	add highlighting + snippets to VS Code-family editors

There is also a lazy path: `skript lead-autopilot.skript` with no subcommand simply runs the file.

`skript run` accepts a few options you will use constantly while testing:

Option	Meaning
`--score 91`	pin the AI score instead of simulating it — great for forcing a branch
`--data '{"name":"Zed"}'`	override fields of the simulated event
`--event lead.created`	when a file has several rules, run just the matching one
`--json`	machine-readable output instead of the human trace

The interpreter and its environment

A skript file is plain text, so everything you already know about files applies. A few conventions worth adopting from day one:

- Name files after what they automate: `lead-autopilot.skript`, `invoice-receipts.skript`.
- Use the `.skript` extension — the editor tooling and the marketplace both key off it.
- Keep one automation concern per file. Files are cheap; tangled rules are not.

Indentation matters in Skript (it is how the language expresses structure, as the next chapter shows), and the interpreter treats a tab as two spaces. Mixing tabs and spaces will work, but your eyes deserve better: configure your editor to indent `.skript` files with two spaces.

One last environmental note. The `skript` CLI is a *local simulator*. It parses with exactly the same grammar and executes with exactly the same semantics as Skode's production runtime, but events are simulated, `wait` does not really pause, and messages are printed instead of delivered. Chapter 16 describes precisely what changes in production. Everything in between — the language itself — is identical in both worlds, which is the point: test on your laptop, deploy the same file.

An Informal Introduction to Skode Skript

In the examples that follow, the program appears in a dark box and the interpreter's trace appears in a light box beneath it. To follow along, put each program in a file and run `skript run file.skript`. Lines beginning with `#` are comments — the interpreter ignores them, humans should not:

```
# this is a comment
log "this is not"   # this part is a comment too
```

Script mode

The natural shape of a Skript program is a *rule*: "**when** something happens, do these steps." But while you are learning — or debugging — you can skip the rule and just write steps. A file of bare steps runs immediately, top to bottom, in what the trace calls *script mode*. It is the closest thing Skript has to a scratchpad, and we will use it heavily in the next few chapters.

```
log "Hello, Skode!"
```

```
▶ example.skript — Skode Skript · local simulation

  ● event: script                    → name: Demo
  • log                              → Hello, Skode!

  ✔ done in 2.1ms
```

The interpreter still shows an event line — script mode gives you a minimal stand-in event so the rest of the language behaves normally — followed by our one step. `log` writes a note into the trace. It is the `print` of Skode Skript: it does nothing to the outside world, but it is how a program shows you what it is thinking, and it is the tool this tutorial uses to make everything else visible.

Strings

A piece of text in quotes is a **string**. Double quotes and single quotes both work; use whichever avoids the quote you need inside:

```
log "A double-quoted string"
log 'and a single-quoted one'
log "escape a quote like \" this"
```

```
▶ example.skript — Skode Skript · local simulation

  ● event: script                    → name: Demo
  • log                              → A double-quoted string
  → assign to "and a single-quoted one" → event assigned
  • log                              → escape a quote like " this

  ✔ done in 2.1ms — assigned to and a single-quoted one
```

Strings are fully Unicode — accents, currency symbols and emoji are all just characters. (The printed edition of this book substitutes a few symbols that its fonts lack, but your terminal will show them faithfully.)

The most useful thing about Skript strings is **interpolation**: inside any string, a field name in curly braces is replaced by that field's value at the moment the string is used:

```
set city to "Kochi"
log "Weather report for {city}"
```

```
▶ example.skript — Skode Skript · local simulation

  ● event: script                    → name: Demo
  ✎ set city                         → = "Kochi"
  • log                              → Weather report for Kochi

  ✔ done in 2.1ms
```

Interpolation reaches anything the program can see — fields of the incoming event, and variables you have created. If the name inside the braces is not set to anything, Skript leaves the braces in place rather than guessing, which makes typos easy to spot in a trace.

Numbers

Skript has ordinary numbers — whole and decimal — and ordinary arithmetic with `+`, `-`, *and* `/`. *Multiplication and division bind tighter than addition and subtraction, exactly as in school. One habit to learn immediately:* ***put spaces around operators****.* `budget 2` is multiplication; `budget*2` is just a strange name.

```
set subtotal to 40 + 2
set total to subtotal + 10 * 3
set half to total / 2
log "subtotal {subtotal}, total {total}, half {half}"
```

```
▶ example.skript — Skode Skript · local simulation

  ● event: script                    → name: Demo
  ✎ set subtotal                     → = 42
  ✎ set total                        → = 72
  ✎ set half                         → = 36
  • log                              → subtotal 42, total 72, half 36

  ✔ done in 2.1ms
```

Dividing by zero is an error, and Skript tells you so in words rather than in symbols. Arithmetic on something that is not a number — say, a name — is also an error; Chapter 13 shows what those look like.

`+` pulls double duty: between two numbers it adds; if either side is a string, it joins:

```
set name to "Ada"
set greeting to "Hello, " + name + "!"
log "{greeting}"
```

```
▶ example.skript — Skode Skript · local simulation

  ● event: script                    → name: Demo
  ✎ set name                         → = "Ada"
  ✎ set greeting                     → = "Hello, Ada!"
  • log                              → Hello, Ada!

  ✔ done in 2.1ms
```

First steps towards programming

Of course, we can use Skript for more than adding numbers to greetings. Here is the beginning of something real — a little program that works out a discount. It introduces two ideas the next chapters will treat properly: variables that change, and a decision:

```
set price to 900
set discount to 0
if price is above 500 then
  set discount to price / 10
set final to price - discount
log "price {price}, discount {discount}, final {final}"
```

```
▶ example.skript — Skode Skript · local simulation

  ● event: script                    → name: Demo
  ✎ set price                        → = 900
  ✎ set discount                     → = 0
  ✓ if price above 500               → yes
  ✎ set discount                     → = 90
  ✎ set final                        → = 810
  • log                              → price 900, discount 90, final 810

  ✔ done in 2.1ms
```

Notice the shape of the `if`: the condition reads like a sentence (`price is above 500`), the word `then` is allowed but optional, and the step that belongs to the `if` is **indented underneath it**. Indentation is not decoration in Skript — it is how the language knows what belongs to what. There are no braces and no `end` keywords; the structure of the program is its visual structure. The interpreter treats two spaces as one level, and honestly, so should you.

This is everything you need to read the rest of the book: steps run top to bottom, quotes make strings, braces interpolate, spaces-around-operators make arithmetic, and indentation makes structure. Now let us meet the thing that makes Skript a Skode language rather than a calculator: events.

Events: when Something Happens

Every real Skript program is a set of **rules**, and every rule has the same shape: a `when` line naming an event, and an indented body saying what to do about it. This is the fundamental unit of the language — everything else in this book happens inside one.

```
when a new lead arrives
  log "A lead! Their name is {name}."
```

```
▶ example.skript — Skode Skript · local simulation

  ● event: lead.created              → name: Ada Lovelace · phone: +1 555 0100 · email: ada@example.com
  • log                              → A lead! Their name is Ada Lovelace.

  ✔ done in 2.1ms
```

The `when` line is written as a phrase, not a code, and the interpreter is deliberately generous about phrasing `when a new lead arrives`, `when a lead is created`, and `when a lead comes in` all mean the same event. Under the hood each phrase resolves to an event key — this one is `lead.created` — which is what the trace shows, so you always know what your phrase actually matched.

The built-in events

Skript knows five families of business events out of the box:

You write (for example)	Event key	The event's subject
`when a new lead arrives`	`lead.created`	a lead
`when a form is submitted`	`form.submitted`	a submission
`when a message arrives`	`message.received`	a message
`when an invoice is paid`	`invoice.paid`	an invoice
`when a deal is won`	`deal.won`	a deal

Each event carries **fields** — the data of the thing that happened — and your program reads them by name, exactly like the variables of the previous chapter. In the simulator, each event comes with realistic demo data so you have something to work with:

```
when an invoice is paid
  log "Invoice {number} for {amount} was paid by {name}"
```

```
▶ example.skript — Skode Skript · local simulation

  ● event: invoice.paid              → number: INV-0461 · amount: 990 · name: Margaret H
  • log                              → Invoice INV-0461 for 990 was paid by Margaret H

  ✔ done in 2.1ms
```

The demo invoice has a `number`, an `amount` and a `name`; the demo lead has `name`, `phone`, `email`, `source` and `budget`; the demo form submission has `form`, `name`, `email` and `message`. You will discover the fields naturally — the event line at the top of every trace lists the first few — and nothing bad happens if you interpolate a field that is not there: the braces stay visible, telling you exactly which name found no value.

Custom events

The five built-ins cover the common ground, but any part of Skode — or any of your own integrations — can emit events with custom names. Quote the name and the rule matches it verbatim:

```
when "billing.overdue" happens
  log "{name} is {days} days overdue"
```

```
▶ example.skript — Skode Skript · local simulation

  ● event: billing.overdue            → name: Acme · days: 12
  • log                               → Acme is 12 days overdue

  ✔ done in 2.1ms
```

Several rules in one file

A file may hold as many rules as you like, and this is where the `--event` option earns its keep. When the interpreter runs a file it fires every rule (each against its own simulated event), unless you narrow it:

```
when a new lead arrives
  log "lead rule ran"

when a deal is won
  log "deal rule ran — {name} closed for {amount}"
```

```
▶ example.skript — Skode Skript · local simulation

  ● event: deal.won                   → name: Skylark Corp · amount: 14000 · owner: Maya
  • log                               → deal rule ran — Skylark Corp closed for 14000

  ✔ done in 2.1ms
```

Only the deal rule ran, because we asked for `deal.won`. In production the narrowing is automatic — a real event fires exactly the rules that match it — so `--event` is purely a testing convenience.

Shaping the test data

The demo data is fine for a first run, but real testing means trying *your* data. `--data` takes a snippet of JSON and lays it over the demo event — fields you provide replace the demo's, fields you omit keep their demo values:

```
when a new lead arrives
  log "{name} has a budget of {budget}"
```

```
▶ example.skript — Skode Skript · local simulation

  ● event: lead.created               → name: Meera · phone: +1 555 0100 · email: ada@example.com
  • log                               → Meera has a budget of 150

  ✔ done in 2.1ms
```

Between `--event`, `--data`, and (as you will see in Chapter 12) `--score`, you can steer a skript into any branch you need to look at, deterministically, from the command line. Chapter 14 turns that into a testing habit.

A note on `it`

Inside a rule, the word `it` refers to the event's subject — the lead, the invoice, the message. You have already seen the idiom `score it with ai`; you will meet `assign it to "Maya"` later. `it` keeps sentences flowing where repeating the noun would be stilted. When you interpolate, `{it}` resolves to the subject's name if it has one.

That is the stage set: events arrive, rules wake up, fields carry the facts. The next three chapters are about what your rules can *know* — variables, types, and lists — and then we return to what they can *do*.

Variables

A **variable** is a named container for a value. In Skript you create one, and later change it, with the same word: `set`.

```
set counter to 0
set counter to counter + 1
set counter to counter + 1
log "counter is {counter}"
```

```
▶ example.skript — Skode Skript · local simulation

  ● event: script                    → name: Demo
  ✎ set counter                      → = 0
  ✎ set counter                      → = 1
  ✎ set counter                      → = 2
  • log                              → counter is 2

  ✔ done in 2.1ms
```

The right-hand side of `set … to …` can be any expression: a literal value, arithmetic, string joining, another variable, or a mix. The trace prints every `set` with the value it stored, which makes variables pleasantly un-mysterious: you can watch each one change.

One namespace, on purpose

Here is the design decision that makes Skript variables feel different from other languages: **variables and event fields live in the same namespace.** When a lead event carries `name` and `budget`, those are simply names your program can read — and `set` can create new names right beside them:

```
when a new lead arrives
  set budget_with_tax to budget * 1.18
  log "{name}: {budget} becomes {budget_with_tax}"
```

```
▶ example.skript — Skode Skript · local simulation

  ● event: lead.created              → name: Ada Lovelace · phone: +1 555 0100 · email: ada@example.com
  ✎ set budget_with_tax              → = 1416
  • log                              → Ada Lovelace: 1200 becomes 1416

  ✔ done in 2.1ms
```

There is no ceremony separating "their data" from "your data", because in an automation the distinction rarely earns its keep. A consequence worth knowing: `set budget to 0` inside a rule *overwrites your view* of the event's field for the rest of that run. That is occasionally exactly what you want, and occasionally a bug; the trace makes either case visible.

Little courtesies: `set the counter to 1` and `set its status to "vip"` work too — Skript quietly ignores a leading `the` or `its` in the name, because forcing you to un-say natural English would be against its religion.

Growing and shrinking

You can always write `set n to n + 1`, but arithmetic-as-sentence is nicer, so Skript has three growth idioms:

```
set n to 10
increase n by 5
decrease n by 3
add 8 to n
log "n is {n}"
```

```
▶ example.skript — Skode Skript · local simulation

  ● event: script                    → name: Demo
  ✎ set n                            → = 10
  ✎ increase n                       → = 15
  ✎ decrease n                       → = 12
  ✎ add 8 to n                       → = 20
  • log                              → n is 20

  ✔ done in 2.1ms
```

All three appear in the trace with their result, and `increase` / `decrease` on a name that does not exist yet treats it as 0 — handy for counters you have not bothered to initialise.

`add … to …` is the most interesting of the three: it is *type-smart*. Added to a number it does addition, as above; added to a string it appends; added to a list (next chapter) it pushes a new item on the end. One sentence, three sensible meanings — the value on the receiving end decides.

```
set label to "Lead"
add "-2026" to label
log "{label}"
```

```
▶ example.skript — Skode Skript · local simulation

  ● event: script                    → name: Demo
  ✎ set label                        → = "Lead"
  ✎ add to label                     → = "Lead-2026"
  • log                              → Lead-2026

  ✔ done in 2.1ms
```

The unset-name rule

What happens if you use a name that was never set? Most languages answer with an error, some with a special nothing-value. Skript answers like English: **an unset name is just a word.**

```
set plan to pro
log "the plan is {plan}"
```

```
▶ example.skript — Skode Skript · local simulation

  ● event: script                    → name: Demo
  ✎ set plan                         → = "pro"
  • log                              → the plan is pro

  ✔ done in 2.1ms
```

There is no variable called `pro`, so `pro` means the word "pro" — which is surely what the author of that line intended. The same forgiveness means a typo'd variable name will not crash your program; it will flow through as text. That is a deliberate trade: automations should degrade gently, not fall over. The trace and interpolation's kept-braces behaviour are your typo detectors, and Chapter 14 shows how to test suspicious paths deterministically.

Two names with reserved meanings

Two names are reserved: `score` and `heat`, produced by the AI scoring action you will meet properly in Chapter 12. They are special because *reading* them always consults the interpreter's own scoring result — writing `set score to 90` will not fool a later `score is hot` check (pin the score honestly with the `--score` option instead). Treat them as the interpreter's contribution to the namespace.

Everything else — event fields, your counters, your labels, your lists — is yours, mutable, and visible in every trace line that touches it.

Data Types

Skript has four kinds of value — **numbers**, **strings**, **booleans** and **lists** — and it wears its types lightly: you never declare them, you just write values, and each value knows what it is. This chapter covers the first three; lists get the next chapter to themselves.

Numbers

Whole numbers and decimals are one type, written the obvious way: `42`, `3.5`, `-1`. The four operators `+ - * /` combine them with schoolbook precedence — multiplication and division before addition and subtraction, evaluated left to right among equals:

```
set a to 2 + 3 * 4
set b to 10 - 4 - 3
set c to 7 / 2
log "a={a} b={b} c={c}"
```

```
▶ example.skript — Skode Skript · local simulation

  ● event: script                    → name: Demo
  ✎ set a                            → = 14
  ✎ set b                            → = 3
  ✎ set c                            → = 3.5
  • log                              → a=14 b=3 c=3.5

  ✔ done in 2.1ms
```

Note `a` is 14, not 20 — precedence — and `b` is 3, because subtraction associates left. Division produces decimals when it must; Skript does not truncate behind your back. There is no parenthesis syntax in expressions; when a formula grows past what precedence expresses clearly, split it across two `set` lines with a named intermediate — future readers will thank you anyway.

Remember the golden habit from Chapter 3: **spaces around operators**. Names in Skript may contain dots and apostrophes (`lead's budget` is a valid name), so without spaces the interpreter cannot tell your minus sign from your hyphen.

Strings

Strings hold text of any length in `"double"` or `'single'` quotes, escape an inner quote with a backslash, and interpolate `{names}`:

```
set who to "world"
set msg to "Hello, {who}!"
log "{msg} ({msg} has no idea it was assembled)"
```

```
▶ example.skript — Skode Skript · local simulation

  ● event: script                    → name: Demo
  ✎ set who                          → = "world"
  ✎ set msg                          → = "Hello, world!"
  • log                              → Hello, world! (Hello, world! has no idea it was assembled)

  ✔ done in 2.1ms
```

Interpolation happens when the string is *used*, and `set` uses its right-hand side immediately, so `msg` above was assembled at assignment time. Joining with `+` you have seen; between a string and a number, `+` joins too, turning the number into text:

```
set n to 3
set badge to "attempt #" + n
log "{badge}"
```

```
▶ example.skript — Skode Skript · local simulation

  ● event: script                    → name: Demo
  ✎ set n                            → = 3
  ✎ set badge                        → = "attempt #3"
  • log                              → attempt #3

  ✔ done in 2.1ms
```

That is the whole of Skript's type coercion story in one rule: **`+` with any string joins; everything else arithmetic wants real numbers**, and complains in plain English otherwise (Chapter 13 shows the complaint).

Booleans

`true` and `false` are values like any other — storable, comparable, and usable directly as conditions:

```
set vip to true
if vip then
  log "the red carpet, please"
if vip is true then
  log "same check, spelled out"
if vip is not false then
  log "and its double negative, for completeness"
```

```
▶ example.skript — Skode Skript · local simulation

  ● event: script                    → name: Demo
  ✎ set vip                          → = true
  ✓ if vip                           → yes
  • log                              → the red carpet, please
  ✓ if vip is true                   → yes
  • log                              → same check, spelled out
  ✓ if vip is not false              → yes
  • log                              → and its double negative, for completeness

  ✔ done in 2.1ms
```

A bare name as a condition (`if vip then`) asks "is this truthy?" — `false`, `0`, an empty string, and a missing value all fail; everything else passes. The spelled-out forms `is true` / `is not false` do exact comparison. Both styles are common; use the bare form for flags and the spelled form when the value might legitimately be several things.

What Skript deliberately does not have

It is worth saying out loud what is missing, because each absence is a decision, not an oversight:

- **No null.** A name either has a value or is just a word (the unset-name rule). Automations do not need a third state of nothingness to reason about.
- **No dictionaries or objects.** The event *is* the record; its fields are already spread out as names. When you truly need structured data, that is Skode's job — a skript orchestrates it rather than modelling it.
- **No type declarations, casts, or conversions.** The one coercion rule above covers the honest cases; anything trickier is a sign the logic belongs in a Function or a connector, not a sentence.

A small language stays small by declining things. What remains — four types, one coercion rule, sentence-shaped operators — is genuinely all of Skript's data model, and you now know it in full.

Lists

A **list** is an ordered collection of values, and Skript's syntax for one is the humblest possible: values separated by commas.

```
set channels to "whatsapp", "email", "sms"
log "we have channels"
```

```
▶ example.skript — Skode Skript · local simulation

  ● event: script                    → name: Demo
  ✎ set channels                     → = ["whatsapp", "email", "sms"]
  • log                              → we have channels

  ✔ done in 2.1ms
```

Look at the `set` line in the trace: the stored value is shown in square brackets, `["whatsapp", "email", "sms"]` — that is how traces render lists, though your program never writes brackets itself. A list can hold any mix of values: `set oddities to 1, "two", true` is legal, if rarely wise.

Doing something with each item

The whole point of a list is the `for each` loop, which runs its indented body once per item, making the item available under a name you choose:

```
set channels to "whatsapp", "email", "sms"
for each ch in channels
  log "notifying via {ch}"
```

```
▶ example.skript — Skode Skript · local simulation

  ● event: script                    → name: Demo
  ✎ set channels                     → = ["whatsapp", "email", "sms"]
  ↻ for each ch in ["whatsapp", "email", "sms"] → 3 items
  • log                              → notifying via whatsapp
  • log                              → notifying via email
  • log                              → notifying via sms

  ✔ done in 2.1ms
```

The loop variable (`ch` here) is an ordinary variable — created by the loop, visible in interpolations and conditions, and still holding the final item after the loop ends. The trace announces the loop once, with the list and its size, then shows each iteration's steps in order.

You can also loop over a list literal directly, without naming it first — pleasant for small fixed sets:

```
for each vip in "Meera", "Tomas"
  log "reserved a seat for {vip}"
```

```
▶ example.skript — Skode Skript · local simulation

  ● event: script                    → name: Demo
  ↻ for each vip in ["Meera", "Tomas"] → 2 items
  • log                              → reserved a seat for Meera
  • log                              → reserved a seat for Tomas

  ✔ done in 2.1ms
```

Growing a list

`add … to …` — the type-smart sentence from Chapter 5 — pushes onto a list when its target is one:

```
set queue to "first", "second"
add "third" to queue
for each item in queue
  log "processing {item}"
```

```
▶ example.skript — Skode Skript · local simulation

  ● event: script                      → name: Demo
  ✎ set queue                          → = ["first", "second"]
  ✎ add to queue                       → "third" → now 3 items
  ↻ for each item in ["first", "second", "third"] → 3 items
  • log                                → processing first
  • log                                → processing second
  • log                                → processing third

  ✔ done in 2.1ms
```

The trace line for a list-add reports the new length, so you can watch a queue grow. Note that a list must *start* as a list — `add` to a name that is not one falls back to the scalar behaviours of Chapter 5. The companion idiom for collecting results in a loop is therefore the **summary string**, which grows by appending:

```
set matched to ""
for each candidate in "Ada", "Grace", "Linus"
  if candidate contains "a" then
    add "{candidate} " to matched
log "matched: {matched}"
```

```
▶ example.skript — Skode Skript · local simulation

  ● event: script                      → name: Demo
  ✎ set matched                        → = ""
  ↻ for each candidate in ["Ada", "Grace", "Linus"] → 3 items
  ✓ if candidate contains "a"          → yes
  ✎ add to matched                     → = "Ada "
  ✓ if candidate contains "a"          → yes
  ✎ add to matched                     → = "Ada Grace "
  ✗ if candidate contains "a"          → no
  • log                                → matched: Ada Grace

  ✔ done in 2.1ms
```

(That example sneaks in `contains`, a condition you will meet formally in the next chapter — note it is case-insensitive, which is why both Ada and Grace matched.)

What lists are for — and not for

In an automation language, lists mostly carry *plans*: channels to try, tags to apply, teammates to notify, steps to log. They are short, they are built and consumed inside a single run, and `for each` is almost always the only tool you need.

What Skript lists are **not** is a database. There is no indexing (`the 3rd item of…`), no sorting, no removal, no searching beyond `contains` — and that is intentional. The moment your logic wants those operations, the data belongs in Skode itself (a segment, a view, a module) and your skript should be the thing that *reacts* to it, not the thing that stores it. Keeping lists primitive keeps programs sentence-shaped, and sentence-shaped programs are the ones your whole team can still read in a year.

With values, variables and lists in hand, it is time for the machinery that gives a program a mind of its own: conditions and loops, in full.

More Control Flow Tools

You have been using `if` informally since Chapter 3. This chapter lays out the full decision toolkit: every condition the language understands, the `otherwise` branch, chaining, and the emergency brake.

The shape of a decision

```
if <condition> then
  <steps when it holds>
otherwise
  <steps when it does not>
```

`then` is optional — pure politeness. `otherwise` (or its synonym `else`) is also optional, must sit at the same indentation as its `if`, and owns its own indented steps. Decisions nest freely: an `if` inside an `if` reads exactly as you would hope.

```
when a new lead arrives
  if budget is above 1000 then
    if source is "referral" then
      log "big budget AND a referral"
    otherwise
      log "big budget, cold source"
  otherwise
    log "modest budget"
```

```
▶ example.skript — Skode Skript · local simulation

  ● event: lead.created                → name: Ada Lovelace · phone: +1 555 0100 · email: ada@example.com
  ✓ if budget above 1000               → yes
  ✓ if source is "referral"            → yes
  • log                                → big budget AND a referral

  ✔ done in 2.1ms
```

Every condition the interpreter evaluates appears in the trace with a `✓` or `✗`, which turns debugging a decision tree from guesswork into reading.

The complete condition catalogue

This is every form a condition can take — the whole list, not a sample:

Condition	Holds when
`x is <value>`	`x` equals the value (text comparison for words, exact for numbers/booleans)
`x is not <value>`	the opposite
`x is above N` / `is over N` / `is greater than N` / `is more than N`	`x` is numerically greater
`x is below N` / `is under N` / `is less than N`	`x` is numerically smaller
`x is at least N`	`x` is greater than or equal
`x contains "text"`	the text appears inside `x`, ignoring case
`x` (a bare name)	`x` is truthy — set, non-zero, non-empty, not false
`score is hot` / `warm` / `cold`	the AI heat band matches (Chapter 12; requires scoring first)

Two demonstrations of corners people meet early. `contains` ignores case — deliberately, because human data arrives in every capitalisation:

```
when a form is submitted
  if message contains "pricing" then
    log "route to sales"
```

```
▶ example.skript — Skode Skript · local simulation

  ● event: form.submitted              → form: Contact us · name: Grace Hopper · email: grace@example.com
  ✓ if message contains "pricing"      → yes
  • log                                → route to sales

  ✔ done in 2.1ms
```

And there is no `and` / `or` *inside* a condition. Skript keeps conditions atomic; you express conjunction by nesting (as in the first example) and disjunction by writing two `if`s. This is a real limit accepted on purpose: compound boolean algebra is where sentence-languages stop reading like sentences. When you feel the pinch, reach for a function (Chapter 10) whose *name* states the combined meaning.

Chaining steps with `and`

Within a single line, `and` chains steps that belong together:

```
when a new lead arrives
  set plan to "starter" and add tag "self-serve" and log "{name} filed under {plan}"
```

```
▶ example.skript — Skode Skript · local simulation

  ● event: lead.created                → name: Priya · phone: +1 555 0100 · email: ada@example.com
  ✎ set plan                           → = "starter"
  ⚑ add tag "self-serve"               → tagged
  • log                                → Priya filed under starter

  ✔ done in 2.1ms — tags: self-serve
```

The chain runs left to right and is pure sequencing — the steps do not become conditional on each other. Style guidance: two or three short steps that form one thought, chain; anything longer, one per line. And a reassurance about quotes: `and` inside a string is text, never a chain — `add tag "food and drink"` is one tag.

Stopping early

`stop` halts the entire run, immediately, from anywhere — the rest of the rule, enclosing loops, everything:

```
when a new lead arrives
  if email is "" then
    log "no email — nothing to do" and stop
  log "this line is never reached"
```

```
▶ example.skript — Skode Skript · local simulation

  ● event: lead.created                → name: Ada Lovelace · phone: +1 555 0100 · email:
  ✓ if email is ""                     → yes
  • log                                → no email — nothing to do
  ■ stop                               → skript halted

  ✔ done in 2.1ms
```

The guard-clause pattern above — check a disqualifier, `stop` early, keep the happy path unindented — is the single best structuring habit for rules that would otherwise nest five levels deep. The trace marks the halt with `■`, and the run ends cleanly; `stop` is a decision, not an error.

Loops — the other half of control flow — are next, and they follow the same philosophy: a small number of forms, each shaped like something you would say.

Loops

Skript has exactly three loops, and they divide the territory cleanly: `repeat` when you know *how many times*, `for each` when you have *a list of things*, and `while` when you know *the condition to keep going*. Each one takes an indented body, announces itself in the trace with `↻`, and can be cut short by `stop`.

repeat N times

The counted loop is for fixed, known repetition:

```
set drumroll to ""
repeat 3 times
  add "ta-" to drumroll
add "daa!" to drumroll
log "{drumroll}"
```

```
▶ example.skript — Skode Skript · local simulation

  ● event: script                    → name: Demo
  ✎ set drumroll                     → = ""
  ↻ repeat 3 times                   →
  ✎ add to drumroll                  → = "ta-"
  ✎ add to drumroll                  → = "ta-ta-"
  ✎ add to drumroll                  → = "ta-ta-ta-"
  ✎ add to drumroll                  → = "ta-ta-ta-daa!"
  • log                              → ta-ta-ta-daa!

  ✔ done in 2.1ms
```

`repeat 1 time` is accepted too — the interpreter does not make you pluralise against your grammar instincts. What the counted loop deliberately lacks is a loop counter of its own; when the body needs to know which pass it is on, keep one honestly:

```
set attempt to 0
repeat 3 times
  increase attempt by 1
  log "attempt number {attempt}"
```

```
▶ example.skript — Skode Skript · local simulation

  ● event: script                    → name: Demo
  ✎ set attempt                      → = 0
  ↻ repeat 3 times                   →
  ✎ increase attempt                 → = 1
  • log                              → attempt number 1
  ✎ increase attempt                 → = 2
  • log                              → attempt number 2
  ✎ increase attempt                 → = 3
  • log                              → attempt number 3

  ✔ done in 2.1ms
```

for each, revisited

You met `for each` in the lists chapter; it is the workhorse of the three. One more pattern worth showing — a loop body that *decides*, combining iteration and branching into the shape most real automations take:

```
when a new lead arrives
  for each threshold in 1000, 500, 100
    if budget is at least threshold then
      log "clears the {threshold} bar"
```

```
▶ example.skript — Skode Skript · local simulation

  ● event: lead.created                → name: Ada Lovelace · phone: +1 555 0100 · email: ada@example.com
  ↻ for each threshold in [1000, 500, 100] → 3 items
  ✗ if budget is "at least threshold"  → no
  ✗ if budget is "at least threshold"  → no
  ✗ if budget is "at least threshold"  → no

  ✔ done in 2.1ms
```

Note that the loop variable `threshold` is doing real work inside the condition — loop variables are full citizens, not just interpolation fodder.

while — and the thousand-pass promise

The condition loop keeps going as long as its condition holds:

```
set countdown to 5
while countdown is above 0
  log "T-minus {countdown}"
  decrease countdown by 1
log "liftoff"
```

```
▶ example.skript — Skode Skript · local simulation

  ● event: script                      → name: Demo
  ✎ set countdown                      → = 5
  ↻ while countdown above 0            →
  • log                                → T-minus 5
  ✎ decrease countdown                 → = 4
  • log                                → T-minus 4
  ✎ decrease countdown                 → = 3
  • log                                → T-minus 3
  ✎ decrease countdown                 → = 2
  • log                                → T-minus 2
  ✎ decrease countdown                 → = 1
  • log                                → T-minus 1
  ✎ decrease countdown                 → = 0
  • log                                → liftoff

  ✔ done in 2.1ms
```

`while` is the one loop that can, in principle, run forever — the body above *must* change `countdown`, or the condition never fails. Every programmer alive has written that bug at least once, so Skript refuses to let it be catastrophic: **a `while` loop is capped at 1,000 passes.** This program forgets to change `c`; the interpreter stops it and asks exactly the question you need:

```
set c to 0
while c is below 5
  log "spinning"
```

```
✗ line 2: the "while c below 5" loop ran 1000 times — it looks infinite (does the loop change what it
checks?)
```

This is the sandbox philosophy in one example: in a general-purpose language an infinite loop hangs the process; in an automation platform it could hang much more than that. In Skript it is a polite, specific error, and everything around it keeps running. (The trace itself has a similar guard — beyond four hundred lines it truncates its *display* while the program continues — so even a heavy legitimate loop cannot bury you in output.)

Choosing the right loop

A rule of thumb that has never failed anyone: say the sentence out loud first.

- "Do this **three times**" → `repeat 3 times`
- "Do this **for each channel**" → `for each ch in channels`

- "Keep going **while there's budget left**" → `while budget is above 0`

If the sentence does not fall naturally into one of those shapes, the odds are excellent that you do not need a loop at all — automations mostly react once, to one event, and the loop you were about to write belongs in the platform (a segment, a broadcast, a scheduled flow) rather than in the skript. The loops are here for when the logic truly is iterative; the language just makes sure that even then, it stays legible and bounded.

Functions

A **function** is a named, reusable block of steps. Skript's syntax for defining one is the same construction English uses — an infinitive:

```
to welcome the lead
  reply on whatsapp "Welcome, {name}!"
  add tag "welcomed"
```

`to`, then a name — which may be several words — then an indented body. And the syntax for *calling* it is even smaller: you write the name as a plain sentence, anywhere a step can go. Here is both halves together, running:

```
when a new lead arrives
  welcome the lead
  log "rule finished"

to welcome the lead
  reply on whatsapp "Welcome, {name}!"
  add tag "welcomed"
```

```
▶ example.skript — Skode Skript · local simulation

  ● event: lead.created                → name: Ada · phone: +1 555 0100 · email: ada@example.com
  ƒ welcome the lead                   → run your function
  ✉ reply on whatsapp                  → "Welcome, Ada!"
  ⚑ add tag "welcomed"                 → tagged
  • log                                → rule finished

  ✔ done in 2.1ms — 1 message · tags: welcomed
```

Trace-reading note: the `ƒ` line marks the call, and the function's own steps follow it. When the body ends, control returns to the caller — `log "rule finished"` ran after the function's two steps, exactly as the order of the trace shows.

Where definitions live, and when

Function definitions sit at the **top level** of the file — beside your `when` rules, never inside them — and their order does not matter. Define after use, use after define; the interpreter reads the whole file before running anything. This is worth a demonstration because it shapes how files get organised — rules first, telling the story; vocabulary after, alphabetically if you like:

```
when a new lead arrives
  file the paperwork

to file the paperwork
  note the arrival and stamp it

to note the arrival
  log "noted: {name}"

to stamp it
  add tag "processed"
```

```
▶ example.skript — Skode Skript · local simulation

  ● event: lead.created                → name: Ada Lovelace · phone: +1 555 0100 · email: ada@example.com
  ƒ file the paperwork                 → run your function
  ƒ note the arrival                   → run your function
  • log                                → noted: Ada Lovelace
  ƒ stamp it                           → run your function
  ⚑ add tag "processed"                → tagged

  ✔ done in 2.1ms — tags: processed
```

That example also shows the second big fact: **functions call functions.** `file the paperwork` is defined entirely in terms of two other functions, and the trace nests all three calls in order. This is how skripts grow past twenty lines without becoming unreadable — the rule stays a five-line summary, and each named step hides one screen of honest detail.

Shared state, not parameters

Skript functions take **no parameters and return no values** — deliberately. They read and write the same namespace as everything else: event fields, your variables, the score. Where another language passes arguments, a skript sets the stage first:

```
set discount to 15
apply the discount

to apply the discount
  set final_price to 200 - 200 * discount / 100
  log "with {discount} percent off: {final_price}"
```

```
▶ example.skript — Skode Skript · local simulation

  ● event: script                    → name: Demo
  ✎ set discount                     → = 15
  ƒ apply the discount               → run your function
  ✎ set final_price                  → = 170
  • log                              → with 15 percent off: 170

  ✔ done in 2.1ms
```

Is that a compromise? Yes — a knowing one. Parameters earn their keep in libraries used by strangers; a skript is a page of logic owned by one team, where "set the variable, then say the sentence" is simpler than a calling convention, and where every function's effect is fully visible in the trace. When you genuinely need parameterised, reusable machinery, that is what marketplace packages and Skode Functions are for — Skript functions are for naming *this file*'s ideas.

Two guardrails

First: since a function may call itself (nothing forbids it), runaway recursion is possible — and, like the infinite `while`, it is caught rather than fatal. Sixteen levels deep, the interpreter calls it what it almost certainly is:

```
when a new lead arrives
  echo

to echo
  echo
```

```
✗ line 5: "echo" calls itself too deeply (16 levels) — probably an accidental loop
```

Second: a definition must not be empty, and defining the same name twice is an error — both caught a `check` time, before anything runs. The error chapter (13) shows the exact messages.

The naming craft

Because calls read as sentences inside your rule, the *name* of a function is user interface. The difference between a good skript and a great one is mostly here. Some conventions that hold up:

- Name the **business meaning**, not the mechanics: `escalate to a human`, not `send msg and tag 2`.
- Keep rule bodies at one altitude: either all raw steps or all function calls. Mixed altitudes read like a recipe that suddenly explains chemistry.
- If you cannot name a block of steps, that is information — the block probably contains two ideas. Split it.

Functions complete the language core: values, variables, lists, decisions, loops, and now named abstraction. Everything after this chapter is vocabulary — the built-in actions that touch the business world — and craft.

Intermezzo: Skript Style

Half the language core remains (the business actions, scoring), but you already know enough to write real programs — which means you know enough to write untidy ones. A short pause, then, for style. Skript has fewer degrees of freedom than most languages, so its style guide fits on two pages; the compound interest it pays does not.

Layout

- **Two spaces per indent level.** The interpreter tolerates more; readers should not have to.
- **One blank line between rules and between function definitions.** None inside them, usually — a rule that needs internal paragraph breaks is a rule that wants functions.
- **Rules first, functions after.** The top of the file should read like a table of contents for what the automation does; the bottom, like a dictionary for how.
- **Comment the contract, not the code.** The steps explain themselves — that is the language's whole trick. What steps cannot say is context: which event feeds this file, which teammate names are placeholders, which command lines were used to test it. Three `#` lines at the top of the file carrying those facts are worth more than a comment per line.

Naming

- **Variables are nouns, lowercase, underscore-separated** when multiword: `followup_delay`, `alerted`. You *can* use spaces and apostrophes in names — the grammar allows `lead's budget` — but reserve that flourish for reading-critical moments; underscores survive editing better.
- **Functions are verb phrases** — they are called as sentences, so name them as things one *does*: `sound the alarm`, `route to sales`, `file the paperwork`. If the name wants to contain "and", it is two functions.
- **Tags are kebab-case labels**: `"follow-up-today"`, `"receipt-sent"`. Tags outlive the skript that set them; name them for the person filtering a list view next quarter.

Shape

- **Guard early, `stop` early.** Disqualify at the top and keep the happy path flat, rather than nesting the whole rule inside an `if`.
- **One altitude per body.** A rule reads either as raw steps or as function calls — mixing the two reads like a recipe that pauses to explain chemistry.
- **Chain with `and` only what forms one thought** — two, maybe three short steps. `set plan to "starter" and add tag "self-serve"` is one thought; five chained actions is a paragraph pretending to be a sentence.
- **Narrate decisions.** A `log` at each major branch ("hot path, assigning to Maya") costs one line and turns every production trace into its own incident report.

Restraint

The strongest style rule in Skript is the one the language enforces socially rather than syntactically: **if the logic stops reading like something you would say, stop writing it in Skript.** Deep arithmetic, elaborate data reshaping, cascades of nested conditions — these are signs the problem has outgrown sentences, and Skode has proper homes for them (Functions, connectors, the SDK). A skript at its best is the *conductor's score*, not the orchestra: short, legible, and unmistakably in charge.

None of this is enforced. All of it is what the marketplace's most-starred packages have in common — which is the only authority a style guide ever really has.

Talking to the Business: Built-in Actions

Everything so far — variables, loops, functions — is the*grammar* of Skript. This chapter is the*vocabulary*: the built-in actions that reach out of the program and into your business. In the simulator they print what they would do; in production they do it.

reply — answer the person in front of you

`reply` messages whoever triggered the event — the lead who arrived, the sender of the message. Its channel defaults to WhatsApp, and the message can sit inline or, when it deserves its own line, indented underneath:

```
when a new lead arrives
  reply on whatsapp "Quick hello, {name}!"
  reply on email
    "A longer note deserves its own line."
    "Each quoted line becomes one line of the message."
```

```
▶ example.skript — Skode Skript · local simulation

  ● event: lead.created                → name: Ada · phone: +1 555 0100 · email: ada@example.com
  ✉ reply on whatsapp                  → "Quick hello, Ada!"
  ✉ reply on email                     → "A longer note deserves its own line.\nEach quoted line becomes one
line of the message."

  ✔ done in 2.1ms — 2 messages
```

The indented-string form is unique to `reply`: write the message as one or more quoted lines beneath it, and they join into a single multi-line message. Notice in the trace where the reply went: the simulator picked the lead's phone for WhatsApp — the *subject's* contact details travel with the event.

send — message anyone, on any channel

Where `reply` answers the event's subject, `send` names its own target:

```
when a new lead arrives
  send email to "maya@skode.ai" "Hot inbound: {name}, budget {budget}"
  send slack "New lead {name} just landed"
```

```
▶ example.skript — Skode Skript · local simulation

  ● event: lead.created                → name: Ada · phone: +1 555 0100 · email: ada@example.com
  ✉ send email to "maya@skode.ai"      → "Hot inbound: Ada, budget 5000"
  ✉ send slack                         → "New lead Ada just landed"

  ✔ done in 2.1ms — 2 messages
```

The channel comes first, an optional `to "address"` next, the message last. Skript knows six channels:**whatsapp, email, sms, instagram, webchat, slack**. Naming any other channel is a parse error that lists the six — no silent guessing about where a message might go.

assign, add tag — route and label

```
when a new lead arrives
  assign to "Maya"
  add tag "inbound" and add tag "priority"
```

```
▶ example.skript — Skode Skript · local simulation

  ● event: lead.created                → name: Ada · phone: +1 555 0100 · email: ada@example.com
  → assign to "Maya"                   → lead assigned
  ⚑ add tag "inbound"                  → tagged
  ⚑ add tag "priority"                 → tagged

  ✔ done in 2.1ms — assigned to Maya · tags: inbound, priority
```

`assign to "Maya"` hands the event's subject to a teammate (`assign it to "Maya"` reads better in some sentences and is identical). `add tag` labels the record; tags accumulate, and the run summary counts them. These two are the bread and butter of routing rules — most production skripts are, at heart, a decision followed by an `assign` and a couple of tags.

create task — leave work for a human

Some follow-ups should not be automated. `create task` puts an item on a teammate's list, with the title interpolated like any string:

```
when a new lead arrives
  create task "Call {name} back today" for "Maya"
  create task "Update the forecast"
```

```
▶ example.skript — Skode Skript · local simulation

  ● event: lead.created                → name: Ada · phone: +1 555 0100 · email: ada@example.com
  ☑ create task "Call {name} back today" → created
  ☑ create task "Update the forecast"  → created

  ✔ done in 2.1ms — 2 tasks
```

The `for "Name"` part is optional — without it the task lands unassigned, for whoever triages the queue.

wait — patience as a step

`wait` pauses the automation: `wait 10 minutes`, `wait 2 hours`, `wait 3 days` (seconds work too). In the simulator it is instant and says so; in production it genuinely schedules the continuation, which is what makes drip sequences possible:

```
when a new lead arrives
  reply on email "Welcome aboard, {name}!"
  wait 2 days
  reply on email "How is it going so far?"
```

```
▶ example.skript — Skode Skript · local simulation

  ● event: lead.created                → name: Ada · phone: +1 555 0100 · email: ada@example.com
  ✉ reply on email                     → "Welcome aboard, Ada!"
  ◌ wait 2 days                        → (simulated — instant here, real delay in Skode)
  ✉ reply on email                     → "How is it going so far?"

  ✔ done in 2.1ms — 2 messages
```

This is the one action whose local and production behaviours meaningfully differ, so the trace flags it loudly. Design consequence worth absorbing: everything after a `wait` runs *later*, against the world as it will be then — keep the steps after a long wait simple and self-contained.

log and say — notes to the trace

You know `log`; `say` is its exact synonym for when a sentence reads better out loud. Neither touches the outside world in any environment — they exist for the humans reading the trace, and seasoned skript-writers sprinkle them at decision points as running commentary. A rule that narrates itself is a rule nobody has to debug.

The whole toolbox, one table

Action	Sentence shape
reply	`reply on <channel> "…"` — or the message indented beneath
send	`send <channel> [to "addr"] "…"`
assign	`assign [it] to "Name"`
tag	`add tag "label"`
task	`create task "title" [for "Name"]`
wait	`wait N seconds/minutes/hours/days`
note	`log "…"` / `say "…"`
score	`score it with ai` — the next chapter
halt	`stop`

Nine actions. With the grammar chapters behind you, you now know — in the strictest sense — the entire language except one action. That action gets its own chapter, because it is the one with a mind of its own.

Scoring with AI

One sentence in Skript carries more machinery than all the others combined:

```
score it with ai
```

In production this hands the event's subject — the lead, the submission, the message — to Skode's AI, which reads everything known about it and produces two values: `score`, a number from 0 to 100, and `heat`, a band derived from it. From that moment both names are available to conditions and interpolation, and `score is hot` becomes the most consequential five letters in your rule.

heat	when
`hot`	score 75 and above
`warm`	45 – 74
`cold`	below 45

The simulator's stand-in

Locally there is no production AI, so the interpreter simulates one — and it does so *honestly* rather than randomly. The simulated scorer reads the same signals a real one would consider — is there a phone number? an email? a meaningful budget? a warm source? intent words like "pricing" in the text? — and builds the score from them, plus a small deterministic wobble. Same input, same score, every time:

```
when a new lead arrives
  score it with ai
  log "scored {score}, {heat}"
```

```
▶ example.skript — Skode Skript · local simulation

  ● event: lead.created                → name: Ada Lovelace · phone: +1 555 0100 · email: ada@example.com
  ⚡ score it with ai                   → score 79 · HOT (simulated ai — signals: phone, email, budget,
source)
  • log                                → scored 79, hot

  ✔ done in 2.1ms
```

The trace's score line lists which signals fired, so a surprising score always explains itself. Strip the demo lead of its signals and the score collapses accordingly:

```
when a new lead arrives
  score it with ai
  log "scored {score}, {heat}"
```

```
▶ example.skript — Skode Skript · local simulation

  ● event: lead.created                → name: Mystery · phone:  · email:
  ⚡ score it with ai                   → score 24 · COLD (simulated ai)
  • log                                → scored 24, cold

  ✔ done in 2.1ms
```

Two runs, two honest scores — a fully-qualified lead runs hot, an anonymous one runs cold, and nothing about it is magic. But remember the caveat that matters: **the simulated score is a development aid.** It approximates the shape of real scoring so your branches exercise realistically; it does not predict what production AI will say about a particular lead.

Pinning the score

Which raises the real question: how do you test the hot branch, the cold branch, and the boundary, deliberately? With `--score`, which overrides the simulation entirely:

```
when a new lead arrives
  score it with ai
  if score is hot then
    log "hot path taken"
  otherwise
    log "cold path taken"
```

```
▶ example.skript — Skode Skript · local simulation

  ● event: lead.created               → name: Ada Lovelace · phone: +1 555 0100 · email: ada@example.com
  ⚡ score it with ai                  → score 75 · HOT
  ✓ if score is hot                   → yes
  • log                               → hot path taken

  ✔ done in 2.1ms
```

75 is exactly the hot boundary — and the trace shows it lands hot, pinning down an off-by-one you would otherwise wonder about. A disciplined habit: any rule that branches on heat gets run at least three times before deploying — `--score 90`, `--score 60`, `--score 10` — and Chapter 14 makes that a one-liner each.

Guardrails, as always

Asking `score is hot` *before* any `score it with ai` has run is not a silent false — it is a specific error, because it is always a genuine mistake:

```
when a new lead arrives
  if score is hot then
    stop
```

```
✗ line 2: "score is hot" needs "score it with ai" first
```

Score as a number

`heat` covers most routing, but `score` is an ordinary number and the comparison conditions all apply — useful when your thresholds differ from the built-in bands:

```
when a new lead arrives
  score it with ai
  if score is at least 80 then
    assign to "Maya" and add tag "priority"
  if score is above 95 then
    create task "White-glove this one" for "Maya"
```

```
▶ example.skript — Skode Skript · local simulation

  ● event: lead.created               → name: Ada Lovelace · phone: +1 555 0100 · email: ada@example.com
  ⚡ score it with ai                  → score 82 · HOT
  ✓ if score at least 80              → yes
  → assign to "Maya"                  → lead assigned
  ⚑ add tag "priority"                → tagged
  ✗ if score above 95                 → no

  ✔ done in 2.1ms — assigned to Maya · tags: priority
```

Both conditions read the same score; one passed, one did not, and the trace shows each verdict separately. This — scoring once, branching several ways — is the backbone of every triage skript you will ever write, and you now have all of its parts.

Errors and How to Read Them

Skript's error messages are written on one principle:**the interpreter answers in the same language you wrote in.** No codes, no stack traces — a line number, a plain-English diagnosis, and wherever possible, the fix. This chapter is a guided tour of the mistakes you will actually make, each one genuinely made and genuinely answered below.

Sentences the language does not know

The flagship error. Write a step Skript has never heard of, and it responds with its entire repertoire:

```
when a new lead arrives
  frobnicate the lead
```

```
✗ /tmp/skbook-w27fse9o/example.skript: line 2: I don't understand "frobnicate the lead".
  Things I know: score it with ai · assign to "Name" · reply on whatsapp "…" · send email "…" · add tag "…"
· set x to value · increase x by 1 · repeat 3 times · for each x in list · while … · to <name> (define a
function) · wait 10 minutes · create task "…" · log "…" · stop
```

Read that reply carefully — it is a complete inventory of the language's verbs, which makes this error message double as documentation. Better still, if your file defines functions, they are appended to the list under "Your functions" — so a *typo in a function call* gets you the menu of your own vocabulary:

```
when a new lead arrives
  welcom the lead

to welcome the lead
  add tag "hi"
```

```
✗ /tmp/skbook-w27fse9o/example.skript: line 2: I don't understand "welcom the lead".
  Things I know: score it with ai · assign to "Name" · reply on whatsapp "…" · send email "…" · add tag "…"
· set x to value · increase x by 1 · repeat 3 times · for each x in list · while … · to <name> (define a
function) · wait 10 minutes · create task "…" · log "…" · stop
  Your functions: welcome the lead
```

The eye falls on `welcom` versus `welcome the lead` immediately. This is the unset-name rule's important exception: unknown *values* flow through as words, but unknown*steps* stop the show — a step that does nothing is never what you meant.

Structural mistakes

Indentation is structure, so structure errors are indentation errors. An empty rule is the classic:

```
when a new lead arrives
```

```
✗ /tmp/skbook-w27fse9o/example.skript: line 1: this "when" has an empty body — indent at least one action
under it
```

The same guard covers empty function bodies and loops (`"repeat 3 times" needs an indented body`). An `otherwise` that has drifted away from its `if` gets caught too:

```
when a new lead arrives
  otherwise
    log "orphaned"
```

```
✗ /tmp/skbook-w27fse9o/example.skript: line 2: "otherwise" needs an "if" right above it
```

And a function defined twice is refused up front, because which body wins should never be a guessing game:

```
to greet
  log "hi"

to greet
  log "hello"
```

```
✗ /tmp/skbook-w27fse9o/example.skript: line 4: "to greet" is defined twice
```

Value mistakes

Arithmetic on a non-number is caught at the moment of the crime, and names the culprit value:

```
set label to "priority"
set nonsense to label * 3
```

```
✗ can't do math with "priority" — it isn't a number
```

Division by zero gets four honest words (`can't divide by zero`), and conditions that need a list (`for each` over a plain value) or need scoring first (`score is hot` before `score it with ai`) each explain their prerequisite — you met both earlier.

check versus run

The errors above split into two families, and the split matters for workflow. *Structural* mistakes — unknown sentences, empty bodies, orphaned `otherwise`, duplicate functions — are caught by `skript check` **without running anything**. *Value* mistakes — bad arithmetic, infinite loops, runaway recursion — can only surface when the program actually runs, since they depend on data.

The workflow follows directly: `check` constantly while writing (or let your editor do it on save — it exits 1 on failure, so it wires into anything), and `run` the branches before you ship. Between the two, every error in this chapter gets met on your laptop, in a terminal, with a line number — rather than in production, silently, at 2 a.m. That is the entire ambition of Skript's error design: **errors should be boring.** You read the sentence, you fix the line, you move on.

Testing Your Skripts

An automation is a program that runs when you are not looking. That makes testing less optional than it is for most code — and Skript's answer is to make the honest test so cheap that skipping it feels silly. Everything in this chapter is a one-liner in a terminal.

The three dials

You have met all three testing options; here they are as a method. Every skript run is fully determined by three things you control from the command line:

- **Which rule fires** — `--event lead.created`
- **What the event carries** — `--data '{"budget":50}'`
- **What the AI says** — `--score 90`

Because the simulator is deterministic (same inputs, same trace, always), a command line*is* a test case. Save the ones that matter next to the skript, in a plain text file or a shell script, and your automation has a regression suite:

```
skript run triage.skript --score 90   # hot path
skript run triage.skript --score 60   # warm path
skript run triage.skript --score 10   # cold path
skript run triage.skript --score 90 --data '{"email":""}'   # hot but unreachable
```

Branch coverage, by hand and by eye

The discipline that pays: **run every branch once before deploying.** Trace `✓`/`✗` marks make coverage visible — if you have never seen a `✓` next to a condition, you have never tested that branch. Here is a rule with an easy-to-miss corner, and the run that catches it:

```
when a new lead arrives
  score it with ai
  if score is hot then
    if email is "" then
      create task "Find contact info for {name}" and stop
    reply on email "Let's talk today, {name}!"
```

```
▶ example.skript — Skode Skript · local simulation

  ● event: lead.created                → name: Ghost · phone: +1 555 0100 · email:
  ⚡ score it with ai                   → score 90 · HOT
  ✓ if score is hot                    → yes
  ✓ if email is ""                     → yes
  ☑ create task "Find contact info for {name}" → created
  ■ stop                               → skript halted

  ✔ done in 2.1ms — 1 task
```

A hot lead with no email address: without that inner guard, the reply would have gone nowhere. The `--data` dial exists precisely to manufacture these awkward customers on demand.

Machine-readable runs

`--json` swaps the human trace for a structured report — the same facts as data:

```
skript run triage.skript --score 90 --json
```

The JSON carries the outcome of the run — score, heat, assignee, tags, every message with its channel and text, every task — which makes assertions trivial in any language or in CI. A typical CI step is exactly this:

```
skript check triage.skript || exit 1
skript run triage.skript --score 90 --json | grep -q '"assigned": "Maya"' || exit 1
```

`check` exits non-zero on any structural error, `run --json` exposes the outcome to your assertion tool of choice, and

the whole thing needs only Node on the runner — no test framework, no fixtures, no mocks.

What the simulator guarantees — and what it cannot

Worth stating crisply, because good testing depends on knowing the boundary.

The simulator guarantees the **language**: parsing, variables, arithmetic, conditions, loops, functions, and the order and content of every action your program takes. If the trace shows `assign to "Maya"` after `✓ if score is hot`, production will do exactly that in exactly that case.

It cannot guarantee the **world**: what production AI will score some particular lead, whether a phone number is a deliverable WhatsApp account, what a teammate does with a task. Those live outside any language. The craft is to keep the boundary clean — make the *logic* exhaustively tested where it is cheap (here), and keep the *world-facing* steps simple enough that no surprise can hide in them.

One more chapter of machinery — sharing skripts through the marketplace — and then we talk about production itself.

The Marketplace

Nobody should write the fifth copy of "send a receipt when an invoice is paid." The Skode developer hub hosts an open-source **marketplace** — connectors, widgets, templates, and ready-made skripts, published by the community — and the CLI you already have is its package manager.

Installing a package

Browse at `developer.skode.ai/marketplace`, or install directly if you know the name:

```
$ skript add invoice-pdf-skript
▼ Invoice → PDF Skript v1.0.0 · Skript · MIT · by Lin
  + invoice-pdf.skript

✔ installed invoice-pdf-skript (1/1 file)
  next: skript run invoice-pdf.skript
```

`skript add` downloads the package's files into your current directory — no hidden directories, no lockfiles, no dependency graph. A skript package *is* its files, and the first thing you should do with one is the same thing you would do with any skript: open it, read it (it is a page of English), and run it in the simulator. The command even suggests the run line when the package contains a `.skript` file.

Because the alias `skode` points at the same CLI, package pages usually print the command as `skode add <name>` — same thing, friendlier sentence.

Every listing on the marketplace is open source with a declared license, shows a repository link, and counts stars and installs so you can gauge what the community actually uses. Installing never talks to your Skode workspace — it copies files to your laptop, nothing more, which is also why it needs no login.

Publishing your own

The reverse direction is one form on the marketplace page: **Publish**. You give it a name, a type (Skript, Connector, Widget, App, or Template), the product it belongs with, a one-line description, a readme, a license, your repository link — and, for a skript package, the file itself. The install command (`skript add your-name`) is derived from the name automatically, and the listing is live the moment you submit. You can edit or unpublish anything you published, from the same page.

What makes a package worth publishing is mostly what makes any skript good, plus generosity at the edges:

- **Make the file self-explanatory.** The reader gets your `.skript` before your readme. Comments at the top stating what event it expects and what it does are worth more than a wiki.
- **Ship it tested.** Include the `--score`/`--data` command lines you verified it with, as comments. The next person runs them first.
- **Name variables for strangers.** `set followup_delay to 2` invites tuning; magic numbers do not.
- **State the seams.** If your skript assumes a tag, a teammate name, or a custom event that the installer must adapt, say so at the top — those are the three things every adopter edits.

A worked example

Here is the shape of a package worth sharing — a complete, self-documenting skript of the kind you find on the marketplace (and yes, it runs; everything in this book runs):

```
# thank-and-receipt.skript — send a thank-you when an invoice is paid.
# Expects: invoice.paid events. Adapt: the teammate name in the task.
# Tested with: skript run thank-and-receipt.skript --event invoice.paid

when an invoice is paid
  reply on email
    "Thank you! Invoice {number} is settled — receipt attached."
  add tag "receipt-sent"
  if amount is above 5000 then
    create task "Send a handwritten note to {name}" for "Maya"
```

```
▶ example.skript — Skode Skript · local simulation

  ● event: invoice.paid                → number: INV-0461 · amount: 990 · name: Margaret H
  ✉ reply on email                     → "Thank you! Invoice INV-0461 is settled — receipt attached."
  ⚑ add tag "receipt-sent"             → tagged
  ✗ if amount above 5000               → no

  ✔ done in 2.1ms — 1 message · tags: receipt-sent
```

Twelve lines, and a stranger could deploy it in the time it took to read them. That is the marketplace's real product: not code, but *confidence* — the sense that adopting someone's automation is as safe as reading a paragraph. Keep your packages that honest and the stars take care of themselves.

From Simulation to Production

Everything in this book so far ran on your laptop. This chapter is about the day the same file runs inside Skode — what stays identical, what becomes real, and what the platform promises about safety.

Deploying

A skript deploys by giving Skode the file: paste it into **Automate → Skript** in your workspace, attached to the workflows it should serve. The file is the deployment artifact — the same bytes you ran locally, which is the whole reason the simulator exists. There is no build step, no transpilation, no environment configuration inside the language.

What stays the same

The language. All of it — parsing, indentation, variables, arithmetic, string interpolation, every condition form, all three loops, functions, `stop`, the error philosophy. A file that passes `skript check` parses in production, full stop, because both run the same grammar. And a file whose trace shows a given sequence of decisions for given inputs will make those same decisions in production for those inputs.

What becomes real

Five things cross from printed to actual, and they are exactly the five you would guess:

In the simulator	In production
events are demo data	events are your real leads, forms, messages, invoices, deals
`score it with ai` simulates from signals	real AI scoring over the full record
`reply` / `send` print the message	messages go out on the real channel
`assign`, `add tag`, `create task` print	records change; tasks appear on real lists
`wait 2 days` is instant, and says so	the run genuinely resumes two days later

Two of these deserve a second sentence. **Scoring:** production scores will not numerically match simulated ones — do not encode simulator-specific thresholds you discovered by trial; encode business thresholds you chose on purpose. **Waiting:** a production `wait` really schedules; the world can change before the continuation runs, so re-check what matters after a long pause rather than trusting a variable set three days ago.

The sandbox promises

Production Skript runs inside the same guardrails you have already met, plus the platform's own:

- **Loops are capped** — the thousand-pass `while` guard is enforced in production exactly as locally, so no skript can spin forever.
- **Recursion is capped** at sixteen levels, same error, same wording.
- **No raw network, no filesystem, no imports.** A skript touches the world only through its nine actions, every one of which is mediated, logged and rate-limited by Skode. There is no sentence in the language that can reach anywhere else — safety by vocabulary, not by review.
- **Runs are isolated.** Each event gets a fresh namespace; a skript cannot corrupt another run's variables, and a failing run fails alone, in its own trace, without taking a queue down with it.
- **Every run leaves a trace.** The same step-by-step account you have been reading locally is recorded for production runs — so "what did the automation do to this lead?" is a lookup, never a mystery.

A deployment ritual that works

Teams converge on some version of this; you may as well start with it.

- `skript check` clean — structural errors never leave the laptop.
- Every branch exercised once with pinned dials (`--score`, `--data`), trace read end to end.
- The awkward-customer runs: empty email, zero budget, weird characters in the name.
- Comments at the top of the file: what event, what it does, which command lines tested it.

- Deploy, then watch the first few *real* traces — not because the logic might differ (it will not) but because real data always has one more shape than you imagined.

That last point is the honest heart of automation work. The language can guarantee its own behaviour, and does; the world guarantees nothing, and the craft is arranging your skripts so the world's surprises land in a `create task` for a human instead of a wrong message to a customer. Skript's contribution is that the arrangement — the actual logic — is never the mystery.

What Now?

You have read the whole language — this is not a figure of speech; Skript is small enough that a tutorial can genuinely be complete — and you have seen it tested, packaged, and deployed. Where to from here depends on which itch brought you.

If you want to build something today: run `skript new`, replace the starter with the automation your team keeps doing by hand, and walk it through the Chapter 16 ritual. The first real skript teaches more than the second reading of any book.

If you want worked material: Appendix B is a cookbook of complete programs — triage, drip sequences, escalation, receipts, win announcements — each one runnable as printed, each one adaptable by changing the quoted strings. Steal freely; that is what it is for.

If you want the terse version of the language: Appendix A is the reference — the grammar in EBNF and every event, action, condition and limit in tables. It says nothing this tutorial has not said, but it says it in two pages, which is what you will want six weeks from now.

If you want the ecosystem: the developer hub at `developer.skode.ai` is the front door — the interactive Skript page, the playground, the marketplace, SDKs for when a problem outgrows sentences, and the community Q&A where the person who hit your exact question yesterday posted the answer. For anything else, `skode.ai/support` reaches humans.

If something in this book was wrong: then a promise failed, because every example was executed against the interpreter during the build, and every output printed is the output produced. Report it on the community hub — a tutorial with executable examples is a tutorial whose bugs are reproducible.

The language will grow — carefully, and sentence-shaped, or not at all. What will not change is the bargain from Chapter 1: what you can say, you can automate, and what you automated, anyone can read. Go say something useful.

Appendix A: Language Reference

This appendix compresses the entire language into tables and one grammar. Nothing here is new; everything here is complete.

The grammar

```
program    = { rule | function | statement } ;
rule       = "when" event-phrase NEWLINE INDENT block DEDENT ;
function   = "to" name-phrase NEWLINE INDENT block DEDENT ;
block      = { statement } ;
statement  = action { "and" action }
           | "if" condition ["then"] block ["otherwise" block]
           | "repeat" NUMBER "times" block
           | ["repeat"] "for each" NAME "in" expr block
           | ["repeat"] "while" condition block ;
action     = built-in-action | name-phrase        (* a function call *) ;
expr       = sum { "," sum }                      (* comma builds a list *) ;
sum        = prod { ("+"|"-") prod } ;
prod       = value { ("*"|"/") value } ;
value      = STRING | NUMBER | "true" | "false" | FIELD ;
condition  = see the conditions table ;
```

Lexical rules: indentation defines blocks (two spaces per level; a tab counts as two spaces); one statement per line; `#` begins a comment outside strings; strings take `"…"` or `'…'` with backslash escapes and `{field}` interpolation; operators require surrounding spaces; keywords are case-insensitive; top-level items (`when` rules, `to` definitions) may appear in any order; a file of bare statements runs in script mode.

Events

Phrase (examples)	Key	Demo fields in the simulator
a new lead arrives / a lead is created	`lead.created`	name, phone, email, source, budget
a form is submitted	`form.submitted`	form, name, email, message
a message arrives	`message.received`	from, name, channel, text
an invoice is paid	`invoice.paid`	number, amount, name
a deal is won	`deal.won`	name, amount, owner
`when "custom.key" happens`	as quoted	none (supply with `--data`)

Actions

Action	Form	Notes
score	`score it with ai`	sets `score` (0–100) and `heat` (hot/warm/cold)
assign	`assign [it] to "Name"`	routes the event's subject
reply	`reply on <channel> "…"`	message inline or as quoted line(s) indented beneath
send	`send <channel> [to "addr"] "…"`	channels: whatsapp, email, sms, instagram, webchat, slack
tag	`add tag "label"`	tags accumulate
task	`create task "title" [for "Name"]`	title interpolates
wait	`wait N second(s)/minute(s)/hour(s)/day(s)`	simulated locally; real delay in production
set	`set name to expr`	creates or replaces a variable
add	`add expr to name`	list → push; number → add; string → append; unset → set
incr	`increase/decrease name by expr`	unset name counts as 0
log	`log "…"` / `say "…"`	trace note; no external effect anywhere
stop	`stop`	halts the run immediately

Conditions

Form	Meaning
`x is v` / `x is not v`	equality / inequality (string-compared for words)
`x is above N` (over, greater than, more than)	numeric >
`x is below N` (under, less than)	numeric <
`x is at least N`	numeric >=
`x contains "t"`	substring, case-insensitive
`x`	truthy: set, non-zero, non-empty, not false
`score is hot/warm/cold`	heat band; requires prior scoring (hot >= 75, warm >= 45)

Conditions are atomic — no `and` / `or` inside one. Compose with nested `if` (conjunction) or sequential `if`s (disjunction).

Data types

Type	Literals	Operations
number	`42`, `3.5`, `-1`	`+ - * /`, schoolbook precedence, left-assoc
string	`"hi"`, `'hi'`	`+` joins (either side); `{name}` interpolates
boolean	`true`, `false`	conditions; bare-name truthiness
list	`v1, v2, v3`	`for each … in`; `add … to` pushes

Coercion: `+` with a string joins as text; other arithmetic requires numbers and errors otherwise. An unset name used as a value is the name itself as a word.

Limits and guarantees

Limit	Value	On breach
`while` iterations	1,000	error naming the loop
function call depth	16	error naming the function
trace display	400 lines	display truncates; execution continues
network / filesystem / imports	none exist	not expressible in the language

The CLI

Command	Purpose
`skript run f.skript [--event k] [--data json] [--score n] [--json]`	execute against a simulated event
`skript check f.skript`	parse only; line-numbered errors; exit 1 on failure
`skript new [name]`	scaffold a starter file
`skript add <name>`	install a marketplace package into the current directory
`skript install-editor`	highlighting + snippets for VS Code-family editors
`skript version` / `skript help`	the obvious

`skode` is an alias for `skript`. Install: `curl -fsSL https://developer.skode.ai/skript/install.sh | bash` (Node >= 18).

Appendix B: Cookbook

Eight complete programs, each solving one recognisable business problem, each runnable exactly as printed (the outputs below are their real runs). Adapt by editing the quoted strings — names, messages, thresholds — and re-running.

1 · The classic: score-and-route triage

The poster program, grown up: three-way routing with a guard for unreachable leads.

```
when a new lead arrives
  score it with ai
  if phone is "" then
    create task "Find contact info for {name}" and stop
  if score is hot then
    assign to "Maya" and reply on whatsapp
      "Thanks {name}! We'll call you in 10 minutes."
  otherwise
    if score is warm then
      assign to "Sam" and add tag "follow-up-today"
    otherwise
      add tag "nurture"
```

```
▶ example.skript — Skode Skript · local simulation

  ● event: lead.created               → name: Ada · phone: +1 555 0100 · email: ada@example.com
  ⚡ score it with ai                  → score 84 · HOT
  ✗ if phone is ""                    → no
  ✓ if score is hot                   → yes
  → assign to "Maya"                  → lead assigned
  ✉ reply on whatsapp                 → "Thanks Ada! We'll call you in 10 minutes."

  ✔ done in 2.1ms — 1 message · assigned to Maya
```

2 · Form follow-up with a task trail

```
when a form is submitted
  score it with ai
  add tag "form: {form}"
  if score is hot then
    assign to "Maya"
    send email to "maya@skode.ai" "Hot form lead: {name} — {message}"
    create task "Reply personally within the hour" for "Maya"
  otherwise
    reply on email "Thanks {name} — we'll be in touch this week."
```

```
▶ example.skript — Skode Skript · local simulation

  ● event: form.submitted             → form: Contact us · name: Grace Hopper · email: grace@example.com
  ⚡ score it with ai                  → score 88 · HOT
  ⚑ add tag "form: Contact us"        → tagged
  ✓ if score is hot                   → yes
  → assign to "Maya"                  → submission assigned
  ✉ send email to "maya@skode.ai"     → "Hot form lead: Grace Hopper — Interested in the Pro plan"
  ☑ create task "Reply personally within the hour" → created

  ✔ done in 2.1ms — 1 message · assigned to Maya · tags: form: Contact us · 1 task
```

3 · A drip sequence

`wait` makes time a step. Locally the waits are instant and labelled; in production this is a real three-touch sequence.

```
when a new lead arrives
  reply on email "Welcome, {name}! Here's what Skode can do."
  wait 2 days
  reply on email "A quick case study you might like, {name}."
  wait 3 days
  reply on email "Any questions? Just hit reply."
  add tag "drip-complete"
```

```
▶ example.skript — Skode Skript · local simulation

  ● event: lead.created               → name: Priya · phone: +1 555 0100 · email: ada@example.com
  ✉ reply on email                    → "Welcome, Priya! Here's what Skode can do."
  ◌ wait 2 days                       → (simulated — instant here, real delay in Skode)
  ✉ reply on email                    → "A quick case study you might like, Priya."
  ◌ wait 3 days                       → (simulated — instant here, real delay in Skode)
  ✉ reply on email                    → "Any questions? Just hit reply."
  ⚑ add tag "drip-complete"           → tagged

  ✔ done in 2.1ms — 3 messages · tags: drip-complete
```

4 · Invoice receipts, with a VIP touch

```
when an invoice is paid
  reply on email "Receipt for invoice {number} — thank you!"
  add tag "paid"
  if amount is above 5000 then
    create task "Send a handwritten thank-you to {name}" for "Maya"
    add tag "vip"
```

```
▶ example.skript — Skode Skript · local simulation

  ● event: invoice.paid               → number: INV-0461 · amount: 9500 · name: Margaret H
  ✉ reply on email                    → "Receipt for invoice INV-0461 — thank you!"
  ⚑ add tag "paid"                    → tagged
  ✓ if amount above 5000              → yes
  ☑ create task "Send a handwritten thank-you to {name}" → created
  ⚑ add tag "vip"                     → tagged

  ✔ done in 2.1ms — 1 message · tags: paid, vip · 1 task
```

5 · Message keyword router

Sequential `if`s make a tidy keyword switchboard; functions keep each route readable.

```
when a message arrives
  if text contains "pricing" then
    route to sales
  if text contains "refund" then
    route to support
  if text contains "partner" then
    create task "Partnership inquiry from {name}" for "Ops"

to route to sales
  assign to "Maya" and add tag "sales-inquiry"
  reply on whatsapp "Great question, {name} — connecting you with Maya now."

to route to support
  assign to "Sam" and add tag "support"
  reply on whatsapp "Sorry to hear that, {name} — Sam is on it."
```

```
▶ example.skript — Skode Skript · local simulation

  ● event: message.received            → from: +1 555 0101 · name: Lin · channel: whatsapp
  ✓ if text contains "pricing"         → yes
  ƒ route to sales                     → run your function
  → assign to "Maya"                   → message assigned
  ⚑ add tag "sales-inquiry"            → tagged
  ✉ reply on whatsapp                  → "Great question, Lin — connecting you with Maya now."
  ✗ if text contains "refund"          → no
  ✗ if text contains "partner"         → no

  ✔ done in 2.1ms — 1 message · assigned to Maya · tags: sales-inquiry
```

6 · Deal-won celebration and handover

```
when a deal is won
  send slack "{owner} closed {name} for {amount}!"
  create task "Kick off onboarding for {name}" for "Success Team"
  add tag "customer"
  set commission to amount / 20
  log "commission earned: {commission}"
```

```
▶ example.skript — Skode Skript · local simulation

  ● event: deal.won                    → name: Skylark Corp · amount: 14000 · owner: Maya
  ✉ send slack                         → "Maya closed Skylark Corp for 14000!"
  ☑ create task "Kick off onboarding for {name}" → created
  ⚑ add tag "customer"                 → tagged
  ✎ set commission                     → = 700
  • log                                → commission earned: 700

  ✔ done in 2.1ms — 1 message · tags: customer · 1 task
```

7 · Multi-channel escalation sweep

The pattern for "make sure someone sees this": alert on every channel, then log an audit line. Note that a channel is part of the *sentence*, not a value — `send slack …` — so each channel gets its own explicit send (a feature, for auditability: the reader sees exactly where alerts go). A function keeps the sweep tidy, and a small loop assembles the audit trail:

```
when a new lead arrives
  score it with ai
  if score is above 95 then
    sound the alarm
    create task "Drop everything and call {name}" for "Maya"
    set alerted to ""
    for each ch in "slack", "email", "sms"
      add "{ch} " to alerted
    log "alerted via: {alerted}"

to sound the alarm
  send slack "URGENT: {name}, budget {budget} — score {score}"
  send email to "maya@skode.ai" "URGENT: {name}, budget {budget}"
  send sms "URGENT lead: {name}"
```

```
▶ example.skript — Skode Skript · local simulation

  ● event: lead.created                → name: Whale Corp · phone: +1 555 0100 · email: ada@example.com
  ⚡ score it with ai                   → score 97 · HOT
  ✓ if score above 95                  → yes
  ƒ sound the alarm                    → run your function
  ✉ send slack                         → "URGENT: Whale Corp, budget 80000 — score 97"
  ✉ send email to "maya@skode.ai"      → "URGENT: Whale Corp, budget 80000"
  ✉ send sms                           → "URGENT lead: Whale Corp"
  ☑ create task "Drop everything and call {name}" → created
  ✎ set alerted                        → = ""
  ↻ for each ch in ["slack", "email", "sms"] → 3 items
  ✎ add to alerted                     → = "slack "
  ✎ add to alerted                     → = "slack email "
  ✎ add to alerted                     → = "slack email sms "
  • log                                → alerted via: slack email sms

  ✔ done in 2.1ms — 3 messages · 1 task
```

8 · The self-auditing rule

The pattern that makes on-call life bearable: narrate decisions as you make them, and end with a summary a human can read in the run log.

```
when a new lead arrives
  score it with ai
  set verdict to "undecided"
  if score is hot then
    set verdict to "routed to Maya" and assign to "Maya"
  if score is warm then
    if budget is above 500 then
      set verdict to "warm but funded - Sam" and assign to "Sam"
    otherwise
      set verdict to "nurture list" and add tag "nurture"
  if score is cold then
    set verdict to "archived" and add tag "cold"
  log "AUDIT {name} from {source}: score {score} ({heat}) -> {verdict}"
```

```
▶ example.skript — Skode Skript · local simulation

  ● event: lead.created                → name: Kiran · phone: +1 555 0100 · email: ada@example.com
  ⚡ score it with ai                   → score 55 · WARM
  ✎ set verdict                        → = "undecided"
  ✗ if score is hot                    → no
  ✓ if score is warm                   → yes
  ✓ if budget above 500                → yes
  ✎ set verdict                        → = "warm but funded - Sam"
  → assign to "Sam"                    → lead assigned
  ✗ if score is cold                   → no
  • log                                → AUDIT Kiran from webinar: score 55 (warm) -> warm but funded - Sam

  ✔ done in 2.1ms — assigned to Sam
```

9 · Budget-band pricing assistant

Thresholds as data: one list, one loop, one decision — change the bands without touching the logic.

```
when a new lead arrives
  set band to "starter"
  for each threshold in 1000, 2500, 10000
    if budget is at least threshold then
      set band to "band-{threshold}"
  add tag "{band}"
  log "{name} with budget {budget} lands in {band}"
```

```
▶ example.skript — Skode Skript · local simulation

  ● event: lead.created                → name: Kiran · phone: +1 555 0100 · email: ada@example.com
  ✎ set band                           → = "starter"
  ↻ for each threshold in [1000, 2500, 10000] → 3 items
  ✗ if budget is "at least threshold"  → no
  ✗ if budget is "at least threshold"  → no
  ✗ if budget is "at least threshold"  → no
  ⚑ add tag "starter"                  → tagged
  • log                                → Kiran with budget 2600 lands in starter

  ✔ done in 2.1ms — tags: starter
```

(A subtlety worth stealing: the loop runs low-to-high and simply*overwrites* `band` at each threshold the budget clears, so the final value is the highest band cleared — no nested `if`s required. Note `add tag "{band}"` interpolating a variable into the label.)

10 · Weekly-source hygiene check

Not every skript talks to customers. This one polices data quality and files the paperwork for a human — automation as janitor.

```
when a new lead arrives
  set problems to ""
  if source is "" then
    add "no-source " to problems
  if email is "" then
    add "no-email " to problems
  if problems is not "" then
    add tag "needs-cleanup"
    create task "Fix lead data for {name}: {problems}" for "Ops"
    log "flagged: {problems}"
  otherwise
    log "clean record"
```

```
▶ example.skript — Skode Skript · local simulation

  ● event: lead.created                → name: Ghost Lead · phone: +1 555 0100 · email:
  ✎ set problems                       → = ""
  ✓ if source is ""                    → yes
  ✎ add to problems                    → = "no-source "
  ✓ if email is ""                     → yes
  ✎ add to problems                    → = "no-source no-email "
  ✓ if problems is not ""              → yes
  ⚑ add tag "needs-cleanup"            → tagged
  ☑ create task "Fix lead data for {name}: {problems}" → created
  • log                                → flagged: no-source no-email

  ✔ done in 2.1ms — tags: needs-cleanup · 1 task
```

11 · The polite decliner

Saying no is an automation too. Small budgets get a kind message and a future-proof tag instead of silence:

```
when a new lead arrives
  score it with ai
  if budget is below 100 then
    reply on email
      "Thanks for your interest, {name}!"
      "We're likely overkill for your current needs - but keep us in mind as you grow."
    add tag "future-fit" and stop
  add tag "qualified"
```

```
▶ example.skript — Skode Skript · local simulation

  ● event: lead.created                → name: Tiny Co · phone: +1 555 0100 · email: ada@example.com
  ⚡ score it with ai                   → score 30 · COLD
  ✓ if budget below 100                → yes
  ✉ reply on email                     → "Thanks for your interest, Tiny Co!\nWe're likely overkill for your
current needs - but keep us in mind as you grow."
  ⚑ add tag "future-fit"               → tagged
  ■ stop                               → skript halted

  ✔ done in 2.1ms — 1 message · tags: future-fit
```

Every recipe here is four ideas deep at most — an event, a decision, a handful of actions, and a note for the humans. That is not because the examples are simplified; it is because that is the honest shape of good automation. When yours grows past it, reach for a function and give the complexity a name.

Appendix C: Glossary

action — a step that does something: `reply`, `send`, `assign`, `add tag`, `create task`, `wait`, `set`, `log`, `score it with ai`, `stop`. The verbs of the language.

band — see *heat*.

block — the indented lines belonging to a rule, function, `if`, `otherwise`, or loop. Blocks are made by indentation alone.

chain — several actions joined on one line with `and`. Pure sequencing; the steps do not depend on each other.

channel — where a message travels: whatsapp, email, sms, instagram, webchat or slack. Part of the sentence, not a value.

check — `skript check`, the parse-only command. Catches structural errors without running anything; exits 1 on failure.

condition — the test inside an `if` or `while`: `x is v`, `is above/below/at least N`, `contains "t"`, bare-name truthiness, or `score is hot/warm/cold`. Always atomic — no `and` / `or` inside one.

continuation string — the quoted line(s) indented beneath a `reply`, joined into its message. The only place a message may leave its own line.

demo event — the realistic stand-in data the simulator supplies for each event kind, overridable field-by-field with `--data`.

event — the thing that happened: a lead arrived, a form was submitted, a message came in, an invoice was paid, a deal was won — or any custom quoted key. Every rule begins with one.

field — a named piece of an event's data (`name`, `budget`, `text`). Fields and variables share one namespace.

function — a named block defined at top level with `to <name>`, called by writing its name as a sentence. No parameters, no return value; reads and writes the shared namespace.

heat — the band derived from *score*: hot (75+), warm (45–74), cold (below 45). Read with `score is hot` and friends.

interpolation — `{name}` inside a string, replaced with that name's value when the string is used. Unset names keep their braces, on purpose.

list — an ordered collection written as comma-separated values. Consumed with `for each`; grown with `add … to`.

namespace — the single pool of names a run can see: event fields, your variables, and (once scoring runs `score` and `heat`.

rule — `when <event>` plus its block. The fundamental unit of a skript.

sandbox — the guardrails around every run: capped `while` loops (1,000), capped recursion (16), no network/filesystem/imports, isolated per-run state.

score — the 0–100 number produced by `score it with ai`. Simulated deterministically from signals locally; real AI in production. Pin it with `--score`.

script mode — what happens when a file has bare steps and no rules: they run top to bottom against a minimal stand-in event. The language's scratchpad.

signals — the facts the simulated scorer reads (phone, email, budget, source, intent words), listed in its trace line so scores explain themselves.

stop — the emergency brake: halts the entire run immediately, from anywhere. A decision, not an error.

trace — the step-by-step account every run produces: `●` the event, `✓` / `✗` decisions, one line per action, `✔` the summary. Recorded in production too.

truthy — what a bare name asks in a condition: set, non-zero, non-empty, and not false.

unset-name rule — an unset name used as a *value* is just its own word; an unknown *step* is an error. Values flow, verbs stop.

variable — a named container made with `set`, changed with `set` / `increase` / `decrease` / `add … to`, visible in every trace line that touches it.

Appendix D: Skript for Python and JavaScript Programmers

If you arrive with a programming language already in your head, this appendix is your fastest route in: the same intent, side by side. (Python shown; the JavaScript translation is the obvious one.)

Intent	Python	Skode Skript
assignment	`counter = 0`	`set counter to 0`
increment	`counter += 1`	`increase counter by 1`
string build	`f"Hi {name}"`	`"Hi {name}"`
concatenation	`"a" + str(n)`	`"a" + n`
list literal	`["a", "b"]`	`"a", "b"`
append	`xs.append(v)`	`add v to xs`
conditional	`if x > 500:` … `else:`	`if x is above 500 then` … `otherwise`
substring test	`"vip" in s.lower()`	`s contains "vip"`
counted loop	`for _ in range(3):`	`repeat 3 times`
iteration	`for ch in chs:`	`for each ch in chs`
while	`while c < 5:`	`while c is below 5`
function	`def welcome():`	`to welcome the lead`
call	`welcome()`	`welcome the lead`
early exit	`return` / `sys.exit()`	`stop`
comment	`# note`	`# note`
print	`print(x)`	`log "{x}"`

The deeper translations are about what is*missing*, and each absence follows from the language's purpose:

- **No parameters or return values.** Skript functions share one namespace with the event — "set the stage, then say the sentence." Where you would design a signature, you name a variable.
- **No boolean operators in conditions.** `and` / `or` compose *steps*, not tests. Conjunction is nesting; disjunction is two `if`s; complex predicates become a well-named function.
- **No objects, dicts, indexing, imports, or I/O.** The event is the record; the nine actions are the I/O; the marketplace is the import system. If you reach for a dictionary, the design is asking for Skode itself (or the SDK) rather than a skript.
- **No exceptions to catch.** Errors halt the run with an English sentence and a line number. Guard clauses `if email is "" then … stop`) replace try/except, and the sandbox replaces defensive programming.

The mental shift that makes Skript pleasant rather than confining: you are not writing an application, you are writing *policy* — the sentence a good operations manager would say out loud. Python remains the right tool for computing things. Skript is the right tool for deciding things, and it will refuse, politely and by design, to become the former.