# The Zeno Book

The whole language, from the first line
to the last corner

Partha

For Zeno version 0.1

*Every journey is halfway done. Forever.*

**The Zeno Book**, for Zeno version 0.1.

Written by Partha, 2026.

Every program in this book was run with the Zeno compiler while the book was built, and every output shown is what it printed. The source of the book is in the `book/` folder of the Zeno repository, and one command builds both this PDF and the HTML version from it.

https://zeno-lang.org

# Contents


**Preface** xiii

**I Starting** 1

**1 Getting Zeno** 2

1.1 What you need . . . 2
1.2 Building it . . . 2
1.3 Your first project . . . 3
1.4 The commands . . . 3
1.5 An editor . . . 3

**2 A First Program** 5

2.1 Hello . . . 5
2.2 Comments . . . 5
2.3 One line, one statement . . . 5
2.4 Names for things . . . 6
2.5 When something is wrong . . . 6
2.6 Checking without running . . . 8

**3 A Tour of Zeno** 9

3.1 Values that change, and values that do not . . . 9
3.2 Lists, and a comprehension . . . 9
3.3 Functions, with types or without . . . 9
3.4 Classes . . . 10
3.5 Enums that carry values, and match . . . 10
3.6 No surprise none . . . 11
3.7 Errors that are checked . . . 12
3.8 Functions are values . . . 12
3.9 Dicts . . . 12
3.10 Numbers with units, and numbers with a size . . . 13
3.11 Generics . . . 13
3.12 And on a board . . . 13

**II The Language** **15**

**4 Values and Variables** **16**
4.1 The kinds of value . . . 16
4.2 let: a name for a value . . . 16
4.3 var: a name whose value changes . . . 17
4.4 const: a value fixed for the whole program . . . 17
4.5 Types are optional . . . 18
4.6 Names . . . 19
4.7 Blocks and scope . . . 19
4.8 Two names for one list . . . 20

**5 Numbers and Operators** **22**
5.1 Writing numbers . . . 22
5.2 Arithmetic . . . 22
5.3 Mixing ints and floats . . . 24
5.4 Changing kind on purpose . . . 24
5.5 Comparing . . . 25
5.6 True and false . . . 26
5.7 Changing a variable: the = forms . . . 26
5.8 Which operator goes first . . . 27
5.9 Built-in functions for numbers . . . 27

**6 Text** **29**
6.1 Characters that cannot be typed . . . 29
6.2 Characters, not bytes . . . 29
6.3 Joining and comparing . . . 30
6.4 Formatted strings . . . 30
6.5 The methods of text . . . 32
6.6 A loop over text . . . 34
6.7 Text from other values . . . 34

**7 Deciding and Repeating** **36**
7.1 if . . . 36
7.2 while . . . 37
7.3 for . . . 37
7.4 break and continue . . . 39
7.5 Loops inside loops . . . 39
7.6 A worked example: primes . . . 40

**8 Functions** **41**
8.1 Declaring and calling . . . 41
8.2 Types are optional here too . . . 41
8.3 Returning on every path . . . 42
8.4 Default values . . . 43
8.5 Named arguments . . . 43
8.6 Parameters do not change . . . 44
8.7 Calling a function above where it is written . . . 45
8.8 Functions that call themselves . . . 45
8.9 Functions inside functions . . . 45

**9 Lists** **48**
9.1 Making a list and reading it . . . 48
9.2 Changing a list . . . 48
9.3 Every method of a list . . . 49
9.4 Joining lists . . . 51
9.5 Comprehensions . . . 51
9.6 map, filter and reduce . . . 52
9.7 Two names, one list . . . 52
9.8 Equal lists . . . 53

**10 Dicts** **54**
10.1 Writing a dict . . . 54
10.2 Keys and values . . . 54
10.3 Reading . . . 55
10.4 Storing and taking out . . . 56
10.5 Order . . . 57
10.6 Looping over a dict . . . 57
10.7 Equal dicts . . . 57
10.8 Every method of a dict . . . 58

**11 Classes** **59**
11.1 Fields . . . 59
11.2 Changing a field . . . 60
11.3 Methods . . . 61
11.4 Making an object with a function . . . 62
11.5 Objects are shared . . . 62
11.6 Equal objects . . . 63
11.7 Objects inside objects . . . 63

11.8 A class with a list . . . 63
11.9 No parent classes . . . 64
11.10 Using a class above where it is written . . . 65

**12 Enums and match** **66**
12.1 An enum of plain choices . . . 66
12.2 match . . . 66
12.3 Cases that carry values . . . 68
12.4 match on numbers and text . . . 68
12.5 Blocks, not arrows . . . 69
12.6 An enum in a state machine . . . 70
12.7 Case names belong to the file . . . 70

**13 Values That May Be Missing** **72**
13.1 none only where the type says so . . . 72
13.2 Asking: is none, is not none . . . 73
13.3 What the checker follows . . . 74
13.4 Functions that may give none . . . 76
13.5 none in a match . . . 76
13.6 none is not zero . . . 77

**14 When Things Go Wrong** **78**
14.1 Bugs and errors . . . 78
14.2 An error that is not caught . . . 78
14.3 try and except . . . 78
14.4 The kinds of error . . . 79
14.5 raise . . . 80
14.6 raises: what a function may raise . . . 81
14.7 An error that carries more . . . 82
14.8 What is not here . . . 83

**15 Functions as Values** **84**
15.1 Passing a function . . . 84
15.2 The type of a function . . . 84
15.3 A function with no name . . . 85
15.4 Functions in lists and dicts . . . 85
15.5 Functions that make functions . . . 86
15.6 Calling something that is not a function . . . 87

**III Growing Programs** **89**

**16 Types, Written and Worked Out** **90**
16.1 The types there are . . . 90
16.2 Writing a type . . . 90
16.3 What Zeno works out . . . 91
16.4 When it cannot be worked out . . . 93
16.5 Strict mode . . . 93
16.6 No quiet conversions . . . 94
16.7 Only none can be added to a type . . . 95

**17 Numbers with Units** **96**
17.1 Writing a quantity . . . 96
17.2 What a quantity can do . . . 97
17.3 Units of one kind work together . . . 97
17.4 A unit is a type . . . 98
17.5 Time . . . 99
17.6 What does not convert . . . 99
17.7 Showing a quantity . . . 99
17.8 What units leave out . . . 100

**18 Sized Numbers and Bits** **101**
18.1 Writing bits . . . 101
18.2 The bit operators . . . 101
18.3 Sized whole numbers . . . 103
18.4 Putting a number back: check or wrap . . . 104
18.5 What goes where by itself . . . 105
18.6 A worked example: a frame from a sensor . . . 105

**19 Generics** **108**
19.1 A generic function . . . 108
19.2 Several type parameters . . . 109
19.3 A generic class . . . 109
19.4 Where the types come from . . . 110
19.5 The body is checked for each type . . . 110
19.6 What a compiled program does . . . 111
19.7 What generics leave out . . . 111

**20 Interfaces** **113**
20.1 Declaring an interface . . . 113

20.2 The rules . . . 114
20.3 Where an interface goes . . . 115
20.4 Through an interface, only its functions . . . 116
20.5 Errors through an interface . . . 116
20.6 No parent classes . . . 117
20.7 What it costs . . . 118

**21 Programs in Several Files** **119**
21.1 A module is a file . . . 119
21.2 Two ways to import . . . 120
21.3 The rules . . . 120
21.4 private . . . 122
21.5 How it works . . . 124

**22 Packages** **125**
22.1 Starting a program with zeno new . . . 125
22.2 The project file . . . 125
22.3 A package beside the program . . . 126
22.4 Other people's packages: zeno fetch . . . 127
22.5 What can go wrong . . . 128

**23 Reading and Writing** **129**
23.1 print . . . 129
23.2 Reading lines: input . . . 129
23.3 The command line . . . 130
23.4 Files . . . 131
23.5 A worked example: a log book . . . 132

**24 Mathematics** **134**
24.1 The functions that need no import . . . 134
24.2 The math module . . . 134
24.3 Angles . . . 135
24.4 What is not here . . . 136
24.5 Five small instruments . . . 136

**25 Testing** **140**
25.1 A first test . . . 140
25.2 assert . . . 140
25.3 Testing that something fails . . . 141
25.4 Where tests go . . . 142

25.5 Testing code that uses the board . . . 143
25.6 What tests leave out . . . 143

**26 The Tools** **145**
26.1 The commands . . . 145
26.2 run and check . . . 145
26.3 format . . . 146
26.4 Seeing what the compiler sees . . . 147
26.5 Your editor . . . 148

**IV Zeno on a Machine** **150**

**27 The Board** **151**
27.1 Blink . . . 151
27.2 The simulated board . . . 151
27.3 Pins . . . 152
27.4 The clock . . . 152
27.5 Registers . . . 153
27.6 A traffic light . . . 154
27.7 A real board underneath . . . 155
27.8 What the board does not have yet . . . 156

**28 Compiling** **157**
28.1 zeno build . . . 157
28.2 Reading the C . . . 157
28.3 Build agrees with run . . . 158
28.4 A build is strict . . . 158
28.5 The options . . . 159
28.6 Through LLVM . . . 159
28.7 For another processor . . . 160
28.8 What does not compile yet . . . 160

**29 Memory** **162**
29.1 What takes memory . . . 162
29.2 The question nobody can answer in advance . . . 162
29.3 Mark and sweep . . . 162
29.4 Seeing it work . . . 163
29.5 When it runs . . . 164
29.6 What it costs . . . 165

29.7 Three ways to manage memory . . . 165

**30 Ownership** **166**
30.1 Three words . . . 166
30.2 The rules . . . 167
30.3 Branches and loops . . . 168
30.4 Seeing the frees . . . 169
30.5 What owned values cannot do . . . 171

**31 Firmware** **172**
31.1 Errors as values . . . 172
31.2 Faults . . . 173
31.3 A real board underneath . . . 174
31.4 What the C of blink looks like . . . 174
31.5 What this does not claim . . . 175

**32 Building with No Collector** **176**
32.1 Why . . . 176
32.2 Every list and object has one owner . . . 176
32.3 An ordinary program, built so . . . 177
32.4 Where memory comes from . . . 178
32.5 What is refused . . . 178
32.6 The tour . . . 179
32.7 What this does not claim . . . 179

**33 Unsafe Code** **181**
33.1 The block . . . 181
33.2 Pointers . . . 182
33.3 What zeno run catches . . . 182
33.4 Registers . . . 183
33.5 A driver . . . 183
33.6 Memory by hand, and the other two . . . 186

**V Inside Zeno** **188**

**34 Inside Zeno** **189**
34.1 The stages . . . 189
34.2 The lexer . . . 189
34.3 The parser . . . 190
34.4 Modules . . . 190

34.5 The checker . . . 190
34.6 Two engines that run . . . 190
34.7 Two engines that compile . . . 191
34.8 Agreement . . . 192
34.9 The files . . . 192

**35 Why Zeno Is as It Is** **194**
35.1 Unambiguous and intuitive . . . 194
35.2 The five principles . . . 194
35.3 What came from where . . . 194
35.4 What was left out, and why . . . 195
35.5 How decisions were made . . . 195
35.6 What 0.1 does not have . . . 196
35.7 The name . . . 196

**A Keywords** **197**

**B Operators** **199**

**C The Library at a Glance** **201**

**D Units at a Glance** **204**

**E The zeno Command** **205**
E.1 Commands . . . 205
E.2 Options . . . 206
E.3 What a command answers . . . 206
E.4 Telling the C compiler . . . 206

**F Kinds of Error** **208**
F.1 Before the program runs . . . 208
F.2 The kinds raised while it runs . . . 208
F.3 When nothing catches it . . . 209
F.4 Faults in a firmware build . . . 209
F.5 Messages from memory . . . 209

**G What Version 0.1 Leaves Out** **211**
G.1 On a board . . . 211
G.2 Numbers . . . 211
G.3 Types . . . 211
G.4 The library . . . 211

G.5 Builds . . . . . 212
G.6 Firmware and no collector . . . . . 212
G.7 Interfaces . . . . . 212
G.8 Tools . . . . . 212
G.9 Not limits . . . . . 213

# Preface

Zeno is a programming language that reads like plain English and is checked like Rust. It was built from nothing, one stage at a time, by a firmware engineer who wanted to know how a language works from the inside: the lexer, the parser, the type checker, two engines that run programs, two compilers that turn them into machine code, and the library and tools around them. This book is the whole of that language, written down once it stopped moving.

## Who this book is for

The book starts from nothing. If you have never written a program, start at the first page and read in order: every chapter uses only what the chapters before it showed you.

If you already know another language, you will find Zeno familiar quickly. It takes its spelling from Python wherever Python's spelling is clear, its braces from C, its checking from Rust, its sized numbers from firmware, and its error messages from the idea that a message should say what to write instead. Read Part I quickly, skim Part II for the places where Zeno differs from what you know, and slow down from Part III on.

If you write firmware, Part IV is for you: it shows how one Zeno program builds for a computer, for a firmware coding standard with no `setjmp`, with no garbage collector at all, and with memory handled by hand, and how each of those builds is told apart.

If you want to know how the compiler itself works, Part V walks through it, file by file.

## How the book is laid out

**Part I, Starting** gets Zeno onto your machine, runs a first program, and takes a quick tour of the whole language so you know what is coming.

**Part II, The Language** is everything a program is made of: values, numbers, text, decisions and loops, functions, lists, dicts, classes, enums, values that may be missing, errors, and functions used as values.

**Part III, Growing Programs** is what a program needs once it is more than a page: types written and worked out, units, sized numbers and bits, generics, interfaces, modules, packages, files, mathematics, tests, and the tools.

**Part IV, Zeno on a Machine** is about where a program runs: the board and its pins, compiling to C and to LLVM, memory and the collector, owned values, firmware builds, builds with no collector, and memory by hand.

**Part V, Inside Zeno** is how the compiler is built and how the language is designed.

**The appendices** are for looking things up: every keyword, every operator, the whole library, every unit, every command, every kind of error, and the list of what version 0.1 leaves out.

Each chapter ends with *Things to try*: small changes to make and questions to answer by running Zeno. They are the fastest way to make a chapter your own.

## The programs in this book are real

Every program in this book was run by the Zeno compiler when the book was built, and every output under a program is what Zeno printed. That includes the error messages: when the book shows a mistake, the message under it is the one Zeno gives. If a later version of Zeno prints something else, the book is rebuilt and the page changes with it.

A program looks like this:

```
print("Hello, Zeno!")
```

```
Hello, Zeno!
```

and what it printed is in the paler box under it. When a program is written to a file of its own name, the name is shown above it. A command typed into a terminal starts with `$`, and its output follows on the next lines.

## The boxes

> **Note.** A note says something worth knowing that the text around it does not stop to explain.

> **Why.** Zeno has a recorded reason for every choice in it. A box like this gives the reason for the rule beside it, and often what was turned down. All of the reasons are kept in `docs/decisions.md` in the Zeno repository, with a number for each, and the book names the number when it is useful.

## The rule above the rules

One sentence decides more about Zeno than any other: *the entire language must be unambiguous and intuitive*. In practice that means five things, and you will see each of them again and again in this book:

1. There is one obvious way to write each thing.
2. No line can be read two ways, by the compiler or by a person.
3. Nothing happens that the line does not say.
4. Every word is a plain word, with one meaning.
5. When Zeno refuses something, the message says what to write instead.

## The name

Zeno of Elea argued that Achilles can never catch the tortoise: before he reaches it he must cover half the distance, then half of what is left, and so on forever. The quantum Zeno effect says that a system watched closely enough never changes. Both suit a compiler that will always be almost finished. Its motto is the first thing `zeno --help` prints:

> *Every journey is halfway done. Forever.*

PART I

# Starting

CHAPTER 1

# Getting Zeno

Zeno is one program, called `zeno`. It checks your programs, runs them, compiles them, tests them, lays them out, and talks to your editor. This chapter puts it on your machine.

## 1.1 What you need

Zeno is written in Rust, and you build it from its source. You need:

- **Rust**, with its build tool `cargo`. The usual way to get both is `rustup`, from `https://rustup.rs`. Zeno uses no libraries beyond Rust itself, so nothing else is downloaded when it is built.
- **git**, to get the source, and later to fetch packages that live in other people's repositories.
- **A C compiler**, `cc`, `gcc` or `clang`, if you want to compile Zeno programs into programs that run on their own (Compiling). `zeno run` needs none.
- **clang**, only if you want the second compiler, which goes through LLVM.

Zeno is built and tested on Linux, macOS and Windows, with the same steps on each. On Windows, every path Zeno writes uses `/`, as it does elsewhere, so a package or a message reads the same on every machine.

## 1.2 Building it

Get the source and build it:

```
git clone https://github.com/partham20/zeno-lang
cd zeno-lang
cargo build --release
```

The finished program is `target/release/zeno`. To use it from any folder, either put that folder on your `PATH`, or let cargo copy the program into its own folder of programs, which `rustup` has already put on your `PATH`:

```
cargo install --path .
```

Check that it works:

```
$ zeno --version
zeno 0.1.0
```

If you want to be sure that the compiler you built is a good one, run its own tests. There are several hundred of them, and each program in them is run by every engine and built every way Zeno can build it:

```
cargo test --release
```

## 1.3 Your first project

`zeno new` makes a folder with a small program in it, ready to run:

```
$ zeno new greeter
Made the program `greeter` in greeter/
  zeno.package   the name of the program, and the packages it needs
  main.zeno      the program
  greeting.zeno  a module, with a test
Run it with `zeno run greeter/main.zeno`, and its tests with `zeno test greeter`.
$ zeno run greeter/main.zeno
Hello, world, from greeter.
$ zeno test greeter
greeter/greeting.zeno
  ok      hello says who it greets

1 test: 1 passed, 0 failed.
```

Three files were made. `main.zeno` is where the program starts. `greeting.zeno` is a *module*, a second file of the program, with a test at its end. `zeno.package` names the program and, later, the packages it needs. You will meet each of them properly: modules in Programs in Several Files, packages in Packages, and tests in Testing. For now, a program can be a single file with any name.

## 1.4 The commands

Everything Zeno does is a word after `zeno`. These are the ones you will use most; The zeno Command has all of them with every option.

| Command | What it does |
|---|---|
| `zeno run file.zeno` | Check the program, and run it if there is nothing wrong |
| `zeno check file.zeno` | Only check it, and report every mistake |
| `zeno test` | Run the tests written in the files of a folder |
| `zeno build file.zeno` | Compile it to C, and then to a program of its own |
| `zeno format file.zeno` | Lay the file out the one way Zeno lays programs out |
| `zeno new name` | Make a new program in a new folder |
| `zeno fetch` | Get the packages a program needs |
| `zeno lsp` | The language server that an editor starts |

## 1.5 An editor

Any editor that can save plain text will do. Zeno files end in `.zeno` and are UTF-8 text. The Zeno repository has colours and setup for VS Code, Vim, Neovim and Helix in its `editors` folder, and `zeno lsp` lets those editors show mistakes as you type, jump to where a name was declared, show a name's type, and lay the file out when you save it. The Tools shows how to set each of them up.

### Things to try

1. Run `zeno --help` and read the list of commands. Which ones need a C compiler?
2. Change the text that `greeting.zeno` gives back, and run `zeno test greeter` again. Read what a failed test prints, then change the test so that it passes.
3. Run `zeno new 2fast` and `zeno new while`. What does Zeno say, and what does it suggest instead?

CHAPTER 2

# A First Program

## 2.1 Hello

Make a file called `hello.zeno` with one line in it:

hello.zeno

```
print("Hello, Zeno!")
```

```
$ zeno run hello.zeno
Hello, Zeno!
```

That is the whole program. There is no `main` function to write, nothing to import and no semicolon at the end. Zeno starts at the top of the file and does each line in turn, and when it runs out of lines the program is over.

`print` shows what it is given and then ends the line. Give it several things and it puts a space between each; give it nothing and it prints an empty line:

```
print("one", "two", 3)
print()
print("after an empty line")
```

```
one two 3

after an empty line
```

Text is written between double quotes, `"like this"`. Numbers are written as they are. Each has a chapter of its own later.

## 2.2 Comments

A `#` starts a comment. Everything from it to the end of the line is for the reader, and Zeno skips it:

```
# Greet the world. This whole line is a comment.
print("Hello")   # and so is the end of this one
```

```
Hello
```

## 2.3 One line, one statement

Each line is one *statement*: one thing for the program to do. The end of the line ends the statement, so Zeno needs no `;`.

A long statement can go on to the next line when its first line is clearly not finished. That is so

inside brackets, and after anything that cannot end a statement, such as +, , or =:

```
let total = 1 +
    2 +
    3
print(total,
      "is the total")
```

```
6 is the total
```

The other way round does not work. A line that could stand on its own *does* stand on its own, so a line that starts with + is a mistake:

```
let total = 1
    + 2
```

```
error: expected an expression, found `+`
 --> main.zeno:2:5
  |
2 |     + 2
  |     ^
```

**Why.** The line break already means "next statement" to the person reading, so a ; on every line only says it again. The rule for going on to the next line, "an unfinished line continues", can never be misread, because a line that looks finished is finished.

## 2.4 Names for things

`let` gives a value a name, and from then on the name stands for the value:

```
let name = "Ada"
let year = 1843
print(name, "wrote the first program in", year)
print(f"Hello, {name}!")
```

```
Ada wrote the first program in 1843
Hello, Ada!
```

The last line is a *formatted string*. The `f` in front of the quotes says that what stands inside `{ }` is to be worked out and put into the text. Values and Variables and Text say much more about both.

## 2.5 When something is wrong

Zeno reads the whole program before it runs any of it, and if it finds a mistake it says so and runs nothing. Here is a program with two:

```
let name = "Ada"
name = "Grace"
print(nmae)
```

```
error: `name` cannot be changed. Declare it with `var` to allow that
 --> main.zeno:2:1
  |
```

```
2 | name = "Grace"
  | ^

error: `nmae` is not defined
 --> main.zeno:3:7
  |
3 | print(nmae)
  |       ^

2 errors found. Nothing was run.
```

Each message says what is wrong, and where: the file, the line and the column, the line itself, and a ^ under the place. Where Zeno can tell what you meant, it says what to write instead. The first mistake here is that a name given with `let` cannot be given another value, and the message names the word that would allow it. The second is a misspelt name.

Habits from other languages get messages of their own:

```
def greet():
    print("hi")
```

```
error: Zeno spells `def` as `function`
 --> main.zeno:1:1
  |
1 | def greet():
  | ^
```

```
let ready = true
if ready && true {
    print("go")
}
```

```
error: Zeno spells this `and`
 --> main.zeno:2:10
  |
2 | if ready && true {
  |          ^
```

```
print("hello");
```

```
error: Zeno has no semicolons. A line break ends a statement
 --> main.zeno:1:15
  |
1 | print("hello");
  |               ^
```

> **Why.** An error message is the language teaching the person who reads it, at the moment the lesson is most useful. So every message in Zeno aims to say what to write instead, not only what is wrong.

Notice that the last three programs each showed one message, while the one before showed two. A mistake in the *shape* of a program, such as a missing brace, leaves Zeno unsure where it is, and what it would report after that could be nonsense caused by the first. So it stops at the first. A mistake in the *meaning*, such as a misspelt name, leaves it sure where it is, so it reports every one it finds.

## 2.6 Checking without running

`zeno run` checks a program and then runs it. `zeno check` only checks it:

greet.zeno

```
let name = "Ada"
print(f"Hello, {name}!")
```

```
$ zeno check greet.zeno
No errors found.
```

It prints nothing when there is nothing wrong. That makes it useful for a program that would take a long time to run, or one that drives hardware you do not want to move yet.

### What this chapter said

A program is a file of statements, done from the top, one to a line. `print` shows values, `#` starts a comment, and `let` names a value. Zeno checks the whole program before it runs a line of it, and a message says what is wrong, where, and usually what to write instead.

### Things to try

1. Print your own name and the year you were born on one line, with one `print`.
2. Leave the closing quote off a string and read the message. Then leave off a closing bracket.
3. Write `elif` where Zeno expects `else if`, and `x := 1` for a variable. What does Zeno say to each?
4. Put two mistakes of meaning in one program and run `zeno check`. How many messages are there? Now add a mistake of shape, such as `print("a"`. How many now, and why?

CHAPTER 3

# A Tour of Zeno

This chapter runs through the whole language quickly, one small program at a time, so that you know what is coming and where it is explained. Nothing here has to be understood yet. Each section names the chapter that goes through it properly.

## 3.1 Values that change, and values that do not

```
let name = "Zeno"          # let: set once, never changes
var count = 3              # var: can change later
count = count + 1          # plain = changes a var
const LIMIT = 100          # const: fixed when the program is built
print(name, count, LIMIT)
```

```
Zeno 4 100
```

A name is declared once, with `let`, `var` or `const`, and from then on plain `=` changes it. A misspelt name is therefore a mistake, and never a new variable. (Values and Variables.)

## 3.2 Lists, and a comprehension

```
let readings = [21.5, 22.0, -99.0, 21.0, 150.0]
let kept = [r for r in readings if -40.0 <= r <= 85.0]
print(kept, sum(kept) / len(kept))
```

```
[21.5, 22.0, 21.0] 21.5
```

Comparisons chain as they do in mathematics: `-40.0 <= r <= 85.0` means what it looks like. (Lists, Numbers and Operators.)

## 3.3 Functions, with types or without

```
function describe(t: float) -> str {
    if t < 18.0 {
        return "cold"
    } else if t < 24.0 {
        return "comfortable"
    }
    return "warm"
}

function average(values) {
    return sum(values) / len(values)
```

```
}

for t in [12.0, 21.5, 30.0] {
    print(t, describe(t))
}
print(average([1, 2, 3, 4]))
```

```
12.0 cold
21.5 comfortable
30.0 warm
2.5
```

Types are optional. Where they are written, they are checked before the program runs. Where they are left out, Zeno works them out from how the function is used. (Functions, Types, Written and Worked Out.)

## 3.4 Classes

```
import math

class Point {
    x: float
    y: float

    function distance_to(self, other: Point) -> float {
        return math.sqrt((self.x - other.x) ** 2 + (self.y - other.y) ** 2)
    }
}

let origin = Point(0.0, 0.0)
let here = Point(x=3.0, y=4.0)
print(origin, here, origin.distance_to(here))
```

```
Point(x=0.0, y=0.0) Point(x=3.0, y=4.0) 5.0
```

A class lists its fields, and gets a way to build one for free. It has no parent class: what classes share is said with an interface. (Classes, Interfaces.)

## 3.5 Enums that carry values, and match

```
enum Reading {
    Value(celsius: float)
    Fault(code: int)
    Missing
}

function show(r: Reading) -> str {
    match r {
        case Value(c) { return f"{c:.1f} degrees" }
        case Fault(code) { return f"fault {code}" }
        case Missing { return "no reading" }
    }
}
```

```
for r in [Value(21.5), Fault(7), Missing] {
    print(show(r))
}
```

```
21.5 degrees
fault 7
no reading
```

Leave a `case` out and Zeno names it before the program runs. (Enums and match.)

## 3.6 No surprise none

```
function find_user(id: int) -> str or none {
    if id == 1 {
        return "ada"
    }
    return none
}

let user = find_user(2)
print(user.upper())
```

```
error: this is str or none. Check it with `is not none` before using `.upper`
 --> main.zeno:9:7
  |
9 | print(user.upper())
  |       ^

1 error found. Nothing was run.
```

A value can be `none` only if its type says `or none`, and then Zeno will not let it be used until the program has asked:

```
function find_user(id: int) -> str or none {
    if id == 1 {
        return "ada"
    }
    return none
}

for id in [1, 2] {
    let user = find_user(id)
    if user is none {
        print(id, "nobody")
    } else {
        print(id, user.upper())
    }
}
```

```
1 ADA
2 nobody
```

(Values That May Be Missing.)

## 3.7 Errors that are checked

```
function parse_port(text: str) -> int raises ValueError {
    let port = int(text)
    if not 1 <= port <= 65535 {
        raise ValueError(f"{port} is not a port")
    }
    return port
}

for text in ["8080", "99999", "http"] {
    try {
        print("port", parse_port(text))
    } except ValueError as err {
        print("refused:", err)
    }
}
```

```
port 8080
refused: 99999 is not a port
refused: cannot turn "http" into an int
```

A function says what it `raises`, and Zeno checks that every error is either caught or passed on on purpose. (Errors.)

## 3.8 Functions are values

```
function counter() {
    var count = 0
    return function() {
        count += 1
        return count
    }
}

let next_id = counter()
print(next_id(), next_id(), next_id())
print([1, 2, 3].map(function(n: int) -> int { return n * 2 }))
```

```
1 2 3
[2, 4, 6]
```

(Functions as Values.)

## 3.9 Dicts

```
let ages = ["Ada": 36, "Alan": 41]
ages["Grace"] = 85
for name, age in ages {
    print(name, age)
}
```

```
Ada 36
```

```
Alan 41
Grace 85
```

(Dicts.)

## 3.10 Numbers with units, and numbers with a size

```
let delay = 1500 ms
print(delay, delay.s, 1 s + 250 ms)

let frame: u8 = 200
print(u8.wrap(frame + 100), frame & 0x0F, frame >> 4)
```

```
1500 ms 1.5 s 1250 ms
44 8 12
```

A number can carry a unit, and units that measure the same thing work together. A whole number can have a size, and a number that does not fit is an error unless you ask for it to wrap. (Units, Sized Numbers and Bits.)

## 3.11 Generics

```
function largest[T](items: list[T]) -> T {
    var best = items[0]
    for item in items {
        if item > best {
            best = item
        }
    }
    return best
}

print(largest([3, 9, 4]), largest(["pear", "fig"]), largest([3.3 V, 5 V]))
```

```
9 pear 5 V
```

Written once, checked for each type it is used with, and compiled as if it had been written by hand for that type. (Generics.)

## 3.12 And on a board

blink.zeno

```
from board import led
from time import sleep

while true {
    led.toggle()
    sleep(500 ms)
}
```

```
$ zeno check blink.zeno
```

```
No errors found.
```

On a computer the board is simulated, and `zeno run` prints each change of a pin. The same program builds for firmware with no garbage collector, no heap and no `setjmp`. (Pins, Time and the Board, Firmware Builds, Building with No Collector.)

### Things to try

1. Take one of the programs in this chapter, break it on purpose, and read the message.
2. In the enum program, add a fourth case `Calibrating` to `Reading` and run it before changing `show`.
3. In the error program, take `raises ValueError` off `parse_port` and run `zeno check --strict` on the file.

PART II

# The Language

CHAPTER 4

# Values and Variables

A program works on *values*: numbers, pieces of text, true and false, lists of things. This chapter is about the values Zeno has, and about the names you give them.

## 4.1 The kinds of value

Every value has a *type*, which says what kind of value it is and what can be done with it. These are the types every program meets first:

| Type | What it holds | Written like |
|---|---|---|
| `int` | a whole number | `42`, `-7`, `0xFF`, `1_000_000` |
| `float` | a number with a fraction | `3.14`, `-0.5`, `2.5e-7` |
| `bool` | true or false | `true`, `false` |
| `str` | text | `"Ada"` |
| `list[int]` | several values, in order | `[1, 2, 3]` |
| `dict[str, int]` | values filed under keys | `["Ada": 36]` |
| `none` | no value | `none` |

```
print(42, 3.14, true, "Ada", [1, 2, 3], ["Ada": 36], none)
```

```
42 3.14 true Ada [1, 2, 3] ["Ada": 36] none
```

Later chapters add more: numbers with a unit (`500 ms`), numbers with a size (`u8`), objects of your own classes, enum values, and functions.

## 4.2 let: a name for a value

`let` gives a value a name:

```
let name = "Ada"
let born = 1815
print(name, "was born in", born)
```

```
Ada was born in 1815
```

A name given with `let` keeps its value for as long as it exists. Trying to give it another is a mistake, and the message names the word that would allow it:

```
let count = 0
count = 1
```

```
error: `count` cannot be changed. Declare it with `var` to allow that
 --> main.zeno:2:1
  |
2 | count = 1
  | ^

1 error found. Nothing was run.
```

## 4.3 var: a name whose value changes

When the value is meant to change, declare the name with `var`. After that, plain = gives it a new value:

```
var count = 0
count = count + 1
count = count + 1
print(count)
```

```
2
```

There is a shorter way to write `count = count + 1`: `count += 1`. Every operator has a form like it, and Numbers and Operators lists them.

A name is declared once, and only declared names can be given a value. So a misspelt name is a mistake, and never a new variable by accident:

```
var count = 0
cuont = 1
```

```
error: `cuont` is not declared. Create it with `let` or `var` first
 --> main.zeno:2:1
  |
2 | cuont = 1
  | ^

1 error found. Nothing was run.
```

> **Why.** In Python, `x = 1` both makes a new variable and changes an old one, so a typing slip quietly makes a second variable and the first one never changes. Separating the two turns the slip into an error. `let` is the short, ordinary word because most values never change once they are made, and a program whose values do not change is easier to follow. (Decision 1.6.)

## 4.4 const: a value fixed for the whole program

```
const LIMIT = 100
const GREETING = "hello"
print(LIMIT, GREETING)
```

```
100 hello
```

A `const` says that the value belongs to the program, not to a moment while it runs: a size, a limit, an address, a name. By custom its name is written in capitals. Like a `let`, it can never be given another value.

Unlike a `let`, its value must be known before the program runs: numbers, text, other constants, the operators between them, and lists and dicts of those. Something only the running program can work out belongs in a `let`:

```
const NAME = input()
```

```
error: this is only known when the program runs, and a `const` is fixed when the program
    ↪ is built. A `const` is written from numbers, text, other `const`s and operators.
    ↪ For a value that is worked out once while the program runs, use `let`
 --> main.zeno:1:14
  |
1 | const NAME = input()
  |              ^

1 error found. Nothing was run.
```

## 4.5 Types are optional

You can write the type of a name after a `:`, or leave it out:

```
let port: int = 8080
let host = "localhost"          # a str: Zeno sees that
var ratio: float = 0.5
print(host, port, ratio)
```

```
localhost 8080 0.5
```

Where the type is left out, Zeno works it out from the value. Where it is written, the value must fit it, and from then on every value given to the name must fit it too:

```
var level: int = 3
level = "high"
```

```
error: the new value should be int, but this is str
 --> main.zeno:2:9
  |
2 | level = "high"
  |         ^

1 error found. Nothing was run.
```

A whole number is not quietly made a float, even where a float is wanted. Write the decimal point:

```
let ratio: float = 1
```

```
error: `ratio` should be float, but this is int. Write a number with a decimal point, as
    ↪ in `1.0`, and convert a value with `float( )`
 --> main.zeno:1:20
  |
1 | let ratio: float = 1
  |                    ^

1 error found. Nothing was run.
```

> **Why.** Turning an `int` into a `float` where one is stored would leave a value whose written type and real type disagree, and in a compiled program it hides an instruction. Arithmetic is different: `1 + 0.5` makes a new value, and nothing is disguised. (Decisions 2.3 and 2.3b.)

A name must have a value from the moment it is declared. There is no `var x` with nothing after it:

```
var total: int
```

```
error: expected `=`, found the end of the line
 --> main.zeno:1:15
  |
1 | var total: int
  |               ^
```

Types, Written and Worked Out goes into all of this in depth: what Zeno works out, what it cannot, and the strict mode in which every type must be known.

## 4.6 Names

A name is made of letters, digits and _, and does not start with a digit. Letters of any alphabet are allowed. Case counts: `total` and `Total` are two names. By custom, variables and functions are written `like_this`, classes and enums `LikeThis`, and constants `LIKE_THIS`.

The words of the language, such as `let`, `if` and `take`, can never be names, and the message says where the word goes instead:

```
let take = 3
```

```
error: expected a name for the variable, found `take`. `take` is a word of the language,
    ↪  so it cannot be a name. It goes in front of the type of a parameter: `frame:
    ↪ take Frame`
 --> main.zeno:1:5
  |
1 | let take = 3
  |     ^
```

The names of types (`int`, `str`, `list`) and of the built-in functions (`print`, `len`) are not reserved. They are names like any other, and a program may declare its own. That is rarely a good idea, since the built-in one is then out of reach in that block.

## 4.7 Blocks and scope

A pair of braces `{ }` makes a *block*, and a name declared in a block exists from its declaration to the end of that block. This stretch of the program is the name's *scope*:

```
let x = 1
if true {
    let x = 2          # a new x, for this block only
    let y = 3
    print(x, y)
}
print(x)               # the first x again
```

```
2 3
1
```

After the block, y is gone:

```
if true {
    let y = 3
}
print(y)
```

```
error: `y` is not defined
 --> main.zeno:4:7
  |
4 | print(y)
  |       ^

1 error found. Nothing was run.
```

An inner block may declare a name that an outer block already has, and inside the inner block the name means the inner one. Two declarations of one name in the same block are a mistake:

```
let speed = 10
let speed = 20
```

```
error: `speed` is already declared in this block
 --> main.zeno:2:1
  |
2 | let speed = 20
  | ^

1 error found. Nothing was run.
```

> **Why.** A variable that lives exactly as long as its block has a lifetime you can see in the layout of the code. In Python a variable made inside an `if` goes on existing after it. (Decisions 3.1 and 3.2.)

## 4.8 Two names for one list

A name gives a value a name; it does not make a copy. When the value is a list or an object, two names can stand for the same one, and a change made through either is seen through both:

```
let a = [1, 2]
let b = a
b.append(3)
print(a)
```

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

This also means that `let` fixes the *name*, not what is inside the value. `a` above can never be given another list, but the list it names can still grow. Numbers, text and true or false cannot be changed inside, so for them the difference never shows. Lists has more on this.

### What this chapter said

Every value has a type. `let` names a value that stays, `var` one that changes, and `const` one fixed for the whole program. Plain = only changes a name that already exists. Types can be

written after `:` or left for Zeno to work out. A name lives until the end of the block it was declared in.

### Things to try

1. Declare `var price = 10` and then give it `10.5`. What does Zeno say, and how would you declare it so that both are allowed?
2. Write `let x: str = 5` and `let y: int = "5"`. Read both messages.
3. Use `print` as a variable name with `let print = 3`, and then try to print something below it. What happened?
4. Make two names for one list, change it through one, and print it through the other. Then do the same with two numbers and `var`. Why is the result different?

CHAPTER 5

# Numbers and Operators

Zeno has two kinds of plain number. An `int` is a whole number, of 64 bits, from $-2^{63}$ to $2^{63} - 1$: a little over nine billion billion either way. A `float` is a number with a fraction, the 64-bit floating point number that every processor with a floating point unit has. Chapters later in the book add numbers with a unit and whole numbers with a smaller size. This chapter is the plain two, and the operators that work on them.

## 5.1 Writing numbers

```
print(42, -7, 1_000_000)          # _ can split the digits, anywhere
print(0xFF, 0b1010_0101)          # hexadecimal and binary
print(3.14, -0.5, 2.5e-7, 1e10)   # floats
```

```
42 -7 1000000
255 165
3.14 -0.5 2.5e-7 10000000000.0
```

A float always prints with a decimal point or an exponent, so that it can never be taken for an int: `3.0` prints as `3.0`, not `3`.

## 5.2 Arithmetic

| Operator | Means | Example |
|---|---|---|
| `+` | add | `7 + 2` is 9 |
| `-` | subtract, or negate | `7 - 2` is 5 |
| `*` | multiply | `7 * 2` is 14 |
| `/` | divide, always giving a float | `7 / 2` is `3.5` |
| `//` | divide and round down | `7 // 2` is 3 |
| `%` | the remainder of `//` | `7 % 2` is 1 |
| `**` | power | `7 ** 2` is 49 |

```
print(7 + 2, 7 - 2, 7 * 2, 7 / 2, 7 // 2, 7 % 2, 7 ** 2)
print(10 / 5, 2 ** 0.5)
```

```
9 5 14 3.5 3 1 49
2.0 1.4142135623730951
```

### 5.2.1 Division

`/` always gives a float, even when the numbers divide exactly. To divide whole numbers and keep a whole number, say so with `//`.

> **Why.** In C, `7 / 2` is `3` and `7.0 / 2` is `3.5`: one operator does two different things depending on what it is given, and the remainder vanishes without a word. In Zeno the operator decides what happens, not the values. (Decision 2.1.)

`//` rounds *down*, toward minus infinity, and `%` takes the sign of the number on its right. Together they always satisfy `(a // b) * b + a % b == a`:

```
print(-7 // 2, -7 % 2, 7 % -2)
print(5.5 // 2, 5.5 % 2)
```

```
-4 1 -1
2.0 1.5
```

That is what makes `i % n` a safe position in a list of length `n` for any `i`, negative or not.

Dividing by zero is an error, with every division operator and with both kinds of number:

```
print(1 / 0)
```

```
error: uncaught ZeroDivisionError: division by zero
 --> main.zeno:1:7
  |
1 | print(1 / 0)
  |       ^
```

### 5.2.2 Powers

`2 ** 10` is an int. A whole number to a negative power would be a fraction, so it is refused, and the message says what to write:

```
print(2 ** -1)
```

```
error: uncaught ValueError: a whole number cannot be raised to a negative power. Use a
    ↪ float, as in `2.0 ** -1`
 --> main.zeno:1:7
  |
1 | print(2 ** -1)
  |       ^
```

> **Why.** If `2 ** -1` were `0.5`, the type of `a ** b` would depend on the *value* of `b`, which nothing can know before the program runs. The operator decides the type, not the values. (Decision 2.3a.)

### 5.2.3 Overflow

An `int` that would go past its largest or smallest value is an `OverflowError`, never a wrong number:

```
let big = 9223372036854775807
print(big + 1)
```

```
error: uncaught OverflowError: the result is too big for an int
 --> main.zeno:2:7
  |
2 | print(big + 1)
  |       ^
```

**Why.** In C, an overflow either wraps round without a word or is undefined. A wrong number that looks plausible is the worst kind of mistake. When a program really does want a number to wrap round, as a checksum does, it says so with `wrap`, which Sized Numbers and Bits explains. (Decision 2.2.)

A float that grows too large becomes `inf`, as floats do everywhere:

```
print(1e308 * 10, -1e308 * 10)
```

```
inf -inf
```

## 5.3 Mixing ints and floats

An int and a float can meet in arithmetic, and the answer is a float:

```
print(1 + 0.5, 3 * 1.5, 1 == 1.0)
```

```
1.5 4.5 true
```

Nothing else changes kind by itself. Text is never turned into a number, or a number into text, without being asked:

```
print("3" + 4)
```

```
error: `+` does not work between str and int
 --> main.zeno:1:7
  |
1 | print("3" + 4)
  |       ^

1 error found. Nothing was run.
```

**Why.** Making an int a float in arithmetic is what people expect from arithmetic. Every other automatic change is a guess at what the programmer meant, and JavaScript shows where those guesses lead: `"3" + 4` is `"34"` there. (Decision 2.3.)

## 5.4 Changing kind on purpose

| Written | Gives |
|---|---|
| `int(x)` | a whole number: from text, or from a float by dropping the fraction |
| `float(x)` | a float: from text or from an int |
| `str(x)` | the value as text, as `print` shows it |

```
print(int("42"), int(" 7 "), int(9.99), int(-9.99))
print(float("1e3"), float(3), str(2.50) + "!")
```

```
42 7 9 -9
1000.0 3.0 2.5!
```

Text that is not a number is a `ValueError`, which a program can catch (Errors):

```
print(int("forty"))
```

```
error: uncaught ValueError: cannot turn "forty" into an int
 --> main.zeno:1:7
  |
1 | print(int("forty"))
  |       ^
```

## 5.5 Comparing

| Operator | Means |
|---|---|
| ==, != | equal, not equal |
| <, <= | less than, less than or equal |
| >, >= | greater than, greater than or equal |

A comparison gives a `bool`. Comparisons *chain*, as they do in mathematics: `0 <= i < n` means `0 <= i and i < n`, with `i` worked out once.

```
let i = 3
print(0 <= i < 10, 1 < 2 < 3, 3 > 2 > 2)
```

```
true true false
```

> **Why.** In C, `0 <= i < n` means `(0 <= i) < n`, which compares true or false with a number, compiles without a word, and is a classic mistake. Refusing chains altogether, as Rust does, is safe but loses the readable form. Zeno gives the line the meaning it looks like it has. (Decision 1.16.)

Comparing two things that can never be equal is a mistake, and Zeno says so instead of quietly answering `false`. It is nearly always a number that was read as text and never turned into one:

```
let typed = "1"
print(typed == 1)
```

```
error: `==` between str and int is always false
 --> main.zeno:2:7
  |
2 | print(typed == 1)
  |       ^

1 error found. Nothing was run.
```

Floats are kept in binary, so most decimal fractions are close to what you wrote but not exactly it. That is true in every language, and it is why floats are compared with care:

```
print(0.1 + 0.2, 0.1 + 0.2 == 0.3)
```

```
print(abs((0.1 + 0.2) - 0.3) < 1e-9)
```

```
0.30000000000000004 false
true
```

## 5.6 True and false

A `bool` is `true` or `false`, and three words work on them: `and`, `or` and `not`.

```
let warm = true
let raining = false
print(warm and not raining, warm or raining, not warm)
```

```
true true false
```

`and` and `or` stop as soon as the answer is known: in `a and b`, `b` is not worked out at all when `a` is false. That makes `len(xs) > 0 and xs[0] == 1` safe for an empty list.

A condition must be a real `bool`. A number, a list or a piece of text is not true or false by itself:

```
let items = [1, 2]
if items {
    print("not empty")
}
```

```
error: a condition must be true or false, but this is list[int]
 --> main.zeno:2:4
  |
2 | if items {
  |    ^

1 error found. Nothing was run.
```

**Why.** "Truthiness" means remembering which values count as false: is it `0`, `""`, `[]`, `none`, `0.0`? The answer differs between languages and causes mistakes such as treating a real reading of `0` as no reading at all. Saying what you test, `len(items) > 0`, is clearer. (Decision 2.4.)

## 5.7 Changing a variable: the = forms

Every operator that takes two numbers has a form that changes a variable, `x op= y`, which means exactly `x = x op y`:

`+=` `-=` `*=` `/=` `//=` `%=` `**=` `&=` `|=` `^=` `<<=` `>>=`

```
var n = 10
n -= 3        # 7
n *= 2        # 14
n //= 3       # 4
n **= 2       # 16
print(n)
var f = 1.0
f /= 4
print(f)
```

```
16
0.25
```

The variable is read first and the right-hand side worked out second, as in `x = x + y`. `/=` on an int is refused, because `/` gives a float and an int variable cannot hold one: use `//=`.

```
var half = 9
half /= 2
```

```
error: the new value should be int, but this is float
 --> main.zeno:2:9
  |
2 | half /= 2
  |         ^

1 error found. Nothing was run.
```

## 5.8 Which operator goes first

When operators are mixed, some bind tighter than others, as in school arithmetic: `2 + 3 * 4` is `14`. From the loosest to the tightest:

| Operators | |
|---|---|
| `or` | loosest |
| `and` | |
| `not` | |
| `==  !=  <  <=  >  >=  in  is` | comparisons, which chain |
| `\|` | bit or |
| `^` | bit exclusive or |
| `&` | bit and |
| `<<  >>` | shifts |
| `+  -` | |
| `*  /  //  %` | |
| `-x  ~x` | negation, bit not |
| `**` | tightest |

```
print(2 + 3 * 4 ** 2, (2 + 3) * 4, -2 ** 2)
```

```
50 20 -4
```

Brackets always win, and they are the best way to make a long expression plain to the reader. The bit operators are in Sized Numbers and Bits, and the whole table again, with every operator, is Operators.

## 5.9 Built-in functions for numbers

```
print(abs(-4.5), min(3, 1, 2), max(2, 8.5), sum([1, 2, 3]))
print(min([5, 3, 9]), max([5, 3, 9]))
```

```
4.5 1 8.5 6
3 9
```

`min` and `max` take either several numbers or one list. Square roots, rounding, logarithms and the trigonometric functions are in the `math` module (The math Module).

### What this chapter said

`int` is a 64-bit whole number, `float` a 64-bit floating point number. `/` always gives a float and `//` rounds down. An int that overflows is an error, never a wrong number. Ints and floats mix in arithmetic and nowhere else. Comparisons chain, conditions must be real `bool`s, and every operator on two numbers has an = form.

### Things to try

1. Work out `-17 // 5` and `-17 % 5` on paper, then check with Zeno. Check that `(a // b) * b + a % b == a` holds.
2. What is the largest power of two that fits in an `int`? Find it by trying `2 ** n` for larger and larger `n`.
3. Write `if 1 <= x <= 10 and x % 2 == 0` for a few values of `x`. Put brackets where Zeno would put them.
4. Print `1 / 3 * 3`. Is it exactly 1? Now try `0.1 * 3`.

CHAPTER 6

# Text

Text in Zeno is a `str`: a row of characters. It is written between double quotes, or between single quotes when it holds double ones:

```
let name = "Ada"
let quote = 'She said "hello"'
print(name, quote, "it's fine")
```

```
Ada She said "hello" it's fine
```

A string must close on the line it opens on.

## 6.1 Characters that cannot be typed

A backslash starts an *escape*: two characters that stand for one.

| Escape | Means |
|---|---|
| `\n` | a line end |
| `\t` | a tab |
| `\r` | a carriage return |
| `\"`, `\'` | a quote |
| `\\` | a backslash |

```
print("first line\nsecond line")
print("a backslash: \\, and a quote: \"")
```

```
first line
second line
a backslash: \, and a quote: "
```

Any other letter after a backslash is a mistake, so a slip such as `"C:\temp\new"` is caught rather than quietly turned into a tab and a line end.

## 6.2 Characters, not bytes

Zeno text is Unicode, kept in UTF-8. Its length, its positions and a loop over it all count *characters*, whatever alphabet they are from and however many bytes each one takes:

```
let word = "naïve"
print(len(word), word[2], word.find("v"))
```

```
5 ï 3
```

A position counts from 0, so `word[0]` is the first character. A position past the end, or below 0, is an `IndexError`:

```
print("abc"[3])
```

```
error: uncaught IndexError: index 3 is out of range for a str of length 3
 --> main.zeno:1:7
  |
1 | print("abc"[3])
  |       ^
```

**Why.** In Python, `text[-1]` is the last character. That is handy, but it means a position that goes below zero *by mistake* quietly reads from the other end instead of failing. A clear error is the safer thing, above all in code that runs on hardware. (Decision 2.6.)

Text cannot be changed inside. Every method that seems to change it gives back new text and leaves the old as it was:

```
let word = "cat"
word[0] = "b"
```

```
error: an item cannot be set on str
 --> main.zeno:2:1
  |
2 | word[0] = "b"
  | ^

1 error found. Nothing was run.
```

## 6.3 Joining and comparing

`+` joins two pieces of text. Both sides must be text; a number is made into text with `str`, or put into a formatted string:

```
let count = 3
print("count: " + str(count))
print("ab" == "ab", "ab" < "b", "B" < "a")
print("ell" in "hello", "x" not in "hello")
```

```
count: 3
true true true
true true
```

Text is ordered character by character, by the number Unicode gives each character, so every capital comes before every small letter. `in` asks whether one piece of text is found in another.

## 6.4 Formatted strings

An `f` in front of a string lets it hold expressions in braces. Each one is worked out and its value put into the text, as `print` would show it:

```
let name = "Ada"
```

```
let items = 3
print(f"{name} has {items} items, {items * 2} halves")
print(f"Shout it: {name.upper()}")
print(f"Braces themselves: {{ and }}")
```

```
Ada has 3 items, 6 halves
Shout it: ADA
Braces themselves: { and }
```

Inside the braces, a string must use the other kind of quote from the one around it.

> **Why.** Without the `f`, any string with a `{` in it would change meaning, and such strings are common: JSON, templates, messages for a device. The one letter marks the strings that hold code. (Decision 1.14.)

### 6.4.1 Formats

After the expression, a `:` and a *format* say how to lay the value out:

| Format | Means | `f"[{v:...}]"` |
|---|---|---|
| `.2f` | two digits after the point | `[3.14]` for `3.14159` |
| `8.3f` | at least 8 wide, 3 digits after the point | `[   3.142]` |
| `6` | at least 6 wide; numbers on the right, text on the left | `[    42]` |
| `<6`, `>6`, `^6` | at least 6 wide: left, right, centred | `[42    ]` |
| `03` | at least 3 wide, with zeros in front | `[042]` |
| `x`, `X` | a whole number in hexadecimal | `[ff]`, `[FF]` |
| `b` | a whole number in binary | `[101]` |
| `d` | a whole number | `[42]` |

```
let pi = 3.14159
let n = 42
print(f"[{pi:.2f}] [{pi:8.3f}] [{n:6}] [{n:<6}] [{n:^6}] [{n:04}]")
print(f"[{255:x}] [{255:X}] [{5:b}] [{0xA5:08b}]")
print(f"[{'ab':>5}] [{'ab':<5}]")
```

```
[3.14] [   3.142] [    42] [42    ] [  42  ] [0042]
[ff] [FF] [101] [10100101]
[   ab] [ab   ]
```

A format that makes no sense for the value is refused before the program runs:

```
let pi = 3.14159
print(f"{pi:x}")
```

```
error: `x` needs an int, but this is a float
 --> main.zeno:2:7
  |
2 | print(f"{pi:x}")
  |       ^
```

## 6.5 The methods of text

A *method* is a function that belongs to a value, and is called with a dot after it: `name.upper()`. Text has these:

| Method | Gives |
|---|---|
| `upper()`, `lower()` | the text in capitals, or in small letters |
| `strip()` | the text without blanks at either end |
| `split()` | a list of the words, split at blanks |
| `split(sep)` | a list of the pieces between each `sep` |
| `lines()` | a list of the lines, without their line ends |
| `replace(old, new)` | the text with every `old` made `new` |
| `starts_with(part)`, `ends_with(part)` | whether it starts or ends with `part` |
| `find(part)` | the position where `part` starts, or `none` |
| `slice(from, to)` | the characters from `from` up to, not including, `to` |
| `slice(from)` | the characters from `from` to the end |
| `code()` | the Unicode number of a text of one character |

```
let line = "  Set Speed 40  "
let words = line.strip().lower().split()
print(words)
print("a,b,,c".split(","), "x=1".replace("=", " is "))
print("hello".starts_with("he"), "hello".ends_with("x"))
print("hello".slice(1, 4), "hello".slice(2))
print("A".code(), char(66), char(8364))
```

```
["set", "speed", "40"]
["a", "b", "", "c"] x is 1
true false
ell llo
65 B €
```

`char(n)` is the other way round from `code()`: the character with that number. It is a function and not a method because there is no text yet to ask: it *makes* text from a number, as `str(n)` does.

### 6.5.1 split and lines

`split()` with nothing in the brackets splits at any run of blanks and never gives an empty piece. `split(sep)` splits at each `sep` exactly, so two in a row give an empty piece between them:

```
print("one  two   three".split())
print("  a b ".split(" "))
print("".split(","), "a,".split(","))
```

```
["one", "two", "three"]
["", "", "a", "b", ""]
[""] ["a", ""]
```

`lines()` is the way to cut text into lines. It knows both kinds of line end, `"\n"` and `"\r\n"`, and a last line with no end:

```
print("a\nb\n".split("\n"))
print("a\nb\n".lines())
print("one\r\ntwo".lines())
```

```
["a", "b", ""]
["a", "b"]
["one", "two"]
```

### 6.5.2 find gives a position or none

`find` gives the position of a piece of text, or `none` when it is not there. Because the answer may be `none`, Zeno will not let it be used as a number until the program has asked:

```
function first_word(line: str) -> str {
    let space = line.find(" ")
    if space is none {
        return line
    }
    return line.slice(0, space)
}

print(first_word("set speed 40"), first_word("stop"))
```

```
set stop
```

> **Why.** Python's `find` gives `-1` when the text is not there, and `-1` is a number, so nothing stops it from being used as a position. `none` cannot be used as a position at all, so the mistake is caught before the program runs. (Decision 6.3.)

Values That May Be Missing is about `none` and how Zeno knows when it has been ruled out.

### 6.5.3 slice takes a part

`slice(from, to)` counts as `range` does: from `from` up to but not including `to`. A position outside the text is an error, not a shorter slice:

```
print("abc".slice(1, 9))
```

```
error: uncaught IndexError: slice 1 to 9 is out of range for a str of length 3
 --> main.zeno:1:7
  |
1 | print("abc".slice(1, 9))
  |       ^
```

If you write Python's `text[1:3]`, Zeno says what its own way is:

```
let text = "hello"
print(text[1:3])
```

```
error: a part of a list or a str is taken with `.slice(from, to)`, as in `xs.slice(1, 3)
    ↪ `
 --> main.zeno:2:13
  |
2 | print(text[1:3])
  |             ^
```

### 6.5.4 Plain letters only

`upper` and `lower` change the letters `a` to `z` and `A` to `Z` and nothing else, and `strip` and `split()` know the space, the tab, the line ends and the form feed as blanks. Letters of other alphabets pass through untouched:

```
print("Straße".upper(), "ÉCOLE".lower())
```

```
STRAßE école
```

> **Why.** The full rules for every alphabet need tables of tens of kilobytes, and a compiled Zeno program has to fit on a microcontroller. A method must also give the same answer under `zeno run` as in a compiled program, so the rule is the same everywhere. (Decision 3.14.)

## 6.6 A loop over text

A `for` loop over text is given each character in turn. With two names it is given each position and the character there:

```
for ch in "abc" {
    print(ch)
}
for at, ch in "hé" {
    print(at, ch)
}
```

```
a
b
c
0 h
1 é
```

Deciding and Repeating has loops in full.

## 6.7 Text from other values

`str(x)` gives any value as text, exactly as `print` shows it. Inside a list, text is shown in quotes, so that `["1"]` and `[1]` look different:

```
print(str(42) + "!", str([1, "a"]), str(none), str(true))
```

```
42! [1, "a"] none true
```

### What this chapter said

A `str` is text, counted in characters. It is written in quotes, with escapes after a backslash, and never changes inside. `+` joins text, `in` finds it, and an `f` string puts values into it, laid out by a format. Its methods give new text, a list of pieces, a position or `none`.

### Things to try

1. Write `function last_word(line: str) -> str`. There is no method that searches from the end. What is the shortest way with what there is?
2. Print a table of the numbers 1 to 5 with their squares and cubes, each column 6 wide, using formats.
3. Print `"€".code()` and work out by hand, from the number, how many bytes the character takes in UTF-8.
4. What does `"a,b,c".split(",")` give? And `",".split(",")`? Say why each piece is there.
5. Write `"C:\temp"` in a program and read the message.

CHAPTER 7

# Deciding and Repeating

A program that only goes from the top to the bottom can only do one thing. This chapter is about the two things that change that: doing a block only when something is true, and doing a block again and again.

## 7.1 if

```
let temperature = 31
if temperature > 30 {
    print("hot")
}
```

```
hot
```

The block runs only when the condition is `true`. There is no `:` after the condition and no need for brackets around it: the `{` already says where the condition ends.

`else` gives a block for when it is false, and `else if` asks again:

```
function describe(t: int) -> str {
    if t < 0 {
        return "freezing"
    } else if t < 18 {
        return "cold"
    } else if t < 25 {
        return "comfortable"
    } else {
        return "hot"
    }
}

for t in [-5, 10, 21, 31] {
    print(t, describe(t))
}
```

```
-5 freezing
10 cold
21 comfortable
31 hot
```

There is one way to write it, so `elif` gets a message:

```
let x = 2
if x == 1 {
    print("one")
} elif x == 2 {
    print("two")
```

```
}
```

```
error: Zeno spells `elif` as `else if`
 --> main.zeno:4:3
  |
4 | } elif x == 2 {
  |   ^
```

> **Why.** With braces, `} else if x {` reads as plain words and needs no keyword of its own. (Decision 1.8.) The braces themselves are there because where a block ends should be written down, not implied by spaces that can be lost when code is pasted into an e-mail. (Decision 1.1.)

The condition must be a `bool`. `if count` for a number, or `if items` for a list, is refused; write what you mean, `count != 0` or `len(items) > 0` (Numbers and Operators).

## 7.2 while

`while` runs its block again and again for as long as its condition is true. The condition is asked before each turn:

```
var n = 1
while n < 100 {
    n *= 3
}
print(n)
```

```
243
```

A loop that should run until something inside it says stop is written `while true`, with a `break`:

```
var tries = 0
while true {
    tries += 1
    if tries * tries > 50 {
        break
    }
}
print(tries)
```

```
8
```

## 7.3 for

`for` goes through the items of something, one at a time:

```
for name in ["Ada", "Grace", "Alan"] {
    print("hello,", name)
}
```

```
hello, Ada
hello, Grace
hello, Alan
```

The name after `for` is new for each turn, belongs to the loop, and is gone after it. It cannot be given another value inside the loop.

A `for` loop can go through a list, the characters of a text, the keys of a dict, or a range of numbers.

### 7.3.1 range

`range` gives a list of whole numbers. With one number it counts from 0 up to, not including, that number. With two it counts from the first up to the second. A third number is the step, which may be negative:

```
print(range(5), range(2, 5), range(10, 0, -3), range(0))
for i in range(3) {
    print("turn", i)
}
```

```
[0, 1, 2, 3, 4] [2, 3, 4] [10, 7, 4, 1] []
turn 0
turn 1
turn 2
```

### 7.3.2 The position and the item

With two names, a `for` loop is given the position of each item and the item itself. The rule is the same for everything a loop can go through: *the second name is what is at the first*.

```
for at, word in ["zero", "one", "two"] {
    print(at, word)
}
for trip, n in range(10, 0, -4) {
    print(trip, n)
}
```

```
0 zero
1 one
2 two
0 10
1 6
2 2
```

Over a dict, the first name is the key and the second the value under it (Dicts).

> **Why.** Without this, the position is a `var` counted by hand beside the loop, or `for i in range(len(words))` and `words[i]` on the next line: two ways, both longer than what they mean. Python's `enumerate` and `items` are two more names to learn, and each builds pairs that Zeno has no type for. (Decision 10.12.)

### 7.3.3 The loop sees the list as it was

A `for` loop goes through the list as it was when the loop began. Adding to the list, or taking from it, inside the loop does not change which items the loop visits:

```
var xs = [1, 2, 3]
for x in xs {
    xs.append(x * 10)
```

```
}
print(xs)
```

```
[1, 2, 3, 10, 20, 30]
```

> **Why.** The other choice, looping over the list as it changes, is the source of two classic mistakes: a loop that never ends because its body keeps appending, and a loop that skips an item because its body removed one. (Decision 3.10a.)

## 7.4 break and continue

`break` leaves the loop at once. `continue` skips the rest of this turn and goes on with the next. Both work in `while` and `for`, and both act on the innermost loop they are in:

```
for n in range(1, 20) {
    if n % 2 == 0 {
        continue        # skip the even ones
    }
    if n > 9 {
        break           # and stop after 9
    }
    print(n)
}
```

```
1
3
5
7
9
```

A `break` outside a loop is a mistake:

```
break
```

```
error: `break` outside a loop
 --> main.zeno:1:1
  |
1 | break
  | ^

1 error found. Nothing was run.
```

## 7.5 Loops inside loops

```
for row in range(1, 4) {
    var line = ""
    for column in range(1, 4) {
        line = line + f"{row * column:4}"
    }
    print(line)
}
```

```
   1   2   3
```

```
   2   4   6
   3   6   9
```

## 7.6 A worked example: primes

Putting the pieces together, this program finds the prime numbers below 50. A number is prime when nothing from 2 up to its square root divides it:

primes.zeno

```
function is_prime(n: int) -> bool {
    if n < 2 {
        return false
    }
    var d = 2
    while d * d <= n {
        if n % d == 0 {
            return false
        }
        d += 1
    }
    return true
}

var found: list[int] = []
for n in range(50) {
    if is_prime(n) {
        found.append(n)
    }
}
print(found)
print(len(found), "primes below 50")
```

```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
15 primes below 50
```

### What this chapter said

`if`, `else if` and `else` choose a block. `while` repeats a block while a condition holds, and `for` goes through the items of a list, a text, a dict or a `range`, with an optional second name for the position or the key. A `for` loop sees its list as it was when it began. `break` leaves a loop and `continue` starts its next turn.

### Things to try

1. Print the numbers from 1 to 30, with `fizz` for those that 3 divides, `buzz` for those that 5 divides, and `fizzbuzz` for those both divide.
2. Change the primes program to stop as soon as it has found ten primes, using `break`.
3. Write a loop that appends to the list it walks through, and check that it ends. Then write the same with `while` and an index, and see that it does not.
4. Print a triangle of `*` five lines high, with a loop inside a loop.

CHAPTER 8

# Functions

A *function* is a block of code with a name, which the rest of the program can run by calling it. Functions are how a program is cut into pieces small enough to understand one at a time.

## 8.1 Declaring and calling

```
function area(width: float, height: float) -> float {
    return width * height
}

print(area(2.0, 3.0))
print(area(1.5, 4.0))
```

```
6.0
6.0
```

The first line is the function's *heading*: the word `function`, a name, the *parameters* in brackets with their types, and after `->` the type of what it gives back. Calling it gives each parameter a value, the *arguments*, and `return` hands a value back to the caller and ends the call.

> **Why.** `function` is a whole word that someone who has never seen the language can read aloud. `fn` and `def` have to be learnt, and editor completion makes the extra letters free. (Decision 1.4.)

A function that gives nothing back leaves out the `->` part. Its call is then worth `none`:

```
function announce(what: str) {
    print("now:", what)
}

let result = announce("lunch")
print(result)
```

```
now: lunch
none
```

## 8.2 Types are optional here too

The types in a heading can be left out:

```
function add(a, b) {
    return a + b
}
print(add(1, 2))
```

```
3
```

Zeno works out what it can. Here every call passes two ints, so `a` and `b` are ints, and the function gives an int. If a call disagrees with how the function uses its parameters, the mistake is reported inside the function, with a note that names the line the type was worked out from:

```
function shout(text) {
    return text.upper()
}
print(shout(5))
```

```
error: int has no field or method `upper`
 --> main.zeno:2:12
  |
2 |     return text.upper()
  |            ^
  = note: `text` was worked out to be int from line 4. Give `text` a type to have each
    ↪ call checked against it instead

1 error found. Nothing was run.
```

When the types are written, every call is checked against them, and a wrong call is reported at the call:

```
function shout(text: str) -> str {
    return text.upper()
}
print(shout(5))
```

```
error: `text` of `shout` should be str, but this is int
 --> main.zeno:4:13
  |
4 | print(shout(5))
  |             ^

1 error found. Nothing was run.
```

That is the reason to write types on anything that other people call. Types, Written and Worked Out says exactly what Zeno works out, and when it cannot.

## 8.3 Returning on every path

A function that promises to return a value must return one on every way through it. Zeno finds the way that falls off the end:

```
function sign(n: int) -> int {
    if n > 0 {
        return 1
    } else if n < 0 {
        return -1
    }
}
```

```
error: `sign` can reach its end without returning the int it promises
 --> main.zeno:1:1
  |
```

```
1 | function sign(n: int) -> int {
  | ^

1 error found. Nothing was run.
```

**Why.** In C, falling off the end of a function that promised a value returns whatever was lying in a register. In Python it returns `None`, which then fails somewhere else. Here it is an error at the function. (Decision 3.13.)

## 8.4 Default values

A parameter can have a default, used when the call leaves it out:

```
function greet(who: str, excited: bool = false) -> str {
    if excited {
        return f"Hello, {who}!"
    }
    return f"Hello, {who}."
}

print(greet("Ada"))
print(greet("Ada", true))
```

```
Hello, Ada.
Hello, Ada!
```

A default is worked out afresh on every call, so a default list is a new empty list each time:

```
function with_one(items: list[int] = []) -> list[int] {
    items.append(1)
    return items
}
print(with_one(), with_one())
```

```
[1] [1]
```

**Why.** In Python, `def add(item, to=[])` makes the list once, and every call shares it, so the second call finds the first call's item. It is one of the most reported surprises in that language. (Decision 3.5.)

## 8.5 Named arguments

Any argument can be given by the name of its parameter, which makes a call readable without looking the function up:

```
function set_timer(millis: int, repeat: bool, start_now: bool) {
    print(millis, repeat, start_now)
}

set_timer(500, true, false)
set_timer(millis=500, repeat=true, start_now=false)
set_timer(500, repeat=true, start_now=false)
```

```
500 true false
500 true false
500 true false
```

Plain arguments come first and named ones after them. A name that is not a parameter, or a parameter given twice, is a mistake:

```
function move(x: int, y: int) {
    print(x, y)
}
move(1, z=2)
```

```
error: `move` needs a value for `y`
 --> main.zeno:4:1
  |
4 | move(1, z=2)
  | ^

error: `move` has no argument named `z`
 --> main.zeno:4:11
  |
4 | move(1, z=2)
  |           ^

2 errors found. Nothing was run.
```

## 8.6 Parameters do not change

A parameter holds what the caller passed, all the way through the function. It cannot be given another value; if the function needs a changing value, it makes a `var` of its own:

```
function count_down(start: int) {
    while start > 0 {
        start -= 1
    }
}
```

```
error: `start` cannot be changed. Declare it with `var` to allow that
 --> main.zeno:3:9
  |
3 |         start -= 1
  |         ^

1 error found. Nothing was run.
```

```
function count_down(start: int) {
    var n = start
    while n > 0 {
        print(n)
        n -= 1
    }
}
count_down(3)
```

```
3
2
```

```
1
```

A list or an object passed as an argument is the caller's own, not a copy, so a function *can* change what is inside it, and the caller sees the change. Lists returns to this.

## 8.7 Calling a function above where it is written

A function can be used anywhere in its file, above or below the place it is declared. That lets a file read from the top, with the main logic first and the helpers below:

```
print(fahrenheit(100.0))

function fahrenheit(celsius: float) -> float {
    return celsius * 1.8 + 32.0
}
```

```
212.0
```

The same goes for classes and enums. Variables are different: a `let` or `var` exists only from its line on.

## 8.8 Functions that call themselves

A function may call itself. This is called *recursion*, and it suits problems that contain smaller copies of themselves:

```
function factorial(n: int) -> int {
    if n <= 1 {
        return 1
    }
    return n * factorial(n - 1)
}

function fib(n: int) -> int {
    if n < 2 {
        return n
    }
    return fib(n - 1) + fib(n - 2)
}

print(factorial(20), fib(20))
```

```
2432902008176640000 6765
```

## 8.9 Functions inside functions

A function can be declared inside another. It is then a name of that block, like a variable, and can use the variables that were declared above it in the blocks around it:

```
function report(readings: list[float]) {
    let limit = 25.0

    function flag(r: float) -> str {
```

```
        if r > limit {
            return "high"
        }
        return "ok"
    }

    for r in readings {
        print(r, flag(r))
    }
}

report([21.0, 27.5])
```

```
21.0 ok
27.5 high
```

What such a function can see is fixed by where it is written: the variables above it, never one declared further down. A function used before a variable it needs has been declared is refused, with the reason:

```
function outer() -> int {
    let result = inner()
    let base = 5
    function inner() -> int {
        return base * 2
    }
    return result
}
print(outer())
```

```
error: `base` is declared further down than the first use of `inner`, which needs it.
    ↪ Use `inner` below the line that declares `base`
 --> main.zeno:5:16
  |
5 |         return base * 2
  |                ^

1 error found. Nothing was run.
```

Functions are also values, which can be stored, passed and returned, and a function written inside another can go on using its variables after the outer one has returned. That is the subject of Functions as Values.

### What this chapter said

A function has a heading, `function name(parameters) -> type`, and a body. Types can be written or left out. A function that promises a value must return one on every path, and one that promises nothing returns `none`. Defaults are worked out on every call, arguments can be named, parameters never change, and functions can be called from above where they are written, call themselves, and live inside other functions.

### Things to try

1. Write `function clamp(x: float, low: float, high: float) -> float` and call it with named arguments in a different order from the heading.

2. Remove the types from `clamp` and call it with ints. Then add a call with a str. Where is the mistake reported?
3. Write `power(base: int, exponent: int) -> int` that calls itself, using `base ** exponent == base * base ** (exponent - 1)`.
4. Write a function with a `while true` loop and a `return` inside it. Does Zeno ask for a `return` after the loop?

CHAPTER 9

# Lists

A *list* holds several values in order. It can grow and shrink, and each of its items has a position, counted from 0.

## 9.1 Making a list and reading it

```
let primes = [2, 3, 5, 7]
print(primes, len(primes), primes[0], primes[3])
print([], [[1, 2], [3]])
```

```
[2, 3, 5, 7] 4 2 7
[] [[1, 2], [3]]
```

The type of a list says what its items are: `primes` is a `list[int]`. Zeno works it out from the items, or you write it, which you must do for an empty list that has nothing yet to work it out from:

```
let names: list[str] = []
let grid: list[list[int]] = [[1, 2], [3, 4]]
print(names, grid[1][0])
```

```
[] 3
```

A position outside the list is an `IndexError`, and so is a negative one (Text gives the reason):

```
let primes = [2, 3, 5, 7]
print(primes[4])
```

```
error: uncaught IndexError: index 4 is out of range for a list of length 4
 --> main.zeno:2:7
  |
2 | print(primes[4])
  |       ^
```

## 9.2 Changing a list

```
let xs = [3, 1, 2]
xs.append(4)          # at the end
xs.insert(0, 9)       # at position 0, moving the rest up
print(xs)
xs[1] = 30            # replace the item at position 1
print(xs)
let last = xs.pop()   # take out the last item, and give it
let first = xs.pop(0) # take out the item at position 0
print(last, first, xs)
```

```
xs.clear()              # take out everything
print(xs)
```

```
[9, 3, 1, 2, 4]
[9, 30, 1, 2, 4]
4 9 [30, 1, 2]
[]
```

All of these change the list itself, which is why `xs` can be a `let`: the name always means the same list, while what the list holds changes. Values and Variables has the rule.

Every item must be of the list's type:

```
let xs = [1, 2]
xs.append("three")
```

```
error: an item added to this list should be int, but this is str
 --> main.zeno:2:11
  |
2 | xs.append("three")
  |           ^

1 error found. Nothing was run.
```

## 9.3 Every method of a list

| Method | Does |
|---|---|
| `append(x)` | puts `x` at the end |
| `insert(at, x)` | puts `x` at position `at`, moving the rest up |
| `pop()` | takes out the last item, and gives it |
| `pop(at)` | takes out the item at `at`, and gives it |
| `clear()` | takes out every item |
| `reverse()` | turns the list round, where it is |
| `sort()` | puts numbers or text in order, smallest first, where it is |
| `find(x)` | the position of the first item equal to `x`, or `none` |
| `slice(from, to)` | a new list of the items from `from` up to, not including, `to` |
| `slice(from)` | a new list of the items from `from` to the end |
| `join(sep)` | for a list of text: one text, with `sep` between the items |
| `map(f)` | a new list of `f` of each item |
| `filter(f)` | a new list of the items for which `f` is `true` |
| `reduce(f, start)` | the items folded into one value with `f`, starting from `start` |

Beside them stand the functions `len(xs)`, `sum(xs)`, `min(xs)` and `max(xs)`, the operator `x in xs`, and `+` to join two lists into a new one.

```
let numbers = [5, 3, 9, 1]
```

```
numbers.sort()
print(numbers, numbers.find(9), numbers.find(4), 3 in numbers)
numbers.reverse()
print(numbers, numbers.slice(1, 3), numbers + [0])
print(["pear", "fig", "apple"].join(", "))
print(sum(numbers), min(numbers), max(numbers))
```

```
[1, 3, 5, 9] 3 none true
[9, 5, 3, 1] [5, 3] [9, 5, 3, 1, 0]
pear, fig, apple
18 1 9
```

### 9.3.1 What there is no method for

Some methods that other languages have are left out, because Zeno already says the thing in one short expression:

| In other languages | In Zeno |
|---|---|
| `xs.contains(x)` | `x in xs` |
| `sorted(xs)` | `let ys = xs.slice(0)`, then `ys.sort()` |
| `xs.remove(x)` | `xs.find(x)`, then `xs.pop(at)` |
| `xs.extend(ys)` | `xs += ys` |
| `xs[1:3]` | `xs.slice(1, 3)` |

**Why.** A method gets into the library only if most programs in the big languages use it and Zeno cannot already say it in one short expression. That keeps one way to write each thing. (Decision 6.1.)

### 9.3.2 sort and reverse change the list

`sort` and `reverse` work where the list is and give back nothing. To keep the old order, sort a copy, which `slice(0)` makes:

```
let readings = [5, 3, 9, 1]
let in_order = readings.slice(0)
in_order.sort()
print(readings, in_order)
```

```
[5, 3, 9, 1] [1, 3, 5, 9]
```

`sort` knows how to order numbers and text, which have one order that nobody has to be told. Anything else it refuses, with the reason:

```
let flags = [true, false]
flags.sort()
```

```
error: `sort` puts numbers or text in order, smallest first, and this is list[bool].
    ↪ There is no one order for those items, so write the loop that puts them in yours
 --> main.zeno:2:1
  |
2 | flags.sort()
```

```
  | ^

1 error found. Nothing was run.
```

**Why.** A new list is new memory, which a program on a small board may not want to spend, and a method that gives the list back invites `let ys = xs.sort()`, where it is not clear whether `xs` changed. `sort` is *heapsort* in every engine: it needs no second list, never calls itself, and is never slow, which is what firmware wants. (Decisions 6.5 and 6.6.)

### 9.3.3 find, and taking out by value

`find` gives a position or `none`. To take out an item by its value, find it first, and say what happens when it is not there:

```
let names = ["Ada", "Grace", "Alan"]
let at = names.find("Grace")
if at is not none {
    names.pop(at)
}
print(names)
```

```
["Ada", "Alan"]
```

## 9.4 Joining lists

`+` makes a new list from two, and `+=` adds a list to the end of a list variable. Both lists must hold the same kind of thing:

```
var log: list[str] = ["start"]
log += ["read", "write"]
print(log + ["stop"])
```

```
["start", "read", "write", "stop"]
```

```
let xs = [1, 2]
print(xs + ["a"])
```

```
error: `+` joins two lists that hold the same kind of thing, and these are list[int] and
    ↪ list[str]
 --> main.zeno:2:7
  |
2 | print(xs + ["a"])
  |       ^

1 error found. Nothing was run.
```

## 9.5 Comprehensions

A *comprehension* builds a new list from another in one line: an expression for each item, a `for`, and if you like an `if` that keeps only some items:

```
let numbers = range(1, 11)
```

```
print([n * n for n in numbers])
print([n for n in numbers if n % 3 == 0])
print([word.upper() for word in ["a", "b"]])
```

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
[3, 6, 9]
["A", "B"]
```

It reads the way it would be said: "n times n, for each n in numbers".

## 9.6 map, filter and reduce

The same work can be done by handing a function to a method. `map` applies it to each item, `filter` keeps the items it says `true` for, and `reduce` folds the list into one value:

```
function square(n: int) -> int {
    return n * n
}
function is_even(n: int) -> bool {
    return n % 2 == 0
}
function add(total: int, n: int) -> int {
    return total + n
}

let numbers = [1, 2, 3, 4, 5, 6]
print(numbers.map(square))
print(numbers.filter(is_even))
print(numbers.reduce(add, 0))
print(numbers.filter(is_even).map(square).reduce(add, 0))
```

```
[1, 4, 9, 16, 25, 36]
[2, 4, 6]
21
56
```

The last line reads left to right, in the order things happen: take the numbers, keep the even ones, square them, add them up.

> **Why.** As methods, `map`, `filter` and `reduce` read left to right. As plain functions, the same chain is `reduce(add, map(square, filter(is_even, numbers)))`, which has to be read from the inside out. (Decision 3.6b.)

A function can also be written in place, with no name. Functions as Values covers that.

## 9.7 Two names, one list

A list is not copied when it is given another name, passed to a function or put in another list. Every name sees the one list:

```
function add_reading(readings: list[float], r: float) {
    readings.append(r)
}
```

```
let mine = [20.5]
let also_mine = mine
add_reading(also_mine, 21.0)
print(mine)
```

```
[20.5, 21.0]
```

When you want a separate list, make one: `xs.slice(0)` copies the whole list.

> **Why.** Sharing is what Python, Java and JavaScript programmers expect, and it is the cheapest thing to do. In an ordinary build a garbage collector makes it safe; a build with no collector gives each list one owner instead, and Building with No Collector explains how. (Decision 3.10.)

## 9.8 Equal lists

Two lists are equal when they have equal items in the same order:

```
print([1, 2] == [1, 2], [1, 2] == [2, 1])
```

```
true false
```

### What this chapter said

A list holds values of one type, in order, from position 0. It grows with `append`, `insert` and `+=`, shrinks with `pop` and `clear`, and is ordered with `sort` and `reverse`, all in place. `find`, `slice`, `join`, `map`, `filter` and `reduce` give new values. Comprehensions build lists in one line. A list is shared, never copied without being asked.

### Things to try

1. Write `function median(values: list[float]) -> float` that does not change the list it is given.
2. Take every second item of a list with a comprehension and `range`.
3. Remove every `0` from a list of numbers, first with a loop and `pop`, then with `filter`. Which is easier to get right, and why?
4. Write a list of lists that is a 3 by 3 grid of zeros, set the middle to 1, and print it row by row.

CHAPTER 10

# Dicts

A *dict*, short for dictionary, files values under *keys*. Where a list answers "what is at position 3?", a dict answers "what is filed under `"Ada"`?".

## 10.1 Writing a dict

A dict is written like a list, in square brackets, with `key: value` where a list has an item. An empty dict is `[:]`:

```
let ages = ["Ada": 36, "Alan": 41]
let seen: dict[str, int] = [:]
print(ages, seen, len(ages))
```

```
["Ada": 36, "Alan": 41] [:] 2
```

Its type is written `dict[str, int]`: the type of the keys, then the type of the values. As with a list, Zeno works it out from what is written, and an empty dict needs it written.

> **Why.** In Zeno a pair of braces is always a block of code, and a pair of square brackets always holds values, so nobody reading a line has to work out which a bracket is. `{}` after `if ready` could be read either way in a language that used braces for both. `[:]` is the empty dict because the colon is what makes a dict a dict. (Decision 6.29.)

## 10.2 Keys and values

A key is a `str` or an `int`. All the keys of one dict are of one type, and so are all its values:

```
let meanings = [404: "not found", 200: "ok"]
print(meanings[200])
```

```
ok
```

```
let mixed = ["a": 1, 2: 3]
```

```
error: the keys of a dict are all of one type. This one is int, and the ones before it
    ↪ are str
 --> main.zeno:1:22
  |
1 | let mixed = ["a": 1, 2: 3]
  |                      ^

1 error found. Nothing was run.
```

> **Why.** A key has to compare exactly. `0.1 + 0.2` is not `0.3`, so a value filed under one float could never be found under the other: a float key is a mistake the language can simply not offer. A list or an object can change while it is a key, and would then be filed in the wrong place. (Decision 6.31.)

A value can be anything a list can hold: numbers, text, lists, objects, other dicts. What is under a key is the thing itself, not a copy, so changing it changes what is in the dict:

```
let readings: dict[str, list[float]] = [:]
readings["volt"] = [12.0]
readings["volt"].append(12.5)
print(readings)
```

```
["volt": [12.0, 12.5]]
```

The one thing a value cannot be is `none`, for a reason that comes up below with `get`.

## 10.3 Reading

There are three ways to ask a dict about a key, and each says what the program knows:

| Written | Means | If the key is not there |
|---|---|---|
| `d[key]` | it is there | a `KeyError` |
| `d.get(key)` | it may not be | `none` |
| `d.get(key, instead)` | it may not be | `instead` |
| `key in d` | is it there? | `false` |

```
let ages = ["Ada": 36, "Alan": 41]
print(ages["Ada"], "Bob" in ages, ages.get("Bob"), ages.get("Bob", 0))
```

```
36 false none 0
```

```
let ages = ["Ada": 36]
print(ages["Bob"])
```

```
error: uncaught KeyError: this dict has no key "Bob"
 --> main.zeno:2:7
  |
2 | print(ages["Bob"])
  |       ^
```

`get` with one argument may give `none`, so its answer must be tested before it is used as a number, exactly as with `find`. That is why a value cannot itself be `none`: in a dict of `int or none`, a `none` from `get` could mean either "no such key" or "the key is there and holds none", and no program could tell which.

```
let scores: dict[str, int or none] = [:]
```

```
error: a value of a dict cannot be `int or none`: `get` gives `none` for a key that is
    ↪ not there, and that could not be told from a value that is `none`. Make it a `
    ↪ dict[..., int]`, and leave the key out where there is no value
 --> main.zeno:1:1
  |
```

```
1 | let scores: dict[str, int or none] = [:]
  | ^

1 error found. Nothing was run.
```

Leaving the key out says the same thing with what is already there.

> **Why.** `d[key]` does not give `none` for a missing key, because then every read would be an `int or none` to test first, even when the key was stored on the line above. Nor does it give 0, which would invent a value nobody stored and turn a misspelt key into a wrong total. (Decision 6.34.)

## 10.4 Storing and taking out

`d[key] = value` stores a value: a new key is added, and a key that is there gets the new value. `pop(key)` takes a key out and gives back its value. `clear()` empties the dict.

```
let ages = ["Ada": 36, "Alan": 41]
ages["Grace"] = 85
ages["Ada"] = 37
print(ages)
print(ages.pop("Alan"), ages)
ages.clear()
print(ages)
```

```
["Ada": 37, "Alan": 41, "Grace": 85]
41 ["Ada": 37, "Grace": 85]
[:]
```

The commonest use of a dict is counting, and it is one line:

```
let words = "the cat and the hat and the bat".split()
let counts: dict[str, int] = [:]
for word in words {
    counts[word] = counts.get(word, 0) + 1
}
print(counts)
```

```
["the": 3, "cat": 1, "and": 2, "hat": 1, "bat": 1]
```

The same key written twice in one dict is surely a slip, so it is refused:

```
let limits = ["low": 1, "low": 2]
```

```
error: this key is written twice in the dict. A dict has each key once, so the second
    ↪ would replace the first. Take one of them out
 --> main.zeno:1:25
  |
1 | let limits = ["low": 1, "low": 2]
  |                         ^

1 error found. Nothing was run.
```

## 10.5 Order

A dict remembers the order its keys were first stored in. Printing it, looping over it, `keys()` and `values()` all follow that order, and storing under a key that is there does not move it.

> **Why.** It is the one order a reader can predict from the program, and it is the same on every run, in every engine and in every build. The order a hash table happens to have would differ between them. (Decision 6.33.)

## 10.6 Looping over a dict

A loop with one name is given the keys. A loop with two names is given each key and the value under it, following the rule for every loop: the second name is what is at the first.

```
let stock = ["apples": 4, "pears": 0, "figs": 12]
for fruit in stock {
    print(fruit)
}
for fruit, count in stock {
    if count == 0 {
        print("out of", fruit)
    }
}
print(stock.keys(), stock.values())
```

```
apples
pears
figs
out of pears
["apples", "pears", "figs"] [4, 0, 12]
```

As with a list, the loop goes through the keys as they were when it began, so the body may store into the dict and take keys out. `keys()` and `values()` give new lists that are the caller's own.

## 10.7 Equal dicts

Two dicts are equal when they have the same keys with the same values. The order does not matter here: it says how a dict was built, and == asks what it holds.

```
print(["x": 1, "y": 2] == ["y": 2, "x": 1])
```

```
true
```

## 10.8 Every method of a dict

| Written | Does |
|---|---|
| `d[key]` | the value under `key`; a `KeyError` if it is not there |
| `d[key] = value` | stores `value` under `key` |
| `key in d` | whether `key` is there |
| `len(d)` | how many keys there are |
| `d.get(key)` | the value, or `none` |
| `d.get(key, instead)` | the value, or `instead` |
| `d.pop(key)` | takes the key out and gives its value |
| `d.keys()`, `d.values()` | new lists of the keys and of the values, in order |
| `d.clear()` | takes every key out |

### What this chapter said

A dict files values of one type under keys of one type, a `str` or an `int`. It is written `["key": value]`, empty `[:]`, with the type `dict[str, int]`. `d[key]` expects the key to be there, `get` does not, and `in` asks. A dict keeps the order its keys arrived in, and a loop with two names is given each key and its value.

### Things to try

1. Count the letters in a sentence, then print them in the order they first appeared.
2. Build a dict from a name to a list of marks, add three marks for each of two names, and print each name with the average of its marks.
3. Turn a `dict[str, int]` round, so that the values become keys. What should happen when two keys have the same value?
4. Write a dict with a key that is a float, and read the message.

CHAPTER 11

# Classes

A *class* is a kind of value that you make up yourself: a reading from a sensor, a point on a screen, a motor with a speed. It puts several named values, its *fields*, together, and gives them functions of their own, its *methods*. Each value made from a class is an *object*.

## 11.1 Fields

A class lists its fields with their types. Its name is the *constructor*: calling it makes an object, with a value for each field in the order they are listed, or by name:

```
class Point {
    x: float
    y: float
}

let p = Point(3.0, 4.0)
let q = Point(y=1.0, x=2.0)
print(p, q)
print(p.x, q.y)
```

```
Point(x=3.0, y=4.0) Point(x=2.0, y=1.0)
3.0 1.0
```

An object is printed with its class name and every field, in the way you would write it to make it again.

A field can have a default, which the constructor uses when it is not given:

```
class Counter {
    name: str
    count: int = 0
    step: int = 1
}

let visits = Counter("visits")
let pages = Counter("pages", step=5)
print(visits, pages)
```

```
Counter(name="visits", count=0, step=1) Counter(name="pages", count=0, step=5)
```

A field without a default must be given, and a field that is not in the class cannot be:

```
class Point {
    x: float
    y: float
}
let p = Point(1.0)
```

```
error: `Point` needs a value for `y`
 --> main.zeno:5:9
  |
5 | let p = Point(1.0)
  |         ^

1 error found. Nothing was run.
```

```
class Point {
    x: float
}
let p = Point(1.0, 2.0)
```

```
error: `Point` takes 1 argument, but 2 were given
 --> main.zeno:4:9
  |
4 | let p = Point(1.0, 2.0)
  |         ^

1 error found. Nothing was run.
```

**Why.** The fields are the shape of the data, and the shape is written in one place, at the top of the class, not discovered by reading assignments scattered through an initialiser. A fixed list of fields is also what lets a class become a plain C struct in a compiled program. (Decision 3.7.)

## 11.2 Changing a field

A field is changed with =, like a variable. The object can be a `let`: the name always means the same object, while what is inside it changes, exactly as with a list.

```
class Motor {
    speed: int = 0
}
let m = Motor()
m.speed = 40
m.speed += 10
print(m.speed)
```

```
50
```

There is no way to add a field the class did not list. A misspelt name is a mistake found before the program runs, not a new field made in passing:

```
class Motor {
    speed: int = 0
}
let m = Motor()
m.sped = 40
```

```
error: `Motor` has no field `sped`
 --> main.zeno:5:1
  |
5 | m.sped = 40
  | ^
```

```
1 error found. Nothing was run.
```

## 11.3 Methods

A method is a function written inside the class. Its first parameter is `self`, the object it was called on:

```
class Counter {
    name: str
    count: int = 0
    step: int = 1

    function bump(self) {
        self.count += self.step
    }

    function report(self) -> str {
        return f"{self.name}: {self.count}"
    }
}

let visits = Counter("visits")
visits.bump()
visits.bump()
print(visits.report())
print(visits)
```

```
visits: 2
Counter(name="visits", count=2, step=1)
```

`visits.bump()` is the call `bump` with `visits` as its `self`. A method takes other parameters after `self`, with types, defaults and names as any function does.

A method must name `self` even when it does not use it:

```
class Lamp {
    on: bool = false
    function flip() {
        print("flip")
    }
}
```

```
error: the method `flip` needs `self` as its first parameter
 --> main.zeno:3:5
  |
3 |     function flip() {
  |     ^

1 error found. Nothing was run.
```

**Why.** Writing `self` shows what a method is: an ordinary function that is given its object as the first argument. There is no hidden `this` that appears from nowhere, and inside the method `self.count` and a plain `count` can never be confused. (Decision 3.8.)

## 11.4 Making an object with a function

There is no constructor of your own to write. The fields in order are the constructor. When making an object needs work, such as checking the values or working one out from another, write a function that does the work and calls the constructor:

```
class Range {
    low: float
    high: float
}

function range_around(middle: float, width: float) -> Range {
    if width < 0.0 {
        raise ValueError(f"a width cannot be negative, and {width} is")
    }
    return Range(middle - width / 2.0, middle + width / 2.0)
}

print(range_around(10.0, 4.0))
```

```
Range(low=8.0, high=12.0)
```

The function's name says what it does, which a second constructor could not. When a class must only ever be made through such a function, its fields are made `private`, which Programs in Several Files explains: then only its own file can call the constructor.

> **Why.** A constructor written by the programmer is a second place where the shape of the object is decided, and it can disagree with the first. Python's `__init__` can set fields the class never mentions. A named function is plain code that anyone can read. (Decision 3.7, as settled in chapter 11 of the decisions.)

## 11.5 Objects are shared

An object, like a list, is not copied when it is given another name, passed to a function or put in a list. Every name sees the one object:

```
class Counter {
    count: int = 0
    function bump(self) {
        self.count += 1
    }
}

function bump_twice(c: Counter) {
    c.bump()
    c.bump()
}

let mine = Counter()
let also_mine = mine
bump_twice(also_mine)
print(mine.count)
```

```
2
```

This is what lets a function change an object it is given, and it is the same rule as for lists (Lists).

## 11.6 Equal objects

Two objects are equal when they are of the same class and every field is equal. == asks what they hold, not whether they are the same object:

```
class Point {
    x: int
    y: int
}
print(Point(1, 2) == Point(1, 2), Point(1, 2) == Point(2, 1))
```

```
true false
```

> **Why.** This is what == means for numbers, text, lists and dicts, so it means the same for objects. Java's == on objects asks whether two names point at one object, and a whole kind of mistake follows from comparing two equal strings and getting `false`. (Decision 2.5.)

## 11.7 Objects inside objects

A field can hold a list, a dict or another object. It can even hold an object of its own class, as long as it may also be `none`, so that the chain can end:

```
class Node {
    value: int
    next: Node or none = none
}

let third = Node(3)
let second = Node(2, third)
let first = Node(1, second)

var at: Node or none = first
var total = 0
while at is not none {
    total += at.value
    at = at.next
}
print(total)
```

```
6
```

`Node or none` says that the field holds a `Node` or nothing. Values That May Be Missing explains why Zeno lets `at.value` be read inside the loop: the condition has already ruled `none` out.

## 11.8 A class with a list

A class that keeps a list of readings is a common shape in firmware:

`log.zeno`

```
class Log {
```

```
    limit: int
    readings: list[float] = []

    function add(self, r: float) {
        self.readings.append(r)
        if len(self.readings) > self.limit {
            self.readings.pop(0)
        }
    }

    function average(self) -> float {
        return sum(self.readings) / float(len(self.readings))
    }
}

let log = Log(3)
for r in [20.0, 21.0, 22.0, 23.0, 24.0] {
    log.add(r)
}
print(log.readings, log.average())
```

```
[22.0, 23.0, 24.0] 23.0
```

The default `[]` is a new list for each object, made afresh every time the constructor runs, just as a default parameter is (Functions). Two logs never share one list by accident.

## 11.9 No parent classes

A class in Zeno does not inherit from another class. Writing one class as another is refused, with what to do instead:

```
class Animal {
    name: str
}
class Dog is Animal {
    breed: str
}
```

```
error: `Animal` is a class, not an interface, so `Dog` cannot be one. A class has no
    ↪ parent class in Zeno. Declare what the two share as an `interface`, and let both
    ↪ be it
 --> main.zeno:4:1
  |
4 | class Dog is Animal {
  | ^

1 error found. Nothing was run.
```

What inheritance is mostly used for, one function that works with several classes, is done by an *interface*: a list of methods that several classes promise to have. Interfaces is about them.

> **Why.** With inheritance, the fields and methods of a class are spread over the class and all its parents, and a method can be replaced anywhere along the chain, so reading a class means reading a family. In Zeno a class is read in one place. (Decision 10.4, and the settled decision on 0.1 and 3.7.)

## 11.10 Using a class above where it is written

Like a function, a class can be used anywhere in its file, above or below where it is declared:

```
print(Pair(1, 2))

class Pair {
    left: int
    right: int
}
```

```
Pair(left=1, right=2)
```

### What this chapter said

A class lists its fields, with types and perhaps defaults, and its name is the constructor, which takes the fields in order or by name. Methods are functions inside the class whose first parameter is `self`. Fields cannot be added after the fact. An object is shared, compared by its contents, and printed as it would be written. A function with a name builds an object that needs checking. There is no inheritance; interfaces do that work.

### Things to try

1. Write a class `Rect` with a width and a height, and methods `area` and `is_square`.
2. Give `Counter` a method `reset` and a field `limit`, so that `bump` goes back to 0 after the limit.
3. Write `function square(side: float) -> Rect` that refuses a negative side with a `ValueError`.
4. Build a chain of four `Node` objects and write a function that gives the largest value in it.
5. Try to set a field that `Rect` does not have, and read the message.

CHAPTER 12

# Enums and match

Some values are one of a short list of choices: a traffic light is red, amber or green; a motor is stopped, running or faulty. An *enum* (short for *enumeration*) is a type that lists the choices, and `match` is how a program does something different for each.

## 12.1 An enum of plain choices

```
enum Light {
    Red
    Amber
    Green
}

let now = Red
print(now, now == Red, Amber != Green)
```

```
Red true true
```

Each choice is a *case*. A case is written by its name, and a value of `Light` can only ever be one of the three. Compare that with keeping the light in an `int` where 0 is red, 1 amber and 2 green: nothing would stop a 7, and nothing would tell a reader what 1 means.

## 12.2 match

`match` takes a value and runs the block of the first `case` that fits it:

```
enum Light {
    Red
    Amber
    Green
}

function next(light: Light) -> Light {
    match light {
        case Red { return Green }
        case Green { return Amber }
        case Amber { return Red }
    }
}

var light = Red
for turn in range(4) {
    print(light)
    light = next(light)
}
```

```
Red
Green
Amber
Red
```

### 12.2.1 Every case must be handled

A `match` on an enum must say what to do for every case. Leave one out and the program does not run, and the message names the case that is missing:

```
enum Light {
    Red
    Amber
    Green
}

function stop_here(light: Light) -> bool {
    match light {
        case Red { return true }
        case Amber { return true }
    }
}
```

```
error: this `match` does not handle Green. Add a `case` for it, or a final `case _`
 --> main.zeno:8:5
  |
8 |     match light {
  |     ^

1 error found. Nothing was run.
```

This is the main reason to use an enum. When a new case is added a year later, say `Flashing`, every `match` in the program that does not yet know about it becomes an error, and the compiler lists them. Nothing falls through in silence.

A final `case _` stands for every case not named above it, when they all mean the same:

```
enum Light {
    Red
    Amber
    Green
}

function may_go(light: Light) -> bool {
    match light {
        case Green { return true }
        case _ { return false }
    }
}
print(may_go(Green), may_go(Amber))
```

```
true false
```

> **Why.** "A reading is either a value or a fault code" is one of the most common shapes in real programs, and most languages make it awkward. An enum says it directly, and a `match` that must name every case is the check that a C `switch` never gave. (Decision 3.11.)

## 12.3 Cases that carry values

A case can carry values of its own, named and typed like the parameters of a function. Then the case is written like a call:

```
enum Shape {
    Circle(radius: float)
    Rect(width: float, height: float)
    Dot
}

let shapes = [Circle(1.0), Rect(2.0, 3.0), Dot]
print(shapes)
```

```
[Circle(1.0), Rect(2.0, 3.0), Dot]
```

All three are `Shapes`, so they can go in one list. Each carries what that case needs and nothing more: a circle has a radius, a rectangle two sides, and a dot nothing at all.

`match` takes the values out again. The names in brackets after a case are new names, given the values that case carries, in order:

```
enum Shape {
    Circle(radius: float)
    Rect(width: float, height: float)
    Dot
}

function area(shape: Shape) -> float {
    match shape {
        case Circle(r) { return 3.14159 * r * r }
        case Rect(w, h) { return w * h }
        case Dot { return 0.0 }
    }
}

for shape in [Circle(1.0), Rect(2.0, 3.0), Dot] {
    print(shape, area(shape))
}
```

```
Circle(1.0) 3.14159
Rect(2.0, 3.0) 6.0
Dot 0.0
```

A value that is not needed is written `_`: `case Rect(w, _)` takes the width and leaves the height.

> **Why.** A value read straight off the enum, such as `shape.radius`, would be a mistake whenever the shape turned out to be a `Rect`, so the checker refuses it and points to `match`. Inside `case Circle(r)`, `r` is there for certain, because the case has already been found to fit.

## 12.4 match on numbers and text

A `match` can also take a number or a piece of text, and compare it with each case in turn:

```
function describe(code: int) -> str {
    match code {
```

```
        case 200 { return "ok" }
        case 404 { return "not found" }
        case _ { return "something else" }
    }
}
print(describe(200), describe(404), describe(500))
```

```
ok not found something else
```

There is no way to list every `int` or every `str`, so such a `match` needs a final `case _`:

```
let answer = "yes"
match answer {
    case "yes" { print("agreed") }
    case "no" { print("refused") }
}
```

```
error: this `match` needs a final `case _` to handle every other value
 --> main.zeno:2:1
  |
2 | match answer {
  | ^

1 error found. Nothing was run.
```

### 12.4.1 Giving the value a name

A plain name as the last case fits anything, like _, and gives the value that name inside the block:

```
for word in ["start", "stop", "jump"] {
    match word {
        case "start" { print("starting") }
        case "stop" { print("stopping") }
        case other { print("no such command:", other) }
    }
}
```

```
starting
stopping
no such command: jump
```

## 12.5 Blocks, not arrows

Each case is `case` something, then a block in braces. A case whose block is short can sit on one line, as above; a longer one is laid out like any other block:

```
let level = 3
match level {
    case 0 { print("off") }
    case 3 {
        print("high")
        print("check the fan")
    }
    case _ { print("on") }
}
```

```
high
check the fan
```

**Why.** Every other block in Zeno is a keyword, something, and braces: `if ready { }`, `while busy { }`. An arrow, as in `3 => ...`, would be a second way to start a block. (Decision 1.9.)

## 12.6 An enum in a state machine

Firmware is full of *state machines*: a device is in one state, and each event moves it to another. An enum names the states and a `match` says what each event does in each:

door.zeno

```
enum Door {
    Closed
    Opening(percent: int)
    Open
}

function step(door: Door) -> Door {
    match door {
        case Closed { return Opening(0) }
        case Opening(p) {
            if p >= 75 {
                return Open
            }
            return Opening(p + 25)
        }
        case Open { return Open }
    }
}

var door = Closed
for tick in range(7) {
    print(tick, door)
    door = step(door)
}
```

```
0 Closed
1 Opening(0)
2 Opening(25)
3 Opening(50)
4 Opening(75)
5 Open
6 Open
```

The compiler checks that `step` knows what to do in every state. Add a case `Jammed` to `Door` and the program will not run until `step` has decided what a jammed door does.

## 12.7 Case names belong to the file

The cases of an enum are names in the file, like functions, which is why a case is written plain `Red`. Two enums in one file therefore cannot both have a case called `Red`:

```
enum Light {
```

```
    Red
    Green
}
enum Flag {
    Red
    White
}
```

```
error: `Red` is already a case of the enum `Light`. A case is written by its name alone,
    ↪ so two enums in one file cannot both have a `Red`. Give one of them another name
 --> main.zeno:5:1
  |
5 | enum Flag {
  | ^

1 error found. Nothing was run.
```

Give the cases names that say which enum they belong to, such as `FlagRed`, or put the enums in different files (Programs in Several Files).

### What this chapter said

An enum lists the cases a value can be, and a case can carry values of its own. `match` runs the block of the case that fits, takes out what a case carries, and must handle every case of an enum, either by name or with a final `case _`. A `match` on numbers or text always needs `case _`, and a plain name as the last case gives the value a name.

### Things to try

1. Write an enum `Coin` with four coins and a function that gives the value of each in cents.
2. Add `Triangle(base: float, height: float)` to `Shape` and run the program before changing `area`. Read the message.
3. Write `enum Reading` with `Value(volts: float)` and `Fault(code: int)`, and a function that prints a list of readings, with the fault codes in hexadecimal.
4. Add `Jammed` to the door, and a `Closing` state, so that an open door closes again.

CHAPTER 13

# Values That May Be Missing

Sometimes there is no answer. A search finds nothing, a sensor has not reported yet, a line that should hold a number holds a word. Zeno has one value for "nothing here": `none`. This chapter is about where it may appear, and how Zeno makes sure a program never uses it by mistake.

## 13.1 none only where the type says so

A value can be `none` only if its type says it may be. The type is written `T or none`, and reads as it means:

```
function find_user(id: int) -> str or none {
    if id == 1 {
        return "ada"
    }
    return none
}

print(find_user(1), find_user(2))
```

```
ada none
```

A plain `str` is always a piece of text. A `str or none` is a piece of text or nothing, and the program has to find out which before it treats it as text:

```
function find_user(id: int) -> str or none {
    if id == 1 {
        return "ada"
    }
    return none
}

print(find_user(1).upper())
```

```
error: this is str or none. Check it with `is not none` before using `.upper`
 --> main.zeno:8:7
  |
8 | print(find_user(1).upper())
  |       ^

1 error found. Nothing was run.
```

**Why.** Using a null value is the commonest crash in C, Java and JavaScript. Tony Hoare, who brought null into programming in 1965, called it his "billion-dollar mistake". In those languages any reference can be null, so every use is a risk. In Zeno only a value whose type says `or none` can be, and the check cannot be forgotten. (Decisions 1.13 and 3.12.)

## 13.2 Asking: is none, is not none

`x is none` and `x is not none` ask the question. Inside an `if` that has found the value is there, Zeno knows it, and the value can be used as a plain value:

```
function find_user(id: int) -> str or none {
    if id == 1 {
        return "ada"
    }
    return none
}

for id in [1, 2] {
    let user = find_user(id)
    if user is not none {
        print(user.upper())
    } else {
        print("no user", id)
    }
}
```

```
ADA
no user 2
```

The question is always asked with `is`. `== none` is refused, with the spelling to use, and so is asking about a value that can never be `none`, because the answer would always be `false`:

```
let count = 3
if count is none {
    print("no count")
}
```

```
error: `is none` is always false: this is int, which is never `none`. Only a value whose
    ↪ type says `or none` can be
 --> main.zeno:2:4
  |
2 | if count is none {
  |    ^

1 error found. Nothing was run.
```

There is no new syntax to learn for this. The program writes the `if` it would have written anyway, and the checker follows it. This is called *narrowing*: inside the `if`, the type of `user` is narrowed from `str or none` to `str`.

`is` is only for `none`. Values are compared with `==`:

```
let x = 3
print(x is 3)
```

```
error: `is` only tests for `none`. Use `==` to compare values
 --> main.zeno:2:12
  |
2 | print(x is 3)
  |            ^
```

> **Why.** In Python, `is` asks whether two names are one object, which gives surprising answers on numbers and text and is a well-known trap. Keeping `is` for `none` leaves one way to compare values and one way to ask whether something is there. (Decision 1.13.)

## 13.3 What the checker follows

The checker follows the ordinary shapes a program takes.

**Leaving early.** After `if x is none { return ... }`, the rest of the block can only be reached when `x` is there:

```
function shout(text: str or none) -> str {
    if text is none {
        return ""
    }
    return text.upper()
}
print(shout("hi"), shout(none))
```

```
HI
```

The same goes for `raise`, `break` and `continue`.

**and, or and not.** A test joined with `and` knows the first half when it asks the second:

```
let name: str or none = "Ada"
if name is not none and len(name) > 2 {
    print(name, "is long enough")
}
```

```
Ada is long enough
```

**Filling in a default.** After an `if` that gives the variable a value when it had none, the variable has a value on both paths, and the checker knows it:

```
function greeting(given: str or none) -> str {
    var name = given
    if name is none {
        name = "friend"
    }
    return f"Hello, {name.upper()}"
}
print(greeting("Ada"), greeting(none))
```

```
Hello, ADA Hello, FRIEND
```

**Walking a chain.** A `while` that asks `is not none` knows it inside, and an assignment in the body makes the variable uncertain again until the next question:

```
class Node {
    value: int
    next: Node or none = none
```

```
}

var node: Node or none = Node(1, Node(2, Node(3)))
var total = 0
while node is not none {
    total += node.value
    node = node.next
}
print(total)
```

```
6
```

**Why.** Without these, the two most ordinary uses of `none`, walking a linked list and filling in a default, would not pass the checker, and the program would have to copy the value into a second variable just to please it. TypeScript and Kotlin follow assignments the same way. (Decision 3.12.)

### 13.3.1 A field is asked about through a name

The checker narrows names, not fields. A field that may be `none` is first given a name of its own, and the name is asked about:

```
class Sensor {
    last: float or none = none
}
let s = Sensor()
s.last = 21.5
if s.last is not none {
    print(s.last + 1.0)
}
```

```
error: `+` does not work between float or none and float. `last` is a field, and a field
    ↪  can be given a new value between a test and its use, so a test of it is not
    ↪ remembered. Copy it into a variable with `let`, and test that for `none`
 --> main.zeno:7:11
  |
7 |     print(s.last + 1.0)
  |           ^

1 error found. Nothing was run.
```

```
class Sensor {
    last: float or none = none
}
let s = Sensor()
s.last = 21.5
let last = s.last
if last is not none {
    print(last + 1.0)
}
```

```
22.5
```

A field can be changed by any code that holds the object, including a method called between the question and the use, so an answer about a field could be out of date by the next line. A local name cannot be changed behind the program's back.

## 13.4 Functions that may give none

The library gives `none` where there may be no answer: `find` on text and lists (Text), `get` on a dict (Dicts) and `input()` when there are no more lines (Reading and Writing). Your own functions do the same by saying `-> T or none`.

A function that says it may give `none` gives `none` when it reaches its end, since that is one of the answers it promised:

```
function first_even(xs: list[int]) -> int or none {
    for x in xs {
        if x % 2 == 0 {
            return x
        }
    }
}
print(first_even([1, 3, 4, 5]), first_even([1, 3]))
```

```
4 none
```

## 13.5 none in a match

A `match` on a value that may be `none` can have `case none`, and every case after it knows the value is there. For an enum, `case none` and a case for each of the enum's own cases make the match complete:

```
enum Reading {
    Value(volts: float)
    Fault(code: int)
}

function describe(r: Reading or none) -> str {
    match r {
        case none { return "no reading yet" }
        case Value(v) { return f"{v:.1f} V" }
        case Fault(code) { return f"fault 0x{code:02x}" }
    }
}

print(describe(none), describe(Value(3.3)), describe(Fault(7)))
```

```
no reading yet 3.3 V fault 0x07
```

Leave `case none` out, and the checker says so:

```
enum Reading {
    Value(volts: float)
    Fault(code: int)
}

function describe(r: Reading or none) -> str {
    match r {
        case Value(v) { return "value" }
        case Fault(code) { return "fault" }
    }
}
```

```
error: this `match` does not handle `none`. Add a `case` for it, or a final `case _`
 --> main.zeno:7:5
  |
7 |     match r {
  |     ^

1 error found. Nothing was run.
```

## 13.6 none is not zero

`none` is not `0`, not `false` and not `""`. It is its own value, and the only things that can be done with it are to print it, store it where the type allows, and ask whether a value is it. Conditions must be `bool`, so `if user` is refused, and the question is always written out: `if user is not none`.

### What this chapter said

`none` means "nothing here", and only a value whose type says `or none` can hold it. Before such a value is used, the program asks `is none` or `is not none`, and the checker follows the answer through `if`, early returns, `and`, defaults and `while`. Fields are asked about through a name of their own. A `match` can have `case none`. `is` is for `none` and nothing else.

### Things to try

1. Write `function parse_volts(text: str) -> float or none` that gives `none` for an empty text, and use it on a list of texts, printing a dash for each `none`.
2. Write a function that gives the last `Node` of a chain, or `none` for an empty one.
3. Take the `greeting` function and remove the `if`. Read the message, and say in your own words what it protects against.
4. Write `function largest(xs: list[int]) -> int or none` with no `max`. What should it give for `[]`, and why is that better than 0?

CHAPTER 14

# When Things Go Wrong

A correct program can still meet things that go wrong: a line that should hold a number holds a word, a divisor turns out to be zero, a file is not there. Zeno calls these *errors*, and gives a program three words to deal with them: `raise` to say that something has gone wrong, `try` and `except` to deal with it, and `raises` to say in a function's heading what it may raise.

## 14.1 Bugs and errors

Zeno draws a line between two kinds of thing that go wrong.

A *bug* is a mistake in the program itself: a misspelt name, `"abc" - 1`, calling something that is not a function. The checker finds almost all of them before the program runs, and the program does not start until they are fixed.

An *error* is something that can go wrong in a correct program: bad input, a division by a value that turned out to be zero, a position that came from outside. These happen while the program runs, and the program can catch them and carry on.

> **Why.** Catching a bug hides it. A program that catches "this name does not exist" and carries on has a mistake in it that nobody will ever see. Since the checker reports bugs before the program runs, there is nothing of that kind left to catch. (Decision 4.1.)

## 14.2 An error that is not caught

An error that nothing catches stops the program, with its kind, its message and the line it came from:

```
let readings = [20.5, 21.0]
print(readings[2])
```

```
error: uncaught IndexError: index 2 is out of range for a list of length 2
 --> main.zeno:2:7
  |
2 | print(readings[2])
  |       ^
```

Here `IndexError` is the *kind* of error, and the rest is its *message*. Every error has both.

## 14.3 try and except

Code that may raise an error goes in a `try` block. If an error is raised inside it, the rest of the block is skipped, and the `except` that names its kind runs instead:

```
for text in ["40", "fast", "7"] {
    try {
        let speed = int(text)
        print("speed", speed)
    } except ValueError {
        print("not a number:", text)
    }
}
```

```
speed 40
not a number: fast
speed 7
```

If nothing is raised, the `except` block does not run at all.

`except Kind as e` gives the error a name. `e.message` is its message and `e.kind` the name of its kind:

```
try {
    print(int("fast"))
} except ValueError as e {
    print(e.kind, "says:", e.message)
}
```

```
ValueError says: cannot turn "fast" into an int
```

A `try` can have several `excepts`. The first that fits is the one that runs:

```
function show(xs: list[int], at: int, by: int) {
    try {
        print(xs[at] // by)
    } except IndexError as e {
        print("position:", e.message)
    } except ZeroDivisionError {
        print("cannot divide by zero")
    }
}

show([10, 20], 1, 5)
show([10, 20], 9, 5)
show([10, 20], 0, 0)
```

```
4
position: index 9 is out of range for a list of length 2
cannot divide by zero
```

An error that no `except` names goes on as if the `try` were not there, up to whoever called this code, and so on until something catches it or the program stops.

## 14.4 The kinds of error

The kinds that Zeno itself raises are these:

| Kind | Raised when |
|---|---|
| `ValueError` | a value is the right type but cannot be used: `int("fast")`, `2 ** -1` |
| `IndexError` | a position is outside a list or a text |
| `KeyError` | a dict has no such key |
| `ZeroDivisionError` | a number is divided by zero |
| `OverflowError` | a whole number is too big for its type |
| `FileError` | a file cannot be read or written |
| `Error` | anything else; and `except Error` catches every kind |

`except Error` catches any error at all, which is useful at the very top of a program, to report what went wrong and stop tidily:

```
let big = 9223372036854775807
try {
    print(big + 1)
} except Error as e {
    print("stopped:", e.kind, e.message)
}
```

```
stopped: OverflowError the result is too big for an int
```

Appendix F lists every message that the library and the runtime give.

## 14.5 raise

A program raises an error of its own with `raise` and the kind, with a message that says what went wrong:

```
function check_speed(speed: int) -> int raises ValueError {
    if speed < 0 or speed > 100 {
        raise ValueError(f"a speed is 0 to 100, and {speed} is not")
    }
    return speed
}

for s in [40, 250] {
    try {
        print("set to", check_speed(s))
    } except ValueError as e {
        print(e.message)
    }
}
```

```
set to 40
a speed is 0 to 100, and 250 is not
```

A message is for the person who reads it when something goes wrong. The best ones say what was expected and what was found, as Zeno's own messages do.

Only an error can be raised, not a piece of text:

```
raise "something went wrong"
```

```
error: only an error can be raised, and this is str
 --> main.zeno:1:1
  |
1 | raise "something went wrong"
  | ^

1 error found. Nothing was run.
```

### 14.5.1 Raising again

An `except` block that cannot fully deal with an error can do what it can, then raise the same error again, so that the caller sees it:

```
try {
    print(int("fast"))
} except ValueError as e {
    print("noted:", e.message)
    raise e
}
```

```
noted: cannot turn "fast" into an int
error: uncaught ValueError: cannot turn "fast" into an int
 --> main.zeno:5:5
  |
5 |     raise e
  |     ^
```

## 14.6 raises: what a function may raise

`raises` after a function's heading lists the errors it may raise, so that whoever calls it can see them without reading the body:

```
function read_volts(text: str) -> float raises ValueError {
    return float(text)
}
print(read_volts("3.3"))
```

```
3.3
```

It follows the rule of every type in Zeno: it can be left out, and then Zeno works it out from the body and passes it on to callers. When it is written, it must be complete. A function that says what it raises and then leaves something out is told so:

```
function read_volts(text: str) -> float raises KeyError {
    return float(text)
}
```

```
error: `read_volts` can raise ValueError here but does not say so. Add `raises
    ↪ ValueError` to it, or catch the error
 --> main.zeno:2:12
  |
2 |     return float(text)
  |            ^

1 error found. Nothing was run.
```

In strict mode (`--strict`, Types, Written and Worked Out), as in every compiled program, it must always be written, so that every heading is a complete promise:

```
function read_volts(text: str) -> float {
    return float(text)
}
print(read_volts("3.3"))
```

```
$ zeno check --strict main.zeno
error: `read_volts` can raise ValueError here but does not say so. Add `raises
    ↪ ValueError` to it, or catch the error
 --> main.zeno:2:12
  |
2 |     return float(text)
  |            ^

1 error found. Nothing was run.
```

### 14.6.1 What counts

Only errors raised *on purpose* must be listed: a `raise`, a call to a function that raises, and `int()` or `float()` on text, which fail on bad input. The errors that operators raise, such as dividing by zero, a position out of range or an overflow, can be caught, but need not be listed.

> **Why.** A division or an index can fail on almost any line. If those had to be listed, nearly every function would say `raises ZeroDivisionError, IndexError`, and a label that is on everything tells nobody anything. Java's checked exceptions are widely disliked for exactly that. (Decision 1.17.)

## 14.7 An error that carries more

An error often needs to carry more than a message: how long was waited, which pin, what code a device sent back. An object of any class can be raised, with whatever fields it needs:

```
class Timeout {
    after_ms: int
    device: str
}

function wait_for(device: str) raises Timeout {
    raise Timeout(30, device)
}

try {
    wait_for("sensor")
} except Timeout as t {
    print("gave up on", t.device, "after", t.after_ms, "ms")
}
```

```
gave up on sensor after 30 ms
```

`except Error as e` catches such an object too. Then `e.kind` is the class's name, and `e.message` is the object as text:

```
class Timeout {
```

```
    after_ms: int
}

try {
    raise Timeout(30)
} except Error as e {
    print(e.kind, "/", e.message)
}
```

```
Timeout / Timeout(after_ms=30)
```

**Why.** A class already means "these named values, together", so there is no second way to declare an error. (Decision 1.17.)

## 14.8 What is not here

Zeno has no `finally`. Code that must run whether or not an error was raised is written after the `try`, with an `except` that catches what must be caught.

On a microcontroller, a compiled program can carry its errors as plain values instead, with no jumps at all, as firmware coding standards require. Nothing in the program changes for that; it is a choice made when building, and Firmware explains it.

**Why.** They are not C++ exceptions: there are no unwinding tables, and code that does not use `try` pays nothing for it. (Decisions 1.17 and 5.5i.)

### What this chapter said

A bug is found before the program runs; an error happens while it runs. `raise Kind("message")` raises an error, and `try { } except Kind as e { }` catches it, with `e.kind` and `e.message`. `except Error` catches every kind. An object of any class can be raised. `raises` lists what a function raises on purpose: worked out when left out, complete when written, and always written in strict mode.

### Things to try

1. Write a loop that asks for numbers with `input()` until it is given one that `int` accepts.
2. Write `function percent(part: int, whole: int) -> float` that raises a `ValueError` with a clear message when `whole` is 0, and catch it.
3. Make a class `SensorFault` with a pin and a code, raise it from a function, and catch it once by its own name and once as `Error`.
4. Remove `raises ValueError` from `check_speed` and run `zeno check --strict`. Then write `raises KeyError` instead. What does each message ask for?

CHAPTER 15

# Functions as Values

A function in Zeno is a value, like a number or a list. It can be given a name, passed to another function, kept in a list or a dict, and given back by a function. This chapter is about what that makes possible.

## 15.1 Passing a function

A function's name, without brackets after it, is the function itself. Passing it hands the function over, to be called later:

```
function twice(f: function(int) -> int, x: int) -> int {
    return f(f(x))
}

function add_one(n: int) -> int {
    return n + 1
}

print(twice(add_one, 5))
print(add_one)
```

```
7
<function add_one>
```

`twice` calls whatever function it was given, twice. You have already seen this with `map`, `filter` and `reduce` in Lists.

## 15.2 The type of a function

The type of a function is written as its heading with the names and the body taken out:

| Function | Its type |
|---|---|
| `function add_one(n: int) -> int` | `function(int) -> int` |
| `function add(a: int, b: int) -> int` | `function(int, int) -> int` |
| `function report(text: str)` | `function(str)` |
| `function now() -> int` | `function() -> int` |

A function fits a type when it can be called with those arguments and gives back what the type says. One that does not fit is refused at the call, with the reason:

```
function twice(f: function(int) -> int, x: int) -> int {
    return f(f(x))
```

```
}
function shout(text: str) -> str {
    return text.upper()
}
print(twice(shout, 1))
```

```
error: `f` of `twice` should be function(int) -> int, but the parameter `text` of this
    ↪ function is str
 --> main.zeno:7:13
  |
7 | print(twice(shout, 1))
  |             ^

1 error found. Nothing was run.
```

**Why.** There is nothing new to learn: the same keyword, the same brackets, the same `->` as a heading. A form such as `(int) -> int`, as in TypeScript and Swift, would be shorter, but it gives brackets a second meaning in types. (Decision 1.11a.)

## 15.3 A function with no name

A function that is used only once can be written where it is used, with the name left out:

```
let double = function(n: int) -> int { return n * 2 }
print(double(21))

let readings = [15.0, 25.0, 30.0]
print(readings.map(function(c) { return c * 1.8 + 32.0 }))
print(readings.filter(function(c) { return c > 20.0 }))
```

```
42
[59.0, 77.0, 86.0]
[25.0, 30.0]
```

When such a function is passed where a function type is expected, as with `map` here, its types can be left out: Zeno works them out from the type it is given.

**Why.** It is exactly a function declaration with the name left out, so there is only one way to write a function. Python's `lambda n: n * 2` is a second, smaller syntax that cannot hold statements, and JavaScript's `n => n * 2` would need the arrow that Zeno left out of `match`. (Decision 3.6a.)

For the commonest case, making a new list from an old one, a comprehension is shorter still: `[c * 1.8 + 32.0 for c in readings]`.

## 15.4 Functions in lists and dicts

Because a function is a value, it can be kept anywhere a value can. A dict from names to functions is a simple way to run a command by name:

```
function add(a: int, b: int) -> int {
    return a + b
}
function multiply(a: int, b: int) -> int {
```

```
    return a * b
}

let operations: dict[str, function(int, int) -> int] = [:]
operations["add"] = add
operations["times"] = multiply
operations["minus"] = function(a: int, b: int) -> int { return a - b }

for name in ["add", "times", "minus"] {
    print(name, operations[name](6, 3))
}
```

```
add 9
times 18
minus 3
```

## 15.5 Functions that make functions

A function can give back a function. The function it gives back can use the variables of the function that made it, even after that function has returned:

```
function scaler(factor: float) -> function(float) -> float {
    return function(x: float) -> float { return x * factor }
}

let to_millivolts = scaler(1000.0)
let halve = scaler(0.5)
print(to_millivolts(3.3), halve(10.0))
```

```
3300.0 5.0
```

`to_millivolts` and `halve` are two functions made by one `scaler`, each with its own `factor`. A function together with the variables it keeps is called a *closure*.

### 15.5.1 It keeps the variable, not a copy

A closure keeps the variable itself. If it changes the variable, the change lasts until it is called again:

```
function counter() -> function() -> int {
    var count = 0
    return function() -> int {
        count += 1
        return count
    }
}

let next_id = counter()
print(next_id(), next_id(), next_id())

let other = counter()
print(other())
```

```
1 2 3
1
```

Each call of `counter` makes a new `count`, so `other` starts again from 1.

> **Why.** This is what the code says when it is read: there is one `count`. If the inner function had a copy, `next_id()` would give 1 every time, and nothing would say why. (Decision 3.6c.)

### 15.5.2 Each turn of a loop has its own name

The name in a `for` loop is new on each turn, so a function made inside a loop keeps the value of that turn:

```
var handlers: list[function()] = []
for at in range(3) {
    handlers.append(function() { print("handler", at) })
}
for handle in handlers {
    handle()
}
```

```
handler 0
handler 1
handler 2
```

In JavaScript with `var`, and in Python, all three would print the last value, 2: a famous puzzle that Zeno does not have.

### 15.5.3 What it can see is fixed where it is written

A function can see the variables declared above it, in the blocks around it, and never one declared further down, even one with the same name:

```
let level = 1
function outer() -> int {
    let read = function() -> int { return level }
    let level = 2
    return read()
}
print(outer())
```

```
1
```

`read` was written where the only `level` in sight was the one at the top, so that is the one it uses, and `outer` gives 1.

> **Why.** The meaning of a name should be settled by where it is written, not by what happens to be declared later. This rule, called *lexical scope*, is the one nearly every modern language follows, and it lets the compiler decide once which variables a function must take with it. (Decisions 3.6c and 3.6d.)

## 15.6 Calling something that is not a function

Only a function can be called, and the checker knows which names hold one:

```
let speed = 5
```

```
speed()
```

```
error: int cannot be called
 --> main.zeno:2:1
  |
2 | speed()
  | ^

1 error found. Nothing was run.
```

### What this chapter said

A function is a value: its name without brackets is the function. Its type is written `function(int, str) -> bool`. A function with no name is written `function(n) { return n * 2 }`. Functions can be passed, stored in lists and dicts, and given back by other functions. A function given back keeps the variables around it, the variables themselves and not copies, as they were in sight where it was written.

### Things to try

1. Write a function `apply_all` that takes `fs: list[function(int) -> int]` and `x: int`, and gives each function's answer for `x` in a `list[int]`.
2. Write a function `compose` that takes two functions of type `function(int) -> int`, `f` and `g`, and gives a function of that type which does `g` and then `f`.
3. Make a counter that starts from a number and counts up by a step, both given to the function that makes it.
4. Build a small calculator: read lines such as `add 2 3` with `input()`, split them, and look the word up in a dict of functions. What should happen with a word that is not in the dict?

PART III

# Growing Programs

CHAPTER 16

# Types, Written and Worked Out

Every value in Zeno has a *type*: `3` is an `int`, `"hi"` is a `str`, `[1, 2]` is a `list[int]`. Types can be written in a program or left out. This chapter says what happens in each case, what Zeno works out by itself, what it does when it cannot, and the strict mode that a compiled program needs.

## 16.1 The types there are

| Type | Holds |
|---|---|
| `int` | a whole number (Numbers and Operators) |
| `float` | a number with a point |
| `bool` | `true` or `false` |
| `str` | text (Text) |
| `list[T]` | items of type `T`, in order (Lists) |
| `dict[K, V]` | values of type `V` under keys of type `K` (Dicts) |
| `function(A, B) -> R` | a function (Functions as Values) |
| `T or none` | a `T`, or nothing (Values That May Be Missing) |
| a class or enum name | an object or a case (Classes, Enums) |
| an interface name | any class that is that interface (Interfaces) |
| `any` | a value of any type, checked as the program runs |

Later chapters add numbers with units such as `ms` and `V` (Numbers with Units), sized whole numbers such as `u8` (Sized Numbers and Bits), and the pins and registers of a board (The Board).

## 16.2 Writing a type

A type goes after a name, with a colon, on a variable, a parameter or a field, and after `->` for what a function gives back:

```
let limit: int = 100
var names: list[str] = []
function scale(value: float, by: float) -> float {
    return value * by
}
class Reading {
    pin: int
    volts: float
}
print(limit, names, scale(2.0, 1.5), Reading(4, 3.3))
```

```
100 [] 3.0 Reading(pin=4, volts=3.3)
```

A written type is a promise. Zeno checks every use against it before the program runs, and never changes it to fit:

```
function scale(value: float, by: float) -> float {
    return value * by
}
print(scale(2, 3))
```

```
error: `value` of `scale` should be float, but this is int. Write a number with a
    ↪ decimal point, as in `1.0`, and convert a value with `float( )`
 --> main.zeno:4:13
  |
4 | print(scale(2, 3))
  |             ^

error: `by` of `scale` should be float, but this is int. Write a number with a decimal
    ↪ point, as in `1.0`, and convert a value with `float( )`
 --> main.zeno:4:16
  |
4 | print(scale(2, 3))
  |                ^

2 errors found. Nothing was run.
```

> **Why.** The name is what a reader scans for, so it comes first, and since the type can usually be left out, the part that may be missing goes last. (Decision 1.11.)

## 16.3 What Zeno works out

Where a type is left out, Zeno works it out from the program, wherever it can be known for certain.

**A variable** has the type of the value it starts with. `let n = 3` is an `int`, and stays one:

```
var count = 3
count = "three"
```

```
error: the new value should be int, but this is str
 --> main.zeno:2:9
  |
2 | count = "three"
  |         ^

1 error found. Nothing was run.
```

**What a function gives back** is worked out from its `returns`, in recursive functions too.

**A parameter** is worked out from the calls. If every call passes ints, the parameter is an `int`, and the body is checked with that knowledge:

```
function add(a, b) {
    return a + b
}
```

```
let total: str = add(1, 2)
```

```
error: `total` should be str, but this is int
 --> main.zeno:4:18
  |
4 | let total: str = add(1, 2)
  |                  ^

1 error found. Nothing was run.
```

**A field** is worked out from every value stored in it:

```
class Reading {
    pin
    volts
}
let r = Reading(3, 2.5)
let label: str = r.volts
```

```
error: `label` should be str, but this is float
 --> main.zeno:6:18
  |
6 | let label: str = r.volts
  |                  ^

1 error found. Nothing was run.
```

> **Why.** Types pay for themselves in large programs and in firmware, where a mistake found on the bench costs hours. They get in the way of a ten-line script. With types that can be left out, one language serves both, and a program can start loose and be tightened a line at a time as it grows, instead of being written again in another language. This is called *gradual typing*; TypeScript and Python's type hints took the same road. (Decision 0.2.)

### 16.3.1 Where the mistake is reported

When a parameter's type was worked out, a call that disagrees is reported inside the function, where the value is misused, with a note that names the call the type came from:

```
function shout(text) {
    return text.upper()
}
print(shout(5))
```

```
error: int has no field or method `upper`
 --> main.zeno:2:12
  |
2 |     return text.upper()
  |            ^
  = note: `text` was worked out to be int from line 4. Give `text` a type to have each
    ↪ call checked against it instead

1 error found. Nothing was run.
```

Zeno cannot know which line is wrong, the function or the call, so it shows both. Writing the type settles it, and then the call is the mistake. That is the reason to write types on anything

other people call.

## 16.4 When it cannot be worked out

Some things have no one type. A function called once with an int and once with text, or a list that holds a number and a word:

```
function same(x) {
    return x
}
let mixed = ["volts", 3.3, true]
print(same(1), same("one"), mixed)
```

```
1 one ["volts", 3.3, true]
```

Such a value is *dynamic*: it carries its type with it, and is checked while the program runs instead of before. Zeno never guesses a type; when two uses disagree, it leaves the value dynamic. The type `any` says so on purpose:

```
let a: any = 3
print(a + 1)
let b: any = "three"
print(b + 1)
```

```
4
error: `+` does not work between a str and an int
 --> main.zeno:4:7
  |
4 | print(b + 1)
  |       ^
```

Here the first `print` runs, and the mistake in the second is found only when the program reaches it. That is the price of a dynamic value: a mistake that a type would have caught before the program started is caught later, or on a path that no test happened to take, never.

## 16.5 Strict mode

A dynamic value needs a tag that says what it is, and usually memory from the heap. A small microcontroller cannot afford either, so a compiled program must know every type before it runs. *Strict mode* is that rule: a value that would be dynamic is an error.

Strict mode is asked for with `--strict` on the command line, or by the word `strict` alone on the first line of the file. `zeno build` always works in strict mode.

```
function same(x) {
    return x
}
print(same(1), same("one"))
```

```
$ zeno check --strict main.zeno
error: strict mode: the parameter `x` of `same` needs a type. It could not be worked out
    ↪  from how it is used
 --> main.zeno:1:1
  |
1 | function same(x) {
```

```
  | ^

1 error found. Nothing was run.
```

```
strict
let mixed: list[any] = ["volts", 3.3]
```

```
error: `any` is not allowed in strict mode
 --> main.zeno:2:1
  |
2 | let mixed: list[any] = ["volts", 3.3]
  | ^

1 error found. Nothing was run.
```

Most programs pass strict mode with few types written, because Zeno works out so much. A program written with no types at all, whose functions are each called with one kind of value, is strict already.

> **Why.** Whether a file may hold dynamic values is a fact about that file: firmware must not, a script may. A flag on the command line lives in a build script, where it can be forgotten; a first line travels with the code and is seen by everyone who reads it. It is a bare word, with no `#` or `@`, because it is part of the language and not a comment. There is no way to turn strict mode off from inside a file: the stricter of the two always wins. (Decision 0.2.)

## 16.6 No quiet conversions

Zeno does not change a value from one type to another behind the program's back. An `int` is not a `float`, though the two mix in arithmetic (Numbers and Operators), and nothing becomes a `bool` or a `str` by itself:

```
let whole = 3
let real: float = whole
```

```
error: `real` should be float, but this is int. Write a number with a decimal point, as
    ↪ in `1.0`, and convert a value with `float( )`
 --> main.zeno:2:19
  |
2 | let real: float = whole
  |                   ^

1 error found. Nothing was run.
```

```
let whole = 3
let real: float = float(whole)
print(real, str(whole) + " items", int("42") + 1)
```

```
3.0 3 items 43
```

The functions `int`, `float` and `str` make the change, where it can be seen.

## 16.7 Only none can be added to a type

A type can be widened with `or none`, and with nothing else. A value that is *either an int or a str* is written as an enum whose cases carry each:

```
let v: int or str = 3
```

```
error: only `or none` can follow a type for now
 --> main.zeno:1:15
  |
1 | let v: int or str = 3
  |               ^
```

```
enum Setting {
    Number(n: int)
    Word(text: str)
}
let settings = [Number(3), Word("auto")]
print(settings)
```

```
[Number(3), Word("auto")]
```

An enum names the choices, and a `match` must then handle each one, which a bare `int or str` would not ask for.

### What this chapter said

Types go after names, `name: type`, and after `->`. A written type is checked and never changed. Zeno works out the types of variables, returns, parameters and fields wherever they are certain, and leaves a value dynamic where they are not; `any` asks for that on purpose. Strict mode, from `--strict` or a first line `strict`, refuses dynamic values, and every compiled program is strict. Nothing changes type by itself.

### Things to try

1. Take a program from an earlier chapter, delete every type in it, and run `zeno check --strict`. Which types does it ask for back?
2. Write a function with no types and call it with an int. Then call it with a float too. What does strict mode say, and why?
3. Give a field a written type, store a value of another type in it, and read the message.
4. Rewrite a `list[any]` of numbers and words as a list of an enum, and print each item with a `match`.

CHAPTER 17

# Numbers with Units

A number in firmware is almost always a measurement of something: a delay in milliseconds, a supply in volts, a speed in revolutions per minute. In most languages the unit lives only in a comment or in the programmer's head, and mixing two of them up is a classic bug: a delay of 500 meant as milliseconds and read as microseconds, or millivolts added to volts. In Zeno a number can carry its unit, and the unit is checked.

## 17.1 Writing a quantity

A number followed by a unit is a *quantity*:

```
let wait = 500 ms
let supply = 3.3 V
let speed = 1200 rpm
print(wait, supply, speed)
```

```
500 ms 3.3 V 1200 rpm
```

The units come from a fixed list:

| Measures | Units |
|---|---|
| time | `ns us ms s min h` |
| frequency | `Hz kHz MHz` |
| voltage | `uV mV V kV` |
| current | `uA mA A` |
| resistance | `ohm kohm Mohm` |
| power | `mW W kW` |
| energy | `J Wh kWh` |
| temperature | `degC`, and `K` apart from it |
| length | `mm cm m km` |
| mass | `mg g kg` |
| angle | `deg`, and `rad` apart from it |
| rotation | `rpm` |

A word that is not on the list is refused, with some that are:

```
let wait = 5 sec
```

```
error: `sec` is not a unit Zeno knows. Some that it does: s, ms, us, Hz, V, mV, A, mA, W,
    ↪ degC
 --> main.zeno:1:12
```

```
  |
1 | let wait = 5 sec
  |            ^

1 error found. Nothing was run.
```

**Why.** If any word after a number were a unit, a typing slip would quietly make a new one, and the bug that units are there to stop would be back. (Decision 1.15.)

## 17.2 What a quantity can do

Quantities of the same unit add, subtract and compare as numbers do. A quantity times or divided by a plain number keeps its unit, and one quantity divided by another of the same kind is a plain number, how many times the one goes into the other:

```
let wait = 500 ms
print(wait + 250 ms, wait * 2, wait / 4, wait > 100 ms)
print(wait / 1 ms, 3.3 V / 1 mV)
```

```
750 ms 1000 ms 125 ms true
500.0 3300.0
```

Dividing by one of a unit, as in `wait / 1 ms`, is how a program takes the unit off and keeps the bare number, saying which unit the number is in.

Quantities of different kinds never meet. Volts do not add to seconds, and a bare number does not add to either:

```
let total = 1 s + 3 V
```

```
error: `+` does not work between a quantity in s and a quantity in V
 --> main.zeno:1:13
  |
1 | let total = 1 s + 3 V
  |             ^

1 error found. Nothing was run.
```

```
let supply = 3.3 V
print(supply + 2)
```

```
error: `+` does not work between a quantity in V and int
 --> main.zeno:2:7
  |
2 | print(supply + 2)
  |       ^

1 error found. Nothing was run.
```

## 17.3 Units of one kind work together

Units that measure the same thing combine, and the answer is in the finer of the two:

```
print(1 s + 500 ms, 1 s == 1000 ms, 1 h / 1 min, 1 kWh == 3600000 J)
```

```
1500 ms true 60.0 true
```

**Why.** Going from seconds to milliseconds multiplies by a whole number, which loses nothing. Going the other way divides, and a millisecond is not exactly a number of seconds in binary. So the direction that is always exact is the one taken without being asked. (Decision 1.15.)

### 17.3.1 Changing unit

A quantity is given in another unit of its kind with a dot and the unit's name. It reads as "delay, in seconds":

```
let delay = 1500 ms
print(delay.s, (3 s).ms, (3300 mV).V)
```

```
1.5 s 3000 ms 3.3 V
```

## 17.4 A unit is a type

The name of a unit is also a type. A parameter of type `ms` takes a quantity in milliseconds, and nothing else:

```
function pause(length: ms) {
    print("pausing for", length)
}
pause(250 ms)
pause((2 s).ms)
```

```
pausing for 250 ms
pausing for 2000 ms
```

```
function pause(length: ms) {
    print("pausing for", length)
}
pause(2 s)
pause(250)
```

```
error: `length` of `pause` should be a quantity in ms, but this is a quantity in s.
    ↪ Convert it with `.ms`
 --> main.zeno:4:7
  |
4 | pause(2 s)
  |       ^

error: `length` of `pause` should be a quantity in ms, but this is int
 --> main.zeno:5:7
  |
5 | pause(250)
  |       ^

2 errors found. Nothing was run.
```

Passing `2 s` where `ms` is wanted is refused, and the message says to write `.ms`.

> **Why.** Converting by itself at a call was tried on paper and turned down. `zeno run` would show `2 s` where a compiled program, which knows that the parameter is `ms`, would show `2000 ms`: one program, two outputs. Writing `.ms` means the same in both. (Decision 1.15.)

## 17.5 Time

The `time` module works in quantities. `sleep` takes a time with its unit, and `now()` gives the time in `ms` since a fixed moment:

```
from time import sleep, now

let start = now()
sleep(20 ms)
let took = now() - start
print(took >= 20 ms)
```

```
true
```

`sleep(500)` is refused. Whether a bare 500 is seconds or milliseconds is the question every delay function in every C library answers differently, and here the program text answers it.

`now()` counts from a moment the program cannot know, such as the moment a board was switched on, so only the difference of two readings means anything.

## 17.6 What does not convert

Degrees Celsius and kelvin differ by an offset, not by a scale, so adding one to the other means nothing, and Zeno refuses it:

```
print(20 degC + 1 K)
```

```
error: `+` does not work between a quantity in degC and a quantity in K
 --> main.zeno:1:7
  |
1 | print(20 degC + 1 K)
  |       ^

1 error found. Nothing was run.
```

Degrees and radians of angle have no exact factor between them, so they too never meet by themselves. A program changes one to the other when it asks, with `.rad` and `.deg`:

```
let half_turn = 180.0 deg
print(half_turn.rad, (half_turn.rad).deg)
```

```
3.141592653589793 rad 180 deg
```

The angle functions of the `math` module take and give angles with their units (Mathematics).

## 17.7 Showing a quantity

A quantity is shown with its unit after a space. A format applies to the number, and the unit stays:

```
let supply = 3.3 V
let angle = 30.456 deg
print(f"[{supply:8.2f}] [{angle:.1f}]")
```

```
[  3.30 V] [30.5 deg]
```

The width counts the unit too, so a column of quantities lines up on the right, as numbers do.

## 17.8 What units leave out

In version 0.1 there are no derived units: `V * A` is not `W`, and quantities of two kinds cannot be multiplied. A program that needs power works it out from the bare numbers and puts the unit back:

```
let volts = 12.0 V
let amps = 2.0 A
let power = (volts / 1 V) * (amps / 1 A) * 1 W
print(power)
```

```
24 W
```

There are no units of a program's own either, and `sum`, `min` and `max` take plain numbers. Both are on the list in Appendix G.

### What this chapter said

A number followed by a unit from Zeno's list is a quantity. Quantities of one kind combine, in the finer unit, and never mix with another kind or with bare numbers. `.ms` and its like change unit; dividing by `1 ms` gives the bare number. A unit's name is a type. `sleep` and `now` work in quantities. A format applies to the number, and the unit is always shown.

### Things to try

1. Write `function blink_period(rate: Hz) -> ms`. What is the bare number you have to work with, and where does the unit go back?
2. Add `250 us` to `1 ms` and print the answer. Then print it in microseconds and in seconds.
3. Try `sleep(3 V)` and read the message.
4. Write a voltage divider: given a supply and two resistances in `kohm`, give the output voltage as a quantity in `V`.

CHAPTER 18

# Sized Numbers and Bits

Firmware talks to hardware, and hardware works in bits: a status register where each bit is a flag, a byte in a frame from a sensor, a 16-bit value split over two registers. This chapter is about the operators that work on bits, and the six whole-number types that say how many bits a number fits in.

## 18.1 Writing bits

A whole number can be written in hexadecimal after `0x`, in binary after `0b` and in octal after `0o`, and _ can split the digits into groups. A format shows a number the same ways (Text):

```
let status = 0b1010_0101
print(status, 0xA5, 0o245)
print(f"{status:08b} {status:02X} {status:x}")
```

```
165 165 165
10100101 A5 a5
```

## 18.2 The bit operators

| Operator | Means | Example |
|---|---|---|
| `&` | and: a bit set in both | `0b1100 & 0b1010` is `0b1000` |
| `\|` | or: a bit set in either | `0b1100 \| 0b1010` is `0b1110` |
| `^` | exclusive or: set in one, not both | `0b1100 ^ 0b1010` is `0b0110` |
| `~` | every bit turned over | `~0` is `-1` |
| `<<` | shift left: times two, `n` times | `1 << 4` is `16` |
| `>>` | shift right: halve, rounding down | `0xF0 >> 4` is `15` |

Each has an = form: `&=`, `|=`, `^=`, `<<=`, `>>=`. They are the everyday tools of a register:

```
let READY = 1 << 0
let ERROR = 1 << 3

var flags = 0
flags |= READY                  # set a bit
flags |= ERROR
print(f"{flags:08b}")
flags &= ~ERROR                 # clear a bit
print(f"{flags:08b}", flags & READY != 0)
flags ^= READY                  # turn a bit over
print(f"{flags:08b}")
```

```
let reading = 0x3A7
print((reading >> 8) & 0xFF, reading & 0xFF)  # high byte, low byte
```

```
00001001
00000001 true
00000000
3 167
```

An `int` is 64 bits in two's complement, so `~x` is `-x - 1` and `-1 & 0xFF` is `255`.

> **Why.** Zeno spells `and`, `or` and `not` as words, and still uses symbols for bits. `status & MASK` is how every datasheet, every C header and every Python script writes it, and a firmware engineer reads it faster than any word. The words left `&` and `|` free for exactly this. (Decision 2.7.)

The C slip, `&` on two `bools`, is caught, and so is `&&`:

```
let ready = true
let safe = false
print(ready & safe)
```

```
error: `&` works on the bits of whole numbers, and these are bool and bool. For two
    ↪ bools Zeno spells this `and`
 --> main.zeno:3:7
  |
3 | print(ready & safe)
  |       ^

1 error found. Nothing was run.
```

### 18.2.1 Bits come before comparisons

`flags & MASK == 0` means `(flags & MASK) == 0`, as it looks. The bit operators bind tighter than the comparisons, as in Python:

```
let flags = 0x81
print(flags & 0x80 == 0x80, flags & 0x02 == 0)
```

```
true true
```

> **Why.** In C, `flags & MASK == 0` means `flags & (MASK == 0)`, which is almost never what was meant and compiles without a word. C's own authors called it a mistake that was too late to fix. (Decision 2.7a.)

### 18.2.2 Shifts are checked

A shift left that would push set bits off the top is an `OverflowError`, as `*` would be, and a shift by a count outside 0 to 63 is a `ValueError`:

```
print(1 << 64)
```

```
error: uncaught ValueError: cannot shift by 64. An int has 64 bits, so a shift is 0 to
    ↪ 63
 --> main.zeno:1:7
```

```
  |
1 | print(1 << 64)
  |       ^
```

In C, a shift by 64 or by a negative count is undefined, and `1 << 40` on a 32-bit int is a wrong number. Each is a quiet wrong answer. To throw away the top bits on purpose, say so with `wrap`, below. (Decision 2.7b.)

## 18.3 Sized whole numbers

An `int` is 64 bits. Six more types say that a whole number fits in fewer:

| Type | Bits | Holds |
|---|---|---|
| `u8` | 8 | 0 to 255 |
| `u16` | 16 | 0 to 65 535 |
| `u32` | 32 | 0 to 4 294 967 295 |
| `i8` | 8 | −128 to 127 |
| `i16` | 16 | −32 768 to 32 767 |
| `i32` | 32 | −2 147 483 648 to 2 147 483 647 |

They are for where the size is real: a byte in a frame, a field of an object on a board with 2 KB of memory, a register. With the size in the type, the checker knows that a `u8` is never handed 300, and a compiled program stores it in one byte.

```
let level: u8 = 200
let offset: i16 = -300
let frame: list[u8] = [0x02, 0x41, 0xFF]
print(level, offset, frame)
```

```
200 -300 [2, 65, 255]
```

A number written out takes the size of where it goes, and one that does not fit is refused before the program runs:

```
let level: u8 = 300
```

```
error: 300 does not fit in a u8 (0 to 255)
 --> main.zeno:1:17
  |
1 | let level: u8 = 300
  |                 ^

1 error found. Nothing was run.
```

> **Why.** The names are Rust's, and close to C's `uint8_t` with the noise taken off. `byte` and `short` say less: how long is a `short`? (Decision 2.8.)

### 18.3.1 Arithmetic gives an int

Sums, products, shifts and the rest on sized numbers give an `int`, so the middle of a calculation never overflows at 255:

```
let a: u8 = 200
let b: u8 = 100
print(a + b, (a + b) // 2)
```

```
300 150
```

The one place a number may not fit is where it is stored, and there the program says what it wants.

> **Why.** Wrapping at the size, as C does, makes `200 + 100` into 44, quietly. Checking at the size, as Rust does when debugging, makes `(a + b) // 2` fail for two numbers whose average fits. Refusing to mix sizes needs a conversion on every line, and people stop reading them. Working in ints and checking where the answer is stored is C's integer promotion without the silent cut at the end. (Decision 2.8a.)

## 18.4 Putting a number back: check or wrap

There are two ways to turn an `int` into a sized number, and they must not look alike:

- `u8(n)` is `n` if it fits, and an `OverflowError` if it does not.
- `u8.wrap(n)` keeps the low 8 bits and never fails: `u8.wrap(300)` is 44, and `i8.wrap(200)` is $-56$.

```
let a: u8 = 200
let b: u8 = 100
print(u8.wrap(a + b), i8.wrap(200), u16(a + b))
```

```
44 -56 300
```

A checksum is *meant* to lose its top bits, so it wraps:

```
function checksum(data: list[u8]) -> u8 {
    var total = 0
    for byte in data {
        total += byte
    }
    return u8.wrap(total)
}
print(checksum([0xFF, 0x02, 0x10]))
```

```
17
```

A length that does not fit in its field is a bug, so it is checked:

```
let length = 300
let field = u8(length)
```

```
error: uncaught OverflowError: 300 does not fit in a u8 (0 to 255)
 --> main.zeno:2:13
  |
2 | let field = u8(length)
  |             ^
```

> **Why.** In C both are written `(uint8_t)x`, and the reader has to guess which was meant. Here the short spelling is the safe one, and wrapping is a word that a reviewer can search for. (Decision 2.8b.)

### 18.4.1 count += 1

`count += 1` on a sized number means `count = u8(count + 1)`: the sum is an int, and it is checked where it is stored. A counter that passes 255 without anybody deciding it should is a bug:

```
var count: u8 = 254
count += 1
print(count)
count += 1
```

```
255
error: uncaught OverflowError: 256 does not fit in a u8 (0 to 255)
 --> main.zeno:4:1
  |
4 | count += 1
  | ^
```

A counter that is meant to go round says so: `count = u8.wrap(count + 1)`. (Decision 10.27.)

## 18.5 What goes where by itself

- A sized number goes wherever an `int` is wanted, with nothing written: every `u8` is an `int`.
- An `int` does not go where a `u8` is wanted, and a `u8` does not go where a `u16` is wanted. Each needs `u8(...)` or `u16(...)`.

```
let reading = 5
let level: u8 = reading
```

```
error: `level` should be u8, but this is int. Convert it with `u8(...)`, which checks
    ↪ that it fits
 --> main.zeno:2:17
  |
2 | let level: u8 = reading
  |                 ^

1 error found. Nothing was run.
```

The first cannot lose anything, so asking for it would be noise. The second can, so it is written. A `list[u8]` is not a `list[int]`, because lists are shared: whoever held it as a `list[int]` could append 300. Copy it, `[int(b) for b in frame]`.

There is no `u64`. An `int` is 64 bits with a sign, so half the values of a `u64` would not fit in one, and "arithmetic gives an `int`" would stop being true. (Decision 2.8d.)

## 18.6 A worked example: a frame from a sensor

A sensor sends a frame of four bytes: a start byte `0x02`, a 16-bit reading high byte first, and a checksum of the first three. This program builds one and reads it back:

frame.zeno

```
function checksum(data: list[u8]) -> u8 {
    var total = 0
    for byte in data {
        total += byte
    }
    return u8.wrap(total)
}

function make_frame(reading: u16) -> list[u8] {
    let frame: list[u8] = [0x02, u8(reading >> 8), u8(reading & 0xFF)]
    frame.append(checksum(frame))
    return frame
}

function read_frame(frame: list[u8]) -> u16 raises ValueError {
    if frame[0] != 0x02 {
        raise ValueError("a frame starts with 0x02")
    }
    if checksum(frame.slice(0, 3)) != frame[3] {
        raise ValueError("the checksum does not match")
    }
    return u16(frame[1] << 8 | frame[2])
}

let frame = make_frame(931)
print([f"{b:02X}" for b in frame])
print(read_frame(frame))

frame[2] = 0x00
try {
    read_frame(frame)
} except ValueError as e {
    print("refused:", e.message)
}
```

```
["02", "03", "A3", "A8"]
931
refused: the checksum does not match
```

### What this chapter said

Whole numbers can be written in `0x`, `0b` and `0o`. `&`, `|`, `^`, `~`, `<<` and `>>` work on bits, bind tighter than comparisons, and shifts are checked. `u8`, `u16`, `u32`, `i8`, `i16` and `i32` say how many bits a number fits in. Arithmetic on them gives an `int`; `u8(n)` puts it back with a check, and `u8.wrap(n)` by cutting. A sized number is an `int` for free; the other way is written.

### Things to try

1. Write `function bit_is_set(value: int, bit: int) -> bool` and `function set_bit(value: int, bit: int) -> int`.
2. Split a `u32` into four `u8`s, high byte first, and join them again.
3. Count the set bits of a number with a loop, `&` and `>>`.
4. Change the checksum in the frame example to an exclusive or of the bytes. Does it

> still need `wrap`? Why not?

CHAPTER 19

# Generics

A stack of numbers and a stack of names are the same code. A function that gives the first item of a list does the same whatever the items are. *Generics* let such code be written once, with the type left open, and still be checked for every type it is used with.

## 19.1 A generic function

A function names the type it leaves open in square brackets after its name, and uses that name where a type would go:

```
function first[T](items: list[T]) -> T {
    return items[0]
}

print(first([3, 1, 2]), first(["ada", "alan"]))
let n = first([10, 20])
print(n + 1)
```

```
3 ada
11
```

`T` is a *type parameter*. Read the heading as "first, for any type T, takes a list of T and gives a T". At each call Zeno works out what `T` is, so `first([10, 20])` is known to be an `int`, and `n + 1` is checked as an int sum.

Without generics, `first` would be written once for each type, or once with no types at all, and then nothing that used its answer could be checked.

> **Why.** Zeno already writes "a type applied to a type" as `list[int]`. `first[T]` is the same sentence, so there is one rule and no new punctuation. The angle brackets of C++, Java and Rust would make `a < b > c` a puzzle for the parser and the reader; Go chose square brackets for that reason. (Decision 3.16.)

A type parameter starts with a capital letter, so a reader can tell it is a type without looking back at the heading:

```
function first[t](items: list[t]) -> t {
    return items[0]
}
```

```
error: a type parameter starts with a capital letter, as in `T` or `Item`, and this is `
    ↪ t`
 --> main.zeno:1:1
  |
1 | function first[t](items: list[t]) -> t {
  | ^
```

```
1 error found. Nothing was run.
```

## 19.2 Several type parameters

A function can leave several types open. Here `convert` takes a list of one type and a function that turns it into another:

```
function convert[T, U](items: list[T], change: function(T) -> U) -> list[U] {
    var out: list[U] = []
    for item in items {
        out.append(change(item))
    }
    return out
}

let labels = convert([1, 2, 3], function(n) { return f"#{n}" })
print(labels, labels[0].upper())
```

```
["#1", "#2", "#3"] #1
```

Zeno works out `T` from the list, then calls the function in its head with an `int` to find what it gives back, and that is `U`.

## 19.3 A generic class

A class can leave types open the same way. `Stack[T]` is a stack of anything, and `Stack[str]` is a stack of text:

```
class Stack[T] {
    items: list[T] = []

    function push(self, item: T) {
        self.items.append(item)
    }

    function pop(self) -> T {
        return self.items.pop()
    }

    function is_empty(self) -> bool {
        return len(self.items) == 0
    }
}

let names: Stack[str] = Stack()
names.push("ada")
names.push("grace")
print(names.pop().upper(), names.is_empty(), names)
```

```
GRACE false Stack(items=["ada"])
```

```
class Pair[A, B] {
    left: A
    right: B
```

```
}

let p = Pair(36, "Ada")
print(p, p.left + 1, p.right.upper())
```

```
Pair(left=36, right="Ada") 37 ADA
```

Inside `Pair[A, B]`, `A` is the type parameter, even though `A` is also the unit of current: the name declared right there in the heading is the nearest, and the nearest name wins. A quantity inside it is still written `3 A`.

## 19.4 Where the types come from

A type argument is never written at the call. There is no `first[int](xs)`. Zeno finds each type parameter in this order:

1. from the arguments;
2. from an argument that is a function, by calling it in its head with what the others gave;
3. from the place the result goes, such as a variable with a written type.

The third is what a stack that starts empty needs: nothing in `Stack()` says what it will hold, so the variable says it, as `let names: list[str] = []` does for a list. If nothing says, strict mode asks for it, and shows how:

```
class Box[T] {
    items: list[T]
}
let box = Box([])
```

```
$ zeno check --strict main.zeno
error: `Box` cannot tell what `T` is from this call: no argument says. Put the result
    ↪ where its type is written, as in `let x: Box[int] = Box(...)`
 --> main.zeno:4:11
  |
4 | let box = Box([])
  |           ^

1 error found. Nothing was run.
```

A written number waits for the other arguments to decide. With `level` a `u8`, `same(level, 9)` is a `u8`, and `same(level, 300)` is an error, so a generic used for bytes never needs `u8(0)` where `0` is meant. (Decision 3.16c.)

## 19.5 The body is checked for each type

`largest` below uses `>`, and nothing in its heading says that `T` must have one:

```
function largest[T](items: list[T]) -> T {
    var best = items[0]
    for item in items {
        if item > best {
            best = item
        }
```

```
    }
    return best
}
print(largest([3, 9, 2]), largest(["pear", "fig", "apple"]))
```

```
9 pear
```

Zeno checks the body once for each type the program uses it with. For ints and for text it passes. For a class, which has no >, the mistake is reported inside `largest`, with a note that names the type and the line that asked for it:

```
class Point {
    x: int
}

function largest[T](items: list[T]) -> T {
    var best = items[0]
    for item in items {
        if item > best {
            best = item
        }
    }
    return best
}
print(largest([Point(1), Point(2)]))
```

```
error: `>` does not work between Point and Point
 --> main.zeno:8:12
  |
8 |         if item > best {
  |            ^
  = note: this is inside `largest[Point]`: `largest` with T = Point, which line 14 asks
    ↪ for

1 error found. Nothing was run.
```

**Why.** A bound, such as "`T` must be something with >", needs a name for "something with >". Checking each use instead is what C++ did for thirty years, and it still finds every mistake before the program runs, which is the promise that matters. The note under the error is there so that the message points back to the call. (Decision 3.16d.)

## 19.6 What a compiled program does

In a compiled program each use becomes its own copy: `largest` for ints and `largest` for text are two functions in the C, each as fast as if it had been written by hand. Nothing is looked up while the program runs. The cost is size: each type a generic is used with adds a copy to the program, which on a microcontroller is flash. That is why an untyped function is not made generic by itself: a program asks for copies by writing `[T]`.

## 19.7 What generics leave out

In version 0.1, these are refused, each with a message that says so: an enum with type parameters, a method with type parameters of its own, a generic function kept as a value or passed along,

and raising an object of a generic class.

```
function first[T](items: list[T]) -> T {
    return items[0]
}
let pick = first
```

```
error: `first` is generic. It can be called, as in `first(...)`, but it cannot be kept
    ↪ as a value or passed along yet
 --> main.zeno:4:12
  |
4 | let pick = first
  |            ^

1 error found. Nothing was run.
```

The commonest generic enum, "a `T` or nothing", is already there: `T or none`. (Decision 3.16e.)

### What this chapter said

`function name[T](...)` and `class Name[T]` leave a type open. Type parameters start with a capital, and are worked out at each call from the arguments, from a function argument, or from where the result goes, never written at the call. The body is checked for every type it is used with, and a compiled program has a copy for each.

### Things to try

1. Write `function last[T](items: list[T]) -> T or none`, which gives `none` for an empty list.
2. Write a generic `class Ring[T]` that holds at most `size` items and drops the oldest when a new one comes. Use it for a ring of `u8` and a ring of `V`.
3. Write a generic `count_if[T]` that takes `items: list[T]` and `test: function(T) -> bool`, and gives how many items pass the test.
4. Give `largest` a list of `bool` and read the message.

CHAPTER 20

# Interfaces

A function that adds up areas should work for circles, for squares, and for the shape somebody writes next year. A function that sends a message should work through a serial port, a radio, or a log kept in memory for testing. An *interface* says what all of them can do, and each class says that it is one.

## 20.1 Declaring an interface

An interface is a list of functions with no bodies: what can be asked, not how it is done.

```
interface Shape {
    function area(self) -> float
    function scale(self, by: float)
}

class Circle is Shape {
    radius: float

    function area(self) -> float {
        return 3.14159 * self.radius * self.radius
    }

    function scale(self, by: float) {
        self.radius = self.radius * by
    }
}

class Square is Shape {
    side: float

    function area(self) -> float {
        return self.side * self.side
    }

    function scale(self, by: float) {
        self.side = self.side * by
    }
}

function total(shapes: list[Shape]) -> float {
    var sum = 0.0
    for shape in shapes {
        sum += shape.area()
    }
    return sum
}
```

```
let shapes: list[Shape] = [Circle(1.0), Square(2.0)]
print(total(shapes))
for shape in shapes {
    shape.scale(2.0)
}
print(total(shapes), shapes)
```

```
7.14159
28.56636 [Circle(radius=2.0), Square(side=4.0)]
```

`total` was written once. `shape.area()` runs the `area` of whatever class the object is, and that is decided while the program runs. An object seen through an interface still prints as the class it is.

## 20.2 The rules

**An interface lists functions,** each with `self` first and a type on every other parameter and on the result. It has no fields and no bodies.

**A class says which interfaces it is,** after `is` on its first line: `class Box is Named, Sized`. It is not one unless it says so.

**The class has every function of the interface, written the same way** : the same parameter names, the same types, the same result. It may have more of its own. A missing function, or one written differently, is reported at the class:

```
interface Shape {
    function area(self) -> float
    function scale(self, by: float)
}

class Dot is Shape {
    function area(self) -> float {
        return 0.0
    }
}
```

```
error: `Dot` says it is a `Shape`, and a `Shape` has `function scale(self, by: float)`.
    ↪ Give `Dot` that function
 --> main.zeno:6:1
  |
6 | class Dot is Shape {
  | ^

1 error found. Nothing was run.
```

```
interface Shape {
    function scale(self, by: float)
}

class Dot is Shape {
    function scale(self, factor: float) {
    }
}
```

```
error: `scale` of `Dot` is `function scale(self, factor: float)`, and `Shape` asks for `
    ↪ function scale(self, by: float)`. Write the two the same
 --> main.zeno:5:1
  |
5 | class Dot is Shape {
  | ^

1 error found. Nothing was run.
```

**Why.** In Go, and in Python's protocols, any class that happens to have an `area` and a `scale` is a `Shape` without a word. That saves a line and costs two things: a class whose `area` means something else is a `Shape` by accident, and a class meant to be one, with `scale` misspelt, is found out far away, at some call that passes it. With `is Shape` the promise is written where the reader of the class sees it, and it is checked there. (Decision 10.4.)

## 20.3 Where an interface goes

An object of a class goes wherever one of its interfaces is wanted: a variable, a parameter, a result, a field, and each of those with `or none`.

A list of the class does not go where a list of the interface is wanted, because whoever held it as a `list[Shape]` could then append a `Square` to a list of circles:

```
interface Shape {
    function area(self) -> float
}
class Circle is Shape {
    radius: float
    function area(self) -> float {
        return 3.0 * self.radius * self.radius
    }
}
function total(shapes: list[Shape]) -> float {
    var sum = 0.0
    for shape in shapes {
        sum += shape.area()
    }
    return sum
}
let circles = [Circle(1.0), Circle(2.0)]
print(total(circles))
```

```
error: `shapes` of `total` should be list[Shape], but this is list[Circle]. Whoever
    ↪ holds it as a list of Shape could append another class to it. Say what the list
    ↪ holds where it is made, as in `let items: list[Shape] = [...]`
  --> main.zeno:18:13
   |
18 | print(total(circles))
   |             ^

1 error found. Nothing was run.
```

The message says what to do: say what the list holds where it is made, by writing `list[Shape]` as the type of `circles`. A list written out in place, as in `total([Circle(1.0)])`, is a list of whatever the place wants. (Decision 10.7.)

## 20.4 Through an interface, only its functions

Through an interface, only the interface's functions can be used. Not the fields, not the class's other functions, and there is no way back from the interface to the class:

```
interface Shape {
    function area(self) -> float
}
class Circle is Shape {
    radius: float
    function area(self) -> float {
        return 3.0 * self.radius * self.radius
    }
}
let s: Shape = Circle(1.0)
print(s.radius)
```

```
error: `radius` is not a function of the interface `Shape`, which has `area`. What else
    ↪ this object has depends on its class, and that is only known when the program
    ↪ runs. Give `Shape` a function for it
  --> main.zeno:11:7
   |
11 | print(s.radius)
   |       ^

1 error found. Nothing was run.
```

What else an object has depends on its class, and that is only known when the program runs. If code needs it, the interface should have a function for it.

> **Why.** A way back, such as Java's casts or Rust's `downcast`, lets code ask "which class are you really?" and do something different for each, which is what the interface was meant to save. When the classes are a fixed list known in advance, an enum and `match` say that directly. (Decision 10.8.)

## 20.5 Errors through an interface

An interface function can say what it raises. A class may raise less than the interface says, never more, because whoever calls it through the interface would not know:

```
class Full {
    waiting: int
}

interface Port {
    function send(self, text: str) -> int raises Full
    function name(self) -> str
}

class Uart is Port {
    speed: int
    queued: int = 0

    function send(self, text: str) -> int raises Full {
        if self.queued >= 2 {
            raise Full(self.queued)
```

```
        }
        self.queued += 1
        return len(text)
    }

    function name(self) -> str {
        return f"uart at {self.speed}"
    }
}

class MemoryLog is Port {
    lines: list[str] = []

    function send(self, text: str) -> int {
        self.lines.append(text)
        return len(text)
    }

    function name(self) -> str {
        return "memory log"
    }
}

function report(port: Port, text: str) {
    try {
        print(port.name(), "sent", port.send(text))
    } except Full as f {
        print(port.name(), "is full with", f.waiting)
    }
}

let ports: list[Port] = [Uart(9600), MemoryLog()]
for turn in range(3) {
    for port in ports {
        report(port, "hello")
    }
}
```

```
uart at 9600 sent 5
memory log sent 5
uart at 9600 sent 5
memory log sent 5
uart at 9600 is full with 2
memory log sent 5
```

`MemoryLog` never fills up and raises nothing, and that is fine. A class whose `send` raised something `Port` does not list would be refused at the class.

## 20.6 No parent classes

There is no inheritance in Zeno: a class cannot be another class, only interfaces. What two classes share is declared as an interface that both are (Classes).

> **Why.** Inheriting fields and code brings the questions every language with it has had to answer: which `area` runs, what the object looks like in memory, what happens when two parents have the same field.

None of them is needed to write `total`. (Decision 10.4.)

## 20.7 What it costs

In a compiled program, each object carries a small number saying which class it is, and a call through an interface is a `switch` on that number to the class's function. There are no tables of function pointers, and nothing is looked up by name. (Decision 10.9.)

### What this chapter said

An interface lists functions with no bodies. A class says which interfaces it is with `class Name is Interface`, and must have each function written the same way. An object goes where its interface is wanted, a list of it only where the list was made as one. Through an interface only its functions are there. A class may raise less than its interface says, never more.

### Things to try

1. Add a `Triangle` to the shapes, and check that `total` needs no change.
2. Write `interface Sensor` with `function read(self) -> float` and two classes: a fake sensor that gives values from a list, and one that always gives the same value. Write a function that averages ten readings from any sensor.
3. Give `Shape` a function `name(self) -> str` and make every class say what it is.
4. Make `MemoryLog.send` raise a `ValueError` for an empty text, and read the message.

CHAPTER 21

# Programs in Several Files

A program of a few hundred lines is easier to read when it is cut into files: one for the sensors, one for the report, one for the serial port. In Zeno a file is a *module*, and `import` brings one file's names into another.

## 21.1 A module is a file

There is nothing to declare and nothing to register. The file `sensors.zeno` is the module `sensors`:

sensors.zeno

```
class Reading {
    name: str
    value: float
}

let highest = ["temperature": 60.0, "humidity": 100.0]

function check(reading: Reading) -> str {
    let limit = highest.get(reading.name)
    if limit is none {
        return "no limit is known"
    }
    if reading.value > limit {
        return f"over {limit}"
    }
    return "good"
}
```

main.zeno

```
import sensors

let readings = [
    sensors.Reading("temperature", 21.5),
    sensors.Reading("humidity", 104.0),
    sensors.Reading("wind", 12.0),
]
for reading in readings {
    print(reading.name, sensors.check(reading))
}
print(readings[0])
```

```
temperature good
humidity over 100.0
wind no limit is known
Reading(name="temperature", value=21.5)
```

`import sensors` reads `sensors.zeno` from the folder of the file that `zeno run` was given. Its names are then written with the module's name in front: `sensors.check`, `sensors.Reading`. A value prints by the name its own file gave it, `Reading`.

## 21.2 Two ways to import

| Written | Means |
|---|---|
| `import sensors` | read `sensors.zeno`; its names are written `sensors.check` |
| `import drivers.uart` | read `drivers/uart.zeno`; a dot is a folder, and the names are written `drivers.uart.send` |
| `from sensors import check, Reading` | read `sensors.zeno`, and let these names be written with nothing in front |

These are the same two forms that the modules which come with Zeno have: `import math` and `from time import sleep`. A file of your own is used exactly as one of those is.

drivers/uart.zeno

```
let speed = 9600
var sent = 0

function send(text: str) {
    print(f"uart> {text}")
    sent += len(text) + 1
}
```

station.zeno

```
import drivers.uart
from sensors import Reading, check

let r = Reading("temperature", 21.5)
drivers.uart.send(f"{r.name}: {check(r)}")
drivers.uart.send("done")
print(drivers.uart.sent, "bytes sent at", drivers.uart.speed)
```

```
uart> temperature: good
uart> done
23 bytes sent at 9600
```

A type from a module is written the same way, `sensors.Reading`, and so is an enum case in a `match`, `case shapes.Circle(r)`, and an error class in an `except`.

> **Why.** A file is already a unit that people name, open and send to each other, so it is the module. A separate declaration of what a module is, as in Java's `package` lines or Rust's `mod`, would be a second name for the same thing that could disagree with the first. (Decision 7.1.)

## 21.3 The rules

**What can be imported.** Everything a file declares at its top: functions, classes, enums and their cases, `let`, `var` and `const`. What the file itself imports is not passed on. A declaration marked

`private` is for its own file only, below.

**When an imported file runs.** Once, before the file that imports it, however many files import it. So imports go at the top of a file.

**Files cannot import each other,** directly or round a longer circle. One of them would have to run first, and none can:

a.zeno
```
import b
let x = 1
```

b.zeno
```
import a
let y = 2
```

```
error: `b.zeno` is the file the program starts in, and no file imports that one. Move
    ↪ what the files share into a file of its own, and import that
 --> a.zeno:1:1
  |
1 | import b
  | ^
```

What two files share goes into a third, which both import.

**Only a module changes its own variables.** Other files read them, and ask the module to change them through a function:

meddle.zeno
```
import drivers.uart
drivers.uart.sent = 0
```

```
error: `drivers.uart.sent` is a variable of the module `drivers.uart`, and only the file
    ↪  of `drivers.uart` gives it a new value. Give `drivers.uart` a function that
    ↪ changes it, and call that
 --> meddle.zeno:2:1
  |
2 | drivers.uart.sent = 0
  | ^
```

**No name has two meanings.** Importing `check` from two modules, or declaring your own `check` beside an imported one, is refused. A name declared inside a function hides an imported one, as it hides any name from the top of the file.

**A misspelt name is named,** with the names the module does have:

typo.zeno
```
import sensors
print(sensors.chek)
```

```
error: module `sensors` has no `chek`. It has `Reading`, `highest` and `check`
 --> typo.zeno:2:7
  |
```

```
2 | print(sensors.chek)
  |       ^
```

A mistake inside an imported file is shown in that file, with its name and line.

## 21.4 private

Every module has parts it is for and parts it only uses to get its work done. `private` in front of a declaration keeps it to the file it is written in. It goes in front of a function, class, enum, `let`, `var` or `const` at the top of a file, and in front of a field or a method of a class.

Here is a queue of a fixed size whose count must always agree with its slots:

ring.zeno

```
class Ring {
    private slots: list[int]
    private first: int
    private count: int

    function put(self, byte: int) -> bool {
        if self.count == len(self.slots) {
            return false
        }
        self.slots[self.slot(self.count)] = byte
        self.count += 1
        return true
    }

    function get(self) -> int or none {
        if self.count == 0 {
            return none
        }
        let byte = self.slots[self.first]
        self.first = wrapped(self.first + 1, len(self.slots))
        self.count -= 1
        return byte
    }

    function size(self) -> int {
        return self.count
    }

    private function slot(self, steps: int) -> int {
        return wrapped(self.first + steps, len(self.slots))
    }
}

function ring(size: int) -> Ring raises ValueError {
    if size < 1 {
        raise ValueError("a ring has room for one byte at least")
    }
    return Ring([0 for slot in range(size)], 0, 0)
}

private function wrapped(place: int, size: int) -> int {
    return place % size
}
```

main.zeno

```
import ring

let queue = ring.ring(3)
for byte in [10, 20, 30, 40] {
    print(byte, queue.put(byte))
}
print(queue.get(), queue.get(), queue.size())
```

```
10 true
20 true
30 true
40 false
10 20 1
```

Each of these, in another file, is refused:

bad1.zeno

```
import ring
let queue = ring.ring(3)
queue.count = 0
```

```
error: `count` is private in the class `ring.Ring`: only the file that declares the
    ↪ class can use it. To use it from another file, take the word `private` off where
    ↪ it is declared
 --> bad1.zeno:3:1
  |
3 | queue.count = 0
  | ^

1 error found. Nothing was run.
```

bad2.zeno

```
import ring
print(ring.wrapped(5, 4))
```

```
error: `wrapped` is private in the module `ring`: only its own file can use it. To use
    ↪ it from another file, take the word `private` off where it is declared
 --> bad2.zeno:2:7
  |
2 | print(ring.wrapped(5, 4))
  |       ^
```

bad3.zeno

```
import ring
let mine = ring.Ring([1, 2], 0, 2)
```

```
error: `ring.Ring` has the private field `slots`, so only the file that declares the
    ↪ class can build one. Write a function in that file that builds one and returns it,
    ↪  and call that function here
 --> bad3.zeno:2:12
  |
2 | let mine = ring.Ring([1, 2], 0, 2)
  |            ^

1 error found. Nothing was run.
```

The last is the important one. A class with a private field can be built only by its own file, because only that file can fill the field in. So the only way to get a `Ring` is through the function `ring`, which checks the size. Every line that changes the count is in `ring.zeno`, in front of whoever reads it.

> **Why.** One word, in front of the thing it is about, at the place a reader is already looking. Python's leading underscore is only a habit that nothing checks; Java's four levels of access are three more than a file needs. Private is for the file and not for the class because the file is what one person writes and reads at a time, and functions beside a class often need to see inside it. (Decisions 10.1 to 10.3.)

## 21.5 How it works

Before the checker sees anything, a stage of the compiler loads every file the program imports, renames each top-level name with its module in front, and joins the files into one program, in the order they must run. Everything after that, the checker, the interpreter and the compilers, sees a single program. `zeno join` shows it:

joined.zeno

```
import drivers.uart
drivers.uart.send("hi")
```

```
$ zeno join joined.zeno
let drivers.uart.speed = 9600
var drivers.uart.sent = 0
function drivers.uart.send(text: str)
    print(f"uart> {text}")
    drivers.uart.sent += (len(text) + 1)
drivers.uart.send("hi")
```

Inside Zeno has more on the stages of the compiler.

### What this chapter said

A file is a module. `import name` reads `name.zeno` from the program's folder, a dot is a folder, and `from name import a, b` brings single names. Imported files run once, first, and cannot import each other. Only a module changes its own variables. `private` keeps a declaration, a field or a method to its own file, and a class with a private field can only be built there.

### Things to try

1. Move the `Log` class from Classes into a file of its own and use it from `main.zeno`.
2. Make the `readings` field of `Log` private, and give it a function `log(limit: int) -> Log` that refuses a limit below 1.
3. Make two files import each other, and read the message. Then move what they share into a third.
4. Run `zeno join` on a program of three files, and find where each file's statements went.

CHAPTER 22

# Packages

Modules put one program into several files. A *package* lets a program use files that are not its own: a folder of modules that lives somewhere else, beside the program or in somebody else's git repository.

## 22.1 Starting a program with zeno new

`zeno new` makes a folder with a small program in it, laid out the way a program that grows will want:

```
$ zeno new weather
Made the program `weather` in weather/
  zeno.package   the name of the program, and the packages it needs
  main.zeno      the program
  greeting.zeno  a module, with a test
Run it with `zeno run weather/main.zeno`, and its tests with `zeno test weather`.
$ zeno run weather/main.zeno
Hello, world, from weather.
$ zeno test weather
weather/greeting.zeno
  ok      hello says who it greets

1 test: 1 passed, 0 failed.
```

`zeno new --package name` makes a package instead: a folder of modules for other programs to use, with no `main.zeno`.

## 22.2 The project file

A folder is a package when it holds a file called `zeno.package`. The one `zeno new` made is comments, which say what can be written, and then one line:

weather/zeno.package

```
package weather
```

There are two kinds of line, and that is all the file can say:

| Line | Means |
|---|---|
| `package weather` | the name of this package |
| `needs filters in "../filters"` | it needs the package in that folder, counted from this file's folder. Nothing is copied |
| `needs units from "address" at "v1.2.0"` | it needs the package in that git repository, at that tag, branch or commit |

A comment starts with #, as in a program. A program with no `zeno.package` is still a program; the file is needed from the day the program needs a package.

## 22.3 A package beside the program

Here is a package of filters for noisy readings, in a folder of its own:

filters/zeno.package

```
package filters
```

filters/filters.zeno

```
function mean(values: list[float]) -> float {
    return sum(values) / float(len(values))
}

function clamp(x: float, low: float, high: float) -> float {
    if x < low {
        return low
    }
    if x > high {
        return high
    }
    return x
}
```

filters/lowpass.zeno

```
from filters import clamp

class Lowpass {
    weight: float
    value: float

    function add(self, reading: float) -> float {
        self.value += self.weight * (reading - self.value)
        return self.value
    }
}

function start(weight: float, first: float) -> Lowpass {
    return Lowpass(clamp(weight, 0.0, 1.0), first)
}
```

And a program, in the folder beside it, that needs the package:

thermostat/zeno.package

```
package thermostat

needs filters in "../filters"
```

thermostat/main.zeno

```
import filters
import filters.lowpass

let readings = [18.2, 18.9, 19.6, 21.0, 22.4, 23.1]
let smooth = filters.lowpass.start(0.5, readings[0])
```

```
for r in readings {
    print(f"read {r:5.1f}  smoothed {smooth.add(r):5.1f}")
}
print(f"mean {filters.mean(readings):.2f}")
```

```
$ zeno run thermostat/main.zeno
read  18.2  smoothed  18.2
read  18.9  smoothed  18.5
read  19.6  smoothed  19.1
read  21.0  smoothed  20.0
read  22.4  smoothed  21.2
read  23.1  smoothed  22.2
mean 20.53
```

### 22.3.1 Which file an import names

The rule for imports has one step more than in the last chapter:

1. If the first word of the import is a package that this file's `zeno.package` needs, the rest is a file in that package's folder. `import filters.lowpass` is `lowpass.zeno` there, and `import filters` alone is the file with the package's own name, `filters.zeno`.
2. Otherwise it is a file of the package the importing file is in.

Inside the package the same rule holds from the package's own side: `lowpass.zeno` writes `from filters import clamp` to get its neighbour. A package sees its own files and the packages its own `zeno.package` needs, and nothing of the program that uses it, so the same package works in every program, and can be run and tested on its own.

## 22.4 Other people's packages: zeno fetch

A package that somebody else keeps is named by its git repository and by one exact version of it:

```
package logger

needs units from "https://github.com/someone/zeno-units" at "v1.2.0"
```

`at` is a tag, a branch or a commit. There are no ranges such as `^1.2`: the version written is the version used. Then, in the program's folder:

```
$ zeno run main.zeno
error: the package `units` has not been fetched: there is no file
`packages/units/zeno.package`. Run `zeno fetch` in the folder of the program

$ zeno fetch
Fetched units v1.2.0 (1c4e326) into packages/units
Fetched prefix main (d4461d9) into packages/prefix
Wrote zeno.lock

$ zeno run main.zeno
42.0
```

Three things happened.

- **Running never fetches.** `zeno run`, `check` and `build` read files and nothing else. Only `zeno fetch` uses the network, with the `git` that is on the machine.
- **What a package needs comes too.** `units` needs `prefix`, so that was fetched as well. All of them go into one folder, `packages/`, beside the program's `zeno.package`, and a program has one package of each name.
- **zeno.lock was written.** It holds the exact commit each package was at. A branch moves, but the next `zeno fetch`, here or on another machine, gets the commit in the lock. `zeno fetch --update` looks every version up again and writes the lock afresh.

Keep `zeno.package` and `zeno.lock` with the program. The folder `packages/` can be made again from them at any time.

**Why.** A range such as `^1.2` means the program built today and the program built next month may hold different code, chosen by somebody else's release. Firmware that passed its tests on the bench must build the same a year later. One exact version and a lock file give that, and there is no solver to understand. Fetching only when asked means a build never depends on the network, which a factory build machine may not have. (Decisions 8.4, 8.9 and 8.10.)

## 22.5 What can go wrong

Each mistake in a `zeno.package` is shown with its file and line, as a mistake in a program is. A few of the messages:

```
error: this does not say which version of `units` is needed. Add `at` and a tag,
a branch or a commit of that repository, as in `needs units from "..." at "v1.0.0"`

error: the package in `lib` is called `filters`, not `filter`. Write `needs filters ...`

error: two packages called `b` are needed: the one in `a/b` and the one in `b`.
A program has one package of each name. Make the `needs` lines agree
```

### What this chapter said

`zeno new` starts a program, `zeno new --package` a package. `zeno.package` names a package and lists what it needs: a folder with `in`, or a git repository and one exact version with `from` and `at`. An import whose first word is a needed package reads that package's folder. `zeno fetch` gets git packages into `packages/` and pins them in `zeno.lock`; nothing else touches the network.

### Things to try

1. Make a package `units_extra` with a function that turns a temperature in `degC` into Fahrenheit as a float, and use it from a program beside it.
2. Misspell the package's name in the program's `needs` line, and read the message.
3. Add a test to `filters.zeno` and run `zeno test` in the package's folder (Testing).
4. Put a package of yours in a git repository, tag it, and need it from a program with `from` and `at`.

CHAPTER 23

# Reading and Writing

A program is more useful when it can take something in and leave something behind: lines typed at a keyboard or sent down a serial line, words on the command line, files on a disk. This chapter covers all three. `print` has been with us since the first page.

| To | Write | Gives |
|---|---|---|
| read a line typed | `input()`, `input("Name: ")` | `str or none` |
| read the command line | `program.arguments()` | `list[str]` |
| read a file | `files.read(path)` | `str` |
| write a file | `files.write(path, text)` | nothing |
| add to a file | `files.append(path, text)` | nothing |
| ask if a file is there | `files.exists(path)` | `bool` |
| remove a file | `files.remove(path)` | nothing |

## 23.1 print

`print` shows its values on one line, with a space between them, and ends the line. Text is shown as it is; inside a list, it is shown in quotes. To lay a line out exactly, build it with an `f` string (Text):

```
let volts = 3.3
print("supply", volts, ["a", 1])
print(f"supply={volts:.2f}V")
print()
print("after an empty line")
```

```
supply 3.3 ["a", 1]
supply=3.30V

after an empty line
```

## 23.2 Reading lines: input

`input()` reads the next line, without its line end. Given a question, it prints the question first and stays on that line. When there are no more lines, it gives `none`: at a keyboard that is Ctrl-D (Ctrl-Z then Enter on Windows), and for a file sent in with < it is the end of the file.

```
let name = input("Name: ")
if name is none {
    print("nobody there")
} else {
    print(f"Hello, {name}!")
```

```
}
```

```
Name: Hello, Ada!
```

Here the line `Ada` was sent in rather than typed, so it does not appear after the question as it would at a keyboard.

Because the answer may be `none`, it must be asked about before it is used as text (Values That May Be Missing). A loop that reads every line breaks on `none`, and after that the checker knows the line is a `str`:

```
var total = 0
while true {
    let line = input()
    if line is none {
        break
    }
    try {
        total += int(line.strip())
    } except ValueError {
        print("not a number:", line)
    }
}
print("total", total)
```

```
not a number: fast
total 49
```

Here the program was given four lines. Run it yourself and type them, or put them in a file and send it in: `zeno run total.zeno < numbers.txt`.

> **Why.** Python's `input()` raises an error at the end of the input, so a loop that reads everything needs a `try`. `none` is the answer "there is no line", and the type makes sure the program handles it. (Decision 6.18.)

On a board, a firmware C library usually ties the input to a serial line, and then `input()` reads a line from the serial port.

## 23.3 The command line

`program.arguments()` gives the words written after the file, as a list of text. The program's own name is not in it, so the first word is `[0]`:

greet.zeno

```
import program

let names = program.arguments()
if len(names) == 0 {
    print("usage: zeno run greet.zeno <names>")
}
for name in names {
    print("hello,", name)
}
```

```
hello, Ada
hello, Grace
```

With `zeno run`, every word after the file belongs to the program, even one that looks like an option of Zeno's; Zeno's own options go before the file: `zeno run --vm greet.zeno Ada`. A compiled program gets the same list from the words after its name: `./greet Ada Grace`.

## 23.4 Files

```
import files

files.write("shopping.txt", "milk\neggs\n")
files.append("shopping.txt", "bread\n")
print(files.read("shopping.txt").lines())
print(files.exists("shopping.txt"))
files.remove("shopping.txt")
print(files.exists("shopping.txt"))
```

```
["milk", "eggs", "bread"]
true
false
```

A file is read whole, as one `str`, and written whole. There is no `open` and no `close`, so there is nothing to forget to close. `write` makes the file, or empties it first; `append` makes it, or adds to its end.

`write` and `append` put down exactly the text they are given. A program that wants lines ends each with `"\n"`, and reads them back with `lines()`, which knows both kinds of line end, so a file from Windows reads the same.

> **Why.** Opening and closing is a pair that every program must get right on every path, errors included, and a file left open is a classic leak. Reading and writing whole files covers what most programs do, with no pair to get wrong. Reading a large file in pieces needs an open file with an owner, and waits for a program that needs it. (Decisions 6.12 and 6.13.)

### 23.4.1 When a file cannot be used

A file that cannot be used raises a `FileError`, with a message that says why:

| What happened | The message |
|---|---|
| nothing is at the path | `there is no file `notes.txt`` |
| it is a folder, or may not be read | `cannot read `notes.txt`` |
| it is not text | `` `notes.txt` does not hold text `` |
| it cannot be made or written | `cannot write `notes.txt`` |
| it cannot be removed | `cannot remove `notes.txt`` |

```
import files

function first_line(path: str) -> str raises FileError {
    let lines = files.read(path).lines()
    if len(lines) == 0 {
        return ""
    }
```

```
    return lines[0]
}

try {
    print(first_line("settings.txt"))
} except FileError as e {
    print("no settings:", e.message)
}
```

```
no settings: there is no file `settings.txt`
```

A missing file is not a mistake in the program: no check the program could make beforehand rules it out, since another program may remove the file in between. So `FileError` can always be caught, even in a firmware build where other errors stop the program. A function that uses a file says `raises FileError`, or catches it. (Decision 6.14.)

### 23.4.2 A file holds text

`files.read` gives a `str`, and a `str` is characters, so the file must be UTF-8 text. A picture or any other bytes are refused with a `FileError`. Reading bytes is not in version 0.1.

## 23.5 A worked example: a log book

This program keeps a log of readings in a file. Each run adds the readings given on the command line, then prints every reading so far and their average:

logbook.zeno

```
import files
import program

let path = "readings.txt"

for word in program.arguments() {
    try {
        let value = float(word)
        files.append(path, f"{value}\n")
    } except ValueError {
        print("skipped:", word)
    }
}

var values: list[float] = []
if files.exists(path) {
    for line in files.read(path).lines() {
        values.append(float(line))
    }
}
print(len(values), "readings:", values)
if len(values) > 0 {
    print(f"average {sum(values) / float(len(values)):.2f}")
}
```

```
$ zeno run logbook.zeno 21.5 22.0
2 readings: [21.5, 22.0]
average 21.75
```

```
$ zeno run logbook.zeno 23.5
3 readings: [21.5, 22.0, 23.5]
average 22.33
```

### What this chapter said

`print` writes a line. `input()` reads one, and gives `none` when there are no more. `program.arguments()` gives the words after the file. `files.read`, `write`, `append`, `exists` and `remove` work on whole files of text, and every one but `exists` may raise a `FileError`, which can always be caught.

### Things to try

1. Write a program that counts the lines, words and characters of the text sent to it with `<`.
2. Write `copy.zeno`, which copies the file named by its first argument to the file named by its second, and says clearly what is wrong when it is given the wrong number of words.
3. Keep a counter in a file: each run reads it, adds one, writes it back, and prints it. What should the first run do?
4. Try to read a picture with `files.read` and read the message.

CHAPTER 24

# Mathematics

The arithmetic operators are in the language itself (Numbers and Operators). A few functions sit beside them with no import, and the `math` module holds the rest: square roots, rounding, angles, logarithms and powers of $e$.

## 24.1 The functions that need no import

| Written | Gives |
|---|---|
| `abs(x)` | `x` without its sign |
| `min(a, b, ...)`, `min(xs)` | the smallest of the numbers, or of a list |
| `max(a, b, ...)`, `max(xs)` | the largest |
| `sum(xs)` | the numbers of a list added up |

```
print(abs(-3), abs(2.5), min(3, 1, 2), max([4, 9, 2]), sum([1.5, 2.5]))
```

```
3 2.5 1 9 4.0
```

## 24.2 The math module

| Written | Takes | Gives |
|---|---|---|
| `math.pi`, `math.e` | | `float` |
| `math.sqrt(x)` | a number, 0 or more | `float` |
| `math.floor(x)`, `ceil`, `round` | a number | `int` |
| `math.sin(a)`, `cos`, `tan` | an angle | `float` |
| `math.asin(x)`, `acos` | a number from $-1$ to 1 | `rad` |
| `math.atan(x)` | a number | `rad` |
| `math.atan2(y, x)` | two numbers | `rad` |
| `math.exp(x)` | a number | `float` |
| `math.log(x)`, `log2`, `log10` | a number above 0 | `float` |

```
import math

print(math.sqrt(2.0), math.pi)
print(math.floor(-1.5), math.ceil(1.2), math.round(2.5), math.round(-2.5))
print(math.log(math.e), math.log2(8.0), math.log10(1000.0), math.exp(0.0))
```

```
1.4142135623730951 3.141592653589793
-2 2 3 -3
1.0 3.0 3.0 1.0
```

### 24.2.1 Rounding

`floor` rounds down, `ceil` rounds up, and `round` goes to the nearest whole number, with a half going away from zero. All three give an `int`, because an index or a register value is usually what is wanted next. To show a number with two decimals, don't round it; format it: `f"{x:.2f}"`.

> **Why.** Python's `round` sends a half to the nearest even number, so `round(2.5)` is 2 and `round(3.5)` is 4. That is better for adding up many rounded numbers, and a surprise to nearly everyone who meets it. Zeno rounds as school taught, as C and Rust do. (Decision 6.25.)

### 24.2.2 Logarithms

`log` is the natural logarithm, with base $e$, as in C, Python, JavaScript and Java. `log2` and `log10` have their bases in their names. There is no `log(x, base)`: `log2(8.0)` is exactly 3, while dividing two natural logarithms gives `3.0000000000000004`. (Decision 6.26.)

### 24.2.3 A number with no answer

The square root of a negative number, or the logarithm of 0, has no answer, and is a `ValueError` with the number in its message:

```
import math
try {
    print(math.sqrt(-1.0))
} except ValueError as e {
    print(e.message)
}
print(math.log(0.0))
```

```
cannot take the square root of -1.0
error: uncaught ValueError: cannot take the logarithm of 0.0
 --> main.zeno:7:7
  |
7 | print(math.log(0.0))
  |       ^
```

> **Why.** C gives a `nan`, "not a number", which spreads through every sum it touches and is printed twenty lines later, far from the line that made it. An error at the line that asked is easier to find. (Decision 6.27.)

## 24.3 Angles

Degrees handed to a function that counts in radians is the classic mistake with angles, and in C it gives a wrong number with no complaint. In Zeno an angle can carry its unit (Numbers with Units), and the functions of `math` read it:

```
import math
```

```
print(math.sin(30 deg), math.sin(0.5 rad), math.sin(0.5))
print(math.cos(180 deg), math.tan(45 deg))
```

```
0.49999999999999994 0.479425538604203 0.479425538604203
-1.0 0.9999999999999999
```

A plain number is taken too, and means radians, because that is what the sine of a number means in mathematics, and signal code is full of `math.sin(2.0 * math.pi * f * t)`. The sine of 30 degrees is not exactly 0.5, because a float cannot hold $\pi/6$ exactly; Python prints the same digits.

The functions that give an angle, `asin`, `acos`, `atan` and `atan2`, give it in `rad`, so the value says what it is. `.deg` changes it:

```
import math

let heading = math.atan2(1.0, -1.0)
print(heading, heading.deg)
print(heading / 1 rad)
```

```
2.356194490192345 rad 135 deg
2.356194490192345
```

`atan2(y, x)` takes `y` first. It is the angle of the point $(x, y)$ seen from the middle, and unlike `atan(y / x)` it knows which side of the circle the point is on.

## 24.4 What is not here

| Elsewhere | In Zeno |
|---|---|
| `pow(x, y)` | `x ** y` |
| `trunc(x)` | `int(x)` |
| `fabs(x)` | `abs(x)` |
| `degrees(x)`, `radians(x)` | `angle.deg`, `angle.rad` |
| `round(x, 2)` | `f"{x:.2f}"` |

The answers are the C library's, in every way of running a program: a compiled program calls `sin` and `log` in the C library, and `zeno run` reaches the same functions. Two machines with different C libraries can differ in the last digit of an answer that is not exact. (Decision 6.28.)

## 24.5 Five small instruments

This program, from the `examples` folder of Zeno, is five pieces of arithmetic that firmware does every day: a compass from a magnetometer, a thermistor, a level meter in decibels, a capacitor charging, and a sine table for an 8-bit converter.

examples/instruments.zeno

```
# Five small instruments, each a few lines of arithmetic that firmware
# does every day. They use the `math` module: angles, logarithms, powers
# of e, and rounding.
#
```

```
#     zeno run examples/instruments.zeno

import math

# A compass. A magnetometer gives the field along x and along y, and
# `atan2` turns the two into an angle, all the way round. The angle comes
# back in radians and says so; `.deg` changes it, and a heading is
# counted from 0 to 360.
function heading(x: float, y: float) -> deg {
    let angle = math.atan2(y, x).deg
    if angle < 0 deg {
        return angle + 360 deg
    }
    return angle
}

# A thermistor. Its resistance falls as it warms, along a curve with a
# logarithm in it (the "beta" equation). 10 kohm at 25 degC, beta 3950.
function thermistor(ohms: float) -> float {
    let kelvin = 1 / (1 / 298.15 + math.log(ohms / 10000.0) / 3950.0)
    return kelvin - 273.15
}

# A level meter. Decibels are twenty times the logarithm, base ten, of
# how many times bigger one voltage is than another.
function decibels(volts: float, reference: float) -> float {
    return 20 * math.log10(volts / reference)
}

# A capacitor charging through a resistor: the part of the supply it has
# reached after `seconds`. `tau` is the resistance times the capacitance.
function charged(seconds: float, tau: float) -> float {
    return 1 - math.exp(-seconds / tau)
}

# A sine table for a digital-to-analogue converter of eight bits: one
# turn in `steps` steps, each a whole number from 0 to 255.
function sine_table(steps: int) -> list[int] {
    var table: list[int] = []
    for step in range(steps) {
        let angle = step * 360 / steps * 1 deg
        table.append(math.round(127.5 + 127.5 * math.sin(angle)))
    }
    return table
}

print("compass")
let fields = [[1.0, 0.0], [1.0, 1.0], [0.0, 1.0], [-1.0, 1.0], [-1.0, 0.0], [0.0, -1.0],
    ↪  [3.0, -4.0]]
for field in fields {
    let angle = heading(field[0], field[1])
    # A format on a quantity is the number's, and the unit follows it.
    # The width counts both: six for the number, four for ` deg`.
    print(f"  x {field[0]:5.1f}  y {field[1]:5.1f}  heading {angle:10.1f}")
}

print("thermistor")
for ohms in [32650.0, 10000.0, 3603.0, 1070.0] {
```

```
    print(f"  {ohms:8.0f} ohm  {thermistor(ohms):6.1f} degC")
}

print("level meter")
for volts in [1.0, 0.5, 0.1, 2.0] {
    print(f"  {volts:4.1f} V  {decibels(volts, 1.0):6.1f} dB")
}

print("capacitor, tau 0.5 s")
for tenths in range(0, 30, 5) {
    let seconds = tenths / 10
    print(f"  after {seconds:.1f} s  {100 * charged(seconds, 0.5):5.1f} %")
}

print("sine table")
print(" ", sine_table(16))
```

```
compass
  x   1.0  y   0.0  heading    0.0 deg
  x   1.0  y   1.0  heading   45.0 deg
  x   0.0  y   1.0  heading   90.0 deg
  x  -1.0  y   1.0  heading  135.0 deg
  x  -1.0  y   0.0  heading  180.0 deg
  x   0.0  y  -1.0  heading  270.0 deg
  x   3.0  y  -4.0  heading  306.9 deg
thermistor
     32650 ohm     0.6 degC
     10000 ohm    25.0 degC
      3603 ohm    49.9 degC
      1070 ohm    85.5 degC
level meter
   1.0 V     0.0 dB
   0.5 V    -6.0 dB
   0.1 V   -20.0 dB
   2.0 V     6.0 dB
capacitor, tau 0.5 s
  after 0.0 s    0.0 %
  after 0.5 s   63.2 %
  after 1.0 s   86.5 %
  after 1.5 s   95.0 %
  after 2.0 s   98.2 %
  after 2.5 s   99.3 %
sine table
  [128, 176, 218, 245, 255, 245, 218, 176, 128, 79, 37, 10, 0, 10, 37, 79]
```

### What this chapter said

`abs`, `min`, `max` and `sum` need no import. `math` holds `pi`, `e`, `sqrt`, `floor`, `ceil`, `round` (half away from zero, giving an int), the angle functions, `exp` and three logarithms. Angles go in with their units, or as radians, and come out in `rad`. A number with no answer is a `ValueError`, never `nan`.

### Things to try

1. Write a function `distance` that takes the four `floats` `x1`, `y1`, `x2` and `y2`, and gives the distance between the two points.

2. Make a sine table of 32 entries for a 10-bit converter (0 to 1023).
3. Work out how many bits are needed to count to a number with `log2` and `ceil`. Check it for 255, 256 and 1000.
4. What does `math.asin(1.5)` do? Catch it and print the message.

CHAPTER 25

# Testing

A test is a small program that checks a part of a bigger one, and says clearly when the part is wrong. Firmware that has been tested on a computer before it reaches a board is firmware that wastes fewer afternoons at the bench. Zeno has tests in the language: a `test` block, an `assert` statement, and the command `zeno test`.

## 25.1 A first test

temperature.zeno

```
function to_fahrenheit(celsius: float) -> float {
    return celsius * 1.8 + 32.0
}

test "water freezes at 32" {
    assert to_fahrenheit(0.0) == 32.0
}

test "water boils at 212" {
    assert to_fahrenheit(100.0) == 212.0
}
```

```
$ zeno test temperature.zeno
temperature.zeno
  ok      water freezes at 32
  ok      water boils at 212

2 tests: 2 passed, 0 failed.
```

A test is the word `test`, a name in quotes that says in ordinary words what it checks, and a block. It stands at the top of a file, beside what it tests. `zeno test` runs the file as `zeno run` would, and then each test, in the order written.

`zeno run` goes past the tests, and `zeno build` leaves them out altogether, so a board pays nothing for them. But they are checked with the program by `zeno check` and `zeno run`, so a test cannot go stale without anybody noticing.

## 25.2 assert

`assert` and a condition say what must be so. When the condition is a comparison, a failure shows what was on each side:

average.zeno

```
function average(values: list[float]) -> float {
    return sum(values) / float(len(values) + 1)
```

```
}

test "the average of one number is that number" {
    assert average([4.0]) == 4.0
}

test "an average lies between the smallest and the largest" {
    let values = [1.0, 2.0, 6.0]
    assert min(values) <= average(values)
}
```

```
$ zeno test average.zeno
average.zeno
  FAILED  the average of one number is that number
  ok      an average lies between the smallest and the largest

FAILED: the average of one number is that number
error: this is false. The left of `==` is 2.0, and the right is 4.0
 --> average.zeno:6:12
  |
6 |     assert average([4.0]) == 4.0
  |            ^


2 tests: 1 passed, 1 failed.
```

The first test failed, and the message shows that the left side was 2.0 where 4.0 was wanted: the `+ 1` is the bug. The second test passed, because a wrong average can still lie in the right range. A failed `assert` ends its test; the next test runs anyway. The command fails when any test fails, so a script or a build server can tell.

> **Why.** Other languages grow a family of asserts, `assert_eq`, `assert_ne`, `assertIn`, each so that a failure can show the values. Zeno has the comparison in front of it, so the one `assert` can show them, and there is one thing to learn. (Decision 10.17.)

`assert` is only for tests. In a program, `if` and `raise` say what is wrong and to whom, and can be caught. A failed `assert` is not an error of the program, so `except Error` does not catch it.

## 25.3 Testing that something fails

Fault paths are half of what firmware has to get right. `assert raises` checks that a block raises an error of a kind:

speed.zeno

```
function check_speed(speed: int) -> int raises ValueError {
    if speed < 0 or speed > 100 {
        raise ValueError(f"a speed is 0 to 100, and {speed} is not")
    }
    return speed
}

test "speeds in range are kept" {
    assert check_speed(0) == 0
    assert check_speed(100) == 100
}
```

```
test "speeds out of range are refused" {
    assert raises ValueError {
        check_speed(101)
    }
    assert raises ValueError {
        check_speed(-1)
    }
}
```

```
$ zeno test speed.zeno
speed.zeno
  ok      speeds in range are kept
  ok      speeds out of range are refused

2 tests: 2 passed, 0 failed.
```

The test fails if the block raises nothing, or raises another kind, and says which. To take an error of any kind, name `Error`: `assert raises Error`.

## 25.4 Where tests go

A test can stand in the file it tests, and then it sees what is private to that file, which is usually where the things worth testing are. A file of tests alone, such as `ring_test.zeno` importing `ring`, is the same thing with nothing else in it.

`zeno test` with a folder tests every file in it, and below it, that has a test. With no folder it means the folder it is run in:

```
$ zeno test ring
ring/ring.zeno
  ok      a new ring is empty
  ok      bytes come out in the order they went in
  ok      a full ring keeps what it has
  ok      the ring goes round
  ok      a ring of no size is refused

5 tests: 5 passed, 0 failed.
```

Each file is tested as a program of its own: its tests start from the state the file leaves, and the tests of two files cannot disturb each other. A file with no test in it is not run at all, so testing a folder does not run `main.zeno` and blink its LED. The tests of an imported file run when that file is tested, not each time another file imports it.

Tests of one file share what the file declares, so a test that changes a `var` at the top of the file leaves it changed for the next. A test that needs a fresh value makes it inside itself; what a test declares is gone when it ends.

> **Why.** A test with a name in plain words says what is checked, and is what a failure is reported under. Functions whose names begin with `test_`, as in Python and Go, are a rule nobody can see in the program. Marks such as `@test` are a new piece of syntax for one use. Tests only in files of their own could not test anything private. `test` is not even a reserved word: `test "` at the start of a statement can mean nothing else, so programs keep the name for their own use. (Decision 10.16.)

## 25.5 Testing code that uses the board

Code that drives hardware can be tested on a computer too, because `zeno run` simulates the board: a pin remembers whether it is on, and a register remembers what was written to it (The Board). A test can drive a pin and then ask it:

heater.zeno

```
from board import pin

let heater = pin(5)

function control(temperature: float) {
    if temperature < 19.0 {
        heater.on()
    } else if temperature > 21.0 {
        heater.off()
    }
}

test "the heater comes on when it is cold" {
    control(15.0)
    assert heater.is_on()
}

test "it stays on until it is warm" {
    control(15.0)
    control(20.0)
    assert heater.is_on()
    control(22.0)
    assert not heater.is_on()
}
```

```
$ zeno test heater.zeno
heater.zeno
pin 5 on
  ok      the heater comes on when it is cold
pin 5 off
  ok      it stays on until it is warm

2 tests: 2 passed, 0 failed.
```

The simulated board says each change of a pin as it happens, which is why `pin 5 on` appears among the results.

## 25.6 What tests leave out

In version 0.1 there is no message after an `assert` (the test's name and the values shown already say most of it), no `assert` outside a test, no code run before and after each test (a function that builds what the tests need does that), no choosing tests by name on the command line, and no running of tests in a compiled program. (Decision 10.21.)

### What this chapter said

`test "what it checks" { ... }` stands at the top of a file. `assert condition` fails the test when the condition is false, and shows both sides of a comparison. `assert raises Kind { ... }` checks that a block raises. `zeno test` runs a file, a folder or the folder it is in, and fails if any test fails. Builds leave tests out.

### Things to try

1. Write three tests for `checksum` from Sized Numbers and Bits, including one where the total passes 255.
2. Fix the `average` function above and run its tests again. Then add a test for an empty list. What should `average` do there, and what does the test say?
3. Write a test that checks `int("fast")` raises a `ValueError`. Then change the kind to `KeyError` and read the failure.
4. Make a test in one file change a `var` at the top of the file, and see the next test find it changed.

CHAPTER 26

# The Tools

Everything Zeno does is done by one program, `zeno`, with a word after it that says what to do. This chapter goes through the words a program's author uses every day, then the ones that show what the compiler sees, and then setting up an editor.

## 26.1 The commands

| Command | Does |
|---|---|
| `zeno run file` | check a program, then run it |
| `zeno check file` | check it for mistakes, without running it |
| `zeno test [file or folder]` | run its tests (Testing) |
| `zeno build file` | compile it to a program of its own (Compiling) |
| `zeno format file` | lay it out the way Zeno lays programs out |
| `zeno new name` | make a folder with a new program (Packages) |
| `zeno fetch` | get the packages a program needs |
| `zeno lsp` | the language server, which an editor starts |
| `zeno tokens`, `parse`, `join`, `bytecode` | show what the compiler sees |
| `zeno --version`, `zeno --help` | the version, and a list of all this |

Appendix E lists every command and option.

## 26.2 run and check

`zeno run` reads the whole program, checks it, and runs it only if the check found nothing. `zeno check` does the first half and stops, which is quick and is what an editor or a build script wants. Both report every mistake they can be sure of, not only the first, and end with how many there were:

two.zeno

```
let speed = "fast" * 2
print(sped)
```

```
$ zeno check two.zeno
error: `*` does not work between str and int
 --> two.zeno:1:13
  |
1 | let speed = "fast" * 2
  |             ^

error: `sped` is not defined
```

```
  --> two.zeno:2:7
   |
 2 | print(sped)
   |       ^

 2 errors found. Nothing was run.
```

**Why.** A mistake found before the program starts costs seconds. The same mistake found halfway through a run, with a motor already turning, costs more. The checker reports only what it is sure of, so untyped code is not held back. (Decision 4.4.)

Options that change how a program is checked or run go before the file. Words after the file are the program's own (Reading and Writing):

| Option | Means |
|---|---|
| `--strict` | every type must be known (Types) |
| `--vm` | with `run`: compile to bytecode and run that, faster |
| `--no-collector` | with `check`: check as a build with no collector would (Building with No Collector) |

## 26.3 format

`zeno format` lays a file out the one way Zeno lays programs out. There is nothing to configure. Here is a file before:

queue.zeno

```
class  Queue{

  size:int      # how many fit


  function put(self,byte:u8)->bool{
    if self.size==0{return false}
  return true
 }
}
```

and after:

```
$ zeno format queue.zeno
Formatted queue.zeno
$ cat queue.zeno
class Queue {
    size: int      # how many fit

    function put(self, byte: u8) -> bool {
        if self.size == 0 { return false }
        return true
    }
}
```

The rules are few. Four spaces for each open bracket. One space around operators, `=` and `->`, and after `,` and `:`. None inside brackets, nor before the bracket of a call or a type. Never two blank

lines in a row, and none at the top or bottom of a block. A comment after code keeps the distance its author gave it.

And three things it never does. It never changes a token: `0x4000_4400` stays spelled so. It never moves a line break: a short `case` on one line stays on one line, and a call broken over three lines stays on three. And it never formats a file with a mistake in it; it shows the mistake instead.

`zeno format folder` lays out every `.zeno` file in the folder, and `zeno format --check folder` changes nothing and says which files are not laid out, for a build server.

> **Why.** When a language has one layout, nobody argues about layout, and every program in it looks familiar to every reader. Go's `gofmt` showed that this is worth more than any one person's taste. Because the formatter only decides spaces and indents between tokens it leaves alone, it cannot change what a program means. (Decisions 9.1 to 9.9.)

## 26.4 Seeing what the compiler sees

Four commands show a program at each stage of the compiler. They are for curiosity and for learning how a compiler works, which Inside Zeno is about.

`zeno tokens` shows the words the lexer cut the file into:

tiny.zeno

```
let total = 2 + 3 * 4
print(total)
```

```
$ zeno tokens tiny.zeno
   1:1   Let
   1:5   Ident("total")
   1:11  Assign
   1:13  Int(2)
   1:15  Plus
   1:17  Int(3)
   1:19  Star
   1:21  Int(4)
   1:22  Newline
   2:1   Ident("print")
   2:6   LParen
   2:7   Ident("total")
   2:12  RParen
   2:13  Newline
   3:1   Eof
```

`zeno parse` shows the tree the parser built from them, where `3 * 4` already sits below the `+`:

tiny.zeno

```
let total = 2 + 3 * 4
print(total)
```

```
$ zeno parse tiny.zeno
let total = (2 + (3 * 4))
print(total)
```

`zeno join` shows the one program that a file and the files it imports are joined into (Programs in Several Files), and `zeno bytecode` the instructions that `zeno run --vm` runs:

tiny.zeno

```
let total = 2 + 3 * 4
print(total)
```

```
$ zeno bytecode tiny.zeno
== main() ==
0000    1  Const        0          ; 2
0001    |  Const        1          ; 3
0002    |  Const        2          ; 4
0003    |  Binary       *
0004    |  Binary       +
0005    2  Const        3          ; <function print>
0006    |  GetLocal     0          ; total
0007    |  Call         1
0008    |  Pop          1
0009    |  Const        4          ; none
0010    |  Return
```

## 26.5 Your editor

`zeno lsp` is a *language server*: a program that an editor starts and talks to in a fixed way, the Language Server Protocol. With it, the editor underlines mistakes as you type, lays the file out when you save, jumps to where a name was declared, and shows a name's type when you rest on it. What it underlines is exactly what `zeno check` would print, because it runs the same checker. VS Code, Neovim, Helix, Zed and Emacs all speak the protocol.

The `editors` folder of the Zeno repository has what each editor needs: an extension for VS Code with colours and the few lines that start `zeno lsp`, and a syntax file for Vim and Neovim. `editors/README.md` has the steps for each. For Neovim, the heart of it is:

```
vim.lsp.start({
  name = "zeno",
  cmd = { "zeno", "lsp" },
  root_dir = vim.fn.getcwd(),
})
```

and for Helix, in `~/.config/helix/languages.toml`:

```
[language-server.zeno]
command = "zeno"
args = ["lsp"]

[[language]]
name = "zeno"
scope = "source.zeno"
file-types = ["zeno"]
language-servers = ["zeno"]
```

For every editor, `zeno` must be somewhere the editor finds it, on the `PATH` (Getting Zeno).

### What this chapter said

`zeno run` checks and runs, `zeno check` only checks, and both report every mistake they are sure of. `zeno format` lays a file out the one way, changing only spaces. `tokens`, `parse`, `join` and `bytecode` show the compiler's view. `zeno lsp` brings the checker and the formatter into any editor that speaks the Language Server Protocol.

### Things to try

1. Write a file as badly laid out as you can, keeping it correct, and run `zeno format` on it.
2. Run `zeno tokens` on a line with a string, a number with a unit and a comment. Which of them are tokens?
3. Run `zeno parse` on `a or b and c`, and on `not a == b`. What does each tree say about which goes first?
4. Set up your editor, make a mistake, and watch it underlined.

PART IV

# Zeno on a Machine

CHAPTER 27

# The Board

Zeno is for programs that run on a board: a microcontroller with pins that drive LEDs and relays, registers that control its peripherals, and a clock. This chapter is about the `board` module, which is how a program reaches them, and about the simulated board that lets the same program run on a computer.

## 27.1 Blink

The program Zeno was made for, written on the first day, before there was a compiler, is this:

```
from board import led
from time import sleep

while true {
    led.toggle()
    sleep(500 ms)
}
```

It never ends, as firmware does not. Run on a computer, it prints `led on`, `led off`, `led on` every half second, until Ctrl-C stops it. Compiled for a microcontroller, the same lines blink the LED.

## 27.2 The simulated board

A computer has no LED on pin 13, so `zeno run` gives the program a *simulated* board. It has 64 pins, all low to begin with, with the LED on pin 13, where an Arduino Uno has it. It prints a line each time a pin *changes* level, which is what a logic analyser clipped to the pin would show:

```
from board import led

print(led, led.is_on())
led.on()
led.on()
led.toggle()
print(led.is_on())
```

```
Pin(13) false
led on
led off
false
```

The second `led.on()` printed nothing, because the pin was already high and did not move.

## 27.3 Pins

`led` is the pin the LED is on, and `pin(n)` is the pin with that number. A pin can do four things:

| Written | Does |
|---|---|
| `p.on()`, `p.off()` | drive it high or low |
| `p.toggle()` | give it the other level |
| `p.is_on()` | is it high? A `bool` |
| `p.number` | which pin it is |

`Pin` is a type, so a function can ask for one, and a pin can go in a list, a field or a `Pin or none`:

```
from board import pin
from time import sleep

function flash(light: Pin, times: int) {
    for turn in range(times) {
        light.toggle()
        sleep(10 ms)
    }
}

let lamps = [pin(2), pin(3)]
for lamp in lamps {
    flash(lamp, 2)
}
```

```
pin 2 on
pin 2 off
pin 3 on
pin 3 off
```

A pin the board does not have is a `ValueError`:

```
from board import pin
let relay = pin(99)
```

```
error: uncaught ValueError: this board has no pin 99. Its pins are 0 to 63
 --> main.zeno:2:13
  |
2 | let relay = pin(99)
  |             ^
```

> **Why.** A pin is not an `int`: if it were, `on(count)` would compile, and so would `led + 1`. It is not an object either, because an object lives in memory from the collector, and blinking an LED should not need one. A pin is a kind of its own to the checker, and a plain number in the compiled program, so it costs what an `int` costs. Units work the same way. (Decision 3.15.)

## 27.4 The clock

`sleep` waits for a time with a unit, and `now()` reads a clock that only goes forward, in milliseconds, from a moment such as when the board was switched on (Numbers with Units). Only the

difference of two readings means anything:

```
from time import sleep, now

let start = now()
sleep(50 ms)
let took = now() - start
print("took at least 50 ms:", took >= 50 ms)
```

```
took at least 50 ms: true
```

## 27.5 Registers

A microcontroller's peripherals, its timers, serial ports and converters, are controlled through *registers*: words of memory at fixed addresses, given in the data sheet. `register(u32, address)` names one, with its size:

```
from board import register

let control = register(u32, 0x4000_4400)
let data = register(u8, 0x4000_440C)

unsafe {
    control.write(u32(control.read() | 1 << 0))
    data.write(0x41)
    print(control.read(), data.read())
}
```

```
register 0x40004400 = 0x00000001
register 0x4000440c = 0x41
1 65
```

A value written must fit the register: `control.read() | 1` is an `int`, as every sum of sized numbers is (Sized Numbers and Bits), so `u32(...)` puts it back, with a check.

Reading and writing a register must be inside `unsafe { }`, because a wrong address or a wrong bit can stop a chip, and Zeno cannot check either. The block marks the lines a reviewer should read twice:

```
from board import register
let data = register(u8, 0x4000_440C)
data.write(0x41)
```

```
error: writing a register goes inside `unsafe { }`. Zeno cannot check that it is right,
    ↪ so the block marks the lines where a mistake can damage memory
 --> main.zeno:3:1
  |
3 | data.write(0x41)
  | ^

1 error found. Nothing was run.
```

On the simulated board a register is a word that remembers what was written, and each write is printed. On a real board it is the real address. Unsafe Code is about `unsafe` and memory by hand.

## 27.6 A traffic light

This program from the `examples` folder is a traffic light on three pins, written as a state machine with an enum. Its times are short, so the whole run takes a fifth of a second:

examples/signal.zeno

```
# signal.zeno: a traffic light on three pins, written as a state machine.
#
#     zeno run examples/signal.zeno      run it on the simulated board
#     zeno build --firmware --no-collector examples/signal.zeno
#                                        the build for a microcontroller:
#                                        no setjmp, no garbage collector
#
# On a computer the board is simulated. A line such as `pin 2 on` is what
# a logic analyser clipped to that pin would show. The times are short so
# that the whole run takes a fifth of a second.

from board import pin, led
from time import sleep, now

enum Light {
    Red
    Green
    Amber
}

let red = pin(2)
let amber = pin(3)
let green = pin(4)

# Switch the lamps over: the old one off, then the new one on.
function show(light: Light) {
    match light {
        case Red {
            amber.off()
            red.on()
        }
        case Green {
            red.off()
            green.on()
        }
        case Amber {
            green.off()
            amber.on()
        }
    }
}

function after(light: Light) -> Light {
    match light {
        case Red { return Green }
        case Green { return Amber }
        case Amber { return Red }
    }
}

# How long each light stays. The unit is part of the type: a function
# that returned `40` here would not compile.
```

```
function hold(light: Light) -> ms {
    match light {
        case Red { return 40 ms }
        case Green { return 30 ms }
        case Amber { return 10 ms }
    }
}

let start = now()
var light = Red
for step in range(7) {
    show(light)
    led.toggle()                # a heartbeat: the program is alive
    sleep(hold(light))
    light = after(light)
}

# Seven steps hold for 200 ms in all. `now` only ever goes forward.
print("took at least 200 ms:", now() - start >= 200 ms)
```

```
pin 2 on
led on
pin 2 off
pin 4 on
led off
pin 4 off
pin 3 on
led on
pin 3 off
pin 2 on
led off
pin 2 off
pin 4 on
led on
pin 4 off
pin 3 on
led off
pin 3 off
pin 2 on
led on
took at least 200 ms: true
```

## 27.7 A real board underneath

A compiled program meets the hardware in four C functions, and these, with the number of pins, are everything Zeno knows about a board:

```
zeno_board_write(pin, level)   drive a pin high or low
zeno_board_read(pin)           is the pin high?
zeno_board_sleep(seconds)      wait
zeno_board_now()               a clock that only goes forward, in ms
```

The simulated board is these four, written for a computer. For a real microcontroller they are written once, over the vendor's library, in a header, and the C compiler is told where it is. Firmware shows how, with a sketch for an STM32. The Zeno program does not change by a line.

**Why.** Four functions is the smallest surface a port can have: a board is working the day those four are. The checks, such as a pin that does not exist or a wait below zero, are done once above them, for every board, so the four can be four lines each. (Decision 5.12.)

## 27.8 What the board does not have yet

In version 0.1 `is_on()` reads back what the program wrote: there is no button, and no way to make a pin an input. Interrupts, timers, PWM, an analogue converter and a serial port are not in the `board` module either. Until they are, a program reaches them through their registers, at the addresses the data sheet gives. Appendix G lists these.

### What this chapter said

`from board import led, pin, register`. A pin is driven with `on`, `off` and `toggle` and asked with `is_on`; `Pin` is a type. `sleep` and `now` keep time with units. A register is read and written inside `unsafe`. `zeno run` simulates the board and prints each change; a compiled program sits on four C functions that a real board provides.

### Things to try

1. Blink the LED in Morse code: `SOS`, with dots of 100 ms.
2. Write a class `Relay` with a pin and a name, and methods `close` and `open` that print what they do.
3. Change the traffic light so that green lasts twice as long as red, and add a pedestrian lamp on pin 5.
4. Write a register without `unsafe` and read the message.

CHAPTER 28

# Compiling

`zeno run` reads a program and does what it says, a piece at a time. `zeno build` turns the same program into C, and the machine's C compiler turns that into a program of its own: no Zeno needed to run it, and many times faster. The C can also be compiled for a microcontroller, which is where Zeno is going.

## 28.1 zeno build

primes.zeno

```
function is_prime(n: int) -> bool {
    if n < 2 {
        return false
    }
    var d = 2
    while d * d <= n {
        if n % d == 0 {
            return false
        }
        d += 1
    }
    return true
}

var count = 0
for n in range(200000) {
    if is_prime(n) {
        count += 1
    }
}
print(count, "primes below 200000")
```

```
$ zeno build primes.zeno
Wrote primes.c
Built primes. Run it with ./primes
$ ./primes
17984 primes below 200000
```

`zeno build` wrote `primes.c`, called the C compiler on the machine (`cc`, which is usually GCC or Clang), and left the program `primes` beside it. On the computer this book was built on, `zeno run` takes about two seconds over it, `zeno run --vm` about one, and `./primes` a fiftieth of a second.

## 28.2 Reading the C

The C file is kept, and it is meant to be read beside the Zeno. Here is `is_prime`:

```
$ zeno build primes.zeno > /dev/null
$ grep -A12 "^static bool z_is_prime(int64_t z2_n) {" primes.c
static bool z_is_prime(int64_t z2_n) {
    if (z2_n < 2) {
        return false;
    }
    int64_t z_d = 2;
    while (zeno_mul(z_d, z_d, 6, 11) <= z2_n) {
        if (zeno_mod(z2_n, z_d, 7, 12) == 0) {
            return false;
        }
        z_d = zeno_add(z_d, 1, 10, 9);
    }
    return true;
}
```

It is the Zeno, line for line. An `int` is an `int64_t`. `d * d` is `zeno_mul`, which checks for overflow with the compiler's own built-in and raises an `OverflowError` with the line and column (the two numbers after the operands) when the answer does not fit. In front of the program stands Zeno's runtime, a small C library for text, lists, errors and `print`, pasted into every file so that the C needs nothing else to compile.

> **Why.** C runs on every processor there is, and every microcontroller vendor ships a C compiler for its chips. Writing C first gave Zeno all of them at once, and C that a person can read is a C that a person can check. (Decisions 5.5 and 5.5b.)

## 28.3 Build agrees with run

A program must print the same whichever way it is run: `zeno run`, `zeno run --vm`, a C build, an LLVM build, a firmware build. Errors included: an uncaught error gives the same message, at the same line and column, in every one. Zeno's own tests run every test program every way and compare the output byte for byte.

> **Why.** A program that behaves one way under test and another on the board is the worst kind to debug. If `zeno run` is to be the place where firmware is tried, it must mean exactly what the compiled program means. (Decision 5.5d.)

## 28.4 A build is strict

`zeno build` always checks in strict mode: every type must be known, because a compiled program has no tags to carry them (Types, Written and Worked Out). A program that runs with dynamic values under `zeno run` is told which value is the trouble:

loose.zeno

```
function same(x) {
    return x
}
print(same(1), same("one"))
```

```
$ zeno build loose.zeno
error: strict mode: the parameter `x` of `same` needs a type. It could not be worked out
    ↪  from how it is used
```

```
 --> loose.zeno:1:1
  |
1 | function same(x) {
  | ^

1 error found. Nothing was built.
```

## 28.5 The options

| Option | Means |
|---|---|
| `--memory` | the program reports, as it ends, how much memory it used |
| `--llvm` | compile to LLVM's language in place of C, and hand it to `clang` |
| `--firmware` | errors travel as values, with no `setjmp`, as coding standards for firmware ask (Firmware) |
| `--no-collector` | no garbage collector; every list and object has one owner, and memory comes from a pool of a fixed size (Building with No Collector) |

lists.zeno

```
var squares: list[int] = []
for n in range(1000) {
    squares.append(n * n)
}
print(len(squares), squares[999])
```

```
$ zeno build --memory lists.zeno
Wrote lists.c
Built lists. Run it with ./lists
$ ./lists
1000 998001
memory: 12 blocks made (16388 bytes), at most 12312 bytes in use at once, 0 collections,
    ↪ 8 blocks freed without the collector
```

Memory explains the report.

## 28.6 Through LLVM

`zeno build --llvm` writes the program in LLVM's intermediate language, the one Clang, Rust and Swift compile through, and hands it to `clang`. The program is the same, and prints the same:

```
$ zeno build --llvm primes.zeno
Wrote primes.ll and primes_runtime.c
Built primes. Run it with ./primes
$ ./primes
17984 primes below 200000
```

The IR is written as text by Zeno itself, with no LLVM library, and `primes.ll` can be read like `primes.c`. LLVM brings its own optimiser and every processor it knows. Inside Zeno shows a

piece of it.

## 28.7 For another processor

`zeno build` compiles for the computer it runs on. For a microcontroller, take the C file it wrote and give it to the microcontroller's own C compiler, with the board's header (Firmware):

```
$ zeno build --firmware --no-collector blink.zeno
$ arm-none-eabi-gcc -O2 -DZENO_BOARD='"nucleo.h"' -c blink.c
```

**Why.** Building C for a target is a job the target's own tools already do well, with the flags, linker scripts and start-up code that only they know. Zeno's part ends at a C file that compiles anywhere. (Decision 5.12c.)

## 28.8 What does not compile yet

Nearly the whole language compiles. A few corners do not, and `zeno build` names each one and points at the line, while `zeno run` still runs the program:

inner.zeno

```
function outer() -> int {
    function inner() -> int {
        return 2
    }
    return inner()
}
print(outer())
```

```
$ zeno build inner.zeno
error: `zeno build` cannot compile a function inside a function or block yet. `zeno run`
    ↪ can still run this program
 --> inner.zeno:2:5
  |
2 |     function inner() -> int {
  |     ^
```

A function written with no name, `let inner = function() { ... }`, does compile. Appendix G has the full list.

### What this chapter said

`zeno build` writes C, compiles it with the machine's compiler, and keeps both. The compiled program prints what `zeno run` prints, errors included, and is many times faster. A build is always strict. `--llvm` goes through LLVM; `--memory`, `--firmware` and `--no-collector` are for the board. For another processor, the C goes to that processor's compiler.

### Things to try

1. Build a program from an earlier chapter, open its C file, and find each of your functions in it.

2. Time `zeno run`, `zeno run --vm` and the built program on the primes example, with a larger limit.
3. Make a program overflow an `int` and build it. Is the message the same as `zeno run`'s?
4. Build with `--llvm`, open the `.ll` file, and find `is_prime`.

CHAPTER 29

# Memory

Numbers and bools live in the variables that hold them, and need no memory of their own. Text that is built, lists, objects, enum values that carry something, and functions that keep variables all need memory while the program runs, and somebody must give it back when the program is done with it. This chapter is about how an ordinary compiled Zeno program does that, with a *garbage collector*. The next three chapters are about the ways firmware can do without one.

`zeno run` needs none of this: the interpreter is a Rust program, and Rust frees its values for it. Everything here is about `zeno build`.

## 29.1 What takes memory

- text that is built: `a + b`, an `f` string, `text.upper()`;
- every list, and every dict;
- every object, and every enum value that carries something;
- a function written inside another, with the variables it keeps.

Each of these is a *block*: a piece of memory from the C library's `malloc`. The question is when to give each block back with `free`.

## 29.2 The question nobody can answer in advance

```
function label(reading: Reading) -> str {
    let text = f"tick {reading.tick}"
    return text
}
```

Is `text` finished with when `label` returns? No: the caller has it now, and may put it in a list, which goes into an object, which a function remembers. In C the programmer keeps track of all that by hand, and the mistakes are famous: freeing too early, so that the program uses memory that is now something else; freeing twice; or never freeing at all.

A collector answers a simpler question, while the program runs: *can the program still reach this block?* If no variable leads to it, directly or through other blocks, nothing can ever use it again, and it is safe to free.

## 29.3 Mark and sweep

Zeno's collector is the oldest and simplest kind. It was invented for Lisp in 1960, and it is what MicroPython runs on microcontrollers.

1. Every block is written in a table as it is handed out.
2. **Mark.** Start from the variables, the *roots*. Whatever a root points at is in use: mark it. Whatever a marked block points at is in use too. Carry on until nothing new turns up.
3. **Sweep.** Go through the table. A block with no mark cannot be reached: free it. Take the marks off the rest.

Two objects that point at each other, and at nothing in use, are freed like anything else: no root reaches them, so neither is marked. That is the case that defeats the other common method, counting references.

The roots are the variables at the top of the file, which the compiler lists, and everything on the C stack: the collector reads the stack from the function running now back to `main`, a word at a time, and asks of each word whether it is the address of one of its blocks. When unsure, it keeps, so it can never free a block in use.

## 29.4 Seeing it work

`zeno build --memory` makes a program that reports, as it ends, what memory it used. This one, from the `examples` folder, logs a reading every tick for 50 000 ticks, and keeps only the last few and the hottest:

memory.zeno

```
# memory.zeno: a program that makes far more than it keeps.
#
#     zeno run examples/memory.zeno              run it with the interpreter
#     zeno build --memory examples/memory.zeno   compile it, with a memory report
#     ./memory
#
# A logger takes a reading every tick. Each reading is an object with some
# text in it, so every tick takes a little memory. Only the last few
# readings and the hottest one are kept. The rest is garbage, and the
# garbage collector in the compiled program gives that memory back.

class Reading {
    tick: int
    celsius: float
    note: str
}

function describe(reading: Reading) -> str {
    return f"tick {reading.tick}: {reading.celsius:.1f} C ({reading.note})"
}

var recent: list[Reading] = []          # the last few readings
var hottest: Reading or none = none
var lines = 0

for tick in range(50000) {
    let celsius = 20.0 + (tick * 37 % 151) / 10.0
    let reading = Reading(tick, celsius, f"sample {tick}")

    # Start a new list every five readings. The old list, and the readings
    # only it knew about, can no longer be reached from any variable.
    if len(recent) == 5 {
        recent = []
```

```
    }
    recent.append(reading)

    if hottest is none or reading.celsius > hottest.celsius {
        hottest = reading
    }

    # Text that is made, measured and forgotten at once.
    lines += len(describe(reading).split())
}

print("words logged:", lines)
print("still held:")
for reading in recent {
    print("  " + describe(reading))
}
if hottest is not none {
    print("hottest: " + describe(hottest))
}

# Built with --memory, the program ends with a line like this one:
#
#   memory: 780027 blocks made (13222784 bytes), at most 1048576 bytes in use at once,
    ↪ 12 collections
#
# Thirteen megabytes were asked for. One megabyte at a time was enough.
```

```
$ zeno build --memory memory.zeno > /dev/null
$ ./memory
words logged: 300000
still held:
  tick 49995: 26.5 C (sample 49995)
  tick 49996: 30.2 C (sample 49996)
  tick 49997: 33.9 C (sample 49997)
  tick 49998: 22.5 C (sample 49998)
  tick 49999: 26.2 C (sample 49999)
hottest: tick 102: 35.0 C (sample 102)
memory: 780027 blocks made (13222784 bytes), at most 1048574 bytes in use at once, 10
    ↪ collections, 60002 blocks freed without the collector
```

The last line is the report. The program asked for thirteen megabytes, a few bytes at a time, and never held more than one megabyte or so at once, because the collector ran when it was needed and gave the rest back. The last number counts blocks freed at a line the compiler could point to, without waiting for the collector: the old storage of a list that has just grown, and the copy that a `for` loop walks.

## 29.5 When it runs

Collecting takes time, in proportion to what is still in use, so it is not done on every allocation. The runtime counts the bytes in use and collects when they pass a limit; after each collection the limit becomes twice what survived. A program that keeps little is collected often and cheaply, and one that keeps a lot is left alone for longer. And if `malloc` itself says there is no more memory, the collector runs and the request is tried again. On a board with 64 KB of memory, that second trigger is the one that matters.

## 29.6 What it costs

- **Pauses.** While the collector runs, the program does not, and the pause grows with the memory in use. A control loop that must answer within microseconds should not make blocks inside the loop. Numbers, bools and fixed text never do.
- **A few words for each block**: a header and its line in the table.
- **It leans on the C compiler.** Reading the stack word by word is not something the C standard describes. Every mainstream compiler and processor allows it, and the Boehm collector, Ruby and MicroPython have relied on it for decades.

**Why.** Sharing, with a collector to make it safe, is what Python, Java and JavaScript programmers expect, and it lets a program be written without thinking about memory at all. Counting references cannot free a cycle and costs on every assignment. Rust's ownership everywhere would make every Zeno program pay the hardest part of Rust. So the collector is the default, and the next chapters give firmware the other choices where it needs them. (Decision 5.7.)

## 29.7 Three ways to manage memory

| Build | Memory |
|---|---|
| `zeno build` | shared values and a collector: this chapter |
| `owned` values | one owner, freed where its block ends, in any build: Ownership |
| `--no-collector` | every list and object has one owner, memory comes from a fixed pool, and there is no collector: Building with No Collector |

Memory a program asks for itself with `allocate` is apart from all three: the collector never looks at it, and the program gives it back with `free` (Unsafe Code).

### What this chapter said

Built text, lists, dicts, objects, enum values that carry something and functions that keep variables take memory. A compiled program frees it with a mark-and-sweep collector, which frees whatever no variable can reach, cycles included. It runs when memory in use passes a limit, or when `malloc` runs out. `--memory` reports what a program used. The cost is pauses, a little memory per block, and a habit of C compilers that every one of them keeps.

### Things to try

1. Build `memory.zeno` with `--memory` and change the number of ticks. Which numbers in the report grow, and which do not?
2. Write a loop that builds a text with `+` a thousand times, and build it with `--memory`. Then build the same text once with `join`. Compare the reports.
3. Make two objects that point at each other through fields that may be `none`, drop every variable that leads to them in a loop, and check that the memory in use stays low.

CHAPTER 30

# Ownership

A collector has two costs that firmware notices. It pauses the program when it runs, and memory comes back at no particular moment. A buffer filled a thousand times a second inside a control loop wants neither. So a value in Zeno can be marked `owned`. An owned value has exactly one owner, and is freed the moment its owner's block ends, at a line you can point to.

Shared values stay the default. Nothing changes for a program that does not say `owned`.

## 30.1 Three words

```
owned class Packet {
    id: int
    data: owned list[int]
}

function make(id: int) -> Packet {
    let data: owned list[int] = [id, id * 2, id * 3]
    return Packet(id, data)       # `data` moves into the packet
}

function size(packet: Packet) -> int {      # looks
    return len(packet.data)
}

function send(packet: take Packet) -> int {  # takes
    return packet.id
}

let packet = make(7)
print(size(packet), packet.data)
print(send(packet))               # `packet` moves into `send`
```

```
3 [7, 14, 21]
7
```

- `owned class`: every object of the class has exactly one owner. It is decided once, where the class is declared.
- `owned list[T]`: a list with exactly one owner.
- `take` on a parameter: the function takes the value over. A parameter without it only *looks*: the function may read the value and change what is inside it, and may not keep it.

There are no lifetimes to write, and no `&`.

## 30.2 The rules

**1. One owner.** A variable, a parameter, or a field of an owned object holds the value. Nothing else does.

**2. Storing it gives it away.** Assigning it to another variable, passing it to a `take` parameter, putting it in a field, or returning it moves it. The old name holds nothing afterwards, and using it is an error that says where it went:

```
owned class Packet {
    id: int
}
function send(packet: take Packet) -> int {
    return packet.id
}
let packet = Packet(1)
print(send(packet))
print(packet.id)
```

```
error: `packet` was given away on line 8 and holds nothing now. Use it before it goes,
    ↪ or give it only to what looks: a parameter without `take`
 --> main.zeno:9:7
  |
9 | print(packet.id)
  |       ^

1 error found. Nothing was run.
```

**3. Looking is free.** Reading a field, calling a method, printing, indexing, comparing and looping over it move nothing.

**4. A parameter looks, unless it says take.** What is only looked at cannot be kept, returned or passed on to something that keeps it, so it can never outlive the call:

```
owned class Packet {
    id: int
}
function keep(packet: Packet) -> Packet {
    return packet
}
```

```
error: `packet` is only looked at here: its owner still has it. So it cannot be kept,
    ↪ returned or given away. If it is a parameter, write `take` in front of its type
    ↪ to take the value over
 --> main.zeno:5:12
  |
5 |     return packet
  |            ^

1 error found. Nothing was run.
```

**5. It is freed where its owner's block ends:** at the closing `}`, or at a `return`, `break` or `continue` that leaves the block early. A `var` given a new value frees the old one at that line.

**6. Owned may hold shared; shared may not hold owned.** A shared list can be reached from many places, and if it held an owned value, each of them would think the value was still there after its owner had freed it. So a list of owned objects must itself be owned:

```
owned class Packet {
    id: int
}
let queue: list[Packet] = []
```

```
error: each item of this list is an owned Packet, which has one owner: the list. So the
    ↪ list needs an owner too. Write `owned list[...]`
 --> main.zeno:4:1
  |
4 | let queue: list[Packet] = []
  | ^

1 error found. Nothing was run.
```

```
owned class Packet {
    id: int
}
var queue: owned list[Packet] = []
queue.append(Packet(1))
queue.append(Packet(2))
let first = queue.pop(0)          # taken out: `first` owns it now
print(first, queue)
```

```
Packet(id=1) [Packet(id=2)]
```

An owned list owns its items, and frees them with itself. A `Packet or none` can own a packet or nothing.

## 30.3 Branches and loops

The checker follows each path. After an `if`, a value is gone if any branch gave it away, unless that branch left with `return`:

```
if not ok {
    discard(frame)
    return
}
send(frame)          # fine: only reached with the frame still here
```

A value given away on one trip round a loop would not be there on the next, so an owner from outside a loop cannot be given away inside it, except just before a `break`:

```
owned class Packet {
    id: int
}
function send(packet: take Packet) {
    print("sent", packet.id)
}
let packet = Packet(1)
for turn in range(3) {
    send(packet)
}
```

```
error: `packet` is given away inside a loop, and the loop may come round again with
    ↪ nothing in it. Give it away after the loop, or just before a `break`
 --> main.zeno:9:10
  |
9 |     send(packet)
  |          ^

1 error found. Nothing was run.
```

In a `try`, an error may leave the body after any line, so the `except` starts with everything the body gives away already gone.

## 30.4 Seeing the frees

This program from the `examples` folder makes 120 frames of 600 samples each, sends some and skips others. Every list and object in it is owned:

buffer.zeno

```
# Owned values: memory that is given back at a line you can point to.
#
#     cargo run -- run examples/buffer.zeno
#     cargo run -- build --memory examples/buffer.zeno && ./buffer
#     cargo run -- build --no-collector --memory examples/buffer.zeno && ./buffer
#
# Most values in Zeno are shared, and a garbage collector frees them some
# time after their last use (docs/memory.md). A value marked `owned` has
# exactly one owner instead, and is freed the moment its owner's block
# ends. docs/ownership.md explains the rules. This file is the short tour.
#
# Every list and object here is owned, so the third command above can
# build the program with no collector in it at all (docs/no-collector.md).

# Every Frame has one owner. When that owner's block ends, the frame and
# the list inside it are freed, there and then.
owned class Frame {
    channel: int
    samples: owned list[int]

    # A method only looks at the object it is called on.
    function peak(self) -> int {
        var highest = 0
        for sample in self.samples {
            if sample > highest {
                highest = sample
            }
        }
        return highest
    }
}

# An ordinary, shared class. An error to raise.
class Overload {
    channel: int
    peak: int
}
```

```
# Makes a frame and hands it to the caller, who owns it from then on.
function capture(channel: int, size: int) -> Frame {
    let samples: owned list[int] = [(n * n + channel * 13) % (400 + channel) for n in
    ↪ range(size)]
    return Frame(channel, samples)    # `samples` moves into the frame
}

# A parameter looks at what it is given and cannot keep it. Any list
# will do here, owned or shared, and the caller still has it afterwards.
function average(samples: owned list[int]) -> int {
    return sum(samples) // len(samples)
}

# `take`: `send` takes the frame over. The caller's variable is empty
# after the call, and the frame is freed when `send` is done with it.
function send(frame: take Frame) -> int raises Overload {
    if frame.peak() > 480 {
        raise Overload(frame.channel, frame.peak())
    }
    return len(frame.samples)
}

var sent = 0
var quiet = 0
var refused: owned list[int] = []

for channel in range(120) {
    let frame = capture(channel, 600)
    if average(frame.samples) < 205 {
        quiet += 1
        continue                      # `frame` is freed here,
    }
    try {
        sent += send(frame)           # or it moves into `send`, which frees it.
    } except Overload as overload {
        refused.append(overload.channel)
    }
}

print(f"sent {sent} samples, skipped {quiet} quiet frames")
print(f"refused {len(refused)} frames, the first on channel {refused[0]}")

# A `var` that owns its value can be given a new one. The old one is freed
# at that line.
var window: owned list[int] = [0, 0, 0, 0]
for round in range(3) {
    window = [value + round for value in window]
}
print(window)

# After a move the old name holds nothing, and the checker says so. Take
# the `#` off the last line to see the message:
let last = capture(7, 4)
print(send(last))
# print(last.channel)
```

```
$ zeno build --memory buffer.zeno > /dev/null
$ ./buffer
```

```
sent 44400 samples, skipped 13 quiet frames
refused 33 frames, the first on channel 84
[3, 3, 3, 3]
4
memory: 1806 blocks made (2646705 bytes), at most 15870 bytes in use at once, 0
    ↪ collections, 1625 blocks freed without the collector
```

It made megabytes of buffers, never held more than a few kilobytes, and the collector never ran: every block was freed at a line, the moment its owner was done.

**Why.** Rust showed that one owner, checked by the compiler, gives memory back at exact moments with no collector and no mistakes. It also showed that borrowing everywhere, with lifetimes written in the program, is the hardest thing in Rust to learn. Zeno takes the first half. A parameter looks, which needs nothing written, and only keeping needs a word. No lifetimes are needed because a look can never outlive the call. Owned values are a choice made where it pays, and shared values stay the easy default. (Decisions 5.8 and 5.15, and the audit, which made looking the default.)

## 30.5 What owned values cannot do

In version 0.1 an owned value cannot sit in a function type, be taken out of a field (that would leave a hole the type does not admit to), be replaced through a look, or be raised. Each is an error that says what to do instead, and allowing any of them later breaks no program.

### What this chapter said

`owned class` and `owned list[T]` have one owner, and are freed where the owner's block ends. Storing gives a value away; looking is free. A parameter looks unless it says `take`. Owned may hold shared, never the other way. The checker follows branches, loops and `try`, and says where a value went when it is used after.

### Things to try

1. Change `size` above to return the packet's list. Read the message, and say in your own words why it is right.
2. Write a queue of owned packets with `put` and `get` methods, where `get` gives `Packet or none`.
3. In `buffer.zeno`, remove `take` from `send`. What changes in the report?
4. Give an owned value away inside a loop and then `break` on the next line. Why does the checker accept that?

CHAPTER 31

# Firmware

Firmware is a program that runs on a board with no operating system under it: it starts when the power comes on and runs until it goes off. It is held to rules a desktop program is not, and often to a written coding standard such as MISRA C. `zeno build --firmware` writes C for that world. This chapter is about what it changes, how to put a real board under the program, and what it does not yet claim.

## 31.1 Errors as values

An ordinary build compiles `try` with C's `setjmp` and `longjmp`: a `raise` jumps straight back to the `try`, out of however many functions are in between. That is cheap, and it is exactly what standards for safety-critical firmware forbid. MISRA C bans `<setjmp.h>` outright, because the jump skips what the functions in between still meant to do, can leave local variables with no defined value, and hides paths from the tools that measure stack depth and test coverage.

A firmware build carries errors as ordinary return values instead. The Zeno does not change:

speed.zeno

```
function parse_speed(text: str) -> int raises ValueError {
    let n = int(text)
    if n < 0 or n > 100 {
        raise ValueError(f"a speed is 0 to 100, and {n} is not")
    }
    return n
}

for text in ["40", "fast", "250"] {
    try {
        print("speed", parse_speed(text))
    } except ValueError as e {
        print("refused:", e.message)
    }
}
```

```
$ zeno build --firmware speed.zeno
Wrote speed.c
Built speed. Run it with ./speed
$ ./speed
speed 40
refused: cannot turn "fast" into an int
refused: a speed is 0 to 100, and 250 is not
$ grep setjmp speed.c | wc -l
0
```

It prints what `zeno run` prints, and there is no `setjmp` in the C (`grep` found 0 lines). Here is what `parse_speed` became:

```
$ zeno build --firmware speed.zeno > /dev/null
$ grep -A11 "^static bool z_parse_speed.*{$" speed.c
static bool z_parse_speed(const char *z2_text, int64_t *zeno_result) {
    int64_t t4;
    if (!zeno_parse_int(z2_text, 2, 13, &t4)) return false;
    int64_t z_n = t4;
    if ((z_n < 0) || (z_n > 100)) {
        zeno_note(zeno_error_new("ValueError", zeno_join(3, "a speed is 0 to 100, and ",
    ↪  zeno_str_int(z_n), " is not"), NULL), 4, 9);
        return false;
    }
    *zeno_result = z_n;
    return true;
}
```

Three things changed. The function returns `true` when it worked and `false` when it raised, and its answer goes out through one more parameter. A `raise` notes the error, with its line and column, and returns `false`. And a call to something that can raise, `int(text)` here, is followed at once by a test. Only the functions that say `raises` are compiled this way; the rest are plain C, with no tests after calls to them. That is why `raises` must be written in a build (When Things Go Wrong).

## 31.2 Faults

Zeno has two sorts of error that `zeno run` can catch:

- errors **the program raises itself**: `raise`, and `int` or `float` of text;
- errors **the runtime finds**: a division by zero, a position past the end, a key that is not there, an overflow, `pop` on an empty list.

A firmware build carries the first sort as values. The second sort is a *fault*: the program stops there, with the message `zeno run` gives for an uncaught error, and no `except` sees it. To carry a division by zero as a value, every function would have to be one that can fail, and nearly every line would need a test after it. The firmware answer is to test first:

```
if divisor != 0 {
    rate = total // divisor
}
```

The compiler does not leave this to be found when the board resets. A `try` that hopes to catch a fault is refused before anything is built:

divide.zeno

```
let readings = [4, 0]
for n in readings {
    try {
        print(100 // n)
    } except ZeroDivisionError {
        print("no reading")
    }
}
```

```
$ zeno build --firmware divide.zeno
error: nothing in this `try` can be caught. In a firmware build, an error the program
    ↪ raises itself (and `int` or `float` of text) travels as a value and can be caught.
```

```
    ↪ A division by zero, a bad index, a key that is not in a dict or an overflow
    ↪ stops the program. Test for it before it happens, or build without `--firmware`
 --> divide.zeno:3:5
  |
3 |     try {
  |     ^
```

A `FileError` is the exception to the exception: no check the program could make rules out a missing file, so it is always a value, and can always be caught.

> **Why.** The line between the two sorts is the line between what the program can check beforehand and what it cannot. A bad index can be ruled out by testing the length; a missing file cannot. Firmware held to a standard is expected to test, and the build makes the cases it did not test visible. (Decisions 4.1 and 5.5l.)

## 31.3 A real board underneath

The board module meets the hardware in four C functions (The Board). For a real board, write them in a header over the vendor's library, and tell the C compiler where it is with `-DZENO_BOARD`:

```
$ zeno build --firmware --no-collector blink.zeno
$ cc -O2 -DZENO_BOARD='"my_board.h"' -o blink blink.c -lm
```

With ST's HAL for an STM32, where the LED of a Nucleo board is on port A pin 5, the header would look like this. It is a sketch, and has not been run on hardware:

```
#define ZENO_PINS 16

static void zeno_board_write(int pin, bool level) {
    if (pin == ZENO_LED) pin = 5;
    HAL_GPIO_WritePin(GPIOA, (uint16_t)(1u << pin),
                      level ? GPIO_PIN_SET : GPIO_PIN_RESET);
}
static bool zeno_board_read(int pin) {
    if (pin == ZENO_LED) pin = 5;
    return HAL_GPIO_ReadPin(GPIOA, (uint16_t)(1u << pin)) == GPIO_PIN_SET;
}
static void zeno_board_sleep(double seconds) {
    HAL_Delay((uint32_t)(seconds * 1000.0));
}
static double zeno_board_now(void) { return (double)HAL_GetTick(); }
```

To the Zeno program the LED is always pin 13, and the board is free to wire that number to wherever its LED really is. The same flag works for an LLVM build, on the `clang` line.

A board whose clock moves only when the program sleeps makes a program that waits run in no time. That is how firmware is tested on a desk, and Zeno's own tests do exactly that.

## 31.4 What the C of blink looks like

All of `blink`'s program, in a firmware build with no collector, is three lines of C:

```
while (true) {
    zeno_pin_toggle(ZENO_LED);
    zeno_sleep((500.0 / 1000.0));
```

```
}
```

A pin is a number, `500 ms` has become seconds, and nothing in it takes memory or can raise.

## 31.5 What this does not claim

Removing `setjmp` removes one obstacle to MISRA, not all of them. The C that a firmware build writes in version 0.1 still uses variable argument lists in the runtime, calls `printf` and `exit`, and uses GCC and Clang built-ins for checked arithmetic. Taking memory from the heap, which MISRA also bans, is answered by `--no-collector`, in the next chapter. None of these is hidden, and each is a later piece of work. A function that can raise cannot yet be passed around as a value in a firmware build, because a function type does not say whether it raises.

### What this chapter said

`zeno build --firmware` carries errors the program raises as return values, tested after each call, with no `setjmp`. Errors the runtime finds are faults that stop the program, and a `try` that hopes to catch one is refused. A real board goes under the program as a header of four functions, named with `-DZENO_BOARD`.

### Things to try

1. Build `speed.zeno` with and without `--firmware`, and compare the C of `parse_speed` in the two.
2. Rewrite the `divide.zeno` example so that a firmware build accepts it.
3. Write a board header of your own for the simulated board that prints `HIGH` and `LOW`, and build `blink` with it.

CHAPTER 32

# Building with No Collector

`zeno build --no-collector` compiles a program to C that has no garbage collector in it. Not a collector that is switched off: the code is not in the file. Nor is `malloc`: memory comes from a pool of a fixed size that is part of the program. Every block the program takes is given back at a line that can be found in the C. This is the build for a microcontroller that must run for years.

## 32.1 Why

The collector is the right default (Memory). It also has three habits that firmware cannot always live with:

- it pauses the program, for a time that depends on how much memory is in use, and a control loop with a deadline cannot plan for that;
- it reads the C stack to find what is in use, which no C standard describes;
- nobody can say when a block will be freed, so nobody can say how much memory the program needs at worst.

Owned values answer all three for the values that are owned (Ownership). This build is what you get when every value is.

## 32.2 Every list and object has one owner

With no collector, there is nothing to free a shared value with. So the build follows one rule by itself:

> **With no collector, every list and every object has one owner: whoever received it.**

In this build `list[int]` means `owned list[int]`, and `class Task` means `owned class Task`, with no word written. The rules of ownership apply to all of them, and the one that matters day to day is that giving a list to another name moves it:

moved.zeno

```
let a = [1, 2, 3]
let b = a
print(a)
```

```
$ zeno check --no-collector moved.zeno
error: `a` was given away on line 2 and holds nothing now. Use it before it goes, or
    ↪ give it only to what looks: a parameter without `take`
 --> moved.zeno:3:7
  |
3 | print(a)
```

```
  |       ^

1 error found. Nothing was run.
```

`zeno check --no-collector` checks as the build would, without building. A parameter only looks, unless it says `take`, so passing a list to an ordinary function moves nothing.

> **Why.** The same file must mean the same thing under `zeno run`, `zeno build` and `zeno build --no-collector`. This build adds refusals and nothing else: a program it accepts never gives a value two owners, and for such a program "one owner" and "shared" cannot be told apart from what it prints. Fewer programs, never different ones. (Decision 5.15.)

## 32.3 An ordinary program, built so

Most programs need no change at all:

names.zeno

```
let names = "ada,grace,alan".split(",")
var lengths: list[int] = []
for name in names {
    lengths.append(len(name))
}
print(names, lengths)
print(f"the longest is {max(lengths)} letters")
```

```
$ zeno build --no-collector --memory names.zeno
Wrote names.c
Built names. Run it with ./names
$ ./names
["ada", "grace", "alan"] [3, 5, 4]
the longest is 5 letters
memory: 12 blocks made (190 bytes), at most 190 bytes in use at once, no collector, 0
    ↪ blocks left at the end
pool: 384 bytes taken, in a pool of 32768
text: at most 49 bytes in use at once, in a room of 1024
```

The report has three lines now. The first says how many blocks were made and that none was left at the end. The second is the pool, and the third the text room, both below.

## 32.4 Where memory comes from

| What | Lives in | Given back |
|---|---|---|
| a list or an object | the pool | when its owner's block ends |
| an enum value that carries something | the pool | when its owner's block ends |
| a list made and given to nobody | the pool | when its statement is over |
| text built while the program runs | the text room | when its statement is over |
| text that is kept | the pool | when its keeper is done with it |
| an error | the pool | when the `except` that caught it ends |
| `allocate` | the pool | at `free`, and not before |

Numbers, bools, quantities, enum cases with nothing in them, and text written in the program need no memory at all.

### 32.4.1 The pool

The pool is one array whose size is fixed when the C is compiled, 32768 bytes unless you say otherwise. It hands out blocks of a few fixed sizes, and keeps a list of spare blocks of each size, so a block given back is used again for the next of its size and the pool cannot break up into pieces too small to use. The report says how much of the pool a run used, exactly, and that is the number to give the C compiler for the board:

```
$ zeno build --no-collector --memory buffer.zeno
$ ./buffer | grep pool
pool: 25384 bytes taken, in a pool of 32768
$ cc -O2 -DZENO_POOL=25384 -o buffer buffer.c -lm
```

It is exact for that run. A program whose lists depend on its input needs the report from its worst input, and some room on top. A full pool is a fault: the program stops with `the memory pool is full`.

### 32.4.2 The text room

Firmware prints, and `print(f"sent {count} samples")` builds text that must live somewhere for a moment. It lives in the *text room*, one more fixed array. Building text writes at the top of the room; when the statement is over, the top goes back to where it was. So a statement takes text room only for as long as it runs, and the report says the most any statement needed. Text that is kept, in a variable, a field or a list, is copied into the pool and kept there.

## 32.5 What is refused

Each of these would make memory that no line gives back, so the build refuses it, with the line and what to write instead:

- a list, object or enum value used after it was given away;
- a function that uses a `var` of the function around it, because the two would share it;
- keeping a function that uses a list or object from outside it;
- a list or object as what a type parameter stands for, and a list of `int or none`, which are limits of version 0.1.

items.zeno

```
var readings: list[int or none] = [1, none, 3]
```

```
$ zeno build --no-collector items.zeno
error: a list of numbers that may be `none` keeps each item in a small block of its own,
    ↪  and `zeno build --no-collector` leaves the collector out, so nothing would free
    ↪ them. Make it a list of an enum that has one case for a missing value and one
    ↪ case that carries the number, say `Missing` and `Value(n: int)`. docs/no-
    ↪ collector.md lists what such a build accepts
 --> items.zeno:1:35
  |
1 | var readings: list[int or none] = [1, none, 3]
  |                                   ^
```

The answer the message gives is an enum with a case for the missing value, which costs nothing for the case that carries nothing.

That is the whole idea of the build. It does not find leaks while the program runs. It refuses to build a program that could have one.

## 32.6 The tour

`buffer.zeno` from Ownership, built with no collector:

```
$ zeno build --no-collector --memory buffer.zeno > /dev/null
$ ./buffer
sent 44400 samples, skipped 13 quiet frames
refused 33 frames, the first on channel 84
[3, 3, 3, 3]
4
memory: 1691 blocks made (2646563 bytes), at most 13336 bytes in use at once, no
    ↪ collector, 0 blocks left at the end
pool: 25384 bytes taken, in a pool of 32768
text: at most 53 bytes in use at once, in a room of 1024
```

It made over 1600 blocks and gave every one back. Search the C for `zeno_collect`, `setjmp` or `malloc`: none is there. `--no-collector` turns `--firmware` on by itself.

## 32.7 What this does not claim

Memory is still taken and given back while the program runs, from a fixed pool and not from the heap. That meets MISRA's rule against `malloc` and `free`, though not its wider advice against dynamic memory of any kind. The build does not bound memory for you: a loop that appends to a list for ever fills the pool. The rest of the list in Firmware stands.

### What this chapter said

`--no-collector` leaves the collector and `malloc` out. Every list and object has one owner, whoever received it, and is freed where that owner's block ends. Memory comes from a fixed pool, sized with `-DZENO_POOL` from the `--memory` report, and built text from a text room that each statement gives back. Whatever could leak is refused at build time.

### Things to try

1. Build a program from an earlier chapter with `--no-collector`. What does it refuse, and how do you fix it?
2. Build `buffer.zeno` with the pool size from its report, then with one byte less, and run each.
3. Replace the `list[int or none]` above with an enum `Reading` with a case `Missing`, and build it.

CHAPTER 33

# Unsafe Code

Zeno looks after memory in two ways: a collector, or one owner for every value. This chapter is about the third way, the one C has: ask for memory, use it through a *pointer*, and give it back yourself. It is for the places where firmware must control every byte, such as a buffer that a peripheral fills, and it lives inside `unsafe { }`, a block that marks the lines Zeno cannot check.

## 33.1 The block

```
let bytes = allocate(u8, 16)

unsafe {
    bytes[0] = 0x41
    bytes[1] = u8(bytes[0] + 1)
    print(bytes[0], bytes[1])
    free(bytes)
}
```

```
65 66
```

`unsafe { }` is a block like the body of an `if`. Inside it, and only there, a program may do five things:

| Inside unsafe only | Why Zeno cannot check it |
|---|---|
| read through a pointer, `p[i]` | a pointer does not know how long its memory is, or whether it was freed |
| write through a pointer, `p[i] = v` | the same, and a wrong write damages something else |
| `free(p)` | nothing says whether `p` is still used somewhere |
| read a register, `r.read()` | the address is taken on trust, and reading can change the hardware |
| write a register, `r.write(v)` | the same |

Outside the block, the checker says so, and why:

```
let bytes = allocate(u8, 16)
print(bytes[0])
```

```
error: reading through a pointer goes inside `unsafe { }`. Zeno cannot check that it is
    ↪ right, so the block marks the lines where a mistake can damage memory
 --> main.zeno:2:7
  |
```

```
2 | print(bytes[0])
  |       ^

1 error found. Nothing was run.
```

The block does not turn anything off. Types are still checked inside it, numbers still report overflow, and lists are still looked after. It only allows those five things. So when a program damages memory, the lines to read are the ones inside the blocks, and `grep unsafe` finds them all.

> **Why.** Rust showed that a language can be safe by default and still let a driver do what drivers do, if the unsafe lines are marked and few. The mark is a block and not a word on a function, so it can be as small as one line. It gives no value, so `let x = unsafe { p[0] }` is refused: the block means one thing, "these lines are trusted".

## 33.2 Pointers

A pointer's type says what one item is: `pointer[u8]` points at bytes. The item is a number of one size: `u8` or `i8` (1 byte), `u16` or `i16` (2), `u32` or `i32` (4), or `int` or `float` (8).

`allocate(u8, 16)` takes the item first and then how many, and gives a `pointer[u8]` to sixteen bytes that are all zero. `p[i]` is item `i`, counted from 0. `p + 1` is the address of the next *item*, not the next byte, as in C. `free(p)` gives the memory back, and wants the pointer that `allocate` gave, not one worked out from it.

Everything else about a pointer is safe, and needs no block:

```
let bytes = allocate(u8, 16)
let second = bytes + 1
print(second - 1 == bytes, bytes)
let all = [bytes, second]
unsafe {
    free(bytes)
}
```

```
true <pointer[u8]>
```

An address that points nowhere is harmless until somebody reads or writes through it.

## 33.3 What zeno run catches

A compiled program holds an address and nothing more, as a C program does. `zeno run` holds more: behind each pointer it keeps the block, its length, and whether it was freed. So it catches the four classic mistakes at the line that makes them, and says what a compiled program would have done instead:

```
let a = allocate(u8, 16)
let b = allocate(u8, 16)
unsafe {
    a[24] = 7
    print(b[0])
}
```

```
error: this writes over item 24 of a block that has 16 (0 to 15). `zeno run` checks
    ↪ every use of a pointer. A compiled program does not: it writes over whatever is
```

```
    ↪ beside the block
 --> main.zeno:4:5
  |
4 |     a[24] = 7
  |     ^
```

```
let a = allocate(u8, 4)
unsafe {
    free(a)
    print(a[0])
}
```

```
error: this reads memory that was freed. `zeno run` checks every use of a pointer. A
    ↪ compiled program does not: it reads whatever is there now, which may be somebody
    ↪ else's
 --> main.zeno:4:11
  |
4 |     print(a[0])
  |           ^
```

```
let a = allocate(u8, 4)
unsafe {
    free(a)
    free(a)
}
```

```
error: this memory was freed already. `zeno run` checks every `free`. A compiled program
    ↪  does not: freeing twice damages the memory it hands out next
 --> main.zeno:4:5
  |
4 |     free(a)
  |     ^
```

Built with `--no-collector`, the first program runs to its end and prints 7: in the pool, `b` lies just after `a`, so item 24 of `a` is item 0 of `b`. Nothing went wrong that the machine could see. That is the whole danger, and the reason to run a program under `zeno run` first.

## 33.4 Registers

A register (The Board) is made with its size and address, `register(u32, 0x4000_4400)`, and its type is `Register[u32]`. Making one and passing it around is safe; `read()` and `write(v)` go inside the block. On a real board a write is one line of C through a `volatile` pointer, which tells the C compiler that somebody else is watching this memory: it may not skip the access, repeat it or move it.

## 33.5 A driver

This program from the `examples` folder is a driver for a made-up serial port. Bytes wait in a queue whose memory the program asks for and gives back by hand, and go out through three registers. Each line that Zeno cannot check is inside `unsafe`:

examples/driver.zeno

```
# driver.zeno: memory by hand and registers, the way a C driver does it.
```

```
#
#     zeno run examples/driver.zeno          every use of a pointer is checked
#     zeno build --memory examples/driver.zeno && ./driver
#     zeno build --no-collector --memory examples/driver.zeno && ./driver
#
# A driver for a made-up serial port. Bytes wait in a queue whose memory
# the program asks for and gives back itself, and they go out through
# three registers at fixed addresses.
#
# Zeno cannot check a line that reads or writes through a pointer, frees
# one, or touches a register. Each such line is inside `unsafe { }`, so
# those are the blocks to read twice. docs/unsafe.md walks through it.

from board import register

# The port's registers, at the addresses its data sheet would give.
let control = register(u32, 0x4000_4400)    # bit 0 turns the port on, bit 3 the sender
let divider = register(u16, 0x4000_4408)    # clock ticks for each bit
let data = register(u8, 0x4000_440C)        # a byte written here is sent

let PORT_ON = 1 << 0
let SENDER_ON = 1 << 3

# Bytes that wait to be sent: first in, first out.
class Queue {
    bytes: pointer[u8]    # `size` bytes, asked for with `allocate`
    size: int
    first: int            # where the oldest byte is
    count: int            # how many are waiting

    # Add a byte at the end. `false` when there is no room.
    function put(self, byte: u8) -> bool {
        if self.count == self.size {
            return false
        }
        let at = (self.first + self.count) % self.size
        unsafe { self.bytes[at] = byte }
        self.count += 1
        return true
    }

    # Take the oldest byte out. `none` when nothing waits.
    function get(self) -> u8 or none {
        if self.count == 0 {
            return none
        }
        let at = self.first
        self.first = (self.first + 1) % self.size
        self.count -= 1
        unsafe { return self.bytes[at] }
    }

    # Give the memory back. The queue must not be used after this.
    function close(self) {
        unsafe { free(self.bytes) }
        self.size = 0
        self.count = 0
    }
```

```
}

function open_queue(size: int) -> Queue {
    return Queue(allocate(u8, size), size, 0, 0)
}

# Turn the port on, keeping every other bit of `control` as it was.
function start(ticks: u16) {
    unsafe {
        divider.write(ticks)
        control.write(u32(control.read() | PORT_ON | SENDER_ON))
    }
}

function stop() {
    unsafe {
        control.write(u32(control.read() & ~(PORT_ON | SENDER_ON)))
    }
}

# Send everything that waits. The answer is how many bytes went out.
function flush(queue: Queue) -> int {
    var sent = 0
    while true {
        let byte = queue.get()
        if byte is none {
            return sent
        }
        unsafe { data.write(byte) }
        sent += 1
    }
    return sent
}

# The sum of `count` bytes from `bytes` on, kept to eight bits.
function checksum(bytes: pointer[u8], count: int) -> u8 {
    var sum = 0
    unsafe {
        for i in range(count) {
            sum = (sum + bytes[i]) & 0xFF
        }
    }
    return u8(sum)
}

let queue = open_queue(4)
start(104)

# Six bytes into a queue of four: the last two do not fit.
var refused = 0
for byte in [0x5A, 0x65, 0x6E, 0x6F, 0x21, 0x0A] {
    if not queue.put(u8(byte)) {
        refused += 1
    }
}
print("waiting:", queue.count, "refused:", refused)

# `queue.bytes + 1` is the same memory, one byte further on. Working out
```

```
# an address is safe. Only reading through it is not.
print("checksum of all four:", checksum(queue.bytes, 4))
print("checksum of the last three:", checksum(queue.bytes + 1, 3))

print("sent:", flush(queue))

# The queue goes round: two more bytes land where the first two were.
queue.put(0x21)
queue.put(0x0A)
print("sent:", flush(queue))

stop()
queue.close()
print(control, data, queue.bytes)
```

```
register 0x40004408 = 0x0068
register 0x40004400 = 0x00000009
waiting: 4 refused: 2
checksum of all four: 156
checksum of the last three: 66
register 0x4000440c = 0x5a
register 0x4000440c = 0x65
register 0x4000440c = 0x6e
register 0x4000440c = 0x6f
sent: 4
register 0x4000440c = 0x21
register 0x4000440c = 0x0a
sent: 2
register 0x40004400 = 0x00000000
Register(0x40004400) Register(0x4000440c) <pointer[u8]>
```

`zeno run`, the VM, and every kind of compiled build print these same lines.

## 33.6 Memory by hand, and the other two

Memory from `allocate` is apart from the collector and from ownership: the collector never looks at it, and no block's end frees it. In a build with no collector it comes from the pool, like everything else. A list, in the same program, is still looked after for you. Only what was asked for with `allocate` is yours to free.

### What this chapter said

`allocate(T, n)` gives a `pointer[T]` to `n` zeroed items, and `free(p)` gives it back. Reading and writing through a pointer, `free`, and reading and writing a register go inside `unsafe { }`, which allows those and changes nothing else. Working out, comparing and keeping addresses is safe. `zeno run` catches out-of-bounds use, use after free, double frees and freeing a moved pointer; a compiled program does not.

### Things to try

1. Allocate ten `u16`s, fill them with the squares of 0 to 9, and print them. What happens with the square of 300?
2. Write a function that copies `n` items from one pointer to another, and test it under

`zeno run` with an `n` that is too big.

3. In `driver.zeno`, find each `unsafe` block and say what could go wrong in it.
4. Take a pointer one item on with `+ 1` and `free` it. Read the message.

PART V

# Inside Zeno

CHAPTER 34

# Inside Zeno

Zeno was built to learn how a language is built, so the compiler is written to be read. It is a Rust program of about fifty thousand lines, with no libraries: the lexer, the parser, the checker, two engines that run programs and two that compile them are all written by hand. This chapter follows one small program through every stage, with the command that shows each.

square.zeno

```
function square(n: int) -> int {
    return n * n
}
print(square(7))
```

```
49
```

## 34.1 The stages

```
files -> lexer -> parser -> modules -> checker -+-> interpreter     zeno run
                                                +-> bytecode -> VM  zeno run --vm
                                                +-> C -> cc         zeno build
                                                +-> LLVM IR -> clang  zeno build --llvm
```

Each stage takes what the one before it made, and every stage after the parser works on the same syntax tree.

## 34.2 The lexer

The lexer turns the characters of a file into *tokens*, the words of the language. It looks at one character, and that is enough to decide what kind of token starts there: a digit starts a number, a quote a string, a letter a name or a keyword, # a comment.

```
$ zeno tokens square.zeno | head -12
   1:1   Function
   1:10  Ident("square")
   1:16  LParen
   1:17  Ident("n")
   1:18  Colon
   1:20  Ident("int")
   1:23  RParen
   1:25  Arrow
   1:28  Ident("int")
   1:32  LBrace
   1:33  Newline
   2:5   Return
```

A line end becomes a `Newline` token only where a statement can end; after `+` or inside brackets it is skipped, which is how a long statement goes on to the next line. The lexer reports every mistake it finds, not just the first, because after a bad character it can simply skip it and carry on. (`src/lexer.rs`, about 700 lines.)

## 34.3 The parser

The parser turns tokens into a *syntax tree*: the program's shape, with each expression inside the one it belongs to. It is a *recursive descent* parser: one function for each kind of thing in the language (a statement, an `if`, an expression), each calling the others for the parts inside it. Precedence comes from which function calls which: the function for `+` calls the function for `*` to read each side, so `*` binds tighter without a table.

```
$ zeno parse square.zeno
function square(n: int) -> int
    return (n * n)
print(square(7))
```

`zeno parse` prints the tree back as text, with brackets that show how it grouped every expression. The parser stops at the first mistake, because after a syntax error it no longer knows where it is, and what follows is usually nonsense caused by the first. (`src/parser.rs`, about 2100 lines.)

## 34.4 Modules

The next stage loads every file the program imports, renames each top-level name with its module in front, and joins them into one program (Programs in Several Files). Every stage after it sees one tree, as if the program had been written in one file, so none of them had to learn about files. `zeno join` shows the result.

## 34.5 The checker

The checker walks the tree and works out a type for every expression. It is the biggest stage, nearly ten thousand lines, because it is where most of the language's promises are kept: types, `none`, units, sized numbers, `match` covering every case, `raises`, owned values, `private`, interfaces and generics. It reports every mistake it is sure of, and marks a faulty expression so that nothing built on it is reported again.

It makes several passes, because types flow backwards as well as forwards: a parameter's type comes from the calls, which may be below the function. Each pass notes what arrived at every untyped parameter, and the next starts from that, until a pass learns nothing new.

## 34.6 Two engines that run

**The interpreter** walks the tree. For `n * n` it works out the left, then the right, then multiplies. It is the simplest way to run a program, and the default for `zeno run`.

**The VM** compiles the tree to *bytecode*, instructions for a small stack machine, and runs those. `n * n` becomes "push `n`, push `n`, multiply":

```
$ zeno bytecode square.zeno
```

```
== main() ==
0000    1  Closure      0          ; square
0001    4  Const        0          ; <function print>
0002    |  GetLocal     0          ; square
0003    |  Const        1          ; 7
0004    |  Call         1
0005    |  Call         1
0006    |  Pop          1
0007    |  Const        2          ; none
0008    |  Return

== square(n) ==
0000    1  Required     0 0        ; "`square` needs a value for `n`"
0001    |  Enter
0002    2  GetLocal     0          ; n
0003    |  GetLocal     0          ; n
0004    |  Binary       *
0005    |  Return
0006    1  Const        1          ; none
0007    |  Return
```

Both engines share one file, `ops.rs`, for what every operator does, so what `*` means is written once. The VM is about twice as fast, and every test program runs on both and must print the same.

## 34.7 Two engines that compile

**The C backend** writes C, as Compiling showed. `square` becomes:

```
$ zeno build square.zeno > /dev/null
$ grep -A3 "^static int64_t z_square(int64_t z_n) {" square.c
static int64_t z_square(int64_t z_n) {
    return zeno_mul(z_n, z_n, 2, 12);
}
```

**The LLVM backend** writes LLVM's intermediate language as text, with no LLVM library. Here the overflow check is spelled out: LLVM's own multiply-with-overflow gives the answer and a flag, and a branch goes to Zeno's overflow error with the line and column:

```
$ zeno build --llvm square.zeno > /dev/null
$ grep -A13 "define internal i64 @z_square" square.ll
define internal i64 @z_square(i64 %arg.n) {
entry:
  %t1 = call { i64, i1 } @llvm.smul.with.overflow.i64(i64 %arg.n, i64 %arg.n)
  %t2 = extractvalue { i64, i1 } %t1, 0
  %t3 = extractvalue { i64, i1 } %t1, 1
  br i1 %t3, label %overflow1, label %fits1

overflow1:
  call void @zeno_overflow(i32 2, i32 12)
  unreachable

fits1:
  ret i64 %t2
}
```

Both backends share the C runtime, `runtime.c`, which every compiled program carries: text,

lists, printing, errors, the collector, the pool, and the board.

## 34.8 Agreement

Four ways of running one program could easily drift into four languages. They do not, because of one rule enforced by the tests: every test program is run by the interpreter, the VM, the C build and the LLVM build, in the build kinds that apply, and the outputs must be the same, byte for byte, error messages included. When a stage is changed, the tests show at once which of the others must change with it.

## 34.9 The files

| File | What it is |
|---|---|
| `main.rs` | the `zeno` command |
| `token.rs`, `lexer.rs` | characters to tokens |
| `ast.rs`, `parser.rs` | tokens to a syntax tree |
| `printer.rs` | the tree back to text, for `zeno parse` |
| `modules.rs`, `package.rs` | files and packages to one tree |
| `checker.rs` | types, and every check before running |
| `value.rs`, `ops.rs`, `builtins.rs` | values and their operations, for both engines |
| `interpreter.rs` | runs the tree |
| `bytecode.rs`, `emit_bytecode.rs`, `vm.rs` | the VM |
| `emit_c.rs`, `runtime.c` | the C backend and its runtime |
| `emit_llvm.rs` | the LLVM backend |
| `format.rs`, `lsp.rs`, `json.rs` | the formatter and the language server |
| `board.rs`, `units.rs`, `ints.rs` | the simulated board, units, sized numbers |

The `docs` folder of the repository has a walkthrough of each stage, written to be read beside the code, and each ends with things to try. They are the next book after this one.

**Why.** No libraries, so that every part of the compiler can be read and understood by one person, which was the reason for building it. An interpreter first, because it is the shortest way to a language that runs, and then C, because C reaches every microcontroller. The interpreter stayed, as the reference every other engine is held to. (Decisions 5.1 to 5.5.)

### What this chapter said

A program goes through the lexer, the parser, the module loader and the checker, and then to one of four engines: the interpreter, the VM, C or LLVM. `zeno tokens`, `parse`, `join` and `bytecode`, and the kept `.c` and `.ll` files, show each stage. Every engine must print the same for every test program.

### Things to try

1. Run `zeno tokens` on a file with a mistake, such as a `$` in the middle of a line. How many errors does the lexer give for two such characters?

2. Compare `zeno bytecode` for a `while` loop and for a `for` loop over `range`.
3. Read `docs/lexer.md` beside `src/lexer.rs`, and add a token of your own.

CHAPTER 35

# Why Zeno Is as It Is

Every chapter of this book has had boxes that give the reason for a choice. This last chapter steps back and looks at the choices together: the principles they come from, what Zeno took from the languages before it, what it left out on purpose, and what version 0.1 does not have yet.

## 35.1 Unambiguous and intuitive

One sentence was set down at the start, and every feature was held to it: *the entire language must be unambiguous and intuitive*. Unambiguous means a line has one reading, and a reader never has to work out which of two things a symbol means. Intuitive means the reading it has is the one a newcomer would guess. When the two pulled apart, a message from the compiler bridged them: `elif`, `&&`, `def` and `;` are each met with the Zeno spelling.

## 35.2 The five principles

Most decisions fall out of five principles. When two choices are otherwise equal, the one higher on the list wins.

1. **Catch the mistake as early as possible.** Before the program runs is better than while it runs; while it runs is better than a wrong answer. Zeno should never quietly do something the programmer did not mean.
2. **Read like plain words.** Code is read far more often than it is written. A whole word beats a symbol or an abbreviation.
3. **Punctuation has to earn its place.** A mark stays only if removing it would make code harder to read or easier to get wrong.
4. **One way to say a thing, with one meaning.** An operator or keyword should not change meaning with the values it is given.
5. **Nothing a microcontroller cannot afford.** Every feature must compile to small, predictable C with no hidden runtime.

Where Python already had a spelling that passed these tests, Zeno used it, because more people can read Python than any other language.

## 35.3 What came from where

Zeno set out to blend the good parts of the big languages. Here is where each part came from:

| From | Zeno took |
|---|---|
| Python | `and`/`or`/`not`, `in`, `try`/`except`/`raise`, f-strings, comprehensions, `import` and `from`, `//` and `**`, the precedence of the bit operators, chained comparisons |
| C | braces, `0x` numbers, the bit operators, compiling to C itself, integer promotion (without its silent cut) |
| C++ | generics checked for each use, compiled to a copy for each type |
| Rust | `let`, one owner for a value, `unsafe` blocks, the sized number names, enums that carry data and a `match` that must be complete |
| Java | `interface`, and a class that says which ones it is |
| JavaScript | functions as values with closures, written in place |
| Go and Swift | no semicolons; Go's square brackets for generics and its one layout for every file |
| Kotlin and TypeScript | a checker that follows `if x is not none` |

## 35.4 What was left out, and why

As much of Zeno's design is in what it does not have:

**No null that can appear anywhere.** Only `T or none` can be `none`, and it must be asked about.

**No quiet conversions.** An `int` does not become a `float`, a `bool` or a `str` by itself, and nothing is true or false but a `bool`.

**No wrapping arithmetic by default.** An overflow is an error; wrapping is a word, `wrap`.

**No negative indexes.** A position below zero is a mistake, not the other end of the list.

**No inheritance.** A class is read in one place; interfaces do what inheritance was mostly for.

**No constructor of your own.** The fields are the constructor, and a named function builds what needs checking.

**No exceptions that unwind.** Errors are declared with `raises` and compile to jumps or to return values, with no tables and no cost where they are not used.

**No mutable default arguments shared between calls,** no `is` for comparing values, no `elif`, no `lambda`, no `=>`.

Each of these is something a language before Zeno had, and that somebody, somewhere, lost an afternoon to.

## 35.5 How decisions were made

Every decision in Zeno has a written reason, in the file `docs/decisions.md` of the repository: what was decided, why, and what was turned down. A decision made on less than it deserved was marked *provisional*. Before version 0.1 each of the eighty or so was read again and given one of three answers: settled, a stated limit of 0.1, or changed. A choice with no good reason should not be in the language, and if a better reason turns up for the opposite choice, the choice should change.

The decisions are numbered, and the boxes in this book cite them, so anyone who wants the full argument for a rule can find it.

## 35.6 What 0.1 does not have

Every limit is refused today with a message, so each can be added later without changing what any program written now means. The main ones:

- **On a board:** reading a pin, an analogue converter or a serial port, interrupts and timers. These must be designed against real hardware.
- **Numbers:** a `u64`, units made from other units, and reading a file as bytes.
- **Types:** unions other than `or none`, interfaces with type parameters, and bounds on generics.
- **The library:** `sort` by a function, reading a file in pieces, `program.exit`, random numbers.
- **Builds:** compiling for another processor in one step, and a few corners such as a named function inside a function.

Appendix G has the full list.

## 35.7 The name

Zeno is named after Zeno of Elea, whose paradoxes say that Achilles can never catch the tortoise: before he reaches it, he must reach halfway, and before that, halfway to halfway, for ever. And after the quantum Zeno effect, where a system watched closely enough never changes. Its motto is *every journey is halfway done, forever*: a fitting name for a language that will always be almost finished, and a good thing for a language to be, since it means there is always the next half to build.

**What this chapter said**

Zeno is held to one sentence, unambiguous and intuitive, and five principles: catch mistakes early, read like words, earn every mark, one way with one meaning, nothing a microcontroller cannot afford. It takes from Python, C, C++, Rust, Java, JavaScript, Go, Swift, Kotlin and TypeScript, leaves out what has cost programmers most, and writes the reason for every choice down.

APPENDIX A

# Keywords

These 37 words are the keywords of Zeno. A keyword is never a name: a variable, function or field cannot be called `match` or `take`, and the compiler says so if one is.

| Word | Means | Chapter |
|---|---|---|
| `and` | true when both sides are | 5 |
| `as` | names what is caught: `except ValueError as e` | 14 |
| `assert` | in a test: this must be so | 25 |
| `break` | leave the loop now | 7 |
| `case` | one arm of a `match` | 12 |
| `class` | declares a class | 11 |
| `const` | a value fixed for the whole program | 4 |
| `continue` | go on with the next turn of the loop | 7 |
| `else` | the other branch of an `if` | 7 |
| `enum` | declares an enum | 12 |
| `except` | catches an error of a kind | 14 |
| `false` | the `bool` that is not true | 5 |
| `for` | goes through the items of something | 7 |
| `from` | `from math import sqrt`; and `needs ... from` in a package | 21 |
| `function` | declares a function, or writes one in place | 8 |
| `if` | runs a block when a condition holds | 7 |
| `import` | brings in a module | 21 |
| `in` | is it in there? and `for x in xs` | 9 |
| `interface` | declares an interface | 20 |
| `is` | with `none` only: `x is none`; and `class C is I` | 13 |
| `let` | declares a name that keeps its value | 4 |
| `match` | runs the case that fits a value | 12 |
| `none` | nothing here | 13 |
| `not` | turns a `bool` round | 5 |
| `or` | true when either side is; `T or none` | 5 |
| `owned` | a class or list with one owner | 30 |
| `private` | for its own file only | 21 |
| `raise` | raises an error | 14 |
| `raises` | in a heading: the errors a function may raise | 14 |
| `return` | gives a value back and ends the call | 8 |

| Word | Means | Chapter |
|---|---|---|
| `strict` | as a file's first line: every type must be known | 16 |
| `take` | on a parameter: the function takes the value over | 30 |
| `true` | the `bool` that is | 5 |
| `try` | runs a block whose errors may be caught | 14 |
| `unsafe` | a block for what Zeno cannot check | 33 |
| `var` | declares a name whose value can change | 4 |
| `while` | repeats a block while a condition holds | 7 |

## Words that are not keywords

A few words have a meaning in one place and are ordinary names everywhere else, because in that place no name could stand:

- `test`, before a string at the top of a file, starts a test (Testing).
- `package`, `needs` and `at` are the words of a `zeno.package` file (Packages).
- The names of the built-in types (`int`, `str`, `list`, `u8` and the rest), of the units (`ms`, `V`) and of the built-in functions (`print`, `len`) are names declared by Zeno, which a program may declare again in an inner block.

## Words from other languages

These are not Zeno, and each gets a message with the Zeno spelling: `def` and `fn` (`function`), `elif` (`else if`), `&&`, `||` and `!` (`and`, `or`, `not`), and a `;` at the end of a line. So do the names other languages give to nothing and to the two truths: `None`, `null`, `nil`, `NULL`, `nullptr` and `undefined` are told that Zeno spells it `none`, and `True` and `False` that it spells them `true` and `false`.

APPENDIX B

# Operators

## Every operator

| Operator | Means | Works on |
|---|---|---|
| `a + b` | add; join text; join lists | numbers, quantities, `str`, lists |
| `a - b` | subtract | numbers, quantities, pointers |
| `-a` | negate | numbers, quantities |
| `a * b` | multiply | numbers; a quantity by a number |
| `a / b` | divide, always giving a `float` | numbers, quantities |
| `a // b` | divide and round down | numbers |
| `a % b` | remainder of `//`, with the sign of `b` | numbers |
| `a ** b` | power | numbers |
| `a & b` | bit and | whole numbers |
| `a \| b` | bit or | whole numbers |
| `a ^ b` | bit exclusive or | whole numbers |
| `~a` | every bit turned over | whole numbers |
| `a << n`, `a >> n` | shift left, shift right | whole numbers |
| `a == b`, `a != b` | equal, not equal, by contents | values of one type |
| `a < b`, `<=`, `>`, `>=` | order; they chain: `0 <= i < n` | numbers, quantities, `str` |
| `x in xs`, `x not in xs` | is it there? | lists, text, dict keys |
| `x is none`, `x is not none` | is it nothing? | `T or none` |
| `a and b`, `a or b`, `not a` | logic, asking the right side only when needed | `bool` |
| `xs[i]` | item `i`, from 0 | lists, text, dicts, pointers |
| `x.name` | a field, a method, a module's name, a unit change | |
| `f(a, b)` | a call | functions, classes, enum cases |

## Which goes first

From the loosest to the tightest:

| Operators | |
|---|---|
| `or` | loosest |
| `and` | |
| `not` | |
| `==  !=  <  <=  >  >=  in  not in  is  is not` | comparisons, which chain |
| `\|` | |
| `^` | |
| `&` | |
| `<<  >>` | |
| `+  -` | |
| `*  /  //  %` | |
| `-x  ~x` | |
| `**` | tightest; and it groups to the right |

The bit operators bind tighter than the comparisons, as in Python and unlike C. So this test means what it looks as if it means, `(flags & MASK) == 0`:

```
let ready = flags & MASK == 0
```

## The = forms

Every operator on two numbers has an = form, which updates a `var`, a field or an item:

```
+=  -=  *=  /=  //=  %=  **=  &=  |=  ^=  <<=  >>=
```

`x += y` means `x = x + y`, with `x` read first. On a sized number the result is checked as it is stored, so `count += 1` on a `u8` of 255 is an `OverflowError`. `/=` is refused on a whole number, because `/` gives a `float`. A list takes += to add another list to its end.

## What raises

| When | Raises |
|---|---|
| a whole-number answer does not fit in 64 bits | `OverflowError` |
| `//` or `%` by zero, or `/` by zero | `ZeroDivisionError` |
| a whole number to a negative power | `ValueError` |
| a shift by less than 0 or more than 63 | `ValueError` |
| a shift left that loses bits | `OverflowError` |
| `xs[i]` outside the list or text | `IndexError` |
| `d[key]` of a key that is not there | `KeyError` |

APPENDIX C

# The Library at a Glance

Everything that comes with Zeno, on a few pages. The chapter that explains each part is named beside it.

## Functions that need no import

| Function | Gives |
|---|---|
| `print(a, b, ...)` | shows the values on one line, with spaces between |
| `input()`, `input(question)` | the next line typed, or `none` at the end (23) |
| `len(x)` | how many items, characters or keys |
| `range(stop)`, `range(start, stop, step)` | a list of whole numbers (7) |
| `int(x)`, `float(x)`, `str(x)` | the value as another type; `int` and `float` of text may raise `ValueError` |
| `char(n)` | the character with Unicode number `n` |
| `abs(x)` | `x` without its sign (24) |
| `min(...)`, `max(...)` | the smallest or largest, of numbers or of a list |
| `sum(xs)` | a list of numbers added up |
| `u8(n)` and `u16`, `u32`, `i8`, `i16`, `i32` | `n` as a sized number, or `OverflowError` (18) |
| `u8.wrap(n)` and the others | the low bits of `n`, never failing |
| `allocate(T, n)`, `free(p)` | memory by hand (33) |

## Text: str

| Method | Gives (6) |
|---|---|
| `upper()`, `lower()` | capitals or small letters, a to z only |
| `strip()` | without blanks at either end |
| `split()`, `split(sep)` | a list of words, or of the pieces between each `sep` |
| `lines()` | a list of the lines |
| `replace(old, new)` | every `old` made `new` |
| `starts_with(s)`, `ends_with(s)` | `bool` |

| Method | Gives (6) |
|---|---|
| `find(s)` | a position, or `none` |
| `slice(from, to)`, `slice(from)` | a part |
| `code()` | the Unicode number of a one-character text |

## Lists: list[T]

| Method | Does (9) |
|---|---|
| `append(x)`, `insert(at, x)` | adds an item |
| `pop()`, `pop(at)` | takes out the last item, or the one at `at`, and gives it |
| `clear()` | takes out every item |
| `reverse()`, `sort()` | in place |
| `find(x)` | a position, or `none` |
| `slice(from, to)`, `slice(from)` | a new list of a part |
| `join(sep)` | for text: one text with `sep` between |
| `map(f)`, `filter(f)` | a new list |
| `reduce(f, start)` | one value |

## Dicts: dict[K, V]

| Written | Does (10) |
|---|---|
| `d[key]`, `d[key] = v` | reads (or `KeyError`), stores |
| `key in d` | is it there? |
| `get(key)`, `get(key, instead)` | the value, or `none`, or `instead` |
| `pop(key)` | takes the key out and gives its value |
| `keys()`, `values()` | new lists, in the order the keys came |
| `clear()` | takes every key out |

## The modules

| Name | Is |
|---|---|
| `import math` (24) | |
| `pi`, `e` | the numbers |
| `sqrt(x)` | square root |
| `floor(x)`, `ceil(x)`, `round(x)` | to a whole number, an `int` |
| `sin`, `cos`, `tan` | of an angle, `deg`, `rad` or a plain number |
| `asin`, `acos`, `atan`, `atan2(y, x)` | an angle in `rad` |
| `exp(x)`, `log(x)`, `log2(x)`, `log10(x)` | powers of $e$ and logarithms |
| `import time` (17) | |
| `sleep(t)` | waits; `t` has a unit of time |
| `now()` | a clock that only goes forward, in `ms` |
| `import board` (27) | |
| `led` | the pin of the LED |
| `pin(n)` | the pin with that number |
| `register(T, address)` | a register of size `T` |
| `p.on()`, `off()`, `toggle()`, `is_on()`, `number` | what a `Pin` does |
| `r.read()`, `r.write(v)`, `r.address` | what a `Register` does, inside `unsafe` |
| `import files` (23) | |
| `read(path)` | the whole file, as text |
| `write(path, text)`, `append(path, text)` | writes, or adds to the end |
| `exists(path)` | `bool` |
| `remove(path)` | removes the file |
| `import program` (23) | |
| `arguments()` | the words after the file |

APPENDIX D

# Units at a Glance

Every unit a number can carry (Numbers with Units). Units in one row work together, and an answer is in the finer of the two. Units in different rows never meet.

| Measures | Units, finest first | Each is |
|---|---|---|
| time | `ns us ms s min h` | 1000, 1000, 1000, 60, 60 of the one before |
| frequency | `Hz kHz MHz` | 1000 of the one before |
| voltage | `uV mV V kV` | 1000 of the one before |
| current | `uA mA A` | 1000 of the one before |
| resistance | `ohm kohm Mohm` | 1000 of the one before |
| power | `mW W kW` | 1000 of the one before |
| energy | `J Wh kWh` | `1 Wh` is `3600 J`; `1 kWh` is `1000 Wh` |
| temperature | `degC` | on its own |
| temperature | `K` | on its own: an offset, not a scale, from `degC` |
| length | `mm cm m km` | 10, 100, 1000 of the one before |
| mass | `mg g kg` | 1000 of the one before |
| angle | `deg` | on its own; `.rad` changes it |
| angle | `rad` | on its own; `.deg` changes it |
| rotation | `rpm` | on its own |

| Written | Means |
|---|---|
| `500 ms` | a quantity |
| `wait.s` | `wait` in seconds |
| `wait / 1 ms` | the bare number of milliseconds |
| `function f(t: ms)` | a parameter that takes milliseconds only |
| `f"{supply:.2f}"` | the format is the number's; the unit stays |

APPENDIX E

# The zeno Command

Everything Zeno does is done by one command, `zeno`, followed by what to do and usually a file. This appendix lists every command and every option of version 0.1. `zeno --help` prints the same list.

## E.1 Commands

| Command | What it does |
|---|---|
| `zeno run f.zeno` | check the program, then run it. Words after the file are the program's own arguments (Reading and Writing) |
| `zeno check f.zeno` | check the program for mistakes, and run nothing |
| `zeno build f.zeno` | compile to C and then to a program that runs by itself (Compiling). The C is kept beside it |
| `zeno test` | run the `test "name" { }` blocks of a file, of every file in a folder, or of this folder when nothing is named (Testing) |
| `zeno format f.zeno` | lay files out the one way Zeno lays them out; several files, or a folder, can be named (The Tools) |
| `zeno new name` | make a folder with a new program in it: a `main.zeno`, a module with a test, and a `zeno.package` (Packages) |
| `zeno new --package name` | make a folder with a new package: modules for other programs to need, and no `main.zeno` |
| `zeno fetch` | get the packages that the `needs` lines of `zeno.package` list, for this folder or a named one |
| `zeno tokens f.zeno` | show the tokens the lexer finds (Inside Zeno) |
| `zeno parse f.zeno` | show the syntax tree the parser builds, with brackets around every expression |
| `zeno join f.zeno` | show the one program that the file and every file it imports are joined into |
| `zeno bytecode f.zeno` | show the bytecode that `zeno run --vm` runs |
| `zeno lsp` | the language server. An editor starts it, to show mistakes as you type; you do not run it yourself |
| `zeno --version` | print the version |
| `zeno --help` | print the list of commands and options |

## E.2 Options

An option goes after the command and before the file.

| Option | With | What it does |
|---|---|---|
| `--strict` | `run`, `check` | every type must be known before the program runs, as a build needs (Types, Written and Worked Out). `build` always works so |
| `--vm` | `run` | compile to bytecode and run that, in place of walking the tree. Same output, about twice as fast |
| `--no-collector` | `check` | check as `build --no-collector` would, without building |
| `--firmware` | `build` | errors travel as return values, with no `setjmp` (Firmware) |
| `--no-collector` | `build` | no garbage collector and no `malloc`: every list and object has one owner, and memory comes from a fixed pool (Building with No Collector). Turns `--firmware` on |
| `--llvm` | `build` | write LLVM IR in place of C and hand it to `clang`. Goes with the three options around it |
| `--memory` | `build` | the program reports, as it ends, how much memory it used (Memory) |
| `--check` | `format` | change nothing; name the files that are not laid out, and fail if there are any |
| `--package` | `new` | make a package, not a program |
| `--update` | `fetch` | look every version up again, and not at the commits that `zeno.lock` pinned |

## E.3 What a command answers

Every command ends with an *exit status*, the number a shell script or a build tool reads to know how it went: 0 when all went well, and 1 when the program had a mistake, an uncaught error, a failing test, or a file that `format --check` would change. Messages go to the error stream, and what the program prints goes to the output stream, so `zeno run f.zeno > out.txt` keeps only what the program printed.

## E.4 Telling the C compiler

A build writes C, and two names can be set when that C is compiled, with `-D` on the `cc` or `clang` line:

| Name | What it sets |
|---|---|
| `ZENO_BOARD` | a header that puts a real board under the program, in place of the simulated one: `-DZENO_BOARD='"my_board.h"'` (Firmware) |
| `ZENO_POOL` | the size of the pool in bytes for a build with no collector, 32 768 unless set: `-DZENO_POOL=25384` |
| `ZENO_TEXT_ROOM` | the size of the text room in bytes, for the same build |

APPENDIX F

# Kinds of Error

Zeno meets a mistake at one of three times: before the program runs, while it runs, or, in a compiled program, as a fault that stops it. This appendix lists what can be met at each, with the message.

## F.1 Before the program runs

The lexer, the parser and the checker find mistakes in the text. Each is reported with the file, the line and the column, the line itself, and a mark under the place, and nothing is run:

```
let count: int = "three"
```

```
error: `count` should be int, but this is str
 --> main.zeno:1:18
  |
1 | let count: int = "three"
  |                  ^

1 error found. Nothing was run.
```

The lexer and the checker report every mistake they are sure of; the parser stops at the first (Inside Zeno). `zeno check` finds the same mistakes as `zeno run` and stops there.

## F.2 The kinds raised while it runs

An error raised while the program runs has a *kind*, which is what `except` names (When Things Go Wrong). The kinds Zeno raises itself are these:

| Kind | Raised when |
|---|---|
| `ValueError` | `int` or `float` is given text that is not a number; the program raises it for a value it refuses |
| `IndexError` | a position is past the end of a list or text, or below zero; `pop` on an empty list |
| `KeyError` | a dict is asked for a key it does not have |
| `ZeroDivisionError` | `/`, `//` or `%` by zero |
| `OverflowError` | a result does not fit in its type: an `int` past $2^{63} - 1$, or a sized number past its range |
| `FileError` | a file cannot be read or written: it is not there, or it may not be opened |

A program can raise any of these itself, and an object of any class of its own. `except Error`

catches every kind, and inside it `e.kind` is the kind's name and `e.message` the text.

## F.3 When nothing catches it

An error that no `except` catches ends the program. It prints `uncaught`, the kind and the message, and where it was raised, and the program's exit status is 1:

```
let ages = ["Ada": 36]
print(ages["Grace"])
```

```
error: uncaught KeyError: this dict has no key "Grace"
 --> main.zeno:2:7
  |
2 | print(ages["Grace"])
  |       ^
```

```
let readings = [3, 4]
print(readings[2])
```

```
error: uncaught IndexError: index 2 is out of range for a list of length 2
 --> main.zeno:2:7
  |
2 | print(readings[2])
  |       ^
```

```
var n = 9223372036854775807
n += 1
```

```
error: uncaught OverflowError: the result is too big for an int
 --> main.zeno:2:1
  |
2 | n += 1
  | ^
```

A compiled program prints these same lines, byte for byte, in every kind of build.

## F.4 Faults in a firmware build

In a build with `--firmware` or `--no-collector`, only `raise`, `int`, `float` and `FileError` travel as values that `except` can catch. Every other kind above is a *fault*: the program stops with the same message, and a `try` that hopes to catch one is refused before anything is built (Firmware).

## F.5 Messages from memory

These come only from builds, or from `unsafe` code, and are always faults:

| Message | Means |
|---|---|
| `the memory pool is full` | a build with no collector needed more than `ZENO_POOL` bytes at once (Building with No Collector) |

| Message | Means |
|---|---|
| `a statement built more text than there is room for` | one statement built more text than `ZENO_TEXT_ROOM` holds |
| `a block of memory was given back twice` | the runtime found a double `free` |
| `out of memory` | `malloc` failed, in a build with a collector |
| `the simulated board remembers …` | more registers were written on the simulated board than it keeps |

Under `zeno run`, `unsafe` code is watched more closely than any compiled program can be, and each message says what a compiled program would have done instead (Unsafe Code):

```
let a = allocate(u8, 4)
unsafe {
    a[9] = 1
}
```

```
error: this writes over item 9 of a block that has 4 (0 to 3). `zeno run` checks every
    ↪ use of a pointer. A compiled program does not: it writes over whatever is beside
    ↪ the block
 --> main.zeno:3:5
  |
3 |     a[9] = 1
  |     ^
```

```
let a = allocate(u8, 4)
unsafe {
    free(a + 1)
}
```

```
error: `free` needs the pointer that `allocate` gave, and this one points 1 item further
    ↪  on. `zeno run` checks every `free`. A compiled program does not
 --> main.zeno:3:5
  |
3 |     free(a + 1)
  |     ^
```

APPENDIX G

# What Version 0.1 Leaves Out

Before version 0.1, every decision in `docs/decisions.md` was read again, and thirty were answered "a limit of 0.1": the thing is left out of this version, and a program that asks for it is told so. Each can be added later without changing what any program written now means. The numbers are the decisions, for anyone who wants the argument.

## G.1 On a board

**Inputs.** Reading a pin, an analogue converter or a serial port, and interrupts and timers. This was the one item taken off the list for 0.1, because it has to be designed against real hardware. The simulated board can write pins and registers and read registers.

## G.2 Numbers

**No `u64`.** Sized numbers stop at 32 bits, and `int` is the 64-bit signed number (2.8d).

**A `list[u8]` takes eight bytes an item.** The type is checked as bytes, but stored as whole numbers (5.13a).

**Units made of units.** Two quantities cannot be multiplied or divided into a third, as `V` times `A` into `W`, and a program cannot name a unit of its own (1.15).

**No bytes from a file.** Files are read and written as text (6.16).

## G.3 Types

**Unions.** Only `or none` can follow a type: `int or str` is refused (1.13). An enum that carries data does the same work.

**Generic interfaces and bounds.** An interface cannot take a type parameter, so a generic function cannot ask that its `T` has a method (3.16d).

**Generic enums.** An enum cannot take a type parameter, and a generic function cannot be kept as a value (3.16e).

## G.4 The library

**`sort` by a function.** `sort` uses the natural order of numbers and text (6.6).

**Files in pieces.** A file is read whole, or as a list of lines (6.12, 6.20).

**`program.exit` and the environment.** A program ends at its last line or with an uncaught error, and cannot read environment variables (6.19).

**A few functions of mathematics.** No `hypot`, no hyperbolic functions, no `gcd` or `clamp`, and no random numbers (6.21).

**A library written in Zeno.** The library is built into the compiler, not written in Zeno itself.

## G.5 Builds

**A named function inside a function** runs under `zeno run`, but is refused by a build. Write it at the top, or as a function value.

**One file at a time.** A build compiles the whole program at once (7.11).

**Another processor in one step.** A build writes C for the C compiler you name; compiling for a board is a second step on the `cc` line.

**GCC and Clang built-ins.** Checked arithmetic uses them, so the C wants one of those two compilers (5.5e).

**`setjmp` in an ordinary build.** An ordinary build uses it for `try`, and a firmware build does not use it at all (5.11l, 12.8).

## G.6 Firmware and no collector

**A raising function as a value.** In a firmware build, a function that says `raises` cannot be passed around, because a function type does not say whether it raises (5.5l).

**Lists of maybe.** `list[int or none]` is refused with no collector; an enum with a case for the missing value is the answer (5.16h).

**Dicts of lists or objects** with no collector (10.14).

**Some moves of an owned value** that have no rule yet for what is left behind (5.8i, 5.15e, 5.17d).

**`raise e` with no collector,** raising again an error that was caught (5.9d).

**The pool's size** is set on the C compiler's line, not in the program (5.9j); spare blocks are not filled with a pattern to catch their use (5.9k).

## G.7 Interfaces

**Functions of an interface** have no defaults for parameters, no type parameters, and cannot be private (10.5).

**An owned class as an interface,** and an interface with type parameters of its own (10.10).

## G.8 Tools

**Hover on a field.** The language server shows the type of a name, but not yet of a field (9.11).

**Choosing tests.** `zeno test` runs every test in the files it is given, and cannot run one by name or test a compiled program. `assert` takes no message, and is only for tests (10.21).

## G.9 Not limits

Three things are not on this list, because they are decided: Zeno has no inheritance, no second way to write a function, and no second name for a module. The reasons are in Why Zeno Is as It Is.