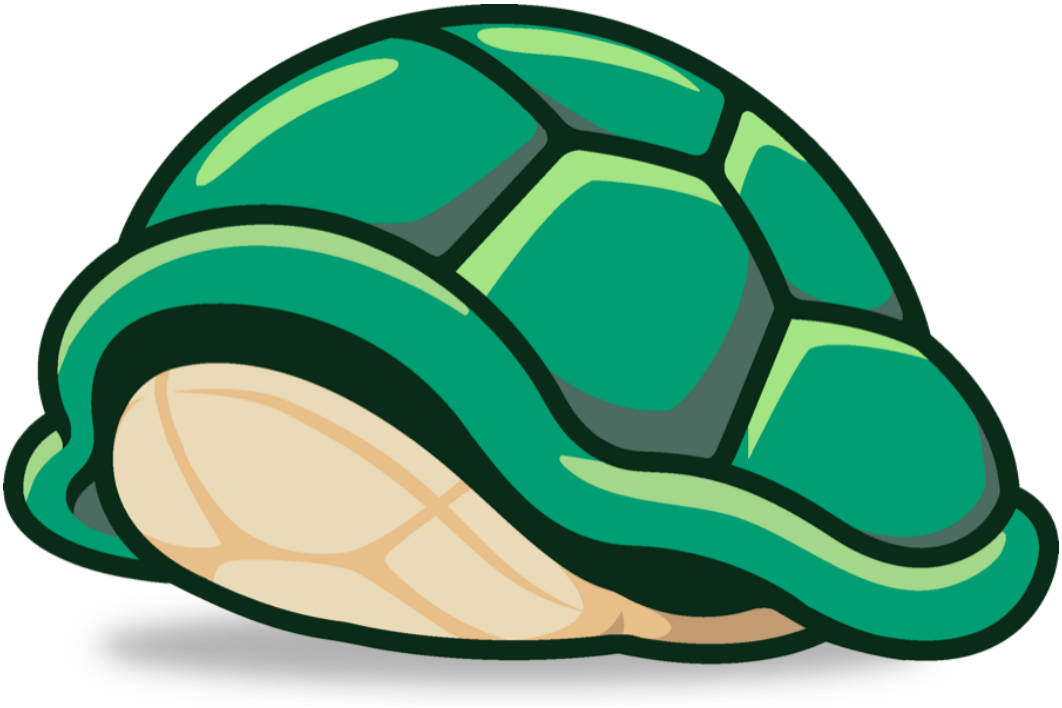
Turtle Programming Language



A language built for expression

Creator

Nigele McCoy

https://turtlelang.dev

Contact: nigelem@turtlelang.dev

Free to read. If Turtle helps you, you can support it with a donation:
https://turtlelang.dev/donate

This book is copyrighted.

Copyright 2017-2026 Nigele McCoy. All rights reserved.

Contents

Part I: The language 11

Chapter 1. Why Turtle . . . 12

1.1 Expressive . . . 12

1.2 Abstraction . . . 12

1.3 Simplicity . . . 12

1.4 Robust . . . 13

1.5 Easy typing . . . 13

Chapter 2. Your first program . . . 14

2.1 Comments . . . 15

2.2 A real program: FizzBuzz . . . 15

Chapter 3. Values and names . . . 17

3.1 Values and types . . . 17

3.2 Variables and assignment . . . 18

3.3 Numbers . . . 18

3.4 Text . . . 19

3.5 Changing a type with change . . . 20

3.6 Truth values and nothing . . . 21

3.7 Example: converting temperatures . . . 21

3.8 Operators . . . 22

3.9 The order of operations . . . 22

Chapter 4. Showing and asking . . . 25

4.1 show . . . 25

4.2 Separators: commas, ; and periods . . . 26

4.3 Asking the user . . . 26

4.4 Warnings . . . 27

4.5 Looking things up . . . 27

Chapter 5. Decisions . . . 30

5.1 Testing for an empty value . . . 31
5.2 Nested decisions . . . 31
5.3 Truth values in conditions . . . 31
5.4 Checking a kind of value . . . 32
5.5 A real program: classifying numbers . . . 32

Chapter 6. Loops and repetition . . . 34

6.1 Going through a list, a set or a map . . . 35
6.2 Looping over a map . . . 35
6.3 Leaving and skipping . . . 37
6.4 Loops inside loops . . . 37
6.5 A real program: stopping and skipping . . . 38
6.6 A real program: skipping multiples . . . 38

Chapter 7. Functions . . . 40

7.1 Three ways to call a function . . . 40
7.1a The semicolon in a call . . . 41
7.2 Brackets hold the arguments . . . 42
7.3 Functions can be called before they are defined . . . 42
7.4 Giving back, not changing . . . 43
7.5 Changing shared data . . . 43
7.6 Anonymous functions and closures . . . 44
7.7 Methods as functions, and longer anonymous functions . . . 45
7.8 Scrolls: a value through steps . . . 45
7.9 Recursion . . . 46
7.10 Functions as values . . . 46
7.11 A real program: recursion . . . 46
7.12 Private functions . . . 47
7.13 Every argument must be given . . . 49
7.14 Calling library functions with a function value . . . 50

Chapter 8. Collections . . . 53

8.1 Lists . . . 53
8.2 Sets . . . 54
8.3 Maps . . . 55

8.4 Changing many values at once . . . 56
8.5 Changing a collection with statements . . . 56
8.6 Methods read from left to right . . . 57
8.7 Joining collections . . . 58
8.8 Converting between kinds . . . 58
8.9 Looping over a map . . . 59
8.10 Reference: built-in functions for lists, sets and maps . . . 59
8.11 Every method, with examples . . . 63
8.12 A real program: a phone book . . . 73

Chapter 9. Assemblies . . . 75

9.1 Changing field values . . . 76
9.2 Mistakes name the fields . . . 77
9.3 Equality . . . 77
9.4 Assemblies in functions . . . 78
9.5 A real program: a point record . . . 78

Chapter 10. Nested structures . . . 80

10.1 Reading a value inside another . . . 80
10.2 Changing a value inside another . . . 81
10.3 Shared and copied values . . . 81
10.4 Looping over records . . . 82
10.5 Sets and maps of assemblies . . . 82
10.6 Assemblies that hold lists . . . 82
10.7 Good practice . . . 83

Chapter 11. Scrolls . . . 84

11.1 What a scroll is . . . 84
11.2 Steps . . . 85
11.3 Saved scrolls . . . 86
11.4 Scrolls over lists . . . 87
11.5 Errors inside a scroll . . . 88
11.6 A complete program . . . 88

Chapter 12. Theories: teaching Turtle new words . . . 90

12.1 A first theory . . . 90
12.2 Values and fixed words . . . 91
12.3 Rules for a theory . . . 92
12.3a Several notations, theorems and proofs . . . 93
12.4 Using a theory with a value of its own . . . 95
12.5 Generators: a function as a value . . . 97
12.6 Checking a theory . . . 97
12.7 A real program: a theory in use . . . 98
12.8 Several theories together . . . 98
12.9 Functions outside a theory . . . 99
12.10 A theory built from theories . . . 100

Chapter 13. When things go wrong . . . 104

13.1 An error stops the program . . . 104
13.2 safe and handle . . . 104
13.3 Error kinds . . . 105
13.4 Raising your own error . . . 107
13.5 Errors that name a fix . . . 108
13.6 A real program: dividing safely . . . 108
13.7 Warnings . . . 109

Chapter 14. Modules . . . 111

14.1 Libraries and your own files . . . 112
14.2 Writing a module . . . 113
14.3 Naming the module in front . . . 113
14.4 Importing some names only . . . 114
14.5 Two modules with the same function name . . . 114
14.6 Setting defaults for an import . . . 115
14.7 Some imports add words to the language . . . 118
14.8 Program layout that reads well . . . 118
14.9 Private functions in a module . . . 119
14.10 Different settings for different calls . . . 120

Chapter 15. Sentence form . . . 122

15.1 One line, one idea . . . 122

15.2 Value first: the subject, then the verb . . . 122
15.3 Statements that read as sentences . . . 123
15.4 Name first in a sentence . . . 123
15.5 Functions as sentences: keep, process and give . . . 124
15.6 Phrases from theories . . . 124
15.7 Writing clean sentence code . . . 124
15.8 A program in sentence form . . . 125
15.9 What sentence form does not do . . . 126

Part II: The standard library **128**

Chapter 16. The library at a glance . . . 129
16.1 Built in (no import) . . . 129
16.2 Data: collections and tables . . . 129
16.3 Numbers and chance . . . 130
16.4 Text . . . 130
16.5 Files, system and settings . . . 130
16.6 The internet . . . 130
16.7 Databases . . . 130
16.8 Time . . . 130
16.9 Security . . . 131
16.10 Development . . . 131
16.11 The import line, in one place . . . 131
16.12 Both forms of each call . . . 131

Chapter 17. Collections and data . . . 134
17.1 Working with a list of records . . . 134
17.2 A real program: grouping records . . . 136
17.3 A real program: a sales summary . . . 137
17.4 Tables and CSV files, with statistics . . . 137
17.5 Functions at a glance: the data, sort and search libraries . . . 139
17.6 Functions at a glance: tables and the sorts that are not yet shown . . . 140

Chapter 18. Numbers, chance and text . . . 142
18.1 Text and patterns . . . 142

18.2 A real program: finding patterns in text . . . 143
18.3 A real program: working on text . . . 143
18.4 Matrices . . . 144
18.5 A real program: matrix products . . . 145
18.6 Random values and simulations . . . 146
18.7 A real program: seeded random values . . . 146
18.8 Functions at a glance: the math library . . . 147
18.9 Functions at a glance: the linear library . . . 148
18.10 Functions at a glance: text methods . . . 149
18.11 Functions at a glance: the rest of the math and random libraries . . . 150

Chapter 19. Files, system and settings . . . 152

19.1 Reading and writing files . . . 152
19.2 A real program: reading command-line arguments . . . 153
19.3 Command-line options . . . 154
19.4 Settings and secrets . . . 156
19.5 Running a shell command . . . 157
19.6 Functions at a glance: the system library . . . 157
19.7 Functions at a glance: memory and size . . . 158

Chapter 20. The internet and databases . . . 160

20.1 Talking to the web and building a server . . . 160
20.2 A real program: writing JSON . . . 161
20.3 Databases . . . 162
20.4 Functions at a glance: JSON values . . . 163
20.5 Functions at a glance: the server library . . . 163
20.6 Functions at a glance: the sql library . . . 164

Chapter 21. Time, security and development . . . 167

21.1 Dates and schedules . . . 167
21.2 Passwords and encryption . . . 168
21.3 Logging . . . 169
21.4 A real program: a logged job . . . 170
21.5 Functions at a glance: the time library . . . 170
21.6 Functions at a glance: the crypt library . . . 171

21.7 Functions at a glance: the log settings . . . 171
21.8 Functions at a glance: the schedule settings . . . 172
21.9 Functions at a glance: the log parts and the output file . . . 172

Part III: Checking your code 174

Chapter 22. Tests . . . 175
22.1 A test file . . . 175
22.2 check: one fact . . . 176
22.3 verify: a rule over every item . . . 176
22.4 validate: a rule over random inputs . . . 177
22.5 Running a folder . . . 178
22.6 Testing theories . . . 178
22.7 Which tests to write . . . 178
22.8 Benchmark tests . . . 178

Chapter 23. Inspection: `diagnose` and `hypothesis` . . . 180
23.1 diagnose: one line, one block, or any code . . . 180
23.2 hypothesis: test a claim . . . 182
23.3 Keeping reports in a file . . . 183
23.4 Diagnose and hypothesis with theories . . . 183

Part IV: The toolkit 185

Chapter 24. The REPL . . . 186
24.1 The REPL's own commands . . . 187
24.2 Keys . . . 187
24.3 Colors . . . 188

Chapter 25. Running, tracing and debugging . . . 190
25.1 Running a program . . . 190
25.2 trace: watch each line run . . . 190
25.3 debug: step through a program . . . 192

Chapter 26. Formatting, building and documenting . . . 198

26.1 fmt: one layout for everyone . . . 198
26.2 build: one program file . . . 199
26.3 doc: the documentation, in the terminal . . . 199

Chapter 27. Editors and the language server . . . 201
27.1 VS Code . . . 201
27.2 Other editors . . . 201

Thank you . . . 203

Resources . . . 204

Appendices 205

A. Keywords . . . 206
Reserved words . . . 206
Words an import turns on . . . 209
Words special only in one place . . . 209

B. Index . . . 211

Glossary . . . 212

Part I: The language

Write Turtle in VS Code. The Turtle extension for Visual Studio Code runs your file with one click, colours the code, reports errors as you type, completes names, shows help on hover and jumps to a definition. Install it from the [Visual Studio Code Marketplace](https://marketplace.visualstudio.com/items?itemName=turtle-lang.turtle-lang), or search for "Turtle Programming Language" in the Extensions view.

About the examples. Every example in this book is also on the Turtle website, at https://turtlelang.dev/docs/book-examples. The website shows each program with the output it prints, and it lists every example in the order it appears in the book, so you can run them and compare.

1

Why Turtle

What a programming language is. A computer follows instructions exactly, one step at a time, and it does not guess what you meant. A **programming language** is a set of words and rules for writing those instructions as text that people can read and check. That text is called **source code**. A program called an **interpreter** reads the source code and carries out each statement in order. In Turtle, the interpreter is the `turtle` command, and a file such as `hello.turtle` holds the source code.

Why there are many languages. Each language makes some kinds of work easier to say. Turtle is built for programs that read like plain sentences and that are comfortable to type for long periods. The five ideas below are the reasons for the choices in the rest of the book.

Turtle is a programming language built to be **expressive**, **abstract**, **simple**, **robust**, and easy to type. This chapter describes the ideas behind the language. The rest of the book shows how each idea appears in practice.

1.1 Expressive

Turtle was designed so that a person can express an idea in more than one way. Some ways are short and direct, for the reader who knows the language well. Others are longer and closer to ordinary sentences, for the reader who is learning. A program can be written in whichever form suits the task and the reader, and the two forms mean the same thing.

A statement is built from words that name what it does, so a program reads as sentences rather than a formula.

1.2 Abstraction

Abstraction means hiding complex code behind a simpler name. Turtle supports this in three ways: **theories**, which add a new phrase to the language; **scrolls**, which carry a value through a series of steps; and sentence-like structures, in which values sit inside a phrase. With these, the code that uses a piece of work can stay short, while the detail is kept in one careful place.

1.3 Simplicity

Turtle is simple to understand, simple to read, and simple to use. It has a small set of rules, and the same rule applies wherever it is needed. A learner does not need to master a large set of special cases before writing a first program, and an experienced programmer does not need to look up how a common task is written.

1.4 Robust

Simplicity must not come at the cost of reliability. Turtle is robust enough to depend on for real tasks, and it is built to stay stable. When a program fails, Turtle reports the problem in plain words and, where it can, says what to do about it.

1.5 Easy typing

Turtle is designed to be easy on the hands. Its syntax uses the brackets `[ ]` for calls and blocks, and it avoids keys that need the Shift key to be held down. The less the Shift key is used, the easier a program is to type over long periods.

Key points

- **Expressive**: one idea can be written in more than one way, from short direct forms to longer, clearer ones.
- **Abstraction**: complex code can hide behind a simpler name, using theories, scrolls and functions.
- **Simplicity**: a small set of rules, applied the same way wherever they are needed.
- **Robust**: Turtle is built to stay stable, and its errors are written to point toward a fix.
- **Easy typing**: brackets `[ ]` mark calls and blocks, and no common symbol needs the Shift key.

Best practice. Write for the next reader, who is often you a month later. Name things after what they mean, and keep each piece small enough to check by reading it.

Common mistake. Expecting the computer to know what you meant. It runs exactly what is written, so a small typo either changes the result or stops the program. Read the error message first, because it usually says where the problem is.

2

Your first program

What a program is. A program is a list of instructions that the computer carries out in order. Each instruction is called a **statement**. Turtle reads a program from the top of the file to the bottom, one statement at a time. Functions are the one exception: they are all defined before the first line runs (section 7.3).

Why it matters. Almost every task, from adding up a column of numbers to showing a web page, is a list of steps. Writing the steps in a file means you can run them again, change one step, and share them with someone else.

How Turtle does it. A Turtle program is a plain text file, usually with the extension `.turtle`. Each statement ends with a period. Lines starting with `//` are comments, which the computer ignores but a reader sees.

A **program** is a text file of statements that Turtle runs from top to bottom. Writing a program involves three steps: writing the statements in a file, running the file with the `turtle` command, and reading what the program prints. This chapter writes the smallest useful program and explains each part of it.

Save this in a file called `hello.turtle`:

```
show "Hello, world" .
```

```
Hello, world
```

Run it:

```
$ turtle hello.turtle
Hello, world
```

There are three ways to run a Turtle program, and each one is shown here with the same file.

- **From a terminal.** Type `turtle` and the file name, as above. This is the way to run a program while you write it, because the output appears in the same window.
- **As a program file.** `turtle build hello.turtle -o hello` makes a single file called `hello` that runs without Turtle being installed. Run it with `./hello`, and it prints `Hello, world`. The file runs on the kind of computer it was built on, so build it on the machine that will run it (section 26.2).

- **By double-clicking.** On a Mac where the Turtle app is installed, double-clicking a `.turtle` file opens it in Terminal and runs it. The app's own description says this, and it also says that opening the app itself gives the Turtle prompt.

Three things to notice.

- `show` prints a value.
- The line ends with a space and a period, `.`. A `show` always ends with a period. This is the one place a period is required, and it tells you where the statement stops.
- The text is in double quotes. Single quotes work too, and are handy when the text itself has double quotes: `show 'She said "hi"' .`

You can also start Turtle with no file and type code straight in. That is the REPL, covered in chapter 24.

2.1 Comments

A **comment** is text that Turtle ignores. Comments explain a program to the person who reads it. A line comment starts with `//` and runs to the end of the line. A block comment starts with `//*` and ends with `*//`, and it can cover several lines:

```
// a line comment, which runs to the end of the line
//* a block comment
   that spans several lines *//
show "Comments are ignored" .
```

```
Comments are ignored
```

2.2 A real program: FizzBuzz

The classic first exercise uses everything in this chapter: a loop, a decision with three branches, and `show`. It is in the Complexity corpus, and it runs as it is:

```
// FizzBuzz from 1 to 30.
[loop][n = 1; n <= 30; n++]
    if ] n % 15 == 0 [
        show "FizzBuzz" .
    else if ] n % 3 == 0 [
        show "Fizz" .
    else if ] n % 5 == 0 [
        show "Buzz" .
    else ]
        show n .
    if [end]
[loop][end]
```

```
1
2
Fizz
4
Buzz
... 25 more lines
```

Two things to notice. The order of the tests matters: `n % 15 == 0` comes first, so 15 prints `FizzBuzz` and not `Fizz`. And a plain `show n .` prints a number without any change. Chapter 6 covers the loop itself.

Best practice. Start each file with a comment that says what the program is for, and keep one idea on each line.

Common mistake. Leaving out the period at the end of a statement. Turtle stops and says the line needs a `.`. This is the most common first error, and the message shows the line where it happened.

Key points

- **Program**: a text file of statements that Turtle runs from top to bottom.
- **Comments**: text after `//` that Turtle ignores, used to explain the program to its reader.
- **The period**: every statement ends with a period, which marks where one idea ends.
- **Running**: write the statements in a file, then run it with `turtle` and the file name.
- **FizzBuzz**: the classic first exercise, which brings together the chapter's ideas in one short program.

3

Values and names

What data is. A program works with data: numbers, text, lists of items, and yes-or-no answers. Each piece of data is a **value**, and every value has a **type**, which tells Turtle what can be done with it. A number can be added to another number. Text cannot be added to a number until it is changed.

In plain words: an integer is a whole number, positive, negative or zero. None is Turtle's word for no value. A truth value is either true or false. Precedence is the order in which operators are applied, which section 3.9 explains.

Literals and variables. A **literal** is a value written directly in the code, such as `5`, `"hi"` or `true`. Its value is fixed by the text itself. A **variable** is a name that holds a value, and it can be changed later. In `total = 12`, the `12` is a literal, and `total` is the variable. Later, `total` can hold another value, but `12` always means 12.

Why names matter. A program is easier to read when a value has a name that says what it means. A **variable** is that name. In Turtle, `total = 12` creates the variable `total` and gives it the value 12. Its value can change later with another assignment.

How Turtle does it. Turtle works out each type from the value itself, so you do not declare types in advance. Section 3.5 shows how to convert a value from one type to another.

Values are the data a program works with. **Variables** give values names so that a program can refer to them. **Statements** are the instructions a program runs, one after another, from the first line to the last. This chapter explains each of these, then the operators that combine values, and the order in which Turtle applies them.

This chapter assumes no prior experience. A Turtle program is a text file of statements. Turtle reads the file from top to bottom and runs each statement in turn. A statement usually fits on one line, and each one does a single thing: it gives a name a value, shows a value, or changes which statement runs next.

3.1 Values and types

A value is a piece of data. Every value has a **type**, which says what kind of data it is. The type decides which operations are allowed on the value. You can add two numbers, but a number and a list cannot be added.

Turtle has these types: **integers** (whole numbers), **floats** (numbers with a decimal point), **text**, **truth values**, **lists**, **sets**, **maps**, and **assemblies** that you define. The `typeof` function gives the type of a value as text:

```
show typeof[5] .
show typeof[2.5] .
show typeof["hi"] .
show typeof[list [1]] .
```

```
integer
float
string
list
```

3.2 Variables and assignment

A **variable** is a name attached to a value. It lets a program refer to a value by a word that says what the value means, and it can be given a new value later. In Turtle, a variable is created by an **assignment**, which has the form `name = value`. A variable exists from the first assignment to its name. There is no separate declaration step, and a variable can be given a different value at any time:

```
x = 5
x = x + 1
show x .
```

```
6
```

An assignment never ends with a period. The period belongs to statements that do something visible, such as `show`. Chapter 7 explains that a variable created inside a function belongs to that call, and that it does not change a variable of the same name outside the function.

3.3 Numbers

Whole numbers are integers, and numbers with a decimal point are floats. Integers have no size limit, so you do not need to think about overflow:

```
import math
show 2 at pow 100 .
```

```
1267650600228229401496703205376
```

A `float` always shows its decimal point, even when it is whole: `6 / 3` is `2.0`.

Division has two operators. `/` always gives the exact answer, and `div` gives the whole part:

```
show 7 / 2 .
show 7 div 2 .
show -7 div 2 .
show -7 % 2 .
```

```
3.5
3
-3
-1
```

`div` rounds toward zero, and `%` is the remainder that goes with it, so `(a div b) * b + a % b` is always `a`.

Turtle uses the word `div` because `//` begins a comment.

Numbers can also be written in scientific notation, with an `e` and a power of ten. A number written this way is a float. A float too large for its range is `inf`, and an operation with no defined result gives `nan`:

```
show 1e-9 .
show 6.02e23 .
show 1e308 * 10 .
show 1e308 * 10 - 1e308 * 10 .
```

```
1e-09
6.02e+23
inf
nan
```

A number literal in the program cannot contain underscores. Text can be converted to a number that does, with `change`:

```
show change "1_000" to integer .
```

```
1000
```

3.4 Text

`+` joins two pieces of text:

```
name = "Nigele"
greeting = "Hello, " + name + "!"
show greeting .
```

```
Hello, Nigele!
```

Text and a number do not mix with `+`. This is an error:

```
show "5" + 0 .
```

```
turtle: line 1: '+' can't join string and integer ("5" + 0): to add, change the text
    to a number first, as (change t to integer) + n; to join, write "{t}{n}"
  1 | show "5" + 0 .
```

Chapter 4 shows the two ways to fix it.

Text can be written in double quotes or in single quotes. Single quotes are useful when the text itself contains double quotes. Inside double quotes, a backslash gives a special character: `\n` is a new line, `\t` is a tab, `\"` is a double quote, and `\\` is a backslash. A backslash before a brace, `\{`, keeps the brace as text. Text in backticks is raw: every character is kept as it is written.

```
show "tab\there" .
show 'She said "hi"' .
show "set \{1, 2}" .
show `raw \n stays` .
```

```
tab here
She said "hi"
set {1, 2}
raw \n stays
```

3.5 Changing a type with change

`change` converts a value. As a statement it changes the name in place. As an expression it gives back a new value and leaves the original alone:

```
raw = "42"
change raw to integer .
show raw + 1 .

n = change "3.5" to float
show n * 2 .
```

```
43
7.0
```

The type names are `integer`, `float`, `string`, `ascii`, `char`, `hex`, `list`, `set`, `keys` and `values`. Some of these convert in both directions, with Turtle working out which one you mean:

```
show change "A" to ascii .
show change 66 to char .
show change 255 to hex .
show change "ff" to hex .
```

```
65
B
ff
255
```

3.6 Truth values and nothing

`true` and `false` are truth values. `none` means no value at all. Chapter 5 shows how conditions use them.

`none` is the value of nothing. It is the result of a function that gives no value, and it is the value of a variable that has been set to nothing. `none` is equal to itself and to no other value, so a test for it uses `==`. Adding `none` to a number is an error:

```
x = none
show x == none .
show x == 0 .
show none + 1 .
```

```
true
false
turtle: line 4: '+' can't add none and integer — none only adds to none; check for it
    first, e.g. if ] x != none [
  4 | show none + 1 .
```

3.7 Example: converting temperatures

This program reads a list of Celsius temperatures and converts each one. It shows numbers at work, and one thing every programmer meets: floats are stored in binary, so some results carry rounding noise.

```
// Convert a list of Celsius temperatures to Fahrenheit and Kelvin.
celsius = list [-40, 0, 37, 100]
[loop][c in celsius]
    f = c * 9 / 5 + 32
    k = c + 273.15
    show c, " C is ", f, " F and ", k, " K" .
[loop][end]
```

```
-40 C is -40.0 F and 233.14999999999998 K
0 C is 32.0 F and 273.15 K
37 C is 98.6 F and 310.15 K
100 C is 212.0 F and 373.15 K
```

Notice `-40.0`. `/` always gives a float, so `f` is a float even when the answer is whole. The Kelvin value `233.14999999999998` is what binary floating point gives for `-40 + 273.15`: a float cannot store `233.15` exactly, and the printed digits show that error. To show two decimal places, round the value. After `import math`, `233.14999999999998 at round 2` gives `233.15`.

3.8 Operators

An **operator** is a symbol that combines values into a new one. Turtle has these:

- **Arithmetic:** `+` adds two numbers or joins two pieces of text; `-` subtracts; `*` multiplies; `/` divides and always gives a float; `div` divides and gives the whole part; `%` gives the remainder.
- **Comparison:** `==` (equal), `!=` (not equal), `<`, `<=`, `>`, `>=`. Each gives `true` or `false`.
- **Logic:** `&&` (and), `||` (or), and `!` (not). They work on truth values.
- **Method and field:** `at` calls a method on the value before it, and `of` reads a field from a value (chapter 9).

Do not confuse `=`, which gives a name a value, with `==`, which asks whether two values are equal.

3.9 The order of operations

When a line has more than one operator, Turtle decides which to do first using a fixed order, highest first. Operators on the same level are done left to right:

Level	Operator	What it does
1 (highest)	`( )`	groups a part of a calculation
2	`at`	calls a method on the value before it
3	`of`	reads a field of a value
4	unary `-`, `!`	applies to the one value in front of it
5	`*`, `/`, `div`, `%`	multiply and divide
6	`+`, `-`	add and subtract, or join text
7	`<`, `<=`, `>`, `>=`	compare
8	`==`, `!=`	equal and not equal
9	`&&`	and
10 (lowest)	`||`	or

Every example below shows the order at work:

```
show 2 + 3 * 4 .
show (2 + 3) * 4 .
show 10 - 4 - 3 .
show 7 div 2 * 2 .
```

```
14
20
3
6
```

`*` comes before `+`, so the first line is `2 + 12`. Parentheses change that order. `-` and the others on the same level run left to right, so `10 - 4 - 3` is `(10 - 4) - 3`. `div` and `*` share a level, so `7 div 2 * 2` is `(7 div 2) * 2`.

Comparisons come before `&&`, and `&&` comes before `||`:

```
show 1 + 2 * 3 == 7 && 2 > 1 .
show 1 > 2 || 3 > 2 && false .
```

```
true
false
```

The first line is `(1 + 2 * 3 == 7) && (2 > 1)`. The second is `(1 > 2) || ((3 > 2) && false)`, which is `false`.

A method and a field bind tighter than arithmetic, so they apply to one value rather than to a whole sum:

```
show "a" + "b" at upper .
assemble O [qty, price]
o = O[3, 1.5]
show qty of o * price of o .
```

```
aB
4.5
```

`"a" + "b" at upper` uppercases only `"b"`, because `at` binds to the value just before it. Use parentheses to uppercase the joined text: `("a" + "b") at upper` gives `AB`. In the second example, `qty of o` and `price of o` are each read first, then multiplied.

A unary minus belongs to the number in front of it, so `-2 * 3` is `(-2) * 3`, and `1 - -2` subtracts a negative number:

```
show -2 * 3 .
show 1 - -2 .
```

```
-6
3
```

When you are not sure of the order, use parentheses. They cost nothing at run time and make the intent visible to the next reader.

Best practice. Name a variable after what it holds, such as `order_total`, not `x`. Use one name for one meaning, and avoid changing a variable's type partway through a program.

Common mistake. Joining text and a number with `+`. Turtle stops with an error, because `"5" + 0` could mean 5 or "50". Change the text to a number first, or put the number inside the text with `"{n}"`. Another common mistake is using a name before it has a value: Turtle stops with `undefined variable`.

Key points

- **Values** are data, and every value has a **type** that decides which operations are allowed.
- **Variables** give a value a name, set with `=`, so a program can refer to it.
- **Numbers** are integers when whole, and floats when they have a decimal point.
- **Text** is joined with `+`.
- `change` converts a value to another type.
- **Truth values** are `true` and `false`, and `none` means no value at all.
- **Operators** combine values, and a line with several follows a fixed order of operations.

4

Showing and asking

Output and input. A program talks to its user in two directions. **Output** is what the program shows, and `show` is the statement that writes it. **Input** is what the user types, read with `?`. Together they make a program interactive, because it can ask a question and act on the answer.

A warning is a message that Turtle prints about a likely problem, without stopping the program.

Why input is text. Whatever a user types arrives as **text**, even when it looks like a number. To use the answer as a number, change it first with `change ... to integer` (section 3.5). Until then, arithmetic on it is an error.

A program communicates in two directions. **Output** is what a program shows, and **input** is what it reads from the user. In Turtle, `show` prints output and `?` asks for input. This chapter covers both, and the separators that control how the values on one line are laid out.

4.1 show

`show` takes one or more values, separated by commas. It puts one space between them, and no space before a comma, period, colon, semicolon or closing bracket:

```
x = 4
y = 9
show "x = ", x, ", y = ", y .
```

```
x = 4, y = 9
```

Inside a text, `{...}` puts a value in place. This is usually the clearest way to build a sentence:

```
x = 4
y = 9
show "x = {x}, y = {y}, total = {x + y}" .
```

```
x = 4, y = 9, total = 13
```

Use the form you find easiest to read. The rule is the same either way: a value inside `{...}` is shown the way `show` shows it.

To write braces as ordinary text, a backslash before the brace is enough: `show "set \{1, 2}" .` prints `set {1, 2}`.

4.2 Separators: commas, `;` and periods

Inside `show`, a comma puts a space between items. A `;` does the same, and it has one more job: it ends the arguments of a function call, so the next item is not taken as one of them. Here `fix` takes two arguments:

```
def fix[a, b]
    return a + b
def [end]

x = 1
show x fix 2 ; "The answer" .
show x fix 2 ; x fix 5 .
show x fix 2 ; 7 ; 8 .
```

```
3 The answer
3 6
3 7 8
```

Without the `;`, the comma belongs to the call. `show x fix 2, "The answer" .` gives the error `function "fix" expects 2 argument(s), got 3`. Use `;` when a call with arguments is followed by more items.

Outside `show`, `;` is not a separator. `y = x fix 2 ; 10` is an error, and so is `x = 1; y = 2`. Each statement that does not end at the end of a line needs its own period:

```
x = 1 . y = 2 . show x + y .
```

```
3
```

A period ends a statement, so two statements can share a line. A `show` line always needs its period, even when it is the last line of a file.

4.3 Asking the user

A question mark asks for a line of input. The text after it is the prompt:

```
name = ? "What's your name? "
show "Hi, {name}!" .
```

The answer is always text. Use `change` to make it a number:

```
raw = ? "Age? "
age = change raw to integer
show age + 1 .
```

Give the question its own line, then convert it. Writing the question inside the `change` is not allowed.

4.4 Warnings

`import system` gives you `warn`, which writes to the error stream and does not stop the program:

```
import system
warn "disk is nearly full" .
```

```
disk is nearly full
```

4.5 Looking things up

Turtle can tell you about its own values and words, from inside a program or from the REPL.

`typeof` gives the kind of a value, as text. `none` is its own kind, and `x == none` is how you test for it:

```
show typeof[2.5] .
show typeof["hi"] .
show typeof[none] .
x = none
show x == none .
show x == 0 .
```

```
float
string
none
true
false
```

`help` takes a name and explains it. Use it on a library, a function, or a keyword:

```
import data
help["sum"]
```

```
sum[collection]        (import data)
  Adds up a list's or set's numbers (a map's values): an integer if they
  all are, 0 when there are none. Sentence form: nums sum.
  Example:
    prices = list [4, 9, 6]
... 8 more lines
```

The help forms. `help` answers in four ways, depending on what you give it. The same information is available from the REPL, and `version` is a separate word that gives the version of Turtle that is running.

Write	It gives
`help[]`	the help topics, and the libraries you can import
`help["sum"]`	a library function or method, with an example
`help["if"]`	a keyword, with an example
`help["keywords"]`	every keyword, in one list (appendix A has the same words)
`help[nums]`	a value: what it is and the methods it has
`stdlib[]`	every library and its functions, as a map

```
help[]
```

```
Turtle 0.9.232+11b4626-dev

help["linear"]     a library: what it's for and its functions
help["reshape"]    one function or method, with an example
help["if"]         a keyword; help["keywords"] lists them all
help[x]            what x is and what you can do with it
stdlib[]           every library and its functions, as a map
... 23 more lines
```

```
nums = list [1, 2]
help[nums]
```

```
This is a list of 2 items.
A list's methods, as x at add[value]:
  add[value], get[index], put[value, index], insert[value, index],
  remove[value], pop, slice[start, end], contains[value], index[value],
  count[value], sort, reverse, clear, len, isempty, tostring
  (put and insert take the value first, then the position: put[99, 1];
   the sentence names each: put 99 to nums at 1 .)
Libraries for it: help["data"], help["sort"], help["search"], help["random"]
```

```
show version .
```

```
0.9.232+11b4626-dev
```

Best practice. Make each prompt say exactly what is expected, such as `"Age in years? "` rather than `"Age? "`, and check the answer before you use it.

Common mistake. Doing arithmetic on input without converting it. In this program, `age = ?` followed by `show age + 1 .` stops with `can't join string and integer` when the user types 5. Converting first, with `show change age to integer + 1 .`, prints 6.

Key points

- `show` prints one or more values, and commas separate them with a space.
- **Separators**: a comma or a `;` puts a space between items in `show`, and a period ends the statement.
- `?` asks the user for a line of input, and the text after it is the prompt.
- `warn` (from `import system`) writes a message to the error stream without stopping the program.
- **Looking things up**: a program can ask Turtle about its own values and words.

5

Decisions

What a decision is. A program often has to choose between actions. A **condition** is a question whose answer is always true or false, and a **branch** is the block of statements that runs when the answer is true. Without decisions, a program does the same thing whatever its data says.

How Turtle does it. Turtle writes a decision with reversed brackets around the condition: `if ] condition [`. Every decision closes with `if [end]`. An `else` branch runs when the condition is false.

A **decision** lets a program take one path or another, depending on a **condition**. A condition is an expression whose result is either `true` or `false`. Turtle writes a decision as an `if` statement with reversed brackets, and a decision can be placed inside another. This chapter covers the forms of `if`, the tests that make a condition, and how to nest decisions.

`if` runs code when a condition holds. The condition is wrapped in reversed brackets: `] ... [` rather than `[ ... ]`. Every `if` closes with `if [end]`.

```
x = 12
if ] x > 10 [
    show "big" .
else if ] x > 0 [
    show "small" .
else ]
    show "non-positive" .
if [end]
```

```
big
```

`else if` and `else` are optional. Conditions use `== != < <= > >=`, and `&&` (and), `||` (or) and `!` (not):

```
age = 21
has_ticket = true
if ] age >= 18 && has_ticket [
    show "come in" .
if [end]
```

```
come in
```

5.1 Testing for an empty value

Lists, sets, maps, text and numbers each have a sense of "nothing there": an empty list, empty text, `0` and `none` are all false. So a condition can just name the value:

```
matches = list []
if ] matches [
    show "found some" .
else ]
    show "none found" .
if [end]
```

```
none found
```

`matches at isempty` asks the same question out loud, and is clearer to someone reading the code for the first time.

5.2 Nested decisions

A decision inside another decision works, and each level gets one more `[` at its start:

```
if ] a > 0 [
    show "a is positive" .
    [if ] b > 0 [
        show "both positive" .
    [else ]
        show "a positive, b not" .
else ]
    show "a not positive" .
if [end]
```

The nested part has no `[end]` of its own. It ends when a clause at the same level or above shows up. This reads well for one or two levels. If you find yourself nesting three deep, a function usually makes the code clearer.

5.3 Truth values in conditions

A condition does not have to be a comparison. Any value can be used as one, and Turtle decides whether it is true. `false`, `none`, `0`, `0.0`, empty text, and an empty list, set or map are false. Everything else is true. `!` gives the opposite:

```
if ] "" [
    show "empty text is true" .
else ]
    show "empty text is false" .
if [end]
show !"" .
show 0 && 1 .
```

```
empty text is false
true
false
```

`&&` and `||` stop as soon as the result is known. A check on the right side of `&&` is skipped when the left side is false. This lets a check guard a second check that would be an error otherwise:

```
x = none
if ] x != none && x > 5 [
    show "big" .
else ]
    show "not big, or none" .
if [end]
```

```
not big, or none
```

5.4 Checking a kind of value

`type` checks what a value is:

```
n = 5
if ] n type integer [
    show "a whole number" .
if [end]
```

```
a whole number
```

5.5 A real program: classifying numbers

This program classifies each number in a list as negative or not, then as even or odd, or zero. The outer `if` splits negative from the rest, and each branch has its own nested `if`. Each nested level adds one `[` to its keywords, as chapter 5 described.

```
// Classify a number as negative or positive, and as even or odd, with nested ifs.
[loop][n in list [-4, -3, 0, 7, 10]]
    if ] n < 0 [
        if ] n % 2 == 0 [
            show n, " is negative and even" .
        else ]
            show n, " is negative and odd" .
        if [end]
    else ]
        if ] n == 0 [
            show n, " is zero" .
        else ]
            show n, " is positive" .
        if [end]
    if [end]
[loop][end]
```

```
-4 is negative and even
-3 is negative and odd
0 is zero
7 is positive
10 is positive
```

Read the output against the program: `-3` fails the even test, and `0` is caught by the nested test in the non-negative branch, before the positive case is reached.

Best practice. Keep a condition short enough to read at a glance. If it grows long, give it a name with an assignment first, such as `is_adult = age >= 18`, and test that name.

Common mistakes. Using `=` to test equality. A single `=` assigns a value, so Turtle says `to compare, use ==`. Also leaving out `if [end]`: Turtle says the `if` is never closed, and it names the line to change. Writing ordinary brackets, `if [ x > 1 ]`, gives an error that shows the reversed form.

Key points

- **Decision**: `if ] ... [` runs code only when a **condition** holds, and every `if` closes with `if [end]`.
- **Empty values**: lists, sets, maps, text and numbers each have a test for "nothing there".
- **Nested decisions**: a decision can sit inside another, one level deeper for each one.
- **Truth in conditions**: any value can serve as a condition, not only a comparison.
- `type` checks what kind of value a name holds.

6

Loops and repetition

What a loop is. A loop repeats a block of statements. Some tasks need the same step done for every item in a list, or many times until a result is reached. A loop does that work without copying the statements.

Iteration means repeating steps once for each item. The word break leaves a loop at once, and continue skips to the loop's next pass.

How Turtle does it. A loop starts with `[loop]` and ends with `[loop][end]`. It can run while a condition holds, or once for each item in a collection. `break` leaves the loop at once, and `continue` skips to the next pass.

A **loop** repeats a block of statements, either while a condition holds or once for each item in a collection. A loop lets a short block of code do the same work on many values. This chapter covers the two forms of `[loop]`, how to go through each item in a list, set or map, how to leave a loop early with `break`, and how to skip to the next pass with `continue`.

`[loop]` repeats code. It has two forms, and both end with `[loop][end]`.

The first form counts. It has three parts separated by semicolons: a start, a condition to keep going, and a step:

```
[loop][i = 0; i < 5; i++]
    show i .
[loop][end]
```

```
0
1
2
3
4
```

The second form repeats while a condition holds. Check it before each pass:

```
count = 0
[loop][count < 3]
    show count .
    count = count + 1
[loop][end]
```

```
0
1
2
```

The step accepts `i++`, `i--`, or a full assignment such as `i = i * 2`.

6.1 Going through a list, a set or a map

The most common loop goes through every item in a collection. A loop over a list gives the items in order:

```
nums = list [4, 8, 15]
total = 0
[loop][x in nums]
    total = total + x
[loop][end]
show total .
```

```
27
```

A loop over a **set** gives each item once. The order of a set is not part of its meaning, so this loop adds the items without caring which comes first:

```
s = set [3, 1, 2]
total = 0
[loop][x in s]
    total = total + x
[loop][end]
show total .
```

```
6
```

For a plain total, the shortest way is `sum` from the `data` library, covered in chapter 17: `show nums sum .` after `import data`. The loop above is the way to learn what a loop does; `sum` is the way to write it.

6.2 Looping over a map

A loop over a **map** gives the keys by default. There are several ways to write it, and each gives the same entries for `ages = map ["Alice": 30, "Bob": 25]`.

The most direct form names the `key` and the value together:

```
ages = map ["Alice": 30, "Bob": 25]
[loop][name, age in ages]
    show name, " is ", age .
[loop][end]
```

```
Alice is 30
Bob is 25
```

To use only the keys, name one variable:

```
ages = map ["Alice": 30, "Bob": 25]
[loop][name in ages]
    show name .
[loop][end]
```

```
Alice
Bob
```

To use only the values, change the map to its values first:

```
ages = map ["Alice": 30, "Bob": 25]
[loop][age in change ages to values]
    show age .
[loop][end]
```

```
30
25
```

`at getkeys` gives the keys as a list, in the order they were added. Looping over that list gives the entries in the same order:

```
ages = map ["Alice": 30, "Bob": 25]
[loop][name in ages at getkeys]
    show name, " is ", ages at get name .
[loop][end]
```

```
Alice is 30
Bob is 25
```

A counting loop works too. It is the longest form, and it shows what the other forms do for you. The key is stored in its own name first, because the value needs it:

```
ages = map ["Alice": 30, "Bob": 25]
keys = ages at getkeys
[loop][i = 0; i < keys at len; i++]
    name = keys at get i
    show name, " is ", ages at get name .
[loop][end]
```

```
Alice is 30
Bob is 25
```

Use the first form when you need both the key and the value. Use `name in ages` when you only need the keys, and `age in change ages to values` when you only need the values.

6.3 Leaving and skipping

`break` leaves the loop at once. `continue` skips to the next pass:

```
[loop][i = 0; i < 10; i++]
    if ] i == 5 [
        break
    if [end]
    show i .
[loop][end]
```

```
0
1
2
3
4
```

6.4 Loops inside loops

Unlike `if`, every loop needs its own `[end]`, nested or not:

```
[loop][i = 0; i < 3; i++]
    [loop][j = 0; j < 3; j++]
        show i , ",", j .
    [loop][end]
[loop][end]
```

```
0, 0
0, 1
0, 2
1, 0
1, 1
... 4 more lines
```

The stricter rule is deliberate. It keeps the end of each loop on the line where you can see it.

6.5 A real program: stopping and skipping

`break` leaves a loop at once. This program scans a list and records the position of the first negative value, then stops looking. It uses `at len` for the list's size and `at get[i]` for each item.

```
// Stop scanning at the first negative value.
readings = list [4, 8, 15, -2, 16, -9]
position = -1
[loop][i = 0; i < readings at len; i++]
    if ] readings at get[i] < 0 [
        position = i
        break
    if [end]
[loop][end]
show "first negative at index ", position .
```

```
first negative at index 3
```

The loop never reads past `-2`, so `16` and `-9` are not examined.

6.6 A real program: skipping multiples

`continue` skips the rest of the current pass and moves to the next one. This program prints the numbers from 1 to 20 except the multiples of four.

```
// Print 1 to 20, skipping multiples of four.
[loop][n = 1; n <= 20; n++]
    if ] n % 4 == 0 [
        continue
    if [end]
    show n .
[loop][end]
```

```
1
2
3
5
6
7
9
10
11
13
14
15
17
18
19
```

The `if` decides before `show` runs, so the skipped numbers never print.

Best practice. Make sure each loop changes something that its condition depends on. Otherwise the loop never ends. When you are unsure, print the variable inside the loop to see its values.

Common mistake. Forgetting that every loop needs its own `[loop][end]`, even when it sits inside another loop. Turtle stops with `this loop is never closed: add [loop][end] after its last line`.

Key points

- **Loop**: repeats a block of statements, either while a condition holds or once for each item in a collection.
- **Going through a collection**: the most common loop visits every item in a list, a set or a map.
- **Maps**: a loop over a map gives the keys by default.
- `break` leaves a loop at once, and `continue` skips to the next pass.
- **Loops inside loops**: every loop needs its own `[end]`, whether nested or not.

7

Functions

A **function** is a named block of statements that can be run again whenever its name is used. A long job can be given a short name, so the same code is written once and used many times, and each part of a program stays small enough to check on its own. A function can receive information from the place where it is used, and it can send a result back.

A function value is a function that is stored in a variable or passed to another function. A return value is the result a function gives back. The scope of a name is the part of the program where that name can be used.

The names in the brackets of a definition are its **parameters**: placeholders for the information the function needs. When the function runs, the values written in the brackets of the call, the **arguments**, take the parameters' places. The function's result is given back with `return`, and the caller can store it, show it, or use it in another expression.

A function does not have to take arguments, and it does not have to return a value. A function with no `return` gives back `none`, Turtle's word for "no value". Such a function is called a void function. `def hi[]` is a function with no parameters, called as `hi[]`.

This chapter covers how to define a function, the two forms of a call, and how the variables inside a function stay separate from the rest of the program.

A function gives a name to a piece of code you want to reuse. It is defined with `def`, closed with `def [end]`, and called with `[ ]`:

```
def plus[a, b]
    result = a + b
    return result
def [end]

sum = plus[1, 2]
show sum .
```

```
3
```

`return` gives a value back. A function that reaches its end without a `return` gives back `none`.

7.1 Three ways to call a function

Turtle lets a function be called in three forms. Each form means the same thing, so a program can be written in the form that reads best for the task:

Form	Example	The arguments are
bracket	`fix[1, 2]`	inside `[ ]`, separated by commas
value first	`x fix 2`	the value before the name, then the rest after it
name first	`fix 1, 2`	after the name, separated by commas

```
def fix[a, b]
    return a + b
def [end]

show fix[1, 2] .
x = 1
show x fix 2 .
show fix 1, 2 .
```

```
3
3
3
```

The bracket form is the clearest when a call sits inside a larger expression. The value-first form reads well for a function that acts on a value, such as `nums sum` or `s upper`, and chapter 16 lists the pairs used in the library. The name-first form reads best as a statement of its own, such as `fix 1, 2` on a line, or `show fix 1, 2 .`.

The first argument of a name-first call can be a literal, such as a number or text, or a variable. Each argument can use arithmetic, so `fix 1, 2 + 3` is `fix[1, 5]`. A negative first argument works for a function the program defines: `fix -4, 2` is `fix[-4, 2]`. A variable is read as a value, so `x -1` is subtraction. For a function held in a variable, write the value first or use brackets, which read clearly:

```
f = x give x * 3
show -1 f .
show f[-1] .
```

```
-3
-3
```

7.1a The semicolon in a call

A semicolon `;` marks where a call's arguments end. It has two uses.

In `show`, `;` ends the arguments, and the item after it is printed separately. Without it, a comma after a name-first call would continue the call:

```
def fix[a, b]
    return a + b
def [end]
show fix 1, 2; " is the sum" .
```

```
3 is the sum
```

Inside brackets, such as a list, `;` ends a call's arguments and also separates the items of the list. A comma may follow the semicolon, so both forms below are valid:

```
def fix[a, b]
    return a + b
def [end]
show list [fix 1, 2;, fix 3, 4;, 5] .
show list[fix 1, 2; fix 3, 4; 5] .
```

```
[ 3, 7, 5 ]
[ 3, 7, 5 ]
```

Without a semicolon, the commas after a name-first call inside brackets belong to the call: `list [fix 1, 2]` is `[ 3 ]`.

7.2 Brackets hold the arguments

Arguments go inside `[ ]` and are separated by commas. A function with no arguments still needs its brackets:

```
def greet[]
    show "hello" .
def [end]

greet[]
```

```
hello
```

7.3 Functions can be called before they are defined

A file's functions are all defined before its first line runs, so the main code can come first and the helpers below it:

```
orders = list [1, 2]
report[orders]

def report[list_of_orders]
    show length of list_of_orders .
def [end]
```

```
2
```

Everything else runs top to bottom. A variable is set when its line runs, so a function called early cannot read a variable set further down.

7.4 Giving back, not changing

A function can read a top-level variable, but an assignment inside a function makes a new local variable. The top-level one stays as it was. To change a value, return the new one and assign it where the function is called:

```
count = 0

def bump[n]
    return n + 1
def [end]

count = bump[count]
show count .
```

```
1
```

If the function tries `count = count + 1` with no local `count`, Turtle stops with an error that tells you to do exactly this.

7.5 Changing shared data

A list, set or map is shared, not copied, when it is passed in. So a function can change one in place, and the caller sees the change:

```
nums = list [1, 2, 3]

def add_value[v]
    add v to nums .
def [end]

add_value[99]
show nums .
```

```
[ 1, 2, 3, 99 ]
```

Use this when a function is meant to fill a collection. Otherwise, return the new collection and assign it, which keeps every change visible at the call.

7.6 Anonymous functions and closures

A function does not always need a name. `x give x * 2` is a function with one argument, written in a single line. It is a value like any other, so it can be stored or passed straight to another function:

```
double = x give x * 2
show double[4] .        // 8: the function doubles 4

plus = [a, b] give a + b
show plus[2, 3] .       // 5: the function adds its two arguments

def apply[f, v]
    return f[v]
def [end]

show apply[x give x + 1, 3] .   // 4: apply runs x + 1 on 3
```

```
8
5
4
```

The three lines print in the order of the `show` statements: `double[4]` gives 8, `plus[2, 3]` gives 5, and `apply` passes 3 to a function that adds 1, giving 4. The function `apply` takes another function as its first argument, which is the idea in the last example.

A function defined inside another function is a **closure**. It remembers the values the outer function had when it was made, even after the outer function has returned:

```
def make_adder[n]
    def adder[x]
        return x + n
    def [end]
    return adder
def [end]

add5 = make_adder[5]
show add5[1] .
```

```
6
```

`make_adder[5]` returns a new function that adds five. The `5` is kept inside it, so `add5` works on its own. Closures are how Turtle keeps a small piece of state, such as the counter in chapter 12, without a class.

7.7 Methods as functions, and longer anonymous functions

A method can also be called as a function, with the value first in the brackets. `trim[s]` and `s at trim` both give the text without its spaces:

```
show trim[" hi "] .
show " hi " at trim .
```

```
hi
hi
```

An anonymous function can run to several lines. Write `[x] give` at the start, and close the body with `give [end]`:

```
big = [x] give
    if ] x > 3 [
        return true
    if [end]
    return false
give [end]
show big[5] .
show big[1] .
```

```
true
false
```

7.8 Scrolls: a value through steps

A `scroll` passes a value through a list of functions, in order, and gives the result. Each step gets the value the step before it gave:

```
def add1[x]
    return x + 1
def [end]

def double[x]
    return x * 2
def [end]

x is scroll 3 into add1, double .
show x .
```

```
8
```

The value 3 goes into `add1`, which gives 4, and that goes into `double`, which gives 8. Use a scroll when a value passes through the same kind of steps in order, and a plain sequence of assignments when each step has a different job.

7.9 Recursion

A function can call itself. Give it a case that stops:

```
def factorial[n]
    if ] n <= 1 [
        return 1
    else ]
        return n * factorial[n - 1]
    if [end]
def [end]

show factorial[5] .
```

```
120
```

Turtle stops after 100,000 calls that are still in progress, and says "recursion too deep". That error almost always means the stopping case is missing.

7.10 Functions as values

A function's name, used without `[ ]`, is the function itself. It can be stored, passed to another function, and returned:

```
def double[x]
    return x * 2
def [end]

def apply[f, v]
    return f[v]
def [end]

show apply[double, 21] .
g = double
show g[4] .
```

```
42
8
```

Anonymous functions (`x give x * 2`), closures and sentence-style calls go further than this chapter. They are described in the language reference, and the standard library uses them heavily (chapter 17).

7.11 A real program: recursion

A function may call itself. `total_to` adds `n` to the total of the numbers below it, and stops at zero. Every recursive function needs that stopping case, or it runs until Turtle reports `recursion too deep`.

```
// Sum the numbers from 1 to n by recursion.
def total_to[n]
    if ] n <= 0 [
        return 0
    if [end]
    return n + total_to[n - 1]
def [end]

show "sum 1..10: ", total_to[10] .
show "sum 1..100: ", total_to[100] .
```

```
sum 1..10: 55
sum 1..100: 5050
```

Chapter 7 showed the factorial form. This sum is the same pattern with addition.

7.12 Private functions

A function whose name starts with ~ is **private** to its file. Other functions in the same file can call it, but no other file can. Use one for a helper that the rest of the program does not need, so that a file offers only the functions it means to.

```
def ~helper[x]
    return x + 1
def [end]
show ~helper[1] .
```

```
2
```

The ~ is part of the name. `~helper` and `helper` are two different names, so a file can have a public `helper` and a private `~helper` at the same time. Chapter 14 shows how a private function behaves when another file imports it.

Use case 1: hide a step. A public function can build its result from a helper that callers never need. Below, `price_line` is the only function the program calls. The helper `~money` formats the amount, and callers do not depend on how it does that.

```
// Format a price with two decimals and a dollar sign.
def ~money[cents]
    return "${cents}.00"
def [end]

def price_line[item, cents]
    return item + ": " + ~money[cents]
def [end]

show price_line["Tea", 3] .
show price_line["Cake", 4] .
```

```
Tea: $3.00
Cake: $4.00
```

Use case 2: share one check. Several public functions can use the same check. The rule lives in one place, and the file offers only the functions a caller needs, not the check itself.

```
// Two public functions share one check that callers never need.
def ~in_range[n, low, high]
    return n >= low && n <= high
def [end]

def check_age[age]
    if ] ~in_range[age, 0, 130] [
        return "ok"
    else ]
        return "age must be from 0 to 130"
    if [end]
def [end]

def check_score[score]
    if ] ~in_range[score, 0, 100] [
        return "ok"
    else ]
        return "score must be from 0 to 100"
    if [end]
def [end]

show check_age[30] .
show check_age[-1] .
show check_score[101] .
```

```
ok
age must be from 0 to 130
score must be from 0 to 100
```

Use case 3: repeat a small job. A helper can do a job the program needs many times, such as padding text on the left so the columns of a table line up.

```
// Pad a value on the left so every row of a table lines up.
def ~pad[value, width]
    result = "{value}"
    [loop][length of result < width]
        result = " " + result
    [loop][end]
    return result
def [end]

def show_row[name, count]
    show ~pad[name, 8] + ~pad[count, 4] .
def [end]

show_row["apples", 12] .
show_row["figs", 3] .
```

```
  apples  12
    figs   3
```

Private means only that other files cannot call the function. It does not hide the code from anyone who can read the file. Section 14.9 shows the same idea with modules.

7.13 Every argument must be given

A call must give a value for every parameter of its function. A `def` line has no default values, so a call that leaves one out is an error:

```
def greet[name, greeting]
    return "{greeting}, {name}!"
def [end]
show greet["Ann"] .
```

```
turtle: line 4: function "greet" expects 2 argument(s), got 1
  4 | show greet["Ann"] .
```

The usual stand-in for a default is none. The caller passes none, and the function puts its own value in its place:

```
def greet[name, greeting]
    if ] greeting == none [
        greeting = "Hello"
    if [end]
    return "{greeting}, {name}!"
def [end]

show greet["Ann", none] .
show greet["Ann", "Hi"] .
```

```
Hello, Ann!
Hi, Ann!
```

When a function takes several settings, pass them in a map and read each one with a fallback. Section 14.6 shows that form. The rule is the same for every function: a call must match the number of parameters exactly.

Best practice. Give each function one job and a name that says what it does, such as `total_price`. Keep it short enough to read on one screen, and return the result rather than changing values outside it.

Common mistakes. Calling a function with the wrong number of arguments: Turtle reports how many it expected and how many it got. Forgetting `return`, so the function gives back `none`, and then using that `none` as a number. And assigning a variable inside a function expecting the top-level variable to change. It makes a new local variable instead (section 7.4).

Key points

- **Function**: a named block of statements that takes **arguments**, and gives back a **result** with `return`.
- **Three call forms**: each form means the same thing, and the semicolon `;` marks where a call's arguments end.
- **Arguments** go inside `[ ]`, separated by commas.
- **Order**: a file's functions are all defined before its first line runs, so they can be called anywhere in it.
- **Giving back, not changing**: a function returns its result, and changing shared data is a separate choice.
- **Anonymous functions** need no name, as in `x give x * 2`.
- **Recursion**: a function can call itself, but it needs a case that stops.
- **Private functions**: a name starting with ~ is visible only inside its own file.

7.14 Calling library functions with a function value

Some library functions, such as `process`, `keep` and `reduce`, need a rule to apply to each item. You give the rule as an **anonymous function**: a name for each item, `give`, then the value to give back. Here is a whole statement, broken into its pieces:

```
import data

nums = list [1, 2, 3]
bigger = nums process x give x * 10
show bigger .
```

```
[ 10, 20, 30 ]
```

Piece	What it is	What it does here
`bigger`	a name you choose	the variable that will hold the result
`=`	assignment	stores the value on the right in `bigger`
`nums`	a list you made	the collection the work is done on
`process`	a library function	runs a rule on every item and collects the results
`x`	a name you choose	stands for one item at a time: first 1, then 2, then 3
`give`	the start of the rule	separates the name for each item from the work done on it
`x * 10`	the rule	what each item becomes: 1 becomes 10, 2 becomes 20, 3 becomes 30

So the statement reads: **take each item of `nums`, call it `x`, give back `x * 10`, and store the new list in `bigger`**. `nums` is not changed.

The same rule works without the name `bigger`, which changes `nums` itself (this is what the sentence form does on its own line):

```
nums process x give x * 10 .
show nums .
```

```
[ 10, 20, 30 ]
```

A map passes two names instead of one, the key and the value, in brackets:

```
ages = map ["Alice": 30, "Bob": 25]
older = ages keep [name, age] give age > 26
show older .
```

```
{ "Alice": 30 }
```

Here `name` stands for each key and `age` for each value. `keep` keeps the entries where the rule gives `true`.

A rule that needs more than one line uses a block between `give` and `give [end]`, and gives back its answer with `return`:

```
nums = list [1, 2, 3, 4, 5, 6, 7]
changed = nums process [x] give
    if ] x > 5 [
        return x * 2
    if [end]
    return x
give [end]
show changed .
```

```
[ 1, 2, 3, 4, 5, 12, 14 ]
```

Rules to remember:

- The name before `give` is yours. `x`, `item` and `n` all work.
- `process` and `keep` used as a value (after `=`) give a new collection and leave the original alone. Used on their own, they change the collection.
- For a map, write the names in brackets: `[name, age]`.
- The same call works in bracket form, `process[nums, x give x * 10]`, and in sentence form, `nums process x give x * 10`.

8

Collections

What a collection is. Many values often belong together: the scores in a game, the cities on a route, the prices in a shop. A **collection** holds those values under one name. Three kinds cover most needs, and choosing between them is the first decision a programmer makes.

- A **list** keeps its items in order, and the same item can appear more than once. Use it when order matters, such as a queue of tasks.
- A **set** keeps only distinct items, and its order has no meaning. Use it to ask "is this item already here?" or to remove repeats.
- A **map** pairs each **key** with a **value**. Use it to look up a value by its key, such as a person's name and age.

How Turtle does it. `list [4, 4, 9]`, `set [4, 4, 9]` and `map ["Ann": 30]` create the three kinds. Items in a list are read by position, starting at 0. Items in a map are read by their key. A **method** is an action that a value can perform, written after `at`, as in `nums at add[7]`.

A **collection** holds many values under one name. A **list** keeps its items in order and allows repeats. A **set** keeps only distinct items. A **map** pairs each **key** with a **value**. The choice of collection is one of the first decisions in most programs, so this chapter explains what each kind is for and how to change it.

8.1 Lists

A **list** is an ordered collection. Its items keep the order they were given, and the same value may appear more than once. Each item has a position, counted from zero. A list is written with `list` and square brackets:

```
nums = list [3, 1, 2]
show nums at get 0 .
show nums at length .
```

```
3
3
```

`get` reads the item at a position, and `length` gives the number of items. Changing a list is done by the statements in section 8.4, or by the method forms in section 8.5. The first item is at position 0, so the last item is at `length - 1`.

A list changes in place. Adding an item with `add` puts it at the end. `insert` puts an item at a chosen position, and `put` replaces the item at a position that already exists:

```
nums = list [3, 1, 2]
add 4 to nums .
insert 9 to nums at 0 .
put 7 to nums at 1 .
show nums .
```

```
[ 9, 7, 1, 2, 4 ]
```

`put` at a position past the end is an error. The list is unchanged, and the message gives the length of the list:

```
nums = list [1]
put 5 to nums at 3 .
```

```
turtle: line 2: index 3 out of range for list nums (length 1)
  2 | put 5 to nums at 3 .
```

Items are removed by value, not by position. `remove 1 from nums .` removes the first item equal to `1`. A list can also be sorted and reversed in place:

```
nums = list [3, 1, 2]
sort nums .
reverse nums .
show nums .
```

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

8.2 Sets

A **set** is a collection of distinct items. Adding an item that is already in the set does nothing, and the order of the items is not part of the set's meaning. A set drops duplicates as it is made:

```
s = set [1, 2, 2]
show s .
```

```
{ 1, 2 }
```

Items are added and removed with `add` and `remove`. Removing an item that is not in the set is not an error, so the set is simply unchanged:

```
s = set [1, 2]
add 3 to s .
add 3 to s .
remove 2 from s .
remove 9 from s .
show s .
```

```
{ 1, 3 }
```

`contains` asks whether an item is in the set, and `length` counts the items:

```
s = set [1, 2]
show s contains 2 .
show s contains 5 .
show length of s .
```

```
true
false
2
```

8.3 Maps

A **map** pairs each key with a value. Keys are unique within a map, and a key can be a number, text, a list or an assembly. A value is read with `get`, and a key is checked with `contains` before it is read, because `get` on a missing key is an error:

```
ages = map ["Alice": 30, "Bob": 25]
show ages at get "Alice" .
show ages contains "Carol" .
```

```
30
false
```

A value is changed by `put`, which sets the value for a key whether or not the key exists. `add` changes a number in place: it adds to the value already stored under the key, and a new key starts at the value given. A key is removed with `delete`. Maps have no `remove` statement:

```
ages = map ["Alice": 30]
put 31 to ages at "Alice" .
add 5 to ages at "Alice" .
put 25 to ages at "Bob" .
delete "Alice" from ages .
show ages .
```

```
{ "Bob": 25 }
```

Each key's value keeps the type it was given. `add` joins text and extends a list stored under a key, so a map can hold many kinds of values at once.

8.4 Changing many values at once

Changing one item at a time is done with the statements in section 8.5, and a loop does the same work by hand. When every item must change in the same way, the `data` library changes the whole list in one call. It needs `import data`:

- `process` gives a new list with a rule applied to each item,
- `keep` gives a new list of the items that pass a test,
- `reduce` combines the items into one value.

`process` does not change the original list, so the result must be kept under a new name:

```
import data
nums = list [1, 2, 3]
new = nums process x give x * 10
show new .
show nums .
```

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

The rule `x give x * 10` is applied to every item, and `nums` is still `[1, 2, 3]`. `keep` works the same way for selection:

```
import data
nums = list [1, 2, 3]
show nums keep x give x > 1 .
```

```
[ 2, 3 ]
```

Chapter 17 covers the `data` library in full. The chapters that follow show far more complex operations on collections: nested collections, grouping and sorting records, and building one collection from another. Each of them starts from the forms in this chapter.

8.5 Changing a collection with statements

Most changes read as a sentence, and end with a period. Each statement changes the collection in place:

```
nums = list [3, 1, 2]
add 4 to nums .
remove 1 from nums .
insert 99 to nums at 0 .
put 7 to nums at 1 .
sort nums .
reverse nums .
```

On a map, `put` and `delete` work with keys:

```
ages = map ["Alice": 30, "Bob": 25]
put 31 to ages at "Alice" .
delete "Bob" from ages .
```

Checking the size reads like English too:

```
nums = list [3, 1, 2]
show length of nums .
show min of nums .
show max of nums .
```

```
3
1
3
```

8.6 Methods read from left to right

When you want a value rather than a change, use `at`. A method works on the value just before it:

```
show nums at get 0 .
show nums at get[0] + nums at get[1] .
```

The statement form, `show nums at get 0 .`, is the shortest way to read one item. Use brackets when the index is an expression, as in `get[len - 1]`.

With or without brackets, a method's argument can be a single value. Use brackets when the argument is more than one thing, or when you want the whole calculation to be the argument. This example reads the last item of a list. Positions start at 0, so the last position of a three-item list is 2, which is its length minus 1:

```
nums = list [4, 9, 2]
show nums .
show nums at get[length of nums - 1] .
last = nums at length - 1
show nums at get last .
```

```
[ 4, 9, 2 ]
2
2
```

The list is in the order it was written, so position 0 holds `4` and the last position holds `2`. The first `show` reads the last item in one line, with brackets, because the index is a calculation. The second reads the same item in two steps: first the index is worked out and given a name, then it is used. Both give the same answer, and the two-step form is easier to follow when the index is complicated.

A method can also be written as a sentence. These two lines do the same thing:

```
show "a" + "b" at upper .
show ("a" + "b") at upper .
```

```
aB
AB
```

The first uppercases only `"b"`, because the method binds to the value just before it. The second uppercases the whole joined text. This is the only place precedence needs a moment's thought, and it is the same rule as arithmetic: a method sticks to the value on its left.

8.7 Joining collections

`+` and `-` work on two collections of the same kind, and always make a new one:

```
a = list [1, 2]
b = list [2, 3]
show a + b .
show a - b .
```

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

Sets combine with `+` (union) and `-` (difference). Maps combine with `+`, and where both have the same key, the left map's value wins.

8.8 Converting between kinds

`change` moves between kinds, and is the quickest way to drop duplicates:

```
nums = list [3, 1, 3, 2, 1]
show change nums to set .
show change "abc" to list .
```

```
{ 3, 1, 2 }
[ "a", "b", "c" ]
```

8.9 Looping over a map

A loop over a map gives the keys. Use `at get` for the values, or name both in a loop over `keys`:

```
ages = map ["Alice": 30, "Bob": 25]
[loop][name in change ages to keys]
    show "{name} is {ages at get name}" .
[loop][end]
```

```
Alice is 30
Bob is 25
```

8.10 Reference: built-in functions for lists, sets and maps

This section lists every built-in operation for the three collections. They need no import. A **method** is written `value at method[arguments]`, or with one argument as `value at method argument`. A **statement** ends with a period, and names the collection as its target. Methods that change a collection, such as `add`, `put`, `delete`, `sort` and `clear`, change it in place. Methods that give a new collection, such as `union` and `invert`, leave the original as it was.

Methods on lists

Method	Arguments	What it gives back or does
`add`	value	adds the value at the end (changes the list)
`length`, `len`	none	the number of items
`isempty`	none	`true` when there are no items
`tostring`	none	the list as text
`clear`	none	removes every item (changes the list)
`count`	value	how many items are equal to the value
`index`	value	the position of the first match, or `-1`
`contains`, `find`	value	`true` when the value is in the list
`get`	index	the item at the index; an error if the index is out of range
`slice`	start [, end]	a new list of the items from `start` up to `end`
`insert`	value, index	inserts the value at the index (changes the list)
`put`	value, index	replaces the item at the index (changes the list)
`remove`	value	removes the first item equal to the value (changes the list)
`pop`	[index]	removes and gives back the item; the last one by default
`sort`	none	sorts the list (changes the list)
`reverse`	none	reverses the list (changes the list)

Methods on sets

Method	Arguments	What it gives back or does
`add`	value	adds the value; a value already present is not added twice
`remove`	value	removes the value; a value not present is not an error
`insert`	value, index	puts the value at a position in the set's order (changes the set)
`get`	index	the item at a position in the set's order; an error if the index is out of range
`pop`	none	removes and gives back the item added most recently (changes the set)
`sort`	none	sorts the items (changes the set)
`length`, `len`	none	the number of items
`isempty`	none	`true` when the set is empty
`contains`, `find`	value	`true` when the value is in the set
`count`	value	`1` when the value is in the set, otherwise `0`
`clear`	none	removes every item
`tostring`	none	the set as text
`union`	other set	a new set with the items of both
`intersection`	other set	a new set with the items in both
`difference`	other set	a new set with the items of this set that are not in the other
`subset`	other set	`true` when every item of this set is in the other
`superset`	other set	`true` when every item of the other set is in this one

A set has no order that means anything, so `put` and `slice`, which are position-based methods of lists, are not available on a set. `insert` puts a value at a position, but this book adds to a set with `add`. Section 8.11 shows each set method with its output.

Methods on maps

Method	Arguments	What it gives back or does
`get`	key	the value for the key; an error if the key is missing
`contains`	key	`true` when the key is in the map
`length`, `len`	none	the number of entries
`isempty`	none	`true` when the map has no entries
`getkeys`	none	a list of the keys, in the order they were added
`getvalues`	none	a list of the values, in the same order
`add`	key, value	adds to the value at the key (changes the map)
`put`	key, value	sets the value at the key, new or not (changes the map)
`delete`	key	removes the key and its value (changes the map); an error if the key is missing
`invert`	none	a new map with keys and values exchanged; warns when two keys share a value and keeps the last key
`tostring`	none	the map as text

Statements

Statement	Applies to	Effect
`add value to target .`	list, set	adds the value (a set skips a value already present)
`add value to target at key .`	map	adds to the value at the key, or starts a new key
`remove value from target .`	list, set	removes the first match; no error if there is none
`delete key from target .`	map	removes the key; an error if the key is missing
`insert value to target at index .`	list	inserts at the index
`put value to target at index .`	list, map	replaces the list item, or sets the map value at the key
`sort target .`	list, set	sorts in place
`reverse target .`	list, set	reverses in place
`length of target .`	list, set, map	prints the count
`min of target .`, `max of target .`	list, set, map	prints the smallest or largest item (for a map, of its keys)

`min[...]` and `max[...]` are the same operations as expressions. `min[nums]` gives the smallest item of a list, and `max[3, 5]` gives the largest of the two values. They can be used anywhere a value can, which the `min of` statement cannot.

The following example uses a few of these operations together, so that each result can be read in order:

```
nums = list [4, 9, 2]
add 7 to nums .
remove 9 from nums .
show nums at index[7] .
show nums at slice[0, 2] .
show min[nums] .
show nums at pop .
show nums .
```

```
2
[ 4, 2 ]
2
7
[ 4, 2 ]
```

8.11 Every method, with examples

This section shows every method on lists, sets and maps, with the output of each example. Each example was run on the Turtle source at commit 84f9a10, which is Turtle v1.0.0. Each one is a complete program: paste it into a file and run it with `turtle`. Under each method name, the description says what the method gives back or changes. The last part of the section shows the errors you will see, and when they happen.

Lists

`add` adds a value at the end of the list (changes the list).

```
nums = list [4, 9]
nums at add[7]
show nums .
```

```
[ 4, 9, 7 ]
```

`get` gives back the item at an index. Positions start at 0.

```
nums = list [4, 9, 2]
show nums at get[1] .
```

```
9
```

`put` replaces the item at an index (changes the list). The value comes first, then the index.

```
nums = list [4, 9, 2]
nums at put[99, 1]
show nums .
```

```
[ 4, 99, 2 ]
```

`insert` inserts a value before an index, so the items after it move up (changes the list).

```
nums = list [4, 9, 2]
nums at insert[0, 1]
show nums .
```

```
[ 4, 0, 9, 2 ]
```

`remove` removes the first item equal to the value (changes the list). A value that is not there is not an error.

```
nums = list [4, 9, 2, 9]
nums at remove[9]
show nums .
nums at remove[42]
show nums .
```

```
[ 4, 2, 9 ]
[ 4, 2, 9 ]
```

`pop` removes an item and gives it back. With no index it takes the last item; with an index it takes that item.

```
nums = list [4, 9, 2]
show nums at pop .
show nums at pop[0] .
show nums .
```

```
2
4
[ 9 ]
```

`slice` gives a new list of the items from a start position up to, but not including, an end position. With no end, it goes to the end of the list. The original list is unchanged.

```
nums = list [4, 9, 2, 9]
show nums at slice[1, 3] .
show nums at slice[1] .
show nums .
```

```
[ 9, 2 ]
[ 9, 2, 9 ]
[ 4, 9, 2, 9 ]
```

`contains` gives `true` when the value is in the list. `find` is another name for `contains`.

```
nums = list [4, 9, 2]
show nums at contains[2] .
show nums at find[7] .
```

```
true
false
```

`index` gives the position of the first item equal to the value, or `-1` when there is none.

```
nums = list [4, 9, 2, 9]
show nums at index[9] .
show nums at index[7] .
```

```
1
-1
```

`count` gives how many items are equal to the value.

```
nums = list [4, 9, 2, 9]
show nums at count[9] .
```

```
2
```

`sort` sorts the list in place (changes the list).

```
nums = list [4, 9, 2]
nums at sort
show nums .
```

```
[ 2, 4, 9 ]
```

`reverse` reverses the list in place (changes the list).

```
nums = list [4, 9, 2]
nums at reverse
show nums .
```

```
[ 2, 9, 4 ]
```

`clear` removes every item (changes the list).

```
nums = list [4, 9, 2]
nums at clear
show nums .
```

```
[]
```

`len` gives the number of items. `length` is another name for `len`.

```
nums = list [4, 9, 2]
show nums at len .
show nums at length .
```

```
3
3
```

`isempty` gives `true` when the list has no items.

```
nums = list []
show nums at isempty .
nums at add[1]
show nums at isempty .
```

```
true
false
```

`tostring` gives the list as text.

```
nums = list [4, 9]
text = nums at tostring
show "the list is {text}" .
```

```
the list is [ 4, 9 ]
```

Sets

`add` adds a value. A value already in the set is not added twice (changes the set).

```
colors = set ["red", "blue"]
colors at add["red"]
colors at add["green"]
show colors .
```

```
{ "red", "blue", "green" }
```

`insert` puts a value at a position in the set's order (changes the set). The value comes first, then the position. Sets have no order that means anything, so the book uses `add` instead.

```
colors = set ["red", "blue"]
colors at insert["green", 0]
show colors .
```

```
{ "green", "red", "blue" }
```

`get` gives the item at a position in the set's order. Positions start at 0. Sets have no order that means anything, so use `get` only to look at the items one by one.

```
colors = set ["red", "blue", "green"]
show colors at get[0] .
show colors at get[2] .
```

```
red
green
```

`remove` removes a value (changes the set). A value that is not there is not an error.

```
colors = set ["red", "blue"]
colors at remove["red"]
colors at remove["pink"]
show colors .
```

```
{ "blue" }
```

`contains` gives `true` when the value is in the set. `find` is another name for `contains`.

```
colors = set ["red", "blue"]
show colors at contains["red"] .
show colors at find["pink"] .
```

```
true
false
```

`count` gives `1` when the value is in the set, and `0` when it is not.

```
colors = set ["red", "blue"]
show colors at count["blue"] .
show colors at count["pink"] .
```

```
1
0
```

`union` gives a new set with the items of both sets. The original sets are unchanged.

```
a = set [1, 2]
b = set [2, 3]
show a at union[b] .
show a .
```

```
{ 1, 2, 3 }
{ 1, 2 }
```

`intersection` gives a new set with the items that are in both sets.

```
a = set [1, 2]
b = set [2, 3]
show a at intersection[b] .
```

```
{ 2 }
```

`difference` gives a new set with the items of this set that are not in the other set.

```
a = set [1, 2]
b = set [2, 3]
show a at difference[b] .
```

```
{ 1 }
```

`subset` gives `true` when every item of this set is in the other set.

```
a = set [1, 2]
b = set [1, 2, 3]
show a at subset[b] .
show b at subset[a] .
```

```
true
false
```

`superset` gives `true` when every item of the other set is in this set.

```
a = set [1, 2]
b = set [1, 2, 3]
show b at superset[a] .
show a at superset[b] .
```

```
true
false
```

`pop` removes an item and gives it back. It takes the item added most recently (changes the set).

```
colors = set ["red", "blue"]
show colors at pop .
show colors .
```

```
blue
{ "red" }
```

`sort` sorts the items (changes the set). The result is the same set, shown in sorted order.

```
nums = set [4, 8, 1]
nums at sort
show nums .
```

```
{ 1, 4, 8 }
```

`clear` removes every item (changes the set).

```
colors = set ["red", "blue"]
colors at clear
show colors .
```

```
{}
```

`len` gives the number of items. `length` is another name for `len`.

```
colors = set ["red", "blue"]
show colors at len .
```

```
2
```

`isempty` gives `true` when the set has no items.

```
colors = set []
show colors at isempty .
```

```
true
```

`tostring` gives the set as text.

```
colors = set ["red"]
text = colors at tostring
show "the set is {text}" .
```

```
the set is { "red" }
```

Maps

`get` gives the value for a key. A key that is not in the map is an error, so check with `contains` first when it may be missing.

```
ages = map ["Alice": 30, "Bob": 25]
show ages at get["Alice"] .
```

```
30
```

`add` adds to the value at a key (changes the map). For a number, the amount is added to the value, and a new key starts from the amount.

```
ages = map ["Alice": 30]
ages at add["Alice", 1]
ages at add["Cy", 5]
show ages .
```

```
{ "Alice": 31, "Cy": 5 }
```

`put` sets the value at a key, whether or not the key is there (changes the map).

```
ages = map ["Alice": 30]
ages at put["Alice", 31]
ages at put["Dee", 19]
show ages .
```

```
{ "Alice": 31, "Dee": 19 }
```

`delete` removes a key and its value (changes the map). A key that is not there is an error.

```
ages = map ["Alice": 30, "Bob": 25]
ages at delete["Alice"]
show ages .
```

```
{ "Bob": 25 }
```

`contains` gives `true` when the key is in the map.

```
ages = map ["Alice": 30]
show ages at contains["Alice"] .
show ages at contains["Zed"] .
```

```
true
false
```

`getkeys` gives a list of the keys, in the order they were added.

```
ages = map ["Bob": 25, "Alice": 30]
show ages at getkeys .
```

```
[ "Bob", "Alice" ]
```

`getvalues` gives a list of the values, in the same order as the keys.

```
ages = map ["Bob": 25, "Alice": 30]
show ages at getvalues .
```

```
[ 25, 30 ]
```

`invert` gives a new map with each key and value exchanged. The values become the keys. When two keys share a value, Turtle prints a warning and keeps only the last key.

```
ages = map ["Bob": 25, "Alice": 30]
show ages at invert .
show ages .
```

```
{ 25: "Bob", 30: "Alice" }
{ "Bob": 25, "Alice": 30 }
```

`len` gives the number of entries. `length` is another name for `len`.

```
ages = map ["Bob": 25, "Alice": 30]
show ages at len .
```

```
2
```

`isempty` gives `true` when the map has no entries.

```
ages = map []
show ages at isempty .
```

```
true
```

`tostring` gives the map as text.

```
ages = map ["Bob": 25]
text = ages at tostring
show "the map is {text}" .
```

```
the map is { "Bob": 25 }
```

Errors and limits

Each of these stops the program with an error. The message names the problem, and the line shows where it happened.

A list position past the end is an error.

```
nums = list [4]
show nums at get[5] .
```

```
turtle: line 2: index 5 out of range for list (length 1)
  2 | show nums at get[5] .
```

`pop` on an empty list is an error.

```
nums = list []
show nums at pop .
```

```
turtle: line 2: 'pop' called on an empty list
  2 | show nums at pop .
```

A map key that is not there is an error for `get` and for `delete`.

```
ages = map ["Bob": 25]
show ages at get["Zed"] .
```

```
turtle: line 2: key "Zed" not found in map
  2 | show ages at get["Zed"] .
```

`put` is not a method of a set.

```
colors = set ["red"]
colors at put["blue", 0]
```

```
turtle: line 2: unknown set method "put"
  2 | colors at put["blue", 0]
```

`clear` is not a method of a map.

```
ages = map ["Bob": 25]
ages at clear
```

```
turtle: line 2: unknown map method "clear"
  2 | ages at clear
```

A set position past the end is an error.

```
colors = set ["red", "blue"]
show colors at get[5] .
```

```
turtle: line 2: index 5 out of range for set (length 2)
  2 | show colors at get[5] .
```

When two keys in a map share a value, `invert` warns and keeps only the last key.

```
ages = map ["Bob": 25, "Ann": 25]
show ages at invert .
```

```
warning: 25 is the value of both "Bob" and "Ann", so the inverted map keeps only
    "Ann" (line 2)
{ 25: "Ann" }
```

8.12 A real program: a phone book

A map is a set of keys with values, and this short program uses every common map operation: a lookup, an insert, a change, a size, and a membership test. It is in the Complexity corpus, and it runs as it is:

```
// A tiny phone book: look up, add and change entries.
book = map ["Ana": "555-0101", "Ben": "555-0102"]
book at put["Cy", "555-0103"]
book at put["Ben", "555-0199"]
show "Ana: ", book at get["Ana"] .
show "Ben: ", book at get["Ben"] .
show "entries: ", book at len .
show "Dee in book? ", book at contains["Dee"] .
```

```
Ana: 555-0101
Ben: 555-0199
entries: 3
Dee in book? false
```

The same map is changed by `put`, which replaces a value and adds a key that is new. Looking up a missing key with `get` is an error, so `contains` is the check to make first. The sentence forms, `book get "Ana"` and `book contains "Dee"`, do the same work. Chapter 16 shows both forms side by side.

Best practice. Choose the kind by the question the program asks: a list when order matters, a set when you only need to know whether something is present, a map when you look things up by a key. Check whether an item exists before you read it.

Common mistakes. Reading a position past the end of a list, which stops with `index 5 out of range`. Asking a map for a key it does not have: `get` stops with `key "Zed" not found`. Taking an item from an empty list with `pop`. Using a list where repeats are not wanted, so duplicates appear in the output.

Key points

- **Collection**: holds many values under one name. A **list** keeps order and allows repeats, a **set** keeps distinct items, and a **map** pairs keys with values.
- **Changing a collection**: most changes are statements that read as sentences and end with a period, such as `add 4 to nums .`
- **Methods** read from left to right with `at`, and give a value without changing the collection.
- `+` **and** `-` join or subtract two collections of the same kind, and always make a new one.
- `change` converts between kinds, and is the quickest way to drop duplicates.
- **Reference**: section 8.10 lists every built-in operation for lists, sets and maps, and section 8.11 shows each one with an example.

9

Assemblies

What a record is. Some data always has the same parts, such as an order with an item, a quantity and a price. A **record** groups those parts under one value, and each part is a **field** with its own name. A record is different from a list, because its fields have names rather than positions, and different records can have different fields.

How Turtle does it. The `assemble` statement names the shape: `assemble Order [item, qty, price]`. Each value of that shape is written with the same name, as in `Order["pen", 3, 1.5]`, and its fields are read with `of`, such as `qty of o`. Use an assembly when the same group of parts appears again and again, and a map when the keys are not fixed in advance.

An **assembly** gives a fixed set of named **fields** to a kind of value. The `assemble` statement defines a new assembly type, and every value of that type has the same fields. Assembly types let a program define as many kinds of structured data as it needs. Use an assembly when the same group of values appears many times, such as a point, an order, or a student.

When the same set of named pieces keeps appearing, such as an order with an item, a quantity and a price, name the assembly once with `assemble`:

```
assemble Order [item, qty, price]

o = Order["pen", 3, 1.5]
show o .
show qty of o .
```

```
Order { item: "pen", qty: 3, price: 1.5 }
3
```

`Order` is now a function that makes orders. It takes one value per field, in order. Reading a field uses `of`, which binds tighter than arithmetic, so the next line multiplies two fields:

```
assemble Order [item, qty, price]
o = Order["pen", 3, 1.5]
qty of o = 10
total = qty of o * price of o
show total .
```

```
15.0
```

Changing a field also uses `of`. The statement `qty of o = 10` changes the field in place.

9.1 Changing field values

A field is read with `of`, and changed with an assignment that names the field. The change is made in the assembly itself. The assembly is the same value, with one field different, so every name that refers to the assembly sees the change:

```
assemble Order [item, qty, price]
o = Order["pen", 3, 1.5]
qty of o = 10
show o .
price of o = price of o * 2
show o .
```

```
Order { item: "pen", qty: 10, price: 1.5 }
Order { item: "pen", qty: 10, price: 3.0 }
```

The second assignment reads `price of o`, multiplies it, and stores the result in the same field. An assignment such as `qty of o = qty of o + 1` is the usual way to change a field by a step.

An assembly kept in a list can be changed in a loop. The loop variable names the same assembly as the list item, so the change is visible in the list afterwards:

```
assemble Order [item, qty, price]
orders = list [Order["pen", 3, 1.5], Order["ink", 1, 4.0]]
[loop][o in orders]
    qty of o = qty of o * 2
[loop][end]
show orders .
```

```
[ Order { item: "pen", qty: 6, price: 1.5 }, Order { item: "ink", qty: 2, price: 4.0
    } ]
```

A function can change an assembly it receives, for the same reason. The caller sees the change, because the function was given the value and not a copy:

```
assemble Order [item, qty, price]

def discount[o, pct]
    price of o = price of o * (100 - pct) / 100
def [end]

orders = list [Order["pen", 3, 1.5], Order["ink", 1, 4.0]]
discount[orders at get 0, 10]
show orders at get 0 .
```

```
Order { item: "pen", qty: 3, price: 1.35 }
```

When a change must not reach the original, make a copy first with `copy[value, true]` (chapter 10). A plain assignment, `p = o`, gives a second name for the same assembly, not a second assembly.

A field that holds a map or a list is changed in the same way, through the field name. An assembly stored inside a map is changed through the map's `get`, in parentheses:

```
assemble Order [item, qty, price]
m = map ["a": Order["pen", 3, 1.5]]
qty of (m at get "a") = 7
show m at get "a" .
```

```
Order { item: "pen", qty: 7, price: 1.5 }
```

Two rules apply to every change. A field name that is not in the assembly is an error, and the error lists the names that exist:

```
assemble Order [item, qty, price]
o = Order["pen", 3, 1.5]
prise of o = 2
```

```
turtle: line 3: Order has no field "prise" (its fields: item, qty, price)
  3 | prise of o = 2
```

The second rule is that the names of the fields are fixed, but their values are not checked. A field can be given a value of any type, so `qty of o = "many"` is accepted. Keep the type of each field consistent in your own code, because Turtle will not stop a change of type.

9.2 Mistakes name the fields

A misspelled field, or a missing value, is an error that lists the fields that exist, so you see the problem at once:

```
Order has no field "prise" (its fields: item, qty, price)
Order needs 3 value(s), one per field (item, qty, price), got 2
```

9.3 Equality

Two orders are equal when they came from the same `assemble` and their fields match. Two assemblies with the same name from different files are never equal, which keeps unrelated data apart.

9.4 Assemblies in functions

An assembly is a value, so it can be stored and passed to a function. A function can return an assembly, and a function can read one, which is how Turtle gives you a small "record" without classes:

```
assemble Point [x, y]

def midpoint[a, b]
    return Point[(x of a + x of b) / 2, (y of a + y of b) / 2]
def [end]

m = midpoint[Point[0, 0], Point[4, 6]]
show m .
```

```
Point { x: 2.0, y: 3.0 }
```

The parentheses there group a calculation for clarity. You never need them to make a call.

9.5 A real program: a point record

`assemble` names an assembly with fields. `Point[3, 4]` makes one, `x of p` reads a field, and `x of p = 10` changes it. Two points with the same fields are equal, as the last line shows.

```
// An assembled type gives a name and fields to a kind of record.
assemble Point [x, y]
p = Point[3, 4]
q = Point[-1, 2]
show p .
show x of p, " ", y of p .
x of p = 10
show p .
show p == Point[10, 4] .
```

```
Point { x: 3, y: 4 }
3 4
Point { x: 10, y: 4 }
true
```

No class was written. The assembly is a name, a list of fields, and nothing more.

Best practice. Name the shape after the thing it describes, and keep its fields the same in every value. If a field may be missing, decide on the value to use for it, such as `none`.

Common mistake. Reading or changing a field the shape does not have. Turtle stops with `Order has no field "prise"` and lists the fields it does have, which is usually a typo in the name.

Key points

- **Assembly**: a fixed set of named **fields** given to a kind of value, defined with `assemble`.
- `of` reads a field, and an assignment that names the field changes it.
- **Field errors**: a misspelled field is an error, and the message lists the fields that exist.
- **Equality**: two assembled values are equal when they come from the same `assemble` and have the same fields.
- **Passing**: an assembly is a value, so it can be stored and passed to a function.

10

Nested structures

What nesting means. Real data has layers. A school has classes, a class has students, and each student has grades. A **nested structure** is a collection or record that holds other collections or records as its values.

Why it matters. Nesting lets one value describe a whole group of related things. The trade-off is that each layer needs its own careful read: you must know which layer a value is in before you ask for it.

How Turtle does it. A list can hold lists, a map can hold lists or maps, and an assembly can hold collections. Positions and keys are chained one after another, as in `school at get["class A"] at get 0`.

Real data is rarely flat. A school has classes, a class has students, and a student has grades. A value that holds other values is a **nested structure**: a map whose values are lists, a list of records, a set of named assemblies. Each level of nesting is one more step to read or change, so the way you write the steps matters as much as what you store.

Turtle nests freely. A list can hold maps, a map can hold lists and maps, a set can hold lists, and an assembled value (chapter 9) can hold any of them. This chapter shows how to read, change, copy and loop over nested values, and where a value is shared rather than copied, which is the rule that causes most surprises.

10.1 Reading a value inside another

Each `at get` takes one step into the structure. Read the steps left to right:

```
school = map ["ann": map ["grades": list [90, 85]], "bo": map ["grades": list [70]]]
show school at get["ann"] at get["grades"] .
show school at get["ann"] at get["grades"] at get[1] .
```

```
[ 90, 85 ]
85
```

The first line takes the map under `"ann"`, then its `"grades"` list. The second line takes one more step, into the list, to reach the second grade.

10.2 Changing a value inside another

To change a nested list, take it out first and name it. Then change it by name, as you would any list:

```
import data
school = map ["ann": map ["grades": list [90, 85]], "bo": map ["grades": list [70]]]
g = school at get["bo"] at get["grades"]
add 80 to g .
show school at get["bo"] .
```

```
{ "grades": [ 70, 80 ] }
```

The list `g` is the same list stored in `school`, not a copy of it, so adding to `g` changes `school`. The next section explains when this happens.

Keys are added and removed the same way:

```
school = map ["ann": map ["grades": list [90, 85]], "bo": map ["grades": list [70]]]
put list [1, 2] to school at "cy" .
delete "bo" from school .
show getkeys[school] .
```

```
[ "ann", "cy" ]
```

`put` adds a key that is new, or replaces one that exists. `delete` removes a key, and is an error if the key is not there.

10.3 Shared and copied values

A plain assignment shares a value. Two names can refer to the same list, map or assembled value, and a change through one name shows through the other. A deep copy makes an independent value, and changes to it leave the original alone:

```
import data
school = map ["ann": map ["grades": list [90, 85]]]
g = school at get["ann"] at get["grades"]
add 95 to g .
c = copy[school, true]
add 60 to g .
show c .
show school .
```

```
{ "ann": { "grades": [ 90, 85, 95 ] } }
{ "ann": { "grades": [ 90, 85, 95, 60 ] } }
```

The copy `c` was made after the `95` and before the `60`, so it keeps only the values that were there when it was copied. Use `copy[value, true]` whenever a function must change a structure

without changing the caller's copy.

10.4 Looping over records

A list of maps is the usual shape for rows. A loop reads each record, and `at get` reads a field from it:

```
rows = list [map ["n": "a", "tags": list ["x"]], map ["n": "b", "tags": list []]]
[loop][r in rows]
    show r at get["n"], " has ", length of (r at get["tags"]), " tags" .
[loop][end]
```

```
a has 1 tags
b has 0 tags
```

Parentheses group the `at get` so that `length of` applies to the list. They are not needed for a call, but they make a nested expression easier to read.

10.5 Sets and maps of assemblies

Sets and map keys can hold lists and assembled values, not only numbers and text. Two equal assemblies count once in a set, and an assembly can be a map key:

```
assemble Point [x, y]
s = set [Point[1, 2], Point[1, 2], Point[3, 4]]
show length of s .
m = map [Point[0, 0]: "origin"]
show m at get[Point[0, 0]] .
```

```
2
origin
```

10.6 Assemblies that hold lists

An assembled value can hold a list. Reading the list through the assembly gives the same list, so changing it through either name changes the assembly:

```
assemble Student [name, grades]
s = Student["Ann", list [90, 85]]
g = grades of s
add 70 to g .
show s .
```

```
Student { name: "Ann", grades: [ 90, 85, 70 ] }
```

This is how a record stays small and readable: the fixed fields are named, and a list field holds the many values for that record.

10.7 Good practice

- **Name the steps.** A chain of three `at get` calls is harder to read than three lines that each take one step and name the result.
- **Use an assembly for a fixed record.** When every record has the same fields, an `assemble` type makes the fields explicit, and a mistyped field name is an error that lists the fields that exist.
- **Copy before you change a shared structure** that the caller still uses.
- **Keep the structure as flat as the problem allows.** Each level of nesting is one more step to read, and one more place for a mistake.

Best practice. Keep nesting shallow. If a structure is three or more layers deep, consider a named assembly for each layer, so each value says what it is.

Common mistake. Thinking that copying a list makes an independent copy. In Turtle, `b = a` makes both names refer to the same list, so a change made through `b` also shows through `a` (`copy[value, true]`, from `import data`, makes an independent copy).

Key points

- **Nested values**: each `at get` takes one step into a structure, and the steps are read from left to right.
- **Changing inside**: take the inner value out and give it a name, then change it by that name.
- **Shared and copied**: a plain assignment shares a list, map or assembly, so two names can change the same value.
- **Records**: a list of maps is the usual shape for rows, and a loop reads each record in turn.
- **Keys and sets** can hold assembled values, not only numbers and text.
- **Assemblies** can hold lists, and reading the list back goes through the assembly's field.
- **Good practice**: name the steps in a long chain of `at get` calls, so each one is easy to follow.

11

Scrolls

What a pipeline is. Many jobs are a chain of steps: take the data, clean it, filter it, then add it up. Each step's output is the next step's input. This shape is called a **pipeline**, and it is easy to read when each step is named.

How Turtle does it. A **scroll** is a value that holds a sequence of steps. It is applied to a starting value with `into`, and the result of each step becomes the input of the next. Scrolls read from top to bottom, in the same order the data flows.

A **scroll** is a value that holds a sequence of steps. Applying a scroll to a starting value passes that value through each step in order, and the result of one step becomes the input of the next. Scrolls read from top to bottom, so a long transformation can be followed line by line. They also need no temporary names for the values between steps.

Scrolls serve three purposes. They are a form of **abstraction**: a long series of steps can be given one name and used as a single operation, the same way a theory hides its detail. They make it possible to build **complex pipelines**, where a value is cleaned, checked, converted and summarized in order, and each stage can be changed or reused without touching the others. And they write a process in a **simplified form**: the reader sees the stages of the work in the order they happen, instead of a sequence of temporary names and nested calls.

This chapter explains what a scroll is and how its steps are written, then shows saved scrolls, scrolls applied to lists, errors inside a scroll, and a complete program that uses one.

11.1 What a scroll is

Without a scroll, a chain of calls is written inside out. The value is in the middle, and each call wraps the one before it:

```
def add1[x]
    return x + 1
def [end]

def double[x]
    return x * 2
def [end]

def half[x]
    return x / 2
def [end]

show half[double[add1[3]]] .
```

```
4.0
```

With a scroll, the same work is written in the order it happens. The word `scroll` introduces the starting value, `into` introduces the steps, and each step is separated by a comma. The period ends the scroll:

```
def add1[x]
    return x + 1
def [end]

def double[x]
    return x * 2
def [end]

def half[x]
    return x / 2
def [end]

x is scroll 3 into add1, double, half .
show x .
```

```
4.0
```

The value 3 goes into `add1`, which gives 4. That goes into `double`, which gives 8, and then into `half`, which gives 4.0. The `is` form stores the result in `x`, and `x = scroll 3 into add1, double, half .` does the same thing.

11.2 Steps

A step is written in one of three ways, depending on what the function needs besides the value:

The step is	Written as	Means
a function that needs only the value	`double`	`double[value]`
a method	`at upper`	`value at upper`
a function that needs other values	`max[here, 5]`	that call, with the value at `here`

By default the value goes first in the call. The word `here` puts the value somewhere else in the call, and it can also stand alone in an expression:

```
def add1[x]
    return x + 1
def [end]

show scroll 3 into add1, here * 10 .
show scroll 5 into list [here, here * 2] .
```

```
40
[ 5, 10 ]
```

A step can also be a method, written with `at`, or a `give` function:

```
show scroll "  hi  " into at trim, at upper .
show scroll 4 into n give n * 5 .
```

```
HI
20
```

11.3 Saved scrolls

A scroll that has no starting value is a **saved scroll**. It can be stored under a name, and it is applied later by calling it with brackets, the way a function is called. The name then holds a value of type `scroll`:

```
def add1[x]
    return x + 1
def [end]

def double[x]
    return x * 2
def [end]

s = scroll add1, double .
show s[3] .
show typeof[s] .
```

```
8
scroll
```

A saved scroll can also be used as one step of a larger scroll. Its steps run in place, in order, so the two scrolls below are equivalent:

```
def add1[x]
    return x + 1
def [end]

def double[x]
    return x * 2
def [end]

first = scroll add1 .
second = scroll double .
show scroll 2 into first, second .
show scroll 2 into add1, double .
```

```
6
6
```

11.4 Scrolls over lists

When the starting value is a list, each step receives the whole list. That suits functions such as `sum` and `keep`, which work on a collection. To change each item in a list, use `process` inside the scroll:

```
import data
rows = list [map ["item": "pen", "amount": 1200], map ["item": "ink", "amount": 300],
    map ["item": "desk", "amount": 2100]]
big = scroll rows into keep r give r at get["amount"] > 1000 .
show length of big .
show scroll rows into process r give r at get["item"] .
show scroll rows into keep r give r at get["amount"] > 1000, process r give r at
    get["item"] .
```

```
2
[ "pen", "ink", "desk" ]
[ "pen", "desk" ]
```

The last line is a two-step scroll. The first step keeps the rows whose amount is over 1000, and the second step takes the item name from each row that is left. The `give` function in the first step ends at the comma that starts the next step.

A saved scroll works on a list too. Here the saved scroll trims each text in a list:

```
import data
clean = scroll process s give s at trim .
show clean[list [" a ", " b "]] .
```

```
[ "a", "b" ]
```

11.5 Errors inside a scroll

An error in one step ends the whole scroll at that step, because the steps after it depend on its result. The message names the step, and the error keeps its kind, so `handle` can still catch it:

```
show scroll 0 into 10 / here .
```

```
turtle: line 1: scroll step 1 of 1 (10 / here): division by zero
  1 | show scroll 0 into 10 / here .
```

A step that gives `none` is not an error. The value `none` is passed to the next step, as the value of any other step would be.

11.6 A complete program

This program uses a scroll to total the large orders in a list. The first step keeps the orders over a threshold, and the second step takes each order's amount. The third step adds the amounts, and the result is shown as text:

```
import data

orders = list [map ["id": 1, "amount": 1200], map ["id": 2, "amount": 300], map
    ["id": 3, "amount": 2100]]

total = scroll orders into keep o give o at get["amount"] >= 1000, process o give o
    at get["amount"], sum[here] .
show "total over 1000: {total}" .
```

```
total over 1000: 3300
```

Read the scroll from left to right. Each step is one plain function call or one `give` function, and each step's result is the input of the next. Put a scroll where a chain of nested calls would be hard to read.

Best practice. Give each step a name that says what it does, and keep steps small enough to test one at a time. Test each step on its own before you join them.

Common mistake. Building one long step that does everything, which makes a failure hard to find. Split it, and the error will point to the step that failed.

Key points

- **Scroll**: a value that holds a sequence of steps. Applying it to a value runs the steps in order.
- **Reading order**: a scroll avoids the inside-out nesting of a long chain of calls.
- **Steps**: each step is written one of three ways, depending on what its function needs.
- **Saved scrolls**: a scroll with no starting value can be stored under a name and used later.
- **Lists**: when the starting value is a list, each step receives the whole list.
- **Errors**: an error in one step ends the whole scroll at that step.

12

Theories: teaching Turtle new words

What a theory is. In everyday programming, a **theory** is a rule for a new word. It lets a program say something in its own terms, such as `tally 1 in nums`, instead of a longer call. A theory also has checks that show the rule works. In mathematics a theorem is a claim that has been proved; in Turtle, a theory's checks are run automatically, and `turtle test` reports whether they hold.

Why it matters. When a program grows, the same idea appears many times. A theory names the idea once, states what must always be true about it, and gives examples that prove the rule. Readers can then trust the word without reading its code.

How Turtle does it. A theory has five parts: a plain-language **abstract**, a **notation** that shows how the word is written, a **definition** with the code, **theorem** lines that every result must meet, and **proof** examples that must give the right answers.

A **theory** is a named piece of knowledge that Turtle can use by its name. A theory has a phrase, which is the way it is written in a program, and a definition, which is the code that does the work. Once a theory exists, a program uses the phrase as if it were part of the language itself.

A theory serves four purposes. It adds a new word to the vocabulary of a program. It gives a name to an abstraction, so a whole calculation is read as one idea. It allows a phrase to read like a sentence, with its values placed between the words. And it hides deep complexity: the definition can be long and careful, while the code that uses the theory stays short. A **promise**, written as a theorem, states what the theory guarantees, and a **proof** gives examples that must hold.

A theory is the most distinctive part of Turtle, and the one most worth learning slowly. You can write a great many programs without ever writing a theory. But a good theory makes a whole program read like the problem it solves.

12.1 A first theory

Here is a theory for counting how many times an item appears in a list:

```
theory tally
    abstract
        tally says how many times b appears in the list a.
    notation tally b in a .
    definition
        n = 0
        [loop][x in a]
            if ] x == b [
                n = n + 1
            if [end]
        [loop][end]
        return n
    theorem result >= 0
    theorem result <= length of a
    proof
        tally 1 in list [1, 1, 0, 3] . is 2
        tally 9 in list [] . is 0
theory [end]

nums = list [1, 1, 0, 3]
show "ones: ", tally 1 in nums .
```

```
ones: 2
```

Read the theory section by section.

Part	What it does	In the theory
abstract	says what the **phrase** means, in plain words. The editor and `turtle doc` show it.	`tally says how many times b appears in the list a.`
notation	shows how the phrase is **written**: the word, then the **values**.	`tally b in a .`
definition	is the **code**. It uses the names from the notation as its values, and `return` gives the **result**.	a loop that counts matches, then `return n`
theorem	lines are **promises** that every result must keep.	the count is never negative, and never larger than the list
proof	holds **examples** that must come out right. If one is wrong, the theory is wrong, and `turtle test` reports it.	`tally 9 in list []` must be `0`

Once the theory is written, the phrase reads like part of the language.

12.2 Values and fixed words

In the notation, each **name** the definition uses is a **value**: a slot you fill in. Every other word is **fixed**, and is part of the phrase itself. In `tally b in a`, the words `tally` and `in` are fixed, and `b` and `a` are values.

This is also why the names in the notation must match the definition. A value that the definition never uses cannot do anything, so Turtle refuses the phrase and says so:

```
theory grab
    abstract
        grab takes the first item of a list.
    notation grab n in g .
    definition
        return g at get 0
theory [end]
show grab 1 in list [4] .
```

```
turtle: program.turtle, line 8: this isn't how grab is written: grab n in g .
  8 | show grab 1 in list [4] .
    |      ^
```

Here `n` is named in the notation but never used in the definition. Either use it, or leave it out of the notation.

12.3 Rules for a theory

A theory is made of parts. Some are required, and Turtle stops with an error if one is missing. Others are optional, and a theory works without them. The table lists every part in the order they are written.

Part	Required?	What it does	If it is missing
`theory word`	required	starts the theory and names its word	not a theory
`abstract`	required	says what the word means, in plain words; it should name the word	error: needs an abstract
`notation`	required, one or more	says how the word is written, with the values in it	error: needs a notation
`definition`	required	the code that does the work, ending in `return`	error: needs a definition
`theorem`	optional, any number	a promise that every result keeps	nothing is checked against it
`proof`	optional	examples that must come out right	a warning, `unproven`, when `turtle test` runs
`theory [end]`	required	closes the theory	error: the theory is not closed

The required parts are the ones that say what the word is and what it does: what it means (`abstract`), how it is written (`notation`), and how it works (`definition`). Turtle will not run a theory without them, and it reports the missing one by name.

The optional parts are the ones that check the theory. A **theorem** is written once and applies to every result. A **proof** is a set of examples, and `turtle test` runs them. A theory with no proof still runs, but `turtle test` warns that it is unproven. Chapter 21 covers testing theories in full.

The rules for the word follow.

- The word after `theory` must be new. It cannot be a keyword, a function, a method, a variable, or another theory.
- A theory comes before its first use, when the file is read from top to bottom. Put theories near the top of the file, after the imports.
- A theory is written at the top level of a file, never inside a function.
- Two theories with the same word cannot be imported into the same file.
- `theory ~name` makes a theory private to its own file, like a function whose name starts with `~`. A file that imports it cannot use the word, but its own tests can test it.

12.3a Several notations, theorems and proofs

A theory can say more than one thing about itself. It can have several **notations**, which are different ways to write the same phrase. Each notation names the values the definition uses, so the definition is written once. It can have several **theorems**, and each one is checked. Its **proof** can hold several checks, and each one is run in turn.

The theory below has two notations, `tally b in a` and `tally b among a`. Both mean the same thing, and the same definition serves both. It has two theorems and three proof checks:

```
theory tally
    abstract
        tally says how many times b appears in the list a.
    notation tally b in a .
    notation tally b among a .
    definition
        n = 0
        [loop][x in a]
            if ] x == b [
                n = n + 1
            if [end]
        [loop][end]
        return n
    theorem result >= 0
    theorem result <= length of a
    proof
        tally 1 in list [1, 1, 0, 3] . is 2
        tally 9 in list [] . is 0
        tally 1 among list [1, 1] . is 2
theory [end]

nums = list [1, 1, 0, 3]
show "ones: ", tally 1 in nums .
show "ones among: ", tally 1 among nums .
```

```
ones: 2
ones among: 2
```

Save the same theory in a file named `test_tally.turtle`, with `import test` at its top, and `turtle test` checks every check and theorem in it. The timing in the output changes on each run:

```
import test
theory tally
    abstract
        tally says how many times b appears in the list a.
    notation tally b in a .
    notation tally b among a .
    definition
        n = 0
        [loop][x in a]
            if ] x == b [
                n = n + 1
            if [end]
        [loop][end]
        return n
    theorem result >= 0
    theorem result <= length of a
    proof
        tally 1 in list [1, 1, 0, 3] . is 2
        tally 9 in list [] . is 0
        tally 1 among list [1, 1] . is 2
theory [end]

nums = list [1, 1, 0, 3]
show "ones: ", tally 1 in nums .
show "ones among: ", tally 1 among nums .
```

```
test_tally.turtle
  PASS  theory tally   proof 3 of 3, 2 theorems hold on 100 random inputs

ok: 0 passed, 0 failed, 1 theory proven (1 file, 452.9µs)
```

Each theorem is checked on the proof cases and on 100 random inputs. If a theorem is wrong for any of them, the theory fails. Adding `theorem result > 1` to this theory makes `turtle test` fail on the proof case `tally 9 in list []`, where the result is `0`.

12.4 Using a theory with a value of its own

A value can be any expression, not only a name. A phrase's value reads through arithmetic, but not past a comparison, `&&` or `||`:

```
theory tally
    abstract
        tally says how many times b appears in the list a.
    notation tally b in a .
    definition
        n = 0
        [loop][x in a]
            if ] x == b [
                n = n + 1
            if [end]
        [loop][end]
        return n
    theorem result >= 0
    theorem result <= length of a
    proof
        tally 1 in list [1, 1, 0, 3] . is 2
        tally 9 in list [] . is 0
theory [end]
nums = list [1, 1, 0, 3]
show tally 1 in nums == 2 .
```

```
true
```

The comparison is outside the phrase, so the phrase is counted first and then compared with `2`.

The arithmetic rule is a trap, so watch for it. The `+` is read as part of the value, so this line means "tally 1 in the list `nums + 1`":

```
theory tally
    abstract
        tally says how many times b appears in the list a.
    notation tally b in a .
    definition
        n = 0
        [loop][x in a]
            if ] x == b [
                n = n + 1
            if [end]
        [loop][end]
        return n
    theorem result >= 0
    theorem result <= length of a
    proof
        tally 1 in list [1, 1, 0, 3] . is 2
        tally 9 in list [] . is 0
theory [end]
nums = list [1, 1, 0, 3]
show tally 1 in nums + 1 .
```

```
turtle: line 20: '+' can't add list and integer — a list, set or map only adds to
    another of its own kind, e.g. list [1] + list [2]
  20 | show tally 1 in nums + 1 .
```

To add one to the count, put the phrase in brackets first: `show (tally 1 in nums) + 1 .`

12.5 Generators: a function as a value

A phrase can take a function as one of its values, as long as its definition calls that function. A **generator** is a function that gives a new value each time it is called, and a theory can make it read like a word:

```
def make_counter[]
    state = list [0]
    def next[]
        state at put[state at get[0] + 1, 0] .
        return state at get[0]
    def [end]
    return next
def [end]

theory yield
    abstract
        yield says the next value of gen.
    notation yield gen .
    definition
        value = gen[]
        return value
theory [end]

c = make_counter[]
show yield c .
show yield c .
show yield c .
```

```
1
2
3
```

The counter keeps its count in a list, not in a plain variable. A function can change a list it holds, but it cannot reassign a variable from outside, so a list is the safe way to keep state between calls. Chapter 7 explains why.

12.6 Checking a theory

`turtle test` checks the theories of a test file and of the files it imports:

- every `proof` example must give what it says,
- every `theorem` must hold for the proof examples, and for 100 random inputs made to suit the values.

A definition that refuses an input with `fail` does not count that input against a theorem. When a theorem breaks, the test shows the smallest input that breaks it. A theory with no proof, or an `abstract` that does not name its word, gets a warning.

Ordinary runs do not check any of this. The checks run only under `turtle test`, and the next chapters show how to look inside a theory when something goes wrong: `diagnose` and `hypothesis`, in chapter 23.

12.7 A real program: a theory in use

A theory gives a phrase its own meaning. `within` is written like a sentence: `within t, 1, 10` asks whether `t` lies from 1 to 10. Its `proof` checks two cases, and its `theorem` says the result always equals the comparison it makes. The loop uses the phrase just as it would use a built-in word.

```
// A theory phrase that tests whether a value lies between two bounds, used as a
    sentence.
theory within
    abstract
        within x, lo, hi says whether x lies from lo to hi inclusive.
    notation within x, lo, hi .
    definition
        return x >= lo && x <= hi
    theorem result == (x >= lo && x <= hi)
    proof
        within 5, 1, 10 . is true
        within 11, 1, 10 . is false
theory [end]

[loop][t in list [0, 4, 10, 12]]
    ok is within t, 1, 10 .
    show t, " in range: ", ok .
[loop][end]
```

```
0 in range: false
4 in range: true
10 in range: true
12 in range: false
```

The theory is checked when `turtle test` runs. Chapter 23 shows how to look inside it with `diagnose`.

12.8 Several theories together

Theories are most useful when they are combined. A program can use any number of theories, and each phrase reads as part of the same sentence. The theories below are used in one loop, and each result is shown on its own line:

```
theory within
    abstract
        within x, lo, hi says whether x lies from lo to hi inclusive.
    notation within x, lo, hi .
    definition
        return x >= lo && x <= hi
    theorem result == (x >= lo && x <= hi)
    proof
        within 5, 1, 10 . is true
theory [end]

theory squared
    abstract
        squared x says the square of x.
    notation squared x .
    definition
        return x * x
    theorem result >= 0
    proof
        squared 3 . is 9
theory [end]

[loop][n in list [2, 11]]
    show n, " -> ", within squared n, 0, 100 .
[loop][end]
```

```
2 -> true
11 -> false
```

The loop reads as one sentence: the square of `n` lies within 0 and 100. Each phrase is checked by its own proof when `turtle test` runs.

12.9 Functions outside a theory

A definition can call a function that is defined elsewhere in the file. The function is an ordinary function, and it keeps its own name and its own checks:

```
def square[x]
    return x * x
def [end]

theory squared
    abstract
        squared x says the square of x, computed by a function outside this theory.
    notation squared x .
    definition
        return square[x]
    theorem result >= 0
    proof
        squared 3 . is 9
theory [end]

show squared 4 .
```

```
16
```

A theory can therefore be a clear statement of a rule, while the work is done by code that other parts of the program can also use. The function must be defined before the theory that uses it, or the program stops with an error naming the function.

12.10 A theory built from theories

A theory can use another theory in its definition, so that a larger phrase is built from smaller ones. Here `small_square` is written with `within` and `squared`:

```
theory squared
    abstract
        squared x says the square of x.
    notation squared x .
    definition
        return x * x
    theorem result >= 0
    proof
        squared 3 . is 9
theory [end]

theory within
    abstract
        within x, lo, hi says whether x lies from lo to hi inclusive.
    notation within x, lo, hi .
    definition
        return x >= lo && x <= hi
    theorem result == (x >= lo && x <= hi)
    proof
        within 5, 1, 10 . is true
theory [end]

theory small_square
    abstract
        small_square x says whether the square of x lies within 0 and 100.
    notation small_square x .
    definition
        return within squared x, 0, 100
    theorem result == (squared x >= 0 && squared x <= 100)
    proof
        small_square 5 . is false
        small_square 3 . is true
theory [end]

show small_square 3 .
show small_square 5 .
```

```
true
true
```

Each theory is checked on its own. The proof of `small_square` depends on the theories it uses being correct, and `turtle test` checks all of them.

Two rules govern how theories are combined:

- **A theory must come before the theory that uses it.** The file is read from top to bottom, so `small_square` above can use `within` and `squared` only because they are defined first. If the order is reversed, the program stops:

```
theory outer
    abstract
        outer x says x plus one, using another theory.
    notation outer x .
    definition
        return inner x
    theorem result == x + 1
    proof
        outer 2 . is 3
theory [end]
theory inner
    abstract
        inner x says x plus one.
    notation inner x .
    definition
        return x + 1
theory [end]
show outer 4 .
```

```
turtle: program.turtle, line 6: inner is defined by a theory on line 11, below this
    line: a theory comes before its word is used
  6 |         return inner x
    |                ^
```

- **A theory is written at the top level of a file.** A theory cannot be defined inside another theory's definition, or inside a function:

```
theory outer
    abstract
        outer x says x plus one, built from an inner theory.
    notation outer x .
    definition
        theory inner
            abstract
                inner x says x plus one.
            notation inner x .
            definition
                return x + 1
        theory [end]
        return inner x
    theorem result == x + 1
theory [end]
show outer 4 .
```

```
turtle: line 6: theory inner: a theory is written at the top level of a file, not
    inside other code
  6 |         theory inner
```

To build one theory from another, define the smaller theory first, at the top level, and use it in the definition of the larger one.

Best practice. Write the abstract first, in plain words, and then write one theorem and one proof example for each important case, including the empty case. Run `turtle test` after each change.

Common mistake. Using a theory before its definition in the file. Turtle reads a file from top to bottom, and it reports `inner is defined by a theory on line 11, below this line`. Move the theory above its first use.

Key points

- **Theory**: a named piece of knowledge that Turtle can use by its name, and it adds a new phrase to the language.
- **Values and fixed words**: each name in a definition is a value, while words such as `in` and `is` stay fixed.
- **Rules**: a theory has required parts, and Turtle stops with an error when one is missing.
- **Any expression** can be the value passed to a phrase, not only a plain name.
- **Generators**: a phrase can take a function as one of its values.
- `turtle test` checks the theories in a test file and the files it uses.
- **Combining**: theories work together, and one theory can be built from another.

13

When things go wrong

What an error is. An **error** is a problem that stops a program from doing what it was asked. Some errors come from the program's logic, such as dividing by zero. Others come from outside it, such as a file that is missing. Good programs expect these problems and handle them on purpose, rather than crashing without warning.

How Turtle does it. By default an error stops the program and prints where it happened. A `safe` block runs code that might fail, and `handle` says what to do for each kind of error, such as `math` or `file`. A `fail` statement raises your own error with a message you choose.

Errors are part of programming. A file might be missing, a number might be zero, or a user might type the wrong thing. Turtle gives you two tools for these moments: `safe` for handling an error yourself, and `fail` for raising your own.

13.1 An error stops the program

By default, an error stops the program and says where it happened:

```
x = 1 div 0
show "carrying on" .
```

```
turtle: line 1: division by zero
  1 | x = 1 div 0
```

The line that did not run is never shown. Nothing after it runs, and the exit code is `1`, so other programs can tell something went wrong.

13.2 `safe` and `handle`

Put code that might fail in a `safe` block. Under `handle`, list the kind of error you expect, name the error, and write what to do:

```
safe
    [read] scores.txt to lines [end]
handle [file] error .
    show "no scores yet: ", error .
    lines = list []
safe [end]
show length of lines .
```

```
no scores yet: line 2: [read] scores.txt: no such file or folder
0
```

The code under `handle` runs only if the error happened. Without it, the program would have stopped at the `[read]` line.

13.3 Error kinds

Each error has a kind, written in the brackets after `handle`:

- `[file]`: a file or folder is missing, or cannot be read,
- `[math]`: a division by zero, or a similar sum that cannot be done,
- `[key]`: a missing key in a map,
- `[type]`: a value of the wrong kind, such as text where a number belongs.

Leave the brackets empty to handle every kind: `handle [] error ..`

Inside a handler, three parts of the error are available:

```
safe
    x = 10 div 0
handle [math] error .
    show "kind ", kind of error, " line ", line of error .
safe [end]
show "carrying on" .
```

```
kind math line 2
carrying on
```

`kind of error`, `line of error` and `message of error` give the three parts. A handler is the right place to recover, log, or tell the user what happened. Do not use it to hide an error you have not understood yet.

Handling more than one kind. List the kinds in the brackets, separated by commas. The handler runs for any of them:

```
values = list [10, 0]
m = map ["a": 1]
safe
    show 10 / values at get 1 .
    show m at get "zz" .
handle [math, key] problem .
    show "handled: ", problem .
safe [end]
show "carrying on" .
```

```
handled: line 4: division by zero
carrying on
```

The division fails first, so the handler runs for the `math` error and the map lookup on the next line never runs. Each kind must be one that Turtle knows. A name that is not a kind is a parse error, and the program does not run at all:

```
safe
    show 1 / 0 .
handle [math, data, sys] e .
    show e .
safe [end]
```

```
turtle: program.turtle, line 3: "data" isn't a kind of error; handle can list file,
    number, math, index, key, name, type, json, date, http, sql, csv, test, pattern,
    crypt, schedule, config, server, scroll, linear, custom, or [] for any error
  3 | handle [math, data, sys] e .
    |               ^
```

The table lists the kinds. An error of a kind that is not listed is not handled. It goes on to an enclosing `safe`, or stops the program.

Kind	Covers
`file`	a missing file, a file that cannot be written, the end of input
`number`	text that cannot be changed to a number, such as `change "abc" to integer`
`math`	division or modulo by zero, a float power or math function out of range, `sqrt` of a negative
`index`	a list index out of range, `pop` or `min of` an empty collection
`key`	a map key that is not found
`name`	an undefined variable, function, method, module or field
`type`	a value of the wrong kind, such as `"a" * 2`, or the wrong number of arguments
`json`	text that is not valid JSON (`json` library)
`date`	text that is not a date, or a date that does not exist (`time` library)
`http`	a web request that failed or returned a 4xx or 5xx status (`http` library)
`sql`	a bad query or a database problem (`sql` library)
`csv`	a `.csv` or `.tsv` file that is not well formed
`test`	a failed `check`, `verify` or `validate` (`import test`)
`custom`	your own error, raised with `fail` (section 13.4)

The other kinds (`pattern`, `crypt`, `schedule`, `server`, `config`, `scroll`, `linear`) cover the matching library or feature, and their rows are in the reference.

13.4 Raising your own error

`fail` raises an error with a message you choose. Use it when a value is wrong in a way Turtle cannot see:

```
def withdraw[balance, amount]
    if ] amount > balance [
        fail "not enough money"
    if [end]
    return balance - amount
def [end]
```

A `fail` inside a function can be caught by a `safe` block around the call, so the caller decides what to do:

```
def withdraw[balance, amount]
    if ] amount > balance [
        fail "not enough money"
    if [end]
    return balance - amount
def [end]
safe
    x = withdraw[10, 50]
handle [] error .
    show "kind ", kind of error, ": ", message of error .
safe [end]
```

```
kind custom: not enough money
```

Errors you raise with `fail` have the kind `custom`.

13.5 Errors that name a fix

Many of Turtle's messages say what to do. The most common are:

- a text and number joined with `+`: change the text with `change ... to integer`, or write the value inside `"{...}"`,
- a library function used without its import: the message names the `import` line to add,
- a value out of range for a list: the message gives the length of the list.

Read the message before you search for the problem. It often has the answer.

13.6 A real program: dividing safely

Division by zero is an error, and an error normally stops the program. This program puts each division in a `safe` block, and `handle` gives a fallback value. The rest of the list is still processed.

```
// Division by zero is an error. Catch it with safe and handle, and print a fallback
    value instead.
pairs = list [list [10, 2], list [7, 0], list [9, 3]]
[loop][p in pairs]
    result = "n/a"
    safe
        result = p at get[0] / p at get[1]
    handle [] error .
        result = "error"
    safe [end]
    show p, " -> ", result .
[loop][end]
```

```
[ 10, 2 ] -> 5.0
[ 7, 0 ] -> error
[ 9, 3 ] -> 3.0
```

The `result` variable is set before the `safe` block, so the fallback is visible whichever branch runs.

13.7 Warnings

A **warning** is a message that Turtle prints when a program runs, but which does not stop the program. A warning points out something that is probably a mistake. Two warnings are common. The first is a function that is defined twice, so only the newer definition is used after its line runs:

```
def f[x]
    return x
def [end]
def f[x]
    return x + 1
def [end]
show f[1] .
```

```
warning: f is defined more than once: the def on line 4 replaces the one on line 1
    (calls use the newer one once its line has run)
2
```

The second is an assignment inside a function to a name that is also a top-level variable. The assignment makes a new variable local to the function, and the top-level variable does not change. A note says so:

```
count = 0
def bump[n]
    count = n
def [end]
bump[2]
show count .
```

```
turtle: note: program.turtle, line 3: count is also a top-level variable, so this
    makes a new local count in bump and the top-level count is not changed. To change
    it, return the new value and assign that, or keep it in a list or map
0
```

Best practice. Catch only the kinds of error you expect, and name them. A catch-all `handle` can hide a bug that you would rather see.

Common mistake. Writing a kind name that Turtle does not know. Turtle stops with `"data" isn't a kind of error` and lists the kinds it does know.

Key points

- **Errors**: an error stops the program by default, and the message says where it happened.
- `safe` **and** `handle`: put code that may fail in a `safe` block, and list the error kinds to catch under `handle`.
- **Error kinds** are written in brackets after `handle`.
- `fail` raises an error with a message of your choice.
- **Named fixes**: many of Turtle's messages say what to do next.
- **Warning**: a message that Turtle prints while the program keeps running.

14

Modules

What a module is. A program is easier to maintain when its code is split into files, each with one job. A file that other files use is called a **module**. The file that starts the program uses the others through **import**.

Why it matters. Separate files let several people work at once, let you reuse a helper in another program, and keep each file short enough to read.

How Turtle does it. Any `.turtle` file is a module, and its name is its file name without `.turtle`. `import shop` brings in `shop.turtle`, and `import data` brings in a **library** that comes with Turtle. A module's functions become available to the importing file, while its variables stay private to it.

A small program fits in one file. A bigger one does not. Once a file holds the rules for orders, the code that reads a CSV, the report and the helper functions, it is hard to find anything, and every change risks breaking something unrelated. A **module** is a file of its own. You bring one in with `import`, and the program uses what the module offers.

Why split a program. Each file does one job, so you can find the code for a job by its file name. A module you wrote once can be used by several programs, without copying its code into each one. Each module can also be tested and changed on its own. This is what keeps a program clean as it grows: the main file says what happens, and the modules say how.

When not to split. A short script, a program you will throw away, or a program of a few dozen lines is easier to read as one file. Each import adds a file to keep track of and a set of names to know. Split when a part of the program has a clear job that another program could also use, not before.

What a module can contain. A module is an ordinary `.turtle` file, and it can have four things:

- **Functions** that any file which imports the module can call. These are the module's public interface, such as `withtax` in `shop.turtle`.
- **Variables** that stay private to the module. The importing file cannot read them. A module's `taxrate` is used by its own functions, and the program that imports it cannot see `taxrate` (section 14.2).
- **Private functions**, whose names start with `~`. Only the module's own functions can call them, and another file that tries gets an error (section 14.9).
- **Setup code** is every line in the module that is not inside a `def` block. The first time a program imports the module, Turtle runs the file from top to bottom. So the setup lines run at that

moment, whether or not the program ever calls a function from the module. Lines inside a `def ... def [end]` block do nothing until that function is called.

Setup code is usually one of two things: a definition, such as the `taxrate = 0.08` that `withtax` reads, or a setting, such as a `settings` map. Keep anything else out of it. A `show` line in a module prints every time the program loads the module, which the reader of the program may not expect.

Here is a module with one setup line that prints, and one that defines a value:

```
// lib/setup.turtle
show "module setup running" .
x = 1
```

A program that shows a line before and after the import gets this output:

```
show "before import" .
import lib/setup
show "after import" .
```

```
before import
module setup running
after import
```

The module's line appears at the import line, in order with the program's own lines. Each module runs only once, even if two files import it, so the line appears one time for the whole program.

How to name a module. The module's name is its file name without `.turtle`. Pick a short noun for the job: `shop`, `money`, `report`. A file in a folder is named by the file alone, so `lib/money.turtle` is the module `money`. Two modules can have a function with the same name. When that happens, put the module name in front of the call (section 14.3).

Where Turtle looks for a file. An import is found from the folder of the file that contains the `import` line, not from the folder you run the program in. So a program in `reports/` can say `import lib/money` and find `reports/lib/money.turtle`, and it still works when run from any other folder. If the file is missing, the error names the file it looked for.

How to import. The forms are shown in section 14.1. Import a whole module with `import shop`. Import only some names with `import time [now]` (section 14.4). Set defaults with `import data [tablerows = 50]` (section 14.6). Put the import lines at the top of the file, as section 14.8 shows.

14.1 Libraries and your own files

`import` brings in either a library that comes with Turtle, or a `.turtle` file you wrote:

```
import data                    // a library that comes with Turtle
import time [now, sleep]       // only some of its names
import shop                    // shop.turtle, next to this program
import lib/money               // lib/money.turtle; its name is "money"
```

A library's names become available after its import. Using one without the import is an error that names the line to add:

```
x = sum[list [1, 2]]
```

```
turtle: line 1: "sum" needs "import data" first
  1 | x = sum[list [1, 2]]
```

14.2 Writing a module

A module is just a file. Its top-level functions become available to any file that imports it. Its variables stay private to it:

```
// shop.turtle
taxrate = 0.08

def withtax[amount]
    return amount + amount * taxrate
def [end]
```

```
import shop
show withtax[100] .
```

```
108.0
```

`taxrate` is not visible to the importing file. The function can still read it, because it was defined in `shop.turtle`.

14.3 Naming the module in front

If two imports have a function with the same name, name the module when you call it:

```
import shop
show shop withtax[50] .
```

```
54.0
```

The module name in front is a sentence-style call: `shop withtax[50]` means "the `withtax` from `shop`". Calling the plain name when two imports clash is an error that tells you to do this. A function you define yourself always wins over an imported one with the same name.

14.4 Importing some names only

`import time [now]` brings in only `now`. Use this when you want a short list of names and no surprises:

```
import time [now]
show "imported names only" .
show now[] > 0 .
```

```
imported names only
true
```

Any other name from `time` is not imported. Calling `sleep` in the same file stops the program with an error that tells you to add it to the list:

```
import time [now]
show sleep[0] .
```

```
turtle: line 2: "sleep" isn't imported — add it to "import time [...]"
  2 | show sleep[0] .
```

To use `sleep`, change the line to `import time [now, sleep]`.

14.5 Two modules with the same function name

Two modules can each define a function with the same name. A program that imports both must say which one it means, and Turtle checks this for you. Take a `shop.turtle` that adds an 8% tax, and a `store.turtle` that adds a 20% tax:

```
// shop.turtle
taxrate = 0.08

def withtax[amount]
    return amount + amount * taxrate
def [end]
```

```
// store.turtle
taxrate = 0.20

def withtax[amount]
    return amount + amount * taxrate
def [end]
```

A program that imports both can name the module in front of each call. This keeps the two functions apart, and it also makes the program easier to read, because a reader can see which tax is used:

```
import shop
import store
show shop withtax[10] .
show store withtax[10] .
```

```
10.8
12.0
```

A module's function is an ordinary value, so it can be stored under a name, and the name then holds that function. The stored name keeps the module's rule:

```
import shop
f = shop withtax
show f[10] .
```

```
10.8
```

If the name is used without a module, Turtle cannot tell which function is meant, so it stops and asks for the module:

```
import shop
import store
show withtax[10] .
```

```
turtle: line 3: "withtax" is provided by more than one import (shop, store) — say
    which one, e.g. shop withtax[...]
  3 | show withtax[10] .
```

Section 14.3 explains the module-name form in more detail. Use it whenever two imports share a name, even if one of them is only used once, so that a later import does not change the meaning of a line that was written earlier.

14.6 Setting defaults for an import

The standard libraries create settings in the importing file, such as `tablerows`, `loglevel` and `seed`. You can set them in the import line, by writing `name = value` inside the brackets:

```
import data [tablerows = 50]
import log [loglevel = "debug"]
import random [seed = 7]
import data [tablerows = 50, process]
```

The setting replaces the library's default. A plain name in the same brackets still chooses what is imported, so `import data [tablerows = 50, process]` imports only `process`. With settings and no names, everything is imported. A setting can also be changed on a later line, as before.

Your own modules work differently. A module's variables are private to it, so an import line cannot set them:

```
import shop [taxrate]
```

```
turtle: line 1: import shop: module "shop" has no "taxrate"
  1 | import shop [taxrate]
```

`import` brings in functions and theories, and a module's own settings are changed through a function. Turtle has four ways to do this. Each one below was run.

1. A `none` argument means "use the default." Inside the function, a `none` argument is replaced with the module's own value. This is the most direct form, and the one to use for a single value:

```
// shop.turtle
taxrate = 0.08

def withtax[amount, rate]
    if ] rate == none [
        rate = taxrate
    if [end]
    return amount + amount * rate
def [end]
```

```
import shop
show withtax[100, none] .
show withtax[100, 0.1] .
```

```
108.0
110.0
```

Turtle has no default values in a `def` line, such as `def greet[name = "world"]`. Writing `none` in the call and testing for it inside the function does the same job. Section 7.13 gives the rule for every function, not only imported ones.

2. A settings map, read with a fallback. When a function takes several settings, pass them as a map. Check for each key with `contains` and use the module's value when the key is missing:

```
// shop.turtle
taxrate = 0.08

def withtax[amount, settings]
    rate = taxrate
    if ] settings contains "rate" [
        rate = settings at get "rate"
    if [end]
    return amount + amount * rate
def [end]
```

```
import shop
show withtax[100, map []] .
show withtax[100, map ["rate": 0.2]] .
```

```
108.0
120.0
```

`get` on a missing key is an error, so check with `contains` first.

3. Command-line options. `options` reads the options given after the program's name, and falls back to the default you write. It returns a map, so take the option out with `get`:

```
import system
import shop

opts = options["--rate": 0.08]
show withtax[100, map ["rate": opts at get "--rate"]] .
```

```
108.0
```

Run it with an option, and the default is replaced:

```
turtle rate.turtle --rate 0.15
115.0
```

Command-line options are read by the `options` function. Section 19.3 has the full rules, with an example for each.

4. Settings the module keeps itself. A module can own a map, and offer a `configure` function that changes it. The program calls `configure` once, near the top. Changing the module's own map inside a function is allowed, because a map is shared, not copied:

```
// shop.turtle
settings = map ["rate": 0.08]

// configure copies each new setting into the module's map.
def configure[changes]
    [loop][key in changes getkeys]
        value = changes at get key
        put value to settings at key .
    [loop][end]
def [end]

def withtax[amount]
    return amount + amount * settings at get "rate"
def [end]
```

```
import shop
show withtax[100] .
configure[map ["rate": 0.25]]
show withtax[100] .
```

```
108.0
125.0
```

Choosing between them: use the `none` argument for one value, a settings map for a group of values, `options` for values a user types on the command line, and `configure` for settings that should hold for the whole program.

14.7 Some imports add words to the language

A few imports also turn on new words, in that file only:

- `import random` turns on `random` (`random integer from 1 to 6`),
- `import test` turns on `check`, `verify` and `validate` (chapter 22),
- `import log` turns on `log` (chapter 21),
- `import linear` turns on `matrix` (chapter 18).

Anywhere else, these words are ordinary names, so a variable can still be called `check` in a file without that import.

14.8 Program layout that reads well

Put the imports first, then the settings, then the main code, then the functions. This reads top to bottom like a letter:

```
import shop
import data

rate = 0.08
orders = list [100, 250, 40]

total = 0
[loop][o in orders]
    total = total + withtax[o]
[loop][end]
show "total: {total}" .

def helper[x]
    return x
def [end]
```

```
total: 421.2
```

Functions can be placed at the bottom because they are defined before the first line runs.

14.9 Private functions in a module

A module can keep some of its functions to itself. A function whose name starts with `~` is private to its file (section 7.12). The module below has one public function, `clamp`, which uses the private `~limit` to do its work:

```
// shop.turtle
def clamp[n]
    return ~limit[n, 0, 100]
def [end]

// Private: only this file can call it.
def ~limit[n, low, high]
    if ] n < low [
        return low
    if [end]
    if ] n > high [
        return high
    if [end]
    return n
def [end]
```

A program that imports `shop` can call `clamp`, because `clamp` is public. Calling `clamp` does not need `~limit`, and the program never sees it:

```
import shop
show clamp[150] .
show clamp[-5] .
```

```
100
0
```

Calling the private function from the program is an error. The same is true if the program names it in an import list:

```
import shop
show shop ~limit[1, 0, 10] .
```

```
turtle: line 2: ~limit is private to shop: a ~ function is for its own file
  2 | show shop ~limit[1, 0, 10] .
```

```
import shop [~limit]
show ~limit[1, 0, 10] .
```

```
turtle: line 1: import shop: ~limit is private to shop (a ~ function is for its own
    file)
  1 | import shop [~limit]
```

Only functions can be private in this way. A module's variables are private to it already (section 14.2), so the `~` is used only for functions.

14.10 Different settings for different calls

A library's settings are variables in the file that imports it, so every function in that file shares them. A setting changed inside a function lasts only for that call, and the file's value is back after the call. One shared setting can therefore have a different value for each call. This example shows the `tablerows` setting of `data` changed by a function that shows the top of a table:

```
import data [tablerows = 50]

rows = list [1, 2, 3, 4, 5, 6, 7, 8]

def show_top[items, limit]
    tablerows = limit
    show table[items] .
def [end]

show "file setting: {tablerows}" .
show_top[rows, 2] .
show_top[rows, 4] .
show "after the calls: {tablerows}" .
```

```
file setting: 50
#  value
-  -----
0      1
1      2
... 9 more lines
```

The same function shows two rows and then four. Each call has its own `tablerows`, and the file's `tablerows` is still 50 after both calls. The output is cut here, and the full output is 14 lines.

Best practice. Name each module after its job, such as `orders` or `report`, and keep the import lines together at the top of the file.

Common mistake. Importing a file that is not where Turtle looks. Turtle looks in the folder of the file that contains the `import`, and it reports the file it expected: `lib/setup.turtle: no such file or folder`.

Key points

- **Module**: a `.turtle` file brought in with `import`. Libraries that come with Turtle are imported the same way.
- **Public functions** are available to any file that imports the module, while **variables** stay private to it.
- **Setup code**, the lines outside any function, runs once when the module is first imported.
- **Name in front**: when two imports share a function name, put the module name before the call, as in `shop withtax[50]`.
- **Some names only**: `import time [now]` brings in just the listed names.
- **Functions as values**: a module's function can be stored in a variable and called later.
- **Defaults**: a library's settings can be changed inside the import brackets, as in `import data [tablerows = 50]`.
- **Words**: some imports turn on new words, but only in the file that imports them.
- **Layout**: put the imports first, then the settings, the main code and the functions.
- **Private functions**: a name starting with `~` can be used only inside its own module.

15

Sentence form

What sentence form is. Some calls read better as words than as brackets. Turtle lets a call be written with its values placed between words, so the code reads like a plain sentence.

Why it matters. A reader can check the meaning of a line without remembering the order of arguments, because the words say which value is which.

How Turtle does it. A theory or a library function can have a sentence form. Each word of the sentence stays fixed, and each value takes the place it names. The bracket form of the same call always works too.

Turtle lets a program read like a sentence. A statement names what is done first, or the value it is done to first, and the rest follows in the order a person would say it. This chapter explains the sentence forms, shows how they combine with the forms from chapter 7, and sets out a way of writing sentence code that stays clear as a program grows. It is written for a reader who already knows functions, lists and maps.

A **sentence** here is one statement, ended by a period. A **subject** is the value a sentence acts on, and a **verb** is the function or keyword that acts on it.

15.1 One line, one idea

Each sentence should say one thing. The period at the end marks where the idea ends, and the reader can take in one line at a time:

```
nums = list [4, 9, 2]
add 7 to nums .
show nums .
```

```
[ 4, 9, 2, 7 ]
```

The first line changes the list, the second shows it. Keeping each change on its own line makes a program easy to follow, and easy to check when something goes wrong.

15.2 Value first: the subject, then the verb

When a function acts on a value, the value can come first, as the subject, and the function's name follows as the verb:

```
import data
import strings
nums = list [4, 9, 2]
show nums sum .
show nums at length .
show "turtle reads code" upper .
```

```
15
3
TURTLE READS CODE
```

This form reads well for an action on a value: `nums sum`, `name upper`, `words join ", "`. The library functions that take a value this way are listed in chapter 16. Value first is the form to use when the value is the thing being described.

15.3 Statements that read as sentences

Some keywords are sentences in their own right. They take a value, a preposition and a collection, and they change the collection:

```
s = set [1, 2]
add 3 to s .
remove 1 from s .
show s .
show s at contains 3 .
```

```
{ 2, 3 }
true
```

`add value to collection .` and `remove value from collection .` are the statement forms. They work on lists and sets, and `at contains` asks a question about a collection.

15.4 Name first in a sentence

The name-first form from section 7.1 is also a sentence: the verb comes first, and its arguments follow it. It suits a short action on plain values:

```
def fix[a, b]
    return a + b
def [end]
show fix 1, 2 .
show fix 1, 2; " is the sum" .
```

```
3
3 is the sum
```

Use the semicolon, as in section 7.1a, when the arguments are followed by more text in the same `show`. Without it, a comma would continue the call.

15.5 Functions as sentences: keep, process and give

A function value written with `give` reads as a sentence too. `keep` and `process` each take a collection and a function, so the sentence says what is done to each item:

```
import data
prices = list [100, 250, 40]
big = prices keep p give p > 50
show big .
sale = prices process p give p - p * 10 div 100
show sale .
show big sum .
```

```
[ 100, 250 ]
[ 90, 225, 36 ]
350
```

A `give` function takes the rest of its line. In the line above, `keep` is given the whole condition `p > 50`. So a sentence call cannot be placed after the function, on the same line. Store the result first, as the next section shows.

15.6 Phrases from theories

A theory (chapter 12) teaches Turtle a new phrase. The phrase is written in a sentence, and it reads like part of the language:

```
theory double
    abstract
        double says twice n.
    notation double n .
    definition
        return n * 2
theory [end]
show double 4 .
x = 5
show double x .
```

```
8
10
```

The notation says where the values go. Here the value follows the word, as it does in `double 4`.

15.7 Writing clean sentence code

Sentence form is easiest to read when a few habits are followed. None of them is a rule the language enforces. They are the habits that keep a program clear as it grows.

- **One idea per line.** A line that changes a value should not also show it. Split the work into two lines.

- **Name steps.** Store each step in a variable with a noun for a name, and let the next sentence use it. This is the most useful habit in this chapter:

```
import data
prices = list [100, 250, 40]
big = prices keep p give p > 50
total = big sum
show "{total} from {big at length} items" .
```

```
350 from 2 items
```

- **Put the subject first.** If a function acts on a value, write the value first: `nums sum`, not `sum nums`. If the arguments are the subject, write the name first: `fix 1, 2`.
- **Use the bracket form inside a larger expression.** When a call sits inside a calculation, or its arguments are long, brackets make the boundaries clear: `fix[n, 2] * 10`.
- **Keep a `give` function on its own line.** It takes the rest of the line, so the line should end with it.
- **Test conditions as sentences.** A condition can be a sentence, which reads naturally in an `if`:

```
words = list ["ann", "ben"]
if ] words at contains "ann" [
    show "ann is here" .
if [end]
```

```
ann is here
```

- **Use the semicolon in `show` to separate items.** It keeps the call's arguments apart from the text that follows.

15.8 A program in sentence form

This program works out the sale price of a list of items. It is written with one idea per line, named steps, and a condition that reads as a sentence:

```
import data
def discount[price, pct]
    return price - price * pct div 100
def [end]

prices = list [100, 250, 40]
big = prices keep p give p > 50
sale = big process p give discount p, 10
total = sale sum
show "sale total: {total}" .
if ] total > 300 [
    show "above the threshold" .
if [end]
```

```
sale total: 315
above the threshold
```

The program has three named steps, `big`, `sale` and `total`. A reader can follow it line by line, and each step can be checked on its own.

15.9 What sentence form does not do

Sentence form has a few limits. Knowing them avoids confusing errors:

- **A function stored in a variable has limits on the name-first form.** For `f = x give x * 3` and `a = 4`, the forms `f[4]`, `f[a]`, `4 f`, `a f` and the name-first `f 4` all give 12. The name-first form stops with an error when the argument is a variable or a negative number: `f a` and `f -1` both fail. For a negative value, write it first, `-1 f`, which gives `-3`, or use brackets, `f[-1]`. A function made with `def` accepts every name-first form, such as `f a` and `f -1`, so the limit applies to function values.
- **Give functions end at the line end.** A `give` function takes the rest of its line, so a sentence call that follows it is taken into the function's body. Put it on its own line.
- **Parentheses do not start a sentence.** `(nums keep x give x > 3) sum` is not valid. Store the result first.
- **Some words are keywords in one place only.** `each` marks the rule in `verify` and `hypothesis`. Elsewhere it is an ordinary name, so `each = 3` works, but `each nums` is an error, because `each` is not a function.
- **A name cannot be both a variable and a function.** When a name is assigned, it is a variable, and a call on it is read as a sentence on that variable.

Best practice. Use the sentence form when it makes a line clearer, and the bracket form when the values are complicated. Keep to one style within a file.

Common mistake. Writing a sentence that Turtle does not recognize, such as a word out of order. The error names the word it expected at that place.

Key points

- **Sentence form**: a statement can read like a sentence, naming what is done before what it is done to.
- **One line, one idea**: each sentence should say one thing, and the period marks where it ends.
- **Value first**: when a function acts on a value, the value can come first, as the subject of the sentence.
- **Keywords** can be sentences in their own right, taking a value, a preposition and another value.
- **Name first** is also a sentence form, where the verb comes first.
- `keep`**,** `process` **and** `give` let a function value read as a sentence.
- **Theory phrases** can be written as sentences, once a theory has taught Turtle the phrase.
- **Limits**: sentence form has a few limits, and knowing them avoids confusing errors.

Part II: The standard library

16

The library at a glance

What a library is. A library is a set of ready-made functions that someone else has already written and tested. Using one saves you from writing and checking the same code again.

How Turtle does it. Turtle's core language is small. Everything beyond it comes from libraries, and a library is turned on with `import` at the top of a file. Once imported, its functions can be called by name.

Turtle's core is small on purpose. The language itself gives you values, decisions, loops, functions and theories. Everything else comes from libraries, and a library is turned on with `import` at the top of a file.

This chapter lists the libraries and their main functions. Every library and every function in it, with its full details and examples, is covered on the Turtle website, at turtlelang.dev. The chapters that follow explain the most common ones.

Some things need no import at all: lists, sets, maps, text, numbers, and the file statements `[read]`, `[write]`, `[append]` and `[directory]`.

This chapter lists every library, grouped by what it is for. Use it to find the library for a job. Chapters 17 to 20 show the jobs in practice.

From the terminal, `turtle doc` lists every library function with its arguments. `turtle doc sql` shows one whole library, and `turtle doc sql_update` shows one function in full.

16.1 Built in (no import)

- **Lists, sets, maps, text and numbers**, with their methods (`length`, `get`, `put`, `upper`, `round` and so on). These are the everyday tools.
- **Files and folders** with `[read]`, `[write]`, `[append]` and `[directory]`.
- **Shell commands** with `sys`.

16.2 Data: collections and tables

- `data`: process, keep and reduce collections; group, unique and range; table reading and writing (`table_read`, `table_write`, `table`); statistics (`mean`, `median`, `stdev`, `percentile`, `describe`, `correlation`). The library to reach for first when you have rows.
- `sort`: put things in order by a function, a field or a position, with the classic algorithms (`quick_sort`, `merge_sort`, `heap_sort`) when you want to choose one.

- `search`: find items by a rule (`find_first`, `find_all`, `count_where`), search sorted lists (`binary_search`, `interpolation_search`), and find where a value belongs (`insert_position`).
- `linear`: matrices and vectors, with solving, inverses, determinants, and decompositions (`solve`, `inverse`, `eigen`, `svd`).

16.3 Numbers and chance

- `math`: square roots, logarithms, trigonometry, rounding and powers, as methods on numbers (`16 at sqrt`, `3.14159 at round 2`).
- `random`: random values of any shape (`random integer from 1 to 6`), and picking, shuffling and sampling from a list (`pick`, `shuffle`, `sample`, `chance`).

16.4 Text

- `strings`: finding, cutting and joining text as sentences (`find`, `substring`, `join`, `isinstring`).
- `pattern`: regular expressions for matching, finding, replacing and splitting (`matches`, `findall`, `replaceall`, `splitby`, `groups`).

16.5 Files, system and settings

- `system`: the command line (`args`), environment variables (`env`), files and folders (`exists`, `contents`, `copyto`, `moveto`, `erase`, `makefolder`), archives (`pack`, `unpack`), and the program itself (`scriptfolder`, `exit`).
- `config`: settings files in TOML, JSON or `.env` format (`config_read`, `config_write`).

16.6 The internet

- `http`: web requests, with `http_get`, `http_post` and the general `http_request`.
- `json`: reading and writing JSON (`load`, `json_text`, `json_read`, `json_write`, `json_get`).
- `server`: a web server built from a map of routes (`serve`, `reply`, `redirect`).

16.7 Databases

- `sql`: SQLite files, and PostgreSQL and MySQL servers, through one set of functions (`sql_open`, `sql_query`, `sql_run`, `sql_save`, `sql_load`, `sql_upsert`, `sql_update`, `sql_delete`).

16.8 Time

- `time`: the clock, dates and time zones, date arithmetic, formatting and waiting (`now`, `today`, `add_time`, `time_between`, `format_date`, `sleep`, `every`).
- `schedule`: many web requests, shell commands or database queries at once (`fetchall`, `runall`, `queryall`).

16.9 Security

- `crypt`: hashes, encodings, ids and tokens, passwords, and encryption (`hash`, `encode`, `uuid`, `token`, `passwordhash`, `passwordcheck`, `encrypt`, `decrypt`).

16.10 Development

- `test`: `check`, `verify` and `validate`, run by `turtle test` (chapter 22).
- `log`: log lines with levels, to the screen or a file (chapter 21).

16.11 The import line, in one place

```
import data
import sort
import search
import linear
import math
import random
import strings
import pattern
import system
import config
import http
import json
import server
import sql
import time
import schedule
import crypt
import test
import log
```

You only import what a file uses. Importing a library you do not need costs nothing in the program's behavior, but it makes the file's dependencies harder to see.

16.12 Both forms of each call

Most library functions can be written two ways. The bracket form puts every argument in `[ ]`. The sentence form puts the value first and the function's name after it, so `nums sum` reads as "nums, sum". Use whichever reads more clearly in place. Each pair below gives the same result, and each was checked by running it:

Job	Bracket form	Sentence form	Result
total of a list	`sum[nums]`	`nums sum`	`6`
mean of a column	`mean[rows, "x"]`	`rows mean "x"`	`3.0`
change each item	`process[nums, x give x * 10]`	`nums process x give x * 10`	`[ 10, 20 ]`
keep some items	`keep[nums, x give x > 1]`	`nums keep x give x > 1`	`[ 2, 3 ]`
combine items	`reduce[nums, 0, [t, x] give t + x]`	`nums reduce 0, [t, x] give t + x`	`6`
drop repeats	`unique[nums]`	`nums unique`	`[ 1, 2 ]`
find in text	`find[s, "42"]`	`s find "42"`	`6`
cut text	`substring[s, 0, 2]`	`s substring 0, 2`	`or`
join items	`join[xs, "-"]`	`xs join "-"`	`a-b`
is in text	`isinstring["ord", s]`	`"ord" isinstring s`	`true`
split text	`split[s, ","]`	`s split ","`	`[ "a", "b" ]`
one item	`get[nums, 1]`	`nums get 1`	`6`
size	`length of nums`	`nums at length`	`2`
read JSON	`json_get[u, "name"]`	`u json_get "name"`	`Ann`
solve	`solve[a, list [3, 5]]`	`a solve list [3, 5]`	`[ 0.8, 1.4 ]`
pick one	`pick[list [7]]`	`list [7] pick`	`7`
days between	`time_between[d, e, "days"]`	`d time_between e, "days"`	`3`
add days	`add_time[d, 3, "days"]`	`d add_time 3, "days"`	a date
round	`round[3.14159, 2]`	`3.14159 round 2`	`3.14`
absolute value	`abs[-4]`	`-4 abs`	`4`
square root	`sqrt[16]`	`16 at sqrt`	`4.0`
upper case	`upper[s]`	`s upper`	`ABC`
has a key	`contains[m, "a"]`	`m contains "a"`	`true`
hash	`hash["x", "sha256"]`	`"x" hash "sha256"`	the digest

Two notes. `length` is written `length of nums` in bracket form, and `nums at length` in sentence form. The `math` functions need `import math`, and the text functions need `import strings`, as they did before.

Best practice. Import only the libraries a file uses, and check the chapter that covers each one before you rely on it.

Common mistake. Using a library function without its import. Turtle stops and names the line to add, such as `"sum" needs "import data" first`.

Key points

- **Built in**: lists, sets, maps, text and numbers need no import, and have their own methods.
- **Data**: the `data` library processes, keeps and groups collections, and works with tables.
- **Numbers**: `math` covers roots, logarithms, trigonometry, rounding and powers.
- **Text**: the `strings` library finds, cuts and joins text, and `pattern` matches patterns.
- **Files and the internet**: `system` gives the command line, environment and file tools, and `http` makes web requests.
- **Databases and time**: `sql` works with SQLite, PostgreSQL and MySQL, and `time` handles clocks, dates and time zones.
- **Security and development**: `crypt` covers hashes and passwords, and `test` runs checks through `turtle test`.
- **Import lines**: every import is listed in one place, and most library functions can be written in two forms.

17

Collections and data

What data work is. Much of programming is working through a collection: changing each item, keeping the ones that match a rule, or adding them up. A table of rows, such as sales records, is a list of maps, one map per row, with the same keys in each.

How Turtle does it. The `data` library provides functions such as `sum` and `mean`, the `sort` library sorts, and `table_read` reads a CSV file, which is a plain text table with commas between the columns, into rows. Sorting needs values of one kind: numbers with numbers, and text with text.

Most real programs work on a collection: a list of orders, a column of sales figures, a table of rows read from a file. A collection holds many values, and the work is usually the same for each one: change it, keep some of them, or add them up. The `data`, `sort` and `search` libraries do that work for you, so a loop is only needed when a rule is special. This chapter covers `data` for summaries and tables, and the programs show how each library call reads.

17.1 Working with a list of records

A record here is a map, or an assembled value, held in a list. The three jobs are to change each item (`data`), order the list (`sort`), and find items in it (`search`).

The most efficient way is a function called once per item:

```
import data

orders = list [100, 250, 40]
with_tax = orders process x give x * 1.08
show with_tax .
```

```
[ 108.0, 270.0, 43.2 ]
```

The sentence form reads left to right: take `orders`, process each item with the rule, and keep the result. The bracket form, `process[orders, x give x * 1.08]`, does the same and is useful inside a larger expression.

```
[ 108.0, 270.0, 43.2 ]
```

`process` changes each item, `keep` keeps the items that pass a test, and `reduce` combines them into one value:

```
import data

nums = list [1, 2, 3, 4]
show nums keep x give x > 2 .
show nums reduce 0, [total, x] give total + x .
show nums sum .
```

```
[ 3, 4 ]
10
10
```

A `reduce` function takes two names: the total so far, and the next item. `sum` is the shortest form when all you want is the total. Sentence-style calls end their arguments at the end of the line, so a `give` function can be the last thing in the line, with its period after it.

Sorting records by one of their fields uses `sort`. The key is the field name, and the third argument says how many to give back, or `"first"` for just the first:

```
import sort

rows = list [map ["n": "b", "v": 2], map ["n": "a", "v": 1]]
show min_sort[rows, "v", "first"] .
```

```
{ "n": "a", "v": 1 }
```

Other ways. Plain numbers and text sort with a key function:

```
import sort

nums = list [5, 2, 9, 1]
show quick_sort[nums, x give x] .
```

```
[ 1, 2, 5, 9 ]
```

`sort` also has the classic algorithms (`merge_sort`, `heap_sort`, `insertion_sort`) when you want one of them for teaching or a specific behavior. For everyday use, `quick_sort` and `merge_sort` are the choices.

Finding records uses `search`. A rule finds the first match, or counts the matches:

```
import search

nums = list [4, 8, 15, 16, 23, 42]
show find_first[nums, x give x > 10] .
show count_where[nums, x give x % 2 == 0] .
```

```
15
4
```

For a sorted list, `binary_search` is faster than looking through every item. It gives the position of the value, or where it would go:

```
import search

nums = list [4, 8, 15, 16, 23, 42]
show binary_search[nums, 16, x give x] .
```

```
3
```

Use `find_first` when the rule is what matters, and `binary_search` when the list is sorted and you are looking for one value.

Grouping records by a key is `group`, and `unique` drops repeats:

```
import data

words = list ["ant", "bee", "ax"]
show group[words, w give w at slice[0, 1]] .
show unique[list [1, 1, 2]] .
```

```
{ "a": [ "ant", "ax" ], "b": [ "bee" ] }
[ 1, 2 ]
```

17.2 A real program: grouping records

`unique` drops repeated values. `group` sorts each item into a map under the key that a function gives it. Here the function is `parity`, which returns `n % 2`, so the keys are `1` for odd numbers and `0` for even ones. The map lists the keys in the order the items first appear.

```
// Unique values and grouping of records by a function.
import data
nums = list [3, 1, 3, 2, 1, 5]
show "unique: ", unique[nums] .
def parity[n]
    return n % 2
def [end]

groups = group[nums, parity]
show groups .
```

```
unique: [ 3, 1, 2, 5 ]
{ 1: [ 3, 1, 3, 1, 5 ], 0: [ 2 ] }
```

A group is a map of lists, so a loop over its keys, with `at get`, processes each group in turn.

17.3 A real program: a sales summary

This program reads a month-by-month list of sales and reports the total, the average, and the highest month. It is the pattern most reports follow: collect the numbers, then summarize them with the `data` library.

```
// Summary of monthly sales figures with the data library: total, average and highest
    month.
import data
sales = list [map ["month": "Jan", "amount": 1200], map ["month": "Feb", "amount":
    1850], map ["month": "Mar", "amount": 990], map ["month": "Apr", "amount": 2100]]
amounts = list []
[loop][s in sales]
    add s at get["amount"] to amounts .
[loop][end]
show "total: ", sum[amounts] .
show "average: ", mean[amounts] .
show "maximum: ", max of amounts .
```

```
total: 6140
average: 1535.0
maximum: 2100
```

Read the loop: `add ... to amounts .` builds a new list one item at a time, and `s at get["amount"]` reads one field from each record. Then `sum`, `mean` and `max` do the summary. The sentence forms work too: `amounts sum`, and `amounts mean` on a plain list of numbers. `mean` always gives a float, which is why the average prints as `1535.0` even though the division is exact.

17.4 Tables and CSV files, with statistics

Libraries: `data`.

What a table is. A table is rows of data, where every row has the same named columns. A list of maps is already a table: each map is one row, and its keys are the column names. The `table` function prints such a list as aligned columns, with the keys as the headings. Its optional second argument limits how many rows are shown, and the rest are counted in a line at the end:

```
import data
rows = list [map ["title": "Emma", "copies": 12], map ["title": "Ida", "copies": 4]]
show table[rows] .
show table[rows, 1] .
```

```
title  copies
-----  ------
Emma       12
Ida         4
title  copies
-----  ------
Emma       12
... 1 more row
```

`table` only prints the rows; it does not change them. The same rows can be shown in full or cut short, and the list stays the same.

The most efficient way to read a CSV file is `table_read`. It gives a list of maps, one per row, and turns plain numbers into numbers:

```
import data

sales = table_read["sales.csv"]
show table[sales, 3] .
```

```
title  copies  price
-----  ------  -----
Emma       12    9.5
Ida         4      7
Leo        30     12
```

`table` lines the rows up in columns. Its second argument limits how many rows are shown.

Statistics come straight from the rows, by naming the column:

```
import data
sales = table_read["sales.csv"]
show sales mean "copies" .
show sales median "copies" .
show describe[sales, "copies"] .
```

```
15.333333333333334
12
{ "count": 3, "mean": 15.333333333333334, "stdev": 13.316656236958787, "min": 4,
    "25%": 8.0, "median": 12, "75%": 21.0, "max": 30 }
```

Filter rows with `keep` and a sentence, and write the result back out:

```
big = sales keep s give copies of s > 10
table_write["big.csv", big]
```

Other ways. A column of codes such as `007` stays text, so its zeros are kept. To force a column type, give `table_read` a map of types:

```
import data
codes = table_read["codes.csv", map ["code": "text"]]
```

For a sample of values, `variance`, `stdev`, `percentile`, `correlation` and `zscores` answer the usual questions. Each takes the rows and the column name.

17.5 Functions at a glance: the data, sort and search libraries

Statistics on a list of numbers use `import data`. `pvariance` and `pstdev` divide by the count, and treat the list as the whole population. `covariance` measures how two lists move together:

```
import data
heights = list [150, 160, 170, 180]
weights = list [50, 58, 70, 81]
show covariance[heights, weights] .
show mode[list ["tea", "coffee", "tea"]] .
show pstdev[list [2, 4, 4, 4, 5, 5, 7, 9]] .
show pvariance[list [2, 4, 4, 4, 5, 5, 7, 9]] .
```

```
175.0
tea
2.0
4.0
```

The sort library puts items in order, and `reverse_list` gives a reversed copy. Each sort takes a rule that gives the value to compare, and returns a new list. `is_sorted` tests whether a list is already in order. `max_sort` gives the largest items, and the number after the rule is how many:

```
import sort
show is_sorted[list [1, 2, 2, 5], x give x] .
show reverse_list[list [1, 2, 3]] .
show max_sort[list [4, 9, 1, 7], s give s, 2] .
show bubble_sort[list [3, 1, 2], x give x] .
show counting_sort[list [3, 1, 2], x give x] .
show radix_sort[list [30, 4, 12], x give x] .
```

```
true
[ 3, 2, 1 ]
[ 9, 7 ]
[ 1, 2, 3 ]
[ 1, 2, 3 ]
[ 4, 12, 30 ]
```

The search library finds items. A search over a list gives the position, and a search over a map gives the key. A search that does not find its item gives none in the functions that return a position. jump_search, exponential_search and ternary_search need a sorted list, and find the position of the value in it:

```
import search
names = list ["Ann", "Bo", "Cy"]
ages = map ["Ann": 30, "Bo": 25]
show linear_search[names, "Bo"] .
show find_key[ages, 25] .
show find_index[names, x give x == "Cy"] .
show find_last[list [1, 4, 6, 8], x give x % 2 == 0] .
show jump_search[list [1, 3, 5, 7], 5, x give x] .
show exponential_search[list [1, 3, 5, 7], 7, x give x] .
show ternary_search[list [1, 3, 5, 7], 3, x give x] .
```

```
1
Bo
2
8
2
... 2 more lines
```

17.6 Functions at a glance: tables and the sorts that are not yet shown

table_column gives every value of one column of a table, as a list. table_row gives one row, counted from 0, as a record. Both take a table, which is a list of records:

```
import data
people = list [map ["first": "Ann", "last": "Lee"], map ["first": "Bob", "last":
    "Park"]]
show table_column[people, "last"] .
show table_row[people, 1] .
```

```
[ "Lee", "Park" ]
{ "first": "Bob", "last": "Park" }
```

selection_sort and shell_sort are two more of the sorts in the sort library. They take the same rule as the others, and they give a new sorted list:

```
import sort
show selection_sort[list [3, 1, 2], x give x] .
show shell_sort[list [5, 2, 9], x give x] .
```

```
[ 1, 2, 3 ]
[ 2, 5, 9 ]
```

Best practice. Read a file once into a list of rows, check that the columns you need are there, and then work on the list. Write results to a new file rather than overwriting the input.

Common mistake. Sorting a list that mixes kinds, such as numbers and text. Turtle stops with `can't sort string and integer together`. Convert the values to one kind first, as chapter 3 shows.

Key points

- **Records**: a record is a map or an assembled value, and a list of records is the usual shape for real data.
- `unique` drops repeated values from a list.
- `group` sorts each item into a map under a key that a function gives it.
- **Summaries**: a program can total and report values from a list of records.
- **Tables and CSV** files are read and written with the `data` library.
- **Statistics** such as `pvariance` and `pstdev` work on a list of numbers.
- `table_column` **and** `table_row` pull one column or one row out of a table.

18

Numbers, chance and text

Numbers in a computer. A whole number such as 7 is stored exactly. A number with a fraction, such as 0.1, is stored in binary, which cannot hold every fraction exactly. That is why a sum of decimals can show a tiny error, and why exact comparisons of floats are risky.

A random seed is the starting number of a random generator, and the same seed gives the same random values each time.

Chance and text. A random value is useful for testing and games, but a fixed **seed** makes the sequence repeat, so a result can be checked. Text is the most common data a program reads, and the `pattern` library finds and changes it.

Numbers, text and chance come into almost every program. A number is exact for whole values and approximate for floats, a random value is only useful if you can repeat it when testing, and text is the most common input a program reads. This chapter covers the `math`, `random`, `strings` and `pattern` libraries, and the `linear` library for matrices, which is numbers arranged in rows and columns.

18.1 Text and patterns

Libraries: `strings` for everyday text, `pattern` for regular expressions.

The most efficient way is the text methods. They read as sentences:

```
import strings
line = "order 42 shipped on 2026-10-30"
show line find "shipped" .
show line substring 0, 5 .
show list ["a", "b", "c"] join "-" .
```

```
9
order
a-b-c
```

`find` gives the position, counted from 0. `substring` takes a start and an end, with the end not included. The bracket forms, such as `find[line, "shipped"]`, give the same answers.

Use `pattern` when the text has a shape, such as dates or numbers in a sentence. Write the pattern in backticks, so its backslashes are kept:

```
import pattern

line = "order 42 shipped on 2026-10-30"
show findall[line, `\d+`] .
show replaceall[line, `\d{4}-\d{2}-\d{2}`, "DATE"] .
```

```
[ "42", "2026", "10", "30" ]
order 42 shipped on DATE
```

Other ways. `matches` answers yes or no, `splitby` cuts text at a pattern, and `groups` returns the parts a pattern captured:

```
import pattern

show splitby["a, b;c", `[,;]\s*`] .
show groups["2026-10-30", `(\d+)-(\d+)-(\d+)`] .
```

```
[ "a", "b", "c" ]
[ "2026", "10", "30" ]
```

Use plain text methods when a literal string will do. A pattern is more powerful, and harder to read at a glance.

18.2 A real program: finding patterns in text

`findall` returns every part of the text that matches a pattern. The pattern `[0-9]+-[0-9]+` matches a run of digits, a hyphen, and another run of digits. The program's comment says it finds email-like words; the pattern in fact finds the number groups, and the output shows exactly that.

```
// Find email-like words and digit groups in a line of text with the pattern library.
import pattern
text = "contact ana at 555-0199 or ben at 555-0042 today"
show "numbers: ", findall[text, "[0-9]+-[0-9]+"] .
```

```
numbers: [ "555-0199", "555-0042" ]
```

Write patterns in backticks, as the text section of this chapter explained, so the backslashes reach the pattern unchanged.

18.3 A real program: working on text

This program capitalises the first letter of each word. `text at split " "` gives a list of words, and `w at slice[0, 1] at upper` changes the first letter. `w at slice[1, w at length]` keeps the rest of the word, and `join` puts the result back together.

```
import strings
// Capitalise the first letter of every word in a sentence.
text = "the quick brown fox jumps over the lazy dog"
words = text at split " "
out = list []
[loop][w in words]
    cap = w at slice[0, 1] at upper + w at slice[1, w at length]
    add cap to out .
[loop][end]
show out at len, " words" .
show join[out, " "] .
```

```
9 words
The Quick Brown Fox Jumps Over The Lazy Dog
```

Text methods read left to right, so each step applies to the result of the one before it.

18.4 Matrices

What a vector is. A vector is an ordered list of numbers. It can describe a point, such as a position `(3, 5)`, or a set of measurements, such as the height, weight and age of one person. Its **length** is how many numbers it holds.

What a matrix is. A matrix is a rectangular grid of numbers, arranged in rows and columns. Each row can be read as a vector. A matrix can hold a table of measurements, one row per item and one column per measurement, or the numbers of a system of equations, as in the example below.

Why they matter. Vectors and matrices are the common shape of data in computing. Graphics rotate and scale points with them, simulations move objects with them, and much of machine learning works this way: a training set is a matrix, and learning a model is largely a long series of matrix products. A matrix product combines each row of the first matrix with each column of the second, so the number of columns of the first must match the number of items in the vector or the rows of the second.

Libraries: `linear`.

The most efficient way is to write the matrix in brackets, one row per line, and solve with `solve`:

```
import linear

a = matrix [
    2, 1
    1, 3
]
x = a solve list [3, 5]
show x .
show a * x .
```

```
[ 0.8, 1.4 ]
[ 3.0, 5.0 ]
```

`*` is the matrix product. To check an inverse, compare with `all_close`, not `==`. Floating-point arithmetic rounds, so two results that are equal in theory can differ in the last digit:

```
import linear
a = matrix [
    2, 1
    1, 3
]
show all_close[inverse[a] * a, identity[2]] .
```

```
true
```

Other ways. `determinant`, `inverse`, `transpose`, `rank` and `trace` answer the usual questions. `eigen`, `svd`, `lu` and `qr` are for decompositions, and `least_squares` fits a line to data that does not lie on one. Use `table_read` with `matrix` to make a matrix from a table.

Best practice. Check the sizes before you multiply. Write each matrix one row per line, so its shape can be read at a glance.

Common mistake. Multiplying a matrix by a list of the wrong length. Turtle stops and says how many numbers the list needs: `the list needs 2, one per column`.

18.5 A real program: matrix products

Matrices multiply by rows and columns, so `A * B` is not the same as `B * A`. This program multiplies two 2 by 2 matrices, then takes the determinant of the product. `determinant` of a 2 by 2 matrix is `ad - bc`, which gives `2` here.

```
// Multiply two 2 by 2 matrices with the linear library and show the determinant of
    the product.
import linear
A = matrix [
    1, 2
    3, 4
]
B = matrix [
    0, 1
    1, 0
]
show A * B .
show determinant[A * B] .
```

```
[ 2  1 ]
[ 4  3 ]
2
```

The `linear` library also solves systems, inverts matrices and factors them. This chapter's matrix section shows those.

18.6 Random values and simulations

Libraries: `random` and `math`.

The most efficient way is the sentence form of `random`:

```
import random

show random list of 5 integers from 0 to 9 .
show list ["red", "green", "blue"] pick .
```

```
[ 5, 0, 5, 8, 0 ]
green
```

The values you get will differ from the ones shown, because each run picks new random values.

For simulations, make the run repeatable with a seed when you need to check a result, and use `chance` for yes-or-no outcomes:

```
import random

show chance[0.5] .
```

```
false
```

`shuffle` gives a shuffled copy, and `sample` picks a number of items at random without repeats.

Math. Number methods work with `at`:

```
import math
show 16 at sqrt .
show 3.14159 at round 2 .
show 2 at pow 10 .
```

```
4.0
3.14
1024
```

18.7 A real program: seeded random values

`seed` makes the random values repeat. With `seed = 384`, the same program always prints the same values, which makes a simulation reproducible and a test possible. The `random` library writes its values as sentences: `random integer from 1 to 6`, `random list of 5 integers from 0 to 9`, and `random string of 6`.

```
// The random library writes values as sentences: integers, lists, strings and dates.
import random
seed = 384
show random integer from 1 to 6 .
show random list of 5 integers from 0 to 9 .
show random string of 6 .
show random digits of 4 .
show random list of 3 integers .
```

```
3
[ 5, 9, 2, 8, 1 ]
BONeDy
4415
[ -67, -678, -727 ]
```

Put `seed` at the top of a program when its output must be repeatable.

18.8 Functions at a glance: the math library

The `math` library turns a number into another number. Its functions are used as methods, with `at`, or in brackets. Every function needs `import math`.

The trigonometric functions work in radians. `atan2` takes the two parts of a point, so it gives the angle of the whole point, in every quadrant:

```
import math
show 1 at atan2[-1] .
show 1 at asin .
show 1 at cos .
show 2 at exp .
```

```
2.356194490192345
1.5707963267948966
0.5403023058681398
7.38905609893065
```

`floor` and `ceil` give the whole number at or below a value, and at or above it. `round` takes the number of decimal places, and rounds to that many:

```
import math
show 4.7 at floor .
show -4.2 at floor .
show 1 at ceil .
show 2.5 at round[0] .
show 1 at abs .
```

```
4
-5
1
3
1
```

Floats are stored with a small error, so `is_close` is the right test for two values that should be equal. The result is `true` when the two are within a small tolerance:

```
import math
show (0.1 + 0.2) at is_close[0.3] .
show (10 / 3) at is_close[3.33, 0.01] .
```

```
true
true
```

18.9 Functions at a glance: the linear library

The `linear` library works on matrices and on lists of numbers as vectors. `dot` is the dot product of two vectors, `norm` is the length of a vector, and `unit` gives the vector of length one that points the same way:

```
import linear
a = matrix [
    1, 2
    3, 4
]
b = matrix [
    5, 6
    7, 8
]
show dot[list [1, 2], list [3, 4]] .
show norm[list [3, 4]] .
show unit[list [3, 4]] .
show diagonal[a] .
show flatten[a] .
show multiply_each[a, b] .
```

```
11
5.0
[ 0.6, 0.8 ]
[ 1, 4 ]
[ 1  2  3  4 ]
... 2 more lines
```

`multiply_each` multiplies matching positions, which is different from the product `a * b`. `flatten` puts every number into one list, row by row.

18.10 Functions at a glance: text methods

Text has methods for padding, searching and changing case. A method is written with `at`, and it gives a new text, so the original is not changed. Padding is useful for columns and for fixed-width codes, such as an order number that must have three digits:

```
show "7" at padleft[3] .
show "ab" at padright[5] + "|" .
show "hello" at indexof["l"] .
show "Hello" at lower .
show "  x  " at trim .
show "hello" at slice[1, 3] .
show "hello" at reverse .
show "a-b-c" at split["-"] .
```

```
  7
ab   |
2
hello
x
... 3 more lines
```

`padleft` adds a fill character on the left until the text is the given width, and `padright` does the same on the right. The fill is a space unless a character is given: `"7" at padleft[3, "0"]` gives `007`. `indexof` gives the position of the first match, or `-1` when there is none. `slice` takes

the characters from the start up to, but not including, the end.

18.11 Functions at a glance: the rest of the math and random libraries

The remaining trigonometric functions are used the same way as `sin` and `cos`. `pi` is the constant the angles are measured with, so `pi` itself is the half turn. The inverse functions give the angle whose ratio is the number:

```
import math
show pi .
show 0 at sin .
show 1 at tan at round[4] .
show 0 at acos .
show 1 at atan at round[4] .
```

```
3.141592653589793
0.0
1.5574
1.5707963267948966
0.7854
```

`random` gives values of the kinds it knows. Each is a sentence, and `list of` makes a list of any of them. The `seed` setting makes the values repeat, which is useful when a test must check the same values each time:

```
import random
seed = 3
show random list of 4 letters .
show random list of 3 booleans .
show random string of 5 .
```

```
[ "T", "u", "o", "P" ]
[ false, true, false ]
pVwnn
```

`linear`'s `cross` gives the vector perpendicular to two vectors in three dimensions:

```
import linear
show cross[list [1, 0, 0], list [0, 1, 0]] .
```

```
[ 0, 0, 1 ]
```

`isnumber` is a text method that says whether the text can be read as a number:

```
show "12" at isnumber .
show "x" at isnumber .
```

```
true
false
```

Best practice. Compare floats with a small tolerance rather than with `==`, and round only for display. Set a seed when a test depends on random values.

Common mistake. Expecting `0.1 + 0.2 == 0.3` to be true. It gives `false`, because the sum is `0.30000000000000004`. Use the difference between the two numbers and a small limit instead.

Key points

- **Text and patterns**: `strings` handles everyday text, and `pattern` matches regular expressions.
- `findall` returns every part of a text that matches a pattern.
- **Working on words**: split a text into a list, change each word, and join the result back together.
- **Matrices** come from the `linear` library, and order matters: `A * B` is not the same as `B * A`.
- **Random values** come from `random`, and `seed` makes them repeat, which suits simulations.
- **Math and text functions**: `math` turns numbers into numbers, and text has methods for padding, searching and changing case.

19

Files, system and settings

What a program can reach. A program works inside a computer, and the computer keeps most of its information in **files**, which are named pieces of data on a disk, arranged in **folders** (also called directories). A program that only computes is limited; one that reads a file of orders, writes a report, or uses a setting from its surroundings can do real work. This chapter covers how a program finds and changes those files, and how it reads the information it is started with.

Paths. To use a file, a program needs its **path**: the text that names where it is. A **relative** path, such as `reports/2026.csv`, is read from the folder you start the program in, so that file must be there. An **absolute** path starts at the top of the disk and names the same file from anywhere, so it works whatever folder you start from. With `import system`, `scriptfolder[]` gives the folder that holds the program's own file.

Settings and arguments. Some values change between runs, such as a port number or a folder to read. A program can take them in three places: from the **command line**, which is the text typed after the program's name (`turtle report.turtle 2026`); from the **environment**, which is the set of named values the operating system gives every program; or from a **settings file** such as a TOML or JSON file. Keeping these values out of the code means the same program can run in a test folder and in a live one without any edits.

How Turtle does it. The file statements, such as `[read]` and `[write]`, need no import, so a program can read and write files directly. The `system` library checks whether a file exists, copies and moves files, and reads the command line with `args`. The `config` library reads and writes settings files, choosing the format from the file's extension.

19.1 Reading and writing files

Libraries: none for the basics. `system` for checking and moving files.

The most efficient way is the file statement. It needs no import:

```
[read] notes.txt to lines [end]
show lines .
```

```
[ "first line", "second line", "third line" ]
```

`lines` is a list with one item per line. To write a file, list the lines in a `[write]` block, and add to a file that already exists with `[append]`:

```
[write] notes.txt
"first line"
"second line"
[end]

[append] notes.txt
"third line"
[end]
```

Text in a file block can use `{...}` like any text, so a loop can fill a file:

```
[write] report.txt
"total is {total}"
[end]
```

Other ways. `[directory] . to entries [end]` lists a folder. For files and folders as things to check and move, use `system`:

```
import system
show exists["notes.txt"] .
show isfile["notes.txt"] .
makefolder["out"]
copyto["notes.txt", "out/notes.txt", true]
show contents["out"] .
```

```
true
true
[ "notes.txt" ]
```

`copyto` and `moveto` take a third argument, `true` to replace what is there. `pack` and `unpack` make and open archives, and `scriptfolder[]` gives the folder your program is in, so a program can find its own data files.

Pick the file statement for reading and writing text, and `system` when you need to ask about a file or move one.

19.2 A real program: reading command-line arguments

`args[]` gives the arguments after the program's name, as a list of text. This program checks how many were given, and falls back to a default when there are none. Run it with `turtle 397_system_arguments_default.turtle 2026`, and the output is `first argument: 2026`.

```
// Read command-line arguments; with none given the program uses a default.
import system
count = args[] at len
if ] count == 0 [
    show "no arguments, using default" .
else ]
    show "first argument: ", args[] at get[0] .
if [end]
```

```
no arguments, using default
```

Test `args[]` with `at len` before reading an item, so a missing argument never causes an index error.

19.3 Command-line options

`args[]` gives every word after the program's name, and leaves you to sort them out. `options` does the sorting. You list each option with a default, and it returns a map of the values the user ended up with. The default also sets the option's type: a number default takes a number, a `true` or `false` default makes a flag, and text takes text.

This program has three options. Its name is `report.turtle`:

```
import system
opts = options["--out": "a.csv", "-v": false, "--count": 10]
show opts .
show args[] .
```

With no options, every default is used, and `args[]` is empty:

```
$ turtle report.turtle
{ "--out": "a.csv", "-v": false, "--count": 10 }
[]
```

Values. Give an option its value after its name, or join them with `=`. Both forms below give the same result:

```
$ turtle report.turtle --out result.csv
{ "--out": "result.csv", "-v": false, "--count": 10 }
[]

$ turtle report.turtle --out=result.csv
{ "--out": "result.csv", "-v": false, "--count": 10 }
[]
```

Flags. A flag takes no value. Writing `-v` turns it on, and `-v=false` turns it off:

```
$ turtle report.turtle -v
{ "--out": "a.csv", "-v": true, "--count": 10 }
[]

$ turtle report.turtle -v=false
{ "--out": "a.csv", "-v": false, "--count": 10 }
[]
```

Words that are not options. Anything that isn't an option goes to `args[]` in the order it was typed:

```
$ turtle report.turtle data.csv --count 5
{ "--out": "a.csv", "-v": false, "--count": 5 }
[ "data.csv" ]
```

The rules. These apply to every option:

- **An option starts with `-` or `--`, and its name must be one you listed.** Single-dash names work too, so `options["-r": 0.08]` accepts `-r 0.3`. A name without a dash, such as `options["rate": 0.08]`, stops the program with an error when it reaches that line.
- **The value is the next word, even when that word starts with a dash.** Here `--out` takes `--count` as its text, and the `3` is left over in `args[]`:

```
$ turtle report.turtle --out --count 3
{ "--out": "--count", "-v": false, "--count": 10 }
[ "3" ]
```

To stop this, join the value with `=`:

```
$ turtle report.turtle --out=--count
{ "--out": "--count", "-v": false, "--count": 10 }
[]
```

- **A number option checks its value.** Text that isn't a number stops the program, and the message names the option:

```
$ turtle report.turtle --count abc
turtle: line 2: option --count takes a whole number; "abc" isn't one
```

- **A negative number is a value.** `--count -3` gives `-3`.
- **A word with a dash that is not listed is an error.** Typing `--zap`, or `---`, stops the program. The message lists the options you did define:

```
$ turtle report.turtle --zap
turtle: line 2: unknown option --zap; the options are --out, -v, --count (--help
    shows them)
```

- **A lone `-`, or a dash followed by a number such as `-5`, is not an option.** These go to `args[]`:

```
$ turtle report.turtle -5
{ "--out": "a.csv", "-v": false, "--count": 10 }
[ "-5" ]
```

- `--` **ends the options.** Every word after it goes to `args[]`, even one that starts with a dash:

```
$ turtle report.turtle -- --out x
{ "--out": "a.csv", "-v": false, "--count": 10 }
[ "--out", "x" ]
```

- **A value must follow.** `--out` on its own stops the program:

```
$ turtle report.turtle --out
turtle: line 2: option --out needs a value after it, as in --out a.csv
```

- **A repeated option takes its last value.** `--count 1 --count 2` gives `2`.

Help. `--help` (or `-h`) prints the options with their defaults, and ends the program. It is added for you, unless you list your own:

```
$ turtle report.turtle --help
usage: turtle report.turtle [options]

options:
  --out    default a.csv
  -v       on or off (off unless given)
  --count  default 10
```

Pick the right tool. Use `args[]` when the program takes a list of files or words. Use `options` when the user sets a named value, such as an output file or a count. Each has its own rules, so check both when you use either.

19.4 Settings and secrets

Libraries: `config` for settings files, `system` for environment variables, `crypt` for passwords and encryption.

The most efficient way to keep settings is a `config` file. Turtle reads the format from the file's extension: `.toml`, `.json` or `.env`:

```
import config

settings = map ["port": 8080, "debug": false, "name": "shop"]
config_write["app.toml", settings]
back = config_read["app.toml"]
show back at get "port" .
```

```
8080
```

For secrets that should not be in a file, read the environment:

```
import system
key = env["SHOP_KEY"]
```

19.5 Running a shell command

`sys` runs the rest of its line as a command of the operating system, and the command shares the program's input and output. It is the way to reach a tool that Turtle does not have, such as a command that lists files. It is deliberately powerful, and it is not limited in what it can do, so use it only with commands you have checked:

```
sys echo hello from turtle
```

```
hello from turtle
```

A shell command's output goes straight to the screen, not through `show`, so the program's own output and the command's output can appear in any order.

19.6 Functions at a glance: the system library

The `system` library works with folders, files, the environment, and the command line. Paths are text, and a folder path may use `/` on any system. The example below makes folders, writes a file, copies and moves it, packs a folder into an archive and unpacks it again. Run it in an empty folder: it creates files and folders, and it stops with an error if something it must create is already there. Each step is checked with a function that reports the result.

```
import system
makefolder["out/2026"]
[write] out/2026/report.txt
"first line"
[end]
show isfolder["out"] .
show isfolder["out/2026/report.txt"] .
show walk["out"] .
copyto["out/2026/report.txt", "backup/report.txt"]
show contents["backup"] .
moveto["backup/report.txt", "archive/old.txt"]
show walk["archive"] .
pack["out", "out.zip"]
unpack["out.zip", "restored"]
show walk["restored"] .
show env["NO_SUCH_VARIABLE_XYZ"] .
show length of args[] .
erase["archive"]
show exists["archive"] .
```

```
true
false
[ "out/2026/report.txt" ]
[ "report.txt" ]
[ "archive/old.txt" ]
... 4 more lines
```

`walk` lists every file under a folder, in its subfolders too. `contents` lists only the names directly inside a folder. `copyto` and `moveto` refuse to replace a file that is already there, unless the third argument is `true`. `erase` removes a file or a folder and everything in it, and there is no undo, so use it with care.

`env` gives an environment variable, or `none` if it is not set. `args` gives the words typed after the program's name, so this program prints `0` when it is run with none. `exit[code]` ends the program at once, with code `0` for success and anything else for failure.

19.7 Functions at a glance: memory and size

A program's values use memory, and `sizeof` estimates how much a value takes. `freememory` gives memory back at once, and reports how many bytes it freed. Turtle frees unused values by itself, so `freememory` is worth calling only between large steps of a program, such as after a large file has been processed:

```
import system
show sizeof["abc"] > 0 .
show freememory[] >= 0 .
```

```
true
true
```

Best practice. Check that a file exists before you read it, and keep settings out of the code so they are easy to change.

Common mistake. Giving a path that does not exist. Turtle stops with `no such file or folder` and names the file it looked for, so check the path and the folder the program runs from.

Key points

- `args[]` gives every word after the program's name, as text, with nothing sorted out for you.
- `options` gives each option a default, and the default sets the type: a number, text, or a flag.
- **Dashes**: options start with `-` or `--`, and a value is the next word even when it starts with a dash, so join it with `=` when that matters.
- **Unknown dashes** stop the program, so a typo shows up at once.
- `copyto` **and** `moveto` refuse to replace a file unless you pass `true`, and `erase` has no undo.
- **Settings** belong in a `config` file, and secrets are read from the environment with `env`.
- `sys` runs a command of the operating system, so use it only with commands you have checked.

20

The internet and databases

Programs that talk to each other. Most programs on the internet work in pairs. A **client** asks for something, and a **server** waits for requests and sends back answers. A web browser is a client, and a web server is the program that answers it. A Turtle program can be either one. It can ask a server for data with `http_get`, or it can run a server that answers requests with `serve`.

`HTTP` is the protocol that programs use to send requests and receive answers over the internet.

Requests and answers. A request names what is wanted and how. A `GET` request asks for data, and a `POST` request sends data to be stored. Each answer carries a **status code**: `200` means the request worked, `404` means nothing is at that address, and a `5xx` code means the server failed. Checking the code before you use the answer keeps a failed request from being mistaken for good data.

Structured data. Data sent over the network needs a shared form, and **JSON** is the most common one. It is text made of maps, lists, text, numbers, `true`, `false` and `null`, which lets two programs written in any language read the same data.

Databases. A **database** keeps data in **tables** of **rows** and **columns**, and it can search, sort and change them quickly and safely. Programs ask a database questions in **SQL**, a language for reading and changing tables. A single program can use a database file, and several programs that share data are usually better served by a database server.

How Turtle does it. The `http` library sends requests, `json` reads and writes JSON text, and `server` builds a server from a map of routes. The `sql` library opens a database file or a server, runs SQL statements, and moves rows to and from CSV files.

20.1 Talking to the web and building a server

Libraries: `http` to ask, `json` to read the answer, `server` to answer others.

The most efficient way to read an API is `http_get` with `load`. `load` reads JSON text into maps and lists:

```
import http
import json

body = http_get["http://127.0.0.1:8765/data"]
users = load[body]
show users json_get "name" .
```

```
Ann
```

`json_get` follows a path of keys and indexes, and gives `none` if the path is missing, so a missing field does not stop the program. The bracket form, `json_get[users, "name"]`, is the same call.

Send data with `http_post`. A map or list sent as the body goes as JSON:

```
import http

reply = http_post["https://example.com/api/users", map ["name": "Ann"]]
```

A server in a few lines. `server` takes a map of routes. Each key is a method and a path, and each value is a function that answers it:

```
import server

def hello[]
    return reply[200, "{\"name\": \"Ann\"}", map ["Content-Type": "application/json"]]
def [end]

routes = map ["GET /data": hello]
serve[routes, 8765]
```

Run it, and `http_get` above reads from it. `reply` gives a status code, a body and optional headers. `redirect` sends a visitor to another address.

Other ways. Use `json_text` to turn a value back into JSON text, and `json_read` and `json_write` to read and save JSON files:

```
import json

user = json_read["user.json"]
json_write["copy.json", user]
```

20.2 A real program: writing JSON

`json_text` turns a map or list, including nested ones, into JSON text. The output is one line, with no spaces, because JSON does not need them. Use `json_write` to save the same text to a file.

```
// Turn a nested map and list into JSON text.
import json
doc = map ["id": 7, "tags": list ["a", "b"], "meta": map ["ok": true]]
show json_text[doc] .
```

```
{"id":7,"tags":["a","b"],"meta":{"ok":true}}
```

`load` reads JSON text back into maps and lists, so the two functions make a round trip.

20.3 Databases

Libraries: `sql`.

The most efficient way uses one SQLite file. `sql_create` makes a new file, and `sql_open` opens one that already exists:

```
import sql

db = sql_create["shop.db"]
sql_run[db, "CREATE TABLE IF NOT EXISTS books (title TEXT, price INTEGER)", list []]
sql_run[db, "INSERT INTO books VALUES (?, ?)", list ["Emma", 900]]
cheap = sql_query[db, "SELECT title FROM books WHERE price < ?", list [1000]]
show cheap .
sql_close[db]
```

```
[ { "title": "Emma" } ]
```

`sql_run` changes the database, and `sql_query` reads from it. Values go in a list, and each `?` in the query takes the next one. This keeps values safe from being read as commands.

Tables to files and back. `sql_save` writes a query's rows to a CSV file, `sql_load` reads a CSV file into a table, and `sql_upsert` adds or replaces rows from a file by a key:

```
import sql

db = sql_open["shop.db"]
sql_upsert[db, "books", "title", "books_in.csv"]
sql_close[db]
```

`sql_update` and `sql_delete` do the same by key, for changing or removing rows. `sql_tables` lists the tables in a database.

Other ways. The same functions work with PostgreSQL and MySQL servers: the address you pass to `sql_open` names the server. Use a file database for a single program, and a server when several programs share the data.

20.4 Functions at a glance: JSON values

A JSON value can be `null`. Turtle reads `null` as `none`, and writes `none` as `null`, so a missing value keeps its meaning when the data is sent or saved:

```
import json
show json_text[map ["x": none]] .
j = load['{"x": null}']
show j at get "x" .
```

```
{"x":null}
none
```

20.5 Functions at a glance: the server library

A server answers requests from a map of routes. Each route is written as a method and a path, and each handler is a function that receives the request. The program below answers a page, a user record with a parameter from the path, a posted user, a redirect and a health check. It runs until it is stopped. Its output is the startup line, followed by the answers to the requests shown after it.

```
import server

// A handler is a function that receives one request, called req.
// It gives back the answer for that request.

// For GET /: return a page of HTML text. The status is 200 by default.
def home[req]
    return "<h1>Hello!</h1>"
def [end]

// For GET /users/:id: the id is part of the path, such as 7 in /users/7.
// params of req is the set of path parts, and id of params gets the one named id.
def showuser[req]
    id = id of params of req
    return map ["id": id, "name": "Ann"]
def [end]

// For POST /users: json of req reads the JSON that the client sent.
// reply gives the status 201, which means "created", and the answer text.
def adduser[req]
    data = json of req
    return reply[201, map ["made": name of data]]
def [end]

// The map of routes. Each key is a method and a path, and each value answers it.
// A function is called for each request. A fixed value is sent as it is.
app = map [
    "GET /": home,
    "GET /users/:id": showuser,
    "GET /users/new": reply[200, "the new-user form"],   // a fixed answer
    "POST /users": adduser,
    "GET /old": redirect["/"],                            // sends the visitor to "/"
    "GET /health": "ok"                                   // a plain text answer
]

// Start the server on port 8093. The program keeps running until you press Ctrl+C.
serve[app, 8093]
```

```
serving on http://localhost:8093 (Ctrl+C stops it)
```

Each route is matched by its method and path. `:id` matches one part of the path, and its value is read from `params`. The most exact route wins, so `GET /users/new` is answered by its own route, not by `GET /users/:id`. A path with no route gives `404`. A `redirect` gives `302` and the address to go to. `reply` sets the status code, and `json of req` reads the JSON that was posted.

20.6 Functions at a glance: the sql library

A database holds tables of rows. The `sql` library creates a database, runs statements that change it, reads rows back as a list of records, and moves rows to and from CSV files. The sequence below does each of these in order, from an empty folder. It writes the two CSV files it needs first, so it runs as it stands:

```
import sql
import data
[write] incoming.csv
"title,price"
"Emma,950"
"New,300"
[end]
[write] gone.csv
"title"
"Ida"
[end]
db = sql_create["shop.db"]
sql_run[db, "CREATE TABLE books (title TEXT PRIMARY KEY, price INTEGER)", list []]
sql_run[db, "INSERT INTO books VALUES (?, ?)", list ["Emma", 900]]
sql_run[db, "INSERT INTO books VALUES (?, ?)", list ["Ida", 1500]]
show sql_tables[db] .
show sql_query[db, "SELECT title FROM books WHERE price < ?", list [1000]] .
sql_save[db, "SELECT title, price FROM books", "books.csv", list []]
show table[table_read["books.csv"]] .
sql_upsert[db, "books", "title", "incoming.csv"]
sql_delete[db, "books", "title", "gone.csv"]
show sql_query[db, "SELECT title, price FROM books ORDER BY title", list []] .
sql_close[db]
```

```
[ "books" ]
[ { "title": "Emma" } ]
title  price
-----  -----
Emma     900
... 2 more lines
```

`sql_run` changes the database, and `?` marks a value that is passed in a list, which keeps values from being read as part of the statement. `sql_query` gives the rows, one map per row. `sql_save` writes a query's rows to a CSV file, and `sql_upsert` adds or replaces rows from a CSV file, matched by the key column. `sql_delete` removes the rows whose keys are listed in a file.

Best practice. Keep the address of a server and the name of a database in settings, not in the code. Check each answer's status before you use its data.

Common mistake. Creating a database file that already exists. Turtle stops with `the file already exists`, and reminds you that `sql_open` opens an existing file.

Key points

- **Talking to other programs**: web servers and APIs send and receive requests, which the `http`, `json` and `server` libraries handle.
- `json_text` turns a map or list, including nested ones, into JSON text.
- **JSON** `null` is read as `none`, and `none` is written as `null`.
- **Server routes**: a server answers requests from a map of routes, and each route is a method.
- **Databases** hold tables of rows, and the `sql` library creates a database and runs statements against it.

21

Time, security and development

Why dates are hard. A date looks simple, but it has rules: months have different lengths, some years have an extra day, and the same moment can have a different clock time in different places. A program that does date arithmetic by adding numbers by hand gets these wrong. A **datetime** holds a date and a time together, and a time library does the calendar work for you. Use it whenever a program adds days, compares dates, or shows a time to a person.

A log level is the importance of a log line, such as debug, info, warn or error.

Why programs keep a log. A program that runs unattended cannot show you what it is doing, so it writes a **log**: one line for each important event, with a time and a level. The levels, such as `debug`, `info`, `warn` and `error`, let a reader show only the lines that matter. When something fails at night, the log is often the only record of what happened.

Why passwords are not stored. A program that checks a password should never keep the password itself. A **hash** is a fixed-length value worked out from the password. The program stores the hash, and when a user types a password it works out the hash again and compares the two. Turtle's `crypt` library does this with `passwordhash` and `passwordcheck`. An **encryption** is different: it hides data so that only someone with the right **key** can read it again.

Running many jobs at once. Some tasks wait on other things, such as several web addresses, and running them one at a time wastes time. A **schedule** starts many jobs together, and `fetchall` gives back each answer in the order the addresses were given.

Keeping secrets out of the code. A password or a key written into a source file can be copied, shared and committed without anyone noticing. Keep secrets in a settings file or the environment, as chapter 19 explains, and keep them out of the log.

How Turtle does it. The `time` library gives the date and time of now with `today[]`, does date arithmetic with `add_time` and `time_between`, and shows dates with `format_date`. The `log` library writes levelled lines to the screen or a file. The `crypt` library makes hashes, passwords and encrypted data. The `schedule` library runs many jobs at once.

Every function in the `time`, `crypt`, `log` and `schedule` libraries, with its full details and examples, is covered on the Turtle website, at turtlelang.dev. The sections that follow explain the most common ones.

21.1 Dates and schedules

Libraries: `time` for dates and clocks, `schedule` for doing many things at once.

The most efficient way to do date arithmetic is `add_time` and `time_between`:

```
import time

now_day = today[]
due = now_day add_time 30, "days"
show now_day time_between due, "days" .
show format_date[due, "YYYY-MM-DD"] .
```

```
30
2026-12-18
```

`today` gives the date, and `now` gives the date and time. Time zones are handled with `to_zone`, and `make_date` builds a date from its parts.

Waiting. `sleep` pauses for an amount of time, and `wait_until` pauses until a given date. `every` runs a job repeatedly, for example checking an inbox every hour:

```
import time
every[1, "hours", check_inbox]
```

`every` keeps the program running, so use it in a program meant to stay up, not in a quick script.

Many jobs at once. `schedule` runs many requests in one call. `fetchall` asks for each web address and gives back the pages in the same order. The addresses above are answered by the server from section 20.1, so start that server first:

```
import schedule

urls = list ["http://127.0.0.1:8765/data", "http://127.0.0.1:8765/data"]
results = fetchall[urls, map []]
show length of results .
```

```
2
```

The settings map can add a limit on how many run at once, or skip items that fail: `map ["limit": 4, "skip_errors": true]`. `runall` does the same for shell commands, and `queryall` for database queries.

Other ways. Use a plain loop with `http_get` when there are only a few requests. Use `schedule` when there are many, because it runs them together.

21.2 Passwords and encryption

Passwords are stored as a hash, never as the password itself. `passwordhash` makes the hash, and `passwordcheck` tests a password against it:

```
import crypt

stored = passwordhash["secret"]
show passwordcheck["secret", stored] .
show passwordcheck["wrong", stored] .
```

```
true
false
```

Encrypting text needs a passphrase. The same passphrase decrypts it:

```
import crypt

box = encrypt["secret", "pw"]
show decrypt[box, "pw"] .
```

```
secret
```

Other ways. `hash` gives a checksum of text, `filehash` one of a file, and `hmac` one keyed with a secret. `token` makes a random string and `uuid` a unique id. Use `loadenv` to load a `.env` file into the environment.

21.3 Logging

Libraries: `log`.

The most efficient way is one line per event, with a level:

```
import log

log info "started" .
log warn "disk at ", 91, "%" .
log error "bad thing" .
```

```
2026-10-30 08:30:52 INFO  started
2026-10-30 08:30:52 WARN  disk at 91%
2026-10-30 08:30:52 ERROR bad thing
```

Lines below the program's level are hidden. The default level is `info`, so a `debug` line does not appear until you lower the level:

```
import log
loglevel = "debug"
log debug "details ", 3 .
```

```
2026-10-30 13:58:49 DEBUG details 3
```

A map adds fields to the line, which is useful for a server that logs each request:

```
import log
name = "ann"
log info "login", map ["user": name] .
```

```
2026-10-30 13:58:49 INFO  login user=ann
```

Other ways. To keep logs in a file, set `logfile`. For machines that read the logs, set `logformat` to `"json"`, which writes one JSON object per line. `logmaxsize` rotates the file when it grows past a size.

21.4 A real program: a logged job

`log` writes a line with a time, a level and a message to the error stream, so it does not mix with the program's own output. `info`, `warn` and `error` are the levels, and a line below the program's `loglevel` setting is hidden.

```
// Log messages at different levels.
import log
log info "starting job" .
log warn "disk is 90 percent full" .
log error "could not open file" .
show "done" .
```

```
done
2026-10-30 13:35:53 INFO  starting job
2026-10-30 13:35:53 WARN  disk is 90 percent full
2026-10-30 13:35:53 ERROR could not open file
```

The times in the output are the times of the run, so yours will differ. The level and message are the same every time.

21.5 Functions at a glance: the time library

A date has parts, and each part is read with `of`. `make_date` builds a date from its parts, and `to_date` reads one from text. `format_date` writes a date with a pattern, and `to_zone` gives the same moment on another zone's clock:

```
import time
d = make_date[2026, 12, 25, 9, 30]
show format_date[d, "DD/MM/YYYY hh:mm"] .
show year of d .
show month of d .
show day of d .
show hour of d .
show minute of d .
show weekday of d .
show zone of d .
```

```
import time
e = to_date["2026-10-03 14:05"]
show format_date[e, "Weekday, Month D"] .
show to_zone[make_date[2026, 12, 25, 9, 30], "UTC"] .
show time_between[make_date[2026, 1, 1], make_date[2026, 1, 8], "days"] .
```

```
Saturday, October 3
2026-12-25 14:30:00 UTC
7
```

A date made without a zone is in the computer's own zone, which is shown as `local`.

21.6 Functions at a glance: the crypt library

`hash` gives a fingerprint of a text. The same text always gives the same fingerprint, and a different text gives a different one. `sha256` is the default, and it should be used for anything new. `md5` and `sha1` are offered only to match older systems, and they must not be used to protect data. `encode` and `decode` change text to and from its representation as text, such as base64 or hexadecimal:

```
import crypt
show hash["abc", "sha256"] .
show hash["abc", "sha1"] .
show hash["abc", "md5"] .
show encode["hi", "base64"] .
show decode[encode["hi", "base64"], "base64"] .
show encode["hi", "hex"] .
show decode["6869", "hex"] .
show length of token[4] .
show length of uuid[] .
```

```
ba7816bf8f01cfea414140de5dae2223b00361a396177a9cb410ff61f20015ad
a9993e364706816aba3e25717850c26c9cd0d89d
900150983cd24fb0d6963f7d28e17f72
aGk=
hi
... 4 more lines
```

`token` and `uuid` make random values, so their contents differ each time. Their length is fixed, which is why the example shows the length of each.

21.7 Functions at a glance: the log settings

The `log` library is controlled by settings that are variables in the file that imports it. `loglevel` chooses the lowest level that is shown, so a `warn` setting hides `info` lines. `logtime` sets the time format, and `none` removes the time from each line:

```
import log
logtime = none
loglevel = "warn"
log info "hidden" .
log error "shown" .
```

```
ERROR shown
```

`logconsole = false` stops the lines from going to the screen. With `logfile`, the lines are added to a file, and `logformat = "json"` writes one JSON object for each line. The example below writes its log to a file, then reads the file back and counts its lines:

```
import log
logconsole = false
logfile = "app.log"
logformat = "json"
log info "started", map ["port": 8080] .
log warn "disk 90 percent full" .
[read] app.log to lines [end]
show length of lines .
```

```
2
```

21.8 Functions at a glance: the schedule settings

`schedulelimit` sets how many requests run at the same time, and `skipschedule_error` decides what happens when one of them fails. With it set to `true`, a failed request gives `none` in its place, and the rest carry on. The addresses below refuse the connection, because nothing listens on port 1:

```
import schedule
schedulelimit = 2
skipschedule_error = true
pages = fetchall[list ["http://127.0.0.1:1/a", "http://127.0.0.1:1/b"]]
show pages .
```

```
[ none, none ]
```

21.9 Functions at a glance: the log parts and the output file

`logparts` chooses which parts of each line are written, in the order given. With only the level and the message, a line has no time and no file name. `outputfile` names a file that also receives everything `show` and `warn` print, so a run can be kept:

```
import log
logconsole = false
logfile = "run.log"
logparts = list ["level", "message"]
log warn "low disk" .
[read] run.log to lines [end]
show lines .
```

```
[ "WARN  low disk" ]
```

The `time` library's `today_utc` gives the current date in UTC, and `year of` reads its year:

```
import time
show year of today_utc[] > 2000 .
```

```
true
```

Best practice. Log what happened and why, but never write passwords, keys or personal details into a log. Store secrets in a settings file or the environment, not in the code.

Common mistake. Writing a date by hand instead of computing it. A date typed in by hand goes out of date, so compute it from the current date with the `time` library.

Key points

- **Dates and schedules**: the `time` library handles dates and clocks, and `schedule` runs many tasks.
- **Passwords** are stored as a hash, never as the password itself.
- `hash` gives a fingerprint of a text, and the same text always gives the same fingerprint.
- **Logging**: `log` writes a line with a time, a level and a message to the error stream.
- **Log and schedule settings** are variables in the importing file, so the program can change them.
- **Date parts**: `make_date` builds a date, and `of` reads each part of it.

Part III: Checking your code

Turtle checks code in three ways. **Tests** say what must be true, and `turtle test` runs them. `diagnose` shows what a piece of code did, in the middle of a run. `hypothesis` tests a claim about your data, and tells you where it breaks. Chapter 22 covers the first, and chapter 23 the other two.

22

Tests

What testing is. A **test** runs part of a program on known inputs and checks the result against what you expected. Tests catch mistakes early, and they show later that a change has not broken anything that worked before. A test that fails points to the input that caused it.

How Turtle does it. A test file has a name that starts with `test_` and ends with `.turtle`. `turtle test` runs every such file, and `check` states one expectation. Theories (chapter 12) are checked the same way.

You do not need a test file to use these checks. `check`, `verify` and `validate` also work in an ordinary program, where a `check` that fails stops the program and prints a report. A test file is where you collect the checks so that `turtle test` runs all of them together.

You can use `check`, `verify` and `validate` together as a suite, in the same file, or use each one separately. A file with only one of them works in the same way.

A **test** is a statement about what a program must do, and it is checked automatically. Turtle reads tests from files whose names start with `test_`, and runs them with `turtle test`. A passing test shows that the cases it checks behave as expected. A failing test shows the input that caused the failure. This chapter covers the checks and how to write them.

22.1 A test file

A test lives in a file whose name starts with `test_`, and ends in `.turtle`. Each test is a function whose name starts with `test_`. Inside a test, you state what must be true:

```
import test
import data

def evens[nums]
    return keep[nums, x give x % 2 == 0]
def [end]

def test_evens_are_even[]
    check length of evens[list [1, 2, 4]] == 2 .
    verify evens[list [1, 2, 4]] each x give x % 2 == 0 .
def [end]

def test_sum_small[]
    check sum[list [1, 2]] == 3 .
def [end]
```

Save it as `test_lists.turtle` and run `turtle test test_lists.turtle`. The output looks like this (the timings will differ on your machine):

```
test_lists.turtle
  PASS  test_evens_are_even   37.6µs     102.4 KB
  PASS  test_sum_small        2.1µs      424 B

ok: 2 passed, 0 failed (1 file, 626.1µs)
```

Run the tests with `turtle test`, naming the file:

```
$ turtle test test_lists.turtle
test_lists.turtle
  PASS  test_evens_are_even   55.2µs     102.4 KB
  PASS  test_sum_small        2.5µs      424 B

ok: 2 passed, 0 failed (1 file, 392.0µs)
```

Each test shows whether it passed, how long it took, and how much memory it used. The last line totals the run.

22.2 `check`: one fact

`check` states one fact. It is true or it is not:

```
import test
import data
nums = list [4, 6]
total = 45
check total == 45 .
check sum[nums] == 10 .
```

When a check fails, the test says what was expected and what was found:

```
  FAIL  test_fails   3.1µs      912 B
        test_tally.turtle line 8: failed: check 1 == 2 .
            got 1, want 2 (1 less)
```

22.3 `verify`: a rule over every item

`verify` checks a rule against every item in a list, and reports the first item that breaks it. It needs `import test` and `import data` first:

```
import test
import data
nums = list [1, 2, 3]
verify nums each x give x < 3 .
```

```
turtle: line 4: failed: verify nums each x give x < 3 .
    1 of 3 items breaks the rule:
      at 2: 3
  4 | verify nums each x give x < 3 .
```

The failure names the item's position and value, so you do not have to find it yourself.

`verify` also reads as English. `verify list [1, 2, 3] at least 2 x give x > 1 .` checks that at least two items pass the rule.

22.4 `validate`: a rule over random inputs

`validate` tries a rule on many random inputs, and reports the smallest input that breaks it. Use it for a function whose correct behavior holds for any input of a kind:

```
import test
import data

def halve[n]
    return n div 2
def [end]

def test_halve_is_smaller[]
    validate halve[n] with n as integer that n >= 0 .
def [end]
```

```
test_validate.turtle
  FAIL  test_halve_is_smaller   106.5µs    130.1 KB
        test_validate.turtle line 9: failed: validate halve[n] with n as integer that
            n >= 0 .
            failed on case 1 of 100 (seed 413097370)
            n = -1   (smallest found; first failed on -341)
            result = 0
            the rule is false: n >= 0
            n is -1, which isn't >= 0
            to repeat this run: seed = 413097370

FAILED: 0 passed, 1 failed (1 file, 529.0µs)
```

This is a deliberate failure, and the output shows what good failure output looks like. Turtle found `-1`, the smallest input that breaks the rule. It also gives the seed, so the same run can be repeated exactly.

To make the rule hold, you can either limit the input (`with n as integer that n >= 0`, which the example already does) or change the rule. The test does not guess which one you meant.

22.5 Running a folder

`turtle test` with no file runs every `test_*.turtle` file in the folder and the folders below it. Give it a folder to limit the run:

```
$ turtle test tests
```

Each file prints its results under its name, and the last line totals every file.

22.6 Testing theories

Theories are checked by `turtle test` as well. The proof examples must give what they say, and each theorem must hold for them and for 100 random inputs. Chapter 12 shows a theory with its proof and theorems, and the output below is from a test file that imports it:

```
test_tally_theory.turtle
  PASS  test_tally_counts   32.9µs     100.8 KB
  PASS  theory tally   proof 2 of 2, 2 theorems hold on 100 random inputs

ok: 1 passed, 0 failed, 1 theory proven (1 file, 518.5µs)
```

The theory line shows how many proof examples passed, and how many theorems held. A theory that breaks its promise is reported with the input that breaks it, the same way `validate` reports a function.

22.7 Which tests to write

A good test file has three kinds of test:

- **Examples**: a `check` for each case you know the answer to, including the edges: empty, zero, negative, one item.
- **Rules**: a `verify` for each property every result must have.
- **Random inputs**: a `validate` for each function that should hold for any input of its kind.

The edge cases matter most, because random inputs find them only by chance.

22.8 Benchmark tests

A test is timed when it runs, and the time is shown with the result. A benchmark runs a test many times, to give an average time for the work it does. Set `benchmark = true` at the top of a test file, or inside one test, and the test is repeated:

```
import test
import data
benchmark = true

def test_sum[]
    check sum[list [1, 2, 3]] == 6 .
def [end]
```

The output shows how many runs were made, the average time of one run, and the fastest and slowest runs. The times depend on the machine, so they differ from one run to the next. The result of the check is the same every time.

Best practice. Write a test for each rule the program depends on, including the edge cases such as an empty list or zero. Run the tests after every change, not just at the end.

Common mistake. Naming a test file without the `test_` start. Turtle says the name must start with `test_` and end with `.turtle`, and the file does not run.

Key points

- **Test**: a statement about what a program must do, checked automatically.
- **Test files** have names starting with `test_` and ending in `.turtle`.
- `check` states one fact, and it is either true or not.
- `verify` checks a rule against every item in a list.
- `validate` tries a rule on many random inputs.
- `turtle test` with no file runs every test file in the folder.
- **Theories** are checked by `turtle test` as well.
- **Benchmarks** time a test when it runs, and show the time with the result.

23

Inspection: `diagnose` and `hypothesis`

Inspecting and claims. A check tells you that something is wrong. Inspecting tells you where and why: which values went into a piece of code, what each step gave back, and which item broke a rule. Without inspection tools, a programmer finds such faults by adding print lines by hand and running the program again. Turtle has two tools built into the language for this job.

In plain words, diagnose shows what a piece of code did, and hypothesis checks whether a claim about a list holds.

`diagnose` shows what a piece of code did while it ran. It can wrap one function call, one expression, or a whole block of statements. For a function, it shows the value it was given, the result it returned, how long it took and how much memory it used. The result is still given back, so the program carries on as before, and the diagnosis is only extra information.

`hypothesis` tests a claim about every item of a list, written in almost plain words, such as "every price is above zero". It says whether the claim holds for all the items. If it does not, it says how many items break the claim and shows the first one that does. It needs no import, and it does not stop the program, so a claim that fails is a finding, not an error.

When to use which. Use `diagnose` and `hypothesis` while you are looking for a fault or exploring an idea. They change nothing in the program. Once you know what must always be true, write it as a `check`, `verify` or `validate` (chapter 22), so that a test catches it from then on.

How Turtle does it. `diagnose[...]` takes the function or expression to inspect, in brackets, and `diagnose` on a line of its own starts a block that ends with `diagnose [end]`. `hypothesis[...]` takes a list, the word `each`, the name for each item and the rule.

23.1 `diagnose`: one line, one block, or any code

`diagnose` shows what a piece of code did. It has three forms.

One use. `diagnose[...]` around a single phrase or call shows its values, its result, each theorem that applies, and how long it took:

```
theory tally
    abstract
        tally says how many times b appears in the list a.
    notation tally b in a .
    definition
        n = 0
        [loop][x in a]
            if ] x == b [
                n = n + 1
            if [end]
        [loop][end]
        return n
    theorem result >= 0
    theorem result <= length of a
    proof
        tally 1 in list [1, 1, 0, 3] . is 2
        tally 9 in list [] . is 0
theory [end]
nums = list [1, 1, 0, 3]
diagnose[tally 1 in nums]
```

```
diagnose theory tally (line 20)
  notation  tally b in a .
  b         1
  a         list of 4: [ 1, 1, 0, 3 ]
  returned  2
  theorem   result >= 0            holds
  theorem   result <= length of a  holds
  took      45.7µs
  memory    100.5 KB allocated
```

A whole theory. `diagnose[tally]` runs the theory's proof and theorems, as `turtle test` does, and shows where they pass or fail:

```
diagnose theory tally (line 2)
  notation  tally b in a .
  proof
    tally 1 in list [1, 1, 0, 3] . is 2  ✓
  theorems, on the proof cases and 100 random inputs (seed 557434950)
    result >= 0                          holds
```

A block of code. A `diagnose` block with its own `[end]` shows each line as it runs, with the value it got:

```
nums = list [4, 8]
diagnose
    first = nums at get 0
    second = nums at get 5
diagnose [end]
show "after" .
```

```
diagnose (lines 3-4)
  line 3      first = nums at get 0         first = 4
  line 4      second = nums at get 5
  ✗ failed: index error: line 4: index 5 out of range for list (length 2)
  memory    2.2 KB allocated
after
```

The block stopped at the failing line, and the program kept going, because a `diagnose` block does not stop the program. Use it to find out what a section of code is doing, without changing it.

A `diagnose` block runs the same code as any other, so a missing import is still an error. Inside a `diagnose` block that calls `sum`, for example, you see the same "needs import data" message as anywhere else.

23.2 `hypothesis`: test a claim

`hypothesis[...]` tests a claim about your data. It says whether the claim holds, how many items break it, and which one breaks it first. The times and memory in its output change on each run:

```
prices = list [5, -2, 9]
hypothesis[prices each p give p > 0]
show "still running" .
```

```
hypothesis prices each p give p > 0 (line 2)
  holds     no: 1 of 3 items (33.3%) breaks it
  prices    list of 3: [ 5, -2, 9 ]
  where     at 1  -2   (p is -2, which isn't > 0)
  took      63.5µs
  memory    103.1 KB allocated
still running
```

Where `verify` stops a test run, `hypothesis` reports on your real data and keeps going. Use it to ask questions of a list before you trust it:

```
nums = list [1, 2, 3]
hypothesis[nums each x give x > 0]
```

```
hypothesis nums each x give x > 0 (line 2)
  holds     yes: all 3 items follow it
  nums      list of 3: [ 1, 2, 3 ]
  took      66.1µs
  memory    102.4 KB allocated
```

A claim can also be a theory's theorem, checked before it goes into the theory. `hypothesis[tally, theorem result >= 0]` tests that theorem on the theory `tally`.

Like `diagnose`, `hypothesis` does not stop the program. It is a question you ask as you go.

23.3 Keeping reports in a file

Both tools write to the screen. To keep their reports, set `reportfile` to a file name:

```
reportfile = "checks.txt"
nums = list [1, 2, 3]
hypothesis[nums each x give x > 0]
```

```
hypothesis nums each x give x > 0 (line 3)
  holds     yes: all 3 items follow it
  nums      list of 3: [ 1, 2, 3 ]
  took      54.6µs
  memory    102.4 KB allocated
```

Each report is added to the file, with a time stamp on its first line:

```
== 2026-10-30 08:10:39  j_report.turtle line 3
hypothesis nums each x give x > 0 (line 3)
  holds     yes: all 3 items follow it
  nums      list of 3: [ 1, 2, 3 ]
  took      83.2µs
```

Set `reportfile` at the top of a program, and every `diagnose` and `hypothesis` report in that program is kept, which makes a run easy to review later.

23.4 Diagnose and hypothesis with theories

Chapter 12 introduced theories. The two tools are how you check one from outside the code:

- `diagnose[tally]` runs its proof and theorems and shows the result,
- `hypothesis[tally]` tests the theory the same way,
- `diagnose[tally 1 in nums]` shows one use of it, with its values.

When a theory is wrong, these tell you which use is wrong, and which theorem it breaks. That is faster than reading the definition line by line.

Best practice. Use `diagnose` while you are finding a fault, then take it out once the fault is fixed. Check a claim with `hypothesis` before you rely on it.

Common mistake. Trusting a claim that has only been checked on a few values. `hypothesis` reports the items it tested, so read the count as well as the result.

Key points

- `turtle test` says whether something is true, and two tools explain why.
- `diagnose` shows what a piece of code did, in three forms.
- `hypothesis[...]` tests a claim about your data, and says whether the claim holds.
- `reportfile` saves the reports of both tools to a file.
- **Theories** can be checked with both tools.

Part IV: The toolkit

Turtle is one program, `turtle`, with a few commands. You can run a file, type code interactively, check, format, package, or read the documentation. This part covers each one, with real output. Run `turtle help` for the short list at any time.

```
$ turtle help
  turtle                          the interactive prompt (REPL)
  turtle script.turtle [args]     run a program
  turtle trace script.turtle      run it, showing each line as it runs (on stderr)
  turtle debug script.turtle      run it a line at a time: step, breakpoints, look at
      values
  turtle test [file | folder]     run the test_ functions in test_*.turtle files
  turtle fmt [file | folder]      lay out .turtle files the standard way (--check:
      only list them)
  turtle build script.turtle      make one program file that runs without Turtle (-o
      name)
  turtle doc [topic]              the standard library's documentation
  turtle lsp                      the language server, for editors (VS Code, Neovim
      ...)
  turtle version                  the version
  turtle help                     this
```

24

The REPL

What the REPL is. A **REPL** (read, evaluate, print loop) is an interactive prompt. It reads one line, runs it, prints the result and waits for the next line. It is the fastest way to try an idea and to learn what a statement does.

How Turtle does it. Run `turtle` with no file name to start it. Values and names you define stay available until you quit, and the REPL has its own commands, such as `help`.

Run `turtle` with no file, and you get an interactive prompt. Type a line, press Enter, and see the result. This is the fastest way to try an idea, check a value, or learn a function.

```
$ turtle
Turtle v1.0.0 – type help for help, quit to leave
>>> nums = list [3, 1, 2]
>>> nums at length * 2
6
>>> def double[x]
...     return x * 2
... def [end]
>>> double[21]
42
>>> 1 div 0
error (math): division by zero
>>> "a" + "b"
"ab"
>>> quit
```

The rules are simple, and they differ a little from a program file.

- **A lone expression shows its value.** `nums`, `1 + 2` and `double[21]` each print what they give. Text shows in quotes, so `"a" + "b"` prints `"ab"`.
- **A statement runs as in a file.** `show nums .` prints the list, the same as a program would.
- **A block waits for its end.** After a line that opens a block (`def`, `if ] ... [`, `[loop][...]`, `safe`), the prompt becomes `...`, and the next line is indented for you. The block runs when you type its end line, such as `def [end]`. An empty line runs an unfinished block so you can see what is wrong with it.
- **Errors do not end the session.** You see `error (kind): message`, and everything you defined is still there.
- **Ctrl-C** cancels the line you are typing, or stops code that is running, such as an endless loop, and returns you to `>>>`.

- **Imports carry on.** After `import random`, the `random` words work on later lines.

24.1 The REPL's own commands

A few words are commands to the REPL itself, typed alone on a line:

Command	What it does
`help`	a short guide to the REPL
`help linear`	a library: what it is for, and its functions
`help reshape`	one function, with an example
`help m`	what your variable `m` is, and what you can do with it
`help if`	a keyword, with where it is explained
`names`	your variables and functions
`load file.turtle`	run a file into this session
`save file.turtle`	write what you have run this session to a file
`colors`	pick a color scheme
`clear`	clear the screen (also Ctrl-L)
`quit`	leave (also Ctrl-D on an empty line)

Here is a short session that uses several of them:

```
>>> load session_part.turtle
>>> double[4]
8
>>> names
functions: double
>>> save session_saved.turtle
saved 2 entries to session_saved.turtle
>>> help if
if        (a keyword)
  Runs code when a condition holds.
  Example: if ] x > 1 [ ... if [end]
```

`save` writes the code you ran, so a session that turns out well can become a program.

24.2 Keys

Keys	What they do
Left / Right, Ctrl-B / Ctrl-F	move the cursor
Ctrl-Left / Ctrl-Right, Alt-B / Alt-F	move a word
Home / End, Ctrl-A / Ctrl-E	start or end of the line
Up / Down, Ctrl-P / Ctrl-N	earlier or later lines (history)
Backspace, Delete	delete a character
Ctrl-W, Alt-Backspace	delete the word before the cursor
Ctrl-K / Ctrl-U	delete to the end or start of the line
Tab	indent by four spaces
Ctrl-L	clear the screen
Ctrl-C	cancel the line, or stop running code
Ctrl-D	leave, on an empty line

24.3 Colors

The REPL colors code as you type, using Turtle's own lexer, so the colors always match the language. Words are grouped by what they do: keywords, imports, functions, data, `show`, text, numbers, and theories. Comments are shown in a muted italic.

`colors` opens a picker. Up and Down choose a scheme, and the program recolors as you move. Left and Right choose the shade: `auto`, `dark` or `light`. Enter keeps the choice, and Esc leaves the colors as they were.

You can also name a scheme without the picker:

```
>>> colors okabe-ito
colors: okabe-ito, dark (auto)
>>> colors classic light
colors: classic, light
>>> colors auto
colors: classic, dark (auto)
```

The choice is saved in `~/.config/turtle/repl`, and used the next time the REPL starts. Delete that file to go back to the default. Set the environment variable `NO_COLOR` to any value to turn colors off entirely.

The schemes match the VS Code extension, so pick the same one in both to keep them consistent. The REPL does not read VS Code's settings. Each keeps its own.

Best practice. Use the REPL to try a single idea, and move anything you want to keep into a file. A file is easier to test, share and change.

Common mistake. Forgetting that each line runs as soon as you press Enter. A variable assigned on one line is changed by the next line that assigns it, so the order of your lines matters.

Key points

- **The REPL** starts when `turtle` runs with no file, and reads one line at a time.
- **REPL commands** are words typed alone on a line, and run inside the REPL itself.
- **Keys** are listed in a table in section 24.2.
- **Colors** follow Turtle's own lexer, so they match the way the language is read.

25

Running, tracing and debugging

Three ways to watch a program. Running executes a program from start to finish. **Tracing** prints each line as it runs, with the values it produces. **Debugging** pauses at each line so you can look at the values and move forward one step at a time.

How Turtle does it. `turtle file.turtle` runs a file. `turtle trace` and `turtle debug` add the other two views. Anything after the file name is passed to the program as arguments.

A program can be run, traced, or debugged. **Running** executes a file from start to finish. **Tracing** prints each line as it runs, with the values it produces. **Debugging** pauses at each line, so you can look at the values and step forward. This chapter covers the `turtle` command and the three ways to run a file.

25.1 Running a program

```
turtle report.turtle
turtle report.turtle 2026 "north region"
```

Anything after the file name goes to the program, and `args[]` gives the list of them. `args` needs `import system`:

```
import system
show "args: ", args[] .
```

```
args: [ "2026", "north region" ]
```

A program can also read its input from a pipe, which lets you feed it text from another command. Piped input runs as a program, so a line that is only an expression does not print. That display belongs to the REPL.

25.2 `trace`: watch each line run

`turtle trace` runs the program as usual, and prints each line on standard error as it runs. A line that sets a value shows the value it got. Output of the program itself still goes to standard output, so the two can be separated:

This program adds up two numbers and doubles the total. Its trace follows:

```
import data
nums = list [5, 7]
total = 0
[loop][x in nums]
    total = total + x
[loop][end]
d = double[total]
show d .

def double[n]
    return n * 2
def [end]
```

```
line 1   import data
line 2   nums = list [5, 7]              nums = [ 5, 7 ]
line 3   total = 0                       total = 0
line 4   [loop][x in nums]
                                         pass 1: x = 5
line 5       total = total + x           total = 5
                                         pass 2: x = 7
line 5       total = total + x           total = 12
line 7   d = double[total]
line 11      return n * 2
line 7   ...                             d = 24
line 8   show d .
24
```

Each loop pass is shown with the values it used. When a line calls a function, the function's lines come next, and the result follows on a `...` line. Lines of an imported file are named with the file, such as `utils.turtle:4`.

A trace of a bug. This program is meant to average three scores, but the average stops with an error. Trace it to see what each line did:

```
scores = list [4, 8, 6]
total = 0
count = 0
[loop][s in scores]
    total = total + s
[loop][end]
average = total / count
show average .
```

```
line 1  scores = list [4, 8, 6]          scores = [ 4, 8, 6 ]
line 2  total = 0                        total = 0
line 3  count = 0                        count = 0
line 4  [loop][s in scores]
                                         pass 1: s = 4
line 5      total = total + s            total = 4
                                         pass 2: s = 8
line 5      total = total + s            total = 12
... 4 more lines
```

The trace shows `total` growing on each pass, but `count` is set once, to 0, and never changes. The last line is the division by zero. The trace has shown the cause: the loop never adds to `count`.

To keep the trace and the output apart:

```
turtle trace report.turtle 2> trace.txt
```

`trace` is the right tool when you want the order of events. For a problem that depends on a value, use `debug`.

25.3 debug: step through a program

`turtle debug` runs the program one line at a time. It stops before the first line and waits for a command. This session sums the numbers in a list, with a breakpoint on the line that adds each one:

```
import data
nums = list [5, 7]
total = 0
[loop][x in nums]
    total = total + x
[loop][end]
d = double[total]
show d .

def double[n]
    return n * 2
def [end]
```

```
turtle debug: stopped before the first line. Enter steps; h shows the commands.
→ line 1  import data
(debug) s
→ line 2  nums = list [5, 7]
(debug) s
→ line 3  total = 0
(debug) s
→ line 4  [loop][x in nums]
(debug) s
→ line 5      total = total + x
(debug) p total
0
(debug) b 5
will stop at 5
(debug) c
→ line 5      total = total + x
(debug) p total
5
(debug) v
here:
  x = 7
globals:
  nums = [ 5, 7 ]
  tablerows = 20
  total = 5
(debug) q
```

The commands are short:

Command	What it does
Enter or `s`	step: run this line, and stop at the next (going into functions)
`n`	next: run this line and any functions it calls
`o`	out: run to the end of the current function
`c`	continue to the next breakpoint, or to the end
`b 5`	stop at line 5 (`b utils.turtle:4` in an imported file); `b` alone lists them
`d 5`	remove that breakpoint
`p total`	show a value: `p total`, or `p nums at length`
`v`	the variables here, then the global ones
`w`	where you are: the functions being run, outermost first
`l`	the lines around this one
`q`	quit the program
`h`	the list of commands

The commands in use. Each example uses this program. A terminal session is shown with each command you type after the `(debug)` prompt.

```
def add_score[total, s]
    return total + s
def [end]
scores = list [4, 8, 6]
total = 0
count = 0
[loop][s in scores]
    total = add_score[total, s]
    count = count + 1
[loop][end]
average = total / count
show average .
```

`s` steps, into a function. Each `s` runs one line. At line 8 the next step goes into `add_score`, and `w` shows where you are:

```
turtle debug: stopped before the first line. Enter steps; h shows the commands.
→ line 4  scores = list [4, 8, 6]
(debug) s
→ line 5  total = 0
(debug) s
→ line 6  count = 0
(debug) s
→ line 7  [loop][s in scores]
(debug) s
→ line 8      total = add_score[total, s]
(debug) s
→ line 2      return total + s
(debug) w
the main code, line 8
└ add_score[], line 2
(debug) q
```

`n` **steps over.** `n` runs the whole call on line 8 and stops at the next line of the main program:

```
(debug) s
→ line 8      total = add_score[total, s]
(debug) n
→ line 9      count = count + 1
```

`v` **lists the variables, and** `o` **runs out of a function.** At line 2, `v` shows the variables of the function first, then the global ones. `o` runs to the end of the function, and `p` shows a value:

```
(debug) v
here:
  s = 4
  total = 0
globals:
  count = 0
  scores = [ 4, 8, 6 ]
(debug) o
→ line 9      count = count + 1
(debug) p total
4
```

`l` **lists the lines around the current one.** At the start, `l` shows lines 1 to 7 with the current line marked by an arrow:

```
(debug) l
     1  def add_score[total, s]
     2      return total + s
     3  def [end]
→    4  scores = list [4, 8, 6]
     5  total = 0
     6  count = 0
     7  [loop][s in scores]
```

`b`, `c` **and** `d` **manage breakpoints.** `b 9` stops the program at line 9, `b` lists the breakpoints, `c` runs to the next one, and `d 9` removes it. With the breakpoint gone, `c` runs to the end:

```
(debug) b 9
will stop at 9
(debug) b
breakpoints: 9
(debug) c
→ line 9      count = count + 1
(debug) d 9
won't stop at 9
(debug) c
6.0
```

A breakpoint stops the program only on a line that runs. If you set one on a blank line, a comment, the `def` line of a function, or a line of a theory such as `notation`, the debugger warns you and the breakpoint never stops. Set it on a statement instead:

```
(debug) b 5
warning: line 5 is a part of a theory, which describes the theory rather than running
    it, so it never runs and this breakpoint won't stop
will stop at 5
```

Turtle code at the prompt. Anything that is not a command runs as Turtle, at that point in the program. Assignment changes a variable, and `p` shows it straight away:

```
(debug) x = 5
(debug) p x
5
(debug) total = total + 100
(debug) p total
100
```

`q` **quits.** The program stops at once and the debugger exits with status 1, so a script can tell that the run was stopped.

Anything else you type is run as Turtle, at that point in the program. `total` shows its value, and `n = 100` changes `n` before the line runs. An error in what you type is shown, and the program carries on.

Asking for a value before it is set is a common mistake, and the debugger says so: `p total` on line 2 gives `error (name): undefined variable "total"`. Step forward first.

A session on the same bug. The same program, stepped with `debug`. Type `s` five times, which runs the first lines and stops in the loop. Then ask for `count` and `total`, and ask where the program is:

```
turtle debug: stopped before the first line. Enter steps; h shows the commands.
→ line 1  scores = list [4, 8, 6]
(debug) → line 2  total = 0
(debug) → line 3  count = 0
(debug) → line 4  [loop][s in scores]
(debug) → line 5      total = total + s
(debug) → line 5      total = total + s
(debug) 0
(debug) 4
(debug) the main code, line 5
```

The first answer, `0`, is `count`, and the second, `4`, is `total` after one pass. The debugger shows that `count` has not moved since line 3. The fix is to add `count = count + 1` inside the loop.

Best practice. Run a program normally first. Trace or debug only when the result is wrong, and look at the first line that gives a surprising value.

Common mistake. Expecting arguments without passing them. If no arguments are given, `args[]` is an empty list, so check its length before you use an argument.

Key points

- **Running** executes a file from start to finish.
- `turtle trace` prints each line as it runs.
- `turtle debug` runs one line at a time, starting before the first line.
- **Running a file** is as simple as `turtle report.turtle`.

26

Formatting, building and documenting

Three tasks after a program works. Formatting lays out source code in one consistent style, so every reader sees the same structure. **Building** packages a program into a single file that others can run. **Documenting** writes the description of each function, so a reader knows how to use it without reading its code.

How Turtle does it. `turtle fmt` formats `.turtle` files, `turtle build` makes a single program file, and `turtle doc` reads the comments above each `def`.

Three tasks follow once a program works: keeping its layout consistent, packaging it so that other people can run it, and documenting its functions. `turtle fmt` lays out source files in one standard form, `turtle build` makes a single program file, and `turtle doc` reads the comments that describe each function. This chapter covers each tool.

26.1 `fmt`: one layout for everyone

`turtle fmt` lays out `.turtle` files the standard way. Each block's lines sit four spaces in from the line that opens it, `else` and `handle` line up with their block, and lines have no spaces at their ends. Two blank lines in a row are the most it allows.

```
$ turtle fmt --check messy.turtle
messy.turtle needs formatting
$ turtle fmt messy.turtle
formatted messy.turtle
```

Before:

```
x   =    5
  def f[a]
return a
  def [end]
```

After:

```
x   =    5
def f[a]
    return a
def [end]
```

`fmt` changes only the space at the start and end of each line. The spaces between words are part of the code, because in Turtle a space can change what a line means: `nums get -1` is not `nums get - 1`. So `x = 5` keeps its gaps. Lines inside a backtick raw string or a `//* *//` comment are left exactly as they are.

Run `turtle fmt` with no file to format every `.turtle` file in the current folder and the folders below it. `--check` changes nothing and lists the files that need it, with exit code 1 if any do. Use it in a script or in continuous integration to keep a project tidy. In VS Code, Format Document does the same.

26.2 `build`: one program file

`turtle build` makes a single program file that runs without Turtle installed. The file holds a copy of Turtle, with your script and every `.turtle` file it imports packed inside:

```
$ turtle build report.turtle -o report_tool
built report_tool (report.turtle, shop.turtle, darwin/arm64)
$ ./report_tool 2026 "north region"
1080.0
args: [ "2026", "north region" ]
```

Some things to know before you share one.

- **It is for the system it is built on.** A build on a Mac makes a Mac program, and on Windows it makes a `.exe`. To give a tool to Windows users, build it on Windows.
- **Data files are not packed.** CSV files and settings are read from the folder the program runs in. `scriptfolder[]` gives the program's own folder, when a file has to be found next to the program.
- **Libraries are built in.** `import json`, `import sql` and the rest are part of every program. Only your own `.turtle` files are packed.
- **It is large.** A program is about the size of Turtle, since the whole interpreter is inside it. The build used for this book made a 23 MB file.
- **Mistakes are found first.** A script that does not parse, or that imports a file that is missing, is not built.
- **Signing.** A program built on your own machine runs there. A copy downloaded from elsewhere may be blocked by the system. On a Mac you can clear that with `xattr -d com.apple.quarantine report`. On Windows, choose "More info", then "Run anyway".

26.3 `doc`: the documentation, in the terminal

`turtle doc` shows what each library function takes and gives back:

```
turtle doc                  every library and its functions
turtle doc sql              one library, every function in full
turtle doc sql_update       one function
turtle doc shop.turtle      the functions of your own file
```

Document your own functions with comments directly above the `def`, or as the first lines in the body. Say what each parameter is, and what the function gives back:

```
// shop.turtle: helpers for the bookshop.

// add_tax adds rate percent to an amount, rounded to the cent.
// Gives back the new amount in cents.
def add_tax[cents, rate]
    return cents + cents * rate / 100
def [end]
```

```
$ turtle doc shop.turtle
shop.turtle

shop.turtle: helpers for the bookshop.

add_tax[cents, rate]
  add_tax adds rate percent to an amount, rounded to the cent.
  Gives back the new amount in cents.
```

A function without a comment is listed with no description. Editors show the same description when you hover over the name, which makes comments do double duty.

Best practice. Format a file before you commit it, and write one comment above each function that says what it takes and what it gives back.

Common mistake. Documenting a function only in your head. A `def` with no comment is listed with no description, so the reader has nothing to go on.

Key points

- `turtle fmt` lays out `.turtle` files in one standard way.
- `turtle build` makes a single program file that runs without Turtle installed.
- `turtle doc` shows what each library function takes and gives back.
- **Three tasks** follow once a program works: layout, packaging and documentation.

27

Editors and the language server

What an editor does for you. An **editor** is where you write code. A **language server** is a helper that runs alongside the editor and gives suggestions, shows errors as you type, and jumps to the definition of a name.

How Turtle does it. `turtle lsp` starts Turtle's language server. An editor that supports the standard protocol uses it, so the same help works in any of those editors.

`turtle lsp` starts Turtle's language server. An editor that speaks the Language Protocol uses it to give you help as you type. It is built into `turtle`, and needs no extra install.

With it, an editor shows:

- **Errors as you type.** The same messages Turtle gives when it runs the file, placed under the spot that caused them. Imports of files that are missing show as warnings.
- **Colors** that match the REPL, in the same twelve schemes.
- **Completion** for keywords, your functions, assembled types, variables, library functions, methods after `at`, and the libraries and files after `import`.
- **Hover** to read a function's description, a library function's `turtle doc` text, or what a keyword means.
- **Go to definition** for your functions and types, including those in imported files.
- **An outline** of the functions, types and top-level variables in a file.
- **Format Document**, which runs `turtle fmt`'s layout.

27.1 VS Code

The VS Code extension adds the language, colors, comment toggling (Ctrl-/), bracket matching and indenting, and a run button (Ctrl-F5) that runs the open file. It starts `turtle lsp` for you.

Its color scheme is set with **Turtle: Color Scheme** in the settings. Pick the same scheme in the REPL with `colors` for a consistent look.

27.2 Other editors

Any editor with language-server support can use `turtle lsp`. Two examples:

```
-- Neovim 0.11 and later
vim.filetype.add({ extension = { turtle = "turtle", trt = "turtle" } })
vim.lsp.config("turtle", { cmd = { "turtle", "lsp" }, filetypes = { "turtle" },
    root_markers = { ".git" } })
vim.lsp.enable("turtle")
```

```
# Helix, in ~/.config/helix/languages.toml
[language-server.turtle]
command = "turtle"
args = ["lsp"]

[[language]]
name = "turtle"
scope = "source.turtle"
file-types = ["turtle", "trt"]
comment-token = "//"
language-servers = ["turtle"]
```

Files end in `.turtle`. The older `.trt` ending still works everywhere `.turtle` does.

Best practice. Let the editor show errors as you type, and fix the first error in a file before the others, because later errors are often caused by it.

Common mistake. Setting up an editor without a way to find the `turtle` command. The language server then cannot start, so check that `turtle` runs in a terminal first.

Key points

- `turtle lsp` starts Turtle's language server.
- **VS Code** gets the language, colors, and comment toggling with Ctrl-/ from the extension.
- **Other editors** can use `turtle lsp` if they support language servers.

Thank you

Thank you for reading. I hope you have learned a lot, and I hope you enjoy writing programs in Turtle. Programming takes practice: write small programs, read the messages they produce, and change one thing at a time. I hope this book helps you on the way to becoming a software engineer.

Resources

- **Website:** turtlelang.dev: the docs, the full reference, the library pages and the downloads.
- **Book examples:** turtlelang.dev/docs/book-examples: every example in this book, with its output.
- **VS Code extension:** [Turtle Programming Language on the Marketplace](https://marketplace.visualstudio.com/items?itemName=turtle-lang.turtle-lang).
- **Source code:** github.com/NigeleM/Turtle.
- **Downloads:** the signed installers for Mac, Windows and Linux, on the [release page](https://github.com/NigeleM/Turtle/releases/tag/v1.0.0) and the website.

Appendices

A. Keywords

Reserved words

Word	What it does	Example	Chapter
`add`	adds an item to a list or set, or a key to a map	`add 4 to nums .`	8
`append`	adds lines to the end of a file	`[append] log.txt` … `[end]`	19
`assemble`	names a kind of value with fields	`assemble Order [item, qty]`	9
`at`	calls a method on the value before it	`nums at get[0]`, `name at upper`	8
`break`	leaves a loop	`break`	6
`change`	converts a value to another type	`n = change "42" to integer`	3
`continue`	goes on to a loop's next pass	`continue`	6
`def`	defines a function	`def total[a, b]` … `def [end]`	7
`delete`	removes a key from a map	`delete "Bo" from ages .`	8
`directory`	lists a folder's entries	`[directory] data to names [end]`	19
`div`	whole-number division	`7 div 2` is `3`	3
`else`	the other branches of an `if`	`else if ] x > 2 [`, `else ]`	5
`end`	closes a block	`def [end]`, `if [end]`, `[loop][end]`	5
`fail`	raises your own error	`fail "no stock"`	13
`false`	the truth value false	`done = false`	3
`from`	in sentences: where from	`remove 1 from nums .`	8
`give`	makes a function without a name	`x give x + 1`, `[a, b] give a + b`	7
`handle`	the part of a `safe` block that runs on an error	`handle [file] e .`	13
`if`	runs code when a condition holds	`if ] x > 1 [` … `if [end]`	5
`import`	brings in a library or your own file	`import sql`, `import lib/utils`, `import time [now]`	14
`in`	the items a loop goes through	`[loop][x in nums]`	6
`insert`	puts an item into a list at a position	`insert 9 to nums at 0 .`	8
`is`	assigns, in sentence form	`r is nums at get 0 .`	3

Word	What it does	Example	Chapter
`length`	how many items or characters	`length of nums`	8
`list`	makes a list	`list [1, 2, 3]`	8
`loop`	repeats code	`[loop][x in nums]`, `[loop][i = 0; i < 3; i++]`	6
`map`	makes a map of keys to values	`map ["a": 1]`	8
`max`	the largest item	`max of nums`, `max[nums]`, `max[a, b]`	8
`min`	the smallest item	`min of nums`, `min[nums]`, `min[a, b]`	8
`none`	no value	`x = none`	3
`of`	a field or key of a value	`qty of order`, `length of nums`	9
`read`	reads a file's lines	`[read] notes.txt to lines [end]`	19
`remove`	removes an item from a list or set	`remove 4 from nums .`	8
`return`	gives a function's answer back	`return total`	7
`reverse`	reverses a list in place	`reverse nums .`	8
`safe`	runs code that may fail, with `handle` for the errors	`safe` ... `handle [] e .` ... `safe [end]`	13
`scroll`	steps a value goes through, in order	`x is scroll 3 into add1, double .`	11
`set`	makes a set, with no repeats	`set [1, 2]`	8
`show`	prints values	`show "total: ", n .`	4
`sort`	sorts a list in place (`import sort` for more)	`sort nums .`	8
`sys`	runs a shell command	`sys ls -la`	19
`to`	in sentences: where to, or what into	`add 4 to nums .`, `change x to integer`	8
`true`	the truth value true	`done = true`	3
`warn`	prints to the error stream (`import system`)	`warn "careful" .`	19
`write`	writes a file's lines	`[write] notes.txt` ... `[end]`	19

Words an import turns on

These mean something only in a file with the import. Anywhere else they are ordinary names, and in such a file they cannot name a function.

Word	Import	What it does	Example	Chapter
`random`	`import random`	a random value of any shape	`random list of 5 integers from 0 to 9`	18
`check`	`import test`	one fact a test checks	`check total[o] == 45 .`	22
`verify`	`import test`	a rule over every item	`verify evens[nums] each x give x % 2 == 0 .`	22
`validate`	`import test`	a rule over random inputs	`validate halve[n] with n as integer that n >= 0 .`	22
`log`	`import log`	a log line, with a level	`log warn "disk at ", pct, "%" .`	21
`matrix`	`import linear`	makes a matrix	`m = matrix [1, 2; 3, 4]`	18

Words special only in one place

These are ordinary names everywhere else, so a variable can still be called `put` or `here`.

Word	Where	Example	Chapter
`put`	first on a line: replace an item	`put 9 to nums at 2 .`	8
`into`	in a scroll: the steps follow	`x is scroll 3 into add1, double .`	11
`here`	in a scroll step: the value so far	`here at get[0]`	11
`diagnose`	a line of its own, or `diagnose[...]`: look inside	`diagnose` … `diagnose [end]`	23
`hypothesis`	`hypothesis[...]`: try a claim, true or false	`hypothesis[prices each p give p > 0]`	23
`reportfile`	a variable: also write diagnose and hypothesis reports to a file	`reportfile = "checks.txt"`	23
`type`	after a value: whether it is that kind, true or false	`if ] n type integer [`	5
`integer`, `float`, `string`, `ascii`, `char`, `hex`, `keys`, `values`	after `change ... to`	`change x to float`	3
`each`, `any`, `not`, `exactly`, `least`, `most`, `pair`, `that`, `with`, `as`, `matches`, `fails`, `close`, `within`	in `check`, `verify` and `validate` sentences	`verify xs at least 2 x give x > 0 .`	22
`debug`, `info`, `error`	right after `log`	`log debug "x is ", x .`	21
`theory`, `abstract`, `notation`, `definition`, `theorem`, `proof`	a theory, and its sections	`theory tally` … `theory [end]`	12

The same words are printed in Turtle by `help["keywords"]`, and section 4.5 shows the other `help` forms.

B. Index

- **Arguments**, calls with `[ ]`: chapters 7, 8
- **Assembly types** (`assemble`): chapter 9
- **Brackets**, the reversed `] ... [`: chapter 5
- **change** (converting types): chapter 3
- **Closures** and anonymous functions (`give`): chapter 7
- **Collections** (lists, sets, maps): chapter 8
- **Commands** (`trace`, `debug`, `test`, `fmt`, `build`, `doc`): chapters 22, 24, 25
- **Databases** (`sql`): chapter 20
- **Dates** (`time`): chapter 21
- **debug** (stepping through a program): chapter 25
- **diagnose**: chapter 23
- **Editors** and `turtle lsp`: chapter 27
- **Errors**, `safe`, `handle`, `fail`: chapter 13
- **Files** (`[read]`, `[write]`, `[append]`): chapter 19
- **fmt** (layout): chapter 26
- **Functions** (`def`): chapter 7
- **Generators** in theories: chapter 12
- **hypothesis**: chapter 23
- **Imports** and modules: chapter 14
- **Libraries**: chapters 16 to 20
- **Logging** (`log`): chapter 21
- **Loops** (`[loop]`): chapter 6
- **Matrices** (`linear`): chapter 18
- **Methods** (`at`): chapter 8
- **Nested structures** (maps, lists, sets, assemblies): chapter 10
- **Operators** and the order of operations: chapter 3
- **REPL**: chapter 24
- **Show** and input (`?`): chapter 4
- **Tests** (`check`, `verify`, `validate`): chapter 22
- **Theories**: chapter 12
- **trace** (watching a run): chapter 25
- **Web** (`http`, `json`, `server`): chapter 20

Glossary

This glossary defines the terms used in the book. A term that is first explained in a chapter is shown in green italics there. Each entry gives the formal meaning from computing and mathematics first, then what the word means in Turtle. The chapter where a term is first explained is listed at the end of its entry.

abstraction. Hiding detail behind a simpler name, so that a piece of work can be used without knowing how it is done. In Turtle, theories, scrolls and functions let a long piece of code be used by a short name. First explained in chapter 1.

anonymous function. A function with no name, written where it is used. In Turtle, `x give x * 2` is a function of one argument, written in one line and used as a value. First explained in chapter 7.

argument. In computing, a value given to a function when it is called. In Turtle, the value written in the brackets of a call, such as `"Ann"` in `greet["Ann"]`. The function's parameter receives it while the function runs. First explained in chapter 7.

assembly. A named record with a fixed set of fields, where each value has the same fields in the same order. In Turtle, `assemble Order [item, qty, price]` names the shape, and `Order["pen", 3, 1.5]` makes one value of it. First explained in chapter 9.

assignment. Storing a value under a name, so the name refers to that value. In Turtle, `name = value` makes a variable, or gives an existing variable a new value. First explained in chapter 3.

branch. One of the paths a decision can take. In Turtle, each `if`, `else if` and `else` block is a branch. First explained in chapter 5.

break. A statement that leaves the innermost loop at once. In Turtle, `break` inside a loop stops it, and the program goes on after the loop's `[end]`. First explained in chapter 6.

build. Turning source code into a finished program that others can run. In Turtle, `turtle build` makes a single program file that runs without Turtle installed. First explained in chapter 26.

call. Running a function by writing its name and its arguments. In Turtle, a call is written with brackets, as in `greet["Ann"]`. First explained in chapter 7.

check. A statement from `import test` that stops the program with a report when its condition is false. It works in an ordinary program, and inside a `test_` file `turtle test` runs it with the others. First explained in chapter 22.

closure. A function that keeps the variables that were visible where it was made, even after that place has finished. In Turtle, `make_adder[5]` returns a function that keeps the 5 (section 7.6). First explained in chapter 7.

collection. A value that holds many other values. In Turtle, the three collections are lists, sets and maps. First explained in chapter 8.

command line. The text typed after a program's name when it is started, such as `turtle report.turtle 2026`. In Turtle, `args` gives that text as a list (section 25.1). First explained in chapter 19.

comment. Text that a program ignores and that explains it to its reader. In Turtle, a `//` starts a comment that runs to the end of the line, and `//*` ... `*//` covers several lines. First explained in chapter 2.

condition. An expression that is either true or false, used to choose a branch or to stop a loop. In Turtle, conditions sit in reversed brackets after `if`, as in `if ] x > 10 [`. First explained in chapter 5.

continue. A statement that skips the rest of the current pass of a loop and goes on to the next pass. In Turtle, `continue` is used inside a loop. First explained in chapter 6.

CSV. Comma-separated values: a plain text file in which each line is a row and commas separate the columns. In Turtle, `table_read` reads a CSV file into a list of maps, one per row (chapter 17). First explained in chapter 17.

database. A store of data organized as tables of rows, usually managed by software. In Turtle, the `sql` library works with SQLite files and with PostgreSQL and MySQL servers (chapter 20). First explained in chapter 20.

datetime. A date with a time of day, stored so that dates can be compared and added to. In Turtle, the `time` library makes and changes dates (chapter 21). First explained in chapter 21.

debug. Finding and fixing faults by running a program and looking at its values as it goes. In Turtle, `turtle debug` pauses at each line so you can look at values and step forward. First explained in chapter 25.

diagnose. Explaining what a piece of code did while it ran. In Turtle, `diagnose` shows each step's values and results, and the program keeps running. First explained in chapter 23.

documentation. Text that explains how to use a piece of code. In Turtle, the comments directly above a `def` describe the function, and `turtle doc` prints them. First explained in chapter 26.

editor. A program for writing and changing source code, which often shows errors and helps with names as you type. Turtle works with editors through its language server (chapter 27). First explained in chapter 27.

environment. The settings that the operating system gives a running program, such as a folder or a user name. In Turtle, the `system` and `config` libraries read them (chapter 19). First explained in chapter 19.

error. A problem that stops a program, such as dividing by zero. In Turtle, an error stops the program by default and the message says where it happened. First explained in chapter 13.

expression. A combination of values and operators that gives one value. In Turtle, `2 + 3 * x` is an expression, and each part of a statement can be an expression. First explained in chapter 5.

field. A named part of a record. In Turtle, the fields of an assembly are written after `of`, as in `qty of o`. First explained in chapter 9.

float. A number with a fraction part, stored in binary, so some results carry small rounding errors. In Turtle, a number with a decimal point, such as `2.5`, is a float. The `/` operator always gives a float. First explained in chapter 3.

format. Arranging source code in one consistent layout. In Turtle, `turtle fmt` lays out `.turtle` files in one standard way. First explained in chapter 26.

function. A named block of statements that can be run again by writing its name. It may take inputs and may return a result. In Turtle, a function starts with `def name[...]` and ends with `def [end]`. First explained in chapter 7.

function value. A function that can be stored in a variable, passed to another function, or returned, just like any other value. In Turtle, a function's name without brackets is the function itself. First explained in chapter 7.

handle. Code that catches an error, so the program can carry on. In Turtle, `handle` follows a `safe` block and names the kinds of error it catches. First explained in chapter 13.

hash. A short fixed-length value calculated from data, so that the same data always gives the same value and different data almost never does. In Turtle, the `crypt` library makes hashes (chapter 21). First explained in chapter 21.

HTTP. The protocol that web browsers and programs use to request and send data over the internet. In Turtle, the `http` library makes requests (chapter 20). First explained in chapter 20.

hypothesis. A claim that is tested against data. In Turtle, `hypothesis[...]` says whether a claim holds for every item, and which item breaks it first (chapter 23). First explained in chapter 23.

import. Bringing another file's functions, theories or library into a program. In Turtle, `import data` brings in a library, and `import shop` brings in `shop.turtle`. First explained in chapter 16.

index. A position in a list or a string, counting from 0 in Turtle. In Turtle, `nums at get[0]` reads the first item. First explained in chapter 8.

input. Data that a program receives from a user or from another source while it runs. In Turtle, `?` asks the user for a line of input. First explained in chapter 4.

integer. A whole number, positive, negative or zero. In Turtle, whole numbers have no size limit. First explained in chapter 3.

interpreter. A program that reads source code and carries out its statements, step by step. In Turtle, the `turtle` command is the interpreter. First explained in chapter 1.

iteration. Repeating a set of steps, once for each item or until a condition changes. In Turtle, a loop that goes through each item of a collection is an iteration. First explained in chapter 6.

JSON. A text format for structured data, made of maps, lists, text, numbers, booleans and null. In Turtle, the `json` library reads and writes it (chapter 20). First explained in chapter 20.

key. The name that identifies one entry in a map. In Turtle, `"Alice"` is the key in `map ["Alice": 30]`. First explained in chapter 6.

language server. A program that gives an editor help as you write code, such as completions and error messages. In Turtle, `turtle lsp` starts it for editors. First explained in chapter 27.

library. A collection of ready-made functions that comes with a language. In Turtle, the standard library is used by writing `import` with the library's name. First explained in chapter 16.

list. An ordered collection of values that can repeat. In Turtle, `list [4, 4, 9]` keeps the order and both 4s. First explained in chapter 8.

literal. In computing, a value written directly in a program's source code, such as `5` or `"hi"`. Its meaning is fixed by the text. In Turtle, a literal is a number, text, `true`, `false` or `none` written in the code, as opposed to a variable, which names a value that can change (section 3.2). First explained in chapter 3.

local variable. A variable that exists only inside one function or block. In Turtle, a variable assigned inside a function is local to that function (section 7.4). First explained in chapter 7.

log. A record of what a program did while it ran, written line by line, usually with a time and a level. In Turtle, the `log` library writes them (chapter 21). First explained in chapter 21.

log level. The importance of a log message, such as debug, info, warning or error, which lets a program show only the messages it needs. In Turtle, `loglevel` sets the lowest level shown (chapter 21). First explained in chapter 21.

loop. A block of statements that runs more than once, either while a condition is true or once for each item in a collection. In Turtle, `[loop]` begins a loop and `[loop][end]` ends it. First explained in chapter 6.

map. A collection of pairs, where each key has one value. In Turtle, `map ["Ann": 30]` pairs the key `"Ann"` with `30`. First explained in chapter 8.

matrix. In mathematics, a rectangular grid of numbers arranged in rows and columns. In Turtle, `matrix [ ... ]` writes one, one row per line, and `linear` library functions such as `solve` work on it. First explained in chapter 18.

method. A function that belongs to a kind of value and is called on that value. In Turtle, a method is written after `at`, as in `nums at add[7]`. First explained in chapter 8.

module. A file that holds functions and variables and is brought into a program with `import`. In Turtle, a module is any `.turtle` file. First explained in chapter 14.

name. A word that stands for a value or a function. In Turtle, a name is written with letters, digits and underscores, and it must start with a letter. First explained in chapter 3.

nested structure. A data structure that holds other data structures. In Turtle, a map whose values are lists, or a list of assemblies (chapter 10). First explained in chapter 10.

none. The value that means no value. In Turtle, a function with no `return` gives back `none`, and `none` cannot be used in arithmetic. First explained in chapter 3.

operator. A symbol that combines values into a new value. In Turtle, `+`, `-`, `*`, `/` and the comparison symbols are operators (section 3.8). First explained in chapter 3.

output. Data that a program produces for the user. In Turtle, `show` writes output to the screen. First explained in chapter 4.

parameter. A name in a function's definition that stands for an input the function expects. In Turtle, the parameters are the names in the brackets of `def`, such as `name` in `def greet[name]`. First explained in chapter 7.

path. The text that names a file or folder, such as `reports/2026.csv`. Relative paths are read from the folder the program is started in (chapter 19). First explained in chapter 19.

pattern. A description of text to look for, written in a small language of its own. In Turtle, the `pattern` library searches and changes text with patterns (chapter 18). First explained in chapter 18.

phrase. The written form of a theory, which says how it is used in a program. In Turtle, the phrase of `tally` is `tally b in a`. First explained in chapter 12.

pipeline. A chain of steps in which each step's output is the next step's input. In Turtle, a scroll is a pipeline (chapter 11). First explained in chapter 11.

precedence. The rules that decide which operator is applied first when a line has several. In Turtle, section 3.9 gives the order. First explained in chapter 3.

private function. A function that only the file that defines it can use. In Turtle, a function whose name starts with `~` is private (section 7.12). First explained in chapter 7.

program. A text file of statements that a computer runs. In Turtle, a program is a `.turtle` file that the `turtle` command runs from top to bottom. First explained in chapter 2.

programming language. A set of words and rules for writing instructions that a computer can carry out, as text people can read and check. In Turtle, the rules are described throughout this book. First explained in chapter 1.

promise. A condition that every result of a theory must meet. In Turtle, a `theorem` line in a theory is a promise (chapter 12). First explained in chapter 12.

random seed. The starting number for a random number generator. Using the same seed gives the same sequence of random values, so a run can be repeated. In Turtle, `import random [seed = 7]` sets it (chapter 18). First explained in chapter 18.

record. A group of values, each with its own name, that belong together as one value. In Turtle, a record is an assembly (chapter 9). First explained in chapter 9.

recursion. A function that calls itself to solve a smaller version of the same problem, with a case that stops the calls. In Turtle, see section 7.9. First explained in chapter 7.

REPL. Read-eval-print loop: an interactive prompt that reads one line, runs it, prints the result, and waits for the next. In Turtle, run `turtle` with no file name to start it, and type a statement to see its result. First explained in chapter 24.

return value. The result a function gives back to its caller. In Turtle, `return` gives a value back. First explained in chapter 7.

route. A rule that sends a request to a function according to its method and path. In Turtle, the `server` library takes a map of routes (chapter 20). First explained in chapter 20.

scope. The part of a program where a name can be used. In Turtle, a name made inside a function exists only while the function runs. First explained in chapter 7.

scroll. A value that holds a sequence of steps, applied in order, with each step receiving the result of the one before. In Turtle, a scroll is written with `scroll` and applied with `into` (chapter 11). First explained in chapter 7.

sentence form. Writing a call so that it reads like a sentence, with the values placed among the words. In Turtle, `show tally 1 in nums .` is in sentence form (chapter 15). First explained in chapter 15.

server. A program that waits for requests and sends back answers. In Turtle, the `server` library builds one with `serve` (chapter 20). First explained in chapter 20.

set. A collection of values with no repeats and no order that means anything. In Turtle, `set [4, 4, 9]` holds 4 and 9 once each. First explained in chapter 8.

source code. The text a programmer writes, which a program such as an interpreter or compiler reads. In Turtle, source code is kept in `.turtle` files. First explained in chapter 1.

SQL. A language for asking a database to store, change and read rows. In Turtle, the `sql` library runs SQL statements (chapter 20). First explained in chapter 20.

statement. An instruction that does something when it runs, such as showing a value or changing a variable. In Turtle, a statement ends with a period. First explained in chapter 1.

table. Rows of data with named columns. In Turtle, `table[x]` shows rows as an aligned table, and `table_read` reads a CSV file as rows (chapter 17). First explained in chapter 17.

test. A check that a program behaves as expected, run automatically. In Turtle, a test is written in a file whose name starts with `test_`, and `turtle test` runs it (chapter 22). First explained in chapter 22.

text. A sequence of characters, such as letters and digits, written in quotes. In Turtle, `"hello"` is text. First explained in chapter 3.

theory. A rule that teaches Turtle a new phrase, with its meaning, its promises and examples that check it. In Turtle, a theory is written with `theory` and `theory [end]`. First explained in chapter

1.

trace. A record of each step a program takes as it runs. In Turtle, `turtle trace` prints each line and the values it produces. First explained in chapter 25.

truth value. A value that is either true or false. In Turtle, `true` and `false` are truth values. First explained in chapter 3.

type. A classification of values that decides which operations can be used on them. In Turtle, `typeof[x]` gives the type of `x` (section 3.1). First explained in chapter 3.

value. A piece of data a program works with, such as a number, text, a list or `none`. In Turtle, every expression gives a value. First explained in chapter 3.

variable. A name that holds a value, which can be changed later. In Turtle, `x = 5` makes `x` a variable. First explained in chapter 3.

vector. In mathematics and computing, an ordered list of numbers, whose length is how many numbers it holds. In Turtle, a list of numbers can be used as a vector, such as the right-hand side of `a solve list [3, 5]`. First explained in chapter 18.

void function. A function that gives no value back. In Turtle, a function with no `return` gives back `none`. First explained in chapter 7.

warning. A message about a likely problem that does not stop the program. In Turtle, `warn` from `import system` writes one, and Turtle also prints warnings of its own (section 13.7). First explained in chapter 4.