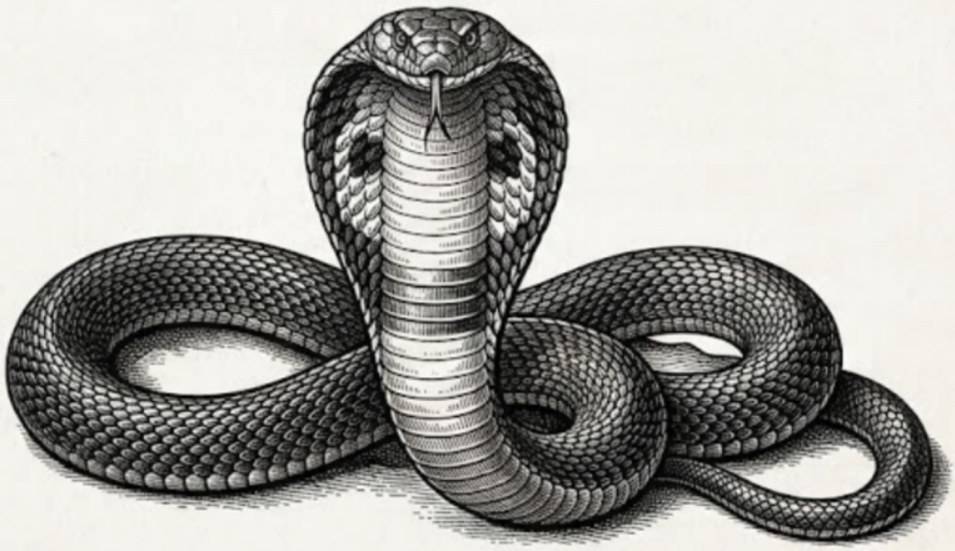
Practical, Fast Software Development

Cobra

The Programming Language

Simple syntax, true parallelism, and two execution engines



Baris AKIN © 2026

Contents

Preface

Programming languages are often pulled toward one of two extremes: either they are easy to write but slow, or they are fast but their syntax wears out your eyes. **Cobra** was designed to reject this dilemma. It offers a clean syntax that does not force significant indentation and closes blocks with `end`; in return, behind the scenes it delivers serious performance through two separate execution engines — a tree-walking interpreter and a bytecode virtual machine. What is more, unlike many languages, it carries no **GIL** (Global Interpreter Lock): tasks run genuinely in parallel across multiple CPU cores.

This book was written to teach Cobra from scratch, at a production level.

0.1 Who is this book for?

The reader of this book is a developer who knows at least one programming language at a basic level. We assume you know what concepts such as variables, loops, and functions are; but we assume you know nothing specific to Cobra. Not being familiar with the language is no problem — we build everything from the ground up.

0.2 How is this book organized?

The book consists of two halves. The first eleven chapters teach you how to **use** the language: starting from variables and types and progressing through collections, functions, object-oriented programming, error handling, modules, concurrency, and the standard library. The final chapter pulls back the curtain and enters the world of those who **build** the language: it covers the lexer that turns source into tokens, the parser that turns tokens into a syntax tree, and the two engines that run that tree.

The chapters build on one another; reading from start to finish is recommended. Still, if you want to quickly look up a particular topic, each chapter is also understandable on its own.

0.3 Conventions used in this book

Throughout the text we use the margin notes in the O'Reilly tradition:

> **Note** A detail that rounds out the topic and is worth keeping in mind.

> **Tip** A practical suggestion that will make your work easier.

> **Warning** A critical caution that warns you against a trap or a common mistake.

Code examples are given in monospaced blocks. In command-line examples, `$` is the prompt sign, and you do not type it:

```
$ cobra merhaba.cobra
```

Cobra source code is shown in the language's own syntax:

```
let mesaj = "Merhaba, Cobra"
print(mesaj)
```

0.4 Running the code

All examples can be run in two forms:

```
$ cobra dosya.cobra          # tree-walking interpreter (default)
$ cobra --vm dosya.cobra     # bytecode virtual machine (faster)
$ cobra                      # interactive REPL
```

The version this book is based on is **Cobra 0.10.0**. Both engines are designed to produce exactly the same output for every example; if you see a difference, that is a bug.

0.5 Acknowledgments

Cobra is dedicated to everyone who believes that a simple language can also be powerful. Now let us write our first line.

— Baris AKIN

Chapter 1

Introduction to Cobra

Every programming language makes a bet. Some believe speed comes before everything; others, safety; others still, the power of abstraction. Cobra's bet is simplicity: that a scripting language can hold on, at the same time, to the side that makes code easy to read and write and the side that makes it powerful enough to put to real work. In this chapter we will look at what Cobra is, which design decisions set it apart from other languages, and a few fundamental concepts you need to know before you even write your first line. By the end you will have run your own "Hello, Cobra" program on two separate engines, and even taken a glance at the language's inner world.

The rest of the book is built on the foundation laid in this chapter. So do not settle for merely reading the examples here; open a terminal and run all of them yourself. The fastest way to learn Cobra is to talk to it.

1.1 What Is Cobra?

Cobra is a scripting language with a simple, easy-to-read syntax. The syntax is familiar but is not an exact copy of any language: it closes blocks not with indentation but explicitly with the `end` keyword; it expresses logical operations not with symbols but with the words `and`, `or`, `not`; and it offers many things you would expect from a modern language (closures, structures, error handling, concurrency, a rich standard library) right out of the box.

Cobra has two features that make it truly interesting. The first is that the language runs on **two separate execution engines**. The second is that its concurrency is genuinely parallel; that is, it **contains no GIL**. We will look at both of these in detail shortly.

First we need to understand the design philosophy, because this philosophy lies behind every decision you will make while working with Cobra.

1.2 Design Philosophy

Cobra is designed around a few clear principles. Grasping these from the start lets you understand why the language does some things and deliberately does not do others.

1.2.1 Simple Syntax

Cobra's primary aim is simple syntax. Rather than overflowing with new operators and symbols, the language rests on a small number of well-chosen constructs. The fact that the logical operators are words rather than symbols is a small but meaningful example of this:

```
let x = 7
if x > 0 and x < 10
    print("single-digit positive")   # single-digit positive
end
```

Here you write `and` instead of `&&`, `or` instead of `||`, `not` instead of `!`. The code reads like an English sentence. This shows that simplicity is not just a slogan in Cobra but a concrete design choice.

> **Note:** Cobra has no `else if`; instead, the single word `elif` is used. Boolean values are written in lowercase: `true` and `false`. The "nothing" value is `null` — not `None`, `nil`, or `undefined`.

1.2.2 Blocks Closed with `end`, Not Indentation

Many modern languages mark blocks either with curly braces (`{ }`) or with significant indentation. Cobra chooses a third way: every block begins with the keyword that opens it (`if`, `while`, `for`, `def`, `struct`...) and closes with `end`.

```
def greet(name)
    if name == ""
        print("Hello, stranger")
    else
        print("Hello, " + name)
    end
end

greet("Ada")   # Hello, Ada
```

This has a practical consequence: **indentation is purely cosmetic.** Cobra does not care how many spaces you indent your code with; it tells where a block ends not from whitespace but from the `end` keyword. Even so, you are expected to indent your code properly for readability; just know that you are not *required* to do so.

> **Tip:** If you are coming from languages where indentation carries meaning, you will never experience the "my code does not run because of wrong indentation" problem in Cobra. Instead of hunting for block boundaries with your eyes, follow the `end` keyword.

1.2.3 One Semantics, Two Engines

Here is Cobra's most unusual aspect. The same language can be run in two different ways:

1. **Tree-walking interpreter:** It turns the source code into an abstract syntax tree (AST) and runs the program directly by walking over this tree. This is the default engine.
2. **Bytecode VM (bytecode virtual machine):** It first compiles the source code into bytecode, then runs this bytecode on a stack-based virtual machine. This engine is roughly **3 times faster** than the tree-walker.

These two engines share the same semantics. That is, whichever engine a Cobra program runs on, it produces **exactly the same output**. This is not just a promise; the language's test suite continually verifies this equivalence. In practice this means you can use the flexible tree-walker during development, then run the same program on the fast VM by adding a single flag — without worrying that the behavior will change.

> **Note:** There is one visible difference between the two engines: the VM catches undefined variables *at compile time* (before the program reaches that line); the tree-walker catches them when that line runs. Runtime error messages, including line numbers, are the same on both engines.

1.2.4 True Parallelism Without a GIL

Many scripting languages offer concurrency behind a "Global Interpreter Lock" (GIL). Because the GIL allows only a single piece of code to run at a time, CPU-bound work cannot truly benefit from multiple cores.

Cobra has **no GIL.** An `async def` call instantly returns a task, and these tasks run genuinely in parallel across CPU cores. This means CPU-bound work actually speeds up.

```
import "time"

async def do_work(n)
    time.sleep(0.05)
    return n * n
end

let results = await [do_work(1), do_work(2), do_work(3)]
print(results)   # [1, 4, 9]
```

This power brings a responsibility: because tasks truly run in parallel, you must explicitly synchronize shared mutable state (with tools like `Mutex`, `Channel`). We will cover concurrency in depth in later chapters of the book; for now the only thing you need to know is that Cobra makes no compromises on this matter.

1.2.5 Little but Enough

Cobra deliberately does not include many features you are accustomed to in other languages. f-strings, list comprehensions, tuples, multiple assignment, `lambda`, `match`/`switch`, sets, and operator overloading are a few of these. These omissions are not a flaw but a consequence of the simplicity philosophy: every feature brings a complexity cost to the language, and Cobra pays this cost only when it is truly necessary.

> **Warning:** If, out of habit from another language, you write `a, b = 1, 2` (multiple assignment) or `"x={x}"` (an f-string), you will get an error. In Cobra, value formatting is done as `"x=" + str(x)`, with `+` and `str(...)`. We will remind you of the list of these restrictions throughout the book as they come up.

1.3 Ways to Run the Code

There are several ways to run Cobra. In this book we will use `cobra` to run from source (if you have a compiled `cobra` binary, you can type `cobra` directly instead of `cobra`). Let us see each of the ways.

1.3.1 Running with the Tree-Walker

The simplest way is to pass a `.cobra` file directly. This uses the default tree-walking engine:

```
cobra file.cobra
```

1.3.2 Running with the Bytecode VM

When you add the `--vm` flag, the same program runs on the bytecode virtual machine. The output stays the same, but execution speeds up by about 4 times:

```
cobra --vm file.cobra
```

If you have a CPU-bound program, `--vm` is almost always the better choice.

1.3.3 Compiling and Running

You can compile a program ahead of time into a bytecode file. The `build` command produces `file.cobrac` from `file.cobra`:

```
cobra build file.cobra      # produces file.cobrac
cobra file.cobrac           # runs the compiled file
```

Running a compiled `.cobrac` file is a bit faster at startup because it skips the parsing and compilation steps. It also helps when you want to distribute the program.

1.3.4 Compiling to Native Machine Code

There is a third way to run a program, and it leaves interpretation behind entirely. With `--native`, Cobra infers the types of your program and compiles it to **machine code** — the processor executes your program directly, with no interpreter in between:

```
cobra --native file.cobra          # compile to machine code, then run
cobra build --native file.cobra    # produce a standalone executable
```

The executable it produces contains no VM and no Cobra runtime; it needs nothing installed to run. On numeric workloads it is dramatically faster than either interpreter — and produces exactly the same output, which the project verifies byte for byte.

Native compilation supports the type-stable core of the language (functions, structs, lists, dicts, strings, loops, the `math` module). A program using closures, `async`, `try`/`catch` or inheritance is reported as *not natively compilable* and runs on the VM instead. We look at how this works — and why dynamic types make it hard — in Chapter 12.

1.3.5 The Interactive REPL

When you run `cobra` with no arguments, an interactive REPL (Read-Eval-Print Loop) opens. Here you can type each line and see its result instantly — this is the fastest way to explore the language and try out small things:

```
cobra           # tree-walker REPL
cobra --vm      # the same REPL, on the bytecode VM
```

The REPL preserves state between lines; that is, you can use a variable you defined on one line on the next line:

```
> let x = 10
> let y = 20
> print(x + y)
30
```

The REPL also understands **multi-line input**. If you type a block that has not yet closed (`def`, `if`, `while`, `for`), an open parenthesis, or a bracket, the REPL notices that the input is not complete and waits for the rest from you with a `..` continuation prompt:

```
> def square(n)
..     return n * n
.. end
> print(square(9))
81
```

When the block closes with `end`, the REPL evaluates the input as a whole. This lets you write functions, loops, and structures even within the REPL.

> **Tip:** If you are not sure how a feature behaves, try it quickly in the REPL before opening a file and diving into the documentation. This will be your best friend while learning Cobra.

1.4 Your First Program: Hello, Cobra

Now it is time to set theory aside and write code. With your favorite text editor, create a file named `hello.cobra` and write this inside it:

```
print("Hello, Cobra")
```

Now run it:

```
cobra hello.cobra
```

You should see this in the terminal:

```
Hello, Cobra
```

Congratulations, you have written your first Cobra program. You can also run the same file on the bytecode VM; the output will be identical:

```
cobra --vm hello.cobra
# Hello, Cobra
```

`print` is a built-in function in Cobra. It can take more than one argument and prints them with a space placed between them:

```
print("total:", 2 + 3)   # total: 5
print("a", "b", "c")      # a b c
```

`print` automatically converts numbers to text. But if you want to join a number with another piece of text using `+`, you must first convert it explicitly with `str(...)` (remember that there are no f-strings in Cobra):

```
let age = 36
print("age " + str(age))   # age 36
```

1.5 Comments

In Cobra, comments begin with the # sign and continue to the end of the line. Comments are completely ignored by the interpreter; they exist to explain your code:

```
# This is a full-line comment
let pi = 3.14159   # and this is an end-of-line comment

# The line below does not run because it is inside a comment:
# print("this won't run")
```

In Cobra there is only this single-line comment form; there is no multi-line block comment (like `/* ... */`). If you want to comment out multiple lines, you put a # at the start of each line.

1.6 Statements, Expressions, and Line Endings

Cobra code consists of **statements**. A statement is a command that can be executed: a variable definition, an assignment, a function call, a loop. An **expression**, on the other hand, is something that reduces to a value: `2 + 3`, `square(9)`, `x and y` are all expressions.

The distinction is important, but Cobra is flexible about it; for example, even an assignment is an expression (it reduces to a value). For now the main rule you need to grasp is this:

Statements end at the end of the line. There are no semicolons.

In most C-like languages you put a `;` at the end of every statement. In Cobra there is no need for this — the end of a line means the statement has also ended:

```
let a = 1
let b = 2
print(a + b)   # 3
```

The three lines above are three separate statements. None of them has a semicolon at the end, and none should.

The only exception to this rule is lines that remain inside brackets. If a list, dictionary, or function call is opened with `(`, `[`, or `{`, it can spread across multiple lines until it closes:

```
let numbers = [
    1, 2, 3,
    4, 5, 6
]
print(len(numbers))   # 6
```

Here, because the list is inside open brackets, the line endings do not terminate the statement; Cobra waits until the bracket closes.

> **Note:** You do not need a separator (such as `;`) to cram more than one statement onto a line, because no such thing exists. In Cobra the "one line, one statement" rule is simple and clear.

1.7 Diagnostic Tools: `--ast` and `--tokens`

Cobra offers two diagnostic flags so you can see how your code is interpreted behind the scenes. These are tools you will not use in everyday programming, but they are very valuable when learning the language or when trying to understand a problem in depth.

The `--tokens` flag, instead of running the source, prints the **raw token stream**. That is, you see which pieces Cobra breaks your text into: keywords, numbers, operators, identifiers:

```
cobra --tokens hello.cobra
```

The `--ast` flag goes one step further and prints the **parsed abstract syntax tree** (AST) — that is, how the tokens are turned into a structured program:

```
cobra --ast hello.cobra
```

These two tools open a window onto the first two stages of Cobra's pipeline (lexical analysis and parsing). For now it is enough to know this much; we will cover reading the output of these flags and the language's inner architecture in detail in Chapter 12.

> **Tip:** When you cannot make sense of a syntax error, looking at the `--tokens` output is often illuminating; you can immediately notice that Cobra broke up your code differently than you thought.

1.8 Map of the Book

In this chapter we laid the foundation: we saw what Cobra is, with what philosophy it is designed, how to run code, and the most fundamental concepts of the language. From here on, each chapter will add one more layer.

In the chapters that follow we will get into values and variables (`let`, `const`, numbers, strings, boolean values), control flow (`if/elif/else`, `while`, `for ... in`, `range`), collections (lists and dictionaries, slicing), functions and closures (closures, decorators), structures (`struct`, `record`, `contract`, inheritance), error handling (`try/catch/finally`, `throw`), modules and the standard library (`import`, `fs`, `math`, `json`, `http`, and more), and the GIL-free concurrency Cobra is proud of (`async/await`, `Mutex`, `Channel`). Toward the end of the book, in Chapter 12, we will examine in depth the architecture behind the `--ast` and `--tokens` tools we touched on in this chapter — from the lexer to the bytecode virtual machine.

We will introduce each new concept first with small, working examples, then show how they come together in real programs. Just as in this chapter, the best way to learn is to run the code yourself.

1.9 Summary

In this chapter we learned the fundamentals of Cobra:

- **Cobra** is a scripting language with simple syntax. It closes blocks with `end`, writes the logical operators with the words `and/or/not`, and treats indentation as purely cosmetic.
- The language has **two execution engines**: the default **tree-walking interpreter** and the roughly 4 times faster **bytecode virtual machine** (`--vm`). The two share the same semantics and produce exactly the same output. Beyond them, `--native` **compiles a program to machine code**, leaving interpretation behind entirely.

- Cobra **contains no GIL**; tasks started with `async def` run genuinely in parallel across CPU cores.
- Ways to run code: `cobra file.cobra` (tree-walker), `cobra --vm file.cobra` (VM), `cobra --native file.cobra` (compiled to machine code), `cobra build file.cobra` + `cobra file.cobrac` (compile and run), and, with no arguments, `cobra` for the interactive **REPL** that supports multi-line input.
- We wrote our first program with `print("Hello, Cobra")` and ran it on both the tree-walker and the VM.
- **Comments** begin with `#`. **Statements end at the end of the line; there are no semicolons.** Lines inside brackets are outside this rule.
- There are two **diagnostic tools**: `--tokens` prints the raw token stream, `--ast` prints the parsed syntax tree. We will see the details in Chapter 12.

You can now write and run a Cobra program, explore the language through the REPL, and have a basic intuition for how code is interpreted. In the next chapter we will take up values and variables and begin to write real programs.

Chapter 2

Basics — Variables, Types, and Operators

Every programming language starts with a few foundational building blocks: where we put values (variables), what kinds of values we can work with (types), and how we combine and transform those values (operators). In this chapter we will cover these three foundations of Cobra from start to finish. Cobra is deliberately a small and readable language; so the rules you will learn here are few but consistent. Once you have learned them, the rest of the language sits neatly on top of these rules.

To run all the examples, you can write them in a `.cobra` file and use the `cobra file.cobra` command, or run the `cobra` command with no arguments to try them one by one in the interactive REPL. Cobra has two engines — the default tree-walking interpreter and the bytecode virtual machine that runs with `--vm` — but both produce the same results, so nothing in this chapter depends on which engine you choose.

2.1 Variables: `let` and `const`

There are two ways to create a variable in Cobra. `let` defines a **mutable** binding; `const` defines an **immutable** one.

```
let counter = 0
const PI = 3.14159

print(counter)   # 0
print(PI)      # 3.14159
```

To assign a new value to an existing variable, you do not write `let` again; you just use the name and =:

```
let counter = 0
counter = counter + 1
print(counter)   # 1
```

An important rule: assigning to a variable that has **not been previously defined** is an error. So that it can catch typos, Cobra requires every assignment to target an existing binding.

```
let total = 10
totl = 20      # ERROR: 'totl' is not defined (typo is caught)
```

If you want a new variable, declare it with `let`; if you are updating an existing one, assign without `let`. This distinction keeps you from accidentally creating a new variable and shadowing the old value.

> **Note** Cobra's two engines catch an undefined variable at different times: the virtual machine (`--vm`) catches it at compile time, while the tree-walking interpreter catches it as the line runs. The error message is the same in both cases.

2.1.1 Assignment is an expression

In Cobra, assignment is not a **statement** but an **expression**: it produces a value. This is useful in some situations, but mostly it is a detail you only need to know conceptually.

```
let x = 0
let y = (x = 5)   # x is assigned 5, and the value of this assignment is also 5
print(x)          # 5
print(y)          # 5
```

2.1.2 `const` freezes the binding, not the value

Reassigning a name defined with `const` is an error. Redeclaring it with `let` or `const` in the same scope is also an error.

```
const MAX = 100
MAX = 200        # ERROR: a constant cannot be reassigned
const MAX = 5    # ERROR: cannot be redeclared in the same scope
```

The most commonly confused point here is this: `const` freezes the **binding** (that is, which value the name points to), but not the **inner contents** of that value. If you bind a list or a dictionary with `const`, you cannot redirect that name to another value; but you can change the contents of the list/dictionary.

```
const xs = [1]
xs.push(2)        # ALLOWED: the value's contents change, not the binding
print(xs)         # [1, 2]

xs = [9, 9]       # ERROR: you cannot change the binding
```

> **Tip** If you want a truly immutable data structure, use a `record` (we will see this in later chapters). `const` only guarantees that the name stays fixed, not the contents.

2.1.3 Shadowing in an inner scope

A `const` (or `let`) in an outer scope can be **shadowed** by a new `let` in an inner scope. This does not change the outer binding; it just makes the same name point to a new value for the duration of the inner scope.

```
const name = "outer"

def show()
    let name = "inner"     # shadows the outer 'name' — not an error
    print(name)         # inner
end

show()
print(name)             # outer (unchanged)
```

2.2 Types

Cobra's core value types are simple. In this chapter we focus on numeric and simple types; composite types such as lists, dictionaries, functions, and structs will be covered in their own chapters.

Type	Example	Description
integer (int)	`42`, `-7`	64-bit signed integer
float	`3.14`, `2.0`	double-precision decimal
decimal	`19.90m`	exact/precise arithmetic (finance)
string	`"hello"`	immutable text (double quotes only)
boolean	`true`, `false`	truth value (lowercase)
null	`null`	"no value"

```
let n = 42            # int
let f = 3.14          # float
let price = 19.90m    # decimal
let s = "hello"       # string
let b = true          # boolean
let nothing = null

print(type(n))        # INTEGER
print(type(f))        # FLOAT
print(type(s))        # STRING
print(type(b))        # BOOLEAN
print(type(nothing))  # NULL
```

> **Note** In Cobra the empty/absent value is always `null`. The `None`, `nil`, or `undefined` from other languages do not exist in Cobra. Likewise, booleans are always lowercase `true`/`false` — `True`/`False` do not work.

2.2.1 Integers are 64-bit

Cobra's integers are 64-bit signed; that is, they cover a range of roughly $\pm 9.2 \times 10^{18}$. There is **no** "big integer" (bignum) support beyond this limit; when overflow happens, the value wraps around; it does not automatically grow into a wider integer type.

2.2.2 Decimal — exact arithmetic for finance

Because floating-point numbers (float) are stored in base two, they produce small rounding errors in calculations like `0.1 + 0.2`. For situations where precision is essential, such as money, Cobra offers the `decimal` type. You write a decimal literal by appending `m` to the end of a number:

```
let price = 19.90m
let quantity = 3
print(price * quantity)   # 59.70  (exact, no rounding error)

print(decimal("0.1") + decimal("0.2"))   # 0.3  (exactly)
```

> **Tip** For calculations like money, tax, and interest, use `decimal` instead of `float`. `float` is designed for scientific/engineering calculations, while `decimal` is designed for finance.

2.3 Working with numbers

2.3.1 A mix of int and float is promoted to float

When you combine an int with a float in an arithmetic operation, the result is a float. This is called **type promotion**.

```
print(1 + 2.5)        # 3.5   (result is float)
print(10 * 2.0)       # 20.0
print(1 == 1.0)       # true  (considered equal in value)
```

2.3.2 Between two ints, / is INTEGER division

This is the rule that most often surprises those coming from other languages. The `/` operator between two integers performs **integer division**: it truncates the result down, toward zero.

```
print(7 / 2)          # 3    (integer division — not 3.5!)
print(10 / 3)         # 3
```

If you want a decimal result, convert at least one of the operands to float:

```
print(float(7) / 2)   # 3.5
print(7 / 2.0)        # 3.5
```

> **Warning** Cobra has **no** `//` (floor division) operator. The `/` between two ints already behaves like floor division; writing `7 // 2` is a syntax error.

2.3.3 Type conversions

To convert a value to another type, you use the built-in functions whose names are the type names:

```
print(int(3.9))       # 3      (truncates toward zero — does not round)
print(int(-3.9))      # -3     (toward zero — not down)
print(float(7))       # 7.0
print(str(42))        # "42"   (number -> text)
print(decimal(10))    # 10
```

These functions also work for string parsing:

```
print(int("100") + 1)       # 101
print(float("3.14"))        # 3.14
print(decimal("19.90"))     # 19.90
```

> **Note** `int(x)` always truncates a decimal number **toward zero**, it does not round down mathematically. So `int(-3.9)` results in `-3`, not `-4`. For actual rounding, see the `math.round`, `math.floor`, and `math.ceil` functions.

2.4 Operators

2.4.1 Arithmetic

```
print(5 + 3)          # 8
print(5 - 3)          # 2
print(5 * 3)          # 15
print(7 / 2)          # 3    (integer division)
print(7 % 3)          # 1    (remainder / modulo)
```

The + operator performs addition on numbers, but **concatenation** on strings:

```
print("hello " + "world")     # hello world
print("age: " + str(36))        # age: 36
```

> **Warning** + requires either two numbers or two strings; you cannot add a string and a number directly. `"age: " + 36` raises an error; convert the number to text first with `str(36)`. Cobra has no f-strings or text interpolation; you build text with + and `str(...)`.

2.4.2 Comparison

Comparison operators always return a boolean:

```
print(3 == 3)         # true
print(3 != 4)         # true
print(3 < 4)          # true
print(3 > 4)          # false
print(3 <= 3)         # true
print(4 >= 5)         # false
```

2.4.3 Logical operators: `and`, `or`, `not`

In Cobra the logical operators are **words** — not &&, ||, !. This makes the code more readable.

```
print(true and false)   # false
print(true or false)    # true
print(not true)         # false
```

`and` and `or` **short-circuit**: they stop after evaluating the first operand sufficient to determine the result. More interestingly, they return not a boolean, but **the operand that decided the outcome itself**.

```
print(0 or "x")         # x       (0 is falsy, moves to the second)
print("a" or "b")       # a       (the first is truthy, returns immediately)
print(1 and 2)          # 2       (the first is truthy, the second decides)
print(0 and 2)          # 0       (the first is falsy, returns immediately)
```

This behavior is very handy for assigning a default value:

```
let name = entered_name or "anonymous"   # if empty, use "anonymous"
```

2.5 Truthiness

In places like `if`, `while`, `and`, `or`, and the ternary expression, values are interpreted as a boolean. In Cobra the following six values are considered **falsy**:

- `false`
- `null`
- `0` (zero — int or float)
- `""` (empty string)
- `[]` (empty list)
- `{}` (empty dictionary)

Everything else is considered **truthy**.

```
if []
    print("this won't run")
else
    print("an empty list is falsy")    # this runs
end

let list = [1, 2, 3]
if list
    print("a non-empty list is truthy")    # this runs
end
```

> **Tip** To check whether a list or dictionary is empty, you do not need to write `len(x) == 0`; `if not x` is enough on its own.

2.6 Augmented assignment

The operators `+= -= *= /= %=` update a binding in place. The expression `x += 1` means exactly `x = x + 1`; therefore it follows all the rules of the `+` operator (string concatenation, int→float promotion, int/int integer division).

```
let n = 10
n += 5     # n = n + 5
print(n)   # 15
n -= 3     # 12
n *= 2     # 24
n /= 5     # 4   (integer division: 24 / 5 = 4)
n %= 3     # 1
print(n)   # 1

let message = "hel"
message += "lo"          # string concatenation
print(message)             # hello
```

Augmented assignment works not only on simple variables, but also on **index targets** and **struct fields**:

```
let xs = [10, 20, 30]
xs[1] += 5
print(xs)                # [10, 25, 30]

let counts = {"a": 1}
counts["a"] += 1
print(counts["a"])        # 2

struct Counter
    def init()
        self.n = 0
    end
    def increment()
        self.n += 1      # augmented assignment on a field
    end
end

let c = Counter()
c.increment()
c.increment()
print(c.n)               # 2
```

> **Warning** Augmented assignment cannot be applied to a `const`; because it changes a binding, writing `TOTAL += 1` on `const TOTAL = 0` is an error.

2.7 Ternary expression

Cobra supports the `condition ? a : b` ternary expression from the C family: it equals `a` if the condition is truthy, and `b` otherwise. **Only the selected branch runs**, so the side effects of the branch that does not run are not triggered.

```
let age = 20
let status = age >= 18 ? "adult" : "child"
print(status)             # adult
```

The ternary expression is **right-associative**, meaning that when chained it groups from the right:

```
# a ? b : c ? d : e   means:   a ? b : (c ? d : e)
let score = 75
let grade = score >= 90 ? "A" : score >= 80 ? "B" : score >= 70 ? "C" : "F"
print(grade)              # C
```

In terms of precedence, the ternary expression binds **looser** than `or`/`and`, but **tighter** than assignment (=). This way you can assign the selected value directly to a variable:

```
let x = 5
let sign = x > 0 ? "positive" : "negative"   # works fine since = is the loosest
print(sign)            # positive
```

> **Note** Cobra has **no** ternary expression of the form `a if c else b`; the only ternary form is `c ? a : b`.

2.8 Helper built-ins: `type` and `hash`

`type(x)` returns the type name of a value as a string. This is useful in debugging and in runtime type checking:

```
print(type(42))          # INTEGER
print(type(3.14))        # FLOAT
print(type("x"))         # STRING
print(type([1, 2]))      # LIST
print(type({"a": 1}))    # DICT
print(type(null))        # NULL
```

`hash(s)` produces an integer digest (hash) for a value; the same value always gives the same digest:

```
print(hash("hello") == hash("hello"))   # true
```

2.9 Operator precedence

When multiple operators come together, **precedence** determines which one is applied first. In the table below, precedence increases from top to bottom (the tightest binding is at the bottom):

Precedence	Operators	Description
loosest	=, +=, −=, …	assignment (right-associative)
	`? :`	ternary expression (right-associative)
	`or`	logical OR (short-circuit)
	`and`	logical AND (short-circuit)
	`not`	logical NOT (unary)
	`== != < > <= >=`	comparison
	`+ −`	addition / subtraction
	`* / %`	multiplication / division / modulo
	unary −	negative sign
tightest	`f(...)`, `x[...]`, `x.y`	call, index, field access

Some practical consequences that follow from this table:

```
print(2 + 3 * 4)         # 14   (* first: 2 + 12)
print(not 3 == 3)        # false  (means not (3 == 3))
print(1 + 2 == 3)        # true  (+ is tighter than comparison: (1+2) == 3)
print(true or false and false)   # true  (and first: true or (false and false))
```

Either state precedence explicitly with grouping parentheses (`(...)`), or use parentheses for readability whenever you are in doubt — grouping parentheses are always allowed in Cobra.

2.10 Summary

In this chapter we learned Cobra's fundamental building blocks:

- **Variables:** `let` defines a mutable binding, `const` an immutable one. Assigning to an undefined variable is an error. Assignment is an expression and produces a value. `const` freezes only the **binding**, not the value's contents (`const xs = [1]` → `xs.push(2)` is allowed, `xs = ...` is forbidden). Inner scopes can shadow an outer binding with `let`.
- **Types:** int (64-bit), float, decimal (`19.90m`, exact for finance), string, boolean (`true`/`false`), and `null`. The empty/absent value is always `null`.
- **Numbers:** a mix of int/float is promoted to float (`1 + 2.5` → `3.5`, `1 == 1.0` is true). Between two ints, `/` is integer division (`7 / 2` → `3`); for a decimal result, `float(7) / 2`. There is no `//` operator.
- **Conversions:** `int(x)` (truncates toward zero), `float(x)`, `str(x)`, `decimal(x)` — all also perform string parsing.
- **Operators:** arithmetic `+ - * / %`; `+` concatenates on strings; comparison `== != < > <= >=`; logical `and`/`or`/`not` (as words). `and`/`or` short-circuit and return **the operand that decided the outcome** (`0 or "x"` → `"x"`).
- **Truthiness:** `false`, `null`, `0`, `""`, `[]`, `{}` are falsy; the rest are truthy.
- **Augmented assignment:** `+= -= *= /= %=`; `x += 1` is exactly `x = x + 1`, works on a variable, an index (`xs[i] += 1`), and a field (`self.n += 1`), cannot be applied to a `const`.
- **Ternary expression:** `condition ? a : b`, right-associative, only the selected branch runs; binds tighter than `=`, looser than `or`/`and`.
- **Helpers:** `type(x)` returns the type name, `hash(s)` returns a digest.

With these basics we can now store, transform, and combine values. In the next chapter we will bring these values together with conditions and loops to give our programs flow.

Chapter 3

Control Flow

The value of a program comes not just from storing data, but from being able to look at that data and *make decisions*. Is a number greater than zero? Are there any elements left to process in a list? Do we need to do a job over and over until a condition is met? The way we tell a program the answers to these questions is called *control flow*.

In Chapter 2 we saw values, variables, and operators. In this chapter we will bring them together and let the program decide which lines will run and how many times a job will be repeated. Cobra's control flow tools are deliberately few and similar to one another; they all use blocks that close with the `end` keyword. By the time you finish this chapter, you will be able to comfortably use conditions (`if`/`elif`/`else`), two kinds of loops (`while` and `for ... in`), lazy `range` sequences, and the `break`/`continue` statements that steer the flow of a loop.

All of Cobra's blocks follow the same rule: a block opens with a keyword (for example `if`, `while`, `for`), consists of one or more body lines, and closes with `end`. Indentation is purely cosmetic; we use it for readability but it has no meaning to Cobra. The only thing that ties blocks together is `end`.

3.1 Conditions: `if`, `elif`, `else`

The most basic decision structure is the `if` statement. It evaluates a condition; if the condition is *truthy*, it runs the corresponding body, otherwise it skips it.

```
let temperature = 32

if temperature > 30
    print("It's hot today")
end
# Output:
# It's hot today
```

If you want to try multiple cases in order, you add `elif` (one word — there is no such thing as `else if`) and `else`. Cobra tries the conditions from top to bottom; it runs the body of the *first true* branch and skips the rest. If none are true, the `else` branch runs (if there is one).

```
let score = 72

if score >= 90
```

```
    print("AA")
elif score >= 80
    print("BA")
elif score >= 70
    print("BB")
else
    print("Passed or failed, check another table")
end
# Output:
# BB
```

All three branches here could technically be true (72 is both `>= 70` and `>= 60`), but Cobra stops at the first matching branch. For this reason, ordering the conditions from the narrowest range to the widest range gives the correct result.

> **Note:** In Cobra, `elif` and `else` are not mandatory. A standalone `if` block is perfectly valid. Also, there is *no* two-word form `else if`; the correct spelling is the single word `elif`.

3.1.1 Let's recall the truthiness rules

The condition of an `if` and `while` does not have to be `true`/`false`; Cobra interprets any value as "is this true?". As we saw in Chapter 2, the following values are considered *falsy*:

- `false`
- `null`
- `0` (numeric zero)
- `""` (empty string)
- `[]` (empty list)
- `{}` (empty dictionary)

Everything else is considered *truthy*. This makes conditions very readable:

```
let name = ""
if name
    print("Hello " + name)
else
    print("Name is empty")
end
# Output:
# Name is empty

let list = [1, 2, 3]
if list
    print("The list has " + str(len(list)) + " elements")
end
# Output:
# The list has 3 elements
```

Remember that the logical operators are also words: `and`, `or`, `not` (never `&&`, `||`, `!`). `and` and `or` short-circuit and return the *operand* that decides the outcome:

```
let user = null
let name = user or "guest"   # "guest" because user is falsy
print(name)
# Output:
# guest
```

> **Tip:** If you only want to choose a *value* between two options, instead of writing a whole `if`/`else` block you can use the ternary operator: `let status = score >= 50 ? "passed" : "failed"`. This is of the form `cond ? a : b` and only the selected branch runs.

3.2 The `while` loop

`while` runs its body over and over as long as a condition is true. At the start of each turn it looks at the condition again; when the condition becomes false the loop ends and the program continues from the line after `end`.

```
let counter = 1
while counter <= 5
    print(counter)
    counter = counter + 1
end
# Output:
# 1
# 2
# 3
# 4
# 5
```

Note: if we do not increment `counter` inside the body, the condition never becomes false and the loop spins forever. Keeping a counter or an accumulator variable is very common in `while` loops. The classic example is computing the sum of a range:

```
let n = 1
let total = 0
while n <= 100
    total += n        # same as total = total + n
    n += 1
end
print(total)
# Output:
# 5050
```

Here there are two variables: a *counter* (`n`, which tracks where we are) and an *accumulator* (`total`, which accumulates the result). This pair is the fundamental pattern of loops.

> **Warning:** When you write a `while` loop, ask yourself "*how* will this condition become false?" If you do not have an answer, you have written an infinite loop. Usually there should be a line in the body that advances the counter or changes the condition.

3.3 `for ... in`: iterating over something

Most of the time we do not want to manage a counter by hand; we just want to process each element of a collection in order. That is exactly what `for ... in` is for. The syntax is as follows:

```
for item in iterable
    # do something with item
end
```

Cobra lets you iterate over four kinds of things: lists, strings, dictionary keys, and `range(...)` sequences. Let's look at each one in turn.

3.3.1 Iterating over lists

This is the case you will encounter most often. `for` gives the list's elements from start to finish, in order:

```
let fruits = ["apple", "pear", "cherry"]
for fruit in fruits
    print(fruit.upper())
end
# Output:
# APPLE
# PEAR
# CHERRY
```

3.3.2 Iterating over strings (character by character)

When you iterate over a string, Cobra gives it to you *character by character*. Each turn is a single-character string:

```
for letter in "abc"
    print(letter)
end
# Output:
# a
# b
# c
```

> **Note:** Iterating over strings works character by character (rune), but *indexing* a string by hand with `s[i]` is byte-based and can corrupt multi-byte UTF-8 characters (for example accented or other non-ASCII letters). When splitting text into pieces, a `for` loop or the `re` module is a safer choice.

3.3.3 Iterating over dictionary keys (in insertion order)

When you iterate over a dictionary (dict), Cobra gives you the *keys* — and in the order they were inserted, because dictionaries in Cobra preserve insertion order.

```
let ages = {"alice": 30, "bob": 25, "carol": 35}
for name in ages
    print(name + ": " + str(ages[name]))
end
# Output:
# alice: 30
# bob: 25
# carol: 35
```

If you want to iterate over both the key and the value together, you can use `items()`; but remember that Cobra has no multiple assignment / unpacking. `items()` gives you two-element lists in the form `[key, value]` and you read them by indexing:

```
let ages = {"alice": 30, "bob": 25}
for pair in ages.items()
    print(pair[0] + " -> " + str(pair[1]))
end
# Output:
# alice -> 30
# bob -> 25
```

> **Warning:** A form like `for key, value in d.items()` *does not work* in Cobra — the language has no tuples and no unpacking. You must read the two-element list with `pair[0]` and `pair[1]`.

3.4 `range`: generating number sequences

Most of the time we want to repeat something a certain number of times or iterate over numbers. `range` generates an arithmetic (equally spaced) number sequence for this. It has three forms:

```
print(range(5))           # from 0 up to 5 (5 not included)
# Output:
# [0, 1, 2, 3, 4]

print(range(2, 6))        # from 2 up to 6 (6 not included)
# Output:
# [2, 3, 4, 5]

print(range(0, 10, 2))    # from 0 to 10, in steps of 2
# Output:
# [0, 2, 4, 6, 8]
```

That is:

- `range(stop)` → `0, 1, ..., stop-1`
- `range(start, stop)` → `start, start+1, ..., stop-1`
- `range(start, stop, step)` → starts at `start` and adds `step` each time, stopping before reaching `stop`.

A typical use is at the start of a `for` loop:

```
for i in range(3)
    print("round " + str(i))
end
# Output:
# round 0
# round 1
# round 2
```

3.4.1 `range` is LAZY — it is not a real list

Although `range` is printed on screen like a list, it is actually a *lazy* object. It does not produce all the numbers in memory; it computes them as needed. When you ask `type()` about it, it returns `"RANGE"` — not `"LIST"`:

```
let r = range(1, 10, 2)
print(type(r))    # not "LIST"!
# Output:
# RANGE

print(r[0])       # indexable
# Output:
# 1

print(r)          # printed like a list
# Output:
# [1, 3, 5, 7, 9]
```

You can index a RANGE, iterate over it, and slice it; but it is *not* a real list. It does not have the mutating methods specific to lists (for example `.push`):

```
let r = range(3)
r.push(4)
# runtime error: line 2: RANGE has no method 'push'
```

> **Tip:** If you want to convert a RANGE into a real, mutable list, you can iterate over it and add elements to a new list, or pass it to higher-order functions like `map`/`filter`; these return you a new list. The laziness of `range` is an advantage because it lets you loop over a range of millions of elements with no memory cost.

The big benefit of `range` is this: writing `for i in range(1000000)` does not create a list of a million elements; it produces each number exactly when it is needed.

3.5 `break` and `continue`: steering the loop flow

We can intervene in the natural flow of a loop in two ways. `break` terminates the loop *entirely*; `continue` skips only *the current turn* and moves to the next. Both work in `while` and `for` loops.

With `break` we can stop a search once we find the target:

```
let numbers = [4, 8, 15, 16, 23, 42]
for s in numbers
```

```
    if s > 20
        print("first number greater than 20: " + str(s))
        break
    end
end
# Output:
# first number greater than 20: 23
```

With `continue` we can skip some elements. Below we only sum the odd numbers:

```
let total = 0
for n in range(1, 6)
    if n % 2 == 0
        continue          # skip even numbers
    end
    total += n
end
print(total)             # 1 + 3 + 5
# Output:
# 9
```

3.5.1 `break`/`continue` outside a loop is a PARSE error

`break` and `continue` are only meaningful inside a loop. If you try to use them *outside* a loop, the program errors during the parse phase, before it even runs:

```
break
# parse error: line 1: 'break' outside loop
```

An important subtlety: even if written *inside* a loop, if `break`/`continue` is inside the body of a `def` (function definition), it is still invalid. Because a function is a block independent of the loop within itself; it is impossible for an inner `break` to jump to the outer loop:

```
for i in range(3)
    def f()
        break        # inside the function, even though written inside the loop
    end
end
# parse error: line 3: 'break' outside loop
```

> **Warning:** This error is caught at *parse* time, that is, before the program ever starts running. This is a good thing: it does not leave a misplaced `break`/`continue` to runtime, but tells you immediately.

3.6 The loop variable lives on after the loop

In some languages the lifetime of the loop variable is limited to the loop body. In Cobra it is *not* so: a `for` loop's variable remains accessible after the loop ends and holds the last value it took.

```
for n in [1, 2, 3]
    # ...
end
print(n)
# Output:
# 3
```

This behavior can be useful when you need to know "where did we last leave off?" after the loop. But it also brings a caveat: if you are going to use a variable of the same name after the loop for another purpose, do not forget that the loop variable still holds that name.

> **Note:** The same applies to `while` — variables you define or update with `let` inside a `while` live on after the loop too. In Cobra, blocks (loops, `if`s) do not open a new scope; only `def` bodies do.

3.7 Nested loops

You can nest loops inside one another. This is natural for, say, producing a table (row × column). The inner loop runs from start to finish for each turn of the outer loop.

```
for i in range(1, 4)
    for j in range(1, 4)
        print(str(i) + " x " + str(j) + " = " + str(i * j))
    end
end
# Output:
# 1 x 1 = 1
# 1 x 2 = 2
# 1 x 3 = 3
# 2 x 1 = 2
# 2 x 2 = 4
# 2 x 3 = 6
# 3 x 1 = 3
# 3 x 2 = 6
# 3 x 3 = 9
```

In nested loops, pay attention to `break` behavior: `break` terminates *only the innermost loop that surrounds it*. The outer loop continues from where it left off.

```
for i in range(2)
    for j in range(3)
        if j == 1
            break          # breaks only the inner (j) loop
        end
        print(str(i) + "," + str(j))
    end
end
# Output:
# 0,0
# 1,0
```

As you can see, the inner loop breaks when `j == 1`, but the outer loop runs again for `i = 0` and `i = 1`.

> **Tip:** Some languages allow exiting the outermost loop in one move with a "labeled `break`"; Cobra has no such thing. If you need to exit a nested loop entirely, you can use a flag variable, or move the relevant code into a `def` and exit with `return`.

3.8 What Cobra DOES NOT HAVE: `for(;;)`, `switch`, `match`

For readers coming from other languages, let's clarify a few important differences. None of these constructs exist in Cobra; if you try to write them, you will get an error.

- **There is no C-style `for(;;)` loop.** In Cobra, `for` only works in the form `for x in iterable`. If you want a counter-based loop, use `range` (`for i in range(n)`) or manage a counter by hand with `while`.
- **There is no `switch` / `case`.** To compare multiple branches, use the `if/elif/else` chain.
- **There is no `match` (pattern matching).** Again, use `if/elif/else`, or use a dictionary to map a value to behaviors.

You can satisfy a longing for `switch` with `if/elif` like this:

```
let command = "save"

if command == "open"
    print("Opening file")
elif command == "save"
    print("Saving file")
elif command == "close"
    print("Closing file")
else
    print("Unknown command")
end
# Output:
# Saving file
```

If there are many cases, using a dictionary as a "dispatch table" can be cleaner — especially when you map values to functions:

```
def add(a, b)
    return a + b
end

def subtract(a, b)
    return a - b
end

let operations = {"+": add, "-": subtract}
let f = operations["+"]
print(f(10, 4))
# Output:
# 14
```

> **Note:** A function body in Cobra is *required* to be on its own line(s); a form squeezed onto a single line like `def add(a, b) return a + b end` gives a parse error. Our ability to store functions as the values of a dictionary stems from functions being first-class values in Cobra (we will see this in depth in Chapter 4).

3.9 Summary

In this chapter we saw Cobra's tools for making programs decide and for repeating work. They were all blocks that close with `end` and follow the same simple pattern:

- **Conditions** are of the form `if cond ... elif cond ... else ... end`. Cobra tries the branches from top to bottom and runs the first true one. Conditions interpret any value according to the *truthiness* rule: `false`, `null`, `0`, `""`, `[]`, and `{}` are falsy, everything else is truthy (Chapter 2).
- **`while cond ... end`** spins as long as the condition is true. Its typical pattern is a *counter* and an *accumulator* variable; make sure the condition will be false someday, or you will write an infinite loop.
- **`for x in iterable ... end`** iterates over lists, strings (character by character), dictionary keys (in insertion order), and `range(...)`.
- **`range`** comes in three forms — `range(stop)`, `range(start, stop)`, `range(start, stop, step)` — and produces a *lazy* RANGE: indexable, iterable, and sliceable, but not a real list (no `.push`) and `type()` returns `"RANGE"` for it.
- **`break`** ends the loop entirely; **`continue`** skips the current turn. Both work in `while` and `for`. Using them outside a loop — or in a `def` body inside a loop — is a *parse* error.
- A **loop variable lives on after the loop** and holds its last value.
- **In nested loops** `break` exits only the innermost loop surrounding it; there is no labeled `break`.
- Cobra has **no C-style `for(;;)`, `switch`, or `match`**; instead you use `range`/`while`, `if`/`elif`/`else`, and when needed a dispatch table with a dictionary.

Now we know how to hold data (Chapter 2) and steer the program's flow. In the next chapter we will turn these pieces into reusable units: functions.

Chapter 4

Collections — Lists, Dictionaries, and Slicing

The value of a program often lies not in stand-alone values, but in the way those values are held together. A shopping cart, a user record, a day's temperature readings — they all call for a *collection*. Cobra offers two fundamental structures for this purpose: the **list**, which holds an ordered sequence, and the **dictionary** (dict), which maps keys to values. These two structures form the backbone of the Cobra programs you will write.

In this chapter we will cover lists and dictionaries from the ground up: how they are created, how they are read, how they are modified; which methods change them in place, and which return a new value. We have set aside a separate section for **slicing**, one of Cobra's most powerful and most surprising features. There is also one subject that calls for care: lists and dictionaries are *reference types*. When you write `let b = a`, you do not get a copy but a second name for the same object. We will see why this is and what its consequences are.

Before we begin, a small reminder: in Cobra, blocks close with `end`, line endings terminate statements, and comments begin with `#`. We will show outputs inside the code examples as `#` comments.

4.1 Creating a list

A list consists of values separated by commas within square brackets. Cobra lists are *heterogeneous*: a single list can hold values of different types.

```
let xs = [1, 2, 3]
let karisik = [1, 2, "x"]
let bos = []

print(xs)        # [1, 2, 3]
print(karisik)   # [1, 2, x]
print(len(bos))  # 0
```

The built-in `len(...)` function gives the number of elements in a list. An empty list is written `[]` and has length zero.

> **Note** Within square brackets a list may span multiple lines. While inside `(`, `[`, or `{`, Cobra does not treat a line ending as the end of an expression, so you can lay out long lists in a readable way:

```
let renkler = [
    "kirmizi",
    "yesil",
    "mavi",
]
```

4.2 Indexing and negative indices

You reach a list's elements with an index that starts at zero.

```
let l = ["a", "b", "c", "d"]
print(l[0])   # a
print(l[2])   # c
print(len(l)) # 4
```

Cobra has a nice convenience: a *negative index* counts from the end. `l[-1]` is always the last element, `l[-2]` the second from last.

```
print(l[-1])  # d
print(l[-2])  # c
```

This way you do not need to write `l[len(l) - 1]` to reach the "last element."

> **Warning** An index that goes outside the list's bounds (for example `l[4]` or `l[-5]` on a four-element list) produces a runtime error. Out-of-bounds values are *not clamped* in indexing. This is an important difference from the slicing we will see shortly — in slicing, out-of-bounds values are silently clamped.

4.3 Index assignment

You can change the value at a particular position in a list by assigning to that position.

```
let l = [10, 20, 30]
l[0] = 99
l[-1] = 0
print(l)   # [99, 20, 0]
```

Augmented assignment also works on index targets; `l[i] += 1` means exactly `l[i] = l[i] + 1`:

```
let sayac = [0, 0, 0]
sayac[1] += 5
print(sayac)   # [0, 5, 0]
```

4.4 Concatenation and equality

The + operator joins two lists and produces a *new* list. The source lists are unchanged.

```
let a = [1, 2]
let b = [3, 4]
let c = a + b
print(c)   # [1, 2, 3, 4]
print(a)   # [1, 2]  (a unchanged)
```

Lists are compared *deeply* with ==: two lists are equal if they have the same length and their corresponding elements are equal. This comparison also applies to nested lists.

```
print([1, 2, 3] == [1, 2, 3])     # true
print([1, [2, 3]] == [1, [2, 3]]) # true  (deep equality)
print([1, 2] == [1, 2, 3])        # false
```

> **Note** Value equality holds across a mix of numeric types: `1 == 1.0` is true, so `[1, 2] == [1.0, 2.0]` also returns `true`.

4.5 Lists are reference types

This is the part that needs attention. In Cobra, when you assign a list to another variable, *no copy is made*; both variables point to the same list. This is called an *alias*.

```
let a = [1, 2, 3]
let b = a        # b is NOT a copy of a — an alias to the same list
b.push(4)
print(a)         # [1, 2, 3, 4]  (a changed too!)
print(b)         # [1, 2, 3, 4]
print(a == b)    # true (it is the same list anyway)
```

A change you make through `b` is also visible through `a`, because there is a single list. The same holds when you pass a list to a function: if you modify the list inside the function, the caller sees that change.

```
def ekle_bir(liste)
    liste.push(99)
end

let veri = [1, 2]
ekle_bir(veri)
print(veri)   # [1, 2, 99]
```

If you really want an independent copy, you can produce a shallow copy using slicing. The expression `a[:]` returns a *new* list containing all the elements (we will look at slicing in detail shortly):

```
let a = [1, 2, 3]
let kopya = a[:]   # a new list
kopya.push(4)
print(a)           # [1, 2, 3]  (unaffected)
print(kopya)       # [1, 2, 3, 4]
```

Tip If you accidentally modify a list shared across several lines and get unexpected results, let aliasing be your first suspect. The answer to "why did this list change too?" is usually "because the two are actually the same list." When you want an independent copy, use `liste[:]`.

4.6 List methods

Lists have a set of methods called with a dot. Some of them modify the list *in place* (mutators), and some return a new value. Telling the two apart makes debugging much easier.

Method	What it does	In place?
`push(v)`	Appends an element to the end	Yes (in place)
`pop()`	Removes and returns the last element	Yes (in place)
`reverse()`	Reverses the list	Yes (in place)
`sort()`	Sorts the list	Yes (in place)
`contains(v)`	Is the element present? `true`/`false`	No (reads)
`find(v)`	First index it occurs at, or `-1`	No (reads)
`count(v)`	How many times it occurs	No (reads)

First, the ones that modify in place:

```
let xs = [3, 1, 2]
xs.push(4)
print(xs)          # [3, 1, 2, 4]

let son = xs.pop()
print(son)         # 4
print(xs)          # [3, 1, 2]

xs.sort()
print(xs)          # [1, 2, 3]

xs.reverse()
print(xs)          # [3, 2, 1]
```

`sort` and `reverse` modify the list *in place* and do not return the list; if you want a "sorted copy," make a copy first (`let sirali = xs[:]` then `sirali.sort()`).

Now the methods that read the list without modifying it:

```
let nesneler = ["elma", "armut", "elma", "kiraz"]
print(nesneler.contains("armut")) # true
print(nesneler.find("elma"))      # 0   (first index it occurs at)
print(nesneler.find("muz"))       # -1  (not found)
print(nesneler.count("elma"))     # 2
```

`find` returns `-1` for an element that is not found; so for the "is it present?" question `contains` is more readable, while `find` is suited to the "where?" question.

4.6.1 Built-in function forms

`push` and `pop` can also be called as built-in functions. `xs.push(v)` does the same thing as `push(xs, v)`; and `xs.pop()` the same as `pop(xs)`.

```
let xs = [1, 2]
push(xs, 3)       # same as xs.push(3)
print(xs)         # [1, 2, 3]

let son = pop(xs) # same as xs.pop()
print(son)        # 3
print(xs)         # [1, 2]
```

Which you prefer is entirely a matter of style. The method form (`xs.push(3)`) reads more fluently in most cases.

4.7 Slicing

Slicing is the short and powerful way to take a *subpart* of a sequence. The general syntax is of the form `seq[low:high:step]`:

- `low` — the start index (inclusive)
- `high` — the end index (exclusive)
- `step` — the step (default 1)

The nicest part is that any piece *can be omitted*. An omitted piece is filled in with a sensible default.

```
let xs = [0, 1, 2, 3, 4, 5]

print(xs[1:4])   # [1, 2, 3]   (from 1 up to 4, excluding 4)
print(xs[:2])    # [0, 1]      (from the start up to 2)
print(xs[3:])    # [3, 4, 5]   (from 3 to the end)
print(xs[:])     # [0, 1, 2, 3, 4, 5]  (a full copy)
```

Note the last line: `xs[:]` returns a *new copy* of the list from start to end. This is exactly the "independent copy" trick we just saw.

4.7.1 Negative indices work in slices too

Just as in single indexing, negative numbers count from the end in slice bounds as well.

```
let xs = [0, 1, 2, 3, 4, 5]
print(xs[-2:])   # [4, 5]      (the last two elements)
print(xs[:-2])   # [0, 1, 2, 3] (all but the last two)
print(xs[-3:-1]) # [3, 4]
```

4.7.2 Step and reversing

The third piece, `step`, says how far to advance each time. Use `2` to go two at a time, and a negative step to go backward.

```
let xs = [0, 1, 2, 3, 4, 5]
print(xs[::2])   # [0, 2, 4]   (skipping by twos)
print(xs[1::2])  # [1, 3, 5]
print(xs[::-1])  # [5, 4, 3, 2, 1, 0]  (a reversed copy)
```

The expression `xs[::-1]` returns a reversed *copy* of the list — do not confuse it with `reverse()`: `reverse()` modifies the list in place and returns nothing, while `xs[::-1]` leaves the original alone and gives a new list.

4.7.3 Out-of-bounds values are clamped

This is the most important difference between slicing and single indexing: if the slice bounds run outside the list, no error occurs; the values are silently *clamped*.

```
let xs = [0, 1, 2]
print(xs[0:100]) # [0, 1, 2]   (clamped to the end instead of 100)
print(xs[-100:]) # [0, 1, 2]   (went too far back, clamped to the start)
print(xs[5:9])   # []          (entirely outside — empty list)
```

This behavior saves you from checking one by one whether a computed index exceeds the bounds. You can write `xs[i:i+3]` and know you will not get an error even when you are near the end of the list.

> **Note** Slicing returns a new list on lists. The same syntax also works on *strings*, but there it is rune-based (UTF-8) — that is, it cuts by characters, not bytes; we will cover this in Chapter 5. When you slice a `range(...)`, the result is a lazy RANGE value (it does not expand all elements into memory).

> **Warning** Cobra has no slice *assignment*. An expression like `xs[1:3] = [9, 9]` is not valid. If you want to change a range, proceed either with individual index assignments (`xs[1] = 9`) or by building a new list and joining with `+` (`xs[:1] + [9, 9] + xs[3:]`).

4.8 Dictionaries (dict)

A dictionary is a collection that maps *keys* to *values*. It is written with `key: value` pairs inside curly braces.

```
let kullanici = {"ad": "cobra", "yas": 3}
print(kullanici["ad"])   # cobra
print(kullanici["yas"])  # 3
```

Keys do not have to be strings. In Cobra, keys can be **strings, integers, booleans, or null**.

```
let d = {"ad": "cobra", 1: "one", true: "evet", null: "yok"}
print(d[1])      # one
print(d[true])   # evet
print(d[null])   # yok
```

Dictionaries **preserve insertion order**: the order in which you write or add the pairs stays the same when you iterate over them.

4.8.1 Reading, adding, updating

The expression `d[k]` reads the value of a key. `d[k] = v` adds the key (if absent) or updates its value (if present).

```
let yaslar = {"ali": 30, "veli": 25}
yaslar["ayse"] = 35        # adds a new key
yaslar["ali"] = 31         # updates an existing key
print(yaslar)              # {ali: 31, veli: 25, ayse: 35}
```

Augmented assignment also works on dictionary values:

```
let puan = {"x": 10}
puan["x"] += 5
print(puan["x"])   # 15
```

> **Warning** Trying to read a key that does not exist with `d[k]` is an *error*; it does not return `null`. If you are not sure the key exists, check first with `has`, or use `get` for safe reading (we will see both below).

4.9 Dictionary methods

Dictionaries also have both a method form called with a dot and a built-in function form.

Method	Built-in form	What it does
`d.keys()`	`keys(d)`	List of the keys
`d.values()`	`values(d)`	List of the values
`d.items()`	—	List of `[key, value]` pairs
`d.has(k)`	`has(d, k)`	Is the key present? `true`/`false`
`d.get(k, default?)`	—	The value, or the default if absent (or `null`)
`d.del(k)`	`del(d, k)`	Deletes the key

```
let yaslar = {"ali": 30, "veli": 25}

print(yaslar.keys())     # [ali, veli]
print(yaslar.values())   # [30, 25]
print(yaslar.has("ali")) # true
print(yaslar.has("xyz")) # false
```

The `get` method enables safe reading by returning a default instead of raising an error for a key that does not exist. If you do not provide a default, `null` is returned for the missing key.

```
let yaslar = {"ali": 30}
print(yaslar.get("ali", 0))   # 30
print(yaslar.get("zzz", 0))   # 0   (no such key, the default was returned)
print(yaslar.get("zzz"))      # null (no default was given)
```

`del` removes a key:

```
let yaslar = {"ali": 30, "veli": 25}
yaslar.del("veli")
print(yaslar)   # {ali: 30}
```

The built-in function forms for `keys`, `values`, and `has` do the same job:

```
let d = {"a": 1, "b": 2}
print(keys(d))      # [a, b]
print(values(d))    # [1, 2]
print(has(d, "a"))  # true
del(d, "a")
print(d)            # {b: 2}
```

> **Note** Dictionaries, like lists, are reference types. `let b = a` makes an alias to the same dictionary; a key added through one is also visible through the other. If you need an independent copy, you have to build a new dictionary by iterating over the keys (there is no dictionary equivalent of list slicing).

4.10 Iterating over a dictionary

When you iterate directly over a dictionary with `for ... in`, the loop variable takes the *keys* (in insertion order):

```
let yaslar = {"ali": 30, "veli": 25}
for ad in yaslar
    print(ad + " -> " + str(yaslar[ad]))
end
# ali -> 30
# veli -> 25
```

When you want to access both the key and the value at once, you use `items()`. `items()` gives you a list made of two-element lists in the form `[key, value]`.

```
for cift in yaslar.items()
    print(cift[0], cift[1])
end
# ali 30
# veli 25
```

> **Warning** Cobra has no *unpacking* (multiple destructuring). That is, a form like `for anahtar, deger in d.items()` is not valid. Instead you take the pair into a single variable and index it as `cift[0]` (the key) and `cift[1]` (the value). The same restriction applies to the outputs of `zip` and `enumerate`.

4.11 What is missing: comprehensions and sets

Two features that those coming from other languages often look for are *deliberately* not present in Cobra:

- There are no **list/dictionary comprehensions**. That is, a syntax like `[x * x for x in xs]` is not valid.
- There is no **set** data type.

This is not a shortcoming but a choice to preserve simplicity. Instead of comprehensions, you use the higher-order functions `map` and `filter`, or a plain `for` loop:

```
def kare(n)
    return n * n
end

let xs = [1, 2, 3, 4]
print(map(kare, xs))   # [1, 4, 9, 16]

# or with a loop:
let kareler = []
for x in xs
    kareler.push(x * x)
end
print(kareler)         # [1, 4, 9, 16]
```

When you need set behavior (membership without duplicates), a common pattern is to use a dictionary as a set: you keep the keys as members and put `true` as the value. Membership checks are done with `has`. We will cover `map`, `filter`, `reduce`, and their friends in detail in Chapter 6.

4.12 Putting it together: a small word counter

Let us bring together what we have learned in a single example. A small program that counts the words in a text, using lists and dictionaries together:

```
def kelime_say(metin)
    let sayim = {}
    for kelime in metin.split(" ")
        let k = kelime.strip().lower()
        if k == ""
            continue
        end
        if sayim.has(k)
            sayim[k] += 1
        else
            sayim[k] = 1
        end
    end
    return sayim
end

let sonuc = kelime_say("kedi kopek kedi kus kedi")
for cift in sonuc.items()
    print(cift[0] + ": " + str(cift[1]))
end
# kedi: 3
# kopek: 1
# kus: 1
```

In this program we see every tool: we build a dictionary empty from the start, check the presence of a key with `has`, increment the counter with `+=`, iterate with `items()`, and reach the key and value with `cift[0]` and `cift[1]`. We could also have read an existing key safely with `get`; for example `sayim[k] = sayim.get(k, 0) + 1` does the same job in a single line.

4.13 Summary

- **Lists** are written with square brackets (`[1, 2, "x"]`), are heterogeneous, and are indexed from zero. A negative index counts from the end (`l[-1]` is the last element). Use `l[0] = v` for in-place assignment and `+` for concatenation; `==` compares deeply.
- **Lists are reference types.** `let b = a` creates an alias, not a copy — a change made through one is also visible through the other. For an independent shallow copy, use `a[:]`.
- **List methods:** `push`, `pop`, `reverse`, `sort` modify in place; `contains`, `find` (`-1` if absent), `count` read the list. For `push` and `pop` there are also the built-in forms `push(xs, v)` / `pop(xs)`.
- **Slicing** is of the form `seq[low:high:step]`; each piece can be omitted (`xs[:2]`, `xs[1:]`, `xs[:]`), negative indices count from the end, a negative step reverses (`xs[::-1]`), and out-of-bounds values are *clamped* without error. On lists it returns a new list, on strings a rune-based result (Chapter 5), and on ranges a lazy RANGE. **There is no slice assignment.**
- **Dictionaries** are written with curly braces (`{"ad": "cobra", 1: "one"}`); keys can be strings, ints, booleans, or null; insertion order is preserved. `d[k]` reads (a missing key is an *error* — check with `has` or use `get`), `d[k] = v` adds/updates.
- **Dictionary methods:** `keys`, `values`, `items`, `has`, `get(k, default?)`, `del`; most also have built-in forms (`keys(d)`, `has(d, k)`, `del(d, k)`).
- **Iteration:** `for ... in d` gives the keys; for key+value use `for cift in d.items()`. Cobra has **no unpacking**, so index as `cift[0]` (the key) and `cift[1]` (the value).
- Cobra has **no comprehensions and no sets.** Instead use `map`/`filter` (Chapter 6) or a plain `for` loop; for set behavior a dictionary whose values are `true` is a common pattern.

Chapter 5

Working with Strings

The place where a program talks to the world is most often text. A line coming from the user, the contents of a file, the body of an HTTP response, everything you print to the screen... All of these arrive as *strings* and leave as strings. In this chapter we will see how you build, split, search, and transform text in Cobra.

Cobra's string model is made up of a few small but sharp rules. Let us state the most important one up front: **strings are immutable**. You cannot change a string in place; every "change" actually produces a *new* string. Although this may look restrictive at first, it simplifies debugging: you can be sure that a string you hold will not be changed behind your back by someone else.

Second, in Cobra there are **no f-strings or string interpolation**. The embedded-expression forms found in some languages, like `f"x={x}"` or `` `x=${x}` ``, do not work here. You build text by hand with the `+` operator and the `str(...)` converter. You will see this pattern over and over throughout this chapter.

5.1 Building strings: + and `str(...)`

The most basic tool for building text is the `+` operator. It joins two strings side by side:

```
let ad = "Ada"
let soyad = "Lovelace"
print(ad + " " + soyad)
# Ada Lovelace
```

But `+` only joins a string with a string. You cannot add a number directly; you first have to convert it to a string with `str(...)`:

```
let yas = 36
print("Yas: " + str(yas))
# Yas: 36
```

`str(...)` converts every value into readable text: integers, decimals, boolean values, even lists and dictionaries.

```
print(str(3) + str(14))     # 314  (string concatenation, NOT addition!)
print(str(3.14))            # 3.14
print(str(true))            # true
print(str([1, 2, 3]))       # [1, 2, 3]
```

> **Warning:** The result of `str(3) + str(14)` is `"314"`, not `17`. `+` performs *concatenation* on two strings, not arithmetic addition. If you want numeric addition, remove the `str(...)` calls: `3 + 14`.

The absence of interpolation means that when you bring multiple values together you have to write `+` and `str(...)` chained one after another:

```
let urun = "kalem"
let adet = 12
let fiyat = 5
print("Siparis: " + str(adet) + " adet " + urun + ", toplam " + str(adet * fiyat) + " TL")
# Siparis: 12 adet kalem, toplam 60 TL
```

> **Tip:** For multi-part messages, `print` also takes multiple comma-separated arguments and puts a space between them: `print("Toplam", adet * fiyat, "TL")`. This does not *produce* a single string, but it is handy for printing to the screen.

5.2 Escape sequences

You add characters that you cannot write directly inside a string literal with *escape sequences*. In Cobra, strings are written only with double quotes, so the escape you will need most often is the double quote itself.

Sequence	Meaning
`\n`	line ending (newline)
`\t`	tab
`\"`	double-quote character
`\\`	backslash character

```
print("satir1\nsatir2")
# satir1
# satir2

print("Ad:\tAda")
# Ad:   Ada

print("o dedi \"selam\"")
# o dedi "selam"

print("yol: C:\\Users")
# yol: C:\Users
```

> **Note:** `\n` is a *single* character (line ending), not two characters. If you want to see an actual backslash inside a string, you must write it as `\\`; otherwise Cobra will try to form an escape sequence together with the next character.

5.3 String methods

Cobra's strings come with a rich set of methods. They are all used with a dot call (`s.upper()`), and none of them change the original string — by the rule of immutability they all return a *new* value.

5.3.1 Upper/lowercase: `upper`, `lower`

```
print("merhaba".upper())   # MERHABA
print("MERHABA".lower())   # merhaba
```

5.3.2 Whitespace trimming: `strip`, `lstrip`, `rstrip`

`strip` trims leading and trailing whitespace; `lstrip` only the left, `rstrip` only the right.

```
print("  bosluk  ".strip() + "|")    # bosluk|
print("   sol".lstrip() + "|")       # sol|
print("sag   ".rstrip() + "|")       # sag|
```

> **Warning:** These three methods take no arguments; they always trim *whitespace* characters (space, tab, newline). A call like `"xxabc".lstrip("x")` raises an error. If you want to strip a specific character, use `replace`.

5.3.3 Splitting and joining: `split`, `join`

`split` breaks a string into parts and returns a list. If you give a separator, it splits on it:

```
print("a,b,c".split(","))      # ["a", "b", "c"]
print("a,b,,c".split(","))     # ["a", "b", "", "c"]
```

If you call it with no arguments, `split` breaks on whitespace and cleverly swallows multiple spaces in between — this is the most practical way to split text into "words":

```
print("Ali   Veli  Deli".split())   # ["Ali", "Veli", "Deli"]
print(len("a b   c".split()))       # 3
```

`join` is the reverse of this: it pulls a list into a single string, putting a separator in between. The separator is the string you call the method on:

```
print(", ".join(["x", "y", "z"]))   # x, y, z
print("-".join(["2026", "06", "27"]))   # 2026-06-27
```

5.3.4 Replacing: `replace`

Replaces *all* occurrences of a substring with another piece of text:

```
print("aaa".replace("a", "b"))           # bbb
print("yol/ile/yol".replace("/", "."))   # yol.ile.yol
```

5.3.5 Searching: `contains`, `startswith`, `endswith`, `find`, `count`

```
print("merhaba".contains("rha"))     # true
print("merhaba".startswith("mer"))   # true
print("merhaba".endswith("aba"))     # true
print("merhaba".find("h"))           # 3   (index of the first occurrence)
print("merhaba".find("z"))           # -1  (not found)
print("banana".count("a"))           # 3   (how many times it occurs)
```

> **Note:** When `find` does not find something it does not raise an error, it returns `-1`. For this reason the expression `if s.find(x) != -1` is the classic way to write the question "is x inside this string?" — but most of the time `s.contains(x)` directly is more readable.

5.3.6 Chaining methods

Since each method returns a new string (or list), you can line up the calls one after another. They read from left to right:

```
print("  a,b  ".strip().split(","))
# ["a", "b"]

print("  Ali, Veli , Deli  ".strip().split(","))
# ["Ali", " Veli ", " Deli"]
```

> **Tip:** In the second example, notice that the spaces after the commas are still sitting inside the parts. To clean those up too, you have to apply `strip` to each part one by one — for example in a `for` loop or with `map`.

5.4 Indexing and slicing

You can access a single position of a string with square brackets:

```
print("merhaba"[0])   # m
print("merhaba"[3])   # h
```

On ASCII text this works flawlessly. But in Cobra strings are stored underneath as UTF-8 byte sequences, and **single-position indexing is BYTE-based, not rune-based.** Because Turkish characters (ş, ı, ğ, ö, ü, ç, İ...) take up more than one byte in UTF-8, landing on a multi-byte character with `[i]` gives a corrupted result:

```
let s = "şehir"
print(s[0])   # Å   (the FIRST byte of ş — corrupted!)
```

This is exactly where **slicing** comes in. Slicing is rune-based, that is, UTF-8 safe, and gives the correct result on Turkish characters:

```
let s = "şehir"
print(s[0:1])   # ş   (correct — rune-based slice)
print(s[1:3])   # eh
```

The same UTF-8 safety also applies to reversing and slicing from the end:

```
print("şığöüç"[::-1])   # çüöğış   (reversed, correct)
print("Türkçe"[::-1])   # eçkrüT
print("Türkçe"[-3:])    # kçe      (the last three runes)
```

The slicing syntax is of the form `s[low:high:step]`; each piece can be omitted, negative indices count from the end, a negative step reverses, and out-of-bounds values are clamped without error — just like with lists (see Chapter 4).

> **Warning:** `len` gives the number of **bytes** in a string, not the number of runes (visible characters). For multi-byte characters the two differ:
>
> ```
> print(len("abc"))   # 3   (3 ASCII characters = 3 bytes)
> print(len("şğ"))    # 4   (2 characters but 4 bytes!)
> ```
>
> If you need to count the number of visible characters, the safe way to split text into runes is to work with the `re` module or with slicing.

> **Tip:** A practical rule: **when you need to pull a single character from Turkish (or any Unicode) text, use the slice `s[i:i+1]` instead of `s[i]`.** If you need complex positioning, switch to the `re` module. Use raw byte indexing only when you are sure the text is entirely ASCII.

5.5 Code points: `chr` and `ord`

Sometimes you need a character's Unicode code point (its number), or the exact reverse, to get a character from a number. There are two built-in functions for this:

- `chr(n)` — gives the single-character string corresponding to code point `n`.
- `ord(s)` — gives the code point of the first character of the string `s`.

```
print(chr(65))    # A
print(chr(351))   # ş
print(ord("A"))   # 65
print(ord("ş"))   # 351
```

These are the reverse of each other: `chr(ord("ş"))` gives `"ş"` again. They are useful for doing character arithmetic (for example shifting a letter through the alphabet) or producing control characters:

```
# the first five letters starting from 'a'
let harfler = ""
for i in range(5)
    harfler = harfler + chr(ord("a") + i)
end
print(harfler)   # abcde
```

> **Note:** `chr`/`ord` work with code *points* (runes), not bytes. For this reason they give the correct number even for multi-byte Turkish characters — unlike raw byte indexing.

5.6 A bridge to regular expressions: the `re` module

For searching with fixed text, the string methods are enough; but when you need to work with *patterns* (a sequence of digits, an email format, multiple separators…), the `re` module comes into play. Here we will only get a taste; the details are in **Chapter 11**.

```
import "re"

print(re.matches("[0-9]+", "abc123"))           # true   (does the pattern match?)
print(re.find("[0-9]+", "abc123def456"))         # 123    (the first match)
print(re.find_all("[0-9]+", "abc123def456"))     # ["123", "456"]
print(re.replace("[aeiou]", "merhaba", "*"))     # m*rh*b*
print(re.split(",\\s*", "a, b,c,  d"))           # ["a", "b", "c", "d"]
```

The `re.split` in the last example splits on the pattern "a comma followed by zero or more spaces," cleaning up the variable spacing that `split(",")` could not handle.

> **Warning:** Cobra's `re` module uses RE2 syntax. RE2 always runs in linear time (there is no catastrophic backtracking), but in return it **does not support lookaround (forward/backward assertions) or backreferences.** Patterns like `(?=...)` or `\1` that you are used to from other languages do not work here.

> **Tip:** Remember that you have to write the `\` characters in your pattern as `\\` in a Cobra string literal. The `,\\s*` above actually reaches the regular-expression engine as `,\s*`.

5.7 Summary

- Strings are **immutable**: every method returns a new value without breaking the original.
- There are **no f-strings / interpolation**; you build text by hand with `+` and `str(...)` (`"deger=" + str(x)`).
- `+` on strings is *concatenation*; convert numbers with `str(...)` before adding them.
- Escape sequences: `\n`, `\t`, `\"`, `\\`.
- String methods: `upper`, `lower`, `strip`, `lstrip`/`rstrip` (no arguments, whitespace only), `split` (splits on whitespace when given no argument), `join`, `replace`, `contains`, `startswith`, `endswith`, `find` (index or `-1`), `count`. They can all be chained: `"  a,b  ".strip().split(",")`.
- Single-position indexing (`s[i]`) is **byte-based** and corrupts multi-byte Turkish characters; **slicing (`s[1:3]`, `s[::-1]`) is rune-based and UTF-8 safe.** For a single character in Unicode text use `s[i:i+1]`.
- `len` counts bytes; for the number of visible characters use slicing/`re`.
- `chr(n)` ↔ `ord(s)` convert between code point and character (rune-based, UTF-8 correct).
- For pattern-based searching/replacing/splitting, the `re` module (RE2; no lookaround or backreferences). Details in Chapter 11.

Chapter 6

Functions

Functions are the building blocks with which you divide your programs into reusable pieces. In Cobra, functions are not merely a "code grouping" tool; they are fully fledged **values**: you can assign them to a variable, pass them as an argument to another function, return them from a function, and keep them alive by "closing over" their surroundings (closure).

In this chapter we will examine the rules of function definition, returning values, what it means to be a first-class value, closures, the features Cobra deliberately *leaves out* (no default values, no `lambda`), the built-in higher-order functions, and finally decorators. Cobra's philosophy is simple syntax; that is why functions, too, derive their power not from a large number of features but from the consistent behavior of a small number of them.

6.1 Defining a Function: `def ... end`

A function begins with the `def` keyword, continues with its name and the parameters in parentheses, and closes with `end`. Cobra closes blocks not with indentation or curly braces, but with **`end`**.

```
def add(a, b)
    return a + b
end

print(add(2, 3))   # 5
```

There is a critical rule here: **the function body must be on the lines AFTER the `def` line.** Cobra has no inline body. That is, you cannot cram everything onto a single line like this:

```
# WRONG — the body cannot be on the def line
def add(a, b) return a + b end
```

This rule is the natural consequence of the language's principle that "every statement ends on one line, and blocks are closed with `end`." The body always sits on its own line(s).

> **Note:** The `func` keyword is exactly synonymous with `def`. If you like, you can write `func add(a, b)`; the behavior is identical. In this book we will use `def` for consistency.

To call a function, you write the arguments in parentheses after its name. The parentheses are mandatory even when there are no arguments:

```
def greet()
    print("hello")
end

greet()   # hello
```

6.2 Returning a Value with `return`

A function gives a value back to its caller with the `return` statement. The moment `return` runs, the function ends; the lines after it do not execute.

```
def absolute(n)
    if n < 0
        return -n
    end
    return n
end

print(absolute(-7))   # 7
print(absolute(3))    # 3
```

6.2.1 Without `return`: `null`

In Cobra, a function without a `return` automatically returns `null`. This means that functions which "return nothing" actually return `null`.

```
def log(msg)
    print("[log] " + msg)
end

let result = log("started")   # [log] started
print(result)                 # null
```

6.2.2 Bare `return`

You can write just `return` without specifying a value. This is the way to exit a function early, and it likewise returns `null`:

```
def process(items)
    if len(items) == 0
        return            # early exit; returns null
    end
    print("processing: " + str(len(items)) + " items")
end

process([])          # (prints nothing)
process([1, 2, 3])   # processing: 3 items
```

> **Tip:** Using a bare `return` as a "guard clause" makes code readable: you filter out invalid cases at the very start and free the main logic from indentation depth.

6.3 Functions Are First-Class Values

In Cobra a function, just like a number or a string, is a value. You can assign it to a variable:

```
def add(a, b)
    return a + b
end

let f = add        # a function is a value — NO parentheses
print(f(2, 3))     # 5
print(type(add))   # FUNCTION
```

Here `add` (without parentheses) refers to the function itself; `add(2, 3)`, on the other hand, *calls* it. This distinction is the basis for passing functions to other functions. You can also store functions in lists or dictionaries:

```
def add(a, b) return a + b end
def sub(a, b) return a - b end

let ops = {"+": add, "-": sub}
print(ops["+"](10, 4))   # 14
print(ops["-"](10, 4))   # 6
```

> **Note:** Spellings like `def add(a, b) return a + b end` above may look in this book like a shorthand merely to demonstrate the *dictionary rule*; but remember: when writing real code the body must be on its own line. Its appearing on a single line here is document formatting — in your own programs move the body to the next line.

6.4 Closures

Cobra functions "close over" the scope in which they are defined — that is, a function defined nested inside another continues to access the enclosing function's variables, even after that function has already returned. This is called a **closure**.

```
def adder(n)
    def add(x)
        return x + n     # captures the outer n
    end
    return add
end

let add5 = adder(5)
let add10 = adder(10)

print(add5(10))    # 15
print(add10(10))   # 20
print(add5(1))     # 6
```

Even after the `adder(5)` call has returned, the `add` function it returned remembers the value `n = 5`. Each `adder` call captures its own independent `n`; that is why `add5` and `add10` do not affect each other. Closures are a powerful way to produce functions that carry state:

```
def counter()
    let n = 0
    def next()
        n = n + 1
        return n
    end
    return next
end

let count = counter()
print(count())   # 1
print(count())   # 2
print(count())   # 3
```

Here `next` not only reads the outer `n`, it also *modifies* it. The captured variable is a shared, living binding.

6.5 Arguments: Exactly As Many As Declared

Cobra functions take exactly as many **positional** arguments as they are declared with — neither one fewer, nor one more. On this point the language is deliberately strict, and it does **not** offer the following:

- **No default values.** There is no such thing as `def f(a, b = 10)`.
- **No keyword arguments.** You cannot call `f(b = 3)`.
- **No variable number of arguments.** User functions have no `*args` or `**kwargs`.

```
def greet(name, greeting)
    return greeting + ", " + name + "!"
end

print(greet("Ada", "Hello"))   # Hello, Ada!

# greet("Ada")  -> ERROR: the function expects two arguments
```

> **Warning:** Calling a function with the wrong number of arguments is an error. If you want "optional parameter" behavior, you must manage it explicitly: either write two functions with different names, or take a single parameter as a dict and fill in the missing values with `get(key, default)`.

A common pattern for behavior resembling default values is to pass a configuration dictionary:

```
def connect(options)
    let host = options.get("host", "localhost")
    let port = options.get("port", 8080)
    return host + ":" + str(port)
end

print(connect({}))                       # localhost:8080
print(connect({"port": 9090}))           # localhost:9090
print(connect({"host": "db", "port": 5432}))   # db:5432
```

6.6 No `lambda`: Use a Nested `def`

Many languages have a `lambda` syntax for small, anonymous functions. Cobra has **no `lambda`.** Instead, when you need an "inline" function, you write a nested `def` and return it. The `adder` pattern we just saw is exactly this:

```
def make_multiplier(factor)
    def multiply(x)
        return x * factor
    end
    return multiply
end

let triple = make_multiplier(3)
print(triple(7))   # 21
```

This approach may look a little longer, but because it always gives the function a name, it gains readability in stack traces and error messages. It is a small price paid for the sake of simplicity.

6.7 Higher-Order Built-in Functions

Because functions are values, you can write functions that take them as arguments. These are called **higher-order** functions. Cobra provides the most frequently needed ones built in. They all share one property: **they are all EAGER**, that is, they return the result not as a lazy iterator but as an immediately constructed **list**.

6.7.1 `map(fn, list)`

Applies `fn` to each element of the list and produces a new list from the results:

```
def square(n)
    return n * n
end

print(map(square, [1, 2, 3, 4]))   # [1, 4, 9, 16]
```

6.7.2 `filter(fn, list)`

Keeps the elements for which `fn` returns truthy:

```
def is_even(n)
    return n % 2 == 0
end

print(filter(is_even, [1, 2, 3, 4, 5, 6]))   # [2, 4, 6]
```

6.7.3 `reduce(fn, list, init?)`

Reduces the list to a single value by folding it **from the left**. `fn` is called at each step in the form `fn(acc, elem)`: `acc` is the value accumulated so far, `elem` is the next element. The optional `init` provides the starting value; if it is not given, the first element is used as the start.

```
def add(acc, x)
    return acc + x
end

print(reduce(add, [1, 2, 3, 4]))        # 10   (start = first element)
print(reduce(add, [1, 2, 3], 100))      # 106  (start = 100)
```

6.7.4 `zip(list, ...)`

Pairs up multiple lists element by element; each is cut off at the shortest list. The result is a list of sublists:

```
print(zip([1, 2, 3], ["a", "b", "c"]))   # [[1, "a"], [2, "b"], [3, "c"]]
print(zip([1, 2], ["a", "b", "c"]))      # [[1, "a"], [2, "b"]]  (by the shortest)
```

6.7.5 `enumerate(list, start?)`

Gives each element together with its index as `[index, value]` pairs. The optional `start` determines where the index begins:

```
for pair in enumerate(["x", "y", "z"])
    print(str(pair[0]) + " -> " + pair[1])
end
# 0 -> x
# 1 -> y
# 2 -> z

print(enumerate(["a", "b"], 1))   # [[1, "a"], [2, "b"]]
```

> **Note:** Cobra has no multiple assignment or unpacking; that is why you cannot write `for i, v in enumerate(...)`. Index the pairs with `pair[0]` and `pair[1]`.

6.7.6 `any(list)` and `all(list)`

`any` returns `true` if there is at least one truthy value in the list; `all` returns `true` if all the values are truthy:

```
print(any([0, 0, 1]))   # true
print(any([0, 0, 0]))   # false
print(all([1, 2, 3]))   # true
print(all([1, 0, 3]))   # false
```

> **Tip:** Cobra has no list comprehension. Instead of `[x * 2 for x in xs]` use `map`; instead of `[x for x in xs if cond]` use `filter`. If you need to combine the two, feed the result of `filter` into `map`: `map(f, filter(g, xs))`.

6.7.7 Parallel Versions: `pmap` and `pfilter`

Cobra has **no GIL**, which means true multi-core parallelism is possible. `pmap` and `pfilter`, the parallel siblings of `map` and `filter`, distribute the work across CPU cores. The call form is the same as `map`/`filter`; the results preserve the input order, and the first error propagates as-is:

```
def heavy(n)
    return n * n
end

print(pmap(heavy, [1, 2, 3, 4]))   # [1, 4, 9, 16]  (parallel, order preserved)
```

On CPU-intensive work, `pmap` gives a speedup with no extra syntax whatsoever. We will cover the details of parallelism — `async def`, `await`, `Mutex`, `Channel` — in depth in Chapter 10.

6.8 Decorators

A **decorator** is a wrapper that transforms or registers a function. It is applied by writing one or more `@expr` lines immediately above a `def`.

In its simplest form, `@d`, it applies this rule:

```
@d
def name(...) ... end
```

This is exactly equivalent to:

```
name = d(name)
```

That is, Cobra defines the function, then passes it to `d` and rebinds the value returned by `d` to the same name. As an example, let us write a simple decorator that times a function:

```
import "time"

def timed(fn)
    def wrapper(x)
        let start = time.clock()
        let result = fn(x)
        let elapsed = time.clock() - start
        print("time: " + str(elapsed) + "s")
        return result
    end
    return wrapper
end

@timed
def slow_square(n)
    return n * n
end

print(slow_square(9))
# time: 0.000...s
# 81
```

Here `slow_square`, after being defined, has been replaced with the `wrapper` returned by `timed(slow_square)`. Now the call `slow_square(9)` actually runs `wrapper(9)`.

6.8.1 Decorators with Arguments: `@d(args)`

If you want to give a decorator configuration, you use the `@d(args)` form. In this case Cobra first calls `d(args)`; the result (a function that is the actual decorator) is applied to the `def`. That is, `@d(args)` is equivalent to: `name = d(args)(name)`.

```
def repeat(times)
    def decorator(fn)
        def wrapper(x)
            let result = null
            let i = 0
            while i < times
                result = fn(x)
                i = i + 1
            end
            return result
        end
        return wrapper
    end
    return decorator
end

@repeat(3)
def shout(msg)
    print(msg + "!")
    return msg
end

shout("hello")
# hello!
# hello!
# hello!
```

`repeat(3)` returns `decorator`; `decorator` in turn wraps `shout`.

6.8.2 Stacking

You can place multiple decorators above a `def`. These are **stacked**, and the order of application follows this rule: **the one closest to the `def` is applied first.** That is:

```
@a
@b
def f(...) ... end
```

is equivalent to: `f = a(b(f))` — first `b`, then `a`. The innermost wrapping is the decorator closest to the `def`.

```
def upper(fn)
    def wrapper(s)
```

```
        return fn(s).upper()
    end
    return wrapper
end

def exclaim(fn)
    def wrapper(s)
        return fn(s) + "!"
    end
    return wrapper
end

@upper
@exclaim
def echo(s)
    return s
end

print(echo("hello"))   # HELLO!
```

Here `exclaim` (the closest) is applied first, producing `"hello!"`; then `upper` wraps it to make `"HELLO!"`.

6.8.3 Decorators Also Work with `async def`

Decorators can also be applied to asynchronous functions. When you place a decorator above an `async def`, the wrapping rules apply exactly the same way (the details of concurrency are in Chapter 10).

6.8.4 The Registry Pattern

One of the most frequently encountered uses of decorators is to **register** functions into a registry table — without wrapping, returning the function as-is. Web routing is the classic example of this:

```
let routes = {}

def route(path)
    def register(fn)
        routes[path] = fn
        return fn          # return the function unchanged
    end
    return register
end

@route("/home")
def home(req)
    return "welcome"
end

@route("/about")
def about(req)
    return "about us"
end
```

```
# Now routes contains the path -> function mapping:
print(routes["/home"]("request"))    # welcome
print(routes["/about"]("request"))   # about us
```

Here `route("/home")` returns `register`; `register` in turn adds `home` to the `routes` dictionary and gives it back **unchanged**. As a result, `home` is still a normal function, but it has also been entered into the `routes` table. This pattern is the idiomatic way for an HTTP server to bind request paths to their handlers.

> **Tip:** A decorator is not required to *wrap* the function. As in the registry pattern, a decorator can simply register the function somewhere and return it as-is. The only rule that matters: a decorator must return a value to be bound as the new value of the `def`.

6.9 Summary

In this chapter we saw every facet of Cobra functions:

- Functions begin with `def name(parameters)`, close with `end`, and the **body is always on the lines after the `def` line** — there is no inline body. `func` is synonymous with `def`.
- `return` returns a value and terminates the function. A function without a `return` returns `null`; a bare `return` also exits early and returns `null`.
- Functions are **first-class values**: they can be assigned to variables, stored in lists/dictionaries, passed, and returned.
- Functions **close over** the scope in which they are defined (closure); a function defined nested inside keeps the outer variables alive and can modify them.
- Functions take **exactly as many positional arguments as declared**. There are **no** default values, keyword arguments, or `*args`/`**kwargs`.
- There is **no `lambda`**; for an inline function, write a nested `def` and return it.
- The higher-order built-ins — `map`, `filter`, `reduce`, `zip`, `enumerate`, `any`, `all` — **are all EAGER and return a list**. `reduce` folds from the left (`fn(acc, elem)`). The parallel versions `pmap` and `pfilter` offer true, GIL-free parallelism.
- **Decorators** are the `@expr` lines above a `def`: `@d` → `name = d(name)`; `@d(args)` first calls `d(args)`. They stack (the closest is applied first) and also work with `async def`. The registry pattern (`@route("/path")`) is the idiomatic way to register functions without wrapping them.

In the next chapter, we will move on to **structs** (simple classes), the way to package state and behavior together — there you will see how functions live as methods.

Chapter 7

Object-Oriented Programming

In the previous chapter we saw how functions carry state by means of closures. But in most programs, state and behavior live together: a bank account has a balance *and* the `deposit`/`withdraw` operations that change that balance; a geometric shape has its dimensions *and* an `area` computation. The way to gather these two under a single package is **object-oriented programming**.

Cobra solves this with a single, simple building block: **`struct`**. Cobra's philosophy is for a small number of concepts to behave consistently; that is why here, instead of dozens of keywords, you will find a single core idea — a type that holds state (fields) and behavior (methods) together. In this chapter we will examine the constructor (`init`) and the implicit `self`, field access, instance methods, properties (`get`/`set`), class constants (`const`), static methods (`static def`), single inheritance and `super`, `record` for immutable value objects, and the runtime contracts (`contract` + `implements`) that take the place of interfaces. We will also explicitly address the features Cobra deliberately *leaves out* — no operator overloading, no multiple inheritance, no `isinstance`.

7.1 `struct`, `class`, `record`: Three Keywords

Cobra has three keywords for defining a type, and the first two are exactly **synonymous**:

- **`struct`** — defines a simple class. This is the main keyword of this book.
- **`class`** — exactly the same as `struct`. It is a convenience for those coming from other languages; there is no difference in behavior.
- **`record`** — a special type that produces **immutable** instances and uses **value-based** equality. We will cover this separately at the end of the chapter.

```
class Counter
    def init()
        self.value = 0
    end
    def increment()
        self.value += 1
        return self.value
    end
end

let c = Counter()
print(c.increment())   # 1
```

```
print(c.increment())   # 2
print(type(c))         # Counter
```

> **Note:** The choice between `class` and `struct` is purely a matter of style. This book uses `struct` for consistency; if your team prefers `class`, nothing changes.

7.2 Our First Struct: `struct` and `init`

A `struct` consists of the `struct` keyword, a type name, and a body closed with `end`. Inside the body there is a special method that is its constructor: **`init`**. `init` runs automatically when you call the type like a function (`BankAccount("Ada", 100)`) and initializes the new instance.

In Cobra, the body of every method has an **implicit `self`** that represents the instance being operated on. You do not write it as a parameter; it is always ready. You access fields with `self.field` and write them with `self.field = value`.

```
struct BankAccount
    def init(owner, balance)
        self.owner = owner          # fields are defined on self
        self.balance = balance
    end

    def deposit(amount)
        self.balance = self.balance + amount
        return self.balance
    end

    def withdraw(amount)
        if amount > self.balance
            throw "insufficient balance"
        end
        self.balance = self.balance - amount
        return self.balance
    end
end

let acc = BankAccount("Ada", 100)
print(acc.deposit(50))    # 150
print(acc.withdraw(30))   # 120
print(acc.owner)          # Ada
```

The call `BankAccount("Ada", 100)` runs `init` with `owner = "Ada"` and `balance = 100`; a new **instance** is returned. The `deposit` and `withdraw` methods read and update `self.balance` — the state lives inside the instance.

> **Tip:** `init` itself does not `return`; Cobra automatically returns the initialized instance. Its only job is to set up the fields of `self`.

7.3 Fields: Reading, Writing, and Adding New Fields

You access an instance's fields with the dot: `acc.balance` reads, `acc.balance = 0` writes. In Cobra, fields do not have to be "declared" beforehand somewhere — you can **add a new field to an instance from the**

outside, at any time you like:

```
let acc = BankAccount("Ada", 100)

acc.note = "VIP customer"    # a new field not present in init
print(acc.note)              # VIP customer
```

When you print an instance, Cobra displays it in the form `TypeName{field: value, ...}`; string values appear in quotes. `type(instance)`, on the other hand, gives the type's name as a string:

```
print(acc)         # BankAccount{owner: "Ada", balance: 100, note: "VIP customer"}
print(type(acc))   # BankAccount
```

> **Warning:** Instances are compared **by identity** (reference equality): two separate `BankAccounts` with the same fields are not equal under ==. If you want value-based equality, use `record` (at the end of the chapter).

7.4 Instance Methods

Every `def` other than `init` becomes an **instance method** of the type. You call them on an instance, with parentheses: `acc.deposit(50)`. The method body can always access the instance's fields and other methods through `self`. Augmented assignment (+=) also works on fields, so writing `self.balance += amount` is the short form of `self.balance = self.balance + amount`:

```
struct BankAccount
    def init(owner, balance)
        self.owner = owner
        self.balance = balance
    end

    def deposit(amount)
        self.balance += amount       # self.balance = self.balance + amount
        return self.balance
    end

    def transfer_to(other, amount)
        self.withdraw(amount)        # call its own method through self
        other.deposit(amount)        # call another instance's method
    end

    def withdraw(amount)
        if amount > self.balance
            throw "insufficient balance"
        end
        self.balance -= amount
        return self.balance
    end
end
```

> **Note:** Parentheses are **mandatory** to call a method: `acc.deposit(50)`. If you write `acc.deposit` without parentheses, it does *not* call the method; you obtain the method value itself. (The `get` properties we will see shortly are the sole exception to this rule.)

7.5 Properties: Computed Reads with `get`

Sometimes a value is not a stored field but a result **derived** from other fields. Instead of obtaining it each time with a method call (`item.subtotal()`), you want to read it as if it were a normal field (`item.subtotal`). This is exactly what `get` provides: it defines a property whose body runs with `self` but is read **without parentheses**.

```
struct LineItem
    def init(name, unit_price, quantity)
        self.name = name
        self.unit_price = unit_price
        self.quantity = quantity
    end

    get subtotal()                  # property: item.subtotal (NO parentheses)
        return self.unit_price * self.quantity
    end
end

let item = LineItem("Keyboard", 350, 3)
print(item.subtotal)    # 1050   (read without parentheses)
item.quantity = 5
print(item.subtotal)    # 1750   (recomputed on every read)
```

Each time `subtotal` is read, the body runs again; that is why when `quantity` changes the result is updated too. From the outside it looks like an ordinary field, but behind it lies a computation.

7.6 Setters: `set` and Two-Way Computed Fields

The counterpart of `get` is `set`: a **setter** that runs when `instance.name = value` is written. The assigned value is bound to the parameter. By defining a `get`/`set` pair together, you can create a two-way computed field that is backed behind the scenes by *another* field. The classic example is a temperature converter: it keeps the data in a single field (Celsius), but it can be read and written as both Celsius and Fahrenheit.

```
struct Thermostat
    def init(celsius)
        self._celsius = celsius          # the real data: the backing field (with _)
    end

    get celsius()
        return self._celsius
    end
    set celsius(value)
        self._celsius = value
    end

    get fahrenheit()
        return self._celsius * 9 / 5 + 32
    end
    set fahrenheit(value)
        self._celsius = (value - 32) * 5 / 9
```

```
    end
end

let t = Thermostat(20)
print(t.celsius)        # 20
print(t.fahrenheit)     # 68

t.fahrenheit = 212      # the setter runs: _celsius = 100
print(t.celsius)        # 100
print(t.fahrenheit)     # 212

t.celsius = 0
print(t.fahrenheit)     # 32
```

Note carefully here: there is *no stored field* named `fahrenheit`. Only `_celsius` is stored; `fahrenheit` is converted to it each time it is read or written. The leading underscore (`_celsius`) carries no special meaning in Cobra — it is merely a common naming convention that says "this is the internal backing field, do not touch it directly."

> **Warning:** A setter **must not write to the field with the same name as itself**, otherwise it falls into infinite recursion. The following code, because it again writes to `self.celsius` inside `set celsius`, retriggers the setter and overflows the stack (stack overflow):
>
> ```
> set celsius(value)
>     self.celsius = value     # WRONG: re-invokes the setter -> infinite loop
> end
> ```
>
> The correct approach is for `set celsius` to write to a *different* backing field such as `self._celsius`. The rule is simple: a property and the field that backs it must have different names.

Getters and setters are also passed down to subtypes through inheritance (we will see inheritance shortly).

7.7 Class Constants: `const`

Inside a `struct` body, `const` defines a **class constant** belonging to that type. It belongs not to a particular instance but to the type itself, and is read with `Type.NAME` (it can also be read through an instance, `instance.NAME`). It is ideal for constant values tied to the type — a physics constant, a default threshold, a conversion factor.

```
struct Temperature
    const ABSOLUTE_ZERO_C = -273
    const FREEZING_C = 0

    def init(celsius)
        self.celsius = celsius
    end
end

print(Temperature.ABSOLUTE_ZERO_C)   # -273   (through the type)

let ice = Temperature(0)
print(ice.ABSOLUTE_ZERO_C)           # -273   (also read through the instance)
```

Note: *Outside* a `struct`, `const` makes an ordinary immutable variable binding (Chapter 2). *Inside* a `struct`, however, its meaning changes and it becomes a class constant. The context is determinative.

7.8 Static Methods: `static def`

`static def` defines a method belonging not to an instance but **to the type itself**. It has no `self` and is called in the form `Type.method(...)`. Its most common use is **factory methods** (alternative constructors) that produce the object from different inputs. While `init` expects a Celsius value, let us write a helper that produces an instance from a Fahrenheit value:

```
struct Temperature
    const FREEZING_C = 0

    def init(celsius)
        self.celsius = celsius
    end

    static def from_fahrenheit(f)        # factory: no self
        return Temperature((f - 32) * 5 / 9)
    end

    static def freezing()
        return Temperature(Temperature.FREEZING_C)
    end
end

let boiling = Temperature.from_fahrenheit(212)
print(boiling.celsius)   # 100

let ice = Temperature.freezing()
print(ice.celsius)       # 0
```

Because static methods carry no `self`, they work only with their arguments and class constants. They are the natural way to reason directly about the type rather than about an instance.

7.9 Inheritance: `struct Dog < Animal`

Cobra supports **single inheritance**: a type can extend another type with `<`. The subtype inherits all of the parent type's methods, properties, and class constants; it can **override** whatever it wants and access the parent version with `super`:

- `super.init(...)` — calls the parent type's constructor (usually the first line of the subtype's `init`).
- `super.method(...)` — calls the parent version of a method you have overridden.

Let us see it with a geometry example: a common `Shape` base and `Circle` and `Rectangle` extending it. Both define `area` in their own way.

```
struct Shape
    def init(name)
        self.name = name
    end

    def area()
        return 0                # base: subtypes override
    end

    def describe()
        return self.name + " area: " + str(self.area())
    end
end

struct Circle < Shape
    const PI = 3.14159

    def init(radius)
        super.init("Circle")    # call the parent constructor
        self.radius = radius
    end

    def area()                  # override
        return Circle.PI * self.radius * self.radius
    end
end

struct Rectangle < Shape
    def init(width, height)
        super.init("Rectangle")
        self.width = width
        self.height = height
    end

    def area()
        return self.width * self.height
    end

    def describe()              # override and extend the parent version
        return super.describe() + " (" + str(self.width) + "x" + str(self.height) + ")"
    end
end

let shapes = [Circle(2), Rectangle(3, 4)]
for s in shapes
    print(s.describe())
end
# Circle area: 12.56636
# Rectangle area: 12 (3x4)
```

There are a few important points here. `Shape.describe` calls `self.area()` within itself; but `self` is the actual instance at runtime (a `Circle` or a `Rectangle`), so that instance's overridden `area` runs. This is called **polymorphism**: a single `describe` body selects the correct `area` according to the subtype. `Rectangle.describe`, on the other hand, calls the base behavior with `super.describe()` and appends its own

additional information to the end.

> **Tip:** If a subtype's `init` needs the parent type's fields, call `super.init(...)` usually on the first line. That way, after the base fields (`self.name`) are set up, you add the subtype-specific fields.

7.10 `record`: Immutable Value Objects

Some types represent not an **identity** but a **value**: a monetary amount, a 2D point, a date. For such "value objects" we want two things: **immutability** (once set up, the fields should not change) and **value-based equality** (two amounts with the same content should be considered equal). `record` gives exactly this.

A `record` is written like a `struct` but with three differences:

1. Its instances are **frozen**: writing to a field after construction is an error.
2. == does a field-by-field **value** comparison (not identity).
3. Every instance has a `.with(dict)` method that produces a **copy** with some fields changed.

```
record Money
    def init(amount, currency)
        self.amount = amount
        self.currency = currency
    end

    def plus(other)
        if other.currency != self.currency
            throw "currencies do not match"
        end
        return Money(self.amount + other.amount, self.currency)
    end
end

let price = Money(100, "TRY")
let same  = Money(100, "TRY")
let other = Money(250, "TRY")

print(price == same)    # true    (value-based equality)
print(price == other)   # false

# .with -> a COPY with some fields changed (the original object is unchanged)
let raised = price.with({"amount": 150})
print(raised.amount)    # 150
print(raised.currency)  # TRY
print(price.amount)     # 100     (the original is unchanged)

let total = price.plus(other)
print(total.amount)     # 350
print(total)            # Money{amount: 350, currency: "TRY"}
print(type(price))      # Money
```

Because a `record` is immutable, the only way to "update" its fields is to produce a new copy. `.with(dict)` changes the named fields in the dictionary, copies the rest as-is, and freezes the result again. Trying to change a field directly raises an error:

```
record Point
    def init(x, y)
        self.x = x
        self.y = y
    end
end

let p = Point(1, 2)
try
    p.x = 99
catch err
    print("error:", err)    # error: line 9: cannot modify immutable record Point
end
print(p.x)                 # 1   (unchanged)
```

> **Tip:** If you think a type carries only data, containing no computation or changing state, choose `record`. Value-based equality and immutability make it easy to think of these objects like dictionary keys and to share them safely within concurrent code.

7.11 `contract` and `implements`: Runtime Contracts

In most languages, the guarantee that "this type must have these methods" is given by **interfaces**. Cobra has no interfaces; in their place there is **`contract`**, a duck-typed runtime check. A contract lists the required method names, one per line:

```
contract Serializable
    to_dict
    from_dict
end
```

`implements(value, Contract)` checks at runtime whether a value has all the methods in the contract and returns `true`/`false`. This is asking "can it do this?" rather than asking "is it of this type?" — which is generally the more robust of the two in object-oriented design.

```
struct User
    def init(id, name)
        self.id = id
        self.name = name
    end

    def to_dict()
        return {"id": self.id, "name": self.name}
    end

    def from_dict(data)
        return User(data["id"], data["name"])
    end
end

struct Tag                      # has to_dict, but NO from_dict
    def init(label)
```

```
        self.label = label
    end
    def to_dict()
        return {"label": self.label}
    end
end

let u = User(1, "Ada")
print(implements(u, Serializable))            # true
print(implements(Tag("urgent"), Serializable)) # false   (from_dict is missing)

def save(value)
    if not implements(value, Serializable)
        throw "cannot save: not Serializable"
    end
    return value.to_dict()
end

print(save(u))    # {"id": 1, "name": "Ada"}
```

Here `save` does not care about the concrete type of the value passed to it; it only requires that it satisfy the `Serializable` contract. `User` and any type you write in the future — as long as it provides `to_dict` and `from_dict` — will work without issue. `Tag`, on the other hand, does not satisfy the contract because it lacks `from_dict`, and the check catches this. The same pattern applies just as well to a `Repository` contract (`save`, `find`, `delete`): without binding to the storage layer's concrete class, you trust only its capability.

> **Note:** `implements` is Cobra's idiomatic answer to the absence of `isinstance`/type checking. Instead of asking "is this object of such-and-such a type?" you ask "does it have the required methods?" — a loosely coupled design in keeping with the spirit of duck typing.

7.12 What Cobra Does Not Have

Cobra's object model is deliberately small. Knowing some features that you may recognize from other languages and that **are not present** in Cobra protects you from writing invalid code:

- **There is no operator overloading / dunder methods.** You cannot define special methods like `__eq__`, `__str__`, `__add__`; you cannot change the behavior of `+` or `==` for your own type. (One exception: `record` provides value-based equality for `==` *built in*.) To add two objects, write an explicit method like `a.plus(b)`.
- **There is no multiple inheritance.** A type can extend only a single parent type (`struct Dog < Animal`). If you want to share behavior from multiple bases, use composition (hold another object in a field) or express the common capability with a `contract`.
- **There is no `isinstance`-like type check.** Instead of asking "is this object of such-and-such a type?" ask "does it have these methods?" with `implements(value, Contract)`. If you need a quick type-name comparison, `type(x)` returns a string (`type(u) == "User"`), but for flexible design `contract` is preferred.

> **Tip:** Although these absences may look like a limitation, most of the time they produce clearer code: named methods instead of hidden operator behaviors, simple composition and contracts instead of complex inheritance hierarchies.

7.13 Summary

In this chapter we saw Cobra's object-oriented face from start to finish:

- **`struct`** (and its synonym **`class`**) defines a simple class that brings state and behavior together. **`init`** is the constructor and there is an **implicit `self`** in every method.
- You access fields with `p.x` and write them with `p.x = v`; you can add a **new field to instances at any time**. Instances are written in the form `Type{field: value}`, are compared **by identity**, and `type(instance)` gives the type name.
- Instance methods are called with parentheses (`acc.deposit(50)`). **`get name()`** makes a property readable without parentheses; **`set name(v)`** runs on assignment. A `get`/`set` pair produces a computed field backed by an internal backing field (`self._x`) — the setter must write to a **different** field, otherwise it falls into an infinite loop.
- **`const NAME`** defines, inside a `struct`, a class constant read with `Type.NAME`. **`static def`** defines a method belonging to the type rather than an instance, with no `self` (`Type.method()`) — ideal for factory methods.
- **Single inheritance** is done with `struct Sub < Base`; **`super.init(...)`** and **`super.method(...)`** access the parent versions. Overriding methods makes polymorphism possible.
- **`record`** produces immutable, **value-equal** instances; **`.with(dict)`** returns a copy with some fields changed. It is the right tool for value objects.
- **`contract Name ... end`** lists the required method names; **`implements(value, Contract)`** checks them at runtime — a duck-typed solution that takes the place of interfaces.
- Cobra has **no operator overloading/dunder methods, no multiple inheritance, and no `isinstance`**; in their place, explicit methods, composition, and contracts are used.

In the next chapter, we will move on to handling **errors** — inevitable as programs grow — writing robust, recoverable code with `try`/`catch`/`finally` and `throw`.

Chapter 8

Error Handling

Real programs rarely run under ideal conditions. A user submits an empty form, a file gets truncated halfway, an HTTP request times out, the incoming JSON is malformed. Instead of crashing in these situations, a well-written program treats them as *expected* events: it catches the error, produces a clear message, closes the resources it opened, and retries when possible.

Cobra's error handling is deliberately small. There is a single block construct (`try ... catch ... finally ... end`) and a single statement (`throw`). Unlike some languages, there are no dozens of *exception* classes or separate `catch` blocks per type. Instead: any value can be thrown, `catch` catches everything, and you do the work of distinguishing the error (if you need to) yourself. In this chapter we will look first at the basic construct, then at how to design error values, the guarantee that `finally` provides, and how errors propagate up the call stack — all through engineering scenarios.

8.1 `try / catch / finally`: The Basic Construct

A `try` block wraps code that "might fail." If an error occurs, Cobra stops the normal flow and transfers control to the `catch` block. The `finally` block, on the other hand, **always** runs — whether or not an error occurred.

```
try
    let parts = "a,b,c".split(",")
    print(parts[5])        # out of range -> runtime error
catch err
    print("caught:", err)
finally
    print("done")
end
```

Output:

```
caught: line 4: list index out of range: 5
done
```

Here `parts[5]` is an invalid index, so Cobra produces a runtime error; `print(parts[5])` never runs, and the flow jumps straight to the `catch err` block. `err` is the caught error value (we will see what kind of value this is shortly). At the very end, the `finally` block runs.

The syntax of the construct is as follows:

```
try
    ...                # protected code
catch err              # 'err' is optional
    ...                # runs if there is an error
finally
    ...                # runs in every case
end
```

Keep two rules in mind from the start:

- **At least a `catch` or a `finally` must be present.** An empty `try ... end` is invalid.
- The **variable name after `catch` is optional.** If you are not going to use the error's value, you can write just `catch`.

If you try to write an empty `try`, the error is caught at the parse stage — that is, before the program even runs:

```
try
    print("hi")
end
```

```
parse error: line 2: try requires catch or finally
```

You can also use `catch` without binding a name; this is enough if you do not care about the error's details:

```
try
    throw "ignored detail"
catch
    print("something failed")
end
```

```
something failed
```

> **Note:** The `catch` block runs no matter which error occurs. In Cobra there is no "catch this type of error, don't catch that one" distinction. If you want to distinguish errors, you do that yourself *inside* the `catch` block with `type(err)` or a field check.

8.2 `throw`: Any Value Can Be Thrown

You throw your own errors with `throw`. This is the most important point where Cobra differs from other languages: **the thrown value can be any type** — string, integer, boolean, dict, list... Whatever the thrown value is, `catch` catches it exactly as is.

```
# any value can be thrown: integer, string, dict...
for v in [42, "text", true]
    try
        throw v
    catch err
        print(type(err), "->", err)
    end
end
```

```
INTEGER -> 42
STRING -> text
BOOLEAN -> true
```

In the simplest case, throwing a string is enough. But in a real system, an error has not only a *message* but also **machine-processable** extra information: an error code, which field is at fault, perhaps an HTTP status code. For this reason, in professional code errors are usually thrown as a **dict**:

```
throw {"code": 404, "message": "record not found"}
```

The side that catches this can both show the user a message and make a decision based on the code:

```
try
    throw {"code": 404, "message": "not found"}
catch err
    print(type(err))       # DICT
    print(err["code"])     # 404
    print(err["message"])  # not found
end

DICT
404
not found
```

8.2.1 Engineering Example: Input Validation

The real benefit of structured error values becomes apparent in a validation layer. The `validate_user` below inspects a form coming from the user and, at the first field that violates a rule, throws an error in the form `{"code", "field", "message"}`. Keeping the error value structured gives the caller the "which field, why" information:

```
# --- Validating user input, with structured error values ---

# If validation fails, throws an error in the form {"code", "field", "message"}.
def validate_user(form)
    if not form.has("name") or form["name"].strip() == ""
        throw {"code": "EMPTY", "field": "name", "message": "name is required"}
    end
    if not form.has("age")
        throw {"code": "MISSING", "field": "age", "message": "age is required"}
    end
    let age = int(form["age"])     # throws a runtime error if not a number
    if age < 0 or age > 130
        throw {"code": "RANGE", "field": "age", "message": "age must be in the range 0-130"}
    end
    return {"name": form["name"].strip(), "age": age}
end

let forms = [
    {"name": "Ada", "age": "36"},
    {"name": "  ", "age": "20"},
    {"name": "Bob"},
```

```
    {"name": "Eve", "age": "200"},
    {"name": "Cem", "age": "thirty"}
]

for form in forms
    try
        let user = validate_user(form)
        print("OK   ->", user["name"] + ", " + str(user["age"]))
    catch err
        if type(err) == "DICT"
            print("ERROR -> field=" + err["field"] + " code=" + err["code"] + ": " + err["message"])
        else
            # runtime errors such as int() arrive as a plain string
            print("ERROR -> " + err)
        end
    end
end
```

```
OK   -> Ada, 36
ERROR -> field=name code=EMPTY: name is required
ERROR -> field=age code=MISSING: age is required
ERROR -> field=age code=RANGE: age must be in the range 0–130
ERROR -> line 11: cannot convert "thirty" to int
```

Note the last line: `int("thirty")` is not an error we threw, but **a runtime error produced by Cobra**, and it arrives as a plain string, not a dict. The `type(err) == "DICT"` check in the `catch` block is exactly what lets us distinguish these two cases from each other. The next section clarifies this distinction.

8.3 Runtime Errors Are Caught as a Message String

Division by zero, out-of-range index, missing dict key, type conversion error… These are not values you threw with `throw`; they are produced by **Cobra's runtime**. When caught with `catch`, their value is the **message string** of that error:

```
try
    let x = 10 / 0
catch err
    print(err)         # line 2: division by zero
    print(type(err))   # STRING
end
```

```
line 2: division by zero
STRING
```

That is, `err` here is not an object or a special "Exception" type, but simply the string `"line 2: division by zero"`. This is a direct consequence of Cobra's "everything is a plain value" philosophy, and in practice it is very convenient: to log the error, show it to the user, or search for a substring inside it, you do not need to learn an extra API; text is text.

This behavior shows its full power when combined with modules that read data from the outside world. For example, if `json.parse` encounters a malformed input, the error it throws is again a string:

```
import "json"

let bad = "{ \"name\": \"ada\", }"
try
    let data = json.parse(bad)
    print(data)
catch err
    print("parse failed:", err)
end
```

```
parse failed: line 5: json.parse: invalid character '}' looking for beginning of object key string
```

> **Tip:** Since runtime errors are strings and your own structured errors are usually dicts, distinguishing the two with `type(err)` in a `catch` block is a common and robust pattern. In the validation example above we did exactly this.

8.4 There Are No Exception Types

In many languages, errors form a class hierarchy and you catch only a specific type, as in `catch (IOException e)`. **There is no such thing in Cobra.** There is a single `catch`, and it catches whatever was thrown. The advantage is simplicity: there is no type hierarchy to learn, no worry about finding the right `except` order. The cost is that you have to write the "catch only this error, let the rest bubble up" behavior yourself.

You obtain this behavior with a condition + a re-throw (`throw`) inside `catch`:

```
def parse_port(s)
    try
        return int(s)
    catch err
        # We only handle the conversion error; let everything else bubble up.
        throw {"code": "BAD_PORT", "message": "invalid port: " + s}
    end
end

try
    print(parse_port("8080"))   # 8080
    print(parse_port("xx"))     # a dict error is thrown from here
catch err
    print(err["code"] + ": " + err["message"])
end
```

```
8080
BAD_PORT: invalid port: xx
```

> **Warning:** Since `catch` catches everything, be careful to catch only errors you can genuinely handle. Doing only a `print` inside a broad `catch` and swallowing the error can hide a real defect in the program. Leaving an error you cannot handle to a higher layer with `throw err` is most often the right thing to do.

8.5 `finally` Always Runs

The `finally` block has a single job, and it does it flawlessly: to run no matter how the `try` is exited. It runs on normal completion, on an error caught with `catch`, and even on early exit via `return`/`break`/`continue`. For this reason, `finally` is tailor-made for **resource cleanup**: you close the file, lock, or connection you opened here.

8.5.1 Engineering Example: Resource Cleanup

The following function writes to a temporary file, processes its contents, and **always** deletes the file inside `finally` — whether the operation succeeds or an error occurs:

```
import "fs"

# --- File/resource cleanup: finally runs in every case ---

# Writes to a temporary file, processes it, then ALWAYS deletes it in finally.
def process_report(path, lines)
    fs.write_lines(path, lines)
    try
        let total = 0
        for line in fs.read_lines(path)
            total += int(line)     # throws a runtime error on a malformed line
        end
        return total
    finally
        # The temporary file is cleaned up whether or not there was an error.
        fs.remove(path)
        print("cleaned up:", path, "exists?", fs.exists(path))
    end
end

let tmp = "/private/tmp/cobra_report.txt"

# 1) Success case
print("total:", process_report(tmp, ["10", "20", "12"]))

# 2) Error case: the file should still be deleted, the error should propagate up
try
    process_report(tmp, ["5", "x", "9"])
catch err
    print("operation failed:", err)
end
```

```
cleaned up: /private/tmp/cobra_report.txt exists? false
total: 42
cleaned up: /private/tmp/cobra_report.txt exists? false
operation failed: line 11: cannot convert "x" to int
```

Note the order of the output. On the first call the operation succeeds; yet **before** "total: 42" is printed, `finally` runs and deletes the file. On the second call `int("x")` throws an error; `return` never runs, but `finally` still kicks in and cleans up the file — *then* the error propagates to the `try` outside `process_report` and is caught. So `finally` does not swallow the error; it just does its job before the flow leaves that point.

8.5.2 An Error Thrown Inside `finally` Wins

What if both `try` (or `catch`) has thrown an error and `finally` throws *its own* error? In Cobra the rule is clear: **the error inside `finally` wins** and takes the place of the first error.

```
try
    try
        throw "original"
    finally
        throw "from finally"
    end
catch err
    print(err)
end
```

```
from finally
```

The `"original"` error is shadowed by the `throw` inside `finally`, and the value that reaches the outer `catch` becomes `"from finally"`.

> **Warning:** This is a good reason to avoid throwing errors inside `finally`. If a cleanup step can fail (for example, closing a remote connection), wrap it in its own `try` so it does not hide the real error.

8.6 `return`, `break`, and `continue` Pass Through `try` and `finally`

`try`/`finally` does *not* get in the way of your control-flow statements. When a `return`, `break`, or `continue` is triggered from inside a `try` block, `finally` runs first, and then the statement takes effect. Even when you `return` from a function inside a `try`, `finally` runs:

```
def f()
    try
        print("try")
        return "from try"
    finally
        print("finally")
    end
end
print(f())
```

```
try
finally
from try
```

`return "from try"` begins to execute; but before the function returns its value, the `finally` block runs ("finally" is printed), *then* the value `"from try"` is returned.

The same holds for `break` and `continue` inside loops:

```
for i in range(4)
    try
        if i % 2 == 0
            continue
        end
        print("odd:", i)
    finally
        print("  finally", i)
    end
end
```

```
  finally 0
odd: 1
  finally 1
  finally 2
odd: 3
  finally 3
```

When `i` is even, `continue` jumps to the next iteration of the loop — but before it jumps, that iteration's `finally` runs. So resource cleanup stays safe even if the loop is cut short early.

8.6.1 Prohibitions Inside `finally`

The `finally` block is for cleanup; statements that *redirect* the flow or *introduce* new names are prohibited there. Concretely, inside `finally`, `return` and declarations (`let`, `const`, `def`, `struct`…) are a **parse error** — the program is rejected before it even runs.

```
def f()
    try
        return 1
    finally
        return 2
    end
end
print(f())
```

```
parse error: line 6: 'return' not allowed inside finally
```

A declaration is rejected in the same way:

```
try
    print("ok")
finally
    let x = 5
    print(x)
end
```

```
parse error: line 5: 'let' not allowed inside finally
```

> **Note:** This restriction is deliberate. If there were a `return` inside `finally`, it would silently swallow the error from the `try` and make the flow ambiguous. Use `finally` only for "cleanup to be done no matter what"; keep the logic inside `try` or `catch`.

8.7 Propagation of Errors Up the Call Stack

When an error occurs, Cobra looks for the **innermost (nearest) `try`** that will catch it. The error "bubbles up" (propagation) from the function in which it occurred to its caller, from there to that one's caller, and so on. The first `try` on the path catches it. If there is no `try` at all, the error stops the program.

```
def parse_age(s)
    let n = int(s)        # throws if s isn't a number
    return n
end

try
    print(parse_age("oops"))
catch err
    print("caught:", err)
end
```

```
caught: line 2: cannot convert "oops" to int
```

The error was born inside `parse_age`, on line `2`; but there is no `try` there. So the error propagated to the place where `parse_age` was called, and the `catch` there caught it. Note that the message still carries `line 2` — no matter at which layer the error is caught, it carries the number of the **line where it was born**.

If no `try` intervenes, the error rises all the way to the top level and terminates the program:

```
def g()
    throw "boom from g"
end
def h()
    g()
end
print("before")
h()
print("after")
```

```
before
runtime error: line 2: uncaught throw: "boom from g"
```

`"before"` was printed, but `g()` inside `h()` threw an error and, since no one caught it, the program stopped — `"after"` never ran. An uncaught dict is reported the same way, with its contents shown:

```
runtime error: line 2: uncaught throw: {"code": 500, "message": "kaboom"}
```

8.7.1 Engineering Example: Error Propagation Across Layers

Real applications are layered: at the bottom a **repository** that touches raw data, above it a **service** that applies business rules, and at the top an **application boundary** that talks to the user. In a good design the lower layer throws the raw error; the middle layer catches it and **wraps it into a meaningful, structured error** (keeping the original as `cause`); and the top layer handles all errors in a single place, translating them into the user's language.

```
import "json"

# --- Error propagation across layers ---
# Layer 1 (repository): parses raw data.
# Layer 2 (service): applies business rules, wraps the lower-layer error.
# Layer 3 (application): translates into a message to show the user.

# Layer 1 — repository
def load_config(raw)
    return json.parse(raw)      # malformed JSON -> runtime error (string)
end

# Layer 2 — service: catches the error and wraps it into a meaningful, structured error
def read_timeout(raw)
    let cfg = null
    try
        cfg = load_config(raw)
    catch err
        throw {"code": "CONFIG_INVALID", "message": "could not read configuration", "cause": err}
    end
    if not cfg.has("timeout")
        throw {"code": "CONFIG_MISSING", "message": "timeout field is missing"}
    end
    return cfg["timeout"]
end

# Layer 3 — application boundary: handles all errors in a single place
def show(raw)
    try
        print("timeout =", read_timeout(raw))
    catch err
        if type(err) == "DICT"
            print("[" + err["code"] + "] " + err["message"])
            if err.has("cause")
                print("    cause:", err["cause"])
            end
        else
            print("unexpected error:", err)
        end
    end
end

show("{\"timeout\": 30}")          # valid
show("{\"timeout\": 30")            # malformed JSON -> wrapped
show("{\"retries\": 3}")           # field missing

timeout = 30
[CONFIG_INVALID] could not read configuration
    cause: line 10: json.parse: EOF
[CONFIG_MISSING] timeout field is missing
```

This pattern lets each layer produce errors “in its own language.” The application boundary does not have to deal with `json.parse`’s raw message; it sees the `CONFIG_INVALID` code and, if it wants, also logs the real lower-level cause from the `cause` field.

8.8 Error Messages Carry Line Numbers

As we have seen repeatedly in the previous examples, every runtime error Cobra produces carries a `line N:` prefix, and that number is the line where the error **was born**. This does not change whether the error is caught where it was born, several functions up — or even stops the program without being caught at all. Uncaught errors arrive with a `runtime error:` prefix, while caught ones arrive merely as the `line N: ...` string.

This consistency holds for both execution engines: whether it is the default tree-walking interpreter or the `--vm` bytecode machine, the runtime error messages, together with the line numbers, are exactly the same.

> **Tip:** When logging an error, printing `err` as is is most often enough; the line number is already inside it. If you also add some context (which operation, which record) to your own structured errors, debugging gets much faster.

8.9 Engineering Example: Retry Loop

A common use of error handling is to **retry** an operation on *transient* errors. Network requests, locked resources, or unstable services may succeed on the second or third attempt. The `with_retry` below attempts a function at most `max_tries` times; it retries on each failure, and if the last attempt also fails, it propagates the error up. Here we see how a `try` can be re-run inside a loop and how the error is re-thrown with `throw err`:

```
# --- Retry loop ---

# An "unstable" resource that fails on the first two calls and succeeds on the third.
let attempts = 0
def fetch_user(id)
    attempts += 1
    if attempts < 3
        throw {"code": "UNAVAILABLE", "message": "temporarily unavailable"}
    end
    return {"id": id, "name": "user-" + str(id)}
end

# Attempts fn at most max_tries times; retries on each failure,
# and propagates the error up if the last attempt also fails.
def with_retry(fn, max_tries)
    let tries = 0
    while true
        tries += 1
        try
            return fn()
        catch err
            if tries >= max_tries
                print("last attempt also failed, propagating error")
                throw err
            end
            print("attempt " + str(tries) + " failed, will retry")
        end
    end
end
```

```
def load()
    return fetch_user(7)
end

let user = with_retry(load, 5)
print("success:", user["name"])

attempt 1 failed, will retry
attempt 2 failed, will retry
success: user-7
```

The first two attempts get the `UNAVAILABLE` error and the loop goes around again; when the third attempt succeeds, `return fn()` runs and we exit `with_retry`. If all five attempts had failed, the final `catch` would propagate the error to the caller with `throw err`, and it would be our responsibility to catch it at the top (or let the program stop).

8.10 Summary

Cobra's error handling, like the rest of the language, rests on the consistent behavior of a small number of parts:

- **There is a single block construct:** `try ... [catch [err]] ... [finally] ... end`. At least a `catch` or a `finally` must be present; otherwise you get a parse error.
- **`throw <value>` can throw any value** — string, integer, dict, list... `catch` catches it exactly as is. For structured errors, the `throw {"code": ..., "message": ...}` pattern gives the caller both a message and machine-readable information.
- **Runtime errors are caught as a message string** (e.g. `"line 3: division by zero"`). With `type(err)` you can distinguish these from your own dict errors.
- **There are no exception types:** `catch` catches everything. If you want to be selective, you check a condition inside `catch` and re-throw the error you do not handle with `throw err`.
- **`finally` always runs** — on normal completion, on an error, and even on early exit via `return`/`break`/`continue`. This is the right place for resource cleanup. An error thrown inside `finally` takes the place of (wins over) the real error.
- **`return`, `break`, and `continue` pass through `try` and `finally`:** `finally` runs before the control-flow statement takes effect.
- **Inside `finally`, `return` and declarations (`let`, `const`, `def`, ...) are a parse error.** Use `finally` only for cleanup.
- **Errors propagate up the call stack to the innermost `try`;** if not caught, they stop the program with `runtime error: ...`.
- **Error messages carry the number of the line where they were born,** and this is exactly the same across the two execution engines (tree-walking and `--vm`).

With these simple rules, you can meet the entirety of real engineering needs — input validation, catching malformed data, guaranteed resource cleanup, retrying, and meaningful error propagation across layers — without learning extra syntax.

Chapter 9

Modules and the Package System

As a program grows, a single file quickly becomes a burden. When your geometry calculations, your configuration constants, and your main flow are intertwined in the same file, it becomes hard to tell what belongs where. The solution is to split the code into meaningful pieces — **modules**. In Cobra every `.cobra` file is a module; one module pulls in what another offers with `import`.

Cobra's module system is built on two simple principles. The first: a module exposes **only what it explicitly marks with `export`** to the outside — the default is private, just like ES modules or Rust's `pub` keyword. The second: a file is executed **once**, its result is cached, and paths are always resolved **relative to the folder of the importing file**. In this chapter we will look at both of these principles in full detail: the `import` and `from ... import` forms, privacy, "barrel" modules, the priority of built-in modules, and finally compiling and packaging an application into a single file.

9.1 Importing a Module: `import`

The most basic form is to import a file by its path. `import` executes the file once and binds a **module object** derived from the file name. You access its members with a dot (`.`):

```
import "lib/geometry.cobra"

print(geometry.PI)              # 3.14159
print(geometry.area_circle(2))  # 12.57
```

The module object's name is the file name with the `.cobra` extension stripped: `lib/geometry.cobra` → `geometry`. The folder path is not part of the name; only the file name counts.

> **Note:** Module paths are **always relative to the folder of the importing file**, not to where you run the program from. The `main.cobra` in the root folder points to a module under `lib/` with `"lib/geometry.cobra"`; but `lib/geometry.cobra` itself points to the `config.cobra` next to it with just `"config.cobra"` — relative to its own folder. (In the REPL there is no "importing file," so paths are resolved relative to the current working directory, i.e. the cwd.)

9.1.1 Alias: `import ... as`

If you want to give a module a different name, you use `as`. This is useful for shortening long file names or avoiding name collisions:

```
import "lib/geometry.cobra" as geo

print(geo.PI)            # 3.14159
print(geo.area_rect(8, 3))  # 24.0
```

9.1.2 Runs Once, Gets Cached

No matter how many times a module is imported, the file is executed **only once**; subsequent `imports` return the same cached module. You can observe this with a side effect at the module's top level (for example, a `print`):

```
# loud.cobra contains a print at the top level:
import "lib/loud.cobra"
import "lib/loud.cobra" as l2     # same module; does NOT run again
print(loud.X + l2.X)

loud.cobra loaded
2
```

The "loud.cobra loaded" output appears only once despite the two `imports`. This caching is important for both performance and correctness: the state of a shared module is the same everywhere.

> **Warning:** Modules cannot form a **cycle**. If `a.cobra` imports `b.cobra` while `b.cobra` also imports `a.cobra`, Cobra reports this as an error:
>
> ```
> runtime error: ... circular import: "a.cobra"
> ```
>
> A circular dependency is usually a design smell; resolve it by extracting the shared piece into a third module.

9.2 Private by Default: `export`

At the heart of Cobra's module system there is a single rule: **a module exposes to the outside only what it marks with `export`.** Everything not marked — variables, functions, structs — is private to the module and not visible from outside. This is the "private by default" approach; making something public requires a deliberate decision.

```
# geometry.cobra
export const PI = 3.14159

export def area_circle(r)      # public
    return _round(PI * r * r)
end

def _round(x)                  # PRIVATE — no `export`
    ...
end
```

Here `area_circle` is exposed to the outside, but `_round` is not. The importing side can call `geometry.area_circle(...)`, but if it calls `geometry._round(...)` it gets an error — because `_round` is not part of the module's public surface.

> **Tip:** The _ prefix (for example, `_round`, `_helper`) is only a **convention**: it conveys the "this is private" message to the reader. What actually provides the privacy is not _, but the absence of `export`. If you like, you can write a private helper without putting a _ on it; as long as there is no `export`, it stays private.

`export` works with all of these declarations: `let`, `const`, `def`, `struct`, and `record`:

```
export let DEFAULT_UNIT = "cm"
export const MAX = 100
export def area(r) ... end
export struct Circle ... end
export record Point ... end
```

9.3 Selective Import: `from ... import`

`import` binds the entire module as a single object, and you access its members with `module.member`. Sometimes you need only a few names and do not want to write the module name every time. `from ... import` does exactly this: it binds selected public members **directly** into the current scope — no prefix:

```
from "lib/geometry.cobra" import area_circle, PI

print(area_circle(2))   # 12.57   — no need for "geometry."
print(PI)               # 3.14159
```

To rename an imported name, you use `as`:

```
from "lib/geometry.cobra" import PI as PI_CONST, area_rect as area

print(area(8, 3))       # 24.0
print(PI_CONST)         # 3.14159
```

9.3.1 Errors Are Caught AT IMPORT TIME

The most valuable feature of `from ... import` is that it reports a typo or a name that is not exported **at import time** — immediately, rather than as a mysterious error that later accesses `null`:

```
from "lib/geometry.cobra" import area_circel    # typo!

runtime error: ... module "lib/geometry.cobra" does not export "area_circel"
```

The same happens when you try to request a name that exists but is **not exported** (private):

```
from "lib/geometry.cobra" import pow10           # exists, but not exported

runtime error: ... module "lib/geometry.cobra" does not export "pow10"
```

In both cases the error surfaces before the rest of the program runs. This is a powerful way to validate the contracts between modules early.

9.3.2 Binding All of Them: `from ... import *`

`from "path" import *` binds **all** of the module's exported names into the current scope at once. It is handy when you want to use a module's public API from start to finish without a prefix:

```
from "lib/geometry.cobra" import *

print(area_circle(2))     # 12.57
print(Circle(3).area())   # 28.27
```

> **Note:** `from ... import *` binds the names only into the **current scope**; it does not automatically *re-export* them. That is, doing `import *` inside a module does not cause those names to be added to your module's public surface. To re-publish a name, you must explicitly `export` it — the "barrel" pattern in the next section shows exactly this.

9.4 Engineering Example: A Multi-File Application

Let us bring together the pieces we have seen so far in a real application. We will build a small "Geometry Workshop" and split it into three responsibilities: configuration, the geometry core, and a "barrel" module that collects them under a single roof. The folder structure will be as follows:

```
project/
├── main.cobra            # entry point
└── lib/
    ├── config.cobra      # application constants
    ├── geometry.cobra    # geometry core (public API)
    └── shapes.cobra      # barrel: re-publishes submodules
```

9.4.1 `lib/config.cobra` — Configuration

Let us start with the smallest module: the application-wide constants. They are all `export`ed, because their purpose is to be read from outside anyway.

```
# config.cobra — application-wide settings.
# Only those marked with `export` are exposed to the outside.

export const APP_NAME = "Geometry Workshop"
export const VERSION  = "1.0.0"

# How many decimal places to round to in calculations.
export const PRECISION = 2
```

9.4.2 `lib/geometry.cobra` — The Geometry Core

The real work is here. This module imports its sibling module `config.cobra` (relative to its own folder, just `"config.cobra"`), offers public area functions and structs, but keeps the rounding helpers **private**:

```
# geometry.cobra — basic plane-geometry operations.
# Public API: PI, area_circle, area_rect, Circle, Rect.
# Names starting with `_` are a convention; the REAL privacy comes from
# the absence of `export` — everything not marked is private to the module.

import "config.cobra"        # sibling module in the same folder

export const PI = 3.14159

# Private helper that rounds a number to config.PRECISION places.
# NO `export` → not visible from outside.
def _round(x)
    let factor = pow10(config.PRECISION)
    return float(int(x * factor + 0.5)) / factor
end

# Another private helper used only by this module.
def pow10(n)
    let result = 1
    let i = 0
    while i < n
        result = result * 10
        i = i + 1
    end
    return result
end

export def area_circle(r)
    return _round(PI * r * r)
end

export def area_rect(w, h)
    return _round(w * h)
end

export struct Circle
    def init(r)
        self.r = r
    end
    def area()
        return area_circle(self.r)
    end
end

export struct Rect
    def init(w, h)
        self.w = w
        self.h = h
    end
    def area()
        return area_rect(self.w, self.h)
    end
end
```

Note: `area_circle` can call the private `_round` from within the module — **inside** a module everything is visible. Privacy only applies *toward the outside*. Likewise, the `Circle.area()` method can use the top-level `area_circle` function directly (without a prefix).

9.4.3 `lib/shapes.cobra` — The "Barrel" Module

A **barrel** module collects the public names of several submodules at a single point; this way users import from one module rather than from four separate places. To re-publish a name, we pull it in (`import ... as`) and then explicitly `export` it:

```
# shapes.cobra — "barrel" module.
# Collects the public names of submodules under a single roof; this way
# users import from one place. To re-export a name, it pulls the name in
# (`import ... as`), then re-publishes it with `export`.
from "geometry.cobra" import area_circle as _area_circle
from "geometry.cobra" import area_rect   as _area_rect
from "geometry.cobra" import Circle       as _Circle
from "geometry.cobra" import Rect         as _Rect

export let area_circle = _area_circle
export let area_rect   = _area_rect
export let Circle      = _Circle
export let Rect        = _Rect
```

> **Tip:** Here we put a _ prefix on the names we pulled in (`_area_circle`), then `export`ed the **clean** name. This little dance separates the private local name from the public re-exported name. As the project grows, users reach the entire geometry API with a single `from "lib/shapes.cobra" import *` — without needing to know how the submodules are split.

9.4.4 `main.cobra` — The Entry Point

Finally, the entry point that brings everything together. We use `config` as a module object (prefixed), while we use the barrel with `import *` without a prefix:

```
# main.cobra — application entry point.
# Module paths are ALWAYS relative to the folder of the importing file;
# since this file is in the root folder, we point to modules in the
# subfolder with "lib/...".

import "lib/config.cobra"                # module object: config.APP_NAME
from "lib/shapes.cobra" import *         # barrel: all public names without a prefix

print(config.APP_NAME + " v" + config.VERSION)

let garden = Circle(5)
let pool = Rect(8, 3)

print("Circle area: " + str(garden.area()))
print("Rectangle area: " + str(pool.area()))
print("Free function: " + str(area_circle(1)))
```

When we run the program from the root folder:

```
cobra main.cobra

Geometry Workshop v1.0.0
Circle area: 78.54
Rectangle area: 24.0
Free function: 3.14
```

You get the same output with `cobra --vm main.cobra` too — the two engines behave exactly the same. This small example carries all the hallmarks of a healthy module design: each file has a single responsibility, the public API is deliberately chosen with `export`, the helpers are private, and the dependencies flow in one direction (`main` → `shapes` → `geometry` → `config`).

9.5 The Priority of Built-in Modules

Cobra offers a set of **built-in (native) modules** such as `math`, `fs`, `json`, and `http`. These are imported without the `.cobra` extension, by their names alone:

```
import "math"
print(math.sqrt(16))    # 4.0
```

An important rule: **built-in modules take priority over files with the same name.** Even if there is a file named `math.cobra` in your folder, `import "math"` always loads the built-in `math` module — not your file:

```
# Even if there is a math.cobra in the folder:
import "math"
print(math.pi)          # 3.141592653589793  (built-in)
print(math.sqrt(16))    # 4.0
```

If you mean your own file, you must give an explicit file path by adding its extension (`import "math.cobra"`); an extensionless name always selects the built-in.

> **Warning:** Because of this priority, avoid giving your own modules names that collide with the built-ins (`math`, `os`, `time`, `json`, `re`, `http`, `fs`, `net`, `proc`, `test`, `runtime`). Otherwise an extensionless `import` always goes to the built-in, and your file is never loaded.

9.6 Compiling and Packaging: `cobra build` and `--bundle`

Every time you run a program, Cobra parses and compiles it. To do this step once and store the result, you use `cobra build`; this writes a `.cobrac` **bytecode** file next to the source file:

```
cobra build main.cobra
# wrote main.cobrac
```

The produced `.cobrac` is directly executable and, since it skips the parse/compile steps, starts faster:

```
cobra main.cobrac
# Geometry Workshop v1.0.0
# Circle area: 78.54
# ...
```

> **Note:** An ordinary `cobra build` compiles only the main file; the imported `.cobra` modules continue to be loaded **from disk at runtime**. So if you are going to distribute the `.cobrac` file, you also have to carry the `lib/` folder along with it — otherwise the modules cannot be found.

9.6.1 Single-File Package: `--bundle`

To embed the entire application into a single, self-contained file, you add the `--bundle` flag. This embeds **all imported `.cobra` modules** inside the `.cobrac` file:

```
cobra build --bundle main.cobra
# wrote main.cobrac (3 modules bundled)
```

"3 modules bundled" — `config.cobra`, `geometry.cobra`, and `shapes.cobra` were all three embedded. Now this single file works even if the `lib/` folder has been deleted:

```
rm -rf lib/          # remove the module files
cobra main.cobrac    # still works — the modules are embedded
# Geometry Workshop v1.0.0
# Circle area: 78.54
# Rectangle area: 24.0
# Free function: 3.14
```

This is the cleanest way to distribute a Cobra application: a single `.cobrac` file, with no external source dependency.

> **Warning:** `--bundle` embeds only `.cobra` modules. Native plugins (`.so` files) are not embedded because they are compiled to machine code; these are still loaded from disk at runtime. When distributing an application that depends on a `.so` plugin, you must carry the plugin alongside the package.

9.7 Summary

In this chapter we saw Cobra's module system from start to finish:

- Every `.cobra` file is a module. `import "lib/x.cobra"` executes the file **once** (caching the result) and binds a **module object** named `x`; you access its members with `x.member`. `import "x.cobra" as y` gives an alias.
- Module paths are **always relative to the folder of the importing file** (in the REPL, relative to the current working directory, the cwd). **Circular** imports are an error.
- **Private by default**: a module exposes to the outside only what it marks with `export`. `export` works on `let`, `const`, `def`, `struct`, and `record`. Unmarked names (by convention like `_helper`) stay private; what provides the real privacy is not _ but the absence of `export`.
- `from "path" import a, b as c` binds selected public members **directly** into the scope (without a prefix). A typo or a name that is not exported gives an error **at import time** — early and clear, not late and mysterious.
- `from "x" import *` binds all public names. To re-export them, you must separately `export` them; "barrel" modules do exactly this (pull them in with `import ... as` and re-publish with `export let`).
- **Built-in (native) modules** take priority over files with the same name; an extensionless `import "math"` always selects the built-in.

- `cobra build file.cobra` produces a fast-starting `.cobrac` bytecode (the modules are still loaded from disk). `cobra build --bundle app.cobra` embeds all imported `.cobra` modules into a single file, creating a self-contained package (`.so` plugins are not embedded).

Modules are the glue that turns small programs into real applications: they separate responsibilities, force public APIs to be deliberately designed, and make code reusable. In the next chapter, we will look at how to run the functions these modules offer **concurrently** — with Cobra's true, GIL-free parallelism.

Chapter 10

Concurrency and True Parallelism

Most scripting languages give you the *illusion* of concurrency. They hide behind a single interpreter lock (the GIL, Global Interpreter Lock): you can write multiple "threads," but at any given moment only one of them runs code. This works for *wait-heavy* (I/O-bound) work like networking and disk, but you hit a wall when you want to split a compute-heavy (CPU-bound) job across four cores for a fourfold speedup: the cores are there, but the language cannot use them.

Cobra is different. **Cobra has no GIL.** When you start a task (`task`), that task runs on its own operating-system thread, **truly at the same time** as other tasks, across multiple CPU cores. This means compute-heavy work is genuinely parallelized; no extra syntax or trick is required. There is a price: you must now explicitly protect shared mutable state yourself. Cobra gives you a plain but complete toolkit for providing that protection: tasks, channels, mutexes, and parallel built-ins.

In this chapter we will see: starting tasks with `async def` and collecting their results with `await`; how errors are re-thrown at the `await` point; protecting shared state with `Mutex` and `RWMutex`; streaming data between tasks with `Channel` and building a worker pool; waiting on multiple channels with `select` (including time-outs and fan-in); and one-line data parallelism with `pmap` / `pfilter` / `pforeach` / `preduce`. Every example has been run and verified.

10.1 No GIL: Tasks Truly Run in Parallel

When you define a function with `async def`, *calling* it does not run the body immediately; instead it **returns a task instantly**, and the body starts running in the background on its own thread. When you want the result, you use `await`; `await` waits until the task finishes and yields its return value.

```
import "time"

async def work(n)
    time.sleep(0.05)
    return n * n
end

# Start three tasks at the same time, then collect them all in order.
let results = await [work(1), work(2), work(3)]
print(results)            # [1, 4, 9]

# Awaiting a single task works the same way.
```

```
let single = await work(4)
print(single)             # 16
```

Here the three `work(...)` calls start almost simultaneously. Even though each sleeps for 50 ms, because they run in parallel they all finish in roughly 50 ms — not 150 ms total. `await [t1, t2, t3]` takes a **list of tasks** and returns **the results in input order**: it does not matter which finishes first, the output is always `[1, 4, 9]`.

We can prove this is genuinely parallel with a compute-heavy job. Below we do the same heavy computation first serially, then in parallel:

```
import "time"
import "runtime"

print("number of cores:", runtime.num_cpu())

# Pure CPU work: no sleep at all, just computation.
def heavy(n)
    let total = 0
    for i in range(3000000)
        total = total + (i % 7)
    end
    return total
end

async def aheavy(n)
    return heavy(n)
end

# Serial: do four jobs one after another.
let t0 = time.clock()
for n in range(4)
    heavy(n)
end
let serial = time.clock() - t0

# Parallel: start four tasks at the same time and collect them.
let t1 = time.clock()
await [aheavy(0), aheavy(1), aheavy(2), aheavy(3)]
let parallel = time.clock() - t1

print("is parallel faster than serial:", parallel < serial)
```

On a machine with multiple cores, the output is:

```
number of cores: 10
is parallel faster than serial: true
```

The parallel version is measurably faster than the serial one, because the four `heavy` calls genuinely run at the same time on four separate cores. In a single-core (or GIL-bound) language this would not be possible.

> **Note:** You do not have to `await` a task in order to start it. The call `work(1)` starts the task immediately; you can hold the returned task value in a variable, do your work, and fetch the result later with `await`. For "fire and forget" style tasks you may never wait for the return at all — but tasks still running when the program ends are dropped (their results are lost).

> **Tip:** You can start many tasks in a loop and accumulate them in a list, then collect them all with a single `await`: `let tasks = []` … `tasks.push(work(i))` … `let all = await tasks`.

10.2 Errors in Tasks: Re-thrown at the `await` Point

If an error occurs in a task's body (a `throw` or a runtime error), that error is re-thrown *not* where you started the task, but **at the point where you `await` it**. This way you catch the error with the usual `try/catch`:

```
async def boom()
    throw "the job blew up"
end

try
    await boom()
catch err
    print("caught:", err)   # caught: the job blew up
end
```

This behavior matters: a task does not silently blow up and disappear in the background. The error stays "stored" in the task until you demand its result (`await`), then it is thrown back to you. When you await a list of tasks with `await [...]`, the **first error** (by input order) propagates.

> **Warning:** If you never `await` a task, you will never see an error in its body — the error stays stored in that task and falls away with the task when the program ends. If you care about a task's success, be sure to `await` it.

10.3 Why Is Shared State Dangerous?

Because tasks truly run in parallel, if two of them write to the *same* mutable value at the same time, a **race condition** occurs. The classic example is a shared counter. `total = total + 1` looks like a single step, but it is actually three: read, add one, write back. If two tasks interleave, one of the updates is lost.

```
# LOCKLESS — intentionally buggy: a lost-update demonstration
let total = 0
def heavy()
    let s = 0
    for i in range(20000)
        s = s + 1
    end
    return s
end
async def bump()
    for i in range(1000)
        heavy()            # a delay to increase the chance of interleaving
        total = total + 1
    end
end
await [bump(), bump(), bump(), bump()]
print("lockless expected 4000, actual:", total)
```

When you run this program several times, the output varies:

```
lockless expected 4000, actual: 4000
lockless expected 4000, actual: 3997
lockless expected 4000, actual: 3999
```

Sometimes correct (4000), most of the time short. This is the most insidious bug in concurrency: it occasionally gives the right answer, so it slips past tests. The solution is to **explicitly synchronize** shared state.

> **Warning:** By design, tasks that touch separate global variables are always safe (the global scope is synchronized internally). The danger appears when you share the *same* value and mutate it concurrently. If multiple tasks will modify a shared number, list, dict, or struct field, you must protect it with a mutex.

10.4 Mutual Exclusion with `Mutex`

A `Mutex` (mutual exclusion) lets only one task at a time enter the region it guards. `m.lock()` acquires the lock (waiting if someone else holds it), `m.unlock()` releases it. The code between the two is the **critical region**: only one task is there at a time.

Let's fix the earlier race condition with a `Mutex`:

```
let m = Mutex()
let total = 0

async def bump()
    m.lock()
    total = total + 1     # critical region: one task at a time
    m.unlock()
end

let tasks = []
for i in range(100)
    tasks.push(bump())
end
await tasks
print(total)              # 100 — always, without exception
```

Now the `read-add-write` triple becomes indivisible (atomic); the output is exactly `100` on every run.

`Mutex` has a third method: `try_lock()`. It tries to acquire the lock *without blocking*; it returns `true` if it can acquire it, or `false` without waiting if the lock is already held.

```
let m = Mutex()
print(m.try_lock())     # true  — the lock was free, acquired
print(m.try_lock())     # false — already locked, returned without waiting
m.unlock()
```

> **Tip:** Always pair `lock` with `unlock`. If the critical region contains code that can throw an error, use `try/finally` to make sure the lock is released: `m.lock()` … `try` … operation … `finally` … `m.unlock()` … `end`. This way even an error does not hold the lock forever.

10.4.1 Protecting a shared dict

Concurrently modifying a `dict` (or a struct field) from multiple tasks is dangerous; Cobra's safety net catches this and turns it into a *catchable* error of the form `"data race in task"`, but that is only a net — the correct approach is to protect the dict with a `Mutex`:

```
let cache = {}
let m = Mutex()

async def store(k, v)
    m.lock()
    cache[k] = v          # shared dict write: under the lock
    m.unlock()
end

let tasks = []
for i in range(50)
    tasks.push(store("k" + str(i), i))
end
await tasks
print(len(cache))         # 50
print(cache["k49"])       # 49
```

> **Warning:** Lockless concurrent modification of a shared *dict* or instance field is worse than lockless modification of a shared number: instead of silently giving a wrong result, it can cause a fatal data race. Always lock every dict that multiple tasks write to.

10.5 `RWMutex`: Many Readers, One Writer

Some shared state is *read-heavy*: many tasks read constantly, while one occasionally writes. A plain `Mutex` is wasteful here, because there is no reason for readers to block one another — only the writer must exclude everyone. `RWMutex` (a readers-writer lock) provides exactly this:

- `rlock()` / `runlock()` — **read lock**: many readers can hold it at the same time.
- `lock()` / `unlock()` — **write lock**: single, exclusive; while held, no reader can enter.
- `try_lock()` — tries the write lock without blocking.

```
let rw = RWMutex()
let config = {"limit": 10}

async def reader(id)
    rw.rlock()
    let v = config["limit"]    # many readers can be here at the same time
    rw.runlock()
    return v
end

async def writer(v)
    rw.lock()
    config["limit"] = v        # while writing, no one can read
    rw.unlock()
```

```
end

await [reader(1), reader(2), writer(20), reader(3)]
print(config["limit"])         # 20
```

> **Tip:** Keep the rule simple: if you are only *reading* a value use `rlock`/`runlock`, if you are *modifying* it use `lock`/`unlock`. Accidentally writing under a read lock instead of a write lock creates a race condition.

10.6 `Channel`: Streaming Data Between Tasks

Locks *protect* shared state; channels, on the other hand, *carry* data from one task to another. A `Channel` is a type-safe pipe between tasks: you put a value in at one end with `send`, and take it out at the other end with `recv`.

There are two kinds of channels:

- `Channel()` — **unbuffered**: `send` waits until a `recv` takes the value (and vice versa). This is a synchronous handshake between two tasks.
- `Channel(n)` — **buffered**: the channel can hold up to `n` values; `send` does not wait until the buffer is full.

```
# Unbuffered channel: one task produces, the main flow consumes.
let ch = Channel()
async def produce()
    ch.send(42)
end
produce()
print(ch.recv())          # 42

# Buffered channel: after closing, the buffer drains first, then null arrives.
let buf = Channel(3)
buf.send(1)
buf.send(2)
print(len(buf))           # 2  — the number of pending buffered values
buf.close()
print(buf.recv())         # 1
print(buf.recv())         # 2
print(buf.recv())         # null — buffer empty + channel closed
```

Two things to note: `len(ch)` gives the number of pending values in the channel; and after `close()` is called, `recv` first drains the values remaining in the buffer, **then starts returning `null`**. This `null` is ideal for use in loops as a "channel done" signal.

`send`/`recv` also have non-blocking siblings: `try_send(v)` and `try_recv()`.

```
let ch = Channel(1)
print(ch.try_send(1))   # true  — there was room, sent
print(ch.try_send(2))   # false — buffer full, returned without waiting
print(ch.try_recv())    # 1
print(ch.try_recv())    # null  — channel empty, returned without waiting
```

> **Note:** Sending to a closed channel is an error. The general rule: the **sending side closes** the channel (only it knows that no more data is coming), not the receiving side.

10.6.1 Worker pool

The most powerful use of channels is the **worker pool**: a fixed number of worker tasks pull work from a "jobs" channel and write the result to a "results" channel. This pattern is the standard way to distribute a large number of independent jobs across a limited number of parallel workers.

```
# 3 workers grind through 8 jobs in parallel.
let jobs = Channel(10)
let results = Channel(10)

async def worker()
    while true
        let job = jobs.recv()
        if job == null         # channel closed: this worker is done
            return null
        end
        results.send(job * job)
    end
end

# Start the worker pool.
let pool = []
for i in range(3)
    pool.push(worker())
end

# Feed the jobs, then close the channel ("done" signal to the workers).
for n in range(1, 9)
    jobs.send(n)
end
jobs.close()

# Collect the results.
let collected = []
for i in range(1, 9)
    collected.push(results.recv())
end
await pool                     # make sure all workers finished cleanly

collected.sort()               # workers are parallel, order may vary; let's sort
print(collected)               # [1, 4, 9, 16, 25, 36, 49, 64]
```

The three workers split the eight jobs among themselves. `jobs.close()` ensures that each worker's `recv` call eventually sees `null`; this way the workers exit their loops cleanly. The arrival order of the results is non-deterministic because it depends on worker speed; that is why we `sort()`ed the list before comparing.

> **Tip:** Choose the number of workers based on the kind of work. For compute-heavy work, as many workers as `runtime.num_cpu()` is typically best; for wait-heavy (network/disk) work, you can use far more workers than the number of cores.

10.7 `select`: Waiting on Multiple Channels

Sometimes you want to deal not with a single channel but with whichever of *several* channels becomes ready first. `select(ops)` does exactly this: it takes a list of channels and acts on the first one that is ready, returning a dict that describes which was selected: `{index, value, ok, send}`.

Each entry in the `ops` list is either a channel (a receive/`recv` operation) or a two-element list of the form `[channel, value]` (a send/`send` operation).

```
import "time"
let a = Channel()
let b = Channel()

async def fill(ch, v, d)
    time.sleep(d)
    ch.send(v)
end
fill(a, "a-result", 0.02)   # a becomes ready earlier
fill(b, "b-result", 0.10)

let r = select([a, b])
print(r["index"], r["value"], r["ok"], r["send"])
# 0 a-result true false   — index 0 (a) was selected, it was a recv (send=false)
```

Because `a` becomes ready earlier (`0.02 s`), `select` chooses it: `index` is 0, `value` is the received value, `ok` says the channel is open, and `send` says this was a receive, not a send (`false`).

`select` has a second, non-blocking form: `select(ops, default)`. If no operation is ready, it does not wait but returns `default` directly.

```
let empty = Channel()
let r2 = select([empty], "empty")
print(r2)                  # empty — no channel was ready
```

10.7.1 Timeout with `select`

The most common use of `select` is the **timeout**: you wait on a work channel and a timer channel at the same time and act according to whichever arrives first. If the work arrives before the timer, you use the result; if the timer arrives first, you say "timeout."

```
import "time"
let work = Channel()
let timeout = Channel()

# Slow work: sends the result after 200 ms.
async def slow()
    time.sleep(0.20)
    work.send("done")
end
slow()

# Timer: fires after 50 ms.
async def fire()
```

```
    time.sleep(0.05)
    timeout.send(true)
end
fire()

let r = select([work, timeout])
if r["index"] == 0
    print("result:", r["value"])
else
    print("timed out")    # the timer arrived first
end
```

Because the work takes 200 ms while the timer fires at 50 ms, the output is:

```
timed out
```

10.7.2 Fan-in: collecting many producers into one channel

The sibling pattern to `select` is **fan-in**: collecting the output of multiple producer tasks into a single channel. Here you do not even need `select` — all producers write to the same channel, a separate task waits for all the producers and closes the channel, and the consumer reads until the channel returns `null`:

```
let out = Channel(10)

async def producer(name, count)
    for i in range(count)
        out.send(name + "-" + str(i))
    end
end

let producers = [producer("A", 3), producer("B", 3)]

# A separate task that closes the channel once all producers are done.
async def closer()
    await producers
    out.close()
end
closer()

# Read until the channel is closed and drained (null).
let received = []
while true
    let v = out.recv()
    if v == null
        break
    end
    received.push(v)
end
received.sort()            # two producers run in parallel; fix the order
print(received)            # ["A-0", "A-1", "A-2", "B-0", "B-1", "B-2"]
```

The `closer` task is the clean way to avoid both pitfalls of closing the channel too early (before the producers are done) or never closing it at all: `await producers` makes sure all producers have finished, then `out.close()` tells the consumer "the stream is over."

10.8 Data Parallelism: `pmap`, `pfilter`, `pforeach`, `preduce`

Most parallelism needs actually come down to a single pattern: "apply this operation to every element of this list." You do not even need to set up tasks by hand for this; Cobra's **parallel built-ins** automatically distribute the work across all cores. They have the same call form as the ordinary `map`/`filter`/`foreach`/`reduce`; their only difference is that they run in parallel.

```
def sq(n)
    return n * n
end
def is_even(n)
    return n % 2 == 0
end
def add(a, b)
    return a + b
end

print(pmap(sq, [1, 2, 3, 4]))           # [1, 4, 9, 16]  — order is preserved
print(pfilter(is_even, [1, 2, 3, 4, 5, 6]))  # [2, 4, 6]
print(preduce(add, [1, 2, 3, 4, 5]))    # 15

# pforeach for side effects: lock the shared list.
let out = []
let m = Mutex()
def collect(n)
    m.lock()
    out.push(n)
    m.unlock()
end
pforeach(collect, [10, 20, 30])
out.sort()
print(out)                              # [10, 20, 30]
```

There are two important guarantees:

1. **Result order is preserved.** `pmap` and `pfilter`, even though they process elements in parallel, arrange the output by input order. Above, `pmap(sq, [1,2,3,4])` always gives `[1, 4, 9, 16]`.
2. **The first error propagates.** If the operation on an element throws an error, then, just as in `map`, the **first** error by input order is thrown out of `pmap`.

`pforeach`, on the other hand, returns no result; it just calls the function for each element (for side effects). In the example above, because `collect` writes to the shared `out` list, we protected it with a `Mutex`; otherwise the parallel `push`es would race. Since the workers run in parallel, the order of additions to `out` may vary, which is why we `sort()`ed before comparing.

> **Tip:** If you have a pure (side-effect-free) transformation, `pmap` is often both shorter and safer than setting up tasks by hand: no locks, no channels, order already guaranteed. When you need a side effect (`pforeach`), don't forget to lock the shared state.

> **Note:** Parallelism has a startup cost (thread setup). On very small lists, or when the work per element is very light, plain `map` may be faster. Prefer `pmap` when the work per element is genuinely heavy.

10.9 HTTP Server: Handlers Run in Parallel

Concurrency does not arise only from your own `async def`s; some built-in modules call your handlers in parallel *on your behalf*. The most important is `http.serve`: for each incoming request it calls your handler function in a separate task. So if two requests arrive at the same time, your handler may run **twice at the same time**.

The practical consequence is this: in a web server you must protect shared state (a request counter, a session table, a cache) with a lock just as you would in your own tasks.

```
import "http"

let m = Mutex()
let hits = 0

def handler(req)
    m.lock()
    hits = hits + 1            # shared counter: handlers run in parallel
    let n = hits
    m.unlock()
    return "request #" + str(n)
end

# http.serve(":8080", handler)   # each request runs in its own task
```

> **Warning:** Modifying a shared dict or counter without a lock inside `http.serve` handlers is exactly the kind of bug that blows up under load (many concurrent requests) and slips past local testing. Lock every shared mutable state in server code.

10.10 Summary

In Cobra, concurrency is not an illusion but real: **there is no GIL** and tasks genuinely run in parallel across multiple CPU cores. This power comes with the responsibility of explicitly synchronizing shared state. What we saw in this chapter:

- **Tasks and `await`.** Calling a function defined with `async def` returns a task immediately and the body starts running in the background in parallel. `await task` collects its result, and `await [t1, t2, ...]` collects the results of multiple tasks **in input order**. An error in a task is re-thrown not where you started the task but **at the `await` point**; in `await [...]` the first error propagates.
- **Race conditions.** If two tasks modify the same value without a lock, updates are lost (numbers) or a fatal error occurs (dict/struct fields). Separate globals are safe, *shared* values are not.
- **`Mutex`.** Limit the critical region to a single task with `lock` / `unlock`; `try_lock` tries without blocking. Protect shared counters, lists, and dicts with it.
- **`RWMutex`.** For read-heavy state: many `rlock`/`runlock` readers at the same time, but a single exclusive `lock`/`unlock` writer.
- **`Channel`.** `Channel()` (unbuffered, synchronous handshake) and `Channel(n)` (buffered). `send` / `recv` / `close` / `try_send` / `try_recv` / `len(ch)`. After closing, `recv` first drains the buffer, then returns `null` — this is the "stream done" signal. The **sending side** closes the channel. It is the basis of the worker pool pattern.
- **`select`.** Waits on multiple channels and selects the first one ready; result `{index, value, ok, send}`. It is the key to the timeout and fan-in patterns. `select(ops, default)` is non-blocking.

- **Parallel built-ins.** `pmap` / `pfilter` / `pforeach` / `preduce` reduce data parallelism to a single line: **result order is preserved**, **the first error propagates**, no extra syntax required.
- **`http.serve`** handlers run in parallel — lock shared state just as in your own tasks.

The next step: combining these primitives in a real program. If you have a compute-heavy job, try `pmap` first; if you have a producer-consumer flow, set up a worker pool with `Channel`; if a shared resource is modified by multiple tasks, put it behind a `Mutex` (or an `RWMutex` if it is read-heavy). The absence of the GIL is what makes these patterns deliver real speed gains — as long as you protect shared state with discipline.

Chapter 11

The Standard Library

What makes a language "usable" is often less the core syntax than the batteries that come with it: reading a file, parsing an API response, measuring time, validating text, running a command. Cobra covers all of these jobs with a small but carefully designed set of **built-in modules**. Unlike in other languages, these modules are not downloaded from a package manager; they come ready inside the interpreter and are imported by their names:

```
import "math"
print(math.sqrt(16))   # 4.0
```

A module's members are always accessed with a dot (`math.sqrt`, `fs.read`, `json.parse`). Built-in modules take precedence over local `.cobra` files of the same name; that is, `import "math"` always brings in the real math module. A module's failures (a nonexistent file, invalid JSON, a network error) are **catchable** runtime errors: you can handle them with `try/catch`.

In this chapter we tour the entire standard library one by one. We will introduce each module not with a toy example but with a small engineering task you will genuinely encounter: processing a log file, decoding an API response, validating an email address, running a subprocess. All runnable examples have actually been run and their outputs verified; the output comments starting with `#` in the code blocks are the program's real output.

> **Note:** Cobra has no string formatting (f-strings, interpolation). We produce output by giving `print` multiple arguments ("a:", x) or by joining with `+` and `str(...)`. You will see both styles in the examples in this chapter.

11.1 fs — File System

The `fs` module provides file input/output. All paths are relative to the working directory (cwd). Members: `read`, `write`, `append`, `read_lines`, `write_lines`, `exists`, `size`, `remove`, `rename`, `mkdir`, `list_dir`, `is_dir`, `cwd`.

A typical task: reading an application's log file line by line and summarizing it by level. The program below first writes a sample log, then reads it back, extracts `INFO`/`WARN`/`ERROR` counts, and pulls out only the error lines into a separate file.

```
# Let's create an application log file and process it line by line.
import "fs"

let log_path = "/private/tmp/cobrabook/app.log"

# First let's write a sample log (in real life the application produces this).
fs.write_lines(log_path, [
    "2026-06-27 09:01:12 INFO  started",
    "2026-06-27 09:01:13 WARN  slow response",
    "2026-06-27 09:02:01 ERROR connection dropped",
    "2026-06-27 09:02:05 INFO  retried",
    "2026-06-27 09:02:06 ERROR timeout"
])

# Did the file actually get created?
print("exists:", fs.exists(log_path))      # exists: true
print("size:", fs.size(log_path) > 0)      # size: true

# Count by level.
let counts = {}
for line in fs.read_lines(log_path)
    let fields = line.split(" ")
    let level = ""
    for f in fields
        if f == "INFO" or f == "WARN" or f == "ERROR"
            level = f
        end
    end
    if level != ""
        counts[level] = counts.get(level, 0) + 1
    end
end

for pair in counts.items()
    print(pair[0] + ": " + str(pair[1]))
end
# INFO: 2
# WARN: 1
# ERROR: 2

# Let's summarize only the ERROR lines into a separate file.
def has_error(l)
    return l.contains("ERROR")
end
let errors = filter(has_error, fs.read_lines(log_path))
fs.write(log_path + ".errors", "\n".join(errors))
print("error lines:", len(fs.read_lines(log_path + ".errors")))  # error lines: 2

# Cleanup.
fs.remove(log_path)
fs.remove(log_path + ".errors")
print("cleaned up:", not fs.exists(log_path))   # cleaned up: true
```

Here the `read_lines`/`write_lines` pair made our job easier by treating the file as a list of lines; if we wanted

the whole content as a single string, we would use `read` and `write`. `append`, meanwhile, adds to the end of an existing file (ideal for accumulating logs). For working with directories there are `mkdir`, `list_dir`, `is_dir`, and `cwd`, which gives the current directory.

> **Tip:** Higher-order functions like `filter` require a function name to be passed; Cobra has no `lambda`. That is why we defined `has_error` with `def` first and gave its name to `filter`.

> **Warning:** `fs.read` throws an error on a nonexistent file. Check first with `fs.exists(path)` or wrap the call in `try/catch`.

11.2 math — Mathematics

`math` provides the basic constants and functions for numerical computation: `pi`, `e`, `sqrt`, `abs`, `floor`, `ceil`, `round`, `pow`, `sin`, `cos`, `tan`, `log`, `exp`, `min`, `max`, `random`. Let's compute the mean and standard deviation of a small series of measurements.

```
# Let's compute basic statistics of a series of measurements with math.
import "math"

let samples = [12.5, 9.0, 15.5, 11.0, 13.0]

# Mean
let total = 0.0
for x in samples
    total = total + x
end
let mean = total / float(len(samples))

# Standard deviation (population)
let var_sum = 0.0
for x in samples
    var_sum = var_sum + math.pow(x - mean, 2)
end
let stddev = math.sqrt(var_sum / float(len(samples)))

print("mean:", mean)                                  # mean: 12.2
# round() returns an integer; for decimal places you must divide by a float.
print("std dev:", math.round(stddev * 100) / 100.0)   # std dev: 2.16
print("smallest:", math.min(samples))                 # smallest: 9.0
print("largest:", math.max(samples))                  # largest: 15.5

# Banker's rounding: a 0.5 exactly in the middle goes to the nearest EVEN number.
print("round(2.5):", math.round(2.5))                 # round(2.5): 2
print("round(3.5):", math.round(3.5))                 # round(3.5): 4
print("pi:", math.round(math.pi * 1000) / 1000.0)     # pi: 3.142
```

Note two points. First, when computing the mean we used `float(len(...))`: in Cobra the division of two integers is **integer division** (`5 / 2` is `2`), so if we want a decimal result we must make at least one side a `float`. Second, `math.round` uses **banker's rounding**: a value exactly in the middle (`2.5`) is rounded to the

nearest *even* number, so `2.5` → `2` but `3.5` → `4`. This is the standard behavior that reduces rounding bias in large data sets.

`math.min` and `math.max` can be called both with separate arguments (`math.max(3, 7, 1)`) and with a single list (`math.max(samples)`). For a random number, `math.random()` gives a float in the range `[0, 1)`.

> **Note:** `math.round` returns an **integer**. When you want to round to a specific number of decimal places, first scale (`* 100`), round, then divide by a `float` (`/ 100.0`) as in the example.

11.3 os — Operating System

The `os` module opens up the process environment: `args` (script path + arguments), `env`, `setenv`, `platform`, `time`. Let's read the arguments and environment variables as a command-line tool would.

```
# Let's read the runtime environment and command-line arguments with os.
import "os"

# os.args() returns a list; [0] is the path of the running script, the rest are
# the user's arguments.
let args = os.args()
print("script:", args[0].endswith(".cobra"))   # script: true
print("platform:", os.platform() == "darwin" or os.platform() == "linux" or os.platform() == "windows") #

# Environment variable: the second argument is the default value.
let mode = os.env("APP_MODE", "development")
print("mode:", mode)                           # mode: development

# Let's set a variable and read it back.
os.setenv("APP_MODE", "production")
print("new mode:", os.env("APP_MODE", "development"))  # new mode: production

# os.time() returns Unix epoch seconds as a decimal (float).
print("time reasonable:", os.time() > 1700000000.0)    # time reasonable: true
```

`os.args` and `os.platform` are both **functions**; they must be called with parentheses (`os.args()`, `os.platform()`). `os.env(name, default?)` takes a default as its second argument; if the variable is not defined it returns that, and if no default is given either it returns `null`. This is ergonomic for programs that read configuration from the environment: `os.env("PORT", "8080")`.

> **Tip:** `os.time()` gives the seconds elapsed since the Unix epoch as a *float*; for calendar formatting you can pass it to `time.format` (see the `time` module in the next section).

11.4 json — JSON

The `json` module has two functions: `parse` (text → value) and `stringify(value, indent?)` (value → text). On parsing, dict **key order is preserved** and integers are distinguished from decimals. Let's parse an API response as if it came from an HTTP client.

```
# Let's parse an API response with json and re-serialize it.
import "json"

# A response text as if it came from an HTTP client.
let response = "{\"user\": {\"id\": 42, \"name\": \"Ada\", \"active\": true}, \"roles\": [\"admin\", \"edi

let data = json.parse(response)
print("name:", data["user"]["name"])     # name: Ada
print("role count:", len(data["roles"])) # role count: 2
print("first role:", data["roles"][0])   # first role: admin
print("active:", data["user"]["active"]) # active: true

# Let's add a field and convert it back to a string. Key order is preserved.
data["user"]["verified"] = true
let out = json.stringify(data["user"])
print(out)   # {"id":42,"name":"Ada","active":true,"verified":true}

# Indented (pretty) output.
print(json.stringify(["a", "b"], 2))
# [
#   "a",
#   "b"
# ]

# Non-string keys turn into strings.
print(json.stringify({1: "one", 2: "two"}))  # {"1":"one","2":"two"}
```

Parsed JSON objects turn into Cobra's usual dicts and lists; you access nested structures by chaining like `data["user"]["name"]`. If you give `stringify` an indent count as its second argument you get readable (pretty) output; if you do not, a compact form with no whitespace is produced. Since Cobra dict keys can also be integers or booleans, these are converted to strings when translated to JSON (`{1: ...}` → `{"1": ...}`), because JSON recognizes only string keys.

> **Warning:** `json.parse` throws an error on invalid JSON. Always wrap external data with `try/catch`:
>
> ```
> try
>     let data = json.parse(text)
> catch err
>     print("invalid JSON:", err)
> end
> ```

11.5 time — Time

`time` provides both duration measurement and calendar operations: `now`, `clock`, `sleep`, `format`, `parse`, `parts`. For measuring duration use the **monotonic** `clock`; the calendar functions (`format`/`parse`/`parts`) work in UTC and support strftime directives (`%Y %m %d %H %M %S %a %A %b %B %p %I %j %y %%`).

```
# Let's measure duration and format timestamps with time.
```

```
import "time"

# clock() is a monotonic clock; it is ideal for measuring duration.
let start = time.clock()
let total = 0
for i in range(1000000)
    total = total + i
end
let elapsed = time.clock() - start
print("total correct:", total == 499999500000)  # total correct: true
print("duration measured:", elapsed >= 0.0)      # duration measured: true

# Let's format a fixed epoch timestamp (UTC).
# 1700000000 = 2023-11-14 22:13:20 UTC
let ts = 1700000000
print(time.format(ts, "%Y-%m-%d %H:%M:%S"))   # 2023-11-14 22:13:20

# Let's parse a date and break it into its parts.
let parsed = time.parse("2026-06-27", "%Y-%m-%d")
let p = time.parts(parsed)
print("year:", p["year"], "month:", p["month"], "day:", p["day"])  # year: 2026 month: 6 day: 27
print("day of week (Mon=0):", p["weekday"])  # day of week (Mon=0): 5
```

`clock` is not absolute calendar time but only for measuring *elapsed time*: you call it at the start and end of a measurement and take the difference. For an absolute timestamp there is `now()`. `format(ts, fmt)` converts an epoch timestamp to readable text; `parse(s, fmt)` does the reverse, and `parts(ts)` returns a dict that breaks the timestamp into year/month/day/hour/minute/second/ weekday/yearday fields. The day of the week is given with the Monday = 0 convention, so June 27, 2026 is a Saturday (5).

> **Tip:** To pause an operation for a specific duration use `time.sleep(seconds)`; the argument is a float, so `time.sleep(0.05)` waits 50 milliseconds. We will use this especially in the concurrency examples to wait for a server to come up.

11.6 re — Regular Expressions

The `re` module provides text matching and extraction: `matches`, `find`, `find_all`, `groups`, `replace`, `split`. Underneath lies the RE2 engine; this gives a linear-time guarantee but **does not support backreferences or lookaround**. Let's start with a classic validation task, email checking.

```
# Let's do email validation and field extraction with re.
import "re"

let email_pat = "^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\\.[a-zA-Z]{2,}$"

let addresses = ["ada@example.com", "invalid@", "bob.smith@mail.co.uk"]
for addr in addresses
    print(addr + " -> " + str(re.matches(email_pat, addr)))
end
# ada@example.com -> true
```

```
# invalid@ -> false
# bob.smith@mail.co.uk -> true

# Let's extract the date and level fields from a log line with groups.
let line = "2026-06-27 ERROR connection dropped"
let g = re.groups("(\\d{4}-\\d{2}-\\d{2})\\s+(\\w+)", line)
print("date:", g[1], "level:", g[2])  # date: 2026-06-27 level: ERROR

# Let's find all the numbers in the text.
print(re.find_all("\\d+", "3 apples, 12 pears, 7 plums"))  # ["3", "12", "7"]

# Let's collapse multiple spaces into a single space.
print(re.replace("\\s+", "a   b     c", " "))  # a b c

# Let's split a CSV-like line by comma OR semicolon.
print(re.split("[,;]", "first;last,age"))  # ["first", "last", "age"]
```

Note the signatures of the functions: `matches`, `find`, `find_all`, `groups`, and `split` take the pattern and the text in that order (`re.matches(pattern, text)`); `replace`, on the other hand, takes three arguments: `re.replace(pattern, text, new)`. `groups` returns the first match as a list `[whole_match, group1, group2, ...]`; that is why we accessed the first capture group with `g[1]`. If a group participates in no match, `null` is found at that position. Inside `replace` you can refer to capture groups with `$1`, `$2`.

> **Warning:** When writing directives like `\d`, `\s`, `\w` in Cobra strings you must **escape** the backslash: `"\\d+"`. Otherwise the string does not contain a literal `\d`. Also, because it is RE2, patterns like `(?=...)` (lookahead) or `\1` (backreference) **do not work** — if you need them you must handle the logic on the code side.

> **Note:** Cobra strings are byte-indexed; manually cutting a substring with an `s[i]` loop can corrupt UTF-8 text. For text extraction, prefer the `re` functions (or the rune-safe slice `s[a:b]`) over a loop.

11.7 http — HTTP Client and Server

`http` contains both a client and a server.

Client side: `get(url, headers?)`, `post(url, body, headers?)`, and the general `request(method, url, body?, headers?)`. All return a dict of the form `{status, body, headers}`. A network error is a catchable error; non-2xx status codes, on the other hand, are not considered errors, they are part of the result.

```
import "http"
import "json"

# Let's call a JSON API (the output depends on the environment/network).
let r = http.get("https://api.example.com/users/42")
if r["status"] == 200
    let user = json.parse(r["body"])
    print("name:", user["name"])
else
    print("error code:", r["status"])
end
```

> **Note:** Because the `http.get` example above makes a real network call, its output depends on the environment it runs in; that is why we do not show a fixed `#` output here. The server example below, on the other hand, is entirely local, so it has actually been run and verified.

Server side: `serve(addr, handler, limit?)` blocks at the given address and for each request calls your Cobra handler with a `{method, path, query, body, headers}` dict. The handler returns either a string (which becomes 200) or a `{status?, body?, headers?}` dict. **Handlers are parallel** (a separate task per request), so you must explicitly lock shared state. If the optional `limit` is given, the server returns after handling that many requests — this is practical for testing finite examples.

The program below both starts a small JSON API server and makes requests to it with the client, all in a single script. Note that we protect the shared `hits` counter with a `Mutex`.

```
# Let's set up a small JSON API server with http and call it with the client.
import "http"
import "json"
import "time"

# Shared counter; we protect it with a Mutex because handlers run in PARALLEL.
let hits = 0
let m = Mutex()

def handler(req)
    m.lock()
    hits = hits + 1
    let n = hits
    m.unlock()

    if req["path"] == "/ping"
        return "pong"
    end
    if req["path"] == "/stats"
        # If we return a dict, we serialize the JSON body manually.
        return {"status": 200,
                "body": json.stringify({"hits": n}),
                "headers": {"Content-Type": "application/json"}}
    end
    return {"status": 404, "body": "not found"}
end

# limit=3: serve() returns after 3 requests are handled (practical for testing).
async def run_server()
    http.serve("127.0.0.1:8088", handler, 3)
end

let srv = run_server()
time.sleep(0.2)   # wait for the server to come up

# Client side.
let base = "http://127.0.0.1:8088"
let r1 = http.get(base + "/ping")
print("ping:", r1["status"], r1["body"])   # ping: 200 pong

let r2 = http.get(base + "/stats")
```

```
let stats = json.parse(r2["body"])
print("stats hits:", stats["hits"])         # stats hits: 2

let r3 = http.get(base + "/none")
print("none:", r3["status"], r3["body"])    # none: 404 not found

await srv
```

We started the server inside an `async def` and waited a short while with `time.sleep`; this way the server began listening before the client made its first request. Because the `hits` counter in the handler increments on every request, its value is 2 when `/stats` is called (first `/ping`, then `/stats`). If a handler throws an error, Cobra turns it into a 500 and the server keeps running.

> **Warning:** Because handlers run in parallel, protect any shared variable (a counter, a cache, a dict) with a `Mutex`. Modifying a dict concurrently without a lock is a fatal error.

11.8 proc — Processes

`proc` runs external commands. It has two functions: `run(cmd, args?, options?)` runs the command directly (without shell interpretation); `shell(cmdline, options?)`, on the other hand, interprets a shell line (pipes, variables, wildcards work). Both return `{code, stdout, stderr}`. As `options` you can give `input` (standard input), `dir` (working directory), and `env`.

```
# Let's run external commands with proc.
import "proc"

# run(cmd, args) runs the command directly without shell interpretation.
let r = proc.run("echo", ["hello", "world"])
print("exit code:", r["code"])           # exit code: 0
print("stdout:", r["stdout"].strip())    # stdout: hello world

# shell(...) interprets a shell line; pipes and variables work.
let s = proc.shell("printf 'a\\nb\\nc\\n' | wc -l")
print("line count:", s["stdout"].strip())  # line count: 3

# Let's feed standard input via options.
let up = proc.run("tr", ["a-z", "A-Z"], {"input": "cobra"})
print("uppercase:", up["stdout"].strip())   # uppercase: COBRA

# A failed command's exit code is the result; it does not throw an error.
let bad = proc.shell("exit 3")
print("failed code:", bad["code"])       # failed code: 3
```

The distinction between `run` and `shell` matters for security: if you are going to pass externally provided input as an argument to a command, prefer `run` with an argument list to avoid shell injection (`proc.run("grep", [pattern, file])`). A command failing to start (for example a nonexistent program) is a catchable error; but a nonzero exit code is not an error, it is a normal result — read from the `code` field as in the `exit 3` example above.

> **Tip:** Always check a command's success from the `code` field; some tools write diagnostic messages to `stderr` but still exit with 0, and some do the opposite.

11.9 net — TCP Sockets

`net` provides raw TCP sockets (via opaque handles): `connect(addr)`, `listen(addr)`, `accept(listener)`, `send(conn, data)`, `recv(conn, max)` (`null` at EOF), `recv_line(conn)`, `close(handle)`. This is useful when you need the layer beneath HTTP or when writing your own protocol. Let's set up a tiny echo server and client in a single script.

```
# A tiny TCP echo server and client with net.
import "net"
import "time"

let addr = "127.0.0.1:8099"

# Server: accepts a connection, reads a line, sends it back, closes.
async def server()
    let listener = net.listen(addr)
    let conn = net.accept(listener)
    let line = net.recv_line(conn)
    net.send(conn, "echo: " + line)
    net.close(conn)
    net.close(listener)
end

let srv = server()
time.sleep(0.1)   # wait for the server to start listening

# Client: connects, sends a line, reads the reply.
let c = net.connect(addr)
net.send(c, "hello\n")
let reply = net.recv(c, 1024)
print(reply.strip())   # echo: hello
net.close(c)

await srv
```

We started the server as a concurrent task (`async def`); `accept` blocks until a connection arrives, which is why we run it in a separate task and run the client in the main flow. `recv(conn, max)` reads at most `max` bytes and returns `null` when the stream ends; for line-based protocols `recv_line` is more practical. Handles (connections and listeners) must be closed with `close` when the job is done.

> **Note:** The `net` layer works with bytes and strings; an application-level protocol (framing, length prefixes, etc.) is your responsibility. In most cases it is easier to use the `http` module directly; prefer `net` only when you need a custom protocol.

11.10 runtime — Runtime and Memory

Cobra does not free memory by hand; a garbage collector (GC) automatically reclaims dead values. The `runtime` module opens a window onto this mechanism for profiling and benchmarking purposes: `gc()` (force collection), `free_os_memory()` (collect, then return memory to the operating system), `mem()` (a statistics dict), `set_gc_percent(pct)` (sets the garbage collection percentage, returns the previous value), `num_goroutines()` (the number of running tasks), `num_cpu()`.

```
# Let's measure memory usage with runtime.
import "runtime"

print("cpu cores:", runtime.num_cpu() >= 1)   # cpu cores: true

# Let's take a clean baseline before measuring.
runtime.gc()
let before = runtime.mem()["heap_alloc"]

# Let's do some memory-hungry work: a list with 100 thousand elements.
let big = []
for i in range(100000)
    big.push(i * i)
end

let after = runtime.mem()["heap_alloc"]
print("list created:", len(big) == 100000)  # list created: true
print("memory increased:", after > before)   # memory increased: true

# Let's look at the GC statistics.
let m = runtime.mem()
print("gc count reasonable:", m["num_gc"] >= 0)  # gc count reasonable: true

# Set the garbage collection percentage: it returns the previous value.
let prev = runtime.set_gc_percent(200)
print("previous percent is an integer:", type(prev) == "INTEGER") # previous percent is an integer: true
runtime.set_gc_percent(prev)
```

`mem()` returns a rich dict: `alloc`, `total_alloc`, `sys`, `heap_alloc`, `heap_sys`, `heap_objects`, `num_gc`, `pause_total_ns`, `gc_cpu_fraction`. When taking a measurement, the typical flow is to call `gc()` at the start to get a clean baseline, do the work, and look at the `heap_alloc` difference. `num_goroutines()` gives the number of running tasks/handlers (since Cobra has no GIL, each task genuinely runs in parallel on its own thread); `num_cpu()` gives the number of cores and is useful for sizing parallel work.

> **Tip:** Memory measurements are inherently noisy; the GC can run whenever it wants in the background. When comparing, look at the trend rather than a single reading, and stabilize the state with `runtime.gc()` immediately before measuring.

11.11 test — Unit Test Framework

Cobra ships with a small xUnit-style test framework inside it. To mark a function as a test you use the `@test.it` decorator (or register one with a custom name using `case(name, fn)`). Assertions throw an error on failure; `run()` runs them all, prints a report, and returns a `{passed, failed, total, ok}` dict.

Assertions: `assert(cond, msg?)`, `assert_eq(actual, expected, msg?)` (value equality like ==), `assert_neq(a, b, msg?)`, and `assert_raises(fn, msg?)` (passes if `fn` throws an error).

Let's write a real test suite that tests two small functions (`slugify` and `divide`).

```
# Let's write a small test suite with the test module.
import "test"

# The code to be tested (in reality it would be imported from a separate module).
def slugify(s)
    return s.strip().lower().replace(" ", "-")
end

def divide(a, b)
    if b == 0
        throw "division by zero"
    end
    return a / b
end

# @test.it marks a function as a test.
@test.it
def slug_turns_space_to_hyphen()
    test.assert_eq(slugify("  Hello World "), "hello-world")
end

@test.it
def slug_already_clean()
    test.assert_eq(slugify("code"), "code")
    test.assert_neq(slugify("A B"), "A B")
end

@test.it
def divide_correct()
    test.assert_eq(divide(10, 2), 5)
    test.assert(divide(9, 3) == 3, "9/3 should == 3")
end

@test.it
def divide_by_zero_throws()
    # assert_raises passes if the function throws an error when called.
    def call_zero()
        return divide(1, 0)
    end
    test.assert_raises(call_zero)
end

# We can also register a test with a custom name using case().
def negative_check()
    test.assert(divide(-6, 2) == -3)
end
test.case("negative division", negative_check)

let summary = test.run()
print("summary:", summary["passed"], "/", summary["total"], "ok:", summary["ok"])
```

The program's output is as follows:

```
ok    slug_turns_space_to_hyphen
ok    slug_already_clean
ok    divide_correct
ok    divide_by_zero_throws
ok    negative division

5 passed, 0 failed, 5 total
summary: 5 / 5 ok: true
```

`assert_eq` uses **value equality** (just like `==`), so it compares nested lists and dicts deeply. To `assert_raises` we pass a **function**; the framework calls it on your behalf and expects it to throw an error — that is why we give `divide(1, 0)` not directly but through a small wrapper function called `call_zero` (recall that Cobra has no `lambda`). If an assertion fails, the framework catches it, marks the relevant test as failed, but continues running the other tests; the `ok` field in the dict returned at the end summarizes whether the whole suite passed.

> **Tip:** Use the `ok` field returned by `run()` in CI/continuous-integration scripts: if it is `false` you can make the exit code nonzero to mark the build as failed.

11.12 Summary

In this chapter we toured all of Cobra's standard library through real tasks:

- **fs** — file and directory operations; we processed a log file line by line.
- **math** — constants and numerical functions; we highlighted banker's rounding and the integer-division pitfall.
- **os** — arguments, environment variables, platform, and time; `args`/`platform` are functions.
- **json** — parsing and serializing while preserving key order; we decoded an API response.
- **time** — the monotonic `clock` for duration measurement and UTC calendar functions (strftime directives).
- **re** — RE2-based matching; we did email validation and field extraction (no backreferences/lookaround).
- **http** — a client returning `{status, body, headers}` and a server with parallel handlers; we set up a small JSON API.
- **proc** — subprocesses with `run`/`shell`; `{code, stdout, stderr}`.
- **net** — raw TCP; we wrote a tiny echo server.
- **runtime** — GC and memory accounting; we measured the memory increase while building a list.
- **test** — the built-in unit test framework; we wrote a real test suite with `@test.it`, `case`, and assertions.

Common patterns to remember: built-in modules are imported by their names and their members are accessed with a dot; module errors are catchable, so wrap code that talks to the outside world (files, network, JSON) with `try`/`catch`; and when parallelism is involved (HTTP handlers, tasks) protect shared state with a `Mutex`. These modules, combined with Cobra's plain syntax, are enough to write short and readable programs that nonetheless do real work.

Chapter 12

The Internals of the Language — From Lexer to Virtual Machine

In the previous chapters we stood on the side that *uses* Cobra: variables, functions, structs, modules. In this final chapter we cross to the other side of the table. We will take apart and inspect the machine that turns Cobra source code into runnable behavior: how does the source text become first words, then a tree, and finally a running program?

Cobra has an interesting architectural decision: there is not one but **two** execution engines, and both run the same language:

- **Tree-walking engine** — the default. It evaluates by walking the abstract syntax tree (AST) directly. This is the engine that is easiest to write and to read.
- **Compiler + virtual machine (VM)** — activated with the `--vm` flag. It first compiles the source into compact **bytecode**, then runs that on a stack-based virtual machine. It is roughly several times faster.

Throughout this chapter we will take a small Cobra program and trace it step by step through both engines' pipelines. Everything described comes from real tool output; you can produce the examples yourself.

The program we will trace is two lines:

```
let n = 2
print(n + 1)
```

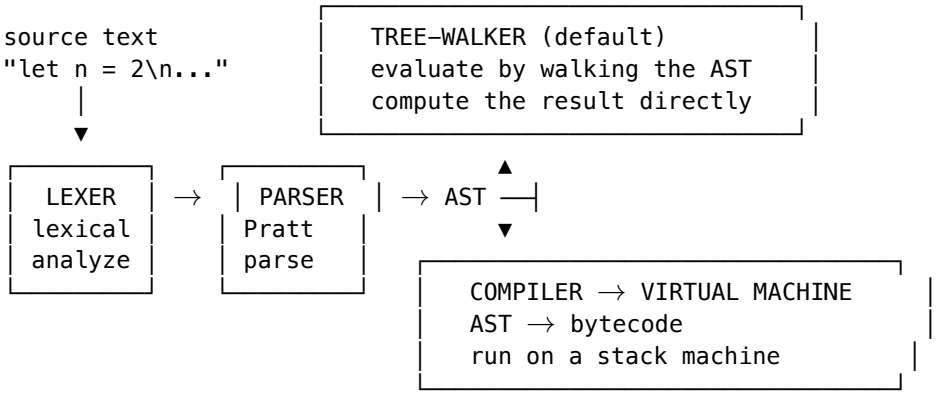
Run both engines; they give the same result:

```
$ cobra ornek.cobra          # tree-walking
3
$ cobra --vm ornek.cobra     # bytecode VM
3
```

> **Note:** "The same behavior" is not a coincidence, it is an architectural necessity. The *semantics* — arithmetic, truthiness, indexing, slicing, method calls, and built-in functions — are defined in a single place. Both engines consult this single definition. Thus they stay naturally aligned — whatever answer one gives, the other gives too.

12.1 The Overall Architecture — From Source to Result

Both engines share the same first two stages. The source text is first converted into a stream of **tokens**, then into an **abstract syntax tree (AST)**. Only after that do the paths diverge:



A command-line flag decides which path is taken. The same source is sent down a different pipe depending on your choice:

- `cobra dosya.cobra` — lexer + parser + **tree-walker**.
- `cobra --vm dosya.cobra` — lexer + parser + **compiler + VM**.
- `cobra --tokens dosya.cobra` — prints only the output of the **lexer**.
- `cobra --ast dosya.cobra` — prints the **AST** produced by the lexer + parser.

The last two flags give us the two diagnostic tools we will use in this chapter: `--tokens` shows what the lexer saw, and `--ast` shows the tree the parser built. We will use both throughout.

12.2 The Lexer — Splitting Text into Tokens

We can sum up the **lexer**'s (lexical analyzer's) job in a single sentence: breaking the raw character stream into meaningful pieces — **tokens**. A token carries three pieces of information:

- **Type:** what the token is — like `IDENT` (identifier), `INT` (integer), `PLUS` (plus), `NEWLINE` (end of line).
- **Text:** the token's actual characters in the source — `"n"`, `"2"`, `"+"`.
- **Line:** the source line on which the token occurs, for use in error messages.

The lexer moves through the input text from left to right like a single read head. It looks at the current character, peeks ahead one or two characters if necessary, produces a token, and advances its head. It repeats this process until the text is exhausted. Each token it produces flows to the next stage — the parser.

The lexer has two interesting internal states, and the secret of Cobra's semicolon-free, indentation-friendly syntax is hidden in them: **has any real token been produced yet?** and **how many open parentheses are we currently inside?** We will return to both shortly.

12.2.1 Skipping whitespace and comments

Before producing a token, the lexer silently swallows meaningless characters: spaces, tabs, and **comments**. In Cobra a comment begins with # and runs to the end of the line; as soon as the lexer sees #, it skips the rest of the line. Comments never turn into tokens — the parser does not even know they exist.

12.2.2 Two-character operators: peeking ahead

= and ==, or + and +=, begin with the same character. The lexer figures out which is meant by **peeking ahead one character**. It first sees the current character (+), then, without advancing the cursor, looks at the next one: if it is =, this is a += token and both characters are consumed at once; otherwise it is a plain +.

The same lookahead pattern works for ==, !=, <=, >=, -=, *=, /=, %= as well. A wrong guess costs nothing, because peeking ahead consumes nothing — it merely looks at the next character.

12.2.3 NEWLINE: end of line instead of semicolon

In Cobra, statements end not with a semicolon but with an **end of line**. The lexer handles this by producing a special `NEWLINE` token. But there are two subtle refinements.

First — line breaks inside parentheses are insignificant. The lexer tracks how many open parentheses (`(`, `[`, `{`) it is inside with a counter. The counter increases on each opening and decreases on each closing. If a line break is seen while the counter is greater than zero, it is ignored:

```
opening  (  [  {   →  counter++   →  inner '\n' ignored (like whitespace)
closing  )  ]  }   →  counter--
while counter == 0 →  '\n' produces a NEWLINE token (statement boundary)
```

This lets list and dictionary literals and function calls span multiple lines; the parser still sees a single logical statement:

```
let renkler = [
    "kirmizi",
    "yesil",
    "mavi"
]
```

Second — leading line breaks are swallowed. Blank lines that come *before* the first real token have no meaning for the parser (there is not yet any statement to terminate). By keeping a "has the first real token been produced?" state, the lexer silently discards these leading line breaks. Also, consecutive blank lines are reduced to a single `NEWLINE`.

12.2.4 Numbers and the `m` decimal suffix

When the lexer sees a digit, it reads the consecutive digits and builds a number token. If there is a dot *with a digit on both sides*, the number becomes a `FLOAT`: `1.5` yes, but `1.` or `.5` no. If there is an `m` letter at the end of the number and it is not followed by a letter or digit, this is an **exact decimal** (DECIMAL) literal — like `19.90m`, used for monetary calculations.

> **Tip:** The condition for the `m` suffix matters: it must not be followed by a letter or digit. `42m` is a DECIMAL, but `42memo` is not — there `42` is read as a number and `memo` as a separate identifier. This prevents variable names that immediately follow a number from being accidentally swallowed.

12.2.5 Keyword or identifier?

When the lexer reads a sequence of letters — for example `let` or `print` — it must decide whether it is a **keyword** or an ordinary **identifier**. The solution is simple: the text read is first looked up in a table of known keywords. If it matches, the type of that keyword (`LET`, `IF`, `DEF`…) is produced; if it does not match, `IDENT` is produced. So `let` is a `LET`, while `print` is an `IDENT`.

12.2.6 The tokens of the example program

Now let us see the real output. The `--tokens` flag prints the stream the lexer produces verbatim:

```
$ cobra --tokens ornek.cobra
line 1   LET      "let"
line 1   IDENT    "n"
line 1   =        "="
line 1   INT      "2"
line 1   NEWLINE  "\\n"
line 2   IDENT    "print"
line 2   (        "("
line 2   IDENT    "n"
line 2   +        "+"
line 2   INT      "1"
line 2   )        ")"
line 2   NEWLINE  "\\n"
```

Note: `let` resolved to a keyword (`LET`), but `n` and `print` remained as `IDENT`s. The `NEWLINE` between the two lines marks the statement boundary. Because the parenthesis counter inside `print(...)` incremented and then decremented by one, any potential line break there would have been ignored.

12.3 The Parser — From Tokens to a Tree

The **parser** converts the token stream into an **abstract syntax tree** (AST). The AST represents the program's structure in tree form: which expression is inside which expression, which operator is applied to which operands.

Cobra uses **Pratt parsing** — an operator-precedence-aware, top-down technique. The reason it is famous is that it handles expression precedence (∗ comes before +) very elegantly.

12.3.1 The precedence ladder

Every operator has a **precedence level**. Sorted from lowest to highest, the following ladder emerges:

```
lowest   ┌─ assignment      x = ...
         │  ternary          kosul ? a : b
         │  logical or       a or b
         │  logical and      a and b
         │  logical not      not a
         │  equality         ==  !=
```

```
          │  comparison       <  >  <=  >=
          │  addition         +  -
          │  multiplication   *  /  %
          │  unary (prefix)   -x  not x
          │  call             f(...)
highest   └─ indexing         liste[i]
```

Thanks to this ordering, `2 + 3 * 4` parses correctly as `2 + (3 * 4)`: because multiplication has higher precedence than addition, the parser gathers `3 * 4` into a single subtree, then makes it the right-hand side of the addition.

12.3.2 Prefix and infix rules

The essence of Pratt parsing is two sets of rules. For each token type, the parser knows:

- **prefix rule:** what to do when the token comes at the *start* of an expression — the – in –x, or an integer literal standing alone, or an opening parenthesis (grouping).
- **infix rule:** what to do when the token comes *between* two expressions — the + in `a + b`, or the == in `a == b`.

An interesting point: `(` and `[` also have infix rules. In the expression `f(x)`, the `(` comes *after* the `f` expression on its left; that is, it behaves like an infix operator and carries very high precedence (call/indexing). Similarly, the `[` in `xs[i]` "swallows" the `xs` on its left. Thus `f(x)` becomes a **call node** and `xs[i]` becomes an **index node**.

12.3.3 The Pratt core

All the magic gathers into a single loop. While parsing an expression, the parser first runs the current token's prefix rule to obtain a "left side." Then, *while the token on the right has high enough precedence*, the infix rules keep "swallowing" the left side over and over:

```
parse_expression(precedence):
    left = curToken.prefix_rule()         # the leading expression first
    loop: while peekToken's precedence > precedence
        left = peekToken.infix_rule(left)  # the left side grows
    return left
```

For `n + 1`: first `n` (an identifier) is taken by the prefix rule. Then, since the next token is + and its precedence is high enough, the infix rule of + runs; it parses the right side (`1`) and builds an **infix node** — a node whose left operand is `n`, whose operator is +, and whose right operand is `1`.

12.3.4 AST nodes

Each node in the AST represents a language construct. Our example program produces the following nodes:

- A **let statement** node: name `n`, value an integer node.
- That **integer node**: value `2`.
- An **expression statement** node: containing a call.
- That **call node**: callee `print`, argument an infix node.

- That **infix node**: left `n`, operator `+`, right `1` (an integer node).

Each node type holds only the information needed for its own meaning: an integer node holds only its numeric value, while an infix node holds the left-operator-right triple. These nodes combine to form the program's complete structure.

12.3.5 The AST of the example program

The `--ast` flag shows the AST in each node's re-textualized form:

```
$ cobra --ast ornek.cobra
let n = 2
print((n + 1))
```

Note how `n + 1` is printed parenthesized as `(n + 1)`. This is not by chance: each infix node re-textualizes itself by explicitly parenthesizing itself, so the parse structure becomes visible to the eye. So the parser has indeed built an addition node as the argument of the `print` call.

> **Note:** The parser enforces syntactic rules *before* runtime. For example, a `break` outside a loop, a `return` inside a `finally` block, or precedence subtleties like `not x == 3` (the parser makes it `not (x == 3)`, because `not` binds more loosely than comparisons) are all resolved at this stage.

12.4 The Tree-Walking Engine — Running the AST Directly

This is the simplest engine and Cobra's default. The tree-walker walks the AST starting from the root and, for each node, answers the question "what does this mean?" *in place*. If it sees an integer node it produces an integer value; if it sees an infix node it first evaluates both sides, then applies the operator.

The computed values are **Cobra values**: integer, float, string, list, dictionary, function, struct instance, and the like. Variable bindings are kept in an **environment**. Nested scopes form a chain: when looking up a variable, the innermost environment is checked first, and if it is not found, the enclosing environment is checked — all the way up to the outermost (global) environment.

Our example program flows through this engine as follows:

1. The program node walks the statements in order.
2. `let n = 2`: the right-hand side (the `2` integer node) is evaluated first and produces an integer value; this value is bound to the name `n` in the environment.
3. `print(n + 1)`: the call node is evaluated. First `n + 1` (an infix node) is computed — `n` is read from the environment (`2`), `+` is applied, the result becomes `3`. Then the `print` built-in function is called with the argument `3` and `3` is printed to the screen.

12.4.1 Propagating control flow upward

While evaluating a block, the engine processes the statements in order — but if a `return`, `break`, `continue`, or an error arises, it must leave the block immediately. It does this by wrapping these signals in special values. After each statement, a block checks "is this a control-flow signal?"; if so, it skips the remaining statements and passes the signal back up a level:

```
eval_block(block, env):
    result = null
    for each statement in block:
        result = evaluate(statement, env)
        if result is a control-flow signal (return/break/continue/error):
            return result immediately   # skip the remaining statements
    return result
```

A `return` value is carried upward as a wrapped "return value" and is unwrapped at the function boundary, becoming the actual result. `break` and `continue` similarly rise up to the nearest loop and are consumed there. This is the classic and most readable way to write a recursive evaluator — but, because it does type discrimination at every node and produces a new object for every value, it is not the fastest. This is exactly why the VM exists.

12.5 The Compiler + Virtual Machine — Going Down to Bytecode

The second engine, instead of executing the AST directly, first compiles it into **bytecode**: a compact instruction stream for a stack-based virtual machine. It then runs this bytecode in a tight, short loop. Instead of walking the tree over and over, it quickly processes a flat, pre-prepared list of instructions.

12.5.1 Opcodes

The building block of bytecode is the **opcode** (operation code): a single primitive command that the virtual machine can understand. Each opcode is one byte and may take optional **operands**. Some of the opcodes Cobra uses:

- `OpConstant <i>` — pushes value number `i` from the constant pool onto the stack.
- `OpSetGlobal <slot>` / `OpGetGlobal <slot>` — writes/reads a global variable.
- `OpSetLocal <slot>` / `OpGetLocal <slot>` — writes/reads a local variable.
- `OpGetBuiltin <i>` — pushes a built-in function like `print`, `len` onto the stack.
- `OpBinary <op>` — pops two values off the stack, applies a binary operator, pushes the result back. The operand says which operator it is (+, −, ∗, ...).
- `OpCall <n>` — makes a call with `n` arguments.
- `OpPop` — discards the value at the top of the stack (when a statement's result is unused).
- `OpHalt` — the end of the main program.

The operand widths of each opcode are fixed. For example, `OpConstant` takes a 2-byte operand (constant pool index), `OpGetBuiltin` a 1-byte one, while `OpPop` takes no operand at all. The VM advances the instruction pointer by exactly this width after each instruction.

Instead of defining separate opcodes for each of the binary operators, a single `OpBinary` is used; its operand specifies which operator to apply. The same ordering (`+ - * / % == != < > <= >=`) is shared by both the compiler and the VM, so the operator numbers mean the same thing on both sides.

12.5.2 What does the compiler do?

The **compiler** also walks the AST — just like the tree-walker — but instead of *computing* values, it *emits* opcodes. When it sees an integer node, it adds it to the constant pool and emits an `OpConstant`. When it sees an infix node, it first compiles the left side, then the right side (both will leave a value on the stack), and finally adds an `OpBinary`.

An important detail: variables are handled not by name but by **slot number**. During compilation, a **symbol table** assigns each name an integer index (according to global, local, free, or built-in scopes). At runtime the VM no longer looks up names, it indexes an array directly — this is a big speed gain.

12.5.3 The bytecode of the example program

Here is our example compiled. The following is a human-readable **dump** (disassembly) of the bytecode; the numbers on the left are each instruction's byte address:

```
=== INSTRUCTIONS ===
0000 OpConstant 0        # push constant[0] = 2 onto the stack
0003 OpSetGlobal 0       # assign to global slot 0 (n)
0006 OpGetBuiltin 0      # push the built-in 'print'
0008 OpGetGlobal 0       # push n
0011 OpConstant 1        # push constant[1] = 1
0014 OpBinary 0          # apply 0 = "+"  →  n + 1
0016 OpCall 1            # call with 1 argument → print(3)
0018 OpPop               # discard the result from the stack (statement)
0019 OpHalt              # end of program

=== CONSTANTS ===
0: INTEGER 2
1: INTEGER 1
```

When you put the token stream, the AST, and this bytecode side by side, you see the entire pipeline: the `2` and `1` literals were moved into the **constant pool**, `n` was reduced to a **global slot**, `+` became an `OpBinary 0`, and the `print` call turned into the `OpGetBuiltin 0` + `OpCall 1` pair.

Note the address column: `0000`, `0003`, `0006`... The gaps between them are the bytes the operands occupy. `OpConstant` occupies 1 (opcode) + 2 (operand) = 3 bytes, which is why the following instruction is at `0003`; `OpGetBuiltin` is 1 + 1 = 2 bytes, which is why the instruction after it is at `0008`. The VM advances the instruction pointer by exactly this much.

12.5.4 OpHalt and the bounds-check-free loop

The compiler always appends an `OpHalt` to the end of the instruction stream it produces. The purpose is performance. If the VM's inner loop had to check after every instruction whether "have we reached the end of the stream?", that check would waste time on every step. Instead, because it knows it will *guaranteed* hit an `OpHalt` at the end of the stream, the loop does no bounds checking at all; when it reaches `OpHalt` it simply stops and returns. Removing even a single comparison from the hot loop provides a noticeable gain in a VM that processes millions of instructions.

12.5.5 How does the virtual machine work?

The VM is a **stack machine**. It does its computations on a value stack: it pushes values, pops them, and pushes results back. It keeps a few preallocated fixed-size arrays: the value **stack**, the array of **global** variables, and the call **frames**. There is also a **stack pointer**; it says where the top of the stack is.

The core is a single loop that turns over opcodes: read the next opcode, do whatever it needs to do, advance the instruction pointer, repeat. Let us trace the execution of our example program step by step — the column on the right shows the stack *after* each instruction:

address	instruction	stack (after)
0000	OpConstant 0	`[2]`
0003	OpSetGlobal 0	`[]` (n = 2)
0006	OpGetBuiltin 0	`[print]`
0008	OpGetGlobal 0	`[print, 2]`
0011	OpConstant 1	`[print, 2, 1]`
0014	OpBinary 0	`[print, 3]`
0016	OpCall 1	`[null]` (print printed 3)
0018	OpPop	`[]`
0019	OpHalt	—

Note how `OpBinary 0` pops two values (`2` and `1`) and puts a single result (`3`) in their place — the typical move of a stack machine. `OpCall 1`, in turn, takes `print` and its argument off the stack and runs the call; what remains is `print`'s return value (`null`), which `OpPop` then discards.

12.5.6 Frames and closures

Each function call opens a **frame**. A frame holds: the function currently running, the instruction pointer specific to that function, and where the local variables begin on the stack (the base pointer). When a function is called, a new frame is "pushed"; when the function returns, the frame is "popped," the stack is wound back to its state before the call, and the return value is left behind. Frames too are kept in a fixed-size array; the depth of this array determines the recursion limit.

Closures are functions that refer to variables in the enclosing scope. If a function accesses a variable outside itself (a "free variable"), the compiler notices this and binds that variable into the function's closure. At runtime, separate opcodes access these free variables. As a result, a function keeps remembering its variable even after the context that produced it has long closed. The classic example is a counter generator:

```
def sayac_yap()
    let n = 0
    def artir()
        n = n + 1     # 'n' is a free variable — carried from sayac_yap into the closure
        return n
    end
    return artir
end

let c = sayac_yap()
print(c())
print(c())
print(c())
```

Both engines give the same result:

```
$ cobra closure.cobra        # tree-walking
1
2
3
$ cobra --vm closure.cobra   # VM
1
2
3
```

12.5.7 Superinstructions: an example of a speed optimization

A real engineering touch in the VM is **superinstructions**. One of the most common patterns in function bodies is of the form `local_variable OP integer_literal` — like `n < 2`, `i + 1`. Normally this triple requires three separate instructions: push the local variable, push the constant, apply the binary operator. The compiler notices this frequent pattern and reduces it to **a single fused instruction**. Instead of a three-instruction dispatch, a single instruction runs; moreover, this instruction has a special fast path for integers and computes the result without ever touching the slow, general operator logic. The programmer does nothing — the gain comes entirely from the agreement between the compiler and the VM.

12.6 Native Compilation — Leaving the Interpreter Behind

The two engines we have seen so far share one property: at run time, *something reads the program and decides what to do*. The tree-walker reads AST nodes, the VM reads opcodes. That reading — the dispatch — is the price of being an interpreter, and no amount of tuning removes it entirely.

There is a third path, and Cobra takes it: **compile the program to machine code and stop interpreting altogether.**

```
cobra build --native mandelbrot.cobra
```

The result is a standalone executable. It contains no VM, no bytecode and no Cobra runtime — just your program, as instructions the processor executes directly.

12.6.1 The obstacle: dynamic types

The reason a scripting language interprets in the first place is that it does not know its types ahead of time. When the VM executes `x + y`, it must ask at run time: are these integers? floats? strings? Each value therefore lives *boxed* — a pointer to a heap object carrying its type — and every operation pays for the check and the box.

Machine code cannot work that way. `ADD` on a processor adds two integers in registers; it does not ask questions. To emit it, the compiler must *know*, at compile time, that `x` and `y` are integers.

12.6.2 The way through: type inference

Here Cobra makes an observation about real programs: although the language is dynamically typed, **actual code is overwhelmingly type-stable**. A variable that starts as a float stays a float; a function that returns an integer keeps returning one. The freedom to change type exists, and is almost never used.

So the native compiler *proves* the types instead of demanding them. It gives every variable, parameter, field and return value a type "cell" that begins empty, then walks the whole program repeatedly, merging into each cell whatever it learns:

- assigning an integer to an empty cell makes it an integer;
- a cell already holding an integer that meets a float becomes a float — exactly the promotion the language performs at run time (`1 + 2.5` is `3.5`);
- a cell holding `null` that meets a struct becomes that struct (this is how `Node(null, null)` in a binary tree infers correctly).

The walk repeats until nothing changes — a *fixpoint*. When it settles, every name in the program has one concrete machine type, and the compiler can emit real instructions. If some name genuinely cannot be pinned down, the compiler does not guess: it reports the program as *not natively compilable* and tells you which construct blocked it.

12.6.3 What comes out

With types known, the translation is direct:

Cobra	native code
`let i = 0`	a 64-bit integer in a register
`let x = 1.5`	a 64-bit float in a register — no box, no allocation
`struct Body` with 7 fields	a flat struct: 7 floats laid out contiguously
`let xs = []`	a native, growable array
`p.x`	a fixed memory offset — no name lookup
`fib(n - 1)`	a machine call instruction

The struct row is where the difference becomes vivid. In the VM, `b.vx * b.vx` means: find the field by name, unbox a float, unbox another, multiply, allocate a box for the result. In native code it is a single multiply instruction between two registers.

12.6.4 Identical, not merely close

A faster engine that answered differently would be worthless, so the same rule that governs the two interpreters governs this one: **the output must be byte-identical.** Two subtleties had to be handled to keep that promise.

Float *printing* must match — Cobra prints `3.0` as `3.0`, not `3`, and switches to scientific notation at precise thresholds — so the native binary carries the same formatting logic the interpreter uses.

Float *arithmetic* is subtler. Modern processors offer a fused multiply-add instruction, which computes `a*b + c` in one step with a single rounding instead of two. It is faster and, arguably, more accurate — and it produces a *different number* in the last decimal place than the interpreter, which rounds twice. Cobra therefore emits an explicit rounding barrier after each float operation, forbidding the fusion. It gives up a little speed to keep the results bit-for-bit equal to the VM's — the right trade for a language runtime.

12.6.5 What it costs, and what it buys

Native compilation supports the type-stable core of the language: functions, structs with `init` and methods, lists, dicts, strings, `while` / `for-in` / `if`, and the `math` module. Closures, `async`, `try/catch` and inheritance

are not compiled — each breaks the "one concrete type per name" model in its own way (a `try` block, for instance, requires *every* fallible operation in the program to carry a checked path and a line number, which is exactly the overhead native code exists to avoid). Programs using them run on the VM, unchanged and unpunished.

What it buys is a different performance class. On the classic numeric benchmarks — n-body, binary-trees, spectral-norm, mandelbrot — the native binary finishes the suite in 0.28 seconds. CPython takes 5.99. PyPy, a tracing JIT that compiles Python to machine code at run time, takes 0.44. The native binary needs no warm-up, no runtime, and nothing installed on the machine that runs it.

The lesson worth taking from this chapter is not the benchmark table. It is that the boxes and the dispatch loop — everything the interpreter chapters spent their effort optimizing — were never essential to the *language*. They were the cost of not knowing the types. Learn the types, and the cost disappears.

12.7 Shared Semantics — Two Engines, One Truth

So far the two engines look separate: one walks the AST, the other runs bytecode. But the *meaning* of the language — what `1 + 2` equals, how a list is sliced, what a field assignment does — is defined in **a single place**. This is the key to Cobra keeping its two engines aligned.

The semantics live as pure "operations on values," independent of both the AST and the bytecode. Both engines consult this single definition. A few examples:

Binary operators. The VM has inline fast paths for integers and strings; but all remaining cases (float, decimal, list + list, equality rules...) fall to the exact same shared operator logic that the tree-walker also uses. That `1 + 2` is an integer, that `1 + 2.5` is promoted to a float, that `"a" + "b"` is concatenation, that list + list produces a new list — all are defined in one place.

Truthiness. `if`, `while`, `and`/`or`, and `?:` decide which value counts as "true" by a single shared rule. The VM's conditional jump opcode also uses this same rule; thus an empty list counts as "false" in both engines.

Indexing and slicing. The logic of `xs[i]` and `xs[low:high:step]` lives in a single shared operation. The VM's indexing and slicing opcodes route to this shared operation. This is why even subtle behaviors like reversal with a negative step (`xs[::-1]`) are **bit-for-bit** identical in the two engines:

```
let xs = [10, 20, 30, 40]
print(xs[1:3])
print(xs[::-1])
```

```
$ cobra slice.cobra          # tree-walking
[20, 30]
[40, 30, 20, 10]
$ cobra --vm slice.cobra     # VM
[20, 30]
[40, 30, 20, 10]
```

Field access and assignment. The logic of `p.x` (read) and `p.x = v` (write) is also shared. The subtlety here is this: property *getters* and *setters* (special methods that intercept parenthesis-free access) must work in both engines. When a setter is found, the shared field-assignment logic can call it whether it is in the tree-walker's function form or the compiled closure form. The struct below keeps the Celsius value inside but can be read and written via Fahrenheit:

```
struct Sicaklik
    def init(c)
        self._c = c
    end
    get fahrenheit()
        return self._c * 9.0 / 5.0 + 32.0
    end
    set fahrenheit(f)              # t.fahrenheit = 212.0 calls this
        self._c = (f - 32.0) * 5.0 / 9.0
    end
end

let t = Sicaklik(0.0)
t.fahrenheit = 212.0
print(t._c)
print(t.fahrenheit)
```

Both the setter (`t.fahrenheit = 212.0`) and the getter (`t.fahrenheit`) go through the same path in both engines:

```
$ cobra setter.cobra         # tree-walking
100.0
212.0
$ cobra --vm setter.cobra    # VM
100.0
212.0
```

There is an elegant bridge connecting the two engines: when the VM starts up, it "teaches" the shared semantics how to run a compiled closure. Thus the shared logic can, when needed, call a compiled function without itself depending on the VM. This is a clever design that breaks the circular dependency between the two components while keeping the semantics in a single point.

> **Caution:** There is *one* known behavioral difference between the two engines: referring to an undefined variable. The tree-walker catches this at runtime; the VM, because it resolves names to slots at compile time, reports the same error *during compilation*. The outcome is the same (the program is rejected), but the moment the error is caught differs.

The practical benefit of shared semantics becomes concrete in the project's benchmark suite: every workload is run on both the tree-walker and the VM, and **both are required to produce exactly the same output**. This output equality is a test; if one diverges from the other, the build breaks.

12.8 Summary

In this chapter we traced Cobra's inner workings from beginning to end, through real tool output:

- **Architecture.** The source text first passes through two shared stages — the **lexer** (tokens) and the **parser** (AST) — then goes to one of two engines: the directly-evaluating **tree-walker** or the path that compiles and runs on the **VM**. The `--tokens`, `--ast`, `--vm` flags open these pipes.

- **Lexer.** Splits characters into tokens. `NEWLINE` is the end of a statement but is ignored inside parentheses; operators like `==`/`+=` are recognized by peeking ahead; numbers and `m`-suffixed decimals are read; `#` comments are skipped; keywords are separated from identifiers by a table lookup.
- **Parser.** Precedence-aware **Pratt parsing**: prefix/infix rules and the precedence ladder running from assignment to indexing combine to build the correctly nested AST.
- **Tree-walker.** Walks the AST node by node; computes values in place, stores them in the environment, propagates control flow upward with wrappers. Simple and readable.
- **Compiler + VM.** The compiler turns the AST into **opcodes** and a **constant pool**; reduces names to slots. The VM runs these in a tight dispatch loop — bounds-check-free with `OpHalt`, with **frames** and **closures**. Superinstructions fuse hot patterns into a single instruction.
- **Native compilation.** `--native` leaves interpretation behind entirely: whole-program **type inference** proves one concrete type per name (a fixpoint that promotes `int` to `float` exactly as the language does), and the program becomes **machine code** — unboxed arithmetic, flat structs, direct calls. The output stays byte-identical to the interpreters', down to float rounding. It covers the type-stable core of the language; everything else runs on the VM.
- **Shared semantics.** The meaning of the language — binary operators, truthiness, indexing, slicing, field access — lives in a single point. Both engines consult this shared definition; a bridge binds compiled closures to the shared logic. The result is a single language, tested and guaranteed by output equality — in two bodies.

Cobra's two-engine design beautifully illustrates a fundamental truth of language design: **separate syntax and semantics from the execution strategy.** The lexer and parser are written once and shared; the semantics are written once and shared; the only remaining choice is how you run the AST — simple and explicit (tree-walker), or fast (VM). Cobra offers both at once and lets you trust that both will give the same answer, without forcing you to make a choice.

Chapter 13

cobranum — Numerical Computing

Cobra's `[1, 2, 3]` list is general purpose: you can put any kind of value into it, grow it, shrink it. This flexibility has a cost. Each element in the list is a separate **boxed** object — even an integer sits behind a pointer in memory, lives at scattered addresses, and requires an indirection on every access. For a few elements this is invisible; but when you want to multiply two 512×512 matrices, take the mean of a million measurements, or solve a linear system with ten unknowns, boxing wastes both memory and the processor cache.

cobranum fills this gap: it adds a **dense, typed, N-dimensional array** (`ndarray`) to Cobra. A cobranum array is, behind the scenes, **a contiguous `float64` buffer** — no boxes, no pointer chasing, cache-friendly. The operations on it are vectorized and multi-core; on macOS, matrix operations go to Apple Accelerate's BLAS/LAPACK routines, while elementwise operations go to hand-written ARM NEON (SIMD) kernels and multiple threads. The result turns Cobra from a scripting language into a numerical computing tool with which you can write linear algebra, statistics, optimization, differential equations, and physics simulations.

This chapter is not a "list of functions." First we will build up how the array works — the memory model, broadcasting rules, numerical stability — then we will put the library to work on real scientific problems: least squares, principal component analysis, PageRank, solving differential equations, heat diffusion, Newton's method, logistic regression. Every code block was actually run against cobranum; the output comments beginning with `#` are the program's real output.

cobranum is not part of the core but a **Go plugin**. You use it by importing `cobranum.so`; for this you need a cobra binary that can load it (plugin-compatible).

13.1 13.1 ndarray: the memory model

A cobranum array consists of two things: a **shape** (`shape`) — the length of each dimension — and a flat, contiguous `float64` buffer. The data is stored **row-major**: in a 2-dimensional `[r, c]` array, element `[i][j]` is at position `i*c + j` in the buffer. This layout turns scanning a row from start to finish from a memory jump into a sweep; modern processors' caches love exactly this access pattern.

cobranum supports 1- and 2-dimensional arrays — vectors and matrices, the great majority of numerical work. An array is an **opaque handle**: instead of printing it directly, you view its contents with `to_list` (a nested Cobra list), `str` (a readable summary), or `shape` (the dimensions).

```
import "cobranum.so"
let np = cobranum
```

```
let M = np.array([[1, 2, 3], [4, 5, 6]])
print(np.shape(M))      # [2, 3]
print(np.str(M))        # ndarray(shape=[2, 3], data=[[1, 2, 3], [4, 5, 6]])
```

13.2 13.2 Creating arrays

You can build arrays from (nested) lists or generate them directly. The generators do not create a large Cobra list; they allocate the data directly on the Go side, which is why they are cheap even for millions of elements.

```
np.array([1, 2, 3])             # 1D vector
np.array([[1, 2], [3, 4]])      # 2D matrix

np.zeros([2, 3])                # all 0
np.ones([4])                    # [1, 1, 1, 1]
np.full([2, 2], 7.0)            # fill with a constant
np.eye(3)                       # identity matrix
np.arange(0, 10, 2)             # [0, 2, 4, 6, 8] (end excluded)
np.linspace(0.0, 1.0, 5)        # [0, 0.25, 0.5, 0.75, 1] (endpoint included)
```

`arange` is a counting sequence (start, end excluded, step); `linspace` divides the interval between two endpoints into **exactly `count` points** — this is the right tool for numerical integration and plotting grids.

13.3 13.3 Indexing, inspection, reshaping

```
let M = np.array([[1, 2, 3], [4, 5, 6]])
print(np.ndim(M), np.size(M))            # 2  6
print(np.get(M, 1, 2))                   # 6   (element [1][2])
np.set(M, 0, 0, 99)                      # in-place single-element assignment
print(np.to_list(np.reshape(M, [3, 2]))) # [[99, 2], [3, 4], [5, 6]]
```

`get`/`set` access single elements (a negative index counts from the end). `reshape` gives the same number of elements in a new shape, as a new copy.

13.4 13.4 Elementwise operations

Arithmetic works element by element on two arrays of the same shape; or with a scalar.

```
let a = np.array([1, 2, 3])
let b = np.array([10, 20, 30])
```

```
np.add(a, b)     # [11, 22, 33]
np.sub(b, a)     # [9, 18, 27]
np.mul(a, 2)     # scalar -> [2, 4, 6]
np.div(b, a)     # [10, 10, 10]
np.pow(a, 2)     # [1, 4, 9]
np.neg(a)        # [-1, -2, -3]
```

Univariate math is also elementwise:

```
np.sqrt(np.array([1, 4, 9]))     # [1, 2, 3]
np.exp(np.array([0, 1]))         # [1, 2.718...]
np.log(np.array([1, 2.71828]))   # [0, 0.999...]
np.sin(a)   np.cos(a)   np.abs(np.array([-1, 2, -3]))
```

13.5 13.5 Broadcasting

To operate on two arrays of different but compatible dimensions, cobranum uses **broadcasting**: the smaller one is "stretched" until it matches the larger. The rule is simple — a dimension must either be equal or be `1` (if it is `1`, it is repeated). A length-`n` vector behaves like a row vector `(1, n)`.

Left shape	Right shape	Result	Meaning
`(r, c)`	`(c,)`	`(r, c)`	row vector to every row
`(r, c)`	`(r, 1)`	`(r, c)`	column vector to every column
`(r, c)`	scalar	`(r, c)`	constant to every element

```
let X = np.array([[1, 2, 3], [4, 5, 6]])
let row = np.array([10, 20, 30])          # (3,) -> broadcast across rows
print(np.to_list(np.add(X, row)))         # [[11, 22, 33], [14, 25, 36]]

let col = np.array([[100], [200]])        # (2,1) -> broadcast across columns
print(np.to_list(np.add(X, col)))         # [[101, 102, 103], [204, 205, 206]]
```

Broadcasting reduces operations like "subtract the mean of each feature" or "divide each row by its norm" to a single line; we will use it constantly in the statistics and machine learning examples in this chapter.

13.6 13.6 Fusion and in-place operations

Writing a chain of computations as separate calls is simple, but each call allocates a new array. `sqrt(a*a + b*b)` means four passes and three temporary arrays. cobranum offers two escape routes:

- **Fused** kernels do the entire computation in a single pass: `hypot(a, b)`, `fma(a, b, c)` (that is, `a*b + c`), `axpy(α, x, y)` (that is, `α*x + y`).
- **In-place** variants (`*_into`) write the result into a preallocated buffer, so hot loops allocate no memory at all.

```
np.hypot(a, b)           # sqrt(a*a + b*b), single fused pass

let dst = np.zeros([3])
np.mul_into(dst, a, a)    # dst = a*a
np.add_into(dst, dst, b)  # dst = dst + b
np.sqrt_into(dst, dst)    # dst = sqrt(dst)
```

Rule: if a compute kernel is inside a loop, prefer the fused or in-place form — they eliminate intermediate arrays and memory allocation.

13.7 13.7 Reductions and statistics

A reduction collapses an array into a single number (or into a vector along an axis).

```
let s = np.array([4, 8, 15, 16, 23, 42])
print(np.sum(s), np.mean(s))          # 108  18
print(np.min(s), np.max(s), np.prod(s))
print(np.var(s), np.std(s))           # 151.66...  12.31...
print(np.norm(s))                     # 53.42...  (root of the sum of squares)
print(np.argmin(s), np.argmax(s))     # 0  5
print(np.to_list(np.cumsum(s)))       # [4, 12, 27, 43, 66, 108]
print(np.to_list(np.clip(s, 10, 20)))  # [10, 10, 15, 16, 20, 20]
```

On a 2-dimensional array you reduce rows or columns by giving an **axis** (axis 0 goes down along the columns, axis 1 along the rows):

```
let M = np.array([[1, 2, 3], [4, 5, 6]])
print(np.to_list(np.sum(M, 0)))   # [5, 7, 9]   (column sums)
print(np.to_list(np.sum(M, 1)))   # [6, 15]     (row sums)
```

More advanced statistical measures are built from these building blocks. For example, the **covariance matrix** of a data matrix is the product `Cᵀ C / n` of the centered data; we will use this for PCA in 13.11.5.

13.8 13.8 Linear algebra

Matrix multiplication, transpose, inverse, determinant, and linear solves — the backbone of numerical computing — run via Accelerate (BLAS/LAPACK) on macOS.

```
let A = np.array([[1, 2], [3, 4]])
let B = np.array([[5, 6], [7, 8]])

print(np.to_list(np.matmul(A, B)))   # [[19, 22], [43, 50]]
print(np.dot(np.array([1,2,3]), np.array([4,5,6])))  # 32  (1D dot product)
print(np.to_list(np.transpose(A)))   # [[1, 3], [2, 4]]
print(np.det(A))                     # -2.0
```

```
print(np.to_list(np.inv(A)))         # [[-2, 1], [1.5, -0.5]]
print(np.trace(A))                   # 5
print(np.to_list(np.diag(A)))        # [1, 4]
```

An important habit: to solve `A x = b`, instead of taking the inverse and multiplying (`inv(A)·b`), use `solve(A, b)`. `solve` works with LU factorization; it is both faster and numerically more stable — the difference is especially pronounced for ill-conditioned matrices (see 13.11.3).

```
print(np.to_list(np.solve(np.array([[3, 2], [1, 2]]), np.array([5, 5]))))  # [0, 2.5]
```

13.9 13.11 Scientific examples

From here on are complete, working programs. Each introduces a numerical method, then implements it in a few lines with cobranum. The outputs are real.

13.9.1 13.11.1 Least squares and goodness of fit (R^2)

We want to fit a line to data points: $y = w_0 + w_1 x$. The least squares solution is the set of weights that minimize the squared error, and it is given in closed form by $w = (X^T X)^{-1} X^T y$; here the first column of X is 1 for the constant term. How good the fit is is measured by the **coefficient of determination** $R^2 = 1 - SS_{res}/SS_{tot}$ (the closer to 1, the better).

```
let xs = [1.0, 2.0, 3.0, 4.0, 5.0]
let ys = [2.1, 3.9, 6.2, 7.8, 10.1]
let rows = []
for x in xs
  push(rows, [1.0, x])
end
let X = np.array(rows)
let y = np.reshape(np.array(ys), [5, 1])
let Xt = np.transpose(X)
let w = np.matmul(np.matmul(np.inv(np.matmul(Xt, X)), Xt), y)

let resid = np.sub(y, np.matmul(X, w))
let ss_res = np.sum(np.mul(resid, resid))
let dev = np.sub(y, np.mean(y))
let ss_tot = np.sum(np.mul(dev, dev))

print("w:", np.to_list(w))            # [[0.05], [1.99]]   (~ y = 2x)
print("R^2:", 1.0 - ss_res / ss_tot)  # 0.9973
```

13.9.2 13.11.2 Ridge (Tikhonov) regularization

When features are interdependent, $X^T X$ becomes nearly singular and the weights blow up. The **Ridge** solution adds a small λ to the diagonal: $w = (X^T X + \lambda I)^{-1} X^T y$. This pulls the weights toward zero (shrinks them) and makes the solution stable.

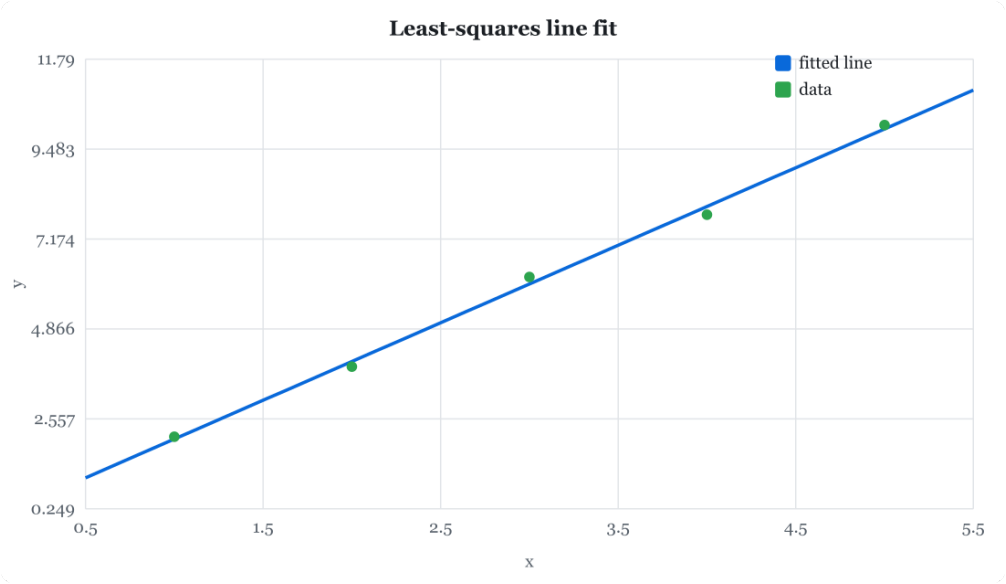


Figure 13.1: Fitting a line with least squares.

```
let lam = 1.0
let I = np.eye(2)
let wr = np.matmul(np.matmul(np.inv(np.add(np.matmul(Xt, X), np.mul(I, lam))), Xt), y)
print("ridge w:", np.to_list(wr))   # [[0.29], [1.89]]  (shrunk toward zero)
```

13.9.3 13.11.3 Conditioning and stability

The **Hilbert matrix** `H[i][j] = 1/(i+j+1)` is the textbook example of ill conditioning: as its size grows, the solution becomes excessively sensitive to tiny errors in the input. For `x = [1,1,1,1]`, let us produce `b = Hx` and solve back; how much the result deviates from exact ones is the effect of conditioning.

```
let Hrows = []
for i in range(4)
  let r = []
  for j in range(4)
    push(r, 1.0 / (i + j + 1))
  end
  push(Hrows, r)
end
let Hm = np.array(Hrows)
let b = np.matmul(Hm, np.reshape(np.array([1,1,1,1]), [4,1]))
print(np.to_list(np.solve(Hm, np.reshape(b, [4]))))
# [0.9999..., 1.0000..., 0.9999..., 1.0000...]  -- small deviations from conditioning
```

13.9.4 13.11.4 Eigenvalues — power iteration and deflation

If we repeatedly multiply a vector by a matrix and normalize at each step, the vector converges to the **dominant eigenvector**, and the Rayleigh quotient $v^{\mathsf{T}}Av$ to its **eigenvalue**. After finding the first eigenvalue, we obtain the second by subtracting $\lambda_1\ v_1 v_1^{\mathsf{T}}$ from the matrix (deflation).

```
def power_iter(A, iters)
  let n = np.shape(A)[0]
  let v = np.ones([n, 1])
  for i in range(iters)
    let Av = np.matmul(A, v)
    v = np.div(Av, np.norm(Av))
  end
  return [v, np.get(np.matmul(np.transpose(v), np.matmul(A, v)), 0, 0)]
end

let A = np.array([[4, 1, 0], [1, 3, 1], [0, 1, 2]])
let r1 = power_iter(A, 200)
let v1 = np.reshape(r1[0], [3])
let A2 = np.sub(A, np.mul(np.outer(v1, v1), r1[1]))   # deflation
let r2 = power_iter(A2, 200)
print("eigenvalue 1:", r1[1], " eigenvalue 2:", r2[1])  # 4.7320...  3.0
```

13.9.5 13.11.5 Principal component analysis (PCA)

PCA finds the direction that carries the most variance in the data. The method: center the data, build the covariance matrix, take its dominant eigenvector — this is the **first principal component**. The proportion of variance it explains is `λ₁ / trace(C)`.

```
let D = np.array([[2.5,2.4],[0.5,0.7],[2.2,2.9],[1.9,2.2],[3.1,3.0],
                  [2.3,2.7],[2.0,1.6],[1.0,1.1],[1.5,1.6],[1.1,0.9]])
let m = np.shape(D)[0]
let mu = np.mul(np.sum(D, 0), 1.0 / m)
let Dc = np.sub(D, mu)
let C = np.mul(np.matmul(np.transpose(Dc), Dc), 1.0 / m)
let pc = power_iter(C, 300)
print("PC1:", np.to_list(np.reshape(pc[0], [2])))     # [0.678, 0.735]
print("explained variance:", pc[1] / np.trace(C))     # 0.963  (96%)
```

13.9.6 13.11.6 PageRank

PageRank is the dominant eigenvector of a transition matrix — that is, another application of power iteration. On a link matrix whose columns are probabilities (column-stochastic), we iterate the initial distribution, dividing by its sum at each step.

```
# links: 0->1,2 ; 1->2 ; 2->0 ; 3->2
let M = np.array([[0, 0, 1, 0], [0.5, 0, 0, 0], [0.5, 1, 0, 1], [0, 0, 0, 0]])
let rank = np.full([4, 1], 0.25)
for i in range(100)
  rank = np.div(np.matmul(M, rank), np.sum(np.matmul(M, rank)))
end
print(np.to_list(np.reshape(rank, [4])))   # [0.4, 0.2, 0.4, 0]
```

13.9.7 13.11.7 Numerical integration — trapezoid and Simpson

We estimate the area under a curve on a grid. The **trapezoid rule** uses half-weights at the endpoints; the **Simpson rule**, with weights `1,4,2,4,…,4,1`, converges much faster. Let us test with `∫₀^π sin = 2`:

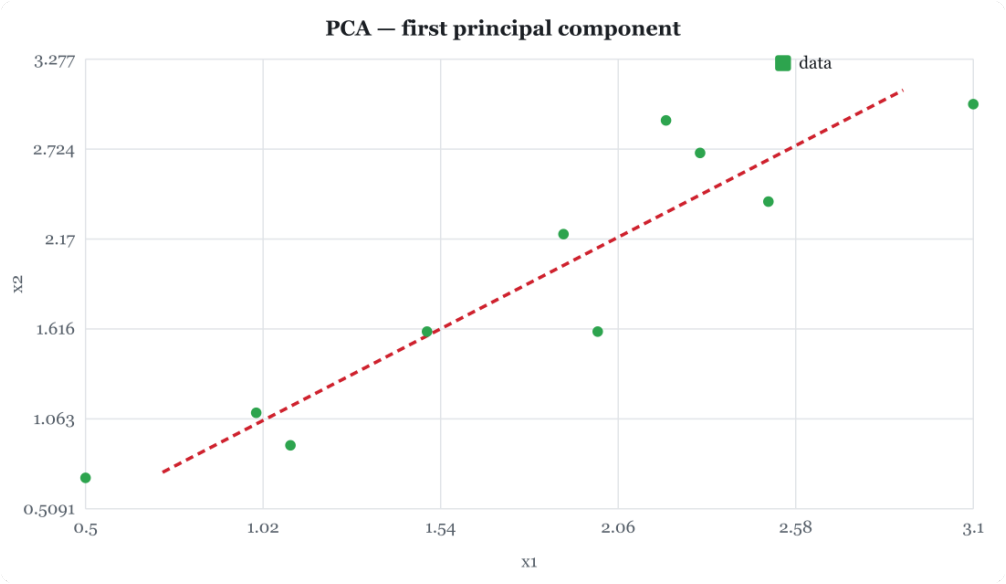


Figure 13.2: The data cloud and the first principal component axis.

```
let n = 1001                          # odd -> even number of intervals
let x = np.linspace(0.0, 3.141592653589793, n)
let y = np.sin(x)
let wlist = []
for i in range(n)
  if i == 0 or i == n - 1
    push(wlist, 1.0)
  elif i % 2 == 1
    push(wlist, 4.0)
  else
    push(wlist, 2.0)
  end
end
let h = 3.141592653589793 / (n - 1)
print((h / 3.0) * np.dot(np.array(wlist), y))   # 2.0000000000  (Simpson)
```

13.9.8 13.11.8 Differential equations — Euler versus RK4

We integrate an initial value problem `y' = f(y)` step by step. The simplest is **Euler** (`y ← y + dt·f(y)`); but its error accumulates. **Fourth-order Runge–Kutta (RK4)** averages four slope samples and is much more accurate. In a harmonic oscillator (`y' = Ky, K = [[0,1],[-1,0]]`) the energy `x²+v²` must be conserved; Euler inflates the energy, RK4 keeps it constant.

```
let A = np.array([[0, 1], [-1, 0]])
def rk4(s, dt)
  let k1 = np.matmul(A, s)
  let k2 = np.matmul(A, np.add(s, np.mul(k1, dt / 2.0)))
  let k3 = np.matmul(A, np.add(s, np.mul(k2, dt / 2.0)))
  let k4 = np.matmul(A, np.add(s, np.mul(k3, dt)))
  let toplam = np.add(np.add(k1, np.mul(k2, 2.0)), np.add(np.mul(k3, 2.0), k4))
  return np.add(s, np.mul(toplam, dt / 6.0))
```

```
end
let s = np.array([[1], [0]])
for i in range(1000)
  s = rk4(s, 0.0628318)            # ~10 periods
end
print("state:", np.to_list(s), "energy:", np.get(s,0,0)*np.get(s,0,0) + np.get(s,1,0)*np.get(s,1,0))
# state: [[~1.0], [~0.0]]   energy: 0.99999...   (RK4 conserves energy)
```

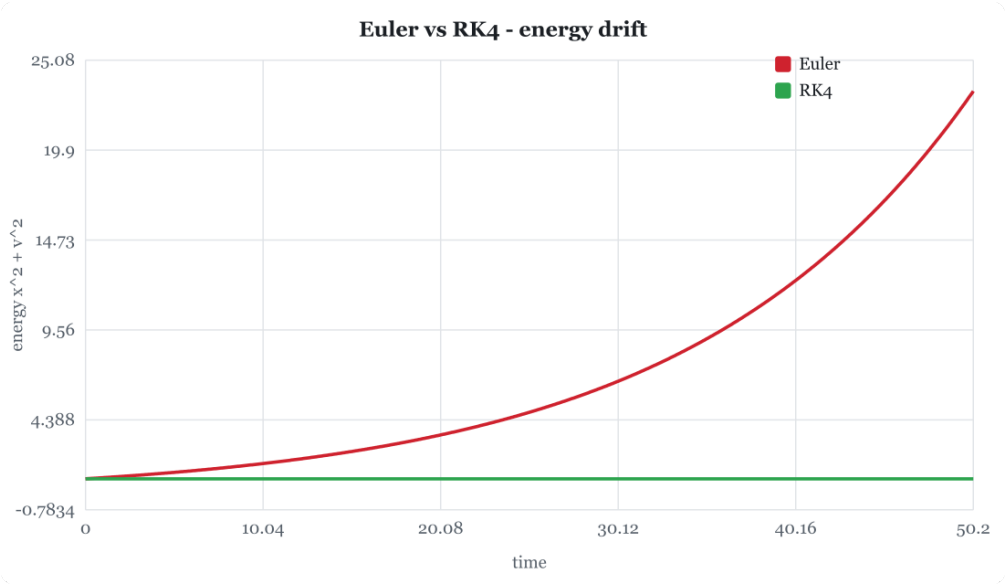


Figure 13.3: At the same step size, Euler inflates the energy while RK4 keeps it constant.

13.9.9 13.11.9 Lotka–Volterra (prey–predator)

We can solve a nonlinear system with the same RK4. The prey `x` and predator `y` populations chase each other through `x' = 1.1x − 0.4xy`, `y' = −0.4y + 0.1xy`. A 2×1 array holds the state; we read the components with `get` and build the derivative.

```
def lv(s)
  let x = np.get(s, 0, 0)
  let y = np.get(s, 1, 0)
  return np.array([[1.1*x - 0.4*x*y], [-0.4*y + 0.1*x*y]])
end
def rk4f(s, dt, f)
  let k1 = f(s)
  let k2 = f(np.add(s, np.mul(k1, dt/2.0)))
  let k3 = f(np.add(s, np.mul(k2, dt/2.0)))
  let k4 = f(np.add(s, np.mul(k3, dt)))
 return np.add(s, np.mul(np.add(np.add(k1, np.mul(k2,2.0)), np.add(np.mul(k3,2.0), k4)), dt/6.0))
end
let st = np.array([[10.0], [5.0]])
for i in range(50)
  st = rk4f(st, 0.1, lv)
end
print("at t=5:", np.to_list(st))   # [[0.68...], [1.73...]]
```

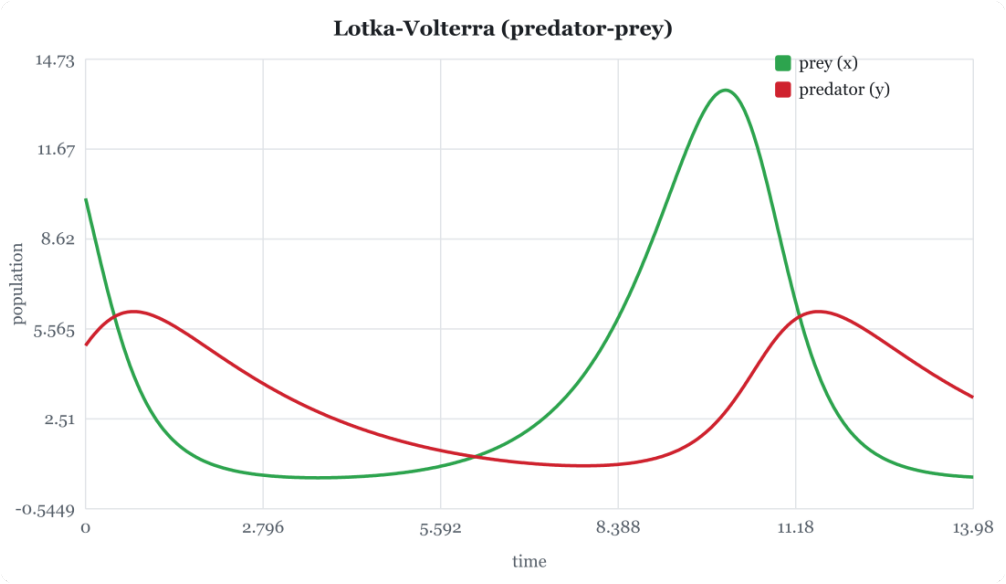


Figure 13.4: The oscillation of prey and predator populations over time.

13.9.10 13.11.10 The heat equation — with a Laplacian matrix

cobranum has no array slicing, but we can turn a partial differential equation that requires neighbor access into a **matrix multiplication**. The discrete Laplacian of the 1D heat equation is a tridiagonal matrix (`1, −2, 1`); the explicit scheme is `u ← u + α·(L u)`. A temperature bump in the middle spreads over time.

```
let m = 11
let Lrows = []
for i in range(m)
  let r = []
  for j in range(m)
    if i == j
      push(r, -2.0)
    elif i == j - 1 or i == j + 1
      push(r, 1.0)
    else
      push(r, 0.0)
    end
  end
  push(Lrows, r)
end
let Lap = np.array(Lrows)
let u = np.zeros([m, 1])
np.set(u, 5, 0, 1.0)              # bump in the middle
for t in range(200)
  u = np.add(u, np.mul(np.matmul(Lap, u), 0.4))
  np.set(u, 0, 0, 0.0)            # Dirichlet boundaries
  np.set(u, m - 1, 0, 0.0)
end
print(np.to_list(np.reshape(u, [m])))   # a symmetric, smooth bell
```

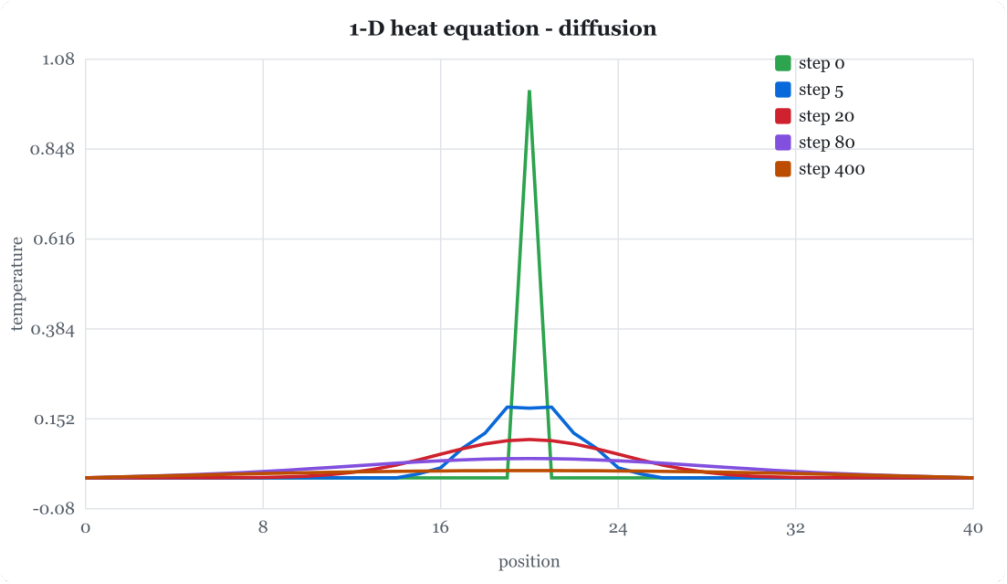


Figure 13.5: The diffusion of the initial temperature bump over time.

13.9.11 13.11.11 Newton's method — a nonlinear system

Newton's method finds `F(x) = 0` by solving the Jacobian `J` at each step: `x ← x − J⁻¹F(x)`. (Instead of taking the inverse we use `solve(J, F)`.) Let us solve `x²+y²=4` and `xy=1`.

```
let s = np.array([[2.0], [0.5]])
for it in range(20)
  let x = np.get(s, 0, 0)
  let y = np.get(s, 1, 0)
  let F = np.array([[x*x + y*y - 4.0], [x*y - 1.0]])
  let J = np.array([[2.0*x, 2.0*y], [y, x]])
  s = np.sub(s, np.reshape(np.solve(J, np.reshape(F, [2])), [2, 1]))
end
print(np.to_list(s))   # [[1.9318...], [0.5176...]]
```

13.9.12 13.11.12 Logistic regression

We train binary classification with the sigmoid `σ(z) = 1/(1+e⁻ᶻ)` and gradient descent. The gradient is `Xᵀ(p − y)/n`; we measure success with the **log loss**.

```
let X = np.array([[1,0.5],[1,1.5],[1,2.0],[1,3.0],[1,3.5],[1,4.5]])
let y = np.reshape(np.array([0,0,0,1,1,1]), [6,1])
let Xt = np.transpose(X)
def sigmoid(z)
  return np.div(1.0, np.add(1.0, np.exp(np.neg(z))))
end
let w = np.zeros([2, 1])
for it in range(3000)
  let p = sigmoid(np.matmul(X, w))
  w = np.sub(w, np.mul(np.matmul(Xt, np.sub(p, y)), 0.5 / 6.0))
end
```

```
let p = sigmoid(np.matmul(X, w))
print("probabilities:", np.to_list(np.reshape(p, [6])))
# [0.000006, 0.0026, 0.053, 0.963, 0.998, 0.99999]  -- the two classes are separated
```

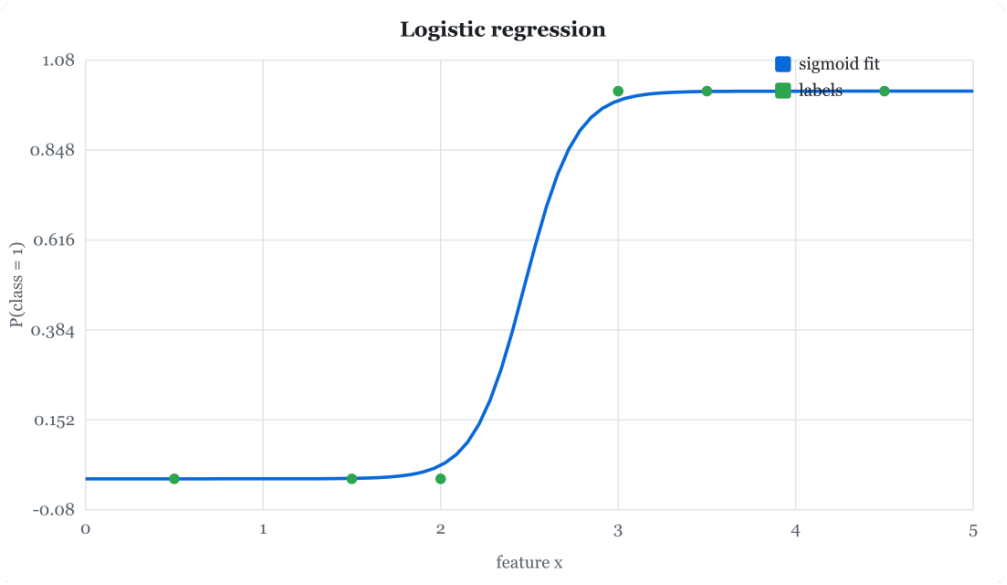


Figure 13.6: The trained sigmoid separating the classes.

13.9.13 13.11.13 Physics — projectile range

The power of vectorization: we compute a projectile's range with `R = v²·sin(2θ)/g` for dozens of angles **simultaneously**, without a loop.

```
let aci = np.linspace(0.0, 90.0, 91)
let rad = np.mul(aci, 3.141592653589793 / 180.0)
let R = np.mul(np.sin(np.mul(rad, 2.0)), 20.0 * 20.0 / 9.81)
print("farthest index:", np.argmax(R))   # 45  (farthest at 45 degrees)
```

13.9.14 13.11.14 Signal — moving average

Without slicing, we smooth a signal by multiplying it with a banded matrix: each row sums only its neighbors with a weight of `1/3`.

```
let sig = np.array([1, 2, 3, 4, 5, 6, 7, 8])
let k = 8
let Brows = []
for i in range(k)
  let r = []
  for j in range(k)
    if j == i - 1 or j == i or j == i + 1
      push(r, 1.0 / 3.0)
    else
```

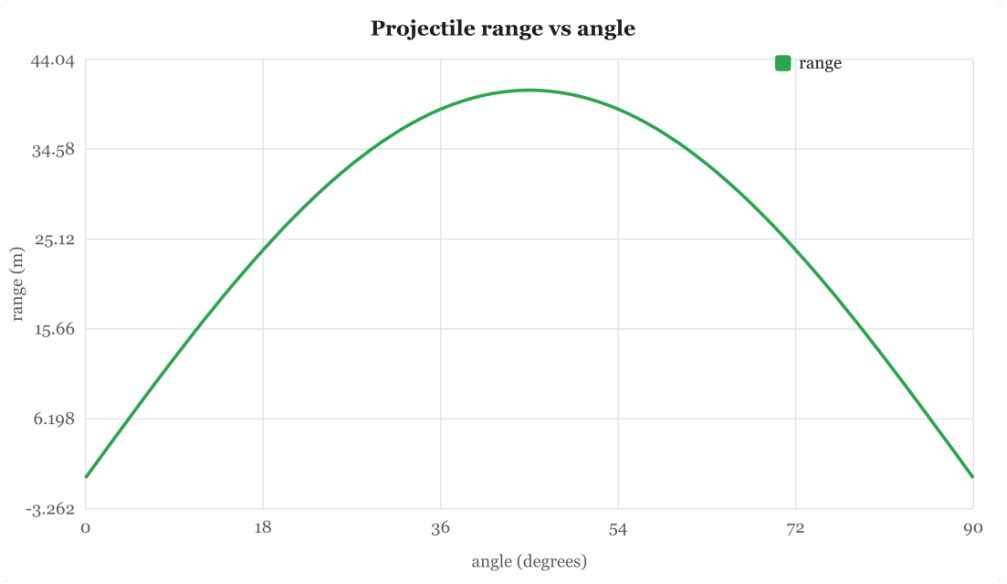


Figure 13.7: The range peaks at 45 degrees.

```
      push(r, 0.0)
    end
  end
  push(Brows, r)
end
print(np.to_list(np.matmul(np.array(Brows), np.reshape(sig, [k, 1]))))
# [[1], [2], [3], [4], [5], [6], [7], [5]]
```

13.10 13.12 Limits and design tips

cobranum is deliberately small. What it does not currently cover, and ways to work around it:

- **No comparison / masking** (`a > b`, `where`, filtering). For this reason, Monte Carlo problems of the "count the points that fall inside the circle" type are not array-natural; instead use integral-based estimation (13.11.7) or a scalar loop with the `random` module.
- **No slicing** (only `get`/`set`). Express work that requires neighbor access (derivative, convolution, PDE) with an **operator matrix** — as in the heat equation and moving average examples.
- **Only 1D and 2D**; no direct eigenvalue/SVD (obtained via power iteration + deflation).
- If you are repeating a computation in a hot loop, use the **fused** (`hypot`, `fma`) and **in-place** (`*_into`) forms instead of separate calls.

13.11 13.13 The source of the speed

cobranum draws its speed from three layers. The first is **dense memory**: an `ndarray` is not boxed objects but a contiguous buffer, cache-friendly. The second is **multi-core**: elementwise operations and reductions

are split across cores thanks to Cobra's GIL-free concurrency. The third is **hardware kernels**: on macOS, matrix multiplication, inverse, and solve go to Apple Accelerate's BLAS/LAPACK routines, while elementwise math goes to hand-written ARM NEON (SIMD) kernels; these use the AMX matrix coprocessor and wide vector instructions.

13.12 13.14 API summary

Group	Functions
Creation	`array`, `zeros`, `ones`, `full`, `eye`, `arange`, `linspace`
Inspection	`shape`, `ndim`, `size`, `reshape`, `to_list`, `str`, `get`, `set`
Elementwise	`add`, `sub`, `mul`, `div`, `pow`, `neg`
Univariate	`sqrt`, `exp`, `log`, `sin`, `cos`, `abs`
Fused / in-place	`hypot`, `fma`, `axpy`, `add_into`, `sub_into`, `mul_into`, `div_into`, `sqrt_into`, `exp_into`, `log_into`, `sin_into`, `cos_into`, `abs_into`, `neg_into`
Reductions	`sum`, `mean`, `min`, `max`, `prod` (+ axis in 2D), `var`, `std`, `norm`, `argmin`, `argmax`, `cumsum`, `clip`
Linear algebra	`matmul`/`dot`, `transpose`, `inv`, `det`, `solve`, `trace`, `diag`, `outer`, `concat`

cobranum combines Cobra's scripting comfort with the power of numerical computing: in a few lines you fit a regression, solve a system, advance a PDE, or train a model — all with a single small plugin.

Chapter 14

Venom — A Web Framework and an ORM

cobranum (Chapter 13) showed Cobra reaching into numerical computing. **Venom** shows it reaching the other way — into web applications. Venom is a Django-inspired web framework written entirely in Cobra: routing, an ORM over a real database, a template engine with inheritance, middleware, sessions, request parsing, static files and CSRF protection. Like cobranum, it is **not part of the `cobra` binary**. It is a library you import: you drop the `venom/` folder next to your app and `import` the pieces you need. The only hard requirement is a `cobra` on your PATH; the ORM additionally needs the native `oxidb.so` plugin.

```
your-project/
  venom/              <- the framework (copied in)
    router.cobra        routing + middleware
    response.cobra      response builders
    request.cobra       cookies / form / JSON body
    template.cobra      template engine (with inheritance)
    orm.cobra           OxiDB models (needs oxidb.so)
    session.cobra       cookie sessions
    static.cobra        static file serving
    config.cobra        settings
    csrf.cobra          CSRF protection
  app.cobra           <- your app
```

Because Cobra has no class inheritance, Venom leans on the patterns the language *does* have: decorators register routes, a *factory* function builds a model (instead of `class User(Model)`), and middleware composes as plain higher-order functions. The result is a familiar, Django-shaped stack expressed in Cobra's own idioms — and, because every piece is ordinary Cobra, you can read and modify all of it.

14.1 14.1 Routing and responses

A view is a function decorated with `@route(method, path)`. It receives the request dict and returns a response dict.

```
import "venom/router.cobra"
import "venom/response.cobra"

let route = router.route          # optional alias for a clean @route(...)

@route("GET", "/")
def home(req)
    return response.html("<h1>Hello from Venom</h1>")
end

@route("GET", "/users/<id>")
def show_user(req)
    return response.ok({"user_id": req["params"]["id"]})
end

router.serve(":8000")
```

```
$ cobra app.cobra
# then, in another terminal:
$ curl localhost:8000/
$ curl localhost:8000/users/42      # -> {"user_id":"42"}
```

A path segment written `<name>` matches exactly one segment (it does not cross `/`) and arrives as `req["params"]["name"]`; `<name*>` matches across `/` (used for static files). Matching is method-aware: the same path under a different method is a different route, and no match yields a 404. The shorthands `@router.get(path)`, `.post`, `.put`, `.delete` mirror `@route`.

The request dict comes straight from the standard library's `http.serve`, plus the `params` the router adds:

```
{ "method", "path", "query", "body", "headers", "params" }
```

Every helper in `response.cobra` returns the `{status, body, headers}` dict that `http.serve` expects:

helper	result
`ok(value)`	200, JSON body
`json_response(status, value)`	given status, JSON body
`html(body)`	200, `text/html`
`text(body)`	200, `text/plain`
`redirect(url)`	302 with `Location`
`status(code, body)`	given status, `text/plain`
`not_found(message)`	404

Remember that `http.serve` handlers run in parallel across cores (Cobra has no GIL, Chapter 10), so views must guard shared mutable state with `Mutex` or a `Channel`. Route registration happens once at startup and is read-only afterward.

14.2 14.2 The ORM, over OxiDB

Venom's data layer is a Django-ish ORM backed by **OxiDB**, a document database, through a native Go plugin called `oxidb.so`.

14.2.1 Under the hood: the `oxidb.so` plugin

The ORM is built on a thin plugin that wraps the OxiDB client. You can use it directly — it speaks documents and queries:

```
import "oxidb.so"

let db = oxidb.connect("localhost", 4444)
oxidb.create_collection(db, "users")
oxidb.insert(db, "users", {"name": "ada", "age": 36})
let docs = oxidb.find(db, "users", {"age": 36})
for doc in docs
    print(doc["name"])           # ada
end
print(oxidb.count(db, "users", {}))   # 1
oxidb.close(db)
```

A *collection* is a set of documents (dicts); a *query* is a dict, with OxiDB's operators (`$gte`, `$ne`, `$in`, …) as nested values. This is enough to build on, but you rarely write it by hand — the ORM gives you a far nicer surface.

14.2.2 Models by factory

Since Cobra has no class inheritance, a model is produced by a factory, `orm.model(db, collection)`, rather than by subclassing. The returned model carries CRUD methods bound to that collection.

```
import "venom/orm.cobra"

let db   = orm.connect("127.0.0.1", 4444)
let User = orm.model(db, "users").ensure()      # ensure() creates the collection

User.create({"name": "ada", "age": 36, "tags": ["math", "logic"]})
User.create_many([{"name": "bob", "age": 25}, {"name": "carol", "age": 41}])

let adults = User.filter({"age": {"$gte": 18}})  # OxiDB query operators
let ada    = User.get({"name": "ada"})           # first match, or null
print(User.count({}))                            # 3

User.update({"name": "ada"}, {"age": 37})        # wrapped in {"$set": ...}
User.delete({"name": "bob"})

orm.close(db)
```

The full model surface:

method	does
`ensure()` / `drop()`	create / drop the collection (returns self / nothing)
`create(doc)` / `create_many(docs)`	insert
`all()`	every document
`filter(query)`	documents matching an OxiDB query
`objects()`	a chainable query builder (below)

method	does
`get(query)`	first match, or **`null`** if none
`count(query)`	number of matches (`{}` = all)
`update(query, changes)` / `update_one(...)`	`$set` the flat `changes` dict
`delete(query)` / `delete_one(query)`	remove
`create_index(field)` / `create_unique_index(field)` / `indexes()`	indexes

Module-level helpers: `orm.connect(host, port)`, `orm.close(db)`, `orm.ping(db)`, `orm.transaction(db, fn)`.

A note on field access: OxiDB stores numbers faithfully, but reading them back you may want `int(doc["age"])` when you need a Cobra integer for comparison or arithmetic. Nested values return as Cobra lists and dicts, so `doc["tags"][0]` works as expected.

14.2.3 The query builder — `objects()`

`model.objects()` returns a **lazy, chainable QuerySet**, in the Django style. Nothing touches the database until a *terminal* method runs.

```
let top = User.objects()
    .filter({"age__gte": 18, "age__lt": 65})    # 18 <= age < 65 (merged on age)
    .exclude({"role": "guest"})
    .order_by("-age").order_by("name")           # primary -age, then name
    .offset(10).limit(5)
    .all()

User.objects().filter({"role": "admin"}).count()
User.objects().filter({"name__regex": "^a"}).first()      # or null
User.objects().filter({"tags__in": ["math"]}).exists()
User.objects().order_by("name").values("name")            # list of one field
User.objects().filter({"age__lt": 13}).delete()           # delete all matches
```

Lookups use Django-style suffixes — Cobra has no keyword arguments, so they are dict keys:

lookup	meaning
`field`	equals
`field__ne`	not equal
`field__gt` / `__gte` / `__lt` / `__lte`	comparisons
`field__in` / `__nin`	in / not in a list
`field__regex` / `__contains`	regex / substring match
`field__exists`	key present (value is a bool)

`exclude()` negates each lookup. Conditions on the **same field merge**, so `.filter({"age__gte": 18, "age__lt": 65})` becomes a single range query. The terminals are `all()`, `first()`, `count()`, `exists()`, `values(field)` and `delete()`. Filters push down to OxiDB; `order_by` / `limit` / `offset` are applied in Cobra on the returned rows. A `QuerySet` is **immutable** — each call returns a new one — so a base query is safe to reuse and branch:

```
let base = User.objects().filter({"active": true})
let admins  = base.filter({"role": "admin"}).all()    # base is unchanged
let recent  = base.order_by("-created").limit(10).all()
```

14.2.4 Indexes and transactions

```
User.create_index("age")           # speeds up queries on age
User.create_unique_index("email")  # also enforces uniqueness
print(len(User.indexes()))         # number of indexes
```

A transaction commits on success and rolls back if the function throws:

```
def add_two()
    User.create({"name": "erin", "age": 33})
    User.create({"name": "frank", "age": 28})
end
orm.transaction(db, add_two)       # both inserts commit together

def will_fail()
    User.create({"name": "ghost", "age": 99})
    throw "boom"                   # the throw triggers a rollback
end
try
    orm.transaction(db, will_fail)
catch e
end
# "ghost" was never committed — the transaction rolled back.
```

14.2.5 A complete ORM session

Putting it together — this is the shape of Venom's own end-to-end ORM test, with the result of each step in a comment:

```
import "venom/orm.cobra"

let db   = orm.connect("127.0.0.1", 4444)
let User = orm.model(db, "people").ensure()
User.drop()                            # start clean
User.ensure()

User.create({"name": "ada", "age": 36, "tags": ["math", "logic"]})
User.create({"name": "bob", "age": 25})
User.create_many([{"name": "carol", "age": 41}, {"name": "dave", "age": 30}])
print(int(User.count({})))             # 4

let ada = User.get({"name": "ada"})
print(ada["name"], int(ada["age"]))    # ada 36
print(ada["tags"][0])                  # math
print(User.get({"name": "nobody"}))    # null

print(len(User.filter({"age": {"$gte": 30}})))   # 3   (ada, carol, dave)

User.update({"name": "bob"}, {"age": 26})
print(int(User.get({"name": "bob"})["age"]))     # 26

User.create_index("age")
```

```
print(len(User.indexes()) >= 1)        # true

def add_two()
    User.create({"name": "erin", "age": 33})
    User.create({"name": "frank", "age": 28})
end
orm.transaction(db, add_two)
print(int(User.count({})))             # 6

User.delete({"name": "frank"})
print(int(User.count({})))             # 5

User.drop()
orm.close(db)
```

14.2.6 Building the `oxidb.so` plugin

The ORM imports `oxidb.so`, a Go plugin. As with any Cobra plugin (Chapter 12's note on plugins), it must be built next to the **exact `cobra` binary that loads it**, and it is Linux/macOS only. Build it from the Cobra repository and copy it beside `orm.cobra`:

```
$ cd <cobra-repo>/plugins/cobra-oxidb && ./build.sh   # builds oxidb.so + matching cobra
$ cp oxidb.so <your-project>/venom/oxidb.so
```

`oxidb.so` is intentionally git-ignored; each environment builds its own.

14.3 14.3 Templates

Server-rendered HTML without f-strings (Cobra has none): a small parser builds a node tree, then a renderer walks it. Values are **HTML-escaped by default**. `template.render(source, context)` is the workhorse; `render_file(path, ctx)` loads the source from disk first.

```
import "venom/template.cobra"

template.render("Hi {{ name }}!", {"name": "Ada"})         # "Hi Ada!"
template.render("[{{ nope }}]", {})                        # "[]"   (missing -> empty)
template.render("{{ user.name }}", {"user": {"name": "Bo"}})   # "Bo"  (dotted path)
template.render("{{ items.1 }}", {"items": ["x", "y"]})    # "y"   (list index)
```

Escaping is automatic; `|safe` opts out. Filters chain after a `|`:

```
template.render("{{ v }}",        {"v": "<b>&"})   # "&lt;b&gt;&amp;"   (escaped)
template.render("{{ v|safe }}",   {"v": "<b>hi</b>"})  # "<b>hi</b>"    (raw)
template.render("{{ v|upper }}",  {"v": "ada"})    # "ADA"
template.render("{{ v|length }}", {"v": [1, 2, 3]})  # "3"
```

Control flow uses `{% ... %}` tags. Conditions accept truthiness, `not path`, and `a == b` / `a != b`:

```
let tpl = "{% if user.admin %}admin{% else %}user{% endif %}"
template.render(tpl, {"user": {"admin": true}})    # "admin"

template.render(
  "<ul>{% for u in users %}<li>{{ u.name }}</li>{% endfor %}</ul>",
  {"users": [{"name": "Ada"}, {"name": "Bob"}]})
# "<ul><li>Ada</li><li>Bob</li></ul>"
```

14.3.1 Inheritance

Set the template directory once, then a child `{% extends %}` a base and fills its `{% block %}`s. `{% include "partial.html" %}` renders another template inline.

```
template.set_dir("templates")
```

`templates/base.html`:

```
<!doctype html><title>{% block title %}Site{% endblock %}</title>
<body>{% block content %}{% endblock %}</body>
```

`templates/guestbook.html`:

```
{% extends "base.html" %}
{% block title %}{{ app_name }} — guestbook{% endblock %}
{% block content %}
<h2>Guestbook</h2>
<ul>{% for msg in messages %}<li>{{ msg.text }} — {{ msg.author }}</li>{% endfor %}</ul>
<form method="post" action="/sign">
  <input type="hidden" name="csrf_token" value="{{ csrf_token }}">
  <input name="message" placeholder="Leave a message">
  <button>Sign</button>
</form>
{% endblock %}
```

A child's blocks override the base's same-named blocks; un-overridden blocks fall back to the base. Render it with `template.render_file("guestbook.html", ctx)` and return it as HTML from a view.

14.4 14.4 Middleware

Middleware wraps every request in an onion. Each is `fn(req, next)`: call `next(req)` to continue down the chain, or return a response to short-circuit. First registered is outermost; the run order for two is `a-in` → `b-in` → `handler` → `b-out` → `a-out`.

```
def logger(req, next)
    print(req["method"] + " " + req["path"])
    let res = next(req)                  # run the rest of the chain
    print("  -> " + str(res["status"]))
```

```
    return res                          # post-process on the way back out
end

def require_token(req, next)
    if req["headers"].has("token")
        return next(req)
    end
    return response.json_response(401, {"error": "unauthorized"})   # short-circuit
end

router.use(logger)                      # outer
router.use(require_token)                # inner
```

A middleware can edit `req` before `next`, edit the response after, or skip `next` entirely. Two hooks handle the edges:

```
router.set_not_found(fn)     # fn(req) -> response       (no route matched)
router.set_error_handler(fn) # fn(req, err) -> response  (a handler threw)
```

Without them, unmatched routes return a plain 404 and a thrown error becomes `500 - <message>`.

14.5 14.5 Sessions and request data

Add `router.use(session.middleware)` and every request gains `req["session"]`, a persistent dict keyed by a cookie (in-memory, mutex-guarded — handlers run in parallel):

```
import "venom/session.cobra"
router.use(session.middleware)

@route("GET", "/")
def home(req)
    let s = req["session"]
    s["views"] = s.get("views", 0) + 1     # survives across requests (same cookie)
    return response.text("views: " + str(s["views"]))
end
```

`request.cobra` parses incoming data:

```
import "venom/request.cobra"

request.cookies(req)      # Cookie header  -> {"session": "abc", ...}
request.form(req)         # urlencoded body -> {"name": "John Doe", ...}
request.json_body(req)    # JSON body -> a value (or null)
```

`form` decodes `+` and `%XX` escapes through the native `http.url_decode` (UTF-8 correct, so multibyte values work). GET query params are already parsed into `req["query"]`.

14.6 14.6 Static files and CSRF

Serve a directory under a URL prefix (the router's `<name*>` catch-all matches across `/`; Content-Type is inferred from the extension and `..` is rejected):

```
import "venom/static.cobra"
static.serve_dir("/static", "public")   # GET /static/css/app.css -> public/css/app.css
```

CSRF protection requires a per-session token on every unsafe request (POST/PUT/PATCH/DELETE); safe methods pass. It runs **after** the session middleware (it stores the token in the session):

```
import "venom/csrf.cobra"
router.use(session.middleware)
router.use(csrf.middleware)

# In a view, pass the token to the template via req["csrf_token"]; the form
# submits it back as a hidden field "csrf_token" (or the X-Csrf-Token header
# for AJAX/JSON). csrf.exempt("/api") skips a stateless prefix.
```

14.7 14.7 Settings

A tiny key/value store — Django's `settings.py`, in one dict:

```
import "venom/config.cobra"

config.load({"app_name": "Venom News", "debug": true, "db_port": 4444})
config.get("app_name", "Venom")     # value, or the fallback if unset
config.set("debug", false)
config.has("db_port")               # true

config.get / set / has / load / all / clear.
```

14.8 14.8 Capstones

14.8.1 A JSON REST API — router + ORM

Routing and the ORM together make a compact REST service. Handlers open a fresh OxiDB connection per request (parallel handlers must not share one socket) and drive listing from the URL query string with the QuerySet builder.

```
import "venom/router.cobra"
import "venom/response.cobra"
import "venom/orm.cobra"
import "json"
```

```
let route = router.route
let COLLECTION = "users"

let boot = orm.connect("127.0.0.1", 4444)     # create the collection once
orm.model(boot, COLLECTION).ensure()
orm.close(boot)

def with_users(fn)                            # fresh per-request connection
    let db = orm.connect("127.0.0.1", 4444)
    let result = fn(orm.model(db, COLLECTION))
    orm.close(db)
    return result
end

@route("POST", "/users")
def create_user(req)
    def run(User)
        let body = json.parse(req["body"])
        User.create(body)
        return response.json_response(201, body)
    end
    return with_users(run)
end

@route("GET", "/users/<name>")
def get_user(req)
    def run(User)
        let user = User.get({"name": req["params"]["name"]})
        if user == null
            return response.not_found("no such user")
        end
        return response.ok(user)
    end
    return with_users(run)
end

@route("GET", "/users")
def list_users(req)
    def run(User)
        let q  = req["query"]
        let qs = User.objects()
        if q.get("q", "") != ""
            qs = qs.filter({"name__contains": q["q"]})    # search
        end
        if q.get("role", "") != ""
            qs = qs.filter({"role": q["role"]})
        end
        return response.ok(qs.order_by(q.get("order", "name")).limit(20).all())
    end
    return with_users(run)
end

router.serve(":8000")
```

```
$ curl -X POST -d '{"name":"ada","age":36,"role":"admin"}' localhost:8000/users
$ curl localhost:8000/users/ada
$ curl 'localhost:8000/users?role=admin&order=-age&limit=10'
```

14.8.2 A server-rendered, database-backed guestbook

The second capstone uses *everything*: sessions, CSRF, templates, the ORM and forms. Visitors leave messages stored in OxiDB, attributed to a name kept in the session.

```
import "venom/router.cobra"
import "venom/response.cobra"
import "venom/template.cobra"
import "venom/session.cobra"
import "venom/csrf.cobra"
import "venom/request.cobra"
import "venom/orm.cobra"
import "time"

template.set_dir("templates")
router.use(session.middleware)
router.use(csrf.middleware)

let COLLECTION = "guestbook"
let boot = orm.connect("127.0.0.1", 4444)
orm.model(boot, COLLECTION).ensure()
orm.close(boot)

def with_messages(fn)
    let db = orm.connect("127.0.0.1", 4444)
    let result = fn(orm.model(db, COLLECTION))
    orm.close(db)
    return result
end

def render_page(req, M)
    let msgs = M.objects().order_by("-ts").limit(20).all()
    return response.html(template.render_file("guestbook.html", {
        "app_name":   "Venom Guestbook",
        "messages":   msgs,
        "name":       req["session"].get("name", ""),
        "csrf_token": req["csrf_token"]
    }))
end

@route("GET", "/")
def home(req)
    def load(M)
        return render_page(req, M)
    end
    return with_messages(load)
end

@route("POST", "/sign")
```

```
def sign(req)
    let form = request.form(req)
    def save(M)
        if form.get("name", "") != ""
            req["session"]["name"] = form["name"]     # remember the visitor
        end
        if form.get("message", "") != ""
            M.create({
                "text":   form["message"],
                "author": req["session"].get("name", "Anonymous"),
                "ts":     int(time.now()),
                "at":     time.format(time.now(), "%Y-%m-%d %H:%M")
            })
        end
        return render_page(req, M)
    end
    return with_messages(save)
end

router.serve(":8000")
```

Paired with the `guestbook.html` template from §14.3, this is a complete little site — persistent messages, a remembered author, CSRF-protected forms, and an escaped, inherited template — in well under a hundred lines.

14.9 14.9 Testing without a socket

You do not need to open a port to test views: `router.dispatch(req)` runs the whole routing + middleware chain on a request dict and returns the response, so tests are plain function calls that run identically on both engines.

```
import "venom/router.cobra"
import "venom/response.cobra"

@route("GET", "/ping")
def ping(req)
    return response.text("pong")
end

let res = router.dispatch({"method": "GET", "path": "/ping",
                           "query": {}, "headers": {}, "body": ""})
print(res["status"], res["body"])   # 200 pong
```

```
$ cobra test.cobra
$ cobra --vm test.cobra      # same result on the bytecode VM
```

The template engine needs no server or plugin at all, so template tests are pure `render(...)` assertions. `router.clear()` resets routes, middleware and hooks between tests.

Between cobranum's numerical computing and Venom's database-backed web stack, Cobra covers a wide arc for a small language with a simple syntax. Venom is a library, not part of the language, so it evolves on its own — and because it is all ordinary Cobra, the whole framework is yours to read, extend and reshape.