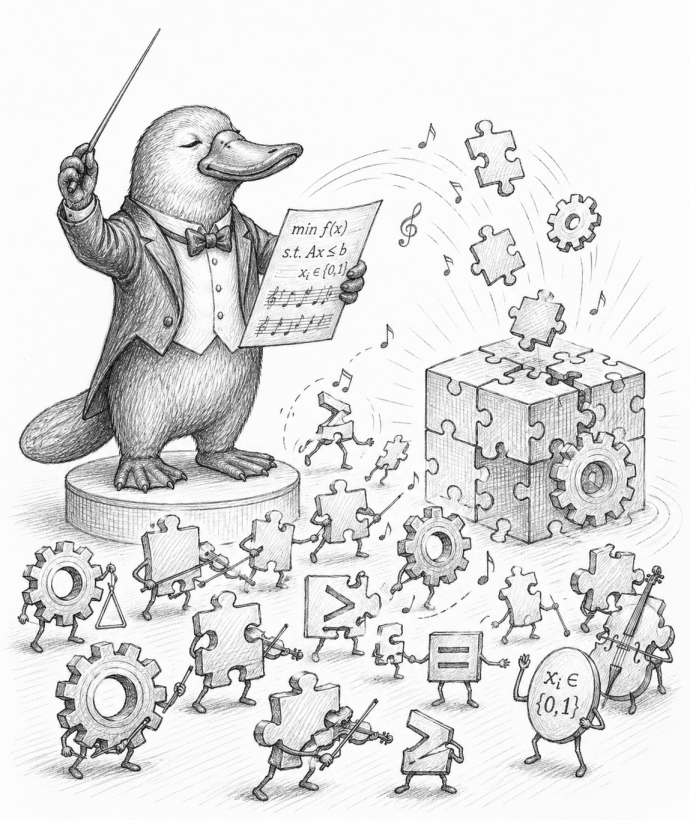
Declarative Modeling with CP-SAT

Managing the Complexity of Optimization Modeling



Dr. Dominik Krupke

Draft — May 6, 2026

Abstract

When combinatorial problems grow beyond what ad hoc modeling can handle, hands-on solver experience alone is not enough. Modelers need a structured way to reason about the problem before writing solver code. This manuscript presents such a structure, organized around four ideas: mathematical notation is a working tool rather than a gatekeeping ritual; optimization models decompose cleanly into five parts—entities, parameters, decision variables, constraints, and an objective; a *verifier*, implemented as a lightweight Python function that judges the feasibility and quality of candidate solutions, provides an orthogonal contract on what the problem actually means, independent of any solver; and the choice of solver shapes how a model is best expressed, with this manuscript committing to CP-SAT.

The manuscript focuses on managing the mental complexity of turning an informal problem into a structured model. It does not aim to cover optimization theory or solver internals in depth, nor does it provide a comprehensive CP-SAT reference. Instead, it emphasizes a disciplined modeling workflow that produces a correct formulation: one that a verifier can check and a solver can execute. Performance engineering—tightening formulations, exploiting structure, and tuning solvers—is treated as a subsequent step.

The primary audience is computer scientists and software engineers with experience in Python and basic discrete mathematics who want a systematic approach to modeling combinatorial optimization problems. Readers with a background in mathematical optimization will find much of the material familiar, although the verifier concept may still offer a useful perspective.

Contents

Chapter 1

From Procedure to Description

> *"Civilization advances by extending the number of important operations which we can perform without thinking about them."* — Alfred North Whitehead

A software engineer's instinct on a combinatorial problem is to design an algorithm: pick a strategy — greedy, backtracking, branch-and-bound, dynamic programming — and commit to the search structure, the data structures, the order of exploration, the pruning rules. The skill is real and worth keeping. Many problems are best solved by writing the procedure that solves them.

On most real combinatorial problems, however, a hand-rolled procedure is far from the state of the art. Industrial solvers represent decades of accumulated engineering — clause learning, constraint propagation, restart strategies, parallel workers, instance-tuned heuristics — that no individual programmer can reproduce in a project. The pragmatic move on a hard combinatorial problem is therefore not to write a solver, but to *describe* the problem in a form that an existing solver can consume, and let that solver search.

This is a paradigm shift, not a swap of one technique for another. The two mindsets answer different questions:

- The **procedural** view answers *how do I compute an answer?* It is the perspective of an algorithm designer. The output is a procedure that, when run, produces a solution.
- The **declarative** view answers *what counts as an answer?* It is the perspective of a problem describer. The output is a *model* — a precise, executable statement of which configurations are acceptable and which is best.

Both views are legitimate, and a working optimization practitioner moves between them. Most readers of this manuscript will have stronger code intuition than mathematical-notation intuition; that intuition does not become useless under the new paradigm — most of the work is still code — but it has to be redirected, since the code you write here *describes* rather than *executes*.

The solver we will commit to in this manuscript is **CP-SAT**, the constraint-programming solver that ships inside Google's OR-Tools. The choice is not innocent: different solvers reward different idioms, and a model written for CP-SAT will not be byte-for-byte the model you would write for a mixed-integer programming solver or a SAT solver. The underlying paradigm shift, and most of the modeling vocabulary, transfers without change. The solver-specific details are local to the last few chapters.

1.1 A Concrete Contrast: 0/1 Knapsack

The classical **0/1 knapsack** makes the shift concrete. Given a set of items I, each with a weight w_i and a value c_i, and a knapsack of capacity C, choose a subset of items whose total weight does not exceed C and whose total value is as large as possible. The problem is a useful contrast because both an algorithmic recipe and a declarative formulation are short enough to fit on one page.

1.1.1 The procedural answer

The textbook procedural solution is a dynamic program. We define a table `dp[i][w]` whose entry is the best total value achievable using the first i items and capacity at most w, and we fill the table by deciding, for each (i, w), whether to take item i or leave it.

```
def knapsack(weights, values, C):
    n = len(weights)
    dp = [[0] * (C + 1) for _ in range(n + 1)]
    for i in range(1, n + 1):
        for w in range(C + 1):
            dp[i][w] = dp[i - 1][w]
            if weights[i - 1] <= w:
                take = dp[i - 1][w - weights[i - 1]] + values[i - 1]
                if take > dp[i][w]:
                    dp[i][w] = take
    return dp[n][C]
```

Several non-trivial decisions sit inside that function. We chose the recursion (take versus leave), the table layout (items by capacity), the fill order (forward in i, forward in w), and the off-by-one indexing between the table and the input arrays. None of these decisions is part of the *problem*; all of them are part of one possible *solution*.

The running time is proportional to $|I| \cdot C$, which is *pseudo-polynomial*: linear in the numerical value of C, not in the bits needed to write C down. When C reaches the millions the table no longer fits in memory, and the algorithm becomes unusable even for trivially small $|I|$. Other algorithmic approaches exist, but they tend to be even more complex to implement.

1.1.2 The declarative answer

The same problem written as mathematics is three lines:

$$\begin{aligned} \max \quad & \sum_{i \in I} c_i \, x_i \\ \text{s.t.} \quad & \sum_{i \in I} w_i \, x_i \;\le\; C \\ & x_i \in \{0, 1\} \qquad \forall i \in I. \end{aligned}$$

Read aloud: maximize the total value over a choice of items, subject to the total weight not exceeding the capacity, where each x_i is a binary decision (take or leave). Three lines describe the entire problem and commit to no algorithm at all. This is the form a paper or a textbook would use, and Chapter 2 develops the notation needed to read it fluently.

The translation from this math into a working CP-SAT model is almost line-for-line:

```
from ortools.sat.python import cp_model
```

```
model = cp_model.CpModel()
x = [model.new_bool_var(f"x_{i}") for i in I]

model.add(sum(w[i] * x[i] for i in I) <= C)
model.maximize(sum(c[i] * x[i] for i in I))

solver = cp_model.CpSolver()
solver.solve(model)
```

One Boolean variable per item, one linear inequality for the capacity, one linear objective, one call to the solver. There is no loop over the capacity, no table, no decision about the order of exploration. CP-SAT does all of that internally. The gain, for this example, is primarily *modeling style*: the code mirrors the mathematical statement directly and is easy to extend. For knapsack specifically, specialized algorithms remain very strong baselines, so the point of the example is the declarative form, not a claim that CP-SAT is the canonical fastest method for this problem family.

1.1.3 Comparing the two

The two formulations differ in three ways that matter.

The first is performance. The dynamic program's $|I| \cdot C$ running time is fine for moderate capacities and bad for large ones, while CP-SAT inherits decades of constraint-solver research — clause learning, propagation, restarts, parallel workers — none of which we had to write. This does not mean CP-SAT is automatically the best method for knapsack; on narrow problem classes a tailored algorithm is often the right baseline. The declarative model gives access to the solver's machinery without requiring its implementation.

The second is reading effort. The procedural version mixes the problem with its solution: to read it six months from now, you have to reconstruct why the recursion is what it is, why the loops run in this order, and what each index means. The declarative version reads as a problem statement — every line corresponds to a fact about the problem, and the model is reviewable and explainable in roughly the same form a colleague would expect on a whiteboard.

The third is extensibility, and it is the deepest of the three. Suppose a new requirement appears: items A and B are mutually exclusive, or at least three of items C, D, E must be taken. In the dynamic program the recursion changes, the table changes, possibly the time complexity changes, and we are back at the whiteboard. In the declarative model each new requirement is one extra line — $x_A + x_B \leq 1$ for the mutual exclusion, $x_C + x_D + x_E \geq 3$ for the cardinality — and the solver absorbs the addition without any change to how the search proceeds. Real problems mutate; their constraints arrive in waves, first the one in the textbook, then the regulatory rule, then the customer's preference, then the operational quirk. A model that absorbs new constraints by adding lines is incomparably easier to maintain than a procedure that has to be redesigned each time.

1.2 Modeling, Not Magic

Handing a problem to a solver does not make it easy. NP-hard problems remain NP-hard; the solver does not abolish complexity, it merely searches more cleverly than you would have. What changes with declarative modeling is not whether the problem is hard but how large an *instance* you can practically solve.

This last point is where the real skill of optimization modeling lives. The same problem can be encoded in many mathematically equivalent ways, and the encodings differ enormously in how well a solver handles them. A naive encoding may time out at fifty variables; an idiomatic encoding may

solve in two seconds at five thousand. The two models are saying the same thing about the same problem, but only one of them is saying it in a way the solver can exploit.

A few patterns recur. Solvers are tuned for certain shapes of constraint, so a formulation that reaches for the solver's native vocabulary — global constraints, indicator variables, well-known reformulations — usually beats one that reinvents the same logic from arithmetic primitives. Encodings that rule out symmetric or equivalent solutions, removing duplicates the solver would otherwise explore, outperform encodings that leave that freedom in, even when both are mathematically correct. None of this is visible from the problem statement; choosing the representation is part of the modeler's job, and two correct models for the same problem can differ in solving time by orders of magnitude.

The remaining chapters cover the basics to get you rolling with building and solving your first models. Performance engineering of models is a topic for later.

1.3 What This Manuscript Develops

Four ideas run through the rest of the book, and each one is the subject of one or more chapters.

The first is that *mathematical notation is a tool, not gatekeeping.* Optimization papers and textbooks state problems in notation because notation is shorter and less ambiguous than prose — not because the community is exclusive. Reading and writing it is a habit anyone working with optimization can pick up. Chapter 2 is a fast alignment layer for that notation: sets, variables, expressions, predicates, and the bridges between the numeric and Boolean parts of the language.

The second is that *optimization models decompose into five parts*: the entities the problem talks about, the parameters that describe a particular instance, the decision variables, the constraints that restrict their combinations, and the objective that ranks the feasible ones. This decomposition is the structure every model in the manuscript uses. Chapter 3 walks through it in a from-scratch recipe.

The third is that *a verifier is an orthogonal contract on what the problem actually means.* A verifier is a small piece of plain Python that takes a candidate solution and answers two questions: is it valid, and how good is it? It does not produce solutions; it judges them. Writing one is independent of which solver eventually does the search, and it pins the problem definition down in code so later modeling decisions cannot drift away from it. Chapter 4 develops the verifier pattern; later chapters treat it as optional but useful.

The fourth is that *knowing the solver's language matters.* The same problem can be modeled in many mathematically equivalent ways, and an idiomatic encoding for one solver looks different from an idiomatic encoding for another. This manuscript commits to **CP-SAT**, the constraint-programming solver that ships inside Google's OR-Tools. CP-SAT has an expressive vocabulary — global constraints for routing, scheduling, packing, and combinatorial structure — and Chapter 5–Chapter 8 lay it out: a first end-to-end model, then variables, constraints, and objectives in reference form. Chapter 9 closes the manuscript by reflecting on these four ideas and naming a few practical moves that come up in real modeling work.

Note

This manuscript works directly with CP-SAT's Python API rather than with an algebraic modeling language such as AMPL, GAMS, Pyomo, MiniZinc, or CPMpy. Algebraic modeling languages add a layer of abstraction above solver APIs and often allow the same model to target multiple solvers. They are valuable tools, but they would obscure the solver-specific idioms this manuscript wants to teach. CP-SAT's Python API is already high-level enough for introductory modeling, while still exposing the modeling choices that matter for performance.

Chapter 2

The Language of Optimization Models

> *"A good notation has a subtlety and suggestiveness which at times makes it almost seem like a live teacher."* — Bertrand Russell

This chapter introduces a common notation for discrete optimization. The focus is on MIP- and CP-style notation that dominates much of the modeling literature; much of the same vocabulary carries over to SAT modeling, even though that community uses its own surface syntax. Other communities and textbooks adopt variants—different letter conventions, bracket styles, and shorthands for global constraints—and we do not claim that the choices below are universal.

Fluency in this notation yields three benefits. First, it makes research papers and textbooks readable, allowing you to extract ideas rather than rederive them. Second, once a problem exceeds what fits in working memory, it is faster to sketch a precise model on paper than to move directly to code. Third, it provides a shared, unambiguous language for personal notes, whiteboard discussions, and paper drafts.

Most of the material is standard, and the chapter also serves as a reference. If you are unsure how much you already know, use the section headings as a checklist.

2.1 Notation in Action

Recall the 0/1 knapsack: items I with weights w_i and values c_i, a knapsack of capacity C, and a binary choice per item. Compressed into a single line, the model reads

$$\max_{x \in \{0,1\}^{|I|}} \Big\{ \sum_{i \in I} c_i \, x_i \;\Big|\; \sum_{i \in I} w_i \, x_i \le C \Big\}.$$

This line specifies, in order, *what we choose* (a binary vector x, one component per item), *what we optimize* (a weighted sum of values), and *what is allowed* (the weighted sum of weights does not exceed C). Each symbol is functional, and a reader familiar with the conventions can reconstruct the problem without additional prose.

Now suppose the values are uncertain: a simulator provides $|S|$ scenarios, and we require a packing that performs well in the worst case. The model changes minimally:

$$\max_{x \in \{0,1\}^{|I|}} \; \min_{s \in S} \Big\{ \sum_{i \in I} c_i^s \, x_i \;\Big|\; \sum_{i \in I} w_i \, x_i \le C \Big\}.$$

The decision variables, parameters, and constraint remain unchanged; only the objective acquires a $\min_{s \in S}$, and the values carry a scenario index. The formulation remains flexible: replacing $\min_{s \in S}$

with $\sum_{s \in S} p_s$ yields an expected-value objective under scenario probabilities p_s; replacing it with a quantile operator yields a value-at-risk objective. The remainder of the model is unaffected. In contrast, a prose formulation would require substantial rewriting for each variant.

Next, consider two knapsacks: the original with capacity C, and a second with capacity C' that only accepts light items, $I' = \{i \in I \mid w_i \leq w'\}$. We must choose one knapsack and pack it, while still maximizing the worst-case value. Rather than inlining everything into a single expression, define the feasible regions:

$$\mathcal{F}_0 \;=\; \big\{\, x \in \{0,1\}^{|I|} \;\big|\; \textstyle\sum_{i \in I} w_i x_i \leq C \,\big\},$$

$$\mathcal{F}_1 \;=\; \big\{\, x \in \{0,1\}^{|I|} \;\big|\; \textstyle\sum_{i \in I} w_i x_i \leq C',\;\; x_i = 0 \;\; \forall i \in I \setminus I' \,\big\},$$

and combine them using a set union:

$$\max_{x \,\in\, \mathcal{F}_0 \cup \mathcal{F}_1} \;\min_{s \in S} \; \sum_{i \in I} c_i^s \, x_i.$$

Each line serves a distinct purpose. The decomposition also exposes structure: the feasible regions $\mathcal{F}_0$ and $\mathcal{F}_1$ are independent of the scenarios, whereas the objective depends on them. Thus, uncertainty affects *what is optimized*, not *what is feasible*. Once defined, these regions can be reused—for example, to state bounds, define relaxations, or compare variants—without repeating their definitions.

The notation integrates naturally with prose: symbols appear inline (I', $\mathcal{F}_1$), in displayed equations, and across paragraphs without friction. The two registers remain consistent and mutually reinforcing.

The remainder of this chapter develops the vocabulary underlying these examples: sets and indices, parameters, graphs, variables and domains, expressions, objectives, constraints, and logical connectives.

2.2 Sets, Indices, and Parameters

Consider a city planning problem: choose locations for ambulance stations. Let C denote the set of neighborhoods to serve and F the set of candidate sites. The goal is to open as few stations as possible while ensuring that every neighborhood lies within a five-minute drive of at least one open station. A drive-time function $d : C \times F \to \mathbb{R}_{\geq 0}$ returns the travel time in minutes; for $c \in C$ and $f \in F$, $d(c, f)$ is the time from c to f. For a fixed site $f \in F$, the neighborhoods reachable within five minutes form a subset of C:

$$N_f \;=\; \{\, c \in C \;\mid\; d(c, f) \leq 5 \,\}.$$

The braces with a vertical bar denote *set-builder notation*: all elements of C that satisfy the condition. The subscript f indicates that this defines one set per candidate site—an indexed family $\{N_f\}_{f \in F}$.

Tip

Maintain a table of sets and parameters as a glossary. For the ambulance problem:

Symbol	Meaning
C	set of neighborhoods to serve
F	set of candidate sites
$d : C \times F \to \mathbb{R}_{\geq 0}$	drive-time function
N_f	neighborhoods reachable from site f within five minutes

If multiple variable types appear, create an analogous table for variables, including their domains.

The decision is to select a subset $F^\star \subseteq F$ of sites to open. The coverage requirement is

$$\bigcup_{f \in F^\star} N_f \;=\; C.$$

Given $F^\star$, the neighborhoods uniquely covered by an open site $f \in F^\star$ are

$$N_f \;\setminus \bigcup_{g \in F^\star \setminus \{f\}} N_g.$$

If this set is empty, site f is redundant. Another useful observation: if $N_g \subseteq N_f$ for some $g \in F$, then g is dominated and can be removed from F before solving, since any coverage it provides is already achieved by f.

The set operators used above, along with several standard ones:

$a \in A$	membership: a is an element of A
$a \notin A$	non-membership: a is not an element of A
$A \cup B$	union: elements in A or B (or both)
$A \cap B$	intersection: elements in both
$A \setminus B$	difference: elements in A but not in B
$A \subseteq B$	subset relation
$A \times B$	Cartesian product: ordered pairs (a, b)
$\lvert A \rvert$	cardinality: number of elements
$\emptyset$	empty set
$\{a \in A \mid P(a)\}$	set-builder notation

Cartesian products describe relations and graphs: for example, an arc set $E \subseteq V \times V$, a set of feasible (worker, job) assignments, or the edge set of a conflict graph.

A few naming conventions recur. Sets typically use uppercase Latin letters, with corresponding lowercase symbols for elements: $i \in I$, $j \in J$, $v \in V$. Certain letters carry conventional meanings—V, E for vertices and edges, T for time periods, K for constraint indices—but these are conventions, not requirements. A symbol is defined by its declaration, not by its name.

Data can be written in two interchangeable forms. A parameter may appear as a subscripted quantity, such as w_i or a_{ij}, or as a function, such as $d(c, f)$. Formally, these are equivalent: $(w_i)_{i \in I}$

and $w : I \to \mathbb{R}$ define the same mapping. Subscripts are compact for tabulated data; functional notation reads more naturally when arguments span multiple index sets or when the value is conceptually computed (e.g., a distance or cost query). The notation $d(c, f)$ reflects the latter interpretation.

2.3 Graphs

A graph is a common form of structured set: a pair $G = (V, E)$, where V is a finite set of vertices and $E \subseteq V \times V$ is a set of arcs. Vertices are typically denoted by v (or w when needed), and $(v, w) \in E$ denotes an arc from v to w, often abbreviated as vw. An *undirected graph* uses unordered pairs:

$$G = (V, E), \qquad \{v, w\} \in E.$$

The choice between directed and undirected graphs depends on the application—flows are directed, whereas relationships such as friendships are not. The modeling choice can significantly affect the resulting formulation.

For a directed graph, the *out-neighborhood* and *in-neighborhood* of a vertex v are

$$N^+(v) = \{\, w \in V \mid (v, w) \in E \,\}, \qquad N^-(v) = \{\, u \in V \mid (u, v) \in E \,\}.$$

The notation follows the standard upstream/downstream interpretation $u \to v \to w$: u is a predecessor of v, and w is a successor. For an undirected graph, there is a single neighborhood,

$$N(v) = \{\, w \in V \mid \{v, w\} \in E \,\}.$$

A related convention reserves $\delta^+(v)$ and $\delta^-(v)$ for the *sets of arcs* leaving and entering v,

$$\delta^+(v) = \{\, (v, w) \in E \,\}, \qquad \delta^-(v) = \{\, (u, v) \in E \,\}.$$

The distinction matters in flow formulations, where the natural sums are over arcs rather than over neighboring vertices.

A *subgraph* is a pair (V', E') with $V' \subseteq V$ and $E' \subseteq E$. It is *induced* by V' if its edge set consists exactly of those edges in G with both endpoints in V':

$$E' = \{\, (v, w) \in E \mid v, w \in V' \,\}.$$

The same definition applies to undirected graphs by interpreting (v, w) as $\{v, w\}$. We write $G[V']$ for the subgraph induced by V'. Induced subgraphs underlie structures such as cliques, independent sets, and dense subgraphs. For example, a clique is a set $V' \subseteq V$ such that $G[V']$ is complete.

As an example, consider constructing a worker's shift from a set of tasks T. Each task $t \in T$ has a start time s_t and an end time $e_t \geq s_t$, and transitioning from t to t' requires time $\tau_{t,t'} \geq 0$. The feasible sequences of tasks can be represented as a graph with

$$V = T \cup \{v_{\text{start}}, v_{\text{end}}\}, \qquad E \;=\; \left\{\, (t, t') \in T \times T \;\middle|\; e_t + \tau_{t,t'} \leq s_{t'} \,\right\} \;\cup\; \left(\{v_{\text{start}}\} \times T\right) \;\cup\; \left(T \times \{v_{\text{end}}\}\right).$$

An arc (t, t') indicates that t' can follow t in a feasible schedule. The vertices v_{start} and v_{end} represent the beginning and end of the shift. Since $e_t \geq s_t$ and $\tau \geq 0$, all arcs move forward in time, so G is a *directed acyclic graph* (DAG).

A single worker's schedule corresponds to a path from v_{start} to v_{end} in G. If every task must be assigned to exactly one worker and each worker performs one feasible sequence, then minimizing the number of workers is the minimum path-cover problem on this DAG, which reduces to bipartite

matching. With per-arc costs, the same structure yields a minimum-cost flow formulation, typically after splitting each task vertex to enforce that exactly one unit of flow uses it.

Tip

Modeling a problem as a graph provides access to a broad range of graph algorithms, which can serve as subroutines or, in some cases, solve the problem directly, as in this example.

2.4 Variables and Domains

The decision variables x_i in the Knapsack problem lived in $\{0,1\}^{|I|}$—one bit per item, indicating whether the item is selected. Consider the same model with a modified domain:

$$\max \sum_{i \in I} c_i \, x_i \quad \text{s.t.} \quad \sum_{i \in I} w_i \, x_i \leq C, \qquad x_i \in [0,1].$$

The objective, constraint, and data are unchanged; only the domain differs. This is the *fractional knapsack* problem, where items can be taken in any proportion between zero and one. It admits a greedy solution in $\mathcal{O}(|I| \log |I|)$ time by sorting items by value density. In contrast, the original formulation with $x_i \in \{0,1\}$ is NP-hard. Changing the domain again to $x_i \in \mathbb{N}_0$ yields the *unbounded knapsack* problem, where any non-negative number of copies of each item may be selected.

Each x_i is a *decision variable*—a quantity determined by the solver—and its *domain* specifies the admissible values. Common domains include:

Domain	Typical use
$\mathbb{R}$	continuous quantities (flows, prices, fractions)
$\mathbb{Z}$	integers (indices, offsets)
$\mathbb{N}_0$	non-negative integers (counts, quantities)
$\{0,1\}$	binary decisions
$\{1,2,\ldots,k\}$	finite enumerations (machines, modes)
$\{\text{red},\text{green}\}$	symbolic enumerations

Bounds are a special case of domains: $l_i \leq x_i \leq u_i$ is equivalent to $x_i \in [l_i, u_i]$, and $x_i \geq 0$ to $x_i \in [0,\infty)$. When multiple variables share a domain, we write $x \in \mathbb{R}^n$, $x \in \mathbb{Z}^n$, or $x \in \{0,1\}^n$.

Variable names follow informal conventions—x for primary decisions, y for auxiliary variables, s for slack variables, and z for objective values—but these conventions are not binding.

2.5 Expressions

Expressions are quantities formed from variables and parameters; once the variables are fixed, an expression evaluates to a scalar. It is useful to distinguish between linear and nonlinear expressions, because the distinction has algorithmic consequences. Models composed entirely of linear

expressions—*linear programs* and *mixed-integer linear programs*—are significantly better understood than their nonlinear counterparts. In practice, nonlinear formulations are often transformed into linear ones, either exactly or through approximation.

2.5.1 Linear expressions

A linear expression is a weighted sum of variables:

$$\sum_{i \in I} c_i x_i.$$

This form appears in many contexts: total value in knapsack, total assignment cost, or total flow into a vertex. Nested sums extend the structure over Cartesian products:

$$\sum_{i \in I} \sum_{j \in J} a_{ij} x_{ij} \;=\; \sum_{(i,j) \in I \times J} a_{ij} x_{ij}.$$

The right-hand form is convenient when the index set is a subset of $I \times J$, such as the edge set of a graph:

$$\sum_{(i,j) \in E} x_{ij}.$$

In vector notation, $\sum_i c_i x_i$ is written as $c^\top x$, and a system of linear constraints as $Ax \le b$. The indexed and vector forms represent the same object, differing only in presentation.

2.5.2 Nonlinear expressions

Any expression that is not a linear combination of variables is *nonlinear*. Three common patterns recur.

The first consists of variables passed through nonlinear functions, such as products or transcendental functions:

$$x_i^2, \quad x_i x_j, \quad \exp(x_i), \quad \log(x_i).$$

The second pattern aggregates over an index set, analogous to $\sum$ in Section 2.5.1, but produces a nonlinear result:

$$\max_{i \in I} g_i(x), \quad \min_{i \in I} g_i(x), \quad \text{median}_{i \in I}\, g_i(x), \quad \text{mode}_{i \in I}\, g_i(x), \quad \text{quantile}^{q}_{i \in I}\, g_i(x).$$

The third pattern connects logical predicates to numeric expressions. The *Iverson bracket* maps a predicate P to a binary value:

$$[P] \;=\; \begin{cases} 1 & \text{if } P \text{ is true,} \\ 0 & \text{otherwise.} \end{cases}$$

For example, the number of variables taking a specific value is

$$\sum_{i \in I} [\, x_i = v \,],$$

and the condition "at most k of the predicates $P_1, \ldots, P_n$ hold" can be written as $\sum_{i=1}^{n} [P_i] \le k$.

2.5.3 Declaring functions

A function is declared by specifying its *signature*:

$$f : A \to B,$$

which defines the function name f, its domain A, and its codomain B. Optionally, a definition follows:

$$f(a) := \ldots.$$

The domain and codomain may be Cartesian products, as in $d : C \times F \to \mathbb{R}_{\geq 0}$. This notation covers parameters (data expressed as functions rather than indexed symbols), derived expressions (e.g., $\text{slack}(x) := C - \sum_i w_i x_i$), and predicates (with codomain $\{0, 1\}$ or $\{\text{true}, \text{false}\}$).

Note

An optimization problem can itself be viewed as a function: parameters map to an optimal value. For the 0/1 knapsack,

$$\begin{aligned} \text{knapsack} &: \mathbb{R}^n_{\geq 0} \times \mathbb{R}^n_{\geq 0} \times \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}, \\ \text{knapsack}(c, w, C) &= \max_{x \in \{0,1\}^n} \Big\{ \textstyle\sum_i c_i x_i \;\Big|\; \sum_i w_i x_i \leq C \Big\}. \end{aligned}$$

2.6 Objectives

An optimization problem selects x from a feasible region $\mathcal{F}$ to minimize a scalar objective $f : D \to \mathbb{R}$:

$$\min_{x \in \mathcal{F}} f(x).$$

Maximization replaces min with max, or equivalently minimizes $-f$. The minimum attained is the *optimal value*,

$$f^\star = \min_{x \in \mathcal{F}} f(x),$$

and the set of *optimal solutions* is

$$\arg\min_{x \in \mathcal{F}} f(x) \;=\; \{\, x \in \mathcal{F} \mid f(x) = f^\star \,\}.$$

We write $x^\star \in \arg\min$ rather than $x^\star = \arg\min$, because the optimum need not be unique; many problems admit multiple optimal solutions, sometimes forming a continuum.

When multiple criteria are considered, we obtain a vector of objective values $(f_1(x), \ldots, f_m(x))$. Such vectors are not minimized by the usual scalar order; one must choose a notion of preference, such as Pareto dominance, a weighted sum, a minimax objective, or a lexicographic order. The choice is a modeling decision and is not pursued here.

2.7 Constraints

A constraint is a *predicate*—a statement that evaluates to true or false for a given assignment of values. While expressions answer "what is the value?", constraints answer "is the condition satisfied?".

The most common predicates in optimization are linear (in)equalities:

$$\sum_{j \in J} a_{ij} x_j \;\leq\; b_i \qquad \forall i \in I.$$

The quantifier $\forall i \in I$ is essential: this defines a family of $|I|$ constraints, not a single one. For example, if $I = \{1, 2, 3\}$, the family expands to

$$\sum_j a_{1j} x_j \leq b_1, \qquad \sum_j a_{2j} x_j \leq b_2, \qquad \sum_j a_{3j} x_j \leq b_3.$$

Equalities and reverse inequalities follow the same pattern.

Linearity is convenient but not required. In general, a constraint has the form

$$P_k(x) \qquad \forall k \in K,$$

where each P_k is a predicate on the variable vector—for example, a linear inequality, an *AllDifferent* constraint, a *NoOverlap* constraint, or another relation.

2.8 Logic and Quantifiers

Many decisions are yes/no decisions: *open this station*, *take this course*, *make this assignment*. A 0/1 variable is enough to encode them. Models built from such Boolean variables use the standard logical connectives

$$\wedge, \quad \vee, \quad \neg, \quad \Rightarrow, \quad \Leftrightarrow,$$

possibly quantified over index sets with $\forall$ and $\exists$. Two common large-operator forms are

$$\bigwedge_{k \in K} \phi_k, \qquad \bigvee_{k \in K} \phi_k.$$

They denote the conjunction and disjunction of an indexed family of Boolean expressions ϕ_k. The first is equivalent to the constraint family $\phi_k; \forall k \in K$; the second means that at least one ϕ_k must hold, and therefore cannot be replaced by a family of separate constraints.

Example. After an office prank, three suspects remain: Anna, Ben, and Cara. Let $x_a, x_b, x_c \in \{0, 1\}$ encode whether each person is guilty. Witnesses provide four facts. Anna would only act with Ben, so $x_a \Rightarrow x_b$. Ben and Cara have never gotten along, so $\neg(x_b \wedge x_c)$. Cara would only join if Anna did, so $x_c \Rightarrow x_a$. Finally, someone did it, so $x_a \vee x_b \vee x_c$. Together, the facts give

$$(x_a \Rightarrow x_b) \;\wedge\; \neg(x_b \wedge x_c) \;\wedge\; (x_c \Rightarrow x_a) \;\wedge\; (x_a \vee x_b \vee x_c).$$

Chaining the implications gives $x_c \Rightarrow x_a \Rightarrow x_b$. But x_b implies $\neg x_c$, because Ben and Cara cannot both be guilty. Hence $x_c \Rightarrow \neg x_c$, so $x_c = 0$. With Cara cleared, the condition that someone

is guilty reduces to $x_a \vee x_b$. Together with $x_a \Rightarrow x_b$, this forces $x_b = 1$, regardless of Anna's role. The clues implicate Ben, clear Cara, and leave Anna's status undetermined.

The connectives are not limited to Boolean variables; they also compose predicates over richer variables. Suppose two jobs share a machine. With start times $s_1, s_2 \in \mathbb{N}_0$ and durations d_1, d_2, the jobs must run one after the other:

$$s_1 + d_1 \leq s_2 \quad \vee \quad s_2 + d_2 \leq s_1.$$

This is a disjunction of two linear inequalities over integer variables. The same logical connectives apply to inequalities, equalities, AllDifferent$(y_1, \ldots, y_k)$, NoOverlap constraints, or any other relation. A predicate, like a function, can be named for reuse:

$$P(x) \;:=\; \sum_{i \in I} w_i x_i \leq C.$$

Once named, it fits into the same logical syntax: $P_1(x) \wedge P_2(x)$, $P(x) \Rightarrow Q(x)$, or $\bigvee_{k \in K} P_k(x)$.

The two layers—Boolean variables and predicates over richer variables—meet in *reification*. Reification ties a Boolean indicator $z \in \{0, 1\}$ to the truth value of a predicate:

$$z \;\Leftrightarrow\; (x \geq 5).$$

Once reified, z can appear in logical connectives like any other Boolean variable. The special case $z \Rightarrow a^\top x \leq b$ is known as an *indicator constraint* in MIP terminology. How a solver encodes such patterns is separate from how the model states them.

2.9 A Standard Model Template

Many optimization models share a common skeleton. In its simplest form, it is

$$\begin{aligned} \min_{x} \quad & f(x) \\ \text{s.t.} \quad & P_k(x) \qquad \forall k \in K, \\ & x \in D. \end{aligned}$$

The objective f, predicates P_k, and domain D are specified by the application. The corresponding feasible region is

$$\mathcal{F} = \{\, x \in D \mid P_k(x)\ \forall k \in K \,\},$$

with optimal value and optimal solution written as

$$f^\star = \min_{x \in \mathcal{F}} f(x), \qquad x^\star \in \arg\min_{x \in \mathcal{F}} f(x).$$

Many formulations you encounter will fit this template, often with only minor notational variations.

Chapter 3

Drafting the Model

"All models are wrong, but some are useful." — George E. P. Box

There are two fundamental challenges to overcome before implementing an optimization model in a solver. The first is to express the problem in a precise form. The second is to translate that form into something the interface of your chosen solver can understand. Further challenges will follow later, especially making the model run fast, but this chapter focuses only on the first two.

The previous chapter gave us the notation needed to start sketching a model. In this chapter, we look at a structured way to do so. The structure is not linear: for a complicated problem, you usually identify the core, formulate it, get it into a runnable form, and then extend it with the remaining rules and objectives.

The five modeling ingredients in this chapter — entities, parameters, variables, constraints, and objective — are therefore not a checklist that you complete once. They are a recurring guide. New constraints may require new variables; a new objective term may change which parameters matter; a newly discovered entity may force you to restructure earlier parts of the formulation. With experience, the iterations tend to become fewer and smaller.

3.1 The Five Components

A complete optimization model has five components that provide structure for thinking and writing:

1. **Entities** — what objects does the problem talk about? They become *sets*.
2. **Parameters** — what data is known up front? They become *constants indexed by the sets*.
3. **Variables** — what is the solver allowed to choose? They become *unknowns with a domain*.
4. **Constraints** — what makes a choice valid? They become *relations on variables*.
5. **Objective** — what makes one valid choice better than another? It becomes *one expression with sense* `min` *or* `max`; multi-objective optimization is a topic of its own.

The order is a guideline. Entities and parameters define the language; variables depend on that language; constraints relate the variables; the objective evaluates valid choices.

For the Knapsack problem from the previous chapter, the five components read:

1. **Entities.** The items, I.

2. **Parameters.** A value $c_i \in \mathbb{R}_{\geq 0}$ and a weight $w_i \in \mathbb{R}_{\geq 0}$ for every $i \in I$, plus a knapsack capacity $C \in \mathbb{R}_{\geq 0}$.

3. **Variables.** One binary variable per item, $x_i \in \{0, 1\}$ for $i \in I$, with $x_i = 1$ meaning *take item* i.

4. **Constraints.** The total weight must not exceed capacity: $\sum_{i \in I} w_i x_i \leq C$.

5. **Objective.** Maximize total value: $\max \sum_{i \in I} c_i x_i$.

Now suppose we want to integrate the scenario-based worst-case objective from the previous chapter. We have to change a few components, but the model still looks similar:

1. **Entities.** Items I and scenarios S.

2. **Parameters.** A per-scenario value $c_i^s \in \mathbb{R}_{\geq 0}$ for every $i \in I$ and $s \in S$; weights w_i and capacity C as before.

3. **Variables.** Unchanged: $x_i \in \{0, 1\}$ for $i \in I$.

4. **Constraints.** Unchanged: $\sum_{i \in I} w_i x_i \leq C$.

5. **Objective.** Maximize the worst-case value: $\max \min_{s \in S} \sum_{i \in I} c_i^s x_i$.

This is a precise paper formulation, but it is not yet in a form suitable for most solvers. The objective contains a min over scenarios inside the maximization, and most solvers will not accept that nested expression as written. The standard remedy is an auxiliary-variable reformulation: introduce a variable z for the worst-case value and bound it from above by every scenario's value.

The solver-facing version is:

1. **Entities.** Items I and scenarios S.

2. **Parameters.** c_i^s, w_i, and C.

3. **Variables.** $x_i \in \{0, 1\}$ for $i \in I$, and an auxiliary variable z representing the worst-case value.

4. **Constraints.** The capacity constraint

$$\sum_{i \in I} w_i x_i \leq C,$$

and the scenario-value constraints

$$z \leq \sum_{i \in I} c_i^s x_i \qquad \forall s \in S.$$

5. **Objective.** Maximize z.

Maximizing z subject to one inequality per scenario forces z up to the smallest scenario value, recovering the original $\min_{s \in S}$ exactly.

3.2 A Worked Example: Hospital Ward Rostering

Consider the following problem. A small hospital ward needs to schedule its nursing staff for the coming week. The ward has a fixed list of shifts to fill. Each shift has a date, a required skill — say, *ICU*, *pediatrics*, or *general care* — and a headcount specifying how many nurses must be on duty. The ward employs a roster of nurses, each of whom holds a personal set of skills and is available on certain days but not others. Assigning a particular nurse to a particular shift incurs a cost that captures wage rates, shift differentials, and union overtime rules.

The ward manager wants to staff every shift, respect availability and skills, and spend as little as possible.

We will now extract the five components. Unlike in the knapsack example, they are no longer almost forced by the problem statement. We have several kinds of objects, several indexed data tables, and constraints that refer to different combinations of workers, shifts, days, and skills. The point of the five-component structure is to keep that complexity visible and organized.

3.2.1 Step 1 — Entities become sets

Read the problem statement and underline every noun that names a class of objects. *Nurses, shifts, days, skills.* Each class becomes a set.

Symbol	Description
W	Workers (nurses) — the staff to be scheduled.
S	Shifts — the work blocks to be filled this week.
T	Days — the calendar dimension, $T = \{1, \dots, 7\}$.
K	Skills — the qualifications the ward tracks.

A set is the *type* of an index. Once we have written $w \in W$ and $s \in S$, we can index any quantity in the model by (w, s) — variables, parameters, constraints, neighborhoods.

3.2.2 Step 2 — Parameters are the input data

Anything known before we solve is a parameter. Parameters are often indexed by the sets from step 1, and each one should carry a domain — the values it may take — together with an informal description that fixes what the symbol means. Some parameters are raw input data; others are derived from that input to make the model easier to write.

Symbol	Indices	Domain	Description
$c_{w,s}$	$w \in W,\ s \in S$	$\mathbb{R}_{\geq 0}$	Cost of assigning worker w to shift s.
$a_{w,s}$	$w \in W,\ s \in S$	$\{0, 1\}$	Availability flag: 1 iff w is available for s.
r_s	$s \in S$	K	Skill required by shift s.
h_w	$w \in W$	2^K	Set of skills held by worker w.
n_s	$s \in S$	$\mathbb{N}_0$	Number of workers needed on shift s.
S_t	$t \in T$	2^S	Subset of shifts that fall on day t (derived from shift dates).

The domains are explicit. $c_{w,s}$ is a non-negative real, $a_{w,s}$ is binary, n_s is a non-negative integer, and h_w is a subset of K (so its domain is the power set 2^K). Each shift requires one skill, while each worker may hold multiple skills — a detail that matters once we write the skill-match constraint.

The last row illustrates that a parameter need not be raw input. S_t is derived from the dates of the shifts: for each day t, it collects exactly those shifts that occur on that day. Including such derived parameters in the table is useful because it gives later constraints a clean vocabulary. Instead of repeatedly writing "all shifts that fall on day t," we can simply write S_t.

3.2.3 Step 3 — Variables are the unknowns

This is the step where the problem stops being only a description and becomes a model. Each decision unit becomes one variable, declared with three things: a name, an index range, and a domain. What counts as a single decision unit is itself a modeling choice: the same problem can often be modeled at different granularities, with real consequences for solver performance.

For the rostering problem, the natural first decision unit is *does worker w take shift s?* — one binary choice per worker–shift pair.

Symbol	Indices	Domain	Description
$x_{w,s}$	$w \in W,\ s \in S$	$\{0,1\}$	`1` iff worker w is assigned to shift s, else `0`.

That single line declares $|W| \cdot |S|$ variables — one per cell of the worker–shift matrix. Many of those combinations may be trivially infeasible: the worker may be unavailable, or may lack the skill required by the shift. In the simple formulation, we still declare the full matrix and let constraints rule out the invalid combinations.

One could prune the index set up front and create variables only for worker–shift pairs that are possible in principle. That is often a useful implementation choice, but it is not needed for the first paper model. At this stage, the full matrix is easier to read: every potential assignment has a variable, and every rule appears explicitly as a constraint. If an implementation detail is important enough to remember, record it as a note rather than complicating the first formulation.

The decision here is *take or leave*, so the natural variable is binary. Other problems suggest other shapes:

Decision phrasing	Variable shape
"Take it or not?"	$x_i \in \{0,1\}$ — binary
"How many?"	$x_i \in \mathbb{N}_0$ — non-negative integer
"When does it start?"	$\mathrm{st}_i \in [0, H]$ — time variable, bounded by horizon H
"Which slot does it go into?"	either $x_i \in \{1, \ldots, K\}$ *or* $y_{i,k} \in \{0,1\}$ — index *vs.* one-hot
"Which edge is used?"	$x_{ij} \in \{0,1\}\ \forall (i,j) \in E$ — one Boolean per edge

The fourth row deserves attention because it is a common modeling decision: when the answer is "which of K options," do you use a single integer variable taking values in $\{1, \ldots, K\}$, or K Boolean variables that sum to 1? Both can be correct. They differ in the kinds of constraints they make easy to write and in the structure a solver can exploit.

Note

Real workforce-scheduling systems often use more aggregate variables than the simple worker–shift assignment variable above. In a column-generation model, for example, a variable may represent an entire shift pattern for one worker over the planning horizon. Since the number of possible patterns is usually enormous, the model is solved iteratively: start with a manageable subset of patterns, solve the restricted model, then generate additional useful patterns through a separate pricing problem. That is a powerful technique, but it is a different modeling level from the first paper formulation developed here.

3.2.4 Step 4 — Constraints encode validity

Constraints are usually the most complex part of the model. For simple textbook examples, it may be enough to translate each "must" sentence directly into one formula. But even the scenario-based knapsack example showed that this is not always the whole story: sometimes a clear paper statement needs auxiliary variables and additional constraints before it becomes solver-friendly. In richer applications, one must also distinguish between hard constraints, which define feasibility, and soft constraints, which may be violated at a penalty. The present example is cleanly specified, so we can ignore soft constraints for now.

We now identify the constraint families and formulate them one by one. For hard constraints, each new rule narrows the feasible region: it removes assignments that are not allowed, but it does not make previously invalid assignments valid. This makes constraints relatively modular. Still, a single rule may require several formulas, and part of a rule may already be enforced by earlier constraints or by variable domains. In this example, however, the mapping is pleasantly direct: four business rules become four constraint families.

Coverage. Every shift must be staffed with exactly the headcount it requires.

$$\sum_{w \in W} x_{w,s} \;=\; n_s \qquad \forall s \in S.$$

This is one equation per shift. The sum on the left counts the number of workers assigned to s, and the right-hand side is the demanded headcount. The $\forall s \in S$ is doing real work: it makes this not a single equation but a family of $|S|$ equations.

Availability. A worker may not be assigned to a shift they are not available for.

$$x_{w,s} \;\leq\; a_{w,s} \qquad \forall w \in W,\; s \in S.$$

If $a_{w,s} = 0$, the inequality forces $x_{w,s} = 0$. If $a_{w,s} = 1$, the inequality becomes $x_{w,s} \leq 1$, which is already implied by the variable's domain, so the constraint is silent.

Skill match. If a worker is assigned to a shift, the worker must hold the skill the shift requires.

$$x_{w,s} = 1 \;\Rightarrow\; r_s \in h_w \qquad \forall w \in W,\; s \in S.$$

This is a clear paper-level statement, but it is not the most useful solver-facing form because the right-hand side is a fact about the input data, not a decision. The direct rewrite is:

$$x_{w,s} \;\le\; [r_s \in h_w] \qquad \forall w \in W,\; s \in S,$$

where the indicator on the right is `1` if the skill requirement is met and `0` otherwise. The compatibility values are computed from the data before the solver is called; incompatible pairs are forced to zero, compatible pairs are left to the other constraints.

No double-booking. No worker may be assigned to more than one shift on the same day.

$$\sum_{s \in S_t} x_{w,s} \;\le\; 1 \qquad \forall w \in W,\; t \in T.$$

For every worker–day combination, the sum runs only over the shifts that fall on that day and requires that no more than one of them be assigned to this worker. This is a typical use of a derived parameter (S_t) to keep the constraint readable.

3.2.5 Step 5 — The objective picks among valid choices

We have specified what counts as a valid roster. Now we say which valid roster we prefer.

Total cost. Minimize the total cost of the assignments we make.

$$\min \sum_{w \in W} \sum_{s \in S} c_{w,s}\, x_{w,s}.$$

The sum runs over all worker–shift pairs, but only the pairs with $x_{w,s} = 1$ contribute.

Two notes on objectives. First, the sense — `min` or `max` — is part of the specification; "optimize f" is incomplete. Second, an objective is a single expression. If the problem talks about several goals at once — cost *and* overtime *and* fairness — you must combine them yourself, by weighted sum or lexicographic ordering. There is no implicit averaging. And a model can have no objective at all: pure feasibility ("is *any* valid roster possible?") is a legitimate question, and forcing an artificial objective onto it is a mistake.

Chapter 4

Verifying Candidate Solutions

> *"Show me your flowchart and conceal your tables, and I shall continue to be mystified. Show me your tables, and I won't usually need your flowchart; it'll be obvious."* — Fred Brooks, *The Mythical Man-Month*

In this chapter, we discuss a concept that is orthogonal to the basic modeling process. A *verifier* is a piece of code that takes an instance and a candidate solution, checks whether the solution is feasible, and computes its objective value.

Writing such a verifier still requires the same five ingredients from optimization modeling: entities, parameters, variables, constraints, and objective. In the verifier, however, the variables appear as concrete values inside a candidate `Solution`, not as symbolic objects controlled by a solver. The difference is the medium. Instead of expressing the problem as mathematical notation or as solver constraints, we express it as ordinary program code. For many software engineers, this feels less abstract and more natural.

A verifier can be developed independently of the solver model, but the two must agree on the problem definition and on the representation of instances and solutions. Once available, the verifier can serve several roles. It can help you test hand-written examples, evaluate simple heuristics, and get a feel for the problem before the solver model is mature. During solver development, it acts as a guardrail: solver-produced solutions can be checked against it, random instances can be smoke-tested, and performance-driven refactors can be validated against the same problem definition.

In that sense, the verifier can be viewed as an executable contract for the problem definition. It does not say how to find a good solution. It says how to recognize a valid one, and how to score it once found.

4.1 What a Verifier Is

A verifier checks candidate solutions. Given an instance and a proposed solution, it answers two questions:

1. Is the solution feasible?
2. What is its objective value?

It does not search for solutions. It does not call a solver. It is not an algebraic modeling language and not a solver model. A solver model describes a search space to an optimization engine; a verifier checks one fully specified candidate after the decisions have already been made. Solvers produce solutions; verifiers consume them.

This difference makes a verifier much easier to write for many rules. In a solver model, a rule must be expressed as a constraint over decision variables, in the vocabulary supported by the solver. In a verifier, the candidate assignment is already known, so the rule can be checked with ordinary program code: branches, lookups, loops, early returns, and descriptive error messages.

The **simple form** keeps all checks in one evaluator function. It returns the objective value if the candidate is feasible and raises an exception naming the first violated rule otherwise. This is short, readable, and sufficient for many small hard-constraint models.

The **richer form** decomposes the checks into separate rule modules and replaces yes/no answers with quantitative metrics: *three minutes of overlap*, *two uncovered shifts*, *one missing skill*. This becomes useful once rules are soft, violations need to be reported in bulk, or several solution methods need to be compared against the same standard.

4.2 The Simple Form: `Instance`, `Solution`, `evaluate`

A simple verifier consists of three pieces. The first two are plain data containers and can be reused elsewhere in the codebase:

1. `Instance` — a frozen dataclass holding the input data.
2. `Solution` — a frozen dataclass holding one completed candidate solution.
3. `evaluate(instance, solution)` — a pure function that checks feasibility and, if the solution is feasible, returns its objective value. If a constraint family fails, it raises an exception that identifies the failing rule.

This is the minimal pattern: two dataclasses and one function. We illustrate it with a rostering verifier.

4.2.1 Type Aliases for the Index Sets

Python is dynamically typed, but type annotations are useful as documentation and for static analysis and editor support. Here, we use simple aliases for the index sets from the paper model:

```
type Worker = str  # Identifier for a worker who can take shifts
type Shift  = str  # Identifier for a shift that needs to be staffed
type Day    = int  # Integer day index, e.g., 0 for Monday, 1 for Tuesday
type Skill  = str  # Skill that workers may hold and shifts may require
```

Workers, shifts, days, and skills are still strings or integers at runtime. The aliases do not add type safety by themselves, but they improve the shape of the code: `dict[tuple[Worker, Shift], int]` says more than `dict[tuple[str, str], int]`.

4.2.2 The `Instance` and `Solution` dataclasses

```
from dataclasses import dataclass

@dataclass(frozen=True)
class Instance:
    workers: frozenset[Worker]
    shifts:  frozenset[Shift]
    days:    frozenset[Day]
```

```
    skills:  frozenset[Skill]

    cost:           dict[tuple[Worker, Shift], int]
    available:      dict[tuple[Worker, Shift], bool]
    required_skill: dict[Shift,  Skill]
    holds_skills:   dict[Worker, frozenset[Skill]]
    headcount:      dict[Shift,  int]
    shifts_on_day:  dict[Day,    frozenset[Shift]]

@dataclass(frozen=True)
class Solution:
    assigned: frozenset[tuple[Worker, Shift]]
```

The `Instance` carries the sets and parameters from the math model; the `Solution` carries the values of the decision variables. The solution uses the *sparse* representation — we record only the pairs where $x_{w,s} = 1$, not the full zero-one matrix — which scales with what was assigned rather than with the size of the index space.

Note

For teaching purposes we use `dataclass`, but at the system boundary — wherever an instance is built from a CSV, a database, or a JSON payload — a properly validated data schema (typically Pydantic) earns its keep. Inconsistent input data is extremely common in practice, and a surprising share of "the model is wrong" debugging sessions turn out to be data problems wearing a model's clothes. The topic deserves its own chapter and we will not develop it here; what matters for the verifier is only that the object handed to `evaluate` has the right attributes.

4.2.3 The Evaluator

The evaluator is where each constraint family receives its explicit check, and where the objective is computed only after all checks have passed.

```
def evaluate(inst: Instance, sol: Solution) -> int:
    """
    Return the objective value, or raise ValueError naming the failing rule.
    """

    # (cover) every shift gets exactly the headcount it requires
    for s in inst.shifts:
        got = sum((w, s) in sol.assigned for w in inst.workers)
        if got != inst.headcount[s]:
            raise ValueError(
                f"(cover) shift {s!r}: need {inst.headcount[s]}, got {got}"
            )

    # (avail) no assignment to an unavailable (w, s) pair
    for (w, s) in sol.assigned:
        if not inst.available[w, s]:
            raise ValueError(
                f"(avail) worker {w!r} not available for shift {s!r}"
            )

    # (skill) every assigned worker holds the shift's required skill
```

```
    for (w, s) in sol.assigned:
        required = inst.required_skill[s]
        if required not in inst.holds_skills[w]:
            raise ValueError(
                f"(skill) worker {w!r} lacks {required!r} required by shift {s!r}"
            )

    # (no-double) no worker is on two shifts on the same day
    for w in inst.workers:
        for t, shifts_today in inst.shifts_on_day.items():
            taken_today = sum((w, s) in sol.assigned for s in shifts_today)
            if taken_today > 1:
                raise ValueError(
                    f"(no-double) worker {w!r} on {taken_today} shifts on day {t}"
                )

    # objective: total cost over assigned pairs
    return sum(inst.cost[w, s] for (w, s) in sol.assigned)
```

The exception messages name witnesses, not just rules. For example, `(skill) worker 'Alice' lacks 'ICU' required by shift 'S5'` identifies the violated rule together with the worker and shift involved. A domain expert can read this message and confirm or dispute it without understanding the solver.

Tip

Quantitative verifier output is also useful for AI-assisted development. A coding agent can use violations, costs, and witness messages as a concrete feedback signal when implementing a heuristic, repairing a solution, or debugging a solver interface. The verifier does not make the agent reliable by itself, but it gives the agent something precise to test against.

4.3 A Richer Form: Quantitative Metrics and Modularized Rules

On larger problems, two refactors of the simple form repeatedly pay off. The exact shape of a richer verifier is project-specific; what follows is the direction, not a blueprint.

4.3.1 Quantitative Metrics Instead of Raise/Continue

The simple `evaluate` function answers a binary question per rule: did the rule pass, or did it reject the solution? That throws away information that becomes essential once a project has *soft* rules — rules that may be violated at a penalty rather than forbidden outright — or any kind of repair-style search. Three minutes of overlap and three hours of overlap are very different solutions, but the simple form cannot distinguish them.

Replacing the raise/continue verdict with a `Metric` object preserves that information:

```
@dataclass(frozen=True)
class Metric:
    rule: str
    violation: float            # 0.0 = rule satisfied; otherwise rule-native units
    cost: float                 # contribution to the objective
    messages: tuple[str, ...]   # one per witness, e.g. "shift 'S3': need 3, got 2"
```

The `violation` field measures how far the solution is from satisfying the rule, in whatever unit is natural for that rule: uncovered shifts, minutes of overlap, or hours over the weekly ceiling. The `cost` field measures the rule's contribution to the objective.

For hard rules, feasibility is decided by `violation`; a solution is feasible exactly when total hard-rule violation is zero. For soft rules, the violation magnitude often becomes part of the cost instead of making the solution infeasible.

The `messages` field carries the witness-naming we already used in the simple form's exception strings — for example, `(cover) shift 'S3': need 3, got 2` — but now per witness rather than only for the first failure. A debugging session can therefore see every place a rule failed instead of only the point at which `evaluate` raised.

The gain is that solutions can now be compared by how badly they fail a rule, which is what makes repair, local search, and soft-objective optimization practical.

Tip

Given such feedback, coding agents can frequently implement or fix algorithms autonomously as it provides a clear signal of whether it is getting closer or not.

4.3.2 Splitting Rules into Modules

The other refactor is structural. The simple `evaluate` puts every rule in one function. With four rules, that is fine; with twenty, it becomes hard to navigate, hard to test in isolation, and hard to give per-rule precomputation a natural place to live.

Splitting the rules across modules — typically as classes, grouped however makes sense for the project, such as one file per rule or one file per related family of rules — addresses all three issues. The example below shows the pattern with a single rule per class:

```python
# verifier/rules/cover.py

class CoverRule:
    """Each shift must be staffed with exactly its required headcount."""

    def __init__(self, instance: Instance):
        self.instance = instance
        # Precompute whatever this rule needs per instance.

    def evaluate(self, solution: Solution) -> Metric:
        violation = 0.0
        messages: list[str] = []

        for s in self.instance.shifts:
            got = sum((w, s) in solution.assigned for w in self.instance.workers)
            need = self.instance.headcount[s]

            if got != need:
                violation += abs(got - need)
                messages.append(f"shift {s!r}: need {need}, got {got}")

        return Metric(
            rule="cover",
```

```
            violation=violation,
            cost=0.0,
            messages=tuple(messages),
        )
```

Each rule gets a constructor, which runs once per instance, and an `evaluate` method, which the top-level verifier may call many times while checking candidates from a solver, heuristic, repair procedure, or test suite. Anything that depends only on the instance can be precomputed in the constructor. How the per-rule metrics are aggregated into a final report is a project-specific decision and is not worth prescribing here.

Chapter 5

First Contact with CP-SAT

"Make it work. Make it right. Make it fast." — Kent Beck

The solver this manuscript commits to is **CP-SAT**, the constraint-programming solver that ships inside Google's OR-Tools. It is open source, distributed under the Apache 2.0 license, and free for any use. Installation is a single `pip` command, runs on Linux, macOS, and Windows, and requires no license server or external dependencies.

The primary reason for this choice is that CP-SAT occupies a useful middle ground for learning. Its constraint vocabulary is rich — global constraints like `circuit`, `no_overlap`, and `all_different` let you express combinatorial structures directly, without the forced linearization a pure MIP solver would require. But CP-SAT's Python interface is still a *solver-specific API*, not an algebraic modeling language that targets multiple backends behind an abstraction layer. You are building a model that a specific solver will run, and you see how each modeling decision maps to what the solver actually works with. That directness is valuable when learning: it keeps the connection between the model and the solver visible.

CP-SAT has won the MiniZinc Challenge — the standard annual benchmark in the constraint-programming community — every year since 2018. For combinatorial problems with discrete decisions, it is one of the strongest tools available.

One property worth stating up front: CP-SAT is **integer-only**. Booleans, finite enumerations, integer-valued amounts, and integer-modeled time variables are all native. Continuous quantities are not supported and must be either integerized (multiply real prices in dollars by 100 and work in cents) or pushed to a different solver (linear programming, mixed-integer programming with continuous variables). For genuinely combinatorial problems the integer restriction is rarely felt; for problems with a continuous core, CP-SAT is the wrong tool.

5.1 What Is Inside the Box

A reader who is new to constraint solvers might reasonably expect CP-SAT to be one algorithm. It is not. CP-SAT is a *portfolio*: several complementary techniques wired together, run in parallel, and sharing learned information with each other.

A short tour of what is inside:

- A **SAT** core handles Boolean reasoning, using the conflict-driven clause-learning algorithm that powers modern SAT solvers. This is what makes CP-SAT extraordinarily good at logical constraints.

- A **CP** layer adds the global constraints — `all_different`, `circuit`, `cumulative`, and others — with dedicated propagators that reason about whole structures at once rather than constraint by constraint.
- A **linear-programming relaxation** runs alongside to provide tight dual bounds. This is what allows CP-SAT to make serious inroads into territory traditionally dominated by MIP solvers.
- **Large Neighborhood Search** (LNS) takes a current solution, freezes most of it, and re-optimizes the rest — a way of escaping local minima that hand-rolled procedures rarely match.
- An aggressive **presolve** simplifies the model before search begins, often removing variables and constraints the modeler did not realize were redundant.
- Several **workers** run different combinations of these techniques in parallel and share learned clauses, so a discovery in one worker can prune the search of another.

The crucial implication for the modeler: *you typically do not pick the strategy explicitly.* You hand the solver a model and call `solve`; the portfolio races itself for the best answer. Advanced usage does involve steering — branching strategies, LNS parameters, worker counts, time limits, and so on — but most of the leverage on real problems comes from the model itself rather than from solver tuning; for the cases where tuning matters, later chapters discuss it. For now, "build a model, call `solve`, get an answer" is a complete mental model.

5.2 Where CP-SAT Fits — and Where It Does Not

CP-SAT is opinionated. It excels in a particular shape of problem and is the wrong tool in others. Knowing the difference is part of using it well.

Where CP-SAT fits. The sweet spot is problems whose decisions are dominantly discrete and whose constraints are richly logical:

- **Scheduling and rostering** — assigning tasks to machines, workers to shifts, courses to rooms.
- **Routing and tour problems** — vehicle routing on small to medium graphs, sequencing.
- **Packing, assignment, and matching** — bin packing, generalized assignment, stable matching with side constraints.
- **Discrete planning** with rich logical rules, especially when the rules are messy in ways that pure linear formulations do not handle gracefully.
- **Feasibility checking** — *is there any valid configuration at all?* CP-SAT is exceptionally good at this; problems with hundreds of logical rules that look intractable are often solved or proven infeasible in seconds.

Where to reach for something else. CP-SAT is not the right tool when:

- Continuous variables dominate the problem. Reach for a linear-programming or nonlinear-programming solver. (Cents-and-dollars integerization works for shallow continuous components; deep continuous structure does not survive it.)
- The problem is purely linear, the LP relaxation is strong, and the scale is large. A dedicated mixed-integer programmer such as Gurobi, CPLEX, or HiGHS often wins on the standard benchmarks. CP-SAT has an LP layer of its own, but it is not optimized for the regimes the dedicated MIP solvers are.
- The problem is the symmetric Traveling Salesman Problem at very large scale (tens of thousands of cities). Specialist solvers like Concorde and LKH remain ahead.
- The objective is differentiable and you want gradients. JAX, PyTorch, and the gradient-based optimization stack are an entirely different shape of tool.

CP-SAT thrives on combinatorial structure with rich logical constraints; mixed-integer programming thrives on polyhedral strength and tight linear relaxations. Many real problems contain both flavors, in which case the right tool comes down to which flavor dominates.

The rest of this manuscript works inside CP-SAT's sweet spot.

5.3 Setup

```
pip install ortools
```

That is the entire installation. The package ships precompiled wheels for the major platforms — Linux, macOS, Windows, on x86-64 and ARM — so there is no C++ build step on a fresh machine. Python 3.9 and newer are supported.

Two imports are used in essentially every CP-SAT program:

```
from ortools.sat.python import cp_model
```

`cp_model.CpModel` is the model *builder* — the object you populate with variables and constraints. `cp_model.CpSolver` is the *solver wrapper* — the object that takes a built model, runs the search, and exposes the answer. Every example in the rest of the manuscript starts with this import.

A note on hardware: a modern laptop is sufficient to get started. CP-SAT does not use a GPU. The two resources that matter are cores (CP-SAT runs multiple workers in parallel) and memory (presolve, LP relaxation, and learned clauses all need RAM). Four cores and 16 GB of RAM is a reasonable starting point for learning, but serious projects usually benefit from more — both more workers and more memory headroom. Smaller machines are still usable for modest instances.

5.4 A Complete Example: Facility Location

We walk through the **uncapacitated facility location problem** (FLP) end to end, with the full five-step recipe from Chapter 3 visible in the code. The FLP is a classical assignment problem with an objective: pick which warehouses to open and which warehouse serves each customer.

The problem statement:

A delivery company is planning to open new warehouses to serve its customers. There is a set F of potential warehouse locations and a set C of customers who need to receive their orders. Opening a warehouse at location $i \in F$ comes with a fixed cost f_i. Shipping goods from warehouse i to customer $j \in C$ costs $c_{i,j}$. Every customer must be served by exactly one warehouse, and a warehouse can only ship goods if it is actually opened. The company needs to decide which warehouse locations to open and which warehouse will serve each customer so that the total cost — opening costs plus shipping costs — is as small as possible.

We apply the five-step recipe from Chapter 3 to this statement, working through the components in order.

Entities. The nouns in the problem that name classes of objects are *warehouses* (or *locations*) and *customers*. Each becomes a set.

Symbol	Description
F	Potential warehouse locations the company is considering.
C	Customers who must be served.

Parameters. The data known before any decision is made consists of two cost families.

Symbol	Indices	Domain	Description
f_i	$i \in F$	$\mathbb{N}_0$	Fixed cost of opening a warehouse at location i.
$c_{i,j}$	$i \in F,\ j \in C$	$\mathbb{N}_0$	Cost of shipping from warehouse i to customer j *if* that assignment is made.

Variables. There are two atomic decisions to make — whether to open each warehouse, and whether each warehouse serves each customer — giving one variable family per decision.

Symbol	Indices	Domain	Description
y_i	$i \in F$	$\{0,1\}$	1 iff the warehouse at location i is opened.
$x_{i,j}$	$i \in F,\ j \in C$	$\{0,1\}$	1 iff customer j is served by warehouse i.

Constraints. The model has two families.

Assignment. Every customer is served by exactly one warehouse.

$$\sum_{i \in F} x_{i,j} \;=\; 1 \qquad \forall j \in C.$$

Linking. A warehouse may serve a customer only if the warehouse is open.

$$x_{i,j} \;\le\; y_i \qquad \forall i \in F,\ j \in C.$$

If $y_i = 0$ (warehouse closed), the inequality forces $x_{i,j} = 0$ for every customer j — that warehouse cannot serve anyone. If $y_i = 1$ (warehouse open), the inequality is $x_{i,j} \le 1$, which the variable's domain already implies, so the constraint is silent. One line captures the implication "closed $\rightarrow$ cannot serve" with no implication arrow on paper.

Objective. Minimize total cost — the fixed cost of every warehouse opened, plus the shipping cost of every customer assignment made.

$$\min \quad \sum_{i \in F} f_i \, y_i \; + \; \sum_{i \in F} \sum_{j \in C} c_{i,j} \, x_{i,j}.$$

That is the full mathematical model: two sets, two parameter families, two variable families, two constraint families, one objective. The CP-SAT translation is line-for-line:

```python
from ortools.sat.python import cp_model

# 1. Data
F = [1, 2, 3]                                   # potential locations
C = [1, 2, 3]                                   # customers
f = {1: 100, 2: 200, 3: 150}                    # opening cost f[i]
c = {1: {1: 10, 2: 20, 3: 30},                  # shipping cost c[i][j]
     2: {1: 15, 2: 25, 3: 35},
     3: {1: 20, 2: 30, 3: 40}}

model = cp_model.CpModel()

# 2. Variables
y = {i: model.new_bool_var(f"y_{i}") for i in F}
x = {(i, j): model.new_bool_var(f"x_{i}_{j}") for i in F for j in C}

# 3. Constraints
# Exactly one warehouse per customer:
for j in C:
    model.add(sum(x[i, j] for i in F) == 1)
# Only open warehouses can serve:
for i in F:
    for j in C:
        model.add(x[i, j] <= y[i])

# 4. Objective
model.minimize(
    sum(f[i] * y[i] for i in F) +
    sum(c[i][j] * x[i, j] for i in F for j in C)
)

# 5. Solve
solver = cp_model.CpSolver()
status = solver.solve(model)

# 6. Read the answer
if status in (cp_model.OPTIMAL, cp_model.FEASIBLE):
    opened = [i for i in F if solver.value(y[i])]
    served_by = {
        j: next(i for i in F if solver.value(x[i, j])) for j in C
    }
    print("opened warehouses:", opened)
    print("customer assignment:", served_by)
    print("total cost:", solver.objective_value)
else:
    print("no feasible solution")
```

Variables. The code declares two families, mirroring the two variable rows in the math model. y is one Boolean per potential location — *is this warehouse opened?* — created by `model.new_bool_var(name)` and corresponding to $y_i \in \{0, 1\}$ on paper. The `name` is metadata for debugging output and does not affect the search. x is one Boolean per (location, customer) pair — *does this warehouse serve this customer?* — keyed by a tuple, corresponding to $x_{i,j} \in \{0, 1\}$. The dictionary keying is a Python convenience that lets us write `x[i, j]` later instead of indexing into a flattened list.

Each paper variable here lifts directly into a CP-SAT variable with the same domain and indexing. In larger models the correspondence is looser — some paper variables are dropped as redundant, others are split into auxiliary solver variables — but for this example the mapping is direct.

Constraints. The two families translate one-to-one into Python. The assignment constraint, *each customer gets exactly one warehouse*, is one linear equality per customer:

```
model.add(sum(x[i, j] for i in F) == 1)
```

Python's built-in `sum(...)` builds a CP-SAT linear expression by overloading addition on variables, and `model.add(... == 1)` registers the linear-equality constraint. The `==` is *building a constraint*, not evaluating to a Python bool. This operator-overloading trick is at the heart of how CP-SAT's Python API reads almost like the math: every arithmetic and comparison operator on variables yields a CP-SAT object that the model collects.

The linking constraint, *only open warehouses can serve*, is one inequality per (location, customer) pair:

```
model.add(x[i, j] <= y[i])
```

This is the line whose semantics we noted on paper: when $y_i = 0$ the right-hand side forces $x_{i,j} = 0$; when $y_i = 1$ the inequality is slack. The Python expression is exactly the math — CP-SAT accepts the linear inequality directly, with no extra machinery.

Objective. `model.minimize(expr)` registers `expr` as the objective and tells the solver to minimize it. The expression is the same arithmetic object the math model wrote: opening costs summed over locations, plus shipping costs summed over (location, customer) pairs. To maximize instead, the corresponding method is `model.maximize`.

Solve. `solver = cp_model.CpSolver()` instantiates the wrapper, and `status = solver.solve(model)` runs the search and returns a status code. For a first model the two outcomes to know are `OPTIMAL` (a solution was found and proven optimal) and `FEASIBLE` (a solution was found, but optimality was not proven — on hard problems this is a first-class outcome, often the answer you ship). The remaining codes (`INFEASIBLE`, `UNKNOWN`, `MODEL_INVALID`) and time-limit handling are covered in Chapter 8.

Read the answer. Three pieces of information are worth reading from the solver after a successful run.

`solver.value(var)` returns the assigned value of each variable as an integer — `0` or `1` for Booleans, the assigned integer for an integer variable. The companion `solver.boolean_value(literal)` returns the same information as a Python `bool` and is the cleaner choice when downstream code wants `True`/`False` directly. Reading the answer through the *solver* (not the *variable*) is important either way — the variable object is just a handle; the value lives in the solver's internal state after the search.

`solver.objective_value` returns the objective value of the solution: the optimum when status is `OPTIMAL`, the best-found value when status is `FEASIBLE` (where, again, optimality has not been proven). For models without an objective there is nothing to read here.

The two derived quantities, `opened` and `served_by`, are simple comprehensions over `solver.value(...)`. They are not part of the solver's output per se; they are how the program turns the variable values into a domain-readable form.

Run the program, and you get something like:

```
opened warehouses: [1]
customer assignment: {1: 1, 2: 1, 3: 1}
total cost: 160
```

In this trivial instance, opening only warehouse 1 — the cheapest at 100 to open, with shipping costs 10/20/30 to the three customers — costs `100 + 60 = 160`, which beats every other configuration. CP-SAT proves this is optimal in essentially zero time.

In a real workflow, the solver's output is run through the verifier from Chapter 4 before being trusted: build a `Solution` object from the `solver.value(...)` calls, then call `evaluate(instance, sol)`. If the verifier accepts the result, the encoding has produced a feasible solution and the costs match. If it rejects, a bug in the encoding has just been caught. The model here is short enough to read end to end, so we skip the step.

5.5 When the Natural Formulation Does Not Translate

The facility-location model translated line-for-line from paper to code. Not every problem is that smooth. Consider the **traveling salesman problem** (TSP): given n cities and pairwise distances `cost[i][j]`, find a shortest tour visiting every city exactly once and returning to the start.

The natural way to think about a tour is as a permutation — city x_i is visited at position i. And indeed, CP-SAT provides an `AllDifferent` constraint that handles exactly this:

```
x = [model.new_int_var(0, n - 1, f"x_{i}") for i in range(n)]
model.add_all_different(x)
```

So far so good. But the objective is `cost[x_0][x_1] + cost[x_1][x_2] + ...` — and x_0, x_1 are *variables*, not constants. We cannot index into a Python list with a CP-SAT variable. Browsing the constraint catalogue, we find `add_element`, which performs variable-indexed lookups — but only into one-dimensional arrays. For a two-dimensional distance matrix, we would need nested element constraints, each requiring auxiliary variables. The natural formulation has no clean translation into CP-SAT.

But browsing further, we find `add_circuit` — a constraint that enforces a Hamiltonian cycle over a set of arcs. This solves the problem directly, if we remodel. Instead of "which city at each position," ask "which arcs are used?" A Boolean per city pair says whether the tour travels directly from i to j:

```
x = {(i, j): model.new_bool_var(f"x_{i}_{j}")
     for i in range(n) for j in range(n) if i != j}

model.add_circuit([(i, j, x[i, j]) for i, j in x])
model.minimize(sum(cost[i][j] * x[i, j] for i, j in x))
```

The objective is now a plain linear sum — `cost[i][j]` is a constant per arc, no variable indexing needed. And `add_circuit` handles subtour elimination internally.

The natural description of a problem and the representation that fits CP-SAT can be quite different things. When a formulation does not translate cleanly, the answer is often not to force it through but to look at what the solver *can* express and remodel accordingly. The constraint catalogue in Chapter 7 is that vocabulary; it is worth knowing what is in it before you start modeling a new problem.

5.6 The Shape of a CP-SAT Program

Every CP-SAT program follows the same six-step shape, with the middle four steps directly mirroring the recipe from Chapter 3:

1. **Build the model object.** `model = cp_model.CpModel()`.
2. **Add variables.** Calls to `model.new_bool_var(...)`, `model.new_int_var(...)`, or the other variable factories — typically one factory call per variable family in the paper model, plus any auxiliary variables the encoding needs (Step 3 of the recipe).
3. **Add constraints.** Calls to `model.add(...)` for linear and logical constraints, and direct method calls (`model.add_all_different(...)`, `model.add_circuit(...)`) for global constraints. A simple paper constraint family often becomes a single call; a more complex one may need several `add(...)` calls and helper variables to encode (Step 4 of the recipe).
4. **Add an objective** (if any). `model.minimize(expr)` or `model.maximize(expr)`. The expression is the same one the math model wrote (Step 5 of the recipe). Pure feasibility models — *is there any valid configuration?* — skip this step.
5. **Solve.** `solver.solve(model)` returns a status code.
6. **Read the answer.** `solver.value(var)` for each variable of interest, `solver.objective_value` for the objective, conditional on a successful status.

The facility-location example used the simplest version of every piece; the TSP example showed that not every problem is that smooth and that knowing CP-SAT's vocabulary matters. The chapters that follow provide that vocabulary. Chapter 6 catalogs the variable types — integers with bounded domains, intervals for scheduling, and several others. Chapter 7 catalogs the constraint families, including the global constraints (`add_all_different`, `add_circuit`, `add_no_overlap`, `add_cumulative`) that make CP-SAT what it is. Chapter 8 covers objectives, status codes, and time-limit handling. Chapter 9 puts the whole workflow loop together.

For now, you have the shape. The rest of the manuscript fills it in.

Chapter 6

The Variable Vocabulary

"The beginning of wisdom is to call things by their proper name." — Confucius

Variables are the nouns of a CP-SAT model. Every decision the solver is allowed to make is held by a variable, and every variable belongs to one of five constructor families. The vocabulary is small by design: most of the modeling work is recognizing which constructor matches a given decision, and the recognition becomes reflex after a few examples.

CP-SAT has no continuous variables — there is no `new_real_var`. Quantities that are naturally fractional (prices, distances, ratios) must be scaled to integers, working in cents, millimeters, or some other unit small enough to capture the precision the problem actually needs. The integer-only restriction is part of the design.

6.1 Integer Variables — `new_int_var`

A bounded integer.

```
x = model.new_int_var(lb=0, ub=100, name="x")  # closed [0, 100]
qty = model.new_int_var(0, 50, "qty")
delta = model.new_int_var(-10, 10, "delta")
```

The constructor takes a lower bound, an upper bound, and a name. Both bounds are *inclusive*: the variable's domain is the integer interval $\{lb, lb+1, \ldots, ub\}$.

Tighter bounds tend to help. A bound is information the solver uses for propagation, branching, and presolve. A variable declared as `[0, 100]` is much cheaper to reason about than the same variable declared as $[0, 10^9]$, even if the second bound is technically valid. The looser bound forces the solver to consider possibilities it never needed to.

Typical uses for integer variables include: quantities (*how many units to ship*, *how many nurses on this shift*); times on a discrete grid (*the start hour of task A*, *the minute at which a meeting begins*); discrete spatial coordinates (*the x- and y-position of a piece on a grid*); counts and totals; indices into a list (*which warehouse serves customer C*, in $\{0, 1, \ldots, n-1\}$). For indices specifically, there is often a competing encoding using one Boolean per choice — see Section 6.2.

6.2 Boolean Variables — `new_bool_var`

Booleans are the real workhorse of CP-SAT — the SAT core is built around them. A Boolean is a 0/1 integer *and* a logical literal at the same time. This dual nature is what makes CP-SAT models concise.

```
take = model.new_bool_var("take_item")
edge_used = model.new_bool_var("edge_a_b")
is_late = model.new_bool_var("is_late")
not_take = ~take  # negation - a free view, no new variable
```

The `\~` operator returns the negation as a *view*: no new variable is allocated, and the negation participates in constraints exactly as the original Boolean would. This is a small but useful efficiency — every Boolean in your model has a free negation that can appear in any constraint position.

The dual nature is on display in expressions like:

```
model.add(take + take2 + take3 <= 2)     # arithmetic: Bool as 0/1
model.add_bool_or([take, take2, take3])  # logic:      Bool as literal
```

The first treats the Booleans as integers and asserts a linear inequality on their sum. The second treats them as logical literals and asserts a disjunction. Both are legitimate, both are efficient, and the same variable can appear in both kinds of constraint without conversion.

Typical uses for Booleans include: yes/no decisions (*include item*, *use edge*, *schedule task*, *hire candidate*); assignments expressed as Bool-per-pair — the matrix idiom from the FLP example of Section 5.5, with $x_{i,j} \in \{0, 1\}$ indexed by pairs; state indicators that other constraints react to (*did the budget get exceeded?*, *is this nurse working a late shift?*); reified relations, where a Boolean is constrained to be true exactly when some other relation holds, used in conditional logic via `only_enforce_if`.

Booleans are typically a natural starting point for routing, scheduling, and graph problems, where CP-SAT's SAT core is at its strongest. A Bool-per-pair assignment encoding often works well because it exposes many logical relations directly to the SAT layer. But it is not universally better than a single integer "which of k" variable: integer-valued encodings can be superior when they enable stronger global constraints or cleaner arithmetic. Treat the choice as a modeling tradeoff and benchmark when it matters.

6.3 Constants — `new_constant`

A variable whose domain is a single value.

```
zero = model.new_constant(0)
seven = model.new_constant(7)
```

We use this sparingly. The main case is *interface uniformity*: a function expects a variable, but for some inputs the value is fixed by the data.

```
def add_capacity(model, weight, capacity):
    model.add(weight <= capacity)

add_capacity(model, model.new_int_var(0, 100, "w"), 50)
add_capacity(model, model.new_constant(42), 50)
```

The second call passes a constant where the function expects a variable, avoiding a separate code path for the fixed-value case. For most expressions, CP-SAT also accepts a plain Python integer in place of a variable, so `new_constant` is genuinely needed only when the API requires a variable object specifically.

Beyond uniformity, there is a modest debugging benefit: `new_constant` creates an actual `IntVar` (with an auto-generated name) that appears in the exported model and the solver logs, where a

bare integer folded into a coefficient would not.

6.4 Interval Variables — `new_interval_var`

An interval variable represents a span on a discrete axis: a *start*, a *size* (duration), and an *end*, with the relation `start + size = end` enforced automatically. CP-SAT offers four flavors, increasing in expressiveness and cost.

```
# Fixed size, mandatory
fix = model.new_fixed_size_interval_var(start=s, size=5, name="fix")

# Variable size, mandatory
flex = model.new_interval_var(start=s, size=d, end=e, name="flex")

# Fixed size, optional (presence Boolean)
opt = model.new_optional_fixed_size_interval_var(
    s, 5, is_present=p, name="opt"
)

# Variable size, optional
opt_flex = model.new_optional_interval_var(
    s, d, e, is_present=p, name="opt_flex"
)
```

The *mandatory* flavors require the interval to be present in any feasible solution. The *optional* flavors take an additional Boolean `is_present`. When `is_present = False`, the interval is treated as *absent* by every interval-aware constraint that touches it: it does not occupy resources, does not interact with no-overlap propagators, and effectively disappears from the scheduling problem for that solution.

The cost rises with each step of expressiveness. A fixed-size mandatory interval has the smallest representation; a variable-size optional interval has the largest. The rule of thumb is to take the cheapest flavor that expresses the problem.

Typical uses for intervals include: tasks on a timeline, with start, duration, and end bundled into one object; optional tasks (*this maintenance might or might not happen this week*); rectangles in 2D packing, where each rectangle is two intervals (one in x, one in y); reservations on a shared resource (meetings, room bookings, machine usage).

An interval is only interesting in combination with an interval-aware constraint. By itself, an interval variable is a small bundle of three integer variables with one arithmetic relation between them — no special semantics. The reason intervals exist is that CP-SAT's *interval-aware constraints* — `add_no_overlap`, `add_no_overlap_2d`, `add_cumulative` — accept intervals as input and run dedicated scheduling propagators on the result.

If you only need a `[start, end]` span and never feed it into one of these constraints, plain integer variables are cheaper and clearer. The matching constraints are the next chapter's subject; for now, treat intervals as the input shape to a scheduling vocabulary you have not yet seen.

Intervals fit some shapes of scheduling better than others. They shine when the time (or space) axis is *somewhat continuous* — start times anywhere in a wide horizon, sizes drawn from a range, optional rectangles to be packed into a free 2D area. When the problem instead has a small number of *discrete slots* and tasks occupy known-sized blocks, a Bool-per-(task, slot) encoding often models the same situation more efficiently than intervals plus `add_no_overlap`. And when transitions between tasks dominate the cost — sequence-dependent setup times, vehicle routing — the problem is closer to routing than to scheduling, and `add_circuit` (next chapter) is the better starting point.

6.5 Sparse Domains — `new_int_var_from_domain`

A variable whose domain is an explicit set of values rather than a contiguous range.

```
allowed = cp_model.Domain.from_values([2, 5, 8, 10, 20, 50, 90])
freq    = model.new_int_var_from_domain(allowed, "frequency_MHz")
```

This is the right tool when the restriction is *part of the problem*: allowed broadcast frequencies, standardized paper sizes, container capacities, supported product configurations, the discrete set of speeds at which a motor can run.

Using sparse domains as a *performance* tool is also legitimate, but it is a later move: CP-SAT uses *lazy order encoding* internally, so declaring `new_int_var(0, 100, ...)` does not eagerly materialize one hundred Booleans, and a wide bound is not, in itself, a wide cost. Pruning the domain to values you believe are reachable can still help — sometimes substantially — but it can also hurt, and worse, it can introduce subtle bugs when a value you excluded turns out to be feasible after all. As a rule, start with the bounds the problem gives you, and treat sparse-domain pruning as a tuning lever to experiment with once a correctness baseline is in place.

Chapter 7

The Constraint Vocabulary

"The limits of my language mean the limits of my world." — Ludwig Wittgenstein

Constraints are the verbs of a CP-SAT model. Where Chapter 6 showed how to declare the unknowns, this chapter shows how to bind them — every relation, restriction, and rule that distinguishes a valid configuration from an invalid one. CP-SAT ships with a curated set of constraint families. Some *global* families in the catalogue carry a dedicated propagator far stronger than what a modeler would write out of plain inequalities; others are flattened internally to clauses or linear constraints. Either way, calling the global by name expresses intent and lets the solver pick the best encoding.

7.1 Linear Constraints — `model.add`

The bread and butter of any CP-SAT model. `model.add(...)` accepts any linear (in)equality and registers it as a constraint.

```
model.add(10*x + 15*y <= 200)
model.add(x + z == 2*y)
model.add(x < y + z)  # rewritten internally to x <= y + z - 1
# Double-bounded:
model.add_linear_constraint(10*x + 15*y, lb=-100, ub=10)

# Variable-length sums are built with Python's built-in sum(...):
model.add(sum(w[i] * take[i] for i in items) <= capacity)
# Exactly-one over a row of Booleans:
model.add(sum(x[i, j] for i in F) == 1)
```

The expressions on the left are built by overloading: `+`, `*`, `-`, `==`, `<=`, `<`, `>=`, `>` on CP-SAT variables (and integer constants) all return CP-SAT expression or constraint objects. Python's built-in `sum(...)` chains the additions for you, so a generator expression over a family of variables — the workhorse pattern for capacity and counting constraints — drops directly into `model.add(...)`. The constraint object that comes out is registered with the model.

CP-SAT works exclusively in integers, so strict inequalities are rewritten to non-strict-with-shift: $x < y$ becomes $x \leq y - 1$. The semantics are identical, but knowing that the rewrite happens prevents confusion when reading the solver's logs or inspecting an exported model. For double-bounded expressions — *the load is between 50 and 100* — `add_linear_constraint(expr, lb, ub)` is occasionally cleaner than two separate `model.add` calls; both forms produce the same model.

The shapes that recur in essentially every model are capacity ($\sum_i w_i x_i \le C$), conservation laws (*inflow equals outflow plus storage*), counting (*exactly k of these are taken*), budget constraints, and flow conservation per node ($\sum_{j \in \delta^+(i)} x_{ij} - \sum_{j \in \delta^-(i)} x_{ji} = b_i$).

With practice, almost any constraint can be *expressed or approximated* as a linear (in)equality — often with a few helper Booleans or auxiliary variables. Mixed-Integer Programming solvers (Gurobi, CPLEX, HiGHS) accept *only* this family of constraints, and entire industries run on models built from `model.add(...)` alone. CP-SAT exposes a much richer vocabulary in the sections ahead — Boolean logic, reification, scheduling globals, routing globals — but the linear shape is the universal one. The richer constraints are *convenience and propagation strength*, not necessity. Understanding linear-only modeling is therefore a transferable skill: it carries over to every MIP solver and remains the lingua franca even when CP-SAT's globals are the better choice for a given problem.

7.2 Boolean Logic — `add_bool_or`, `add_bool_and`, `add_implication`

Pure logical constraints over Boolean *literals* — Booleans or their negations.

```
model.add_bool_or([a, b, c])       # at least one true
model.add_at_least_one([a, b, c])  # alias
model.add_at_most_one([a, b, c])
model.add_exactly_one([a, b, c])
model.add_bool_and([a, ~b, c])     # all true (mostly inside only_enforce_if)
model.add_implication(a, b)        # a => b  (binary only)
model.add_bool_xor([a, b, c])      # odd number true
```

The arguments are *literals*: Boolean variables or their negations via `\~`. An integer variable with domain $\{0, 1\}$ is *not* a literal and will be rejected — if the model needs logical operations, declare the variable with `new_bool_var` from the start.

CP-SAT does not let you nest Boolean constraints — each call takes *literals*, never formulas. In modeling, however, the rules you want to express rarely arrive in flat form. A typical specification might read *"at least one of three production lines is fully prepared"* — formally $(a_1 \wedge b_1) \vee (a_2 \wedge b_2) \vee (a_3 \wedge b_3)$, a disjunction of conjunctions — or *"if either of two combined conditions holds, ship today"*, mixing conjunction, disjunction, and implication in a single formula. CP-SAT will not accept these as written. Unfolding the formula into flat clauses is part of the modeling work, and it is *classical propositional logic* — the same machinery any first course on logic introduces: De Morgan's laws, distribution, conjunctive normal form, and Tseitin's transformation.

Two routes are available, and which one fits depends on the size and shape of the formula:

Direct unfolding by distribution. Push negations inward via De Morgan ($\neg(x \wedge y) \equiv \neg x \vee \neg y$, $\neg(x \vee y) \equiv \neg x \wedge \neg y$) and distribute disjunction over conjunction ($x \vee (y \wedge z) \equiv (x \vee y) \wedge (x \vee z)$) until the formula is in *conjunctive normal form* — a conjunction of clauses, where each clause is a disjunction of literals. Each clause maps to one `add_bool_or`. Distribution can blow up the formula size: a disjunction of n binary conjunctions has up to 2^n clauses in CNF, which is fine for a few cases but quickly impractical.

Tseitin's transformation with auxiliary Booleans. When direct distribution would be too large, introduce a fresh Boolean t_S for every sub-expression S and constrain it with $t_S \Leftrightarrow S$ (three short clauses for a binary conjunction or disjunction). The compound formula then refers to the auxiliaries in place of their sub-expressions, and the encoding grows *linearly* in the size of the formula. This is standard SAT preprocessing, in widespread use since Tseitin's 1968 paper.

A worked example, on the formula above:
$(a_1 \wedge b_1) \ \vee \ (a_2 \wedge b_2) \ \vee \ (a_3 \wedge b_3)$

Direct CNF. Distribute disjunction over the three conjunctions: pick one literal from each, giving $2^3 = 8$ clauses such as $a_1 \vee a_2 \vee a_3$, $a_1 \vee a_2 \vee b_3$, …, $b_1 \vee b_2 \vee b_3$. Eight `add_bool_or` calls. With four pairs the count rises to sixteen, with eight pairs to 256 — direct distribution does not scale.

Tseitin form. Introduce t_1, t_2, t_3 with each $t_i \Leftrightarrow a_i \wedge b_i$, then assert the disjunction over the auxiliaries:

```
t = [model.new_bool_var(f"t_{i}") for i in range(3)]
for i in range(3):
    # t_i <=> (a_i AND b_i): three clauses
    model.add_implication(t[i], a[i])           # t_i => a_i
    model.add_implication(t[i], b[i])           # t_i => b_i
    model.add_bool_or([~a[i], ~b[i], t[i]])     # a_i AND b_i => t_i
model.add_bool_or(t)                            # t_1 OR t_2 OR t_3
```

Three auxiliary Booleans, ten clauses total, scaling linearly with the number of pairs. The two encodings are logically equivalent on the original variables; they differ in how they exploit the solver's machinery.

`only_enforce_if` (Section 7.3) automates a narrow slice of this — specifically, conditional enforcement of a single constraint under a literal or short conjunction of literals. It does not generalize to arbitrary nested formulas; for those, the manual unfolding above is the standard approach.

A concrete consequence for the basic vocabulary above: `add_implication` is binary only ($a \Rightarrow b$). Implications with several antecedents ($a \wedge b \Rightarrow c$) flatten by definition of implication to $\neg a \vee \neg b \vee c$ — a single call to `add_bool_or([\~a, \~b, c])` — without needing an auxiliary.

Three Boolean patterns appear in essentially every Boolean-rich model: exactly-one over an assignment row — *each customer goes to exactly one warehouse*, the FLP assignment constraint from Section 5.5 — at-most-one for mutually exclusive flags such as pricing tiers or compatibility classes, and binary implication for *if A then B* rules.

Short clauses — binary `add_bool_or([a, b])` in particular — are among the cheapest propagation CP-SAT performs, and the SAT layer's clause-learning machinery tends to do well when the encoded logic decomposes into many small clauses rather than a few wide ones.

7.3 Reification — `only_enforce_if`

Apply a constraint only *when* a Boolean (or a conjunction of Booleans) is true.

```
load = model.new_int_var(0, 1000, "load")

# Capacity depends on which truck we use.
model.add(load <= 50).only_enforce_if(truck_a)
model.add(load <= 80).only_enforce_if(truck_b)

# Conjunction of literals: enforced only when b1 is true and b2 is false.
model.add(x + y == 10).only_enforce_if([b1, ~b2])
```

`only_enforce_if` is the conditional-constraint operator. The constraint is in force when the literal (or all of the literals in the list) is true; when any of them is false, the constraint is treated as inactive — neither violated nor satisfied, simply absent for that solution. It is the typical way to express that a constraint should depend on a Boolean.

Not every constraint supports `only_enforce_if`, and the cost of using it varies considerably across

those that do — cheap on linear and Boolean constraints, much heavier on the scheduling and routing globals, where a native mechanism (optional intervals, self-loop literals) is usually the better expression of conditionality. The OR-Tools documentation lists which constraints accept it.

7.4 Full Reification — A Boolean Equivalent to a Relation

`only_enforce_if` from Section 7.3 is *half* reification: it forces a constraint when a literal is true but says nothing about the false case. Posting it in both directions — once for the literal, once for its negation — gives the full equivalence and turns a numeric relation (*exceeds threshold*, *is over budget*) into a Boolean the rest of the model can reason about.

```
over = model.new_bool_var("over_limit")

# over == 1   iff   x >= threshold
model.add(x >= threshold).only_enforce_if(over)
model.add(x <= threshold - 1).only_enforce_if(~over)
```

The two `only_enforce_if` lines together establish the equivalence over $\Leftrightarrow$ ($x \geq$ threshold). Once reified, `over` is a first-class Boolean: it can appear in the objective (*penalty for going over*), in further logical constraints (*if over the limit, route through warehouse two*), or anywhere else a literal is accepted.

Any time the model needs to *talk about* whether a numeric relation holds — to penalize it, branch on it, combine it with other logic — full reification produces the Boolean to talk about.

Often, however, half-reification is enough. If the Boolean's only purpose is to drive a penalty in the objective — pay when `over` goes high, no rule when it does not — the second `only_enforce_if` is unnecessary: the minimization keeps `over` at zero whenever the relation does not force it to one. Full reification is required only when the *false* direction itself drives logic — *if not over, then route through warehouse two*. Posting both directions when one would do costs extra constraints and propagation, so drop the half that is not needed.

7.5 `add_all_different`

A list of variables must take pairwise distinct values.

```
model.add_all_different([x1, x2, x3, x4])

# Affine expressions also work - the classic n-queens diagonal trick:
model.add_all_different([queens[i] + i for i in range(n)])
model.add_all_different([queens[i] - i for i in range(n)])
```

Mathematically equivalent to $\binom{n}{2}$ pairwise `!=` constraints, but with a *strictly stronger* propagator in the domain-reasoning sense: the global form reasons about the entire list at once — bipartite matching, in essence — and prunes domain values that no pairwise `!=` could discover.

One small note: CP-SAT will auto-lift a cluster of pairwise `!=` constraints into an all-different propagator when no `add_all_different` is already declared on the same variables, so writing the pairwise form does not necessarily forfeit global propagation. Declaring both forms on the same variables disables that auto-lift, which is worth being aware of.

The constraint accepts not only bare variables but also affine expressions. The n-queens diagonal trick is the elegant illustration: queen i sits in column `queens[i]`, on diagonal `queens[i] + i`, and on

anti-diagonal `queens[i] - i`. No two queens may share any of the three, and three `add_all_different` calls express the entire problem.

Typical uses are conflict and uniqueness rules in real models. Time-slot conflicts where no two tasks may share a start time. Strict orderings, where the variables encode positions in a permutation and every position must be distinct. Unique resource assignment — each task takes its own machine, each request gets its own identifier, each vehicle a distinct depot. Frequency assignment where neighboring transmitters must not share a frequency. Anything that reads *"no two X share a Y"* fits naturally; sudoku and n-queens are the textbook illustrations of the same pattern, but the constraint earns its place by appearing in scheduling, allocation, and routing models far more often than in puzzles.

7.6 `add_circuit` — TSP and Routing

A list of `(tail, head, literal)` triples; the chosen arcs (literal = true) must form a single Hamiltonian circuit.

```
arcs = [
    (u, v, model.new_bool_var(f"e_{u}_{v}"))
    for (u, v) in directed_edges
]
model.add_circuit(arcs)
model.minimize(sum(cost[u, v] * lit for (u, v, lit) in arcs))
```

The circuit propagator handles subtour elimination internally — generating the cuts that exclude shorter cycles on the fly — so the model does not need position variables, element constraints, or any of the manual scaffolding the natural formulation would require.

Several adaptations fall out of the same shape. Optional vertices: add a self-loop `(v, v, skip_v)`; if `skip_v = 1`, the vertex is bypassed. Shortest path between two endpoints: zero-cost self-loops on every non-endpoint plus a forced closing arc. Prize-collecting TSP: penalize the skip-loop literals in the objective, letting the solver decide which vertices are worth visiting. Multiple tours sharing a depot are covered by `add_multiple_circuit`, the vehicle-routing variant.

CP-SAT's `add_circuit` handles surprisingly large routing instances — a few hundred nodes, in many cases, and with rich side constraints. For *very* large pure TSP — tens of thousands of cities — specialist solvers like Concorde and LKH still win, as Section 5.2 noted.

7.7 `add_no_overlap` — Single-Machine Scheduling

A list of intervals that may not overlap on the time axis.

```
intervals = [
    model.new_fixed_size_interval_var(start[i], duration[i], f"task_{i}")
    for i in range(n_tasks)
]
model.add_no_overlap(intervals)
```

Captures a single resource — machine, room, employee — that does at most one thing at a time. Optional intervals (Section 6.4) are respected automatically: when their presence Boolean is false, they are ignored.

For job-shop scheduling — multiple machines, each running its own subset of tasks — the natural shape is one `add_no_overlap` call per machine:

```
for m in machines:
    model.add_no_overlap([intervals[t, m] for t in tasks if can_run(t, m)])
```

The propagator behind `add_no_overlap` draws on *edge finding*, *detectable precedences*, and *energetic reasoning*. Hand-rolling the same constraint with pairwise disjunctions on integer start times is correct but tends to be slower on non-trivial sizes.

7.8 `add_no_overlap_2d` — 2D Packing

The two-dimensional analogue: axis-aligned rectangles in the plane that may not overlap.

```
x_intervals = [
    model.new_fixed_size_interval_var(xs[i], w[i], f"x_{i}")
    for i in range(n)
]
y_intervals = [
    model.new_fixed_size_interval_var(ys[i], h[i], f"y_{i}")
    for i in range(n)
]
model.add_no_overlap_2d(x_intervals, y_intervals)
```

Each rectangle is described by two intervals — one for its x-extent, one for its y-extent. The constraint enforces that for every pair of rectangles, *either* their x-intervals are disjoint *or* their y-intervals are disjoint, which is exactly the geometric definition of non-overlap.

Typical uses are 2D bin packing, pallet loading, floor planning, chip placement, and cutting-stock with rectangular pieces. Optional rectangles work the same way — items that may or may not be packed have their presence Boolean carried by both interval components.

7.9 `add_cumulative` — Resource-Capacity Scheduling

The capacitated generalization of `add_no_overlap`. Each interval has a *demand*; at every moment, the sum of demands of intervals running at that moment must not exceed a *capacity*.

```
model.add_cumulative(
    intervals=[i0, i1, i2, i3],
    demands=[2, 3, 1, 4],     # paired by index
    capacity=5,
)
```

Typical uses are resource-constrained project scheduling (*RCPSP*), where tasks share a pool of workers, machines, or electricity; parallel-machine settings with shared capacity, like meeting rooms holding N simultaneous sessions; and inventory in a warehouse, where the "demand" of an interval is the units it consumes and the capacity is the warehouse size.

Capacity itself can be a CP-SAT variable rather than a constant: declaring `capacity` as an integer variable and minimizing it returns the smallest pool that still completes the work — a useful answer to capacity-planning questions.

When the capacity is exactly 1, the specialized `add_no_overlap` exists for the single-resource case and tends to propagate more strongly than `add_cumulative` on that specialization.

7.10 Equality Helpers — Max, Min, Abs, Multiplication, Division, Modulo

The non-linear corners of integer arithmetic — `max`, `min`, $|x|$, $x \cdot y$, $x \div y$, $x \mod m$ — are not legal inside `model.add` directly. CP-SAT exposes them as auxiliary-variable equalities:

```
m = model.new_int_var(0, 100, "max")
model.add_max_equality(m, [x, y, z])

mn = model.new_int_var(0, 100, "min")
model.add_min_equality(mn, [x, y, z])

abs_x = model.new_int_var(0, 100, "abs_x")
model.add_abs_equality(target=abs_x, expr=x)

xy = model.new_int_var(-1_000_000, 1_000_000, "xy")
model.add_multiplication_equality(xy, [x, y])

q = model.new_int_var(0, 100, "q")
model.add_division_equality(q, x, z)  # integer division: 5 // 2 == 2

r = model.new_int_var(0, 2, "r")
model.add_modulo_equality(r, x, 3)
```

The pattern is uniform: declare an auxiliary variable, then call the matching `add_*_equality` to bind it to the non-linear expression. The auxiliary then participates in further constraints just like any other integer variable.

Some of these can be sidestepped with linear inequalities — *minimizing the max* of values, for instance, needs only $m \geq \text{value}_i$ for each i and `model.minimize(m)`. Reach for the equality helpers when the relation is genuinely nonlinear in the model.

7.11 A Few More, Briefly

A handful of constraints we will not dwell on but that are worth recognizing when the problem calls for them.

`add_element` — binds a target to an array entry whose index is a decision variable: `target == array[index]`. Useful for lookup tables, piecewise functions on a discrete domain, and any "pick the value at this index" rule. Array entries may be constants or variables.

`add_allowed_assignments` / **`add_forbidden_assignments`** — a tuple of variables must take its joint value from (or avoid) an explicit list of allowed combinations. Useful for allowed shift patterns, compatible feature bundles, and truth-table-style rules. Cost grows with the number of rows; when the rule has sequential structure, `add_automaton` below is often more compact.

`add_multiple_circuit` — vehicle-routing-style problems where several tours share a depot. Same triple-list shape as `add_circuit`; the propagator allows multiple disjoint circuits provided they all pass through the designated depot vertex.

`add_reservoir_constraint` — a running level (inventory, staff on duty, water in a tank) bounded by $[\min, \max]$ as discrete `+/-` events fire over time. Inputs are a list of times and a list of per-event level changes; the constraint enforces that the running sum stays within the bounds.

`add_automaton` — a sequence of integer variables must follow a finite-state machine. Useful for forbidden patterns in a roster (*no more than three night shifts in a row*), protocol compliance, batch-scheduling state transitions. The right tool when the rule is naturally stated as "the sequence is

accepted by this DFA."

`add_inverse(v, w)` — two equal-length variable arrays that must be each other's inverses: `v[i] = j` if and only if `w[j] = i`. Used in stable-matching shapes and any time a problem has two natural representations of the same combinatorial object that must agree (the modeling pattern is called *channelling between dual encodings*).

The CP-SAT documentation is the authoritative reference for the corners of each.

Chapter 8

Objectives and Status

"Tell me how you measure me, and I'll tell you how I'll behave." — Eliyahu Goldratt

This chapter covers the objective on the model side, and the status code, bounds, and parameters that let us interpret the result on the solver side.

8.1 Stating the Objective

A CP-SAT model has at most one objective.

```
model.minimize(linear_expression)
model.maximize(linear_expression)
```

The expression must be *linear* in the variables — sums, scaled sums, and integer-coefficient combinations. As elsewhere in CP-SAT, mixing integer and Boolean variables is fine: a Boolean is a $\{0, 1\}$ integer in arithmetic, and negations via `\~` are accepted in objectives just like in constraints.

```
# Total cost of selected items
model.minimize(sum(cost[i] * take[i] for i in range(n)))

# Mixing terms - Booleans contribute 0 or 1, with negation if useful
model.minimize(sum(weight[i] * x[i] for i in range(n)) - 100 * ~all_done)
```

Calling `minimize` or `maximize` a second time *replaces* the previous objective; it does not add to it. CP-SAT does not have a built-in concept of multiple simultaneous objectives. To combine several goals — *minimize lateness AND minimize cost* — the modeler commits to a combination strategy, and Section 8.3 covers the two standard ones.

A model with no objective is a *constraint-satisfaction* problem: the solver returns any feasible solution it can find, with no preference among them. CP-SAT is strong in this mode — Sudoku, schedule feasibility, and configuration validation all fall here. A dummy objective like `model.minimize(0)` is not needed; feasibility-only tells the solver *find any solution*, which is the natural question for many problems and frees the search from optimization overhead.

8.2 Nonlinear Objectives via Auxiliaries

The objective expression must be linear, but *the quantity we want to optimize* often is not — a maximum, an absolute value, a multiplicative cost. The standard move: introduce an auxiliary

variable bound to the nonlinear quantity, then make the objective linear in the auxiliary. The pattern is the same one Chapter 7's equality helpers (Section 7.10) used for nonlinear *constraints*, applied to the *objective*.

The most common case is *minimize a maximum* — a makespan, a worst-case latency, a peak load. The direct route uses `add_max_equality`:

```
makespan = model.new_int_var(0, horizon, "makespan")
model.add_max_equality(makespan, [end[t] for t in tasks])
model.minimize(makespan)
```

The linear-bound trick from Section 7.10 expresses the same thing without the equality constraint:

```
makespan = model.new_int_var(0, horizon, "makespan")
for t in tasks:
    model.add(makespan >= end[t])
model.minimize(makespan)
```

The bounds say *makespan is at least every finish time*, and the minimization pulls the bound tight to the actual maximum on its own. The mirror pattern — *maximize a minimum* — uses $m \leq$ value for each value with `model.maximize(m)`.

For other nonlinear shapes, the auxiliary-variable pattern carries through unchanged:

```
# Total absolute deviation from per-task target dates
total_dev = model.new_int_var(0, big, "total_dev")
deviations = []
for t in tasks:
    d = model.new_int_var(0, big, f"dev_{t}")
    model.add_abs_equality(d, finish_time[t] - target[t])
    deviations.append(d)
model.add(total_dev == sum(deviations))
model.minimize(total_dev)
```

Same recipe: scaffold an auxiliary for each nonlinear piece (here an absolute value per task), sum the auxiliaries, make the sum the objective. The pattern handles modulo objectives, multiplicative penalties, and any other nonlinear corner of integer arithmetic.

8.3 Combining Multiple Goals

When the model has more than one goal, two standard patterns combine them into a single objective. They differ in whether the goals are negotiable against each other (weighted sum) or strictly prioritized (lexicographic).

8.3.1 Weighted sum

The simplest combination: pick weights and minimize the weighted sum.

```
W_COST       = 1
W_LATENESS   = 100
W_PREFERENCE = 10

model.minimize(
      W_COST       * total_cost
    + W_LATENESS   * total_lateness
```

```
    - W_PREFERENCE * preference_score
)
```

The weights encode how much one unit of one goal is worth in units of another — *one minute of lateness is worth one hundred dollars of cost.* Weighted sums are the right tool when the goals are genuinely commensurable and the trade-off rate is known or can be estimated.

Setting one weight extremely large to mean "this matters most" — `W_LATENESS = 10**9` or similar — tends to backfire: the objective becomes effectively *only* the high-weighted term with the others lost in the noise, and the solver can struggle numerically when coefficients span many orders of magnitude. When a goal is genuinely top-priority — optimized first, others as tie-breakers — *lexicographic ordering* expresses that directly.

8.3.2 Lexicographic priorities

When goals are strictly ranked — *first satisfy lateness, then minimize cost* — solve in *stages.* Each stage optimizes one objective; before moving to the next, the previous objective's optimal value is locked in as a constraint.

```
# Stage 1: minimize lateness
model.minimize(total_lateness)
status = solver.solve(model)
best_lateness = round(solver.objective_value)

# Stage 2: lock lateness in, optimize cost as a tie-breaker
model.add(total_lateness <= best_lateness)
# model.clear_objective() is the explicit form; minimize() replaces it.
model.minimize(total_cost)

# Warm-start: the stage-1 solution is still feasible and usually
# near-optimal for the new objective - hint from it.
for v in decision_vars:
    model.add_hint(v, solver.value(v))
solver.parameters.repair_hint = True

status = solver.solve(model)
```

The pattern generalizes to any number of stages: optimize, lock, optimize the next. The result is a strict priority ordering — every later objective is optimized only over the optimal-for-the-earlier set. The hinting step matters: changing the objective does not change feasibility, so the previous solution is a free starting point for the next stage and can speed up the next solve substantially.

A practical note: by the time a model reaches its third or fourth lexicographic stage, the search space has been trimmed aggressively by the locked-in constraints, and the remaining problem can be unexpectedly hard. Setting `solver.parameters.max_time_in_seconds` on the secondary stages keeps a tie-breaker from consuming the entire time budget.

8.4 Reading the Solver's Answer

The solver returns a *status*, an *objective value* (when an objective is set), a *best objective bound*, and a wall-clock time.

8.4.1 Solver parameters

A handful of parameters get set on essentially every nontrivial solve:

```
solver = cp_model.CpSolver()
solver.parameters.max_time_in_seconds = 30.0  # soft time limit
solver.parameters.num_workers = 8             # parallel portfolio
solver.parameters.log_search_progress = True  # readable progress log

status = solver.solve(model)
```

`max_time_in_seconds` is the most important. Without it, the solver runs until it finds the optimum, proves infeasibility, or exhausts memory — which on hard instances can mean *days*. Set a budget appropriate to the problem and the moment.

`num_workers` controls the parallel portfolio — different workers run different strategies and share learned clauses with each other. Eight is a sensible default on a modern laptop, larger on a workstation; Section 5.3's hardware sketch translates directly to a worker count. The default is *all* cores including hyperthreads, which is not always best — on heavily contended machines, capping at the number of physical cores often runs faster.

`log_search_progress` turns on a human-readable trace of the search. When debugging a slow model, the log is the first place to look — it shows when bounds tighten, when the LP relaxation bites, and when LNS finds an improvement. The full parameter list is large; the OR-Tools documentation is the authoritative reference for the corners.

8.4.2 Status codes

The status returned by `solver.solve(...)` is one of five values, each with a precise meaning:

Code	Meaning
`OPTIMAL`	A solution was found *and* proven to be the best possible.
`FEASIBLE`	A solution was found, but optimality has not been proven.
`INFEASIBLE`	The solver proved that *no* solution exists.
`MODEL_INVALID`	The model itself is malformed (empty domains, integer overflow, API misuse).
`UNKNOWN`	The solver gave up before deciding either way.

Three of these are *proofs*: `OPTIMAL`, `INFEASIBLE`, and `MODEL_INVALID` are mathematical certainties about the model. `FEASIBLE` is a *partial* result — there is a solution, and we have it, but we have not proven that no better one exists. `UNKNOWN` is the absence of any conclusion at all.

The most common confusion is between `OPTIMAL` and `FEASIBLE`. A solver that returns `FEASIBLE` after a 30-second time limit has not failed — it has found a solution and reports its objective value, but the search ran out of time before proving optimality. The objective value is the best-known; the next subsection describes how to assess how close that best-known might be to the actual optimum.

A typical branching pattern:

```
if status in (cp_model.OPTIMAL, cp_model.FEASIBLE):
    # We have a solution we can act on. Read it.
    ...
elif status == cp_model.INFEASIBLE:
    # The problem is over-constrained - the model and the data
    # together describe an impossible instance.
    ...
```

```
elif status == cp_model.MODEL_INVALID:
    # The model itself is broken. Read the log carefully.
    ...
else:  # UNKNOWN
    # The solver gave up. More time, more workers, or a tighter model.
    ...
```

`OPTIMAL` and `FEASIBLE` go to the same branch because the *answer* is usable in both cases — the difference matters for monitoring and reporting, not for acting on the result.

8.4.3 The bound: what optimality looks like

When an objective is set, the solver maintains two values. The *best known objective value* is the best feasible solution found so far. The *best objective bound* is a proven limit beyond which the objective cannot improve: for a minimization problem, a *lower bound* on the optimum; for maximization, an *upper bound*.

```
print(solver.objective_value)       # best-known solution (float)
print(solver.best_objective_bound)  # best proven bound (dual side)
print(solver.wall_time)             # seconds elapsed
```

When status is `OPTIMAL`, `objective_value` and `best_objective_bound` are equal (modulo floating-point representation): the solver has both found a solution and proven that nothing better exists. When status is `FEASIBLE`, the two differ, and the *gap* between them is what tells us how close the best-known is to provable optimality.

The standard relative-gap metric:

$$\text{gap} = \frac{|\text{objective\_value} - \text{best\_objective\_bound}|}{|\text{objective\_value}|}.$$

A gap of zero is a proof of optimality. A small gap (a few percent) means the best-known is within that fraction of the optimum, and continuing the search may not improve the answer much. A large gap means the bound is loose, the answer might be far from optimal, and either more time, a better model, or a stronger formulation is warranted. (The formula divides by `|objective_value|` and is therefore undefined when the objective evaluates to zero; the OR-Tools logs handle this by guarding the denominator, but in user code one usually special-cases the zero-objective case.)

If "close enough" is good enough for the application, the solver can stop on its own once the gap is below a threshold:

```
solver.parameters.relative_gap_limit = 0.01    # stop at 1 %
solver.parameters.absolute_gap_limit = 5       # or within 5 units
```

This is often a better fit for production than a fixed time limit, because it gives the solver permission to stop early on easy instances and keep working on hard ones.

`objective_value` is returned as a Python `float` even on integer-only models. Round it explicitly if the downstream code expects an integer.

8.4.4 What to do with each status

A short field guide that combines the status, the bound, and the action:

- `OPTIMAL` — done. Read the solution and act on it.

- `FEASIBLE` with a small gap — usually done. Read the solution; consider whether the gap matters for the application.
- `FEASIBLE` with a large gap — the model needs more time, a better formulation, or revisiting representation choices (Section 5.5) or the model's scope (Chapter 9).
- `INFEASIBLE` — the problem and the data, *together*, describe an impossible instance. Use the verifier from Chapter 4 to confirm: hand-craft a solution you believe should be valid and run it through the evaluator. If the evaluator accepts it, the encoding is wrong; if the evaluator rejects it, your understanding of the rules was wrong. Common fixes: relax a hard constraint into the slack-and-penalty form of Section 8.5, fix a data error, or accept that no plan exists for the given inputs.
- `MODEL_INVALID` — the model itself is broken. Read the log; look for variables with empty domains, expressions outside the supported integer range, or API misuses. (Logical contradictions between *constraints* surface as `INFEASIBLE`, not `MODEL_INVALID` — the latter is reserved for structural problems with the model itself.)
- `UNKNOWN` — the solver had no time to conclude. More time, more workers, or a smaller model.

8.5 Soft Constraints: Slack and Penalty

Some rules are preferred but not absolute — *total hours should not exceed the cap, but overtime is allowed at a cost*; *the schedule should respect preferences, but the schedule must be filled even if some preferences cannot be honored*. The standard CP-SAT pattern combines a relaxed constraint with a penalty in the objective:

```
overtime = model.new_int_var(0, max_overtime, "overtime")

# The hard cap is relaxed; overtime is the slack.
model.add(total_hours <= regular_cap + overtime)

# The penalty discourages - but does not forbid - overtime.
model.minimize(total_cost + 1000 * overtime)
```

The slack variable `overtime` measures *how much* the original cap would have been violated, and its coefficient in the objective is the per-unit penalty. The solver chooses on its own when paying for slack is cheaper than the alternatives — the optimization performs the trade-off.

This is the CP-SAT realization of the *hard versus soft* distinction from Section 4.3. A *hard* rule is encoded as a constraint with no slack; a *soft* rule is encoded as a constraint *with* slack, and the slack's coefficient in the objective is the rule's per-unit penalty. A model with only hard rules has a binary feasibility verdict; a model with soft rules has a richer answer that captures *how much* and *where* the rules were violated.

On the verifier side, the matching `Rule` reports its violation through `Metric.cost` rather than `Metric.violation` — slack semantics stay in lockstep on both sides.

The slack pattern also has a workflow purpose: it is the standard response to an `INFEASIBLE` status from Section 8.4 when the infeasibility is real and the rule is genuinely soft. Rather than refusing to plan, the model returns the *cheapest* violation of the soft rule that the rest of the constraints permit. This is often what is wanted in production — a binary "no plan exists" answer is rarely actionable, while "the cheapest plan violates rule X by 3 units at cost 3000" is.

Chapter 9

Conclusion

> *"In theory, theory and practice are the same. In practice, they are not."*
> — attributed to various sources

Four ideas run through this manuscript. *Notation is a tool, not gatekeeping* — mathematical notation gives a problem a precise shape that prose cannot, and reading it is a habit anyone working with optimization can pick up rather than a barrier put up by a community. *Optimization models decompose into five parts*: the entities the problem talks about, the parameters that describe an instance, the decision variables, the constraints that restrict their combinations, and the objective that ranks the feasible ones. *The verifier is orthogonal* to whichever solver eventually finds a solution — it does not help find one, but it pins the problem definition down in code and flags any candidate that breaks a rule, which protects every later modeling decision from drifting away from what was actually meant. *Knowing your solver's language matters* — CP-SAT has an expressive vocabulary (reified Booleans, `add_circuit`, `add_no_overlap`, `add_cumulative`, and the rest of Chapter 7), and an idiomatic CP-SAT model often looks different from the same problem expressed in a MIP solver or another constraint framework, even when the paper formulation is identical.

Two further moves come up often enough in practice to be worth naming, even if a thorough treatment is outside this manuscript's scope. *Trimming the model's scope:* not every dimension a problem carries needs to be a decision variable. Some "decisions" are downstream — they follow trivially from the genuinely hard ones and can be resolved by a post-processing step. An exam planner that fixes day and room and then packs the times greedily within each day is a typical example. Whenever a dimension is separable from the rest in this way, pushing it out of the solver shrinks the search space without changing the answer; the discipline is verifying that the decomposition really is lossless before committing to it. *Two models, one verifier:* a model that mirrors the paper formulation closely is sometimes too slow for the instances a project actually faces. Once the speed modifications grow beyond a tweak — alternative representations, symmetry breaking, problem-specific reductions — it can be cleaner to maintain a *faithful* model alongside a *tractable* one and let the verifier judge both on the same instances. Most projects never need this, and benchmarking should drive the decision rather than instinct.

CP-SAT itself covers a large slice of the discrete, combinatorial landscape, but not all of it: continuous variables push toward LP/MIP solvers (Gurobi, HiGHS, CPLEX), instances too large for exact methods push toward metaheuristics, and gradient-based optimization lives in another world entirely. What carries across is the part of this manuscript that is not CP-SAT-specific — the paper notation of Chapter 2 and the coded verifier of Chapter 4. A `Solution` and an `evaluate(instance, sol)` know nothing about which solver produced the input, and the habit of writing a yardstick cheaper than the artifact it judges transfers wherever optimization is done.