Exploring Rego

Declarative policy for the modern cloud

John Bristowe, Steve Fenton & Matt Allford

First edition

Octopus Deploy

Exploring Rego

Declarative policy for the modern cloud

John Bristowe,
Steve Fenton & Matt Allford

Exploring Rego

Declarative policy for the modern cloud

John Bristowe, Steve Fenton & Matt Allford

Hardback ISBN-9798189341762
Paperback ISBN-9798189345586

Ed.1.2026.1

PART I

Getting to know Rego

PART II

Variables, rules, comprehensions, and equality

PART III

Control and logic

PART IV

Built-in functions and code organization

PART V

Scaling up from working to production-ready

Appendices

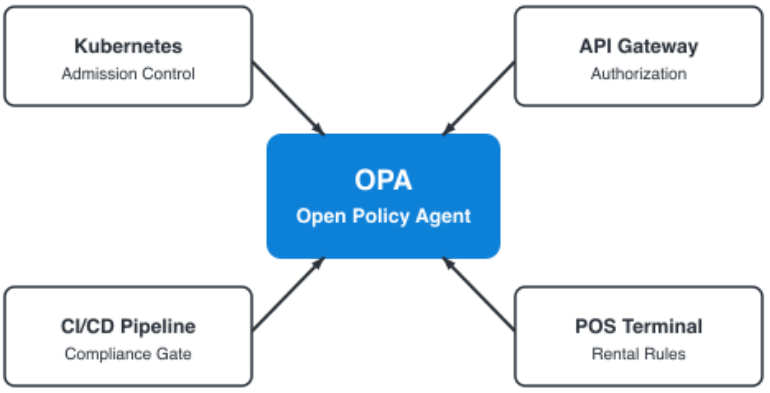
PREFACE

Rego rules OK

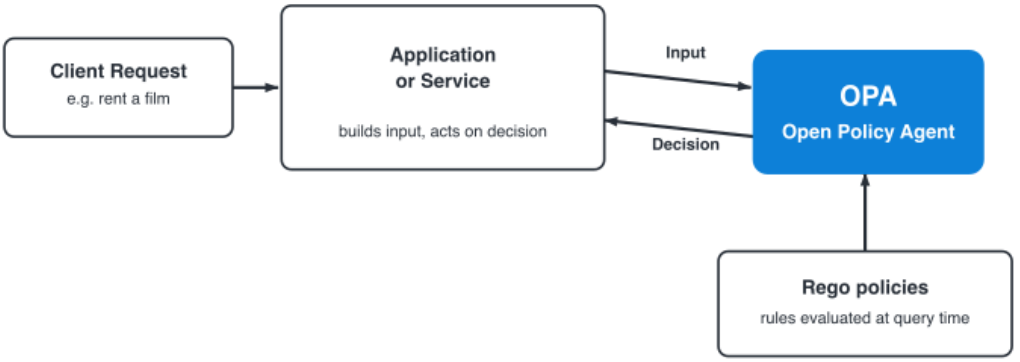
Any significant software system needs rules, and so do the pipelines that build and deploy it. You may have rules about allowable configurations, to govern transactions, or to make many of the decisions managed by modern software and the teams that build it. Traditionally, these rules are scattered throughout the application's code, buried among other logical routines. That makes it hard to find, change, and reason about them.

That's where policy-as-code comes in. These rules are so crucial that they deserve to be decoupled from the systems they control, made explicit, and stored in version control. They should be testable and easy to comprehend when you read their definitions.

Open Policy Agent, known as OPA (pronounced "oh-pah"), is the most widely adopted engine for evaluating policy-as-code. You can run it as a sidecar, a library, or a standalone service, and it answers one question: *given this context, what does the policy say?* Kubernetes clusters use it to validate admission requests. API gateways use it to authorize HTTP calls. Continuous Delivery pipelines use it to gate deployments. Anywhere a system needs to make a policy decision, OPA can be the engine that makes it.



Rego ("ray-go") is the language OPA uses to express those policies. It was inspired by Datalog, a decades-old logic programming language, and extended to work naturally with the JSON-shaped data that modern systems produce. Rego can look unfamiliar to developers trained in imperative languages, where code runs step-by-step with loops and conditions, creating different pathways with different outcomes. Instead, Rego lets you describe the end state, or *what should be true*, and OPA figures out whether it is. That shift in thinking is the real subject of this book.



Why we wrote this book

We're heavy users of Rego at Octopus, and also big fans of the language. Our policy engine uses Rego, and we've learned a great deal writing and maintaining policies. We can confirm it works at scale in production. This book distills that experience. You'll learn the patterns that work, the pitfalls that bite, and the mental model that makes everything click into place. What we've learned about Rego and the design of input schemas will help you avoid problems and succeed in your policy journey.

Who this book is for

We wrote this book for technically minded readers. Developers, platform engineers, security engineers, and anyone who writes code or configuration for a living. If you want to learn Rego properly, you've come to the right place. We have some expectations, for example, we assume you can read code, but we don't assume that you've ever seen a declarative or logic-based language before. We introduce everything in this book from first principles.

If you have already written a few Rego policies and found them confusing, this book will help. The language has a small number of core ideas, but they interact in ways that aren't always obvious. Working through them in order makes a significant difference.

Book structure

There are 5 parts to this book, each divided into short chapters. The chapter structure doesn't just organize the information; it creates natural breaks for you to grab a coffee and refuel for the next set of features.

Part I: We set the stage with an introduction to crucial concepts, the Rego type system, and simple policies that show you how Rego works.

Part II: You'll learn Rego variables, references, rules, and comprehensions. You'll also find out about the 3 equality operators that trip up almost every newcomer.

Part III: We'll take you deeper into control and logic, including negation, quantification, functions, and keywords that control rule evaluation.

Part IV: You'll get your hands on the built-in function library and see how you can organize policies into modules and packages. You'll also see how to add metadata annotations and JSON schemas to reduce errors.

Part V: Finally, you'll see what it takes to transform working policies into production-ready ones. That means writing tests, applying good practices, and using the linter.

You'll tackle things one small step at a time, because working in small batches is how you do big things. If you read the chapters in sequence, you'll find each step up is easily manageable.

Illustrating with examples

All programming books need examples, and that's what we'll set up next. To reduce the number of sample setups, we'll create one long-running example that you can re-use and modify as you work through the book. Reading example code can be a little dry, so we're going to take you back in time to make this more interesting.

You'll have to imagine some special video effects that transition us from the present all the way back to our nostalgic example. It's before streaming services, before the algorithms that decide what you should watch next, before pausing and resuming a television series on multiple devices. This is the era of the video rental store, where you had to walk the aisles, browse the empty display boxes, and take the ones you wanted to rent to the counter at the front of the store.

At the checkout desk, the video store employee made a series of decisions before hunting the drawers filled with rental tapes for you. *Are you a member? Have you returned your last tape? Have you paid your late fees? Are you old enough for this movie?* It's these decisions that form a policy, and they can be more structured than you might first think.

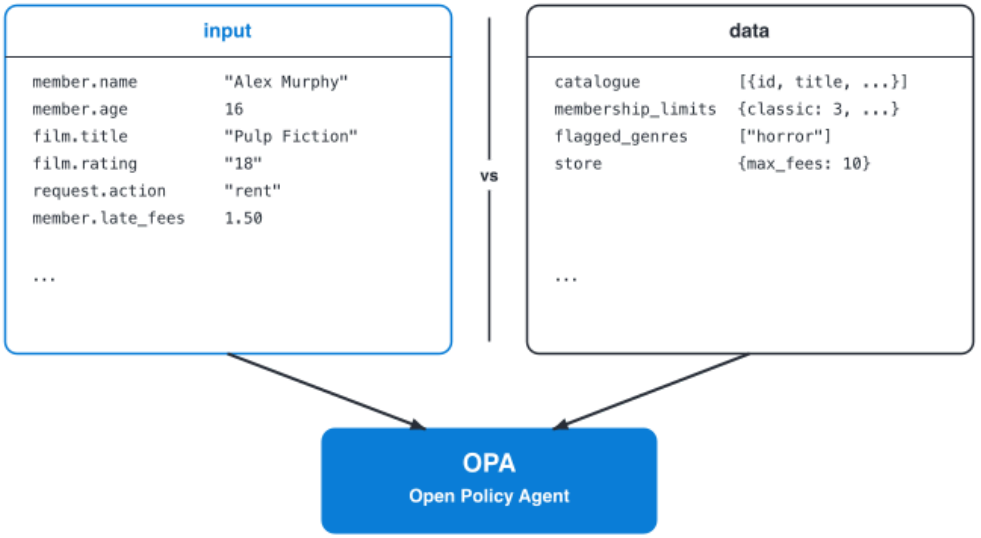
Throughout this book, we will make use of the fictitious bricks-and-mortar store, EmptyBox Rentals, as a running example. We'll transport you back to 1995 (cue those video effects), 10 years after Marty McFly went on his own time-traveling adventure. There are only five billion people on Earth, and everyone is wearing flannel shirts and listening to rock music. At least, that's what I remember.

Our story concerns a 16-year-old movie fan, Alex Murphy. Alex is a young but loyal customer of EmptyBox rentals with two rentals at home, including an overdue copy of *The Lion King*. Alex is back in the store and looking for something more exciting (with a little less singing). Alex has just dropped the display box for *Pulp Fiction* on the counter, and the employee needs to decide whether to allow the rental.

Since the store's computer is an old PowerBook 140, validating the request takes a long time for two crucial reasons. First, you have to load all the data from a floppy disk; second, nobody has created the software or collected the data to support the video store. That means each employee has to remember the rules, and that often isn't a great way to consistently apply policies. That's all about to change as you're going to supply EmptyBox rentals with a futuristic, polished aluminum box that runs Open Policy Agent with Rego.

Along the way, you'll discover that this apparently simple check involves membership validation, age restrictions, stock availability, overdue detection, late fee thresholds, and even opening hours. When you're done, you'll have a well-structured, thoroughly tested, and production-ready policy. When you return to current times, EmptyBox Rentals will have gone out of business, but the Rego patterns you created will still be running.

The contextual information about the customer, their account, and their request to rent a movie make up the policy's *input*. It's a JSON object, even though Douglas Crockford won't invent JSON until 2001. There will also be static data about the store and the videos in stock, which is in another JSON object called *data*.

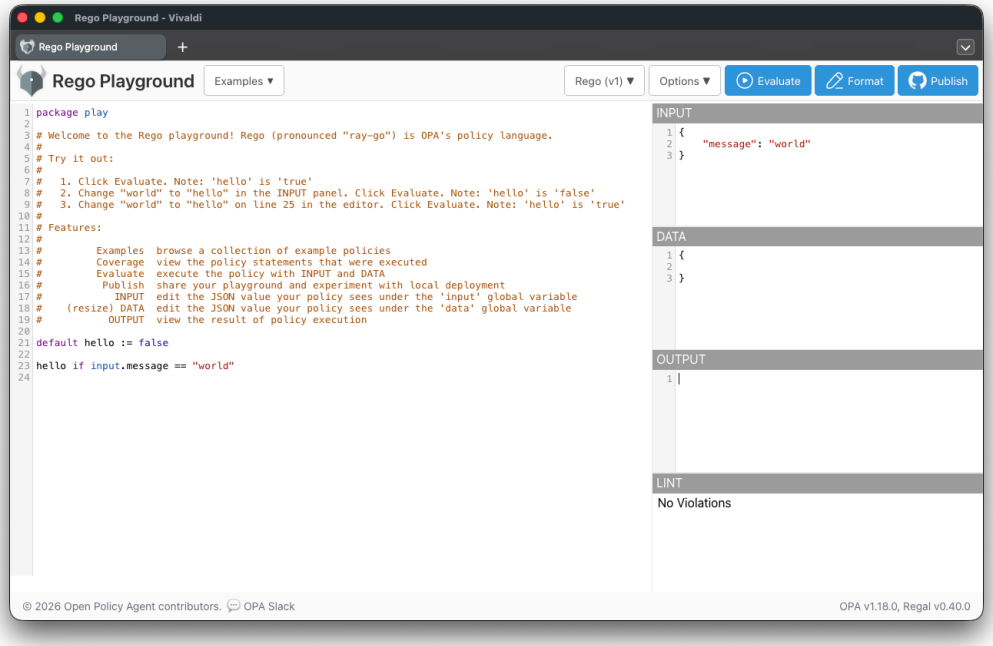


For most of the book, Alex and the copy of *Pulp Fiction* he wants to watch form the example that demonstrates the Rego language. As the book progresses, the policies governing this scenario become more sophisticated. By the end, they will use schema validation, metadata annotations, layered packages, and a full test suite. The scenario stays constant; the language features that express it accumulate chapter by chapter.

Using the playground

You can follow along with the examples and try out your own policy code in the Rego playground. You can use the playground to test policies against test input and data, so it's a quick way to experiment with the language.

play.openpolicyagent.org



The Rego Playground has a code editor on the left. That's where you'll place Rego policy code as you work through the examples. It also has panels for **INPUT** and **DATA**. The pre-populated playground link will set these based EmptyBox Rentals and Alex's rental request. When you evaluate a policy, the result will be shown in the **OUTPUT** panel. The **LINT** panel will highlight any issues with the Rego code. These can usually be resolved with the **Format** button.

When you are ready to work locally, the OPA CLI is a single self-contained binary available for Linux, macOS, and Windows. You can find installation instructions in Appendix D. The CLI adds a REPL, a test runner, a linter, and a bundle builder. You'll use these tools in the later chapters.

Let's begin

You'll learn Rego faster if you can get your hands on. Each chapter in this book has examples you can use and adjust to see how the lanaguage works. By the end of the book you'll be able to read and understand Rego policies and confidently write new ones. You'll also learn how to structure them and test them, so they are robust for use in production. Flip the page, open the playground, and let's write some policy code!

Part I

Getting to know Rego

CHAPTER 1

Thinking declaratively

Alex Murphy slides an empty video case across the counter of EmptyBox Rentals. The *Pulp Fiction* videotape is stashed with hundreds of other titles in the large drawers behind the employee, but Alex is sixteen, and the film is rated 18. The employee at the checkout desk has to decide if Alex can rent the movie, and everyone in line is waiting.

Making that decision is what this book is about, and before you write Rego to decide the answer, you need one mental shift: Rego is a *declarative* language. You describe *what should be true* for a rental to be allowed, not the step-by-step procedure that computes the answer.

This chapter will steer you away from patterns and practices you've likely picked up from modern programming languages and shows you how Rego works differently. It covers how OPA uses the policies you write, and how you can use `input` and `data` in your rules. There's also a crucial lesson on why something that isn't true isn't necessarily false, which is one of the most important concepts in Rego.

Alex will wait patiently at the counter while those ideas land, while the customers queued behind him check their Swatch watches and wish that smartphones had been invented to fill the boring delay.

The problem at the counter

To determine if Alex can fulfill his dream of watching the hit Tarantino movie, the store employee must remember to run through a set of standard checks:

- Is Alex an active member?
- Is the film in stock?
- Is Alex old enough for this film's rating?

That's only three questions, and more could be required: overdue tapes, late fees, how many rentals are already out, and, when click-and-collect arrives, whether the store is still open. You'll add all of these as you make your way through the book. For now, these three checks are enough to get started in Rego and understand why the way you write the policy can matter just as much as the policy itself.

How you probably think about code

If you have written Python, JavaScript, Go, or most mainstream languages, you are used to *procedures*: step-by-step instructions that tell the machine what to do, and in what order.

Here's how you might write the basic rental check for Alex in Python:

```
def can_rent(member, film):
  if member["age"] < int(film["rating"]):
    return False
  if not member["membership"]["active"]:
    return False
  if film["available_copies"] < 1:
    return False
  return True
```

The code is readable, though you have to create a mental map of whether subsequent lines are reachable, as earlier conditionals may have returned a result. The code works and does what we required, as it only allows a rental if all checks pass.

This is imperative programming: a series of steps that lead to a result. You logically branch at each check, creating many pathways through the code. Imperative programming mixes two concerns when it comes to policy: the rules and the mechanics. When you add more rules, things get too complex for that mental mapping task, and it becomes an exhausting task to work out what's happening.

When the manager adds a new rule, you have to place it exactly where it needs to be to ensure it takes effect. After a while, you open the function to find you have 20 code branches, and nobody really knows how it works without adding a breakpoint and firing up the debug tools.

Declarative policy

Rego, like other declarative language, flips the decision around. Instead of describing the steps taken to determine an answer, you describe the conditions under which the answer is true. You create a picture of what "yes" looks like, and OPA compares that picture to reality and answers the question.

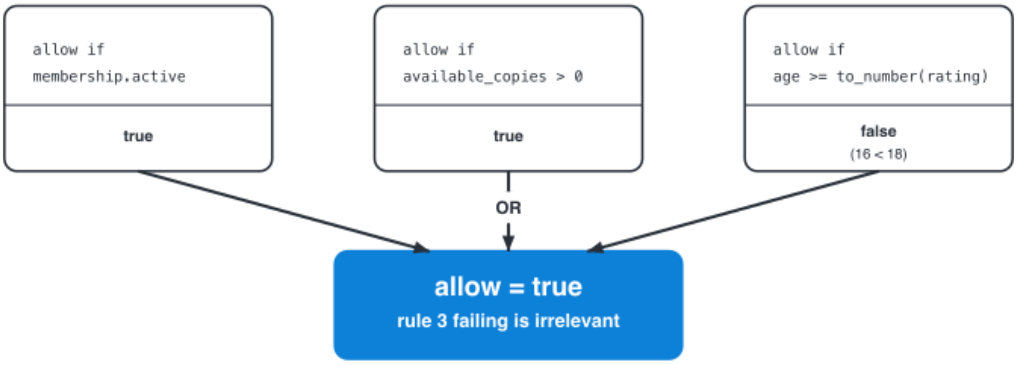
Let's try this out using the Rego Playground. Use this link to open a pre-populated playground: **oc.to/rego-playground**. You can then replace the policy panel with the example below to try it out. Let's create a policy to handle EmptyBox Rental's basic checks, handling active membership, the availability of the film, and the member's age:

```
package rental

default allow := false

allow if input.member.membership.active
allow if input.film.available_copies > 0
allow if input.member.age >= to_number(input.film.rating)
```

Add this to the left-hand code panel in the playground and use the "Evaluate" button to run it. You'll get the following result:

```
{
    "allow": true
}
```

It's not a bad attempt, but it's not quite right. When you place multiple `allow if` rules in a policy like this, they work like a logical OR. That means an active membership is enough to allow the rental, because the age check doesn't need to pass for "allow" to be true if at least one of the other checks passes.

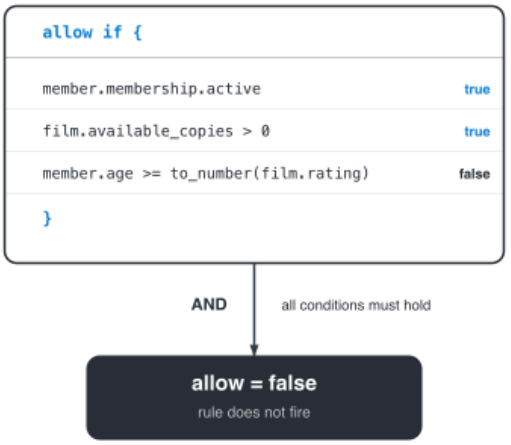


Active membership alone is enough, the age check has no veto.

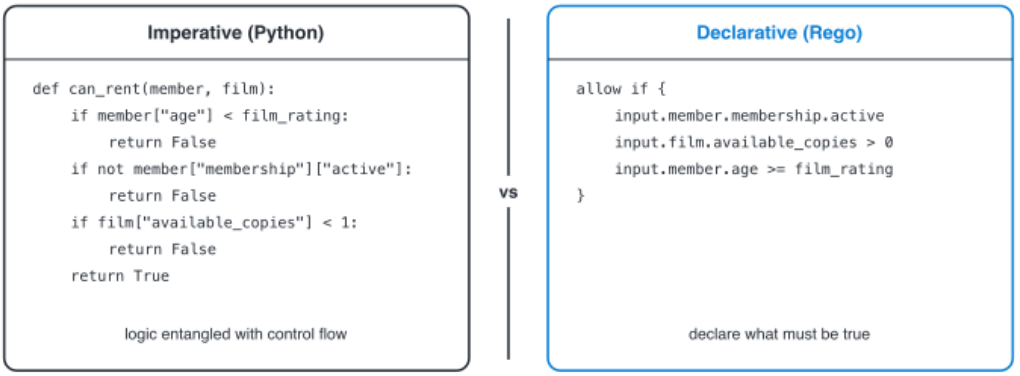
That's definitely not how it should work. Instead, the rental should be allowed only if all three checks pass, more like a logical AND. You define this in Rego by placing the conditions inside a single `allow if` block.

```
package rental

default allow := false

allow if {
 input.member.membership.active
 input.film.available_copies > 0
 input.member.age >= to_number(input.film.rating)
}
```

This policy will provide the expected answer. The rental isn't allowed. All of the rules must *hold* (which means the rules are successfully met) for "allow" to be true, so Alex will have to wait a couple of years before watching *Pulp Fiction*.



You may have noticed that there are no `return` statements in this example. You also don't specify the order because it doesn't matter. You can move the rules around, and OPA will require all of them to hold. Rego policies don't describe how to make a decision. Instead, they describe what reality must match for a decision to be reached. You supply one or more examples of situations where a movie rental would be allowed, and the policy engine works out whether any of those situations holds.



If you evaluate the updated policy in the Rego Playground, you'll get the result:

```
{
    "allow": false
}
```

While Alex has an active membership and the movie is available, Alex isn't old enough to watch it. You can change Alex's age in the input panel and re-evaluate the policy to see a successful response, but don't forget to restore the input to its original state, so the next example works as expected.

> You may have noticed the Rego code begins with `package rental`. For now, think of this like a namespace. You'll read more about packages and modules in Chapter 2.

Adding new rules

The EmptyBox manager came up with an idea earlier to let premier-tier customers rent movies without an age check. We can add this new rule without touching the existing one.

```
allow if {
 input.member.membership.active
 input.film.available_copies > 0
 input.member.membership.tier == "premier"
}
```

Premier members must still have an active subscription, and the movie must be available, but there is no age check, so if Alex has a premier-tier membership, he gets to watch *Pulp Fiction*. By leaving existing rules alone when you extend the policy, you are less likely to break them. Because Rego policies describe conditions, not procedural steps, OPA can analyze them, perform structural checks and optimizations, and provide clearer failure modes.

Rules as assertions

You can think of each rule as an *assertion*. It's a claim that something is true under certain conditions.

Take the following example:

```
allow if input.member.membership.active
```

You can say it aloud as "I assert that `allow` is true whenever the member's membership is active."

One crucial building block of these policies is that when the condition isn't met, the rule doesn't fire. That means the rule doesn't return "true or false"; it returns "true" or does nothing at all. Keeping each set of conditions in a separate rule makes them easier to reason about, so don't over-optimize for duplication in your policies and instead optimize for easy understanding of each rule. Rules also have no sequence. They are parallel facts about the world.

Your first policy, end to end

To put this model to work, open the pre-populated playground (**oc.to/rego-playground**) and enter this policy. Remember, you can use this link to get back to the baseline example whenever you need to.

```
package rental

default allow := false

allow if input.member.membership.active
```

When you evaluate the policy, you'll get the result:

```
{
  "allow": true
}
```

In the **Input** tab, change the membership active flag to false:

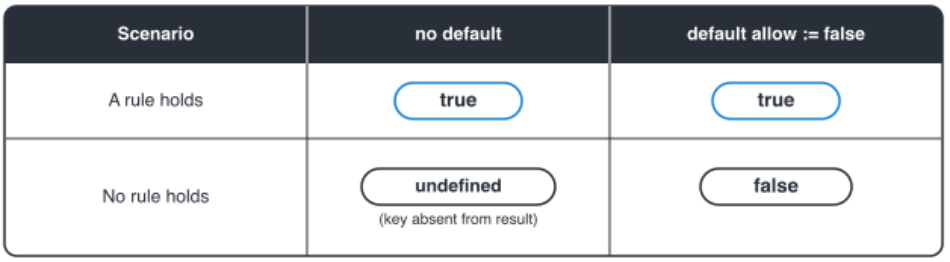
```
    ...
    "membership": {
      "active": false
      ...
```

Re-evaluate the policy to see the new result:

```
{
  "allow": false
}
```

The `default` keyword provides a value to use when no rule holds. Without this, `allow` wouldn't even appear in the result. `default allow := false` means: unless a rule proves otherwise, treat `allow` as false. There's more information on `default` in chapter 11, but for now you can read it as "unless there's a rule that says otherwise, we'll deny the request".

Remove the `default` line and evaluate with the membership in the input panel set to `active: false`. The output is now `{}` not `{ allow: false }`. In fact, it's not there at all. *Undefined* versus *false* is one of Rego's central ideas; you will meet it again throughout the book.



Undefined is not false. Without a default, a failing policy returns no result at all.

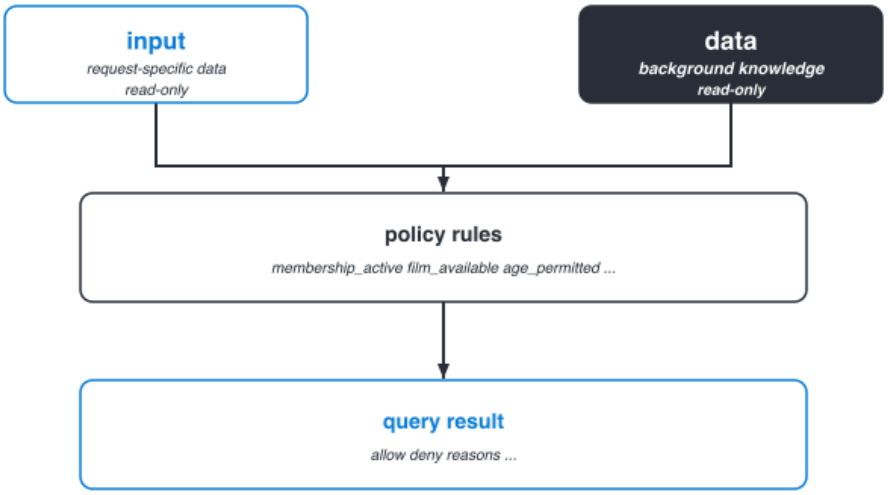
The OPA document model

There are two special documents used in every OPA evaluation.

The **input** document is request-specific. That's where Alex, his account, and his rental request are supplied. Policies can *read* from input, but they can't change it.

The **data** document is background knowledge that doesn't change for each request. For EmptyBox Rentals, this might include the movie catalog and store opening times.

Your policy reads `input` (and `data` when needed) and produces a result. OPA evaluations have those documents in scope and return whatever your query produces, usually `allow`, and later `deny`, along with reasons for the decision.



Policies can read from input and data but cannot modify either document.

The input we will use throughout this book

When you used the short link to open the playground, you may have noticed that the `input` and `data` boxes were pre-completed for you. You'll use these documents throughout the book, so it's worth familiarizing yourself with each document's contents. You can load a fresh playground at any time by visiting **oc.to/rego-playground** to load the input and data documents.

Input

Alex is sixteen and wants to rent *Pulp Fiction*, which is rated 18. He already has two tapes out (including an overdue rental of *The Lion King*) and owes 1.50 in late fees. Here's a breakdown of the `input` document with this information. We'll split it up for readability here, but it will all be combined into one big document in the playground.

Alex's membership information

Everything you need to know about Alex, like his id, name, and age, and details of his membership and account.


```
{
  "member": {
    "id": "MBR-0042",
    "name": "Alex Murphy",
    "age": 16,
    "membership": {
      "tier": "classic",
      "active": true,
      "joined": "1993-09-15T00:00:00Z",
      "fees_outstanding": false
    },
    "late_fees": 1.50
  },
  "membership_limits": {
    "classic": 3,
    "plus": 5,
    "premier": 10
  }
}
```

The films Alex has at home

All the rentals Alex has taken home that haven't yet come back into the store, and details about his rental limits.

```
{
    "rentals": {
      "current_count": 2,
      "max_allowed": 3,
      "items": [{
          "film_id": "VHS-0871",
          "title": "Speed",
          "due_date": "1995-11-08T00:00:00Z",
          "overdue": false
      },{
          "film_id": "VHS-1204",
          "title": "The Lion King",
          "due_date": "1995-11-06T00:00:00Z",
          "overdue": true
      }]
    }
}
```

Alex's rental request

The request Alex is making to take another rental home.

```
{
  "film": {
    "id": "VHS-2209",
    "title": "Pulp Fiction",
    "director": "Quentin Tarantino",
    "year": 1994,
    "rating": "18",
    "genres": ["crime", "drama"],
    "available_copies": 1,
    "rental_price": 3.50,
    "rental_period_days": 3
  },
  "request": {
    "action": "rent",
    "store": "emptybox-central",
    "timestamp": "1995-11-09T18:30:00Z"
  }
}
```

Data

You also have access to some store data, including the movie catalog and store information. You can see that each movie includes information on genres, age ratings, and rental prices.


```
{
  "catalogue": [{
    "genres": [
      "crime",
      "thriller"
    ],
    "id": "VHS-0871",
    "rating": "12",
    "rental_price": 2.75,
    "title": "Speed"
  },{
    "genres": [
      "animation",
      "children"
    ],
    "id": "VHS-1204",
    "rating": "U",
    "rental_price": 2.75,
    "title": "The Lion King"
  }
  ...
  ],
  "flagged_genres": [
    "horror"
  ],
  "store": {
    "max_late_fees_before_block": 10
  },
  "membership_limits": {
    "classic": 3,
    "plus": 5,
    "premier": 10
  }
}
```

You can keep the scene in mind as you read through the following chapters. Alex is tentatively holding the empty box for *Pulp Fiction* and has an overdue videotape on his account. As we introduce new constructs, consider how they may change what happens next as Alex stands before the employee at the counter.

What's next

You've evaluated your first policies and familiarized yourself with the Rego Playground. Hopefully some of the mystery surround policies is now gone and you're ready for more. Rego policies describe *what is true*, not *how to compute truth*. Each rule is an independent assertion, and if none of them fire, you get nothing (not false) unless you specify a default. OPA evaluates your assertions against the `input` and `data` documents.

In the next chapter, you'll write some policies from scratch in the playground. Grab yourself a tea or coffee, flip the page, and let's put Rego to work.

Key concepts

- Instead of specifying steps to reach an answer, Rego lets you illustrate what an answer looks like.
- Each rule in your policy is independent and multiple rules can contribute to the same result.
- You might determine something is `true` or `false`, but you might not determine it at all, in which case it will be `undefined`.
- You can use the `default` keyword to provide a fallback when you don't want something to be undefined.
- Policies get input and data that can be used to make a decision.

CHAPTER 2

Your first policies

The previous chapter explained the declarative mindset. You describe what something should look like to be true, and OPA determines how to reach the answer. You learned how to pass information to OPA as input and data, and how it can use many independent rules to produce a result (or no result if none fire). You also saw how you could provide a default to avoid undefined results.

In this chapter, you'll build further on the basic counter checks you used before and find out more about the structure of a Rego project.

Modules and packages

In Rego, a *module* is a file (or a string of text, in programmatic contexts) that holds your policy code. Every module must begin with exactly one `package` declaration.

```
package rental
```

A *package* is a namespace. It groups related rules and prevents name collisions. You've defined an "allow" name to describe if a rental should be allowed. However, there may be many other policies that allow or deny something, so you don't want the policy that allows staff breaks to get confused with the policy that allows rentals. Placing the rental rules in a package called "rental" means your "allow" rule won't conflict with any other rules with the same name elsewhere.

Package names can be simple identifiers or dotted paths:

```
package rental
package rental.age
package store.inventory
```

Dotted names create a hierarchy, so `rental.age` is a child of `rental`. That makes a meaningful difference when you're organizing large policy codebases, as you'll learn in chapter 13. For the examples in this chapter, a single flat name is sufficient. While the package name doesn't have to match the filename, the convention in Rego is that it should. A file named `rental.rego` should contain `package rental`.

Writing your first rules

With `package rental` in place, you can name facts the employee already knows, like store name, rental period, and whether the branch is open. Other rules can refer to them by name instead of repeating literals. Assigning a value to a name uses the `:=` assignment operator.

```
package rental

store_name := "EmptyBox Central"
```

We have defined a `store_name` with the value "EmptyBox Central". If you evaluate this in the playground at **play.openpolicyagent.org**, you'll get the following result:

```
{
  "store_name": "EmptyBox Central"
}
```

That is a rule in its most basic form: a name, the `:=` assignment operator, and a value.

Rules can hold any value Rego supports, like strings, numbers, booleans, or null:

```
package rental

store_name      := "EmptyBox Central"
max_rental_days := 7
rental_price    := 3.50
is_open         := true
head_office     := null
```

You can define constants in many languages, and that's really what you've done here, though you'll notice they get called rules, just like everything else. You can't change the value assigned to a rule later, as it would trigger a rule conflict, which happens when two complete rules with the same name produce different results. You can use these constants to define values once, instead of repeating them many times in your policy.

Rules with bodies

A membership check is not always true. Alex's account might be inactive, or there might be outstanding fees. A rule with a *body* expresses "this is true only when all of these conditions hold":

```
package rental

membership_valid if {
 input.member.membership.active
 not input.member.membership.fees_outstanding
}
```

Everything inside the curly braces is part of the body and each item must hold at once for the rule to be true. In imperative languages, you'd use a logical AND for this. If the body holds, the rule produces its value. In this case, "true", which is the default when no value is specified. If any condition fails, the rule produces *nothing*. Not `false`. Nothing. We'll repeat this often because it's a crucial Rego concept!

In older versions of Rego, you could omit the `if` keyword, but it's now required before the rule body. If you forget, the parsing error is very clear that you need to add it. Using the `if` keyword makes your rule "read out loud", for example "membership is valid if all the things in the curly braces hold".

Load up the example in the Rego Playground (**oc.to/rego-playground**). Then add the policy:

```
package rental

membership_valid if {
 input.member.membership.active
 not input.member.membership.fees_outstanding
}
```

Alex's has an active membership as part of the information in the input panel, so when you evaluate the policy, you'll get the following result:

```
{
  "membership_valid": true
}
```

Update the input panel to set `membership.fees_outstanding` to `true` and evaluate the policy again. The second condition in the rule body fails, and the query returns nothing:

```
{}
```

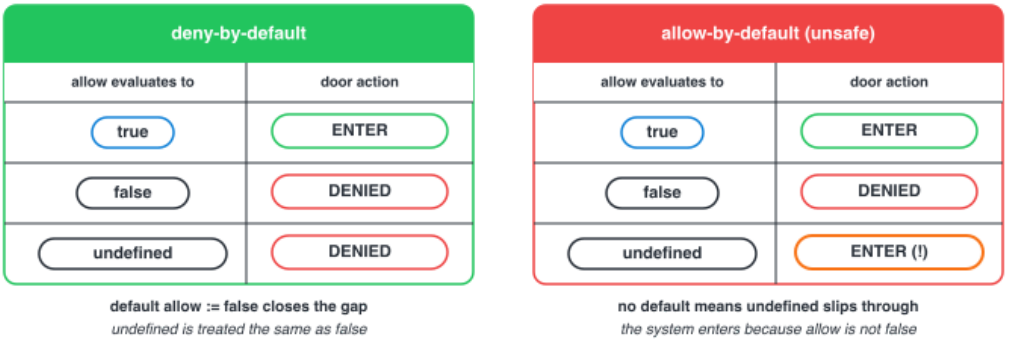
The input no longer matches the shape of a valid membership in the policy, so the rule doesn't fire.

Understanding undefined

This is one of the most important ideas in Rego, so it deserves careful attention. In most languages, a boolean function returns `true` or `false`. In Rego, a rule can return there are three possible outcomes: `true`, `false`, and *undefined*. When the answer is undefined, it doesn't appear in the result unless you have provided a default. Let's use an example to understand why that's important.

If you have a secure door that only allows entry when "allow" is true, it doesn't matter whether "allow" is false or missing. But if you allow people to enter unless "allow" is false, you could be letting unrecognized people through the door. That means undefined and false aren't interchangeable, and you need to know what you intend in each case.

Understanding when a rule is undefined versus when it explicitly returns `false` is essential to writing policies that behave correctly in all cases. The general principle is: use `default` for rules that should always produce a value. Omit it when undefined is a meaningful, intentional state.



Undefined and false are different. Use default to ensure a value is always produced.

Building a real policy

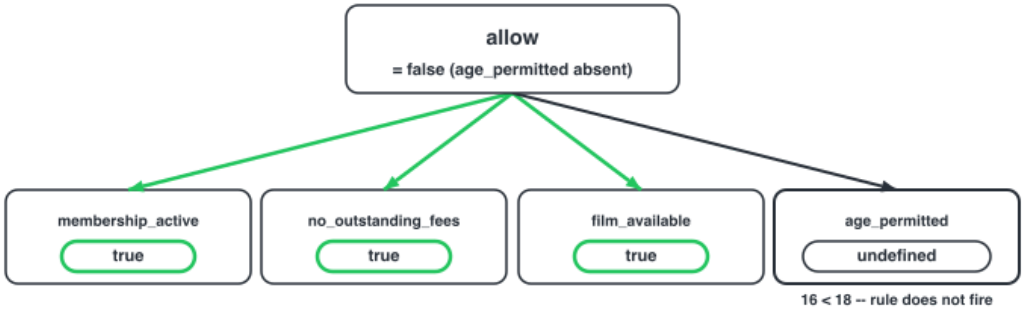
You now have some helpful concepts under your belt, so let's put them to work. Read the policy below, which defines whether a movie rental is allowed. Alex is waiting to find out the answer to this very question. The rule body doesn't define the conditions directly, but references individual named rules. This makes the policy highly readable.

```
package rental

default allow := false

allow if {
  membership_active
  no_outstanding_fees
  film_available
  age_permitted
}

membership_active if input.member.membership.active

no_outstanding_fees if not input.member.membership.fees_outstanding

film_available if input.film.available_copies > 0

age_permitted if input.member.age >= to_number(input.film.rating)
```

Add this policy to the playground (**oc.to/rego-playground**) and evaluate it to see the outcome. You'll get the final results, and also which rules fired:

```
{
  "allow": false,
  "film_available": true,
  "membership_active": true,
  "no_outstanding_fees": true
}
```

You can see the rental isn't allowed and also which rules passed. The film is available, the membership is active, and there are no outstanding fees. Absent from this list is `age_permitted`. That's because the rule didn't fire, because Alex is 16 and needs to be 18 to watch *Pulp Fiction*. Because there is no default for `age_permitted`, it will only appear in the result if the rule holds.



Three helpers fire and produce true. age_permitted does not fire. Because all four conditions must hold, allow remains false.

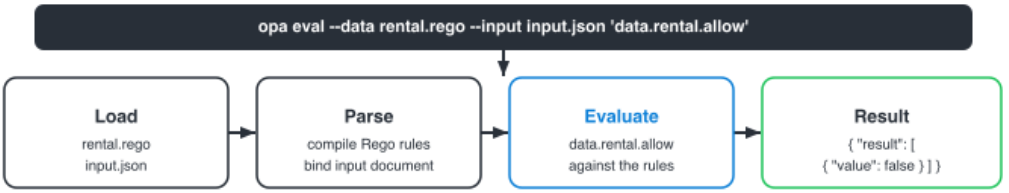
In the input panel, change Alex's age temporarily to `18` and evaluate the policy once again. This time, the rental is allowed and you also see that `age_permitted` is in the result.

```
{
  "age_permitted": true,
  "allow": true,
  "film_available": true,
  "membership_active": true,
  "no_outstanding_fees": true
}
```

The policy states what must be true for rules to be met and the result depends on whether the input matches those rules. That's declarative programming in action.

Using the OPA CLI

The playground is ideal for learning, but day-to-day work usually happens on the command line. Once you have OPA installed (see Appendix D), the two commands you will use most are `opa eval` and `opa run`.



Use --format pretty for concise output, or --explain full to trace rule evaluation. opa run opens an interactive REPL where you can load policies and run queries live.

opa eval: Evaluates a query against a policy file and optional input An example command is shown below:

```
opa eval \
  --data rental.rego \
  --input input.json \
  'data.rental.allow'
```

Create a file named `rental.rego` and add a short policy:

```
package rental

default allow := false

allow if input.member.membership.active
```

Now create a second file named `input.json` and add the following data:

```
{
  "member": {
    "membership": {
      "active": true
    }
  }
}
```

When you run the `opa eval` command and pass these files, you'll get the following result:

```
{
  "result": [{
    "expressions": [{
      "value": true,
      "text": "data.rental.allow",
      "location": { "row": 1, "col": 1 }
    }]
  }]
}
```

You can reduce the noise in the command's response using the `--format pretty` flag, which gives more readable output:

```
opa eval \
  --data rental.rego \
  --input input.json \
  --format pretty \
 'data.rental.allow'
```

```
true
```

opa run: starts an interactive REPL (read-eval-print loop) where you can load policies and input documents and run queries interactively:

```
opa run
```

The REPL lets you interact with OPA in a live session. For example, type `x := "Hello"` in the REPL and you'll see a message that "Rule 'x' defined in package repl". You can type `show` in the REPL to see what you've created so far.

In this case you'll see:

```
package repl

x := "Hello"
```

When you're finished with your interactive session, type `exit` to end it.

A note on the ".rego" extension

Rego policy files use the `.rego` extension. OPA tools look for this extension when loading policies from directories. Some editors and linters (such as Regal) use it to apply Rego-specific rules. Always name your policy files with `.rego`.

What comes next

You now know about the structure of Rego module and some tricks to make your policies easy to read. Readability should be high on your list of priorities when you author policies. We re-visited the crucial concept of undefined values and why they might cause an issue (with the security door example). You also know how to run policy evaluations using the playground, or using the CLI. You're well on your way to becoming a Rego pro.

Next, we'll look at *values* and the different types of value that Rego supports. You'll find out how to construct them and when to use each one. Before you move on, feel free to experiment with the example in the playground to see what happens if you adjust a policy or change the input.

Key concepts

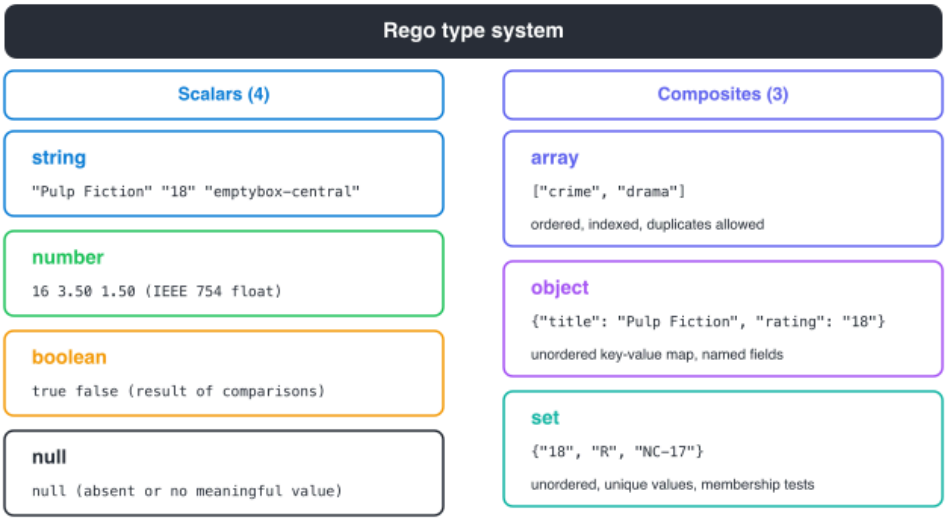
- Each Rego file is a module that declares exactly one package
- The package name is the namespace that prevents naming collisions
- Rules can be simple value assignments, conditional bodies, or both
- Undefined doesn't mean "false" and you should use a default when a value should be produced even if no conditions are met

CHAPTER 3

Values and types

So far, you've learned to structure a module, write conditional rules, and reason about undefined versus `false`. You are probably getting quite familiar with the data about Alex and the film he wants to rent; the people behind him sure know about it. The playground examples rely on Alex's rental *input*, a JSON document containing different data types. Before you add more rules to the rental policy, let's take a look at the handful of types used in Rego. In your policies, you'll want to avoid confusing `"18"` with `18`, for example. Mistakes with types are a common source of silent bugs.

This chapter walks through scalars and composites in the order they appear in Alex's record, so you know which literal or collection form to reach for in the rules that follow. Rego uses *scalars* and *composites*. Scalars are single indivisible values, like a number or a string. Composites are collections built from these simpler values, or even from other composites. For example, your address is a composite made from a number (house number) and some strings (street and town).



Scalar values

There are four scalars in Rego's type system: strings, numbers, booleans, and null. You can find examples of these in the input panel containing Alex's rental request.

Strings

Film titles, member names, and age ratings all arrive as text. In Rego, a string is a sequence of Unicode characters wrapped in double quotes.

```
package rental

film_title := "Pulp Fiction"
director   := "Quentin Tarantino"
rating     := "18"
store_id   := "emptybox-central"
```

You'll need to escape some special characters if they occur in a string. Your JSON serializer will usually take care of this, but if you're hand-cranking a document, you'll need to escape double quotes `\"`, new lines `\n`, and the backslash itself `\` (otherwise it looks like you're trying to escape the character *after* the backslash). The member IDs at EmptyBox follow a pattern of letters and numbers. If you ever use a regular expression to validate this pattern, you'll likely find the expression is full of special characters. As regular expressions are already tricky to read, escape sequences are incredibly unhelpful.

If you have a string like a regular expression that would be more readable without escape sequences, you can use a Rego *raw string*. These use backticks instead of double quotes:

```
package rental

# Double-quoted strings require escaped backslashes
id_pattern_escaped := "MBR-\\d{4}"

# Raw strings can be more readable
id_pattern_raw := `MBR-\d{4}`
```

The values in `id_pattern_escaped` and `id_pattern_raw` are the same. The raw form is easier to read and less error-prone because it doesn't need as many escape characters .

String interpolation

When a store employee rejects a rental request, they don't usually shout "FALSE" and usher the customer away. A better way to handle a policy failure is to explain why Alex can't rent the film. For example, the employee might say, "I'm sorry, Alex, but you're not old enough to rent this film. Can I interest you in some popcorn and a copy of Aladdin?"

You can create helpful messages in Rego with string interpolation, which allows a template to be combined with contextual values. Template strings use a `$` before the first double quote, with each contextual value wrapped in curly braces.

```
package rental

member := input.member
film := input.film

denial_message := $"{member.name} is not old enough to rent {film.title}"
```

If you add this to the pre-populated playground (**oc.to/rego-playground**) and evaluate it, and you should see:

```
{
  "denial_message": "Alex Murphy is not old enough to rent Pulp Fiction"
}
```

The contextual values can come from any expression that produces a value, like variables, references, arithmetic, or function calls.

```
package rental

film := input.film

rental_summary := $"Renting {film.title} for {film.rental_period_days} days"
```

With the above example, a summary of the rental request will be produced, like a receipt for the request. You can create summaries, rejection reasons, or other useful information with this technique.

When you use an undefined value with string interpolation, the text `<undefined>` will appear. In other languages, this might cause an exception to be thrown.

```
package rental

# Will be: "film <undefined> not available"
unavailable_message := $"Film {input.film.missing_field} not available"
```

Because curly braces are used to bring in contextual information, you need to escape them if you want to use them in an interpolated string using `\{`.

Numbers

There are several examples of numbers in the rental request. Alex is `16`. The rental is `3.50`. His late fees are `1.50`. Rego represents all of these with an IEEE 754 floating point number, though whole numbers behave like integers in practice:

```
package rental

member_age       := 16
max_rentals      := 3
rental_price     := 3.50
late_fee_per_day := 0.50
```

When you work with numbers, you can use standard arithmetic operators: `+`, `-`, `*`, `/`.

To perform integer division, you can use the built-in function `div` and use `%` to get the remainder. There's more on arithmetic built-ins in Chapter 12.

```
package rental

start_number := 5

added       := start_number + 3
subtracted  := start_number -1
multiplied  := start_number * 2
divided     := start_number / 2
```

When you run this in the Rego playground, you'll see the following result:

```
{
  "added": 8,
  "divided": 2.5,
  "multiplied": 10,
  "start_number": 5,
  "subtracted": 4
}
```

The rental example contains a special case. A film's rating is stored as a string. That's because films can be rated with a mixed type: "U", "PG", "12", "15", and "18".

That means it needs to be converted to a number if we need to compare it to Alex's age. The built-in `to_number(input.film.rating)` does this.

Booleans

Membership checks and stock checks boil down to yes-or-no answers. The film is either available or it's not. Rego represents those outcomes as *booleans*, which have values `true` or `false`.

```
package rental

# true
membership_active  := input.member.membership.active

# false
fees_outstanding   := input.member.membership.fees_outstanding

# true
film_available     := input.film.available_copies > 0
```

You will use booleans constantly as rule values. They are also what you get back from comparisons, like `input.member.age >= 18`. Although a boolean can only be true or false, remember that if a rule doesn't fire, you might not get a value back at all, and will need to handle the undefined case.

Null

A `null` means there's no meaningful value available. In Rego, you're more likely to receive nulls in the input and data documents than you are to create one from scratch.

```
package rental

no_head_office := null
```

You rarely assign `null` yourself. Its everyday role is something you *test for* to see if fields in the input or data documents are missing a value.

Composite values

If you look at the input document for Alex's rental request, you'll see it has a structure. There's a list of rentals, and the member's information is a nested object. There are three composite shapes: arrays, objects, and sets. Making the right selection can help keep your policies easy to read and faster to evaluate.

Arrays

When the employee asks "What films does Alex still have booked out?", the answer is an ordered list. In Rego, that shape is an *array*, which is an indexed sequence that can hold any mix of types, including other composites:

```
package rental

genres     := ["crime", "drama"]
film_ids   := ["VHS-0871", "VHS-1204", "VHS-2209"]
mixed      := [1, "two", true, null]
```

Elements in an array are zero-based, so the first item is "item 0" not "item 1". You can use bracket notation to select items by their index:

```
package rental

genres := ["crime", "drama"]

# "crime"
first  := genres[0]

# "drama"
second := genres[1]
```

In Alex's input, `input.member.rentals.items` follows this pattern: an array of objects, each with information about the film and when it was due to be returned. *Speed* and *The Lion King* are elements `0` and `1`. Arrays are useful when the order is important, or when duplicates are meaningful.

Objects

The member record is a bundle of named fields. Rego's *objects* are unordered key-value maps. This is the same idea as a JSON object or a dictionary in other languages. String keys are the common case:

```
package rental

film := {
  "title":    "Pulp Fiction",
  "director": "Quentin Tarantino",
  "year":     1994,
  "rating":   "18",
}
```

You can look up values with dot notation when the key is a simple identifier. In other cases, you can use bracket notation to handle situations where the key would cause a parser error:

```
package rental

film  := {"title": "Pulp Fiction", "rating": "18"}

# dot notation
title  := film.title

# bracket notation
rating := film["rating"]
```

Bracket notation is required when the key contains characters that are not valid in an identifier. If the key contains hyphens, spaces, or dots, or starts with a number, you'll need to use bracket notation. You can also use it when a variable supplies the key, for example: `stores[store_id]`. There's no limit to the depth of nesting, though very deeply nested objects can be harder to work with. In the example, Alex's membership tier is three levels deep at `input.member.membership.tier`.

```
package rental

# "classic"
member_tier := input.member.membership.tier
```

Use objects when you look things up by name or when you need a structured record with several fields that belong together.

Sets

Some store rules are really membership tests: "Is this rating on the restricted list?" A *set* is an unordered collection of unique values. When you use a set, duplicates disappear:

```
package rental

restricted_ratings := {"18", "R", "NC-17"}
adult_genres       := {"adult", "horror"}
```

When working with a set, you usually use the `in` keyword to check whether a value belongs to the set. The `in` keyword is easy to read and efficient to evaluate.

```
package rental

default age_restricted := false

restricted_ratings := {"18", "15"}

age_restricted if input.film.rating in restricted_ratings
```

The rental request has a rating of `"18"`, so `age_restricted` will be true. You can change the rating in the input panel to `"12"` and evaluate the policy to see `age_restricted` return false because `"12"` isn't in the set.

Here's a quick guide to choosing between composite types:

- Use a **Set** when you need unique values, and the question is whether something is in the set.
- Use an **Array** when you need to keep duplicates or the order matters.
- Use an **Object** when you need named fields and nested records.

Property	set	array	object
Syntax	`{"18", "R"}`	`["a", "b"]`	`{"k": "v"}`
Order	unordered	**ordered**	unordered
Duplicates	no (unique values)	**yes**	keys unique
Access	`val in s`	`s[i]`	`s.key` `s["key"]`
Best for	membership tests e.g. rating in restricted	ordered sequences e.g. rentals.items[0]	named records e.g. film.title

When the question is "is X a member?", reach for a set. When order matters, use an array. For named fields, use an object.

The empty set gotcha

Because objects and sets both use curly braces, you might wonder what empty curly braces mean. In Rego, `{}` is always an *empty object*, never an empty set. This is consistent with JSON, where `{}` is an object.

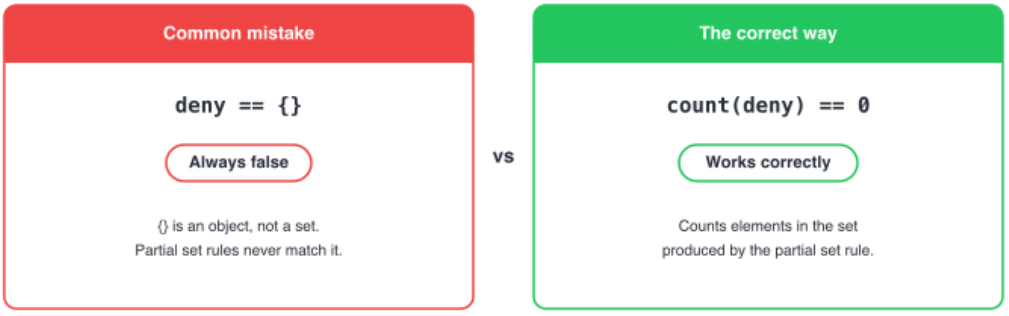
If you need an empty set, use the `set()` constructor:

```
package rental

# this is an object
empty_object := {}

# this is a set
empty_set    := set()
```

If you compare an empty object and an empty set, they are different values:

```
package rental

# False because {} is an object
are_equal := {} == set()
```

You rarely need to create an empty set as they are usually created from rules and comprehensions, which you'll learn more about in Chapter 7. When you do need one, write `set()`, not `{}`.



In Rego, {} is always an empty object (JSON-compatible). Use set() or count() when working with sets.

Composite values as constants

When you first write rules, you'll find yourself hard-coding values, like `"15"` and `"18"`. This can get very noisy and is hard to update when something changes. A better pattern is to name the collection once and refer to it from the rules:

```
package rental

# Ratings that require an age check
age_restricted_ratings := {"15", "18"}

# Tiers allowed to rent more than the default amount
extended_rental_tiers := {"plus", "premier"}

# Genres that trigger a warning
flagged_genres := {"horror", "adult"}

allow if {
  input.film.rating in age_restricted_ratings
  input.member.age >= to_number(input.film.rating)
}

extended_rental_allowed if {
  input.member.membership.tier in extended_rental_tiers
}
```

Anyone reading the top of the file sees the store's vocabulary before they decode the rule bodies. You can change a tier name or add a rating in one place, and every rule that references the set picks it up.

What comes next

You now know the full Rego vocabulary of four scalars and three composites. You also know to avoid the confusion between empty objects and sets. In the next chapter, you'll put this to work by navigating Alex's nested input with references, and reusing intermediate results so the counter rules stay short and readable.

Open the pre-populated playground with Alex's rental request (**oc.to/rego-playground**), and try out your new skills. When you are comfortable with the shapes and have topped up your coffee, turn the page.

Key concepts

- There are four scalar types (string, number, boolean, and null) and three composite types (object, set, array).
- You can use raw strings for regular expressions and other backslash-heavy content.
- String interpolation lets you embed expressions into templates.
- You can name collections as constants at the top of a policy to make your rules more readable.

Part II

Variables, rules, comprehensions, and equality

CHAPTER 4

Variables and references

Now you're armed with Rego's vocabulary for types, the next step is to look at how policy reaches those values. That means thinking about references and variables. You get references using paths through input and data. These are called dot-paths as they navigate the hierarchy with dots, like `input.member.age`. Variables are implicitly created on their first use. When you create a variable in the head of a rule, it acts as both an input and an output. The variables you add to the rule's body are scoped to that rule.

In this chapter, you'll get comfortable with references and variables, which you'll use later to make your policies short and comprehensible.

Variables in Rego

In most languages, a variable is either an argument (input) or something you assign (output). In Rego, the same name can play *both roles in the same rule*, depending on what you ask OPA at query time. That sounds abstract until you see it in an example.

Input and output in one rule

Suppose you want every late-fee amount above £15 from a short list—standing in for a handful of fee lines the till might total up. You are not assigning `x` up front; you are describing which values of `x` satisfy a condition and letting OPA find them.

```
package rental

# X is an output: OPA finds all values of x that make this true
result contains x if {
  some x in [10, 20, 30]
  x > 15
}
```

Evaluate `result` in the playground at **play.openpolicyagent.org**. OPA searches `[10, 20, 30]` for every `x` with `x > 15`, binds those that work, and adds them to the set. You should see:

```
{
  "result": [20, 30]
}
```

Here `x` behaved as *output*: OPA discovered the values. Now query `result[20]` instead of `result`. You are telling OPA that `x` is `20` and asking whether that value belongs in the set. The same rule runs, but `x` is *input*. OPA checks whether `20` is in the list and whether `20 > 15`. The query succeeds. It's the same variable and the same rule; the role flips with how you call it. OPA handles that without extra syntax on your side.

Local variables

When the employee looks up a price, they might use a scratch figure on the notepad that does not change the posted shelf price. In Rego, `:=` inside a rule body creates a *local* binding—visible only in that rule and able to *shadow* a package-level name with the same identifier:

```
package rental

# Package-level rule named 'price'
price := 3.50

p if {
  # local variable, shadows the package-level 'price'
  price := 1.00

  # true — refers to the local price, which is 1.00
  price != 3.50
}
```

Evaluate `p` with this policy in the playground. The rule succeeds: inside the body, `price` means `1.00`, not the `3.50` defined at package scope. Nothing outside the rule changes. That scoping matters as soon as you have reusable names like `price` or `tier` at the top of a file and need a temporary value inside one check.

Variables must be safe

Before Alex's late fees can feed a comparison, the policy must *bind* the variable—give it a definite value in a non-negated expression. Rego's *safety* rule rejects bodies where a variable is used before it could be known:

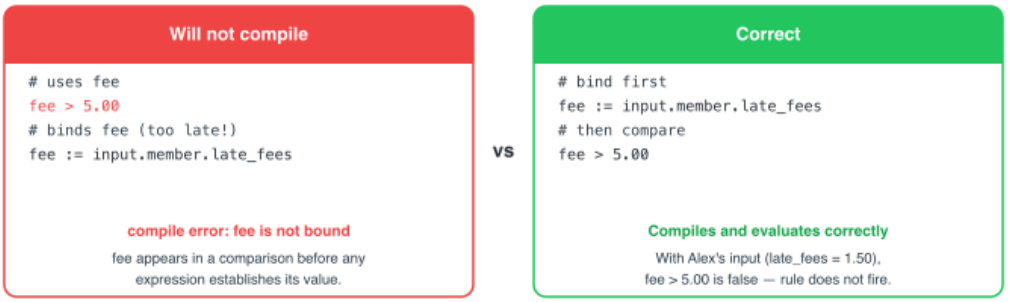
```
package rental

# Error: fee is not bound before use
p if {
  fee > 5.00
  fee := input.member.late_fees
}
```

This module will not compile: `fee` appears in `fee > 5.00` before any expression establishes what `fee` is. OPA can reorder some expressions inside a rule body, but not when the order would leave a variable ambiguous—especially under negation.

The fix for Alex's account is straightforward: bind first, then compare (or rely on reordering only when the binding is clearly in the same body and not negated):

```
package rental

p if {
  fee := input.member.late_fees
  fee > 5.00
}
```

With Alex's fixture input (`late_fees` is `1.50`), `p` does not hold. Set `late_fees` to `6.00` in the input document and evaluate again—the rule succeeds. In practice, writing `:=` assignments above the expressions that use them is the clearest style and avoids surprises.



Assign a variable before you use it. OPA rejects bodies where a variable is compared before it is bound.

References

A *reference* is a path into a document. Every time you wrote `input.member.age` or `input.film.rating` in earlier chapters, you were already referencing. The employee's checklist is the same idea: start at Alex's member record, then membership, then tier.

Dot notation and bracket notation

Dot notation is the readable default when keys look like ordinary identifiers:

```
input.member.age
input.member.membership.tier
input.film.rental_price
```

Bracket notation uses string keys in square brackets and is equivalent when the key is a valid identifier:

```
input["member"]["age"]
input["member"]["membership"]["tier"]
input["film"]["rental_price"]
```

Both forms yield the same values on Alex's input. Prefer dots where you can; use brackets when you must:

1. The key contains characters that are not valid in an identifier (hyphens, spaces, dots): `input.request["store-id"]`
2. The key is a number: `input.member.rentals.items[0]`
3. The key is a variable (covered below)
4. The key is a composite value (covered below)

Real policies often mix both in one path:

```
input.member.rentals.items[0]["film_id"]
```

Alex's request uses store `"emptybox-central"`. A hyphen is not allowed in dot notation, so you write `input.request["store"]`—not `input.request.emptybox-central`, which would be a syntax error.

Paste Alex's input from Chapter 1 (or `samples/fixtures/input.json`) into the playground and evaluate `input.member.membership.tier`. You should see `"classic"`.

Navigating arrays

Alex has two tapes out. The first slot in `input.member.rentals.items` is *Speed*; the second is the overdue *Lion King*. Numeric indices pick elements from an array:

```
package rental

# "Speed"
first_rental_title  := input.member.rentals.items[0].title

# "The Lion King"
second_rental_title := input.member.rentals.items[1].title
```

Evaluate `first_rental_title` and `second_rental_title` with the shared rental input. If you reference an index that does not exist—`items[9]`, say—the reference is *undefined*. OPA does not throw; the rule or query simply has no value for that path.

Navigating objects

Object fields use the same dot or bracket forms. Keys that are not valid identifiers must use brackets:

```
package rental

# "classic"
tier             := input.member.membership.tier

# false
fees_outstanding := input.member.membership["fees_outstanding"]
```

A missing key is like a missing index: undefined, not an error. Chaining steps walks from the root of the document to the field you need—Alex's tier is three steps in from `input`:

```
package rental

# "classic"
member_tier := input.member.membership.tier
```

Read the path left to right: `input`, then `member`, then `membership`, then `tier`.

Variable keys: iterating collections

The employee does not ask "is rental slot zero overdue?" and stop; they scan the whole list. When a *variable* appears where a reference expects a key or index, OPA iterates: it tries each position (or each object key), binds the variable, and keeps the combinations that satisfy the rest of the rule.

Overdue titles from Alex's rentals

Collect every overdue title from Alex's current rentals:

```
package rental

# Collect the titles of all overdue rentals
overdue_titles contains title if {
  some i
  item := input.member.rentals.items[i]
  item.overdue
  title := item.title
}
```

Use Alex's input in the playground and evaluate `overdue_titles`. Index `i` is a variable in the path `items[i]`. OPA tries `0` and `1`, binds `item` each time, and keeps rows where `item.overdue` is true. Only *The Lion King* qualifies:

```
{
  "overdue_titles": ["The Lion King"]
}
```

Declare iterator variables with `some` when they appear as keys in references. Without `some`, a name like `i` might be resolved as an existing rule in the package instead of a fresh binding—we go deeper on that in Chapter 9.

Iterating object keys

Store policy caps concurrent rentals by membership tier. The limits live in `input.membership_limits` as an object; a variable key walks every tier:

```
package rental

# Find all tiers where the rental limit exceeds 4
generous_tiers contains tier if {
 some tier
 input.membership_limits[tier] > 4
}
```

On Alex's input, `plus` (5) and `premier` (10) exceed 4; `classic` (3) does not:

```
{
  "generous_tiers": ["plus", "premier"]
}
```

The underscore: a throwaway iterator

Sometimes you only need the value at each step, not the index. The underscore `_` means "some index or key I do not care to name":

```
package rental

# Collect all film ids from the member's current rentals
all_rented_ids contains film_id if {
 film_id := input.member.rentals.items[_].film_id
}
```

With Alex's input, evaluate `all_rented_ids`. OPA walks every rental slot and collects both IDs:

```
{
  "all_rented_ids": ["VHS-0871", "VHS-1204"]
}
```

Each `_` is a separate, anonymous iterator. You cannot refer to "the value of that underscore" elsewhere in the rule.

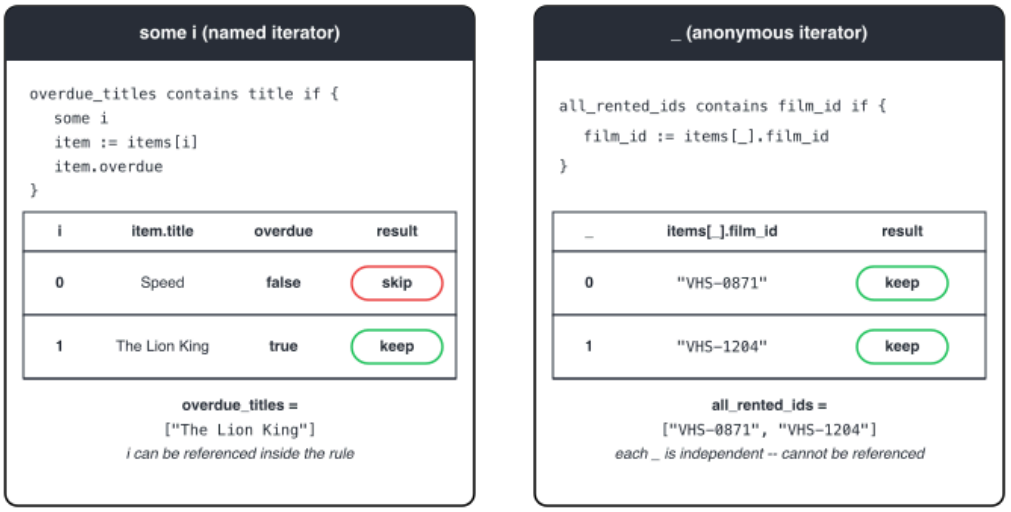
Nested collections use multiple underscores. For a catalogue-shaped document in **Data**, add:

```
{
  "catalogue":[{
    "genres":[
      "crime",
      "drama"
    ]
  },{
    "genres":[
      "action"
    ]
  }]
}
```

Then evaluate:

```
package rental

all_genres contains genre if {
  genre := data.catalogue[_].genres[_]
}
```

The first `_` steps through catalogue entries; the second steps through genres inside each entry. OPA collects every `genre` that appears in any valid pairing—effectively a nested walk without naming every index. Under the hood, each `_` becomes a unique variable name so nothing collides. Remember: **each `_` is independent**; you cannot tie two underscores together across expressions.



some i names the iterator for reuse; _ is a throwaway that iterates without creating a bindable variable.

Multiple references: implicit joins

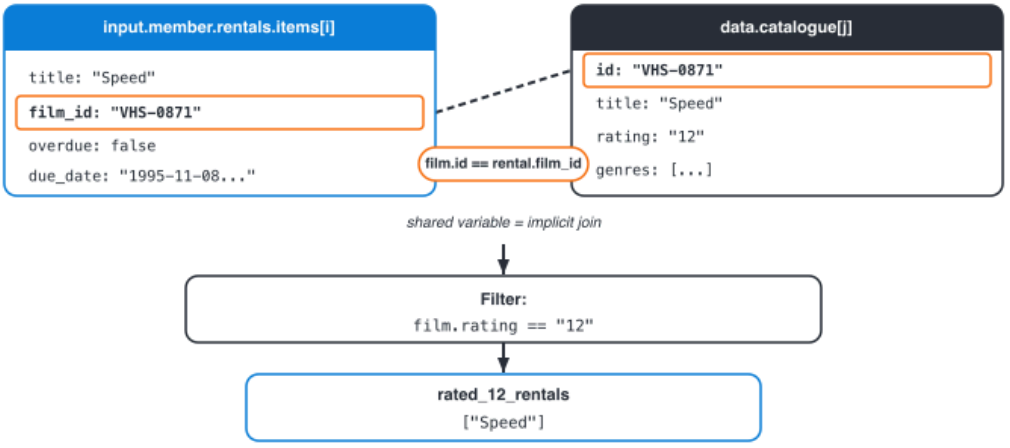
Alex's rental items carry a `film_id`. The store catalogue lists films by `id`, rating, and title. To ask "which of Alex's current tapes are rated 12 in the catalogue?", you need rows where the rental's `film_id` matches a catalogue entry's `id` *and* that entry's rating is `"12"`. Two references that share a variable express that join—no explicit `JOIN` keyword:

```
package rental

# Find the titles of all 12-rated films Alex currently has out
rated_12_rentals contains title if {
  some i
  rental := input.member.rentals.items[i]
  some j
  film   := data.catalogue[j]

  # the join condition
  film.id == rental.film_id

  film.rating == "12"
  title := rental.title
}
```

Put Alex's input under **Input** and a catalogue under **Data** where at least one of his rented films has `"rating": "12"` (for example, add `"rating": "12"` to the *Speed* entry matching `VHS-0871`). OPA only emits a title when both sides agree on the shared `film.id` / `rental.film_id` and the rating matches. You might see:

```
{
  "rated_12_rentals": [
    "Speed"
  ]
}
```

This is one of Rego's strengths for policy: relationships that would take loops and index bookkeeping in imperative code fall out of shared variables and paths.



A shared variable (film.id == rental.film_id) links two collections without an explicit JOIN keyword.

Composite keys

Most of Alex's data uses strings and numbers as keys. Rego also allows *composite* keys—arrays or objects used as lookup keys—but only when the collection is a *set*, not an array or object document. That shows up when you store structured pairs in a set and want to test or filter by part of the structure:

```
package rental

# A set of known member-film pairs that have been flagged
flagged_pairs := {["MBR-0042", "VHS-2209"], ["MBR-0099", "VHS-0012"]}

# Check whether a specific pair is flagged,
# will be the pair if it exists, undefined otherwise
is_flagged := flagged_pairs[["MBR-0042", "VHS-2209"]]

# Find all pairs flagged for member MBR-0042
member_flags contains pair if {
  pair := flagged_pairs[["MBR-0042", _]]
}
```

Evaluate `is_flagged` and `member_flags` in the playground. The first query confirms Alex's member id with *Pulp Fiction*'s tape id is in the flagged set; the second collects every flagged pair whose first element is `MBR-0042`. You will not need composite keys often, but they are the right tool when a set's elements are themselves small structures and you want partial matching.

Putting it together

Back at the counter, the employee cares about *which* overdue titles block a new rental and *how long* each has been late. The following rules combine iteration, references into Alex's rentals, and a time calculation against `input.request.timestamp`:

```
package rental

# Collect the due dates of all overdue items
overdue_due_dates contains due_date if {
  some item in input.member.rentals.items
  item.overdue
  due_date := item.due_date
}

# Calculate the total number of overdue items
overdue_count := count(overdue_titles)

overdue_titles contains title if {
  some item in input.member.rentals.items
  item.overdue

  title := item.title
}

# The member has a block-worthy overdue situation if any item
# Has been overdue for more than 7 days since the request timestamp
seriously_overdue contains title if {
  some item in input.member.rentals.items
  item.overdue

  request_ns  := time.parse_rfc3339_ns(input.request.timestamp)
  due_ns      := time.parse_rfc3339_ns(item.due_date)
  days_late   := (request_ns - due_ns) / 1000000000 / 86400
  days_late > 7
  title := item.title
}
```

Load Alex's input and evaluate the package (or query each rule). `overdue_titles` lists every tape still flagged overdue; `overdue_due_dates` collects their due dates; `overdue_count` counts titles via `count(overdue_titles)`; `seriously_overdue` applies the seven-day threshold using the request time on the counter clock. With the fixture timestamps, *The Lion King* is overdue but not yet "seriously" late by that rule:

```
{
  "overdue_count": 1,
  "overdue_due_dates": [
    "1995-11-06T00:00:00Z"
  ],
  "overdue_titles": [
    "The Lion King"
  ],
  "seriously_overdue": []
}
```

Two rules, one input document, separate concerns—exactly how you will structure larger rental policy in the chapters ahead.

What comes next

You can now navigate Alex's input, iterate his rentals and the store's limits, and join rental rows to catalogue rows with shared variables. Chapter 5 looks at the different *shapes* rules can take—complete definitions, partial sets, partial objects, and incremental rules—and how to choose among them when you model allow, deny, and detail at the counter.

Key concepts

- Variables in Rego are simultaneously input and output depending on what is known at query time.
- Local variables are declared with `:=` and are scoped to their rule. They shadow package-level names.
- References navigate documents using dot notation or bracket notation. Bracket notation is required for non-identifier keys, numbers, and variables.
- Using a variable as a key in a reference causes OPA to iterate over that collection.
- The underscore `_` is a throwaway iterator. Each `_` is independent and cannot be referenced by name.
- Shared variables across multiple references create implicit joins — OPA only produces results where both references agree.

CHAPTER 5

Rules

You can now navigate nested `input` elements, iterate over Alex's rentals, and join rows to the catalog using variables and references. All these techniques live inside *rules*, but Rego rules are not one-size-fits-all. Some decisions are based on a single value, while others depend on multiple inputs.

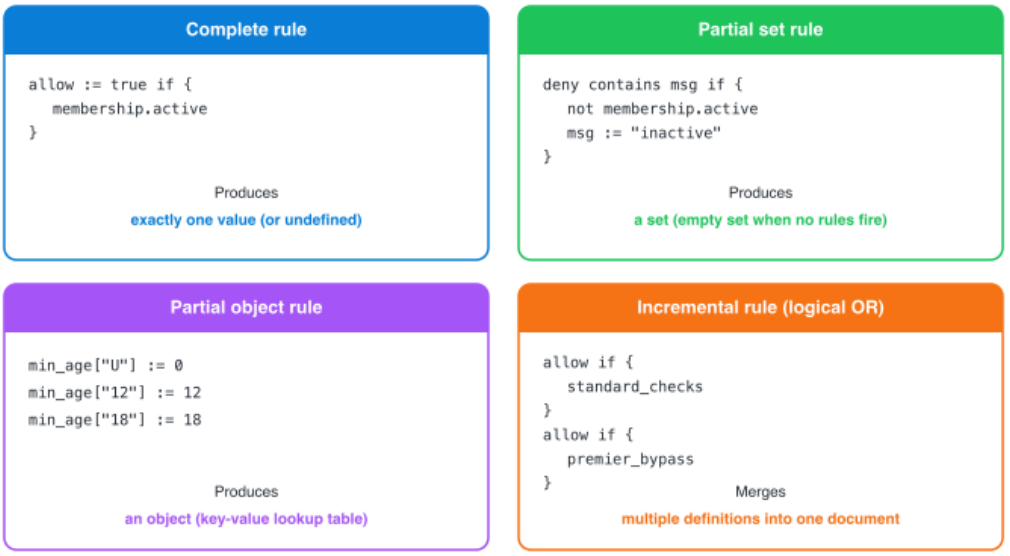
This chapter maps the rule forms OPA supports, like complete definitions, partial sets and objects, incremental contributions, and reference heads. You can use these to match the shape of the rule to the shape of the answer needed at the checkout desk. At the highest level, every Rego rule shares the same skeleton. It begins with a *head*, and optionally has a *body*.

Form	Head	Body
`user := "Name"`	`user := "Name"`	implicit `true`
`user := "Name" { true }`	`user := "Name"`	`true`
`allow { input.x == 1 }`	`allow`	`input.x == 1`
`result := x { x := input.n }`	`result := x`	`x := input.n`

The presence or absence of a key and a value in the head determines what kind of document the rule produces:

```
<name> <key>? <value>? <body>?
```

You have already written several of these forms without naming them. The sections below start from Alex's situation at the checkout desk, show the Rego that expresses it, and highlight what you should see when OPA evaluates it.



Choose the form that matches the shape of the answer: single value, collection, lookup, or merged contributions.

Complete definitions

Some questions demand a single, unambiguous answer, like whether the membership is valid. This is a *complete* rule, because at any point in time it has one value, or is undefined. It doesn't hold two different values at once.

```
package rental

# No body, always defined, always 3
default_rental_days := 3

# Has a body and only defined if the condition holds
membership_valid := true if {
  input.member.membership.active
  not input.member.membership.fees_outstanding
}
```

It doesn't matter what the type of the value is. A rule with no body is always defined, while one with a body is defined only when every expression in the body holds for the given input. Complete rules are for decisions that must not waffle. Things like the rental price, the maximum allowed rentals, or a simple yes-or-no. If you have multiple definitions that would produce different values for the same input, the policy engine raises a conflict error. Preventing these conflicts removes ambiguity in your policies.

If the manager at EmptyBox allowed longer rental periods for 18-rated films, and also let premier members keep films longer, you could end up with two complete rules that both fire. Renting *Pulp Fiction* with a "premier" membership means the rating condition and the membership tier condition both apply, but they might not provide the same answer.

```
package rental

# Both conditions apply
max_rental_days := 7 if input.film.rating == "18"
max_rental_days := 14 if input.member.membership.tier == "premier"
```

If you run this policy in the pre-populated playground (**oc.to/rego-playground**), you'll see the maximum rental period is 7 days. Now update Alex's membership in the input panel to upgrade him from the "classic" to the "premier" tier. When you evaluate the policy, it creates a conflict, and you'll receive the error:

> Complete rules must not produce multiple outputs

The fix is to make conditions mutually exclusive, or to use `else` when one case should win over another, which you'll learn in more detail in Chapter 11. Don't forget to downgrade Alex to the "classic" tier when you finish trying out this example.

Partial definitions: generating sets

While complete rules have a single answer, you sometimes need a way to collect many values, like all the reasons a policy failed. In the input panel in the playground, deactivate Alex's membership by setting the `active` flag to `false`, and set `fees_outstanding` to `true`. This is what the membership would look like if Alex hadn't paid the monthly fee.

You've likely been in a situation where a company rejected your request without being able to adequately explain why. That's what you should avoid when creating policies that may fail for multiple reasons at once. Your policy should be able to supply all the reasons for a failure. A *partial* set rule builds that list incrementally. The head has a key but no value, and the `contains` keyword marks it as a rule that produces a set:

```
package rental

# Produces a set of all denial reasons
deny contains msg if {
  not input.member.membership.active
  msg := "membership is not active"
}

deny contains msg if {
  input.member.membership.fees_outstanding
  msg := "member has outstanding fees"
}

deny contains msg if {
  input.film.available_copies < 1
  msg := "no copies available"
}
```

You can read the rule as "the deny set contains this message if..." For each body that evaluates to true, the policy engine adds the value of `msg` (defined in the body) to the set named `deny`. Nothing is lost, because all values join the set.

When you evaluate this rule against Alex with his unpaid membership and inactive account, you'll get the following result:

```
{
  "deny": [
    "member has outstanding fees",
    "membership is not active"
  ]
}
```

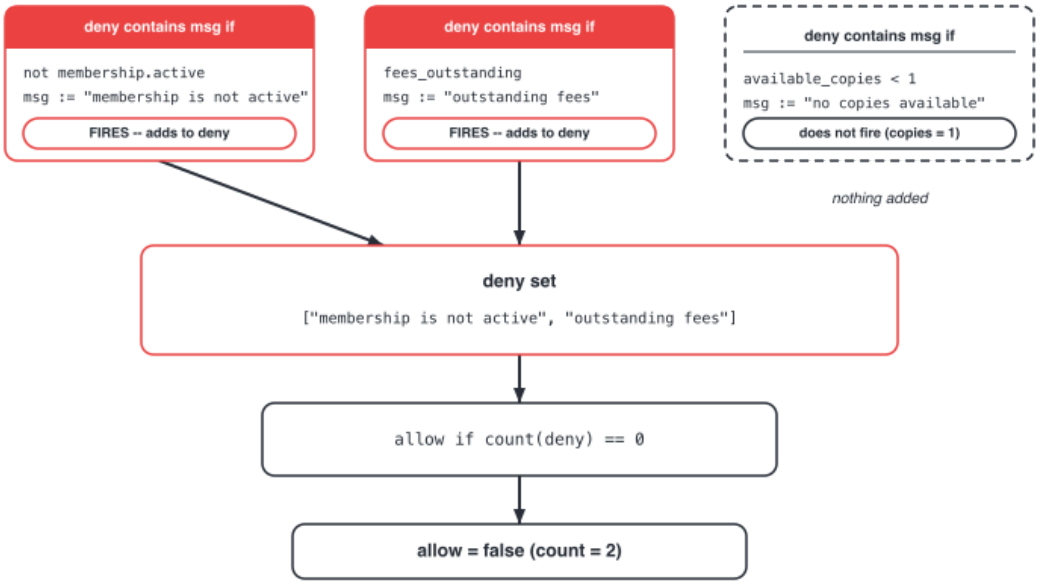
When you use partial sets, you'll always get a value in the result, never *undefined*. If you return Alex to an active membership with no outstanding membership fees, you'll get an empty set in the result:

```
{
  "deny": []
}
```

None of the rule conditions fired, so there are no deny reasons. Downstream rules can safely count the deny reasons, as they will always be present. For example, when every check passes, the allow rule below will be true.

```
allow if count(deny) == 0
```

This technique lets you add new rules over time without having to redefine the success condition. You can manage each condition in isolation, which helps you scale your policies without getting tangled.



Each deny rule contributes independently. An empty deny set (all checks pass) means count(deny) == 0 and allow is true.

Partial definitions: generating objects

Sometimes the key, rather than the value, is the point of interest. EmptyBox employees don't memorize the rental cap for every membership tier; they look it up. A *partial object* rule has both a key and a value in the head and builds a mapping entry by entry:

```
package rental

# Produces an object mapping tier name to maximum rental count
tier_limits[tier] := limit if {
 some tier, limit in data.membership_limits
}
```

Run this policy in the pre-populated playground (**oc.to/rego-playground**), and you'll get a list of limits listed by tier:

```
{
  "tier_limits": {
    "classic": 3,
    "plus": 5,
    "premier": 10
  }
}
```

Partial objects follow the same conflict rule as complete rules. If two definitions assign different values to the same key, you'll get a conflict error. You can assign the same value to a key from multiple definitions without an error. You can also create partial objects using static rules; for example, EmptyBox could use an age policy table to set the allowable age for different rental ratings, which addresses the challenge of some ratings being strings while others are numbers (but stored as strings).

```
package rental

# Minimum age required for each rating
min_age_for_rating["U"]   := 0
min_age_for_rating["PG"]  := 0
min_age_for_rating["12"]  := 12
min_age_for_rating["15"]  := 15
min_age_for_rating["18"]  := 18

age_required := min_age_for_rating[input.film.rating]
```

Running this policy for Alex produces an `age_required` of `18`, which you can use to check whether the rental is allowed without parsing strings to numbers.

Incremental rules and logical OR

Rego lets you define the same rule name more than once. Those definitions are *incremental*: each one contributes to the same document, and the result is the union of what each definition produces. That is how Rego expresses logical OR. For sets, incremental definitions are natural. Each `deny contains msg if ...` block is already an independent contributor to the same set.

When you create complete rules with multiple bodies, you are telling the policy engine that any of those paths is enough. EmptyBox might allow a rental when Alex meets the usual checks *or* when he is a premier member who bypasses the age gate:

```
package rental

default allow := false

# Standard requirements
allow if {
  input.member.membership.active
  not input.member.membership.fees_outstanding
  input.film.available_copies > 0
  input.member.age >= to_number(input.film.rating)
}

# Premier bypass
allow if {
  input.member.membership.active
  not input.member.membership.fees_outstanding
  input.film.available_copies > 0
  input.member.membership.tier == "premier"
}
```

Read together, those blocks mean:

```
allow IS true IF (standard requirements) OR (premier override)
```

You can try this policy in the playground and adjust either Alex's age or membership tier to make the policy pass. Any one satisfied body is enough to allow the rental. If you add a new rule that allows staff to rent any movie, you can add another body instead of editing existing ones. That structure is easier to read and test than one rule that attempts to cover all scenarios. This is one of the most common patterns in production policies for a reason: each path is a separate story that describes a scenario in which a rental is allowed.

Rule heads with references

The rule names you've used so far have been flat. Names like `deny`, `allow`, or `tier_limits`. When you write real policies, you may need to create nested documents, like grouping rentals by status or attaching metadata under a film path. Rego lets you put a reference in the rule head that names the path you want to define:

```
package rental

# Two complete rules at nested paths
film.title    := "Pulp Fiction"
film.director := "Quentin Tarantino"
```

These define `film.title` and `film.director` under the package. OPA constructs the containing `film` object implicitly based on your paths.

References become more powerful when variables define parts of the path. Any segment after the first can be dynamic, which is how you build nested objects from a flat list:

```
package rental

rental_status(item) := "overdue" if item.overdue
rental_status(item) := "current"  if not item.overdue

# Build a nested summary: rentals grouped by overdue status, then by film_id
rentals_by_status[status][film_id] := item if {
  some item in input.member.rentals.items
  film_id := item.film_id
  status  := rental_status(item)
}
```

When you evaluate this policy, you'll get a list of Alex's rentals by their status: *Speed* (current) and *The Lion King* (overdue):


```
{
  "rentals_by_status": {
    "current": {
      "VHS-0871": {
        "due_date": "1995-11-08",
        "film_id": "VHS-0871",
        "overdue": false,
        "title": "Speed"
      }
    },
    "overdue": {
      "VHS-1204": {
        "due_date": "1995-11-06",
        "film_id": "VHS-1204",
        "overdue": true,
        "title": "The Lion King"
      }
    }
  }
}
```


You created two levels of nesting from a flat list in just a few lines. This is useful for dashboards, printed receipts, or tests that verify the correct tapes landed in the correct bucket.

You can mix constants and variables in the same structure. Suppose you also want the film Alex is *requesting* to appear under a fixed `pending` branch while the rest of the grouping stays dynamic:

```
package rental

rental_status(item) := "overdue" if item.overdue
rental_status(item) := "current"  if not item.overdue

# General rule: group rentals by status
rentals_by_status[status][film_id] := item if {
  some item in input.member.rentals.items
  film_id := item.film_id
  status  := rental_status(item)
}

# Also add the requested film under "pending"
rentals_by_status["pending"][film_id] := {
    "film_id": film_id,
    "title": input.film.title,
} if {
    film_id := input.film.id
}
```

The second definition adds an entry under `pending` regardless of how the first rule classified existing rentals.

Conflicts in reference heads

Variable paths are still subject to the uniqueness rules you saw for complete and partial object rules. Two definitions must not write different values to the same path.

```
package rental

# R1: dynamic path
film_info[field].value := "Pulp Fiction" if { field := "title" }

# R2: constant path — overlaps with R1's dynamic extent
film_info.title.value := "Pulp Fiction 2"
```

Evaluating this policy fails with:

> Object keys must be unique

When OPA can see at compile time that two rules both target `film_info.title.value` with different values, it reports the conflict before evaluation. When overlap depends on runtime bindings, the error occurs during evaluation instead. The practical habit: if multiple definitions might touch the same path, either they must always agree on the value, or only one of them can fire for any given input.

Choosing the right rule form

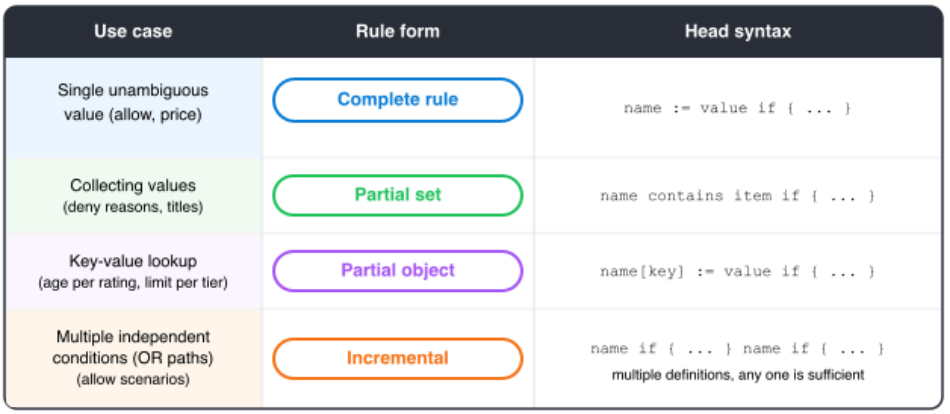
Four shapes cover most of what you will write at EmptyBox Rentals:

Use a complete rule when a decision must have exactly one value, like the `allow` verdict, a computed price, or a resolved tier. Pair it with `default` when undefined is not an acceptable outcome.

Use a partial set rule when you are collecting a group of values, like all denial reasons, all overdue titles, all ratings that need a manager sign-off. Multiple definitions add members; an empty set is still defined.

Use a partial object rule when lookup by key is the point, as it is for minimum age per rating, rental limit per tier, and price per membership level. The key is meaningful at evaluation time.

Use incremental definitions when independent conditions or data sources should contribute to the same document. This is often several `allow if` paths, several `deny contains` lines, or a table built from more than one rule. Prefer several small bodies over one large boolean knot.



Match the rule form to the shape of the answer: one value, a collection, a lookup, or a union of scenarios.

What comes next

You now have a map of Rego's rule forms and when to use each one. In the next chapter, you'll tackle something that trips up almost everyone new to Rego: the three equality operators. Understanding the difference between assignment, comparison, and unification is essential to reading and writing rule bodies correctly.

Key concepts

- Complete rules produce exactly one value or are undefined.
- Multiple definitions with different values for the same input cause a conflict error.
- Partial set rules (`contains`) collect values into a set. They are safe to leave empty, as an empty set is not undefined.
- Partial object rules (`[key] := value`) build key-value mappings. Different definitions may not assign different values to the same key.
- Incremental definitions of the same rule name express logical OR. The policy engine merges all contributions into the final document.
- Reference heads (`film.title := "..."`) let you define nested structures concisely. Variables in the head create dynamic nested objects.
- Choose the rule form that matches the shape of the decision: single value → complete, collection → partial set, lookup table → partial object, multiple sources → incremental.

CHAPTER 6

Equality, assignment, and comparison

You've learned how to pick from Rego's rule shapes, and you know when to emit one value, a set of reasons, or an object keyed by rating. Most of the expressions inside the rule bodies you've seen so far compare or bind values, like membership active, age against rating, or fee thresholds.

Rego offers three operators where other languages offer one or two: `:=` for assignment, `==` for comparison, and `=` for unification. They are not interchangeable. This chapter introduces each one, shows the rental-counter cases that call for it, and walks through what the policy engine does when you evaluate the examples.

The three operators are:

Operator	Name	Purpose
`:=`	Assignment	Declare and bind a local variable
`==`	Comparison	Test whether two already-known values are equal
`=`	Unification	Assign and compare simultaneously

We'll look at each of these in the order you will use most often: to name a value, test a value, then unify when you need pattern matching or bidirectional solving.

Default to := for naming and == for testing. Reach for = only when you need bidirectional solving or destructuring.

Assignment (:=)

Before the employee compares Alex's age to the film rating, they mentally label two things: *this member* and *this tape*. The `:=` operator does the same in a rule; it declares a local name and binds it to a value in one step. You have already used it in earlier chapters:

```
package rental

age_check if {
  member := input.member
  film   := input.film
  member.age < to_number(film.rating)
}
```

In this example, `member` and `film` are declared and bound by `:=`. They are local to this rule. Once bound, you can't rebind them. This is a one-time binding, not a mutation. The following does not compile:

```
package rental

p if {
  tier := "classic"

  # error: tier is already assigned
  tier := "premier"
}
```

> var tier assigned above

The compiler also catches use that occurs before the assignment. A variable must be bound before it appears in any expression that depends on its value:

```
package rental

p if {
  # error: age is not yet bound
  age != 16

  # it gets bound here
  age := input.member.age
}
```

> var age referenced above

These checks rule out a class of bugs common in dynamic languages, using names before they have a value or overwriting a binding halfway through a rule. The `:=` operator is strict so that your rules stay predictable.

When to use ":="

Use `:=` whenever you want to give a name to a value for use within a rule body. This is the form of "equality" you will write most often in Rego:

```
package rental

default allow := false

allow if {
  member := input.member
  membership := input.member.membership
  film   := input.film

membership.active
  not membership.fees_outstanding
  film.available_copies > 0
  member.age >= to_number(film.rating)
}
```

Creating intermediate values with `:=` makes rules easier to read. You can avoid repeating long paths, like `input.member.membership.active` and instead reference a short, meaningful name.

Comparison: (==)

"Is Alex exactly sixteen?" is a yes-or-no question about values you already know. The `==` operator tests whether two values are equal. Both sides must already be bound, meaning they must refer to values known when the policy engine evaluates the expression. It can be bound in the body or the package.

```
package rental

p if {
  age := 16
  age == 16    # true
}

max_rentals := 3

q if {
  max_rentals == 3   # true
}
```

When you perform a comparison using `==`, nothing gets bound. The operation only tests the two values and returns true if they are equal. If they aren't the same, the condition doesn't hold, and the rule doesn't fire.

Attempting to use `==` with an unbound variable is a compile-time error:

```
package rental

p if {
  # error: tier is unsafe (not bound)
  tier == "premier"
}
```

> var tier is unsafe

When to use "=="

Use `==` to test the value of something that is already known. After binding a variable with `:=`, use `==` to check conditions on it:

```
package rental

premier_member if {
  tier := input.member.membership.tier
  tier == "premier"
}
```

Evaluate this policy in the pre-populated playground (**oc.to/rego-playground**). Alex has a classic membership, so the rule doesn't fire. You can change the membership tier in the input panel to "premier" and evaluate the policy again to see a match. Naming with `:=`, then testing with `==`, mirrors how the employee first looks up the tier and then decides whether it's premier.

You can also use `==` when both sides are literals or references to existing rules:

```
package rental

default action_type := "unknown"
action_type := "rental"  if input.request.action == "rent"
action_type := "return"  if input.request.action == "return"
```

If you're *checking* a value, use `==`. If you are *naming* a value, use `:=`.

Unification: (=)

Sometimes you do not know which side of an expression should supply the answer. The catalog might list a film as `["VHS-2209", "Pulp Fiction"]` while a rental line only has an ID and an empty title slot. The `=` operator combines assignment and comparison. It asks the policy engine to find values for any unbound variables on either side that will make both sides equal.

If both sides are bound, `=` behaves like `==` and performs the comparison.

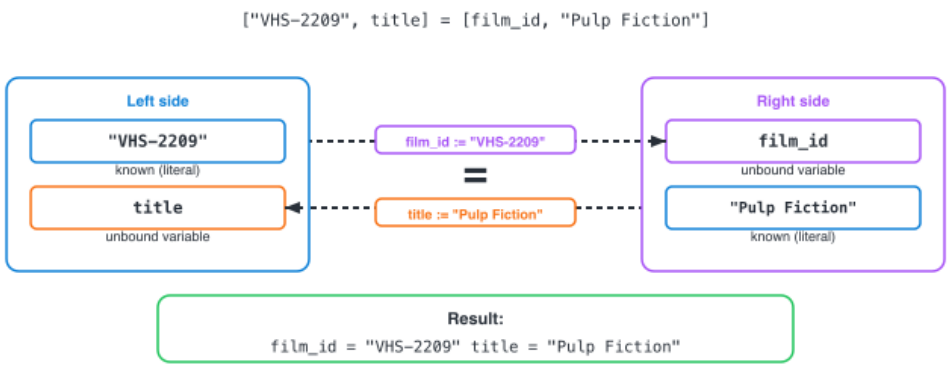
```
package rental

p if {
  age := 16
  age = 16    # same as using ==
}
```

When one side has an unbound variable, `=` binds it:

```
package rental

result := [film_id, title] if {
  ["VHS-2209", title] = [film_id, "Pulp Fiction"]
}
```

In this example, the title is missing on the left of the operator. On the right of the operator, the film ID is missing. The policy engine determines that for `["VHS-2209", title]` to equal `[film_id, "Pulp Fiction"]`, `film_id` must be `"VHS-2209"` and `title` must be `"Pulp Fiction"`. Both variables are bound in one step, and `result` becomes `["VHS-2209", "Pulp Fiction"]`.

Evaluate this in the playground, and you'll see that the policy engine fills in both elements. Unification solved for the unknowns instead of you assigning them line by line.



Unification solves both variables in one step. Each position is matched by the known value on the opposite side.

This bidirectional solving is called *unification*, borrowed from logic programming languages like Prolog. It is powerful, but it can make a rule harder to trace because expression order matters less than with `:=` followed by `==`.

When to use "="

Unification has its place, but the recommended style in modern Rego is to use `:=` and `==` unless you specifically need unification's bidirectional solving.

The most common legitimate use of `=` is pattern matching against structured values. For example, pulling values out of an array in a single expression:

```
package rental

# The request timestamp uses ISO 8601: "1995-11-09T18:30:00Z"
# time.date returns [year, month, day]

evening_rental if {
  request_ns  := time.parse_rfc3339_ns(input.request.timestamp)
  [_, _, _]    = time.date(request_ns)
  [hour, _, _] = time.clock(request_ns)
  hour >= 18
}
```

The `=` on the `time.clock` line simultaneously pattern-matches the three-element array and binds `hour`. You could achieve the same with `:=` and explicit indexing (`hour := time.clock(request_ns)[0]`), but the destructuring form is more expressive when you only need some elements.

Another valid use is when you need to express a query rather than a sequential computation:

```
package rental

# Find all rentals whose film_id matches a film in the catalogue
matched_rentals contains rental if {
  some rental in input.member.rentals.items
  some film in data.catalogue
  rental.film_id = film.id
}
```

You can add the following to the **Data** section of the Rego playground to test the query:

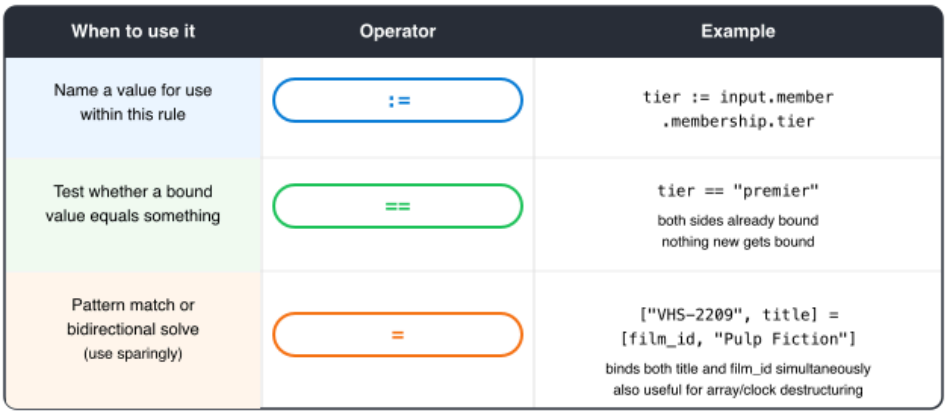
```
{
  "catalogue":[{
    "id": "VHS-0871"
  }]
}
```

Then run the policy with Alex's input (his *Speed* rental uses `VHS-0871`). OPA returns the single rental record whose `film_id` matches a film in the catalog. Remove that film from `catalogue` and evaluate again. This time, the set is empty. Here, the order of lines does not matter; OPA finds bindings that make `rental.film_id` and `film.id` equal. That's the query-style that unification enables.

The operator comparison table

	`:=`	`==`	`=`
Binds new variables	Yes	No	Yes (if unbound)
Tests equality	No	Yes	Yes
Requires both sides bound	No	Yes	No
Compiler error if var already assigned	Yes	No	No
Recommended default	✓ for assignment	✓ for comparison	Only when needed

Under the hood, `:=` and `==` are syntactic sugar built on top of `=` plus some extra compiler checks. The extra strictness is a feature that catches mistakes early.



Start with := and ==. Only add = when you genuinely need pattern matching or a query-style join.

Destructuring with assignment

The employee doesn't always need every field on a form, just the year someone joined, or whether membership is active. *Destructuring* binds several variables at once by matching the shape of a composite value.

Array destructuring

Use positional placeholders (_ for positions you do not need) to extract values from known positions:

```
package rental

# time.date(ns) returns [year, month, day]
joined_prior_to_this_year if {
  joined_ns   := time.parse_rfc3339_ns(input.member.membership.joined)

  # year=1993, month and day ignored
  [year, _, _] := time.date(joined_ns)

  year < 1994
}
```

This works with `:=` as well as `=`. The `:=` form is preferred for local bindings because it gives you the compiler safety checks described above. With Alex's membership joined date in the fixture input, evaluate `joined_prior_to_this_year`. If the parsed year is before 1994, the rule fires; otherwise, it doesn't. You get the year in one line without indexing `[0]` yourself.

Nested destructuring

Destructuring can reach into nested structures, but object patterns behave differently from array patterns. With arrays, you can ignore positions using `_`. With objects, the pattern must list every key on the value being matched. You also cannot put literal values on the left of `:=`; only variables (and `_`) are allowed there. Check literal values with `==` instead.

A practical pattern is to bind the nested object first, then destructure it:

```
package rental

classic_member if {
  membership := input.member.membership
  {
    "tier": tier,
    "active": active,
    "joined": _,
    "fees_outstanding": _
  } := membership
  tier == "classic"
  active
}
```

The first line reaches into `input.member` and the destructuring line binds `tier` and `active` while ignoring `joined` and `fees_outstanding` with `_`. Next, the `tier == "classic"` check filters out non-classic members. Evaluate this in the playground: The rule fires as the tier is `"classic"` and the membership is active. You can't destructure `input.member` in a single object pattern, because Alex's record also has other keys that are not in the pattern. That's why most rental checks use a few plain `:=` bindings at the top of a rule: they stay readable when the input document grows.

If you need to match literal values inside a pattern, use `=` rather than `:=` as unification allows literals on the left. Nested destructuring is expressive but easy to overuse. You can use it when the pattern check is itself part of the policy meaning, but avoid it as a shorthand for several `:=` lines you could write explicitly.

Comparison operators

Beyond equality, Rego supports the full set of ordering comparisons:

```
a == b    # equal
a != b    # not equal
a <  b    # less than
a <= b    # less than or equal
a >  b    # greater than
a >= b    # greater than or equal
```

These work on numbers, strings (lexicographic order), and, with some built-ins, on other types. Like `==`, they require both operands to be bound; they do not perform any variable binding themselves.

The age restriction at EmptyBox is a familiar combination: bind Alex's age and the film's minimum age, then compare:

```
package rental

age_permitted if {
  age         := input.member.age
  min_age     := to_number(input.film.rating)
  age >= min_age
}
```

Late fees use the same pattern against store configuration:

```
package rental

fees_acceptable if {
  fees := input.member.late_fees
  fees >= 0
  fees < data.store.max_late_fees_before_block
}
```

With Alex's `late_fees` of `1.50` and `max_late_fees_before_block` of `10` in the shared fixtures, `fees_acceptable` is true. Raise `late_fees` above the store limit in the input, or lower the threshold in the data, and the comparison fails because the rule doesn't fire. Bind first, compare second; the operators never invent values for you.

Common pitfalls

These mistakes often show up once rules grow beyond a single line.

Pitfall 1: Using "==" when you mean ":="

```
package rental

# Bug: this will error because 'tier' is not bound
p if {
  # should be :=
  tier == input.member.membership.tier
  tier != "classic"
}
```

If `tier` has not been declared with `:=` or appeared in another binding expression first, using `==` is a compiler error. Switch to `:=`.

Pitfall 2: Reassigning a variable

```
package rental

p if {
  count := input.member.rentals.current_count

  # error: count is already assigned
  count := count + 1
}
```

Variables in Rego are immutable once bound. To transform a value, bind the result to a new name:

```
package rental

p if {
  current_count  := input.member.rentals.current_count
  projected_count := current_count + 1
  projected_count <= input.member.rentals.max_allowed
}
```

The employee might think "add one more tape to the count on the pad," but the policy records `current_count` and `projected_count` as two facts rather than as a single slot that changes.

Pitfall 3: Reaching for "=" by default

Newcomers from Prolog or other logic languages sometimes use `=` everywhere. While it often works, it skips the compiler checks that `:=` and `==` provide. Using unification this way in Rego can make rules harder to reason about. Stick to `:=` and `==` as your defaults, and reach for `=` only when its bidirectional power is genuinely needed.

Pitfall 4: Forgetting that comparison does not bind

```
package rental

p if {
  # tests the tier, does not create a binding
  input.member.membership.tier == "classic"

  # error: 'tier' is not bound
  tier != "premier"
}
```

`==` on `input.member.membership.tier` does not bind a variable named `tier`. If you need the value bound, use `:=` first:

```
package rental

p if {
  tier := input.member.membership.tier
  tier == "classic"
  tier != "premier"
}
```

What comes next

You can now name values at the counter with `:=`, test them with `==`, and reach for `=` when pattern matching or catalog queries need unification. That is enough machinery for substantial rule bodies.

In the next chapter, we look at comprehensions, Rego's concise syntax for building composite values from subqueries. It's one of the most useful ways to turn Alex's rental list into sets, objects, and derived facts that the employee can act on.

Key concepts

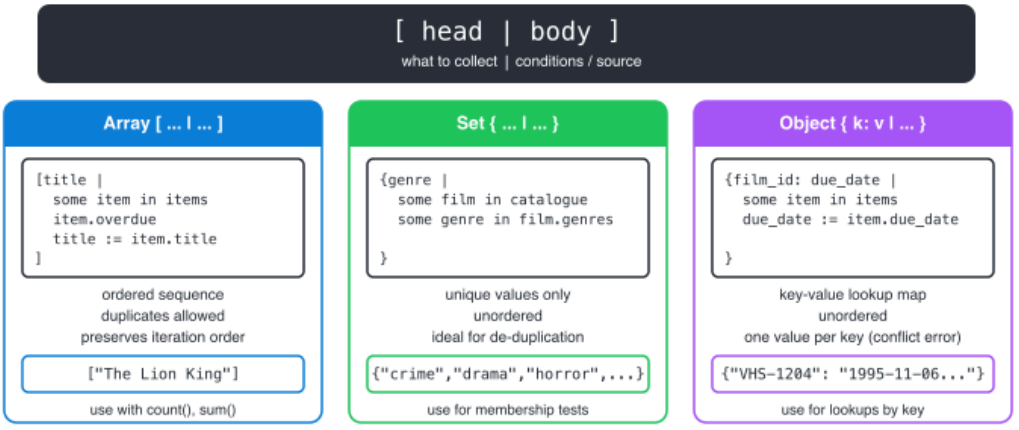
- `:=` declares and binds a local variable.
- Variables are immutable once bound, and the compiler enforces this.
- `==` compares two already-bound values but doesn't create bindings.
- `=` unifies: it binds unbound variables to make both sides equal. Use it sparingly, when you need bidirectional solving or pattern matching.
- The recommended style is `:=` for assignment and `==` for comparison. Only reach for `=` when you specifically need unification.
- Destructuring with `:=` or `=` lets you bind multiple variables from a composite value in a single expression.
- Comparison operators (`<`, `<=`, `>`, `>=`, `!=`) require both operands to already be bound.

CHAPTER 7

Comprehensions

In the previous chapter, you got the hang of using assignments, comparisons, and unification to write rule bodies. You now have the machinery for substantial rule bodies, but as things grow, you'll want ways to assemble collections of data before you reason about it. You could write separate named rules for each of these intermediate collections, but it gets verbose if you only need one collection at a time. That's where comprehensions are useful.

Comprehensions let you define collections inline, right where you need them, using a shorthand syntax. You can use comprehensions to generate collections like all the films Alex has, or get a list of genres without duplicates. This chapter covers array, object, and set comprehensions, which are the main ways Rego assembles composite data from sub-queries.



Array comprehensions

You might need to list every title on Alex's account before filtering to see overdue items. An array comprehension can create that ordered list in one step.

Comprehensions share the same shape as other rules:

```
[ <term> | <body> ]
```

Each time the body succeeds, the policy engine evaluates the head term and appends it to the result array.

All current titles

Let's build this step-by-step, starting with the unfiltered list:

```
package rental

current_titles := [title |
  some item in input.member.rentals.items
  title := item.title
]
```

If you run this in the pre-populated playground (**oc.to/rego-playground**), you'll see it in action. It walks `input.member.rentals.items`, binds each `item`, and sets `title := item.title`. The head collects all the titles.

```
{
  "current_titles": [
    "Speed",
    "The Lion King"
  ]
}
```

This shows the two rentals Alex currently has out.

Filtering to overdue titles

To filter the items, add a second condition to the body. Conditions are separated by a `;` character, like other rule expressions.

```
package rental

overdue_titles := [title |
  some item in input.member.rentals.items
  item.overdue
  title := item.title
]
```

This filters the items so only those that are overdue contribute to the `overdue_titles`. The result is a single-element array.

```
{
  "overdue_titles": [
    "The Lion King"
  ]
}
```

Array comprehensions preserve the order in which OPA evaluates the body. In practice, that usually follows the source collection, but it is not guaranteed in every case. If you need to sort the items, use the `sort` built-in (which you'll learn in Chapter 12).

The body iterates the source and filters items. The head collects values for each body that succeeds. Use count(), sum(), min(), or max() to aggregate a comprehension into a single number.

The Python parallel

If you have written Python list comprehensions, the shape will feel familiar:

```
# Python
overdue_titles = [item["title"] for item in items if item["overdue"]]
```

```
# Rego equivalent
overdue_titles := [title |
  some item in input.member.rentals.items
  item.overdue
  title := item.title
]
```

You can see the same ingredients in both versions: a head expression, a source to iterate over, and an optional filter. Rego puts the head first and uses `;` to separate body conditions, rather than Python's `for` and `if`.

Set comprehensions

In the film catalog, each film has a list of genres. You can use comprehensions to get the distinct genres as a set.

```
package rental

unique_genres := {genre |
  some film in data.catalogue
  some genre in film.genres
}
```

If you run this in the playground, you'll get a unique list of genres from the catalog.

```
{
  "unique_genres": [
    "animation",
    "children",
    "comedy",
    "crime",
    "drama",
    "horror",
    "thriller"
  ]
}
```

Order isn't significant for sets, and you'll only get an item once, even if it appears many times in the source. Set comprehensions are ideal for de-duplication.

Film IDs currently on Alex's account

You could use set comprehensions to check whether an account has already rented a film before allowing Alex to rent another copy of the same tape.

```
package rental

# Build the set of film ids currently rented by this member
currently_rented_ids := {film_id |
  some item in input.member.rentals.items
  film_id := item.film_id
}

not_already_rented if not input.film.id in currently_rented_ids
```

Alex has rented The Lion King (VHS-1204) and Speed (VHS-0871). As *Pulp Fiction* has the ID VHS-2209, `not_already_rented` is true. The membership test on the second line stays short because the comprehension already did the collection work.

Object comprehensions

You might also want to perform a quick lookup, like when is *The Lion King* due back? An object comprehension builds a map from keys to values in one expression:

```
{ <key>: <value> | <body> }
```

Each successful body iteration adds one entry.

Due dates keyed by film ID

```
package rental

# Map film_id to due_date for all current rentals
due_dates_by_id := {film_id: due_date |
  some item in input.member.rentals.items
  film_id  := item.film_id
  due_date := item.due_date
}
```

When you evaluate this in the playground, you'll see the following result:

```
{
  "VHS-0871": "1995-11-08T00:00:00Z",
  "VHS-1204": "1995-11-06T00:00:00Z"
}
```

You can query it directly, for example, `due_dates_by_id["VHS-1204"]` is `"1995-11-06T00:00:00Z"`, the overdue *Lion King* due date the employee is staring at.

No conflicting entries

Object comprehensions obey the same rule as partial object rules: one key, one value. If Alex's input ever listed the same `film_id` twice with different `due_date` values, OPA would raise a conflict error instead of choosing a winner. When duplicates are possible, and you need every value, collect into an array or set comprehension first, then process the result.

Referring to outer variables

Alex didn't ask for every film in the catalog, just *Pulp Fiction*. While we can't let Alex rent that movie, we could suggest a suitable alternative based on the genres. The comprehension body can read variables from the *outer* rule, which here is `input.film`, while it iterates the catalog:

```
package rental

# Find films in the same genre
related_film_titles := [title |

  # input.film.genres = ["crime", "drama"]
  some genre in input.film.genres

  some film in data.catalogue

  # shares a genre with the requested film
  genre in film.genres

  # not the same film
  film.id != input.film.id

  title := film.title
]
```

`input.film.genres` and `input.film.id` come from Alex's rent request. The comprehension only adds films that share one of the genres and are not *Pulp Fiction* itself. The policy engine evaluates outer bindings before the comprehension runs, so you can rely on `input` being fully available inside the body.

The same scoping applies when a comprehension sits inside a larger rule body. For example, mapping each overdue rental to how many days late it is:

```
package rental

# For each overdue item, compute how many days overdue it is
days_overdue_by_id[film_id] := days if {
  some item in input.member.rentals.items
  item.overdue
  film_id    := item.film_id
  request_ns := time.parse_rfc3339_ns(input.request.timestamp)
  due_ns     := time.parse_rfc3339_ns(item.due_date)
  days       := (request_ns - due_ns) / 1000000000 / 86400
}
```

Here, the partial object rule uses `input.request.timestamp` from the outer scope along with each `item.due_date`. For Alex's overdue *Lion King* and request time `1995-11-09T18:30:00Z`, you get the result:

```
{
  "VHS-1204": 3.7708333333333335
}
```

That number is what a late-fee or denial message might cite next.

Comprehensions for grouping and aggregation

Real policies rarely stop at a flat list. The counter often needs items *grouped* by some key, or a single *count* across a filtered collection. Comprehensions pair naturally with both.

Grouping

Split Alex's rentals into "current" and "overdue" buckets for a summary panel:

```
package rental

# Group rental items by overdue status
rentals_by_status := {status: items |
  some status in {"overdue", "current"}
  items := [item |
  some item in input.member.rentals.items
  item.overdue == (status == "overdue")
  ]
}
```

The outer object comprehension chooses each `status` key. The inner array comprehension fills `items` for that key. Nested comprehensions can look dense at first, but the pattern is worth recognizing: the outer level picks keys, and the inner level builds the value for each key.

```
{
  "rentals_by_status": {
    "current": [{
      "due_date": "1995-11-08T00:00:00Z",
      "film_id": "VHS-0871",
      "overdue": false,
      "title": "Speed"
    }],
    "overdue": [{
      "due_date": "1995-11-06T00:00:00Z",
      "film_id": "VHS-1204",
      "overdue": true,
      "title": "The Lion King"
    }]
  }
}
```

You've created a single object containing two lists.

Aggregation

When you need to know how many, rather than which ones, you can wrap a comprehension in one of the built-ins that aggregate, like `count`, `sum`, `min`, and `max`:

```
package rental

total_rentals  := count(input.member.rentals.items)

overdue_count  := count([item |
  some item in input.member.rentals.items
 item.overdue
])

current_count  := count({item.film_id |
  some item in input.member.rentals.items
  not item.overdue
})
```

For Alex: two rentals total, one overdue, one current:

```
{
  "current_count": 1,
  "overdue_count": 1,
  "total_rentals": 2
}
```

`count` on a filtered array comprehension is the idiomatic way to answer "how many match this condition?" without first building a named set rule.

`sum` needs numbers. If EmptyBox tracked replacement value per tape in the catalog, you could total the rental prices of everything Alex still has out:

```
package rental

# This requires film prices to be stored in the catalogue
total_rental_value := sum([film.rental_price |
  some item in input.member.rentals.items
  some film in data.catalogue
  film.id == item.film_id
])
```

Matching Alex's two items to catalog prices (`2.75` each) gives:

```
{
  "total_rental_value": 5.5
}
```

Chapter 12 covers the full family of aggregation built-ins; for now, remember that comprehensions produce the collection, and built-ins collapse it to a single number.

Comprehensions versus incremental rules

You can build `overdue_titles` as a comprehension or as an incremental set rule. Both work; the choice is mostly about reuse and readability.

Use an **incremental rule** when:

- The collection is referenced in multiple places
- The logic deserves its own name and tests
- You want to query or document it independently

```
package rental

# Defined as a rule, reusable, testable
overdue_titles contains title if {
  some item in input.member.rentals.items
  item.overdue
  title := item.title
}

# Easy to reference in multiple places
has_overdue_items if count(overdue_titles) > 0
deny contains msg if {
  has_overdue_items
  msg := sprintf("Member has %d overdue rental(s)", [count(overdue_titles)])
}
```

When you evaluate this in the playground, you'll get the following result:

```
{
  "deny": [
    "Member has 1 overdue rental(s)"
  ],
  "has_overdue_items": true,
  "overdue_titles": [
    "The Lion King"
  ]
}
```

Use a **comprehension** when:

- The collection is only needed once, inline
- The logic is short enough to read in place
- It is part of a larger expression (for example, inside `count` or `sum`)

```
package rental

# Inline — fine for a one-off use
deny contains msg if {
  n := count([item | some item in input.member.rentals.items; item.overdue])
  n > 0
  msg := $"Member has {n} overdue rental(s)"
}
```

You can use named incremental rules to tame growing policy files, as they are easier to test and locate.

Both produce identical results. The trade-off is compactness versus reusability and independent testability.

What comes next

Comprehensions let you assemble arrays, sets, and objects from any sub-query in one expression. You've seen it in action, obtaining titles on Alex's account, genres in the catalog, due-date lookups, grouped rentals, and counts for the deny message. Together with the rule forms from Chapter 5, you now have the main ways Rego builds composite data.

The next chapter takes up *negation*: how to say that something must *not* exist or must *not* be true. That idea shows up everywhere in rental policy ("no overdue items," "membership not active"), and it is one of the trickiest parts of declarative languages. Turn the page when you are ready to make those denials precise.

Key concepts

- Comprehensions build composite values inline from sub-queries. They have a head (what to collect) and a body (the conditions).
- Array comprehensions `[ term | body ]` produce ordered lists, including duplicates.
- Set comprehensions `{ term | body }` produce unordered collections of unique values.
- Object comprehensions `{ key: value | body }` produce key-value mappings. Conflicting keys cause an error.
- Comprehension bodies can reference variables from the outer rule scope, and the policy engine reorders expressions to ensure outer bindings are available.
- Use comprehensions for one-off inline values. Use incremental rules for collections that are reused or complex enough to benefit from a name.
- `count`, `sum`, `min`, and `max` work directly on comprehension results for aggregation.

Part III

Control and logic

CHAPTER 8

Negation

You can use comprehensions to build arrays, sets, and objects from sub-queries. You applied that knowledge to create lists of overdue titles, genre sets, and due-date maps. At the checkout desk, many of the checks are looking at what should be true, like the membership should be active or the film must be in stock. But you'll just as often want to make decisions about things that shouldn't be true, like no outstanding fees. In Rego, this is more than a syntax switch, because `not`, undefined values, and loop structures interact in ways that look correct, but deliver the wrong result. This chapter walks through negation in Rego and shows you how to avoid the loop mistake almost every newcomer makes.

Negation with "not"

Before Alex can rent any movie, the EmptyBox employee must ensure there are no unpaid fees. This is a negative check as you need to allow the rental, but not if there are outstanding fees. In Rego, you use the `not` keyword to negate an expression. If the expression would have been true, `not` makes it false. If the expression would have been false or undefined, `not` makes it true.

```
package rental

allow if {
  not input.member.membership.fees_outstanding
}
```

Evaluate this policy in the pre-populated playground (**oc.to/rego-playground**). You'll get the result:

```
{
  "allow": true
}
```

You can set the membership's `fees_outstanding` field in the input panel to true and re-evaluate the policy to see a false result. So far, so predictable. The subtle trip hazard is the third case, which happens if `fees_outstanding` is missing from the input object. This makes it undefined, which is negated to `true`, allowing Alex's rental.

```
# fees_outstanding absent → undefined → not ... is true
```

This makes `not` robust to missing fields. In the example, we're assuming that `fees_outstanding` will be present and set to true if there is an unpaid fee, and in all other cases, payments are up to date. You'll want to check and document any assumptions you make, and one way to do that is with tests, which you'll learn more about in Chapter 16. If you don't want this permissive default, there are ways to distinguish between absent and false later in this chapter.

expr evaluates to	not expr	meaning
true	false	condition holds, negation fails
false	true	condition fails, negation succeeds (as expected)
undefined (field absent)	true	field absent = permissive useful, but can surprise use object.get to control

not treats false and undefined identically. A missing field is permissive unless you guard with object.get.

Negating a rule

You can use `not` on any expression, including calls to other rules. If EmptyBox decides to lapse memberships after ten years, rentals would only be allowed if the expiry rule doesn't fire:

```
package rental

membership_expired if {
  joined_ns   := time.parse_rfc3339_ns(input.member.membership.joined)
  request_ns  := time.parse_rfc3339_ns(input.request.timestamp)
  years_held  := (request_ns - joined_ns) / 1000000000 / 86400 / 365
  years_held > 10    # memberships lapse after 10 years
}

allow if {
  not membership_expired
}
```

In the `membership_expired` rule, the join date and the request timestamp are parsed into date types. Then the number of years is calculated from them. If the result is greater than 10 years, the membership has expired. In our example, it's 1995, and Alex joined EmptyBox in 1993. That means the membership is well within the 10-year term, so `membership_expired` is false and `not membership_expired` is true. Defining a positive helper rule is a great way to make your rules easier to understand. Each piece of logic gets a name that you can query, and using `not` with that name is a clear way to negate the condition.

The safety rule

There's a constraint placed on negation in Rego, which requires that any variable that appears inside a negated expression must also appear in a non-negated expression elsewhere in the same rule body. This is called the safety rule. The safety rule ensures that every variable is grounded, meaning it has been bound to a concrete value before it's used. This will also flag if you have a variable that only appears head of a rule, or if you compare a variable that hasn't been bound.

When you have a positive expression, like `item.title == "Speed|`, the policy engine hunts for bindings of `item` that satisfy the condition. When you use `not item.title == "Speed"`, there is no finite search. There are infinitely many values that are not "Speed," and the policy engine can't enumerate them. The safety rule prevents this problem by requiring variables in negated expressions to be bound first.

```
package rental

# Safe: film_id is bound by the non-negated expression before the negated one
rentals_not_in_catalogue contains film_id if {
  some item in input.member.rentals.items

  # film_id is bound here
  film_id := item.film_id

  # safe: film_id is already bound
  not film_id in {f.id | some f in data.catalogue}
}
```

With Alex's rentals, `film_id` is bound from each item before the negated membership test runs. The policy engine checks whether that specific ID appears in the catalog set comprehension. There's no open-ended search.

The following would fail the safety check:

```
package rental

# Unsafe: film_id only appears inside the negated expression
p if {
  # error: film_id is not bound anywhere else
  not film_id == "VHS-2209"
}
```

The policy engine refuses to compile this and provides the error:

> var film_id is unsafe

You must bind `film_id` first with `:=` from a known source, then negate. In practice, the safety rule rarely blocks you once you are in the habit of binding from `input` or `data` before the `not`. If the compiler flags a violation, it usually means the rule needs a helper that establishes the positive case first. That's the same pattern you'll use for overdue rentals below.

The classic pattern: Membership negation

EmptyBox customers might have membership tiers that represent blocked accounts, like "suspended", "expired", or "banned" (which Alex might get if he keeps trying his luck). You can use a negation to ensure the membership tier isn't on the blocked list:

```
package rental

blocked_tiers := {"suspended", "expired", "banned"}

allow if not input.member.membership.tier in blocked_tiers
```

Alex's membership has a "classic" tier, which isn't blocked. So `not ... in blocked_tiers` is true. Change Alex's membership to "banned" and re-evaluate the policy.

As your membership logic grows, you can keep it maintainable by splitting the positive case into a named rule and negating it:

```
package rental

blocked_tiers := {"suspended", "expired", "banned"}

tier_is_blocked if input.member.membership.tier in blocked_tiers

allow if not tier_is_blocked
```

Both forms are equivalent. Prefer the helper when the check is complex enough to deserve its own name, or when you want to test `tier_is_blocked` independently.

The classic mistake: "!=" in a loop

Now it's time to look at the mistake that catches almost all Rego newcomers. EmptyBox introduces a policy to refuse a new rental while any tape is overdue. An intuitive first attempt might look like this:

```
package rental

# This does not mean what you think it means
no_overdue_rentals if {
  some item in input.member.rentals.items
  not item.overdue
}
```

When you read this code, it looks like it walks the rentals and is false if any are overdue, but that's not what happens. This is an existential operation, so it works if at least one item is not overdue. A single rental that isn't overdue results in `no_overdue_rentals` being true. In Alex's case, even though *The Lion King* is overdue, the rule fires because *Speed* isn't overdue.

The same bug appears with `!=`:

```
package rental

# BUG: same problem with !=
no_overdue_rentals if {
  some item in input.member.rentals.items

  # true as long as at least one rental is not overdue
  item.overdue != true
}
```

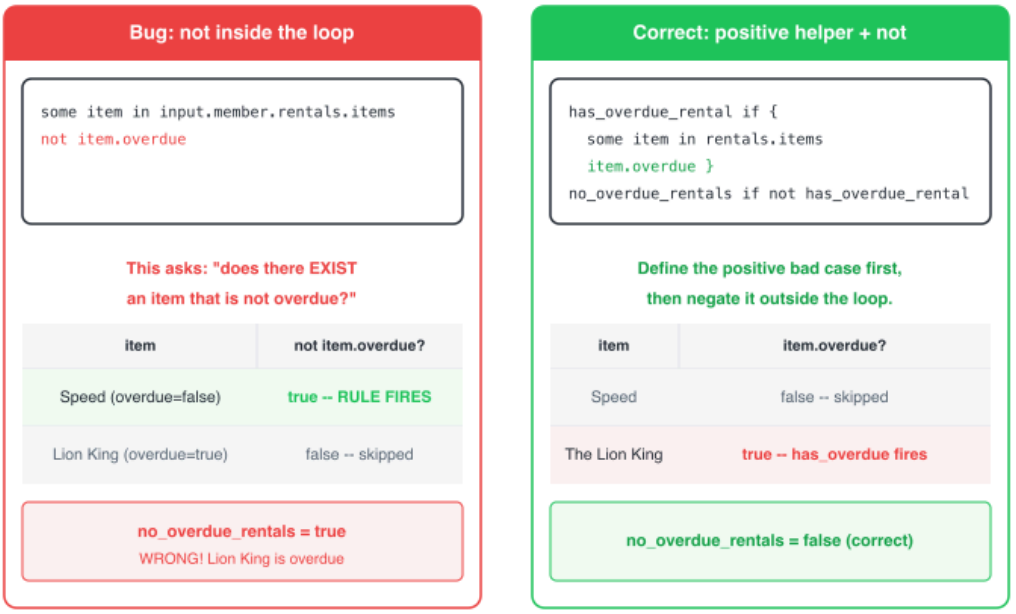
Variables in Rego are *existentially quantified*: A rule body that iterates with `some` asks "does there exist an element for which all conditions hold?" It does not ask "do *all* elements satisfy this condition?" To mean "no element is overdue," do not negate inside the loop. Define the positive bad case, then negate it.

```
package rental

# True if the member HAS any overdue rental
has_overdue_rental if {
  some item in input.member.rentals.items
  item.overdue
}

# True only if the member has NO overdue rentals
no_overdue_rentals if not has_overdue_rental
```

This version works where the previous attempts failed.

```
{
  "has_overdue_rental": true
}
```

The flag would become false only when all rental items are within their rental limits. Named rules made things more readable before, but in this case, they also prevent the mistake. Define the positive case (a bad thing exists), then negate it in your policy (no bad thing exists). You then avoid testing elements in isolation and instead ask whether the bad condition exists.



not inside a loop is existential ("there exists"). Define the positive bad case, then negate it outside the loop.

Chapter 9 (Universal Quantification) goes further on "for all" and "there exists"; for rental counters, this helper-plus-`not` shape is the workhorse.

Negation and undefined

It is worth being explicit about how `not` handles undefined, because it drives several EmptyBox defaults.

`not expr` is true in two situations:

1. `expr` evaluates to `false`
2. `expr` is undefined

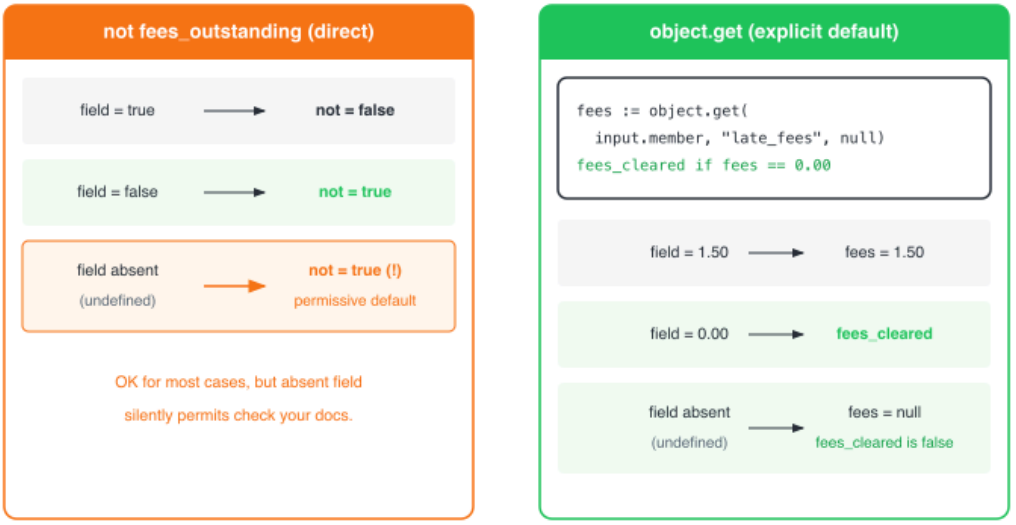
So `not` can express the *absence* of something, which is useful when the policy should stay permissive until a problem is explicitly present:

```
package rental

# True if the member has no late_fees field, or if it is false
no_fees_recorded if not input.member.late_fees
```

When `late_fees` is missing, it's undefined, and `not ... late_fees` is true. When you need to distinguish between `late_fees` being zero, and `late_fees` being missing, use `object.get` with a default value, like `null`:

```
package rental

late_fees       := object.get(input.member, "late_fees", null)
fees_cleared    if late_fees == 0.00
```

In most cases, the `not input.member.late_fees` is correct, but for security flags or audit trails, absence and falseness differ, and the explicit form using `object.get` is necessary.



For most policy checks the permissive default is fine. Use object.get when you need to distinguish absent from false.

Combining negation with other expressions

Negation composes with the rest of the Rego language. It works with built-in calls, comprehensions, and nested rule bodies. The three deny rules below extend the rental example with time-of-day checks, rental history, and fee thresholds:

```
package rental

# Deny if the request is outside store opening hours
deny contains "store is closed" if not store_is_open

store_is_open if {
  request_ns  := time.parse_rfc3339_ns(input.request.timestamp)
  [hour, _, _] := time.clock(request_ns)
  hour >= data.store.opening_hour
  hour < data.store.closing_hour
}

# Deny if the member has no rentals at all (nothing to return)
deny contains "no current rentals" if {
  not any_current_rentals
}

any_current_rentals if {
  some item in input.member.rentals.items
  not item.overdue
}

# Deny if the total late fees exceed the block threshold
deny contains "late fees too high" if {
  fees := input.member.late_fees
  not fees < data.store.max_late_fees_before_block
}
```

For a Saturday-afternoon request within `data.store` hours, `not store_is_open` is false, and the "store is closed" deny does not apply.

With `late_fees` at 4.50 and a block threshold above that in the fixtures, `not fees < data.store.max_late_fees_before_block` is false as well, so the late-fee deny stays off.

The middle deny lines up with however `any_current_rentals` evaluates on your input. This is worth stepping through in the playground alongside the overdue helpers above.

The last example is equivalent to `fees >= data.store.max_late_fees_before_block` and could be written that way; `not fees < threshold` is unusual style but valid. Occasionally, the negated form reads closer to the policy author's intent, especially when the positive side is a named helper like `store_is_open`.

What comes next

You now have a working model of negation. You know how `not` treats false and undefined, why the safety rule exists, how to negate membership in a set, and why `not` inside a `some` loop is not "for all." The next chapter builds on these ideas with *quantification*, which is how to say "there exists" and "for all" precisely in Rego, and when each form is the right tool at the counter.

When you've had a play with the examples and refreshed your beverage, turn the page to Chapter 9.

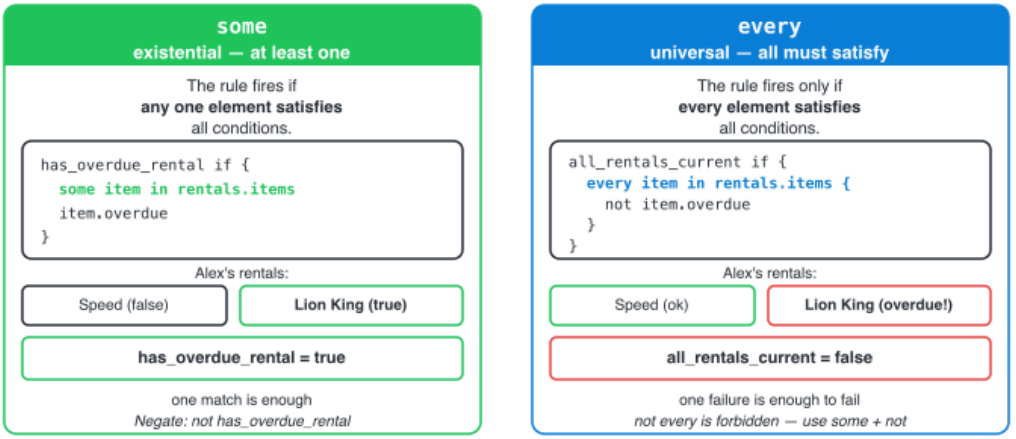
Key concepts

- `not expr` is true when `expr` is false or undefined. It treats missing fields the same as false.
- Variables inside a negated expression must also be bound by a non-negated expression in the same rule body, the safety rule.
- The correct pattern for "none of these should exist" is to define a positive helper (there exists a bad thing) and negate it.
- Using `!=` or `not` inside a loop checks whether *any* element fails the condition, not whether *all* do. This is one of the most common bugs in Rego policies.
- `not` composes with any expression: rule calls, built-ins, comprehension results.
- To distinguish "field is false" from "field is absent", use `object.get` with a default rather than relying on `not`.

CHAPTER 9

Quantification

In Chapter 8, you learned all about `not`. That included some good practices for creating named positive rules to negate safely, and for how false and undefined are handled. You also learned more about the policy engine's safety rules and why you should be careful with negation within loops. As part of this, you came across the idea of existential conditions, which are situations where the presence of at least one matching element makes the condition hold. This chapter will make you an expert in this topic.



some fires if any one element qualifies. every fires only if all elements pass. An empty collection is vacuously true for every.

Existential quantification with "some"

Before the employee can rent *Pulp Fiction* to Alex, they glance at the account. Is *any* current rental overdue? This is an example of an existential question, where one matching item is enough to confirm the answer. Variables in Rego are existentially quantified by default. When a rule body walks a collection, the policy engine asks whether there exists a binding for those variables that makes every body expression true. If there is, the rule fires. The `some` keyword makes that intent explicit. It declares a locally scoped existential, and the policy engine searches for values that satisfy the conditions.

```
package rental

has_overdue_rental if {
  some item in input.member.rentals.items
  item.overdue
}
```

You'll find Rego rules work well when you read them aloud. For this example, the sentence is "has overdue rental if some item in the rental items is overdue." The `some` keyword is a useful reminder that this applies if any item is overdue.

Alex has *The Lion King* marked overdue, so `has_overdue_rental` is true. Evaluate it in the pre-populated playground (**oc.to/rego-playground**), and you'll get the result:

```
{
  "has_overdue_rental": true
}
```

If you clear the overdue flag on *The Lion King* and the same query becomes undefined, because the rule no longer holds.

"some" with index variables

If you need the position of matches during an existential condition, the `some` keyword lets you declare index or key variables for bracket-notation access.

```
package rental

overdue_indices contains i if {
  some i
  input.member.rentals.items[i].overdue
}
```

In this example, `i` is the array index. When you use `some i` it declares a fresh local variable, so you won't accidentally collide with package-level rules named `i`. That's a real risk in larger policy files. If you run this in the playground, you'll receive an array of overdue array indices (if you remembered to change *The Lion King* back to being overdue!)

```
{
  "overdue_indices": [
    1
  ]
}
```

If there are no overdue items, you'll have an empty set.

The "in" operator

The `in` operator expresses membership and, with `some`, iteration with explicit binding. In the example below, we check whether the film's rating is in a set of allowed ratings.

```
package rental

allow if input.film.rating in {"U", "PG", "12"}
```

As Alex is trying to rent *Pulp Fiction*, the rule doesn't fire. If you change the film's rating to "12", you can see the rule fire.

Iteration with `some`: Walk each element and collect what matches:

```
package rental

overdue_film_ids contains film_id if {
  some item in input.member.rentals.items
  item.overdue
  film_id := item.film_id
}
```

Each overdue item found in the rental items will contribute its `film_id`, which for Alex is just "VHS-1204".

```
{
  "overdue_film_ids": [
    "VHS-1204"
  ]
}
```

You can use the two-variable form to collect the index and value together. For arrays, you can collect the index and the item using `some i, item` and for objects, you can collect the key and value using `some key, val`.

```
package rental

# i is the index, item is the value
overdue_indices contains i if {
  some i, item in input.member.rentals.items
  item.overdue
}

# key is the field name, val is the field value
non_null_member_fields contains key if {
  some key, val in input.member
  val != null
}
```

You may need to use parentheses in some contexts to avoid parsing ambiguity, like when you're within array or set literals. Within rule bodies, they aren't needed.

Evaluate the example, and you'll get the following result:

```
{
  "non_null_member_fields": [
    "age",
    "id",
    "late_fees",
    "membership",
    "name",
    "rentals"
  ],
  "overdue_indices": [
    1
  ]
}
```

Universal quantification with "every"

There are times you don't want to check for at least one matching element. Sometimes you need to make sure every item passes. For example, "Has every rental been returned on time?"

This is called universal quantification, and you apply it with Rego's `every` keyword:

```
package rental

all_rentals_current if {
  every item in input.member.rentals.items {
  not item.overdue
 }
}
```

For this rule to hold, every item must have `overdue` equal to false (or not present). Alex has an overdue film, so the rule won't hold, but you can remove the overdue entry for *The Lion King* to see an example where it does. When a collection is empty, `every` is vacuously true because there are no elements that fail to match the condition. This is typically correct, as if Alex has no rentals, none of them can be overdue.

"every" with a key

Like `in`, `every` supports a two-variable form that exposes the index alongside the value:

```
package rental

all_items_have_valid_ids if {
  every i, item in input.member.rentals.items {
    i < input.member.rentals.max_allowed
    regex.match(`^VHS-\d{4}$`, item.film_id)
  }
}
```

When you declare an index variable, you must use it in the body, or the policy engine reports `declared var i unused`. You can try that out by removing the `i < input.member...` line from the policy.

In real policies, you might use the index to enforce ordering, slot limits, or alignment with another array.

"every" does not introduce bindings

Variables bound inside an `every` block are not available outside it. The keyword evaluates its body for each element but does not export bindings to the surrounding rule:

```
package rental

p if {
  every item in input.member.rentals.items {
    not item.overdue
  }

  # item is NOT available here as it was scoped to the every block
}
```

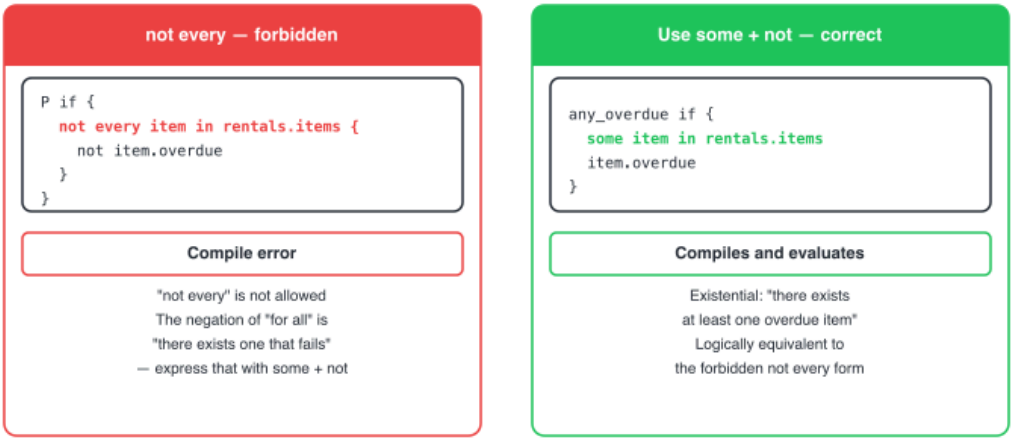
If you need values from the collection outside the `every` block, bind them separately with `some`, a comprehension, or another rule before or after the universal check.

Negating "every"

Rego forbids `not every ...`. The negation of "for all x, P(x)" is "there exists some x where not P(x)," and Rego already expresses that with `some` and `not`. The compiler rejects `not every` and points you at the alternative:

```
package rental

# Forbidden:
P if {
  not every item in input.member.rentals.items {
    not item.overdue
  }
}

# Correct
any_overdue if {
  some item in input.member.rentals.items
  item.overdue
}
```

For Alex's input, `any_overdue` is true; that's the same existential fact as `has_overdue_rental`, stated without nesting `not` around `every`.



The negation of "for all x: P(x)" is "there exists x: not P(x)". Express that with some + not, not not every.

Three ways to express "for all"

EmptyBox might require that *every* genre on the requested film is off the store's flagged list (`data.flagged_genres` contains `["horror"]`). *Pulp Fiction* carries `crime` and `drama`, so a well-formed "all clear" rule should pass for Alex. Rego offers three equivalent styles; each has trade-offs.

Option 1: "every"

```
package rental

all_genres_permitted if {
  every genre in input.film.genres {
    not genre in data.flagged_genres
  }
}
```

This is the most readable form as the requirement reads like an employee's checklist. Use `every` when the condition is simple, and the rule should read as natural language.

Option 2: negation + existential

```
package rental

film_has_flagged_genre if {
  some genre in input.film.genres
  genre in data.flagged_genres
}

all_genres_permitted if not film_has_flagged_genre
```

This is more verbose, but the positive case has its own name. `film_has_flagged_genre` is independently testable, and the final rule states the denial plainly. You can use this form when the positive case is complex, has been reused elsewhere, or warrants naming for clarity, much like `has_overdue_rental` and `no_overdue_rentals` in the sample policy.

Option 3: comprehension + count

```
package rental

all_genres_permitted if {
  flagged := {g | some g in input.film.genres; g in data.flagged_genres}
  count(flagged) == 0
}
```

This builds the set of violating genres and checks it is empty. It is less immediately readable than `every`, but you keep the violators in `flagged` for debugging or denial messages. That means you know *which* genres failed, not only whether they did. Swap in *Death in Brunswick* from the catalog (it includes `horror`) and `flagged` becomes non-empty; `all_genres_permitted` fails.

Choosing between them

Approach	Best for
`every`	Simple conditions; intent reads naturally
Negation + helper	Complex conditions; helper is reused or tested independently
Comprehension + count	When you need the set of violating elements, not just a boolean

All three are correct. `every` exists to make universal quantification readable; when in doubt, start there.

Approach	Example	Best for

every
built-in keyword
	`all_genres_permitted if {`
	`  every genre in film.genres {`
	`  not genre in flagged_genres`
	`  }`
	`}`
		Simple conditions
		Intent reads naturally.
		Start here by default.

negation
+
helper rule
	`film_has_flagged if {`
	`  some genre in film.genres`
	`  genre in data.flagged_genres`
	`}`
	`all_genres_permitted if not film_has_flagged`
		Complex or reused logic
		Helper is independently testable with opa test.
		Clear naming for denial reasons.

comprehension
+
count
	`all_genres_permitted if {`
	`  flagged := {g | some g in film.genres;`
	`  g in data.flagged_genres}`
	`  count(flagged) == 0`
	`}`
		Need the violating elements
		flagged contains the failures for debugging or denial messages.

All three express the same logical condition. every is most readable; the others trade brevity for testability or detail.

Existential inside universal: mixed quantification

Real policies combine both quantifiers. EmptyBox might insist that every tape on Alex's account appears in the catalog:

```
package rental

all_rentals_catalogued if {
  every item in input.member.rentals.items {
    some film in data.catalogue
    film.id == item.film_id
  }
}
```

The outer `every` walks rental items (universal). Inside each iteration, `some film in data.catalogue` searches for at least one catalog row whose `id` matches (existential). The rule holds only if, for every rental, such a row exists. Alex's `VHS-0871` and `VHS-1204` both appear in the data, so `all_rentals_catalogued` is true for his input. The reverse nesting of an outer existential with an inner universal is less common but equally expressible. "Is there a rental whose film has only approved genres?"

```
package rental

some_rental_fully_approved if {
  some item in input.member.rentals.items
  some film in data.catalogue
  film.id == item.film_id
  every genre in film.genres {
    not genre in data.flagged_genres
  }
}
```

Here `some` picks a rental and a catalog match; `every` then requires all of that film's genres to clear the flagged list.

A note on "some" versus implicit existentials

Older Rego often omits `some` and relies on implicit existential quantification:

```
package rental

# Older style with implicit existential
has_overdue if {
  input.member.rentals.items[_].overdue
}

# Modern style with explicit some
has_overdue if {
  some item in input.member.rentals.items
  item.overdue
}
```

Both are valid and produce the same result for Alex's account. Prefer the modern style: explicit intent, no accidental name capture, and support for the two-variable `in` form. When you read legacy policies, the underscore form is common; you can read it as "there exists some index where this holds."

What comes next

You can now state existential and universal conditions in Rego without conflating them. The next chapter moves from conditions to computation: user-defined functions let you name and reuse logic that goes beyond a single rule body.

Key concepts

- Variables in Rego are existentially quantified by default: a rule fires if *any* binding of the variables satisfies all conditions.
- `some` makes existential quantification explicit and prevents accidental name capture. Use it as the default style.
- The `in` operator tests membership and, with `some`, iterates over collections with one or two variable bindings.
- `every` expresses universal quantification: every element in the collection must satisfy the body. An empty collection is vacuously true.
- Variables bound inside an `every` block are not available outside it.
- `not every` is forbidden. Use `some x ...; not P(x)` to express "there exists a failing element."
- Three approaches to "for all": `every` (most readable), negation + helper (most testable), comprehension + count (when you need the failing elements).

CHAPTER 10

Functions

You've completed Chapter 9, and you're halfway on your journey to becoming a Rego pro. Your next step takes you into functions, which are reusable named chunks of logic. You can reference functions throughout your policies, so you only need to define their logic once. They are also easy to test in isolation from your policies. This chapter covers how to define functions, what they may return, and how they differ from rules.

Defining a function

The film's age rating on the shelf label might not match what lands in the computer. You might need to handle differences caused by manual data entry in poorly written user interfaces: `"18"`, `" 18"`, or `"18 "`. While the employee would normalize these in their head (if they even notice), policies are precise, and the differences will matter.

A function will help you capture the subconscious normalization process and use it whenever you need it. Functions are similar to rules, but have a parameter list in parentheses after the name.

```
package rental

# A function that normalizes a rating string to uppercase, trimmed
normalize_rating(rating) := result if {
  trimmed := trim(rating, " ")
  result  := upper(trimmed)
}

rating := normalize_rating("   18   ")
```

The function `normalize_rating` takes one input named `rating`, and returns one output, `result`. Like a rule body, the function body has a set of conditions and assignments that must all hold for the function to produce its value.

If you evaluate this example, you'll get the result:

```
{
    "rating": "18"
}
```

You can call it the same way you would call a built-in function:

```
package rental

normalize_rating(rating) := result if {
  trimmed := trim(rating, " ")
  result  := upper(trimmed)
}

age_permitted if {
  rating  := normalize_rating(input.film.rating)
  min_age := to_number(rating)
  input.member.age >= min_age
}

deny if not age_permitted
```

Evaluate this policy in the pre-populated playground (**oc.to/rego-playground**), and you'll get the result:

```
{
  "deny": true
}
```

Alex is 16, the film's rating is 18, so Alex isn't allowed to rent it. If you temporarily update Alex's age to 18 or above, you'll see the rental is allowed.

Functions let you name the concept and centralize it, to avoid replicating logic like the string handling throughout your policies. Functions have access to `input` and `data`, which are part of the global state, so they aren't pure functions in the mathematical sense. You can choose to write functions that only read from their named parameters if you wish. The example below compares a function that uses global state with one that asks callers to supply information.

```
package rental

is_flagged_genre(genre) if {
  genre in data.flagged_genres
}

is_flagged_genre_alternative(genre, flagged) if {
  genre in flagged
}

rental_denied if {
  some genre in input.member.rentals.genres
  is_flagged_genre(genre)
  is_flagged_genre_alternative(genre, data.flagged_genres)
}
```

Inputs and outputs

Functions in Rego take one or more inputs and return exactly one output. That is stricter than many languages, which allow functions to return multiple values or tuples. If you need multiple results at once, wrap them in an array, an object, or a set.

Alex's overdue *Lion King* rental is a natural place to bundle a fee and a label the employee might print on the account screen:

```
package rental

severity_label(days_overdue) := "high"   if days_overdue > 7
severity_label(days_overdue) := "normal" if days_overdue <= 7

# Returns an object with both the late fee and a severity label
late_fee_summary(days_overdue) := {
  "fee":      days_overdue * 0.50,
  "severity": severity_label(days_overdue),
}

summary := late_fee_summary(3)
```

Evaluate this example in the playground and you should see `{"fee": 1.5, "severity": "normal"}`. Bump `days_overdue` past 7 and the same call returns `"high"` severity with a larger fee—one object, one unambiguous answer per input.

The output term—the value after `:=` in the function head—can be any Rego value. If you omit it entirely, the output defaults to `true`, and the function behaves like a boolean predicate:

```
package rental

# Returns true if the film_id matches the expected format, undefined otherwise
valid_film_id(id) if {
  regex.match(`^VHS-\d{4}$`, id)
}

deny contains "invalid film ID" if {
  not valid_film_id(input.film.id)
}
```

EmptyBox tapes carry IDs like `VHS-0042`. Called as a condition in a rule body the validation reads naturally.

If `input.film.id` is not a valid VHS ID, `valid_film_id` is undefined, `not valid_film_id` is true, and the denial fires. A well-formed ID makes `valid_film_id` true, so `not valid_film_id` fails and that denial reason stays out of the set.

Function arguments can be composite

Arguments are not limited to scalars. You can pass arrays, objects, or sets, and you can use pattern matching in the argument position:

```
package rental

# Extract title and rating from a film object by matching its structure
film_summary({"title": title, "rating": rating}) := $"{title} (rated {rating})"
```

This function expects an object with `"title"` and `"rating"` keys. If the argument matches that structure, `title` and `rating` are bound immediately. If the argument does not match, the function is undefined for that input. Pass `input.film` from the fixtures and evaluate `film_summary(input.film)`; you should get a single string such as `"Pulp Fiction (rated 18)"`. Pattern matching in the argument position is a concise alternative to structural checks in the body—use it when the shape of valid inputs is fixed and worth documenting in the signature.

The one-output constraint

Functions must produce exactly one output value for any given set of inputs. If a function's body can bind the output variable to multiple different values, OPA raises a conflict error. That often shows up when the body iterates over a collection without collecting results:

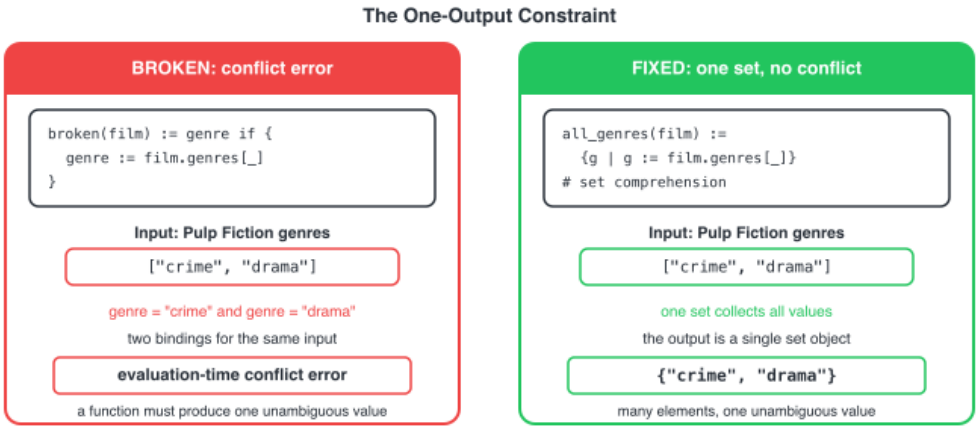
```
package rental

# Error: genre can be bound to "crime" or "drama" for the same input
broken(film) := genre if {
  genre := film.genres[_]
}

result := broken(input.film)
```

This fails because `genre` would take two different values—one for each genre in the array—for the same input film. A function must be unambiguous; the employee cannot print two different "the genre is ..." answers for one tape.

The fix is to collect the multiple values into a single composite:

```
package rental

# Fine: returns a set containing all genres
all_genres(film) := {g | g := film.genres[_]}

# {"crime", "drama"}
result := all_genres(input.film)
```

Now the output is one set, even though it contains several elements. Evaluate `all_genres(input.film)` when the film object has multiple genres; you get one set value, not an error.



Iterating without collecting binds the output to multiple values; a set comprehension wraps them into one value.

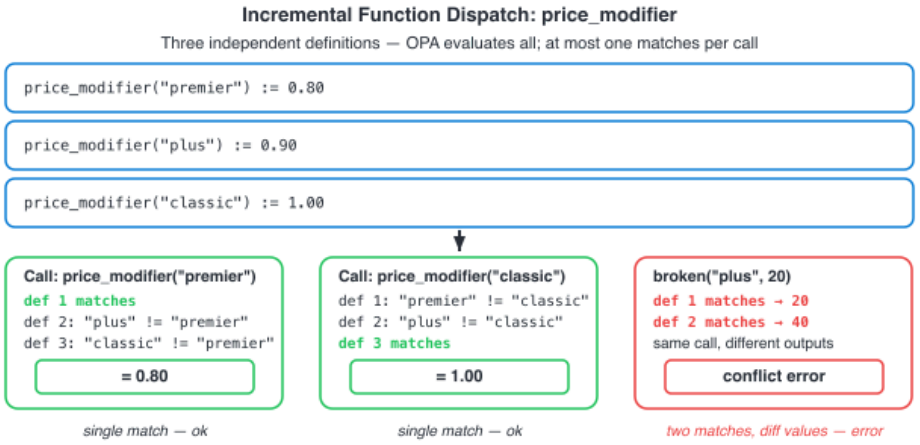
Incremental function definitions

Just like rules, functions can be defined multiple times with the same name. Each definition is an independent case, and OPA evaluates all of them. If any definition fires and produces a value, that is the result. If multiple fire and produce the same value, that is fine. If multiple fire and produce *different* values, that is a conflict error.

That pattern fits tier-based dispatch at EmptyBox—the same function name, different answers depending on membership:

```
package rental

# Rental price modifier based on membership tier
price_modifier("premier") := 0.80   # 20% discount
price_modifier("plus")    := 0.90   # 10% discount
price_modifier("classic") := 1.00   # no discount
```

Calling `price_modifier("plus")` matches only the second definition and returns `0.90`. Calling `price_modifier("premier")` matches only the first and returns `0.80`. Alex's fixture tier is `"classic"`, so `price_modifier(input.member.membership.tier)` evaluates to `1.00` with no discount.



OPA evaluates all definitions; one match is fine, zero matches is undefined, two matches with different values is an error.

A fuller counter check: how many days may a rental run before the store treats it as late?

```
package rental

max_rental_days("premier") := 14
max_rental_days("plus")    := 7
max_rental_days("classic") := 3

rental_deadline if {
  tier  := input.member.membership.tier
  limit := max_rental_days(tier)
  input.film.rental_period_days <= limit
}
```

Each tier gets its own limit in a one-liner. Adding a new tier is a single new line without touching the others. With Alex on `"classic"` and a seven-day request on *Pulp Fiction*, compare `input.film.rental_period_days` to `max_rental_days("classic")`—if the period exceeds 3, `rental_deadline` does not hold.

Conflict between incremental definitions

If a call matches multiple definitions and they produce different outputs, OPA raises an error at evaluation time. That is deliberate: a function's result must be unambiguous for any given input.

```
package rental

# Both definitions match when tier="plus" and age=20
# but produce different outputs
broken("plus", x)  := x
broken(x, 20)      := 40

# conflict error
result := broken("plus", 20)
```

Evaluate `broken("plus", 20)` and OPA reports a conflict—two definitions matched with different values for the same call.

If a call matches no definition, the function is undefined, not an error—just no result. That matters when functions are used as conditions:

```
package rental

max_rental_days("premier") := 14
max_rental_days("plus")    := 7

# If input.member.membership.tier is "classic", max_rental_days is undefined,
# and this rule does not fire
rental_within_limit if {
  input.film.rental_period_days <= max_rental_days(input.member.membership.tier)
}
```

A `"classic"` member with only the two definitions above leaves `max_rental_days` undefined, so the comparison in `rental_within_limit` never succeeds. Use `default` (next section) when you want a fallback for unmatched inputs.

Default functions

The `default` keyword works with functions just as it does with rules. A default function definition provides a fallback value when no other definition matches:

```
package rental

default max_rental_days(_) := 3

max_rental_days("premier") := 14
max_rental_days("plus")    := 7
```

Now `max_rental_days("classic")` returns `3` instead of being undefined. The default uses `_` for the argument, meaning it matches any input that no other definition covers.

Default function arguments follow the same rules as default rules:

- Arguments must be plain variables (or `_`), not composite patterns
- Argument names must not be repeated
- The default fires only when no other definition matches

One subtlety: a default function will not fire if any of its arguments are themselves undefined. Arguments are evaluated before the function is called, and an undefined argument halts the call entirely—the default does not rescue it:

```
package rental

default max_rental_days(_) := 3

# If input.member.membership.tier is undefined,
# max_rental_days(input.member.membership.tier) is undefined
# even with the default because the argument is evaluated first
limit := max_rental_days(input.member.membership.tier)
```

If you need to handle potentially missing input values, use `object.get` or a separate rule to supply a safe value before passing it to the function.

No arity overloading

Rego does not support two functions with the same name but a different number of parameters. That is a compile-time error:

```
package rental

# Error: arity mismatch
rental_summary(film)         := $"Film: {film.title}"
rental_summary(film, member) := $"Film: {film.title}, Member: {member.name}"

result := rental_summary(input)
```

OPA's type checker needs a fixed arity at compile time to verify calls. Two definitions with different arities would make that impossible.

When you genuinely need optional context—film only versus film plus member—the idiomatic workaround is a single object argument and field checks in the body:

```
package rental

rental_summary(args) := sprintf("Film: %s", [args.film.title]) if {
    not args.member
}

rental_summary(args) := sprintf(
    "Film: %s, Member: %s",
    [args.film.title, args.member.name],
) if {
    args.member
}

result := rental_summary(input)
```

That trades some compile-time safety for flexibility. Reserve it for cases where variable shape is unavoidable and the call sites are few and well documented.

Functions versus rules: when to use each

Functions and rules both encapsulate logic; the choice is not always obvious. These guidelines match how the rental counter actually behaves:

Use a function when the logic is parameterized and it must behave differently depending on a value that is not always read straight from `input` or `data`. Price modifiers, format validation, and tier lookups are function territory.

Use a rule when the logic always describes the current request against global `input` and `data`. Authorization outcomes, `deny` reason sets, and flags like `late_fees_ok` in `samples/rental/policy.rego` stay as rules.

Use a function when you call the same logic from several places with different arguments. `valid_film_id(id)` can check the requested film and each item in rental history. A rule named `valid_film_id` would hard-code `input.film.id` and could not be reused.

Use a rule when you want incremental definitions to build a collection. Functions cannot grow a set the way partial set rules can (Chapter 5).

In practice the boundary is intuitive: if you want to pass an argument, use a function. If you are stating a condition about Alex's request as it stands at the counter, use a rule.

Use a...	When...	Example
function	**Logic is parameterized** result depends on an argument, not just input/data	`normalize_rating(rating)`
function	**Same logic with different arguments** check the requested film AND each item in rental history	`valid_film_id(input.film.id)` `valid_film_id(item.film_id)`

rule	**Logic describes the current request**
	always reading from global input and data
	no argument needed — input is implicit
		`allow if {`
		`  membership.active ...`
		`}`

rule	**Need incremental set accumulation**
	multiple independent definitions each contribute to the same deny set
		`deny contains msg if {`
		`  not membership.active`
		`}`

Simple rule: if you want to pass an argument, use a function; if you are describing Alex's request, use a rule.

What comes next

Functions complete the core toolkit for expressing policy logic. The next chapter covers the control-flow keywords `default`, `else`, and `with`—fine-grained control over how rules evaluate and interact, including how to mock `input` and built-ins when you test policies in the playground or in `opa test`.

Key concepts

- Functions are parameterized rules. They take one or more inputs and return exactly one output.
- If the output term is omitted, the function returns `true`, behaving as a boolean predicate.
- Arguments can be composite values, and pattern matching in the argument position binds variables from the structure.
- A function must produce a single unambiguous output for any given input. Multiple bindings of the output variable cause a conflict error.
- Incremental function definitions express conditional dispatch. Each definition is an independent case.
- `default` provides a fallback value when no other definition matches. It does not fire if arguments are undefined.
- Rego does not support arity overloading. Use an object argument and field checks as a workaround when needed.
- Use functions for parameterized logic; use rules for logic that always operates on the global `input` and `data`.

CHAPTER 11

Control flow keywords

Having learned functions, you have the complete toolkit for reusable logic. You can use functions to organize your policies, to give meaningful names to concepts, and to keep your rules, the primary way to state what should be true for a request, clean and readable. This chapter talks about the keywords you've already seen in passing as you progressed through the book. Let's take a moment to dive deeper into each of these.

"default": fallback values

At the EmptyBox Rentals checkout desk, the employee needs an answer for when they can't tick every box for a rental. Instead of silence, they need to hear a "no". The same is often true of the applications and services that use a policy; they expect a `true` or `false` answer. The `default` keyword gives a complete rule a fallback value that the policy engine will provide if every other definition of that rule is undefined. You have been using it since Chapter 1, and here's how it behaves in the store policy.

```
package rental

default allow := false

allow if {
  input.member.membership.active
  not input.member.membership.fees_outstanding
  input.film.available_copies > 0
  input.member.age >= to_number(input.film.rating)
}
```

With `default allow := false`, querying `allow` always produces a value. Without it, the result would be empty, as Alex is 16 and doesn't pass the age check. With the default in place, a clear response is generated. You can run this example in the pre-populated playground (**oc.to/rego-playground**).

```
{
  "allow": false
}
```

You can try removing the `default` (line 3) to compare the output with and without it.

Syntax rules

A `default` declaration has a restricted syntax. The fallback must be a fixed value, not something computed from `input` or another rule:

```
default <name> := <term>
```

The term must be a scalar value, a composite value, or a comprehension. It cannot be a variable or a reference to another rule. That keeps default values simple and unconditional:

```
package rental

# scalar
default max_rental_days := 3

# composite
default approved_ratings := {"U", "PG"}

# boolean
default allow := false
```

The following are not allowed and will produce an error "illegal default rule (value cannot contain ref)":

```
package rental

# error: references not allowed
default allow := input.fallback

# error: references not allowed
default allow := other_rule
```

"default" and "undefined"

Because `default` only fires when all non-default definitions are undefined, any definition that produces a value will replace it. That includes values like `false` or an empty set. The policy engine only uses `default` as a last resort.

```
package rental

# fallback is true
default allow := true

# explicit false for suspended members
allow := false if {
  input.member.name == "Alex Murphy"
}
```

This policy returns `"allow": false`, demonstrating that the rule's false value takes precedence over the default.

When to use "default"

Use `default` for any complete rule where undefined is not an acceptable result. For example, authorization decisions, rental limits, and price calculations. Omit it when undefined is meaningful, such as when a calling rule should treat "not applicable" differently from "explicitly false."

"else": priority rules

Not every decision produces a single yes/no answer. The employee might assign a tier label or a fee-waiver cap by checking membership in order: premier first, then plus, then everyone else. The `else` keyword chains rule definitions in the order they appear. The policy engine evaluates each definition from top to bottom and stops at the first one that produces a value:

```
package rental

rental_tier_label := "premier"  if {
  input.member.membership.tier == "premier"
} else := "plus" if {
  input.member.membership.tier == "plus"
} else := "classic" if {
  input.member.membership.active
} else := "inactive"
```

If the rental tier is "premier", the first condition matches, the policy engine sets the label to "premier", and evaluation stops. Otherwise, it continues to the next rule, and so on. The final `else` with no body will produce a result when nothing else does. In this case, "inactive".

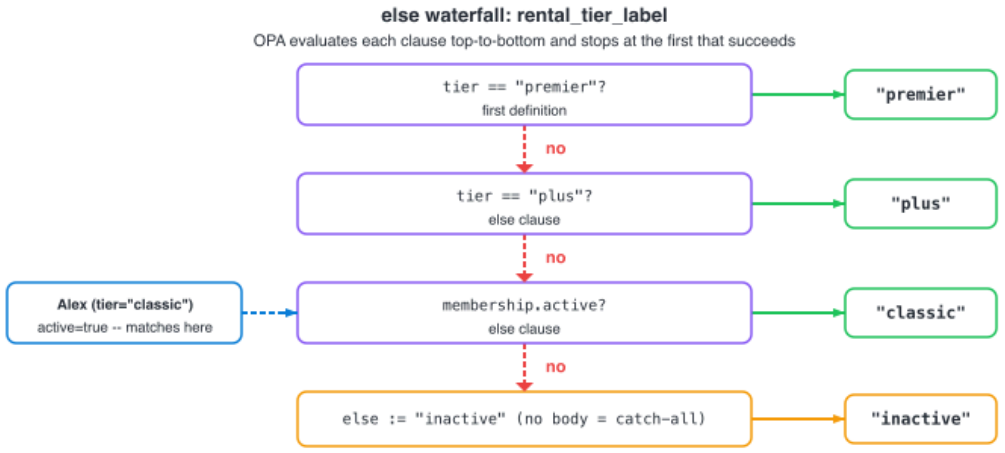
For Alex, the output shows the classic membership tier:

```
{
  "rental_tier_label": "classic"
}
```

A more policy-relevant use would be to set different maximum late-fee waiver limits based on the member's tier.

```
package rental

fee_waiver_limit := 10.00 if {
  input.member.membership.tier == "premier"
} else := 5.00 if {
  input.member.membership.tier == "plus"
} else := 2.00
```

As Alex has a classic membership, the fee waiver limit is `2.00`. You can change the membership tier in the input panel to see different results. This is Rego's equivalent of a switch statement or a chain of if-else-if in an imperative language, without naming a single "current step" variable.



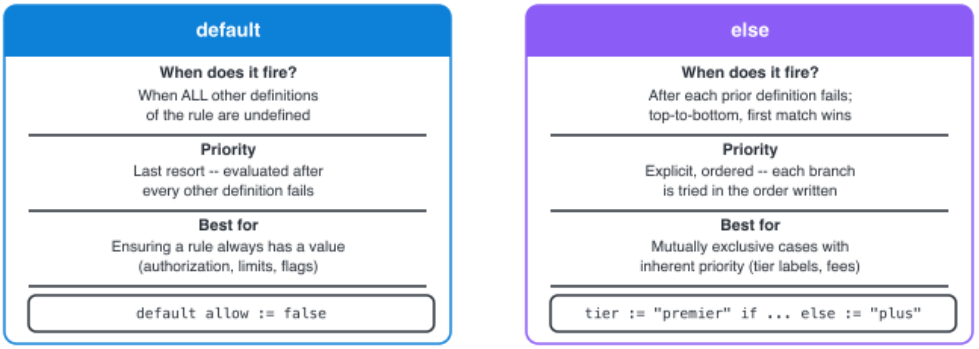
For Alex (tier "classic", active): the first two clauses fail, the third matches, evaluation stops; result is "classic". The bare else at the bottom is a catch-all that fires only when all conditions above it fail.

"else" versus incremental rules

The critical difference between `else` and multiple incremental definitions is *ordering*. Incremental definitions have no priority, and the policy engine evaluates all of them. For complete rules, the policy engine expects them to agree. In contrast, `else` definitions have explicit priority, and the policy engine stops at the first match. Use `else` when the definitions are mutually exclusive by intent but you do not want to repeat exclusion logic in every condition.

Use "else" sparingly

Because Rego's superpower lies in declarative rules, you should use `else` sparingly. Long chains are hard to reason about because each branch is the result of the accumulation of prior conditions. A good rule of thumb is to use `else` when the chain has two or three steps, and the priority is genuinely meaningful. If you find yourself writing five or six `else` clauses, consider helper rules with explicit conditions or a lookup object in `data`.



default fires when every other definition is undefined; else fires as a conditional fallback in a top-to-bottom priority chain.

"with": replacing values for a query

While Alex can't change their birthday, you do sometimes need to evaluate a policy to find out what the result would be under a simulated condition. What if Alex *was* 18? Would he be allowed to rent *Pulp Fiction*?

You can use the `with` keyword to override input, a path in `data`, or even a built-in function. The override only applies for the duration of a single expression. The most common use of `with` is when you write a test, though it's valid anywhere in a rule body.

Overriding "input"

```
package rental

allow if {
  input.member.membership.active
  not input.member.membership.fees_outstanding
  input.film.available_copies > 0
  input.member.age >= to_number(input.film.rating)
}

test_allow_active_member if {
  allow with input as {
    "member": {
    "age": 18,
      "membership": {
        "active": true,
        "fees_outstanding": false
      }
    },
    "film": {
      "rating": "18",
      "available_copies": 1
    }
  }
}
```

This evaluates `allow` as if `input` were the given object, regardless of what `input` is at query time. The override applies only to the expression it is attached to, which in this case is the call to `allow` inside the test rule.

You can override a path instead of the whole document:

```
package rental

allow if {
  input.member.membership.active
  not input.member.membership.fees_outstanding
  input.film.available_copies > 0
  input.member.age >= to_number(input.film.rating)
}

test_allow_older_member if {
  # Replaces the age (16) with a new age (18)
  allow with input.member.age as 18
}
```

When you override a path, the rest of the `input` object stays as it was. In the example, only Alex's age is changed and only within the `test_allow_older_member` definition.

Overriding "data"

The same syntax works for `data`. That is useful when testing policies that depend on configuration loaded into the policy engine's data store. You supply the configuration inline instead of loading it separately. There is a restriction: you cannot use `with` to partially override a virtual document (a document produced by a rule). You can replace the whole document or specific leaf values, but you cannot inject a value beneath a path that a rule already defines.

```
package rental

deny_reasons contains "fees outstanding" if
    input.member.membership.fees_outstanding

allow if count(deny_reasons) == 0

# Error: cannot partially override a virtual
# document produced by deny_reasons
result := r if {
  r := allow with data.rental.deny_reasons.fees_outstanding
    as ["fees outstanding"]
}
```

The policy engine refuses this because it would partially modify a virtual document. The error is the point: `with` replaces values; it does not splice into the rule-generated structure.

Overriding built-in functions

You can also use `with` to replace built-in functions with mock implementations. For example, to provide a predictable value for `time.now_ns()` so tests produce the same results regardless of when they run.

Say the policy calculates how many rental days remain:

```
package rental

days_remaining := days if {
  now_ns := time.now_ns()
  due_ns := time.parse_rfc3339_ns(input.rental.due_date)
  diff   := due_ns - now_ns
  days   := div(diff, 86400000000000)
}

test_days_remaining if {
  data.rental.days_remaining == 2
  with input as {"rental": {"due_date": "1995-11-11T18:30:00Z"}}
}
```

Without pinning the clock, a test asserting `days_remaining == 2` would pass the day it was written and then fail from the next day onward. Replacing `time.now_ns` with a mock fixes that:

```
package rental

days_remaining := days if {
  now_ns := time.now_ns()
  due_ns := time.parse_rfc3339_ns(input.rental.due_date)
  diff   := due_ns - now_ns
  days   := div(diff, 86400000000000)
}

mock_now_ns() := 815941800000000000   # 1995-11-09T18:30:00Z

test_days_remaining if {
  data.rental.days_remaining == 2
  with input as {"rental": {"due_date": "1995-11-11T18:30:00Z"}}
  with time.now_ns as mock_now_ns
}
```

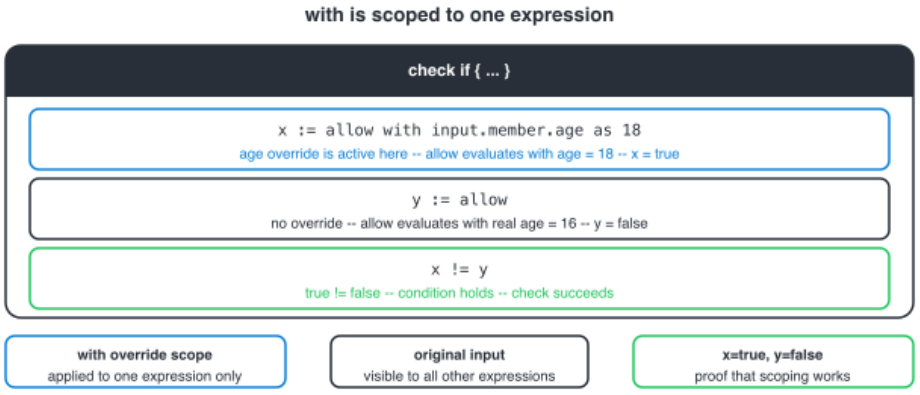
The replacement must have the same arity as the function it replaces. `time.now_ns` takes no arguments, so the mock takes none either. The return type must be compatible: the mock returns a number, matching what `time.now_ns` normally produces. Run the test in the playground with this package: `days_remaining` should be `2` for that due date and frozen "now." You can use any number of `with` modifiers in an expression. All overrides apply together for the duration of the expression.

Scope of "with"

`with` affects only the expression it is attached to. Later expressions in the same rule body see the original values. The example below uses `default allow := false`, so that when the age rule does not apply, `allow` is `false`, not undefined. That way `x != y` is a well-defined comparison. (Without `default`, `y := allow` would be undefined here, and the rest of the body would not succeed.)

```
package rental

default allow := false

# Rental allowed only when the member meets the film's age rating.
allow if {
  input.member.membership.active
  input.member.age >= to_number(input.film.rating)
}

# With input: active member aged 16, film rated "18" → `allow` is false
# `x` evaluates `allow` as if age were 18 (override applies only to that call)
# `y` evaluates `allow` with the real age 16, so `x != y`.
check if {
  x := allow with input.member.age as 18
  y := allow
  x != y
}
```

Evaluate `check` with Alex's input: `x` is true (age overridden to 18) and `y` is false (real age 16), so the rule succeeds and you have a concrete proof that `with` is scoped to one call.



The with override replaces age only for the expression it is attached to; the next line sees Alex's real age of 16.

"if" and "contains": readable rule heads

The `if` and `contains` keywords exist to make rule heads read "out loud". For example, `allow if` some condition matches.

"if"

The `if` keyword was optional in Rego's early days, but since version 1, the policy engine requires it before every rule body. You might see an example online that doesn't include it, so if you copy it, you now know why it fails and can add the `if` to get it working.

```
package rental

# Preferred: `if` before the body (required in v1)
allow if {
  input.member.membership.active
}

# Single-condition rules can be written on one line
allow if input.member.membership.active
```

The one-line form is particularly clean for simple conditions.

"contains"

The `contains` keyword marks a rule as a partial set rule.

```
package rental

deny contains msg if {
  not input.member.membership.active
  msg := "membership is not active"
}

deny contains msg if {
  input.member.rentals.current_count >= input.member.rentals.max_allowed
  msg := "too many rentals are out"
}
```

The `contains` form makes the intent clear: this rule adds members to a set called `deny`. If you stumble across an older policy, you might see this expressed in the retired form: `deny[msg] { ... }`.

Putting it together

Here is a policy that uses all four keywords together in the rental scenario. It denies by default, has a bypass for premier members, has tier label priority, and produces a list of denial reasons that might be shown on screen:

```
package rental

member := input.member
membership := input.member.membership
film := input.film

# Default: deny unless a rule explicitly allows
default allow := false

# Premier members bypass age restrictions
allow if {
  membership.active
  not membership.fees_outstanding
  film.available_copies > 0
  membership.tier == "premier"
}

# All other members must meet the age requirement
allow if {
  membership.active
  not membership.fees_outstanding
  film.available_copies > 0
  member.age >= to_number(film.rating)
}

# Tier label, in priority order (single `else` chain)
tier_label := "premier" if membership.tier == "premier"
else := "plus" if {
  membership.tier == "plus"
} else := "standard"

# Continued...
```

```
# ...continued

# Collect all denial reasons
deny contains msg if {
  not membership.active
  msg := "membership is not active"
}

deny contains msg if {
  membership.fees_outstanding
  msg := $"member has outstanding fees of £{member.late_fees}"
}

deny contains msg if {
  film.available_copies < 1
  msg := $"no copies of {film.title} are available"
}

deny contains msg if {
  membership.tier != "premier"
  member.age < to_number(film.rating)
  msg := concat(" ", [
    $"member is {member.age} years old",
    $"but {film.title} is rated {film.rating}",
  ])
}
```

`default` ensures `allow` always has a value. The `else` chain on `tier_label` establishes priority. `deny` uses `contains` to accumulate problems. `if` keeps each head readable.

Paste the policy into the playground with Alex's fixture input and query the package. You should see something like:

```
{
  "allow": false,
  "deny": [
    "member is 16 years old but Pulp Fiction is rated 18"
  ],
  ...
}
```

Alex is active, and the shelf has stock. So far, so good, except Alex is not a premier-tier member and is under 18. That makes `allow` false, and the age message appears in `deny`. If you change Alex's `tier` to `"premier"` and evaluate again: `allow` becomes true and the age denial drops out. That is the whole keyword set working as the employee's checklist would, but explicit and testable.

What comes next

You now have the complete Rego keyword set. The next chapter looks into the built-in function library. This is the standard toolkit of string operations, math, aggregation, encoding, and type checking that every policy author reaches for when the counter questions outgrow what `input` and `data` alone can express. Have a play with the final example in this chapter if you want to, then grab some popcorn and turn the page to learn about built-ins.

Key concepts

- `default` provides a fallback value for a complete rule when all other definitions are undefined. The term must be a literal value, not a reference.
- `else` chains rule definitions in priority order. OPA stops at the first definition that produces a value. Use it sparingly.
- `with` overrides `input`, paths under `data`, or built-in functions for a single expression. It is the primary tool for testing policies with controlled inputs.
- Multiple `with` clauses can be chained on a single expression. Overrides apply only to that expression; subsequent expressions see original values.
- `if` and `contains` structure rule heads in v1. `if` separates the head from the body (required in v1); `contains` marks a partial set rule. Legacy v0 omitted `if` and used bracket heads for sets; modern style uses both keywords.

Part IV

Built-in functions and code organization

CHAPTER 12

Built-in functions

Your toolkit now includes a range of keywords and smart ways to use them to write policies that are robust, readable, and ready for rental. When you're writing policies, you'll often need utilities to trim and compare strings, count collection, sum up fees, work with dates, and merge objects. That's where Rego's built-in functions come in to do the hard work.

The chapter contains the functions you're most likely to need to work with types and collections, perform encoding, and manage time. You can find a complete collection by visiting **openpolicyagent.org/docs/policy-reference**.

Every built-in uses the same calling shape. Value-producing built-ins assign their result; boolean predicates can sit directly in a rule body as conditions:

```
result := function_name(arg1, arg2, ...)
```

Or, for boolean predicates, called directly in a rule body without capturing the result:

```
allow if {
  startswith(input.film.id, "VHS-")
}
```

Evaluate `startswith(input.film.id, "VHS-")` in the pre-populated playground (**oc.to/rego-playground**): Alex's film is `VHS-2209`, so the condition holds. Change the ID to `DVD-2209` and `allow` will be undefined as the ID doesn't meet the condition.

Strings

Film titles, member IDs, and denial messages are all strings. Before Alex's age is compared to the rating on the label, someone has to normalize messy text; before a denial is shown on screen, someone has to phrase it clearly. The policy engine's string built-ins cover testing, transformation, splitting, joining, formatting, and pattern matching.

Testing

The employee checks whether an id *looks like* a member card or a tape before typing it in. These predicates express that habit:

```
# true if str begins with prefix
startswith(str, prefix)

# true if str ends with suffix
endswith(str, suffix)

# true if substr appears anywhere in str
contains(str, substr)
```

They return booleans, so you can use them as conditions in a rule body, or capture the result with `:=` when you need the value elsewhere:

```
package rental

allow if startswith(input.film.id, "VHS-")

deny contains msg if {
  not startswith(input.member.id, "MBR-")
  msg := $"Member ID {input.member.id} does not match expected format"
}
```

With Alex's fixture input, `startswith(input.film.id, "VHS-")` is true, and `startswith(input.member.id, "MBR-")` is true. That means the `allow` guard passes and the format denial does not fire. Temporarily change `input.member.id` to `"GUEST-0042"` in the playground, and the second rule will add a message to `deny`.

Transformation

Ratings and titles do not always arrive in the shape the policy expects. They might have extra spaces, inconsistent casing, or prefixes from another system. Transformation built-ins clean that up before comparison:

```
package rental

# "HELLO" → "hello"
a := lower("HELLO")

# "hello" → "HELLO"
b := upper("hello")

# "Hello!" → "Hello"
c := trim("Hello!", "!")

# " Hello " → "Hello"
d := trim_space(" Hello ")

# "Well Hello" → "Hello"
e := trim_prefix("Well Hello", "Well ")

# "Hello there" → "Hello"
f := trim_suffix("Hello there", " there")

# "Hello there" → "Howdy there"
g := replace("Hello there", "Hello", "Howdy")
```

Splitting and joining

Member IDs and human-readable denial text sometimes need to be taken apart or stitched together. You might need to split `MBR-0042` into prefix and number, or combine several words into one message for the register:

```
package rental

# "MBR-0042" → ["MBR", "0042"]
a := split("MBR-0042", "-")

# ["crime", "drama"] → "crime drama"
b := concat(" ", input.film.genres)
```

You'll always get an array from `split`, even when the splitting character doesn't appear in the string (you'd get a single entry of the whole string in that case). With `concat`, you'll always get a string, even if the array has no items.

Formatting

You sometimes need to format a value when you use it to create a message. You can use `sprintf` to achieve C-style formatting.

```
# C-style format string
sprintf(format, args)
```

```
package rental

# "Member Alex Murphy rented Pulp Fiction at £3.50 for 3 days"
message := sprintf("Member %s rented %s at £%.2f for %d days", [
  input.member.name,
  input.film.title,
  input.film.rental_price,
  input.film.rental_period_days,
])
```

In the pre-populated playground (**oc.to/rego-playground**) the message returned is:

> Member Alex Murphy rented Pulp Fiction at £3.50 for 3 days

For everyday denial and log lines, interpolation is usually enough:

```
package rental

# "Member Alex Murphy rented Pulp Fiction"
message := $"Member {input.member.name} rented {input.film.title}"
```

Regular expressions

We performed a prefix check by splitting the member ID earlier, but sometimes your requirements are more precise. For example, you might want to validate that the number has exactly 4 digits. You can use regular expressions to do this:

```
# true if str matches pattern
regex.match(pattern, str)

# capture groups
regex.find_all_string_submatch_n(pattern, str, n)

# replace matches
regex.replace(str, pattern, value)
```

Use raw strings (backticks) for patterns so backslashes are not doubled:

```
package rental

valid_member_id(id) if regex.match(`^MBR-\d{4}$`, id)
valid_film_id(id)   if regex.match(`^VHS-\d{4}$`, id)

deny contains "invalid member ID format" if not valid_member_id(input.member.id)
deny contains "invalid film ID format"   if not valid_film_id(input.film.id)
```

Alex's `MBR-0042` and `VHS-2209` both match; neither format denial fires on the default fixture. Try changing Alex's member ID to `MBR-42` in the playground to see the corresponding denial message.

```
{
  "deny": [
    "invalid member ID format"
  ]
}
```

Numbers and arithmetic

Late fees, price caps, and "how many days overdue" all need numbers. You can use standard arithmetic operators (`+`, `-`, `*`, `/`, `%`) to handle most situations, but there are also built-ins to cover rounding, ranges, and integer division:

```
# absolute value: abs(-5) → 5
abs(n)

# round up: ceil(1.2) → 2
ceil(n)

# round down: floor(1.8) → 1
floor(n)

# round to nearest: round(1.5) → 2
round(n)

# integer division: div(7, 2) → 3
rem(a, b)               # remainder: rem(7, 2) → 1
numbers.range(a, b)     # inclusive range: numbers.range(1, 5) → [1,2,3,4,5]
```

Alex still has an overdue *Lion King*; the policy can total late fees at £0.50 per day and keep rental prices within what the manager allows:

```
package rental

# Calculate the total late fee for overdue items at £0.50/day
late_fee_total := sum([(days_late * 0.50) |
  some item in input.member.rentals.items
  item.overdue
  request_ns  := time.parse_rfc3339_ns(input.request.timestamp)
  due_ns      := time.parse_rfc3339_ns(item.due_date)
  days_late   := ceil((request_ns - due_ns) / 1000000000 / 86400)
])

# Validate the rental price is within an acceptable range
valid_rental_price if {
  price := input.film.rental_price
  price >= 0.50
  price <= 10.00
}
```

With the fixture timestamps, the comprehension picks up the overdue item, and `ceil` turns fractional days into billable whole days. `valid_rental_price` holds for *Pulp Fiction* at £3.50. You can combine arithmetic with comprehensions when the amount depends on which rentals are in Alex's bag, not on a single scalar field.

Aggregates

EmptyBox Rental employees often think in totals. How many tapes are out? What are they worth together? Which due date comes first? Aggregate built-ins summarize a collection into one value:

```
# number of elements
count(collection)

# sum of numeric elements (array or set)
sum(collection)

# product of numeric elements
product(collection)

# smallest value
min(collection)

# largest value
max(collection)

# sorted array (input can be array or set)
sort(collection)
```

They pair naturally with comprehensions as you can build the list of numbers or dates you care about, then fold it:

```
package rental

total_rental_value := sum([item_price |
  some item in input.member.rentals.items
  some film in data.catalogue
  film.id == item.film_id
  item_price := film.rental_price
])

overdue_count := count([item |
  some item in input.member.rentals.items
  item.overdue
])

all_due_dates := sort([item.due_date |
  some item in input.member.rentals.items
])

earliest_due := all_due_dates[0]
```

On Alex's input, `overdue_count` is 1, and `earliest_due` is the earlier of the two due dates in the fixture. `sort` on ISO 8601 date strings orders them chronologically because lexicographic order matches calendar order for that format. Aggregates quickly get you to the one thing you care about.

Collections

Beyond summing and counting, policies often need to rearrange collections, like when you need to reverse a list for display, slice the first two overdue items, read a field that might be missing, or compare tier sets.

Arrays

```
# concatenate two arrays
array.concat(a, b)

# sub-array from start (inclusive) to end (exclusive)
array.slice(arr, start, end)

# reverse an array
array.reverse(arr)
```

Objects

You'll work with objects all the time in policies. The input and data panels in the playground contain objects and you're likely to create others along the way. The built-ins for objects can read and reshape them without repeating fragile field chains:

```
# safe field access with fallback
object.get(obj, key, default)

# set of all keys
object.keys(obj)

# array of all values
object.values(obj)

# merge two objects (b wins on conflict)
object.union(a, b)

# new object without specified keys
object.remove(obj, keys)

# new object with only specified keys
object.filter(obj, keys)
```

`object.get` is the safe way to read a field that might be absent, as you can specify a default if the field is undefined.

```
package rental

# Safe: returns 0.00 if input.member.late_fees is missing
late_fees := object.get(input.member, "late_fees", 0.00)

# Safe: returns "classic" if tier is absent
tier := object.get(input.member.membership, "tier", "classic")
```

Alex's fixture includes `late_fees` of 1.50, so `object.get` returns that value. Remove the field in the playground, and the same expression returns `0.00` without leaving `late_fees` undefined.

Sets

When working with more than one set, you can produce unions, intersections, and diffs using operators:

```
# union
a | b

# intersection
a & b

# difference (elements in a but not b)
a - b
```

Unions produce a new set with all unique values from both sets. Intersections produce a new set containing items found in both sets. Diffs produce a new set containing items found in only one of the sets (i.e., anything that would be missing from an intersection).

Here's an example that uses these operators to create EmptyBox Rental policies:

```
package rental

# Tiers that allow extended rentals
extended_tiers := {"plus", "premier"}

# Tiers that get a discount
discount_tiers := {"plus", "premier"}

# Tiers eligible for both (the intersection)
vip_tiers := extended_tiers & discount_tiers

# Ratings the member cannot rent (age restricted ones they don't qualify for)
all_restricted_ratings := {"12", "15", "18"}

member_permitted_ratings := {r | some r in all_restricted_ratings; to_number(r)
prohibited_ratings := all_restricted_ratings - member_permitted_ratings
```

Alex is 16, so `member_permitted_ratings` includes `"12"` and `"15"` but not `"18"`. You can create `prohibited_ratings` from what remains, listing the ratings Alex can't watch. That is the same subtraction the employee does mentally after checking his age against the chart on the wall.

Type checking

There's a suite of type-checking built-ins for ensuring you receive valid data. For example, `age` should be a number, but an upstream system might send the string `"16"` by mistake. You can fail a policy with a clear message rather than quietly mis-compare types.

```
# true if x is a string
is_string(x)

# true if x is a number
is_number(x)

# true if x is a boolean
is_boolean(x)

# true if x is null
is_null(x)

# true if x is an array
is_array(x)

# true if x is an object
is_object(x)

# true if x is a set
is_set(x)

# returns the type as a string: "string", "number", etc.
type_name(x)
```

```
package rental

deny contains msg if {
  val := input.member.age
  not is_number(val)
  msg := $"member.age must be a number, got {type_name(val)}"
}

deny contains msg if {
  val := input.film.rating
  not is_string(val)
  msg := $"film.rating must be a string, got {type_name(val)}"
}
```

On the valid fixture, neither denial fires. Change `input.member.age` to `"16"` in the playground and you should see a message naming `"string"`. The takeaway: type built-ins are early guards that keep bad input from masquerading as a normal denial.

Encoding and decoding

Real integrations rarely hand you a perfect JSON document. Payloads arrive as strings, base64 blobs, or tokens. Encoding built-ins translate between Rego values and those wire formats:

```
# Rego value → JSON string
json.marshal(val)

# JSON string → Rego value
json.unmarshal(str)

# true if str is valid JSON
json.is_valid(str)

# Rego value → YAML string
yaml.unmarshal(str)

# YAML string → Rego value
yaml.marshal(str)

# encode to base64
base64.encode(str)

# decode from base64
base64.decode(str)

# URL-safe base64
base64url.encode(str)

# URL-safe base64 decode
base64url.decode(str)

# percent-encode a query string value
urlquery.encode(str)

# decode a percent-encoded string
urlquery.decode(str)
```

If a legacy lane passes the member record as a JSON string instead of a parsed object, `json.unmarshal` turns it into something the rest of the policy can use:

```
package rental

# If the member record is passed as a JSON string rather than a parsed object
parsed_member := json.unmarshal(`{"membership": {"active": true}}`)

member_is_active if parsed_member.membership.active
```

For member cards backed by signed tokens rather than the store's own database row, JWT built-ins verify or decode in one step:

```
# verify HMAC-SHA256 signature
io.jwt.verify_hs256(token, secret)

# [header, payload, sig] — no verification
io.jwt.decode(token)

# decode and verify in one step
io.jwt.decode_verify(token, constraints)
```

In 1995, EmptyBox might not have issued JWTs at the counter, but the same rental policy often ran later behind an API gateway that did. Knowing these built-ins exist keeps the Alex story intact when *input* arrives in a different envelope. Maybe EmptyBox can make the leap from physical store to streaming giant!

Time

You don't need to work with computers for long to discover that dates and times are a garden littered with tools. Almost every coder has stepped on a date-related rake and been hit in the face with the handle at some point. That makes Rego's time built-ins a must-know feature. At EmptyBox Rentals you need to work with opening hours, due dates, and overdue fees.

Dates and times are stored in nanoseconds internally, and as RFC3339 strings at the boundary:

```
# current time as Unix nanoseconds
time.now_ns()

# RFC3339 string → nanoseconds
time.parse_rfc3339_ns(str)

# parse with Go time layout → nanoseconds
time.parse_ns(layout, str)

# nanoseconds → RFC3339 string
time.format(ns)

# nanoseconds → "Monday", "Tuesday", etc.
time.weekday(ns)

# nanoseconds → [hour, minute, second]
time.clock(ns)

# nanoseconds → [year, month, day]
time.date(ns)

# [years, months, days, hours, minutes, seconds]
time.diff(ns1, ns2)

# add duration to timestamp
time.add_date(ns, years, months, days)
```

Imagine you're processing a click-and-collect order for Alex. You'd want to make sure the store is open to avoid a disappointing journey:

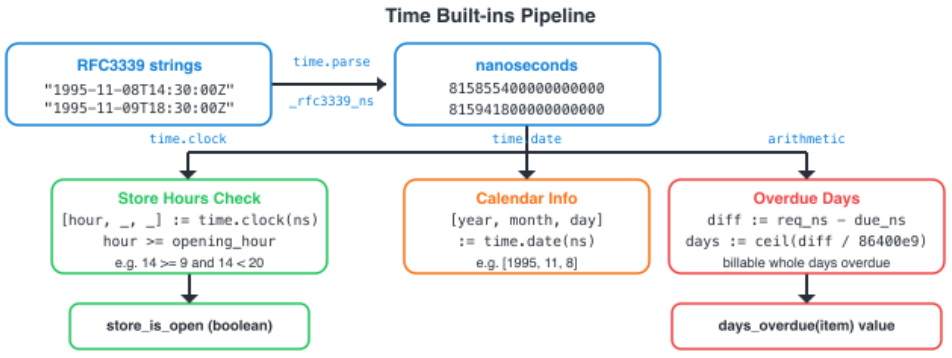
```
package rental

store_is_open if {
  request_ns  := time.parse_rfc3339_ns(input.request.timestamp)
  [hour, _, _] := time.clock(request_ns)
  hour >= data.store.opening_hour
  hour < data.store.closing_hour
}

deny contains "store is closed" if not store_is_open
```

You can adjust `input.request.timestamp` or the store's hours in `data` in the playground. If the request is within the window, `store_is_open` holds and the denial stays empty; otherwise, `"store is closed"` comes back as the `deny` message.

Overdue logic reuses the same parse-and-compare pattern per rental line:

```
package rental

days_overdue(item) := days if {
  request_ns := time.parse_rfc3339_ns(input.request.timestamp)
  due_ns     := time.parse_rfc3339_ns(item.due_date)
  days       := ceil((request_ns - due_ns) / 1000000000 / 86400)
}

seriously_overdue contains item.title if {
  some item in input.member.rentals.items
  item.overdue
  days_overdue(item) > 7
}
```

Alex has an overdue copy of *The Lion King*, but it's not "seriously" overdue yet. Change the request timestamp from 1995 to 1996 in the playground, and the title moves into `seriously_overdue`. For tests, pair this with `with` from Chapter 11 to replace `time.now_ns` to make tests date-independent.



parse_rfc3339_ns converts boundary strings to nanoseconds; time.clock extracts the hour for store-hours checks; arithmetic on nanosecond differences plus ceil gives billable overdue days.

Handling runtime errors

Even for robust and well-tested policies, there's always the chance the calling system will provide bad input or data. Not every payload will decode, and there's always the chance that a consumer will send the wrong data type. When built-ins hit a runtime error, the result isn't an exception but `undefined`. That means failures will fail quietly without raising an error, which is often what you want within a policy. If the member's age is missing or is not a number, you would deny the rental rather than crash out.

When you do want to detect the failure, you can use negation.

```
package rental

deny contains "invalid age: cannot convert to number" if {
  not to_number(input.member.age)
}
```

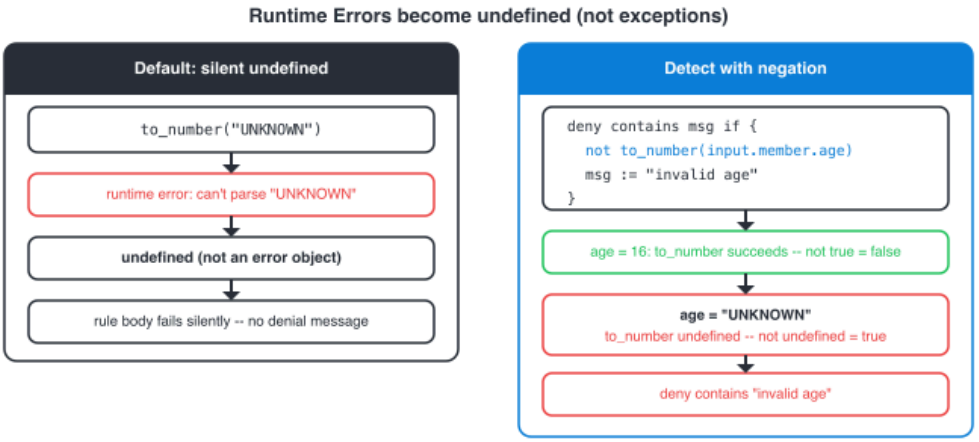
With Alex's age set to either `16` or `"16"` the denial doesn't appear. Update it to `"UNKNOWN"` and evaluate the policy in the pre-populated playground (**oc.to/rego-playground**) to see the result:

```
{
  "deny": [
    "invalid age: cannot convert to number"
  ]
}
```

If you want all failures to be loud, you can switch on strict built-in errors by passing a flag to the CLI (`--strict-builtin-errors`). This can be useful for debugging:

```
opa eval --strict-builtin-errors \
  --data rental.rego \
  --input input.json \
  'data.rental.allow'
```

Use that flag when you want built-in mistakes to surface immediately instead of disappearing into undefined.



By default, runtime errors become undefined and fail silently; use not to detect failures, or --strict-builtin-errors to surface them immediately.

What comes next

Let's be honest, you've developed serious Rego skills already. You could stop now, and you'd still be the go-to for Rego questions on your team. We've covered all the policy syntax you could need to create rules for EmptyBox rentals.

But you're an explorer. Two-thirds of the journey isn't enough for you, and you know the best parts aren't the syntactical details but those that concern design, organization, and maintainability of your policy code. Perform some warm-up stretches to get ready for the best bits, as they are coming up next.

Key concepts

- Built-ins are called with standard function syntax. Boolean predicates can be used directly as conditions; other built-ins return values captured with `:=`.
- The string library covers testing (`startswith`, `contains`), transformation (`lower`, `trim`, `replace`), splitting/joining, formatting (`sprintf`), and regular expressions.
- Aggregate functions (`count`, `sum`, `min`, `max`, `sort`) work on any collection and compose naturally with comprehensions.
- `object.get(obj, key, default)` is the safe way to access a field that may be absent.
- Set operators (`|`, `&`, `-`) express union, intersection, and difference directly in expressions.
- Encoding built-ins (`json`, `yaml`, `base64`, `io.jwt`) handle the structured data formats common in real policies.
- Time built-ins work in nanoseconds. Use `with time.now_ns as ...` in tests to fix the clock.
- By default, runtime errors in built-ins produce undefined, not exceptions. Use `--strict-builtin-errors` to change this behavior.

CHAPTER 13

Modules, packages, and imports

Congratulations, you're an expert in the Rego language features you need to create policies. Before you create an incredible mega-policy, it's worth thinking through how you'll organize your code to make it easy to read and change. So far, you've put all examples in a single rental package, designated by the `package rental` line at the top of the file. As you scale from example size to production size, you'll outgrow single-file policies.

You can stay organized by splitting your code into modules (files), packages (namespaces), and imports (short names for cross-package rules). This chapter will show you how to slice up responsibilities, avoid circular dependencies, load files into a single data tree, and structure policies for EmptyBox rentals. It's time to leave the playground behind and open up your code editor!

Modules revisited

A *module* is a single `.rego` file. Every module must start with exactly one `package` declaration. Once you've checked off these requirements, you can add any number of imports and rule definitions.

Rego has no concept of top-level executable code. Each file is always a named bundle of rules.

```
module
├── package declaration (exactly one, must be first)
├── import statements (zero or more)
└── rule definitions (zero or more)
```

When the policy engine loads policies, it reads all modules and merges their rules into a single virtual document tree rooted at `data`. Multiple modules can contribute to the same package, and their rules are merged as if they were in a single file. The package merging is intentional, as you can split a large `package rental` across `allow.rego` and `deny.rego` without an explicit linker, and the policy engine will evaluate it as a single combined policy.

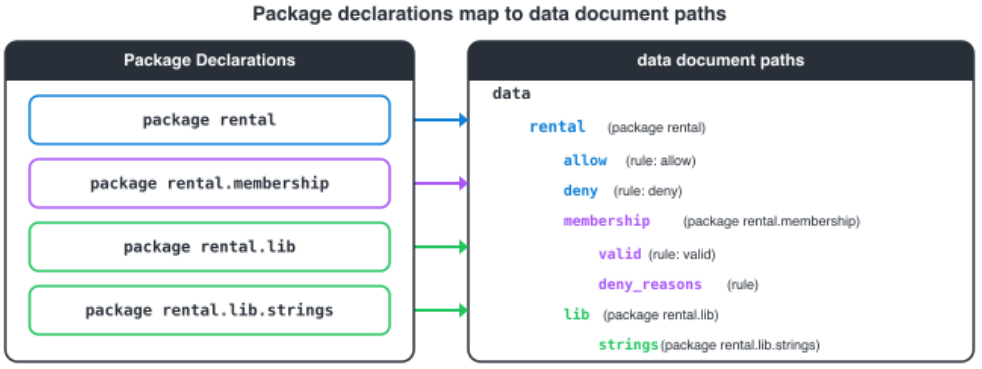
For EmptyBox, think of each `.rego` file as one drawer in the back office. The membership rules are in one drawer, and the age rules are in another. The policy engine opens every drawer and treats matching package names as one policy.

Packages as namespaces

A package declaration places every rule in that module under a namespace. Membership checks might live under `rental.membership` so they don't collide with age or inventory rules:

```
package rental.membership
```

When you add a `valid` rule in this module, you make it available on the path `data.rental.membership.valid`. The dotted path represents the hierarchy of keys in the `data` document, with each segment corresponding to one level down the tree.



Each dot-separated package segment becomes one level in the data tree; a valid rule in package rental.membership is queried as data.rental.membership.valid.

Valid package names

Package names are references, and they follow the same rules as other references in Rego. Each segment must be a valid identifier (letters, digits, and underscores, and they can't start with a digit), or a string in brackets when a segment needs special characters:

```
#simple
package rental

#dotted path
package rental.membership

#deeper hierarchy
package rental.lib.strings

#bracket notation for special chars
package rental["v2"].age
```

These are not valid:

```
#error: starts with a digit
package 1rental

#error: non-string key in package path
package rental[1].age
```

Convention

By convention, package names mirror the policy project's directory structure. A file at `rental/membership/policy.rego` would typically declare `package rental.membership`. You can predict where a rule lives from its fully qualified name, and navigate the repo the same way you navigate `data.rental.membership.valid` in a query.

For the video rental store, a natural layout separates the questions the employee already asks in sequence:

```
rental
├── policy.rego              # package rental — allow, deny, decision_stream
├── policy_test.rego
├── request
│   └── policy.rego          # package rental.request
├── membership
│   └── policy.rego          # package rental.membership
├── overdue
│   └── policy.rego          # package rental.overdue
├── ratings
│   └── policy.rego          # package rental.ratings
├── inventory
│   └── policy.rego          # package rental.inventory
├── store
│   └── policy.rego          # package rental.store
└── stream
    └── policy.rego          # package rental.stream
```

Imports

You can navigate cross-package references without imports using the full path, like `data.<package>.<rule>`. This works, but it's verbose and brittle, as renaming a package will result in many paths changing. You can use the `import` statement to reduce the blast radius of a rename, reduce path length in rules, and make dependencies clear. An import brings a package, rule, or document path into scope with a shorter local name.

Importing a package

Alex's allow rule needs membership, age, and stock to all pass. Importing each sub-package keeps the body readable:

```
package rental

import data.rental.membership
import data.rental.overdue
import data.rental.ratings

default allow := false

allow if {
  membership.valid
  membership.slots_available
  overdue.none
  ratings.permitted
}
```

After `import data.rental.membership`, you write `membership.valid` instead of `data.rental.membership.valid`. The last segment of the import path becomes the local alias (`membership`, `age`, `inventory`). The list of imports gives you an idea of the number of dependencies your policy has.

Importing a specific rule

You can import a single rule instead of the whole package:

```
package rental

import data.rental.membership.valid
import data.rental.age.permitted

default allow := false

allow if {
  valid
  permitted
}
```

Now `valid` and `permitted` appear in the rule body without a prefix. That is convenient when a file calls the same helpers repeatedly, but in large modules, it can also make it harder to understand where a name came from. Many teams import whole packages for entry-point files and reserve rule-level imports for small, focused modules.

Aliasing with "as"

When the default local name would clash with another name, or when you want something more descriptive, use `as`:

```
package rental

import data.rental.age as age_policy

allow if age_policy.permitted
```

Aliasing also helps when a package path is long and referenced often, as a short alias keeps rule bodies easier to scan.

Importing "input" paths

`import` is not limited to `data`. You can shorten paths into `input` the same way:

```
package rental

import input.member
import input.film
import input.request


allow if {
  member.membership.active
  film.available_copies > 0
}
```

After `import input.member`, you write `member.membership.active` instead of `input.member.membership.active`. Some teams prefer this in rule bodies; others keep the `input.` prefix so every decision still visibly depends on request data. Both styles are valid, but pick one per project and stay consistent.

Implicit imports

Every module implicitly imports `data` and `input` at the top level. You never need:

```
import data    # never needed
import input   # never needed
```

Those two roots are always in scope. You still use explicit imports when you want a *subpackage* or a nested `input` field under a short name.

Cross-package references

A rule in one package can reference another package with the full `data.<path>` address. You don't need an import as the reference alone is enough:

```
package rental

allow if data.rental.membership.valid
allow if data.rental.age.permitted
```

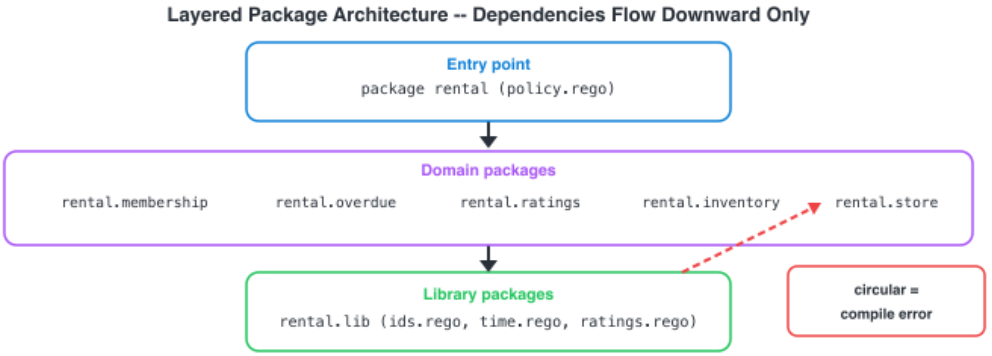
Imports are there for readability. Under the hood, `membership.valid` after an import is the same reference as `data.rental.membership.valid`.

Circular dependencies

The policy engine doesn't allow circular dependencies between packages. If `package rental.age` references `package rental.lib`, then `package rental.lib` must not reference anything in `package rental.age`. The policy engine detects circular dependencies at compile time and reports an error. The practical rule for EmptyBox-shaped projects: layer packages like the store's dependencies. Shared utilities sit at the bottom; domain packages sit in the middle; the entry-point package that the policy engine queries sits at the top.

```
entry-point packages  (rental)
 ↓
domain packages  (rental.membership, rental.overdue, rental.inventory)
 ↓
lib packages  (rental.lib)
```

Organizing your folder structure this way makes it easy to avoid circular dependencies, since you can enforce a rule that allows you to reference packages only in lower-level folders. The top-level file orchestrates the rules in the domain layer, so nothing in the domain layer needs to reference another file in the same folder.



The entry-point package imports domain packages; domain packages import lib packages; no package may import one above it -- circular deps are a compile error. This enforces a single direction of dependency: orchestration at top, shared utilities at bottom.

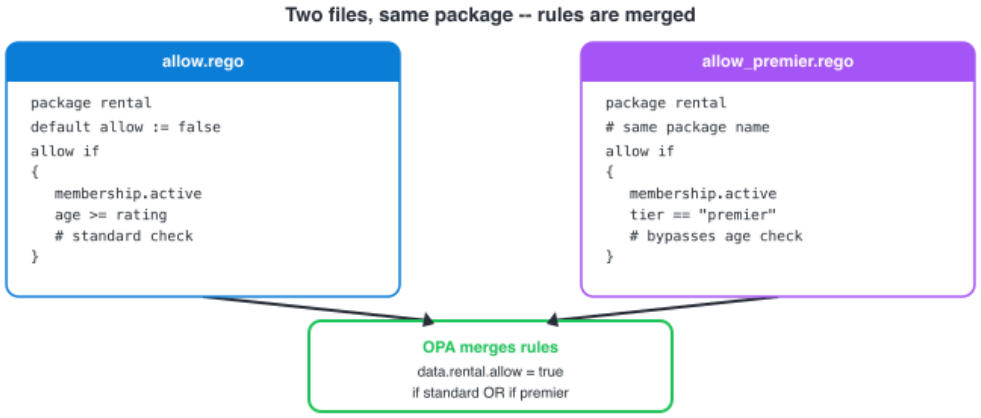
Multiple modules in the same package

Two modules can declare the same package and the policy engine will merge their rules:

```
# allow.rego
package rental

default allow := false

allow if {
  input.member.membership.active
  not input.member.membership.fees_outstanding
  input.film.available_copies > 0
  input.member.age >= to_number(input.film.rating)
}
```

```
# allow_premier.rego
package rental

# Premier members bypass the age restriction
allow if {
  input.member.membership.active
  not input.member.membership.fees_outstanding
  input.film.available_copies > 0
  input.member.membership.tier == "premier"
}
```

The policy engine treats these as one module. `data.rental.allow` is the union of both definitions. It will be true if the standard conditions hold *or* if the member is premier tier (even when age would otherwise block *Pulp Fiction*). That pattern fits natural file boundaries: one file for the default rental path, another for premier exceptions, without inventing separate packages that must import each other.



Both files declare package rental; OPA merges their allow definitions with OR semantics -- if either fires, allow is true. No linker needed - - matching package names are treated as one policy across all loaded files.

There is a constraint, though: `document` and `package` scope annotations (Chapter 14) may be declared only once per package across all files. If two files in the same package each declare a `package`-scope metadata block, the policy engine raises an error.

Loading policies at runtime

How the policy engine loads files determines which packages exist in a single evaluation.

From a directory: `opa eval --data ./policies/` recursively loads every `.rego` file. Packages merge into the document tree automatically. This is how you run the split rental layout locally.

From a bundle: OPA bundles are tarballs of policies and data. Loaded bundles merge with any other policies on the instance. Bundles are the usual distribution format in production (sidecars, admission controllers, CI gates).

From multiple sources: `opa eval --data rental.rego --data membership.rego --data ./lib/` loads all paths at once. Every rule from every source shares one evaluation context.

In all cases, package names are global within one OPA instance. There is no "local" package namespace, so everything loaded contributes to the same `data` tree. That is why naming and layering matter: two teams loading different tarballs into the same OPA must not accidentally both own `package rental`.

A worked example: structuring the rental policy

Here is a realistic layout for EmptyBox with the rental policy as the entry point, the domain packages beneath, and a set of library packages logically beneath.

```
rental/
├── policy.rego             # Entry point
├── policy_test.rego
│
├── stream/                 # UI
│   └── policy.rego
│
├── lib/                    # Utilities
│   ├── ids.rego
│   ├── time.rego
│   └── ratings.rego
│                           # Domain
├── request/policy.rego
├── membership/policy.rego
├── inventory/policy.rego
├── ratings/policy.rego
├── overdue/policy.rego
└── store/policy.rego
```

Let's work from the lowest level of detail to the top. The file `rental/lib/ids.rego` captures the logic for validating member and film IDs.

```
package rental.lib

valid_member_id(id) if regex.match(`^MBR-\d{4}$`, id)

valid_film_id(id) if regex.match(`^VHS-\d{4}$`, id)
```

One level above, `rental/membership/policy.rego` uses `ids.rego`:

```
package rental.membership

import data.rental.lib
import input.member.membership

valid if {
  lib.valid_member_id(member.id)
  membership.active
  not membership.fees_outstanding
}
```

And stepping up another level, `rental/policy.rego`is the entry point that orchestrates policies:

```
package rental

import data.rental.membership
import data.rental.age
import data.rental.inventory

default allow := false

allow if {
  membership.valid
  age.permitted
  inventory.available
}

# Aggregate all denial reasons
deny contains msg if some msg in membership.deny_reasons
deny contains msg if some msg in age.deny_reasons
deny contains msg if some msg in inventory.deny_reasons
```

At each level, packages have a similar level of abstraction. The entry point coordinates policy evaluation and aggregates the results. The domain level has packages that each know a concept, like membership, ratings, or inventory. The library level knows how to perform specific low-level tasks.

You can think about whether two things change for the same reason when you decide where to put them. When you make a change, it's ideal if it affects the fewest possible files. Where things change for different reasons, they deserve to be in different files. Make all your dependencies point in one direction, from the entry point downwards through the domain to the library files.

What comes next

You now have organizational superpowers to apply to your Rego knowledge. You can split a policy across many files and packages and use imports to improve the readability of references. The next chapter covers metadata and annotations, which are structured comments that attach documentation, schemas, and entrypoint declarations to policies. That helps teammates and tools understand your policy structure.

Key concepts

- A *module* is one `.rego` file: one `package`, zero or more `import`s, zero or more rules—no top-level executable code.
- Multiple modules may declare the same `package`; OPA merges their rules, which lets you split large packages by concern (e.g. standard vs. premier `allow`).
- Package names are hierarchical; by convention they mirror directory layout under your policy root.
- `import` brings a `data` path, a rule, or an `input` subtree into scope under a short name; use `as` when the default alias is unclear or conflicting. `data` and `input` roots are always implicit.
- Cross-package references always work via full `data.<path>`; imports are optional readability.
- OPA forbids circular package dependencies—structure in layers: `lib` at the bottom, domain packages in the middle, entry-point packages at the top.
- All policies loaded into one OPA instance share a single global `data` namespace.

CHAPTER 14

Metadata and annotations

Now you know how to split a growing policy across files, name packages clearly, and import only what each module needs. That improves your ability to scale, but it may not go far enough to explain *why* a rule exists or which document to query by default. *Annotations* attach that context in the source. You can use them to add titles, descriptions, owners, links, severity, and entry point markers for bundle builds. The policy engine parses them from `# METADATA` comment blocks at compile time, and policies can read them at evaluation time.

This chapter covers the syntax, the fields you'll use most, how *scope* (`rule`, `document`, `package`, `subpackages`) decides what an annotation applies to, and patterns that keep metadata maintainable as EmptyBox's policy grows.

The "# METADATA" block

At EmptyBox Rentals, the employee's checklist lives next to the till, not in the back office. Annotations work the same way. They are made from structured YAML in a comment block immediately above the rule or package they describe:

```
# METADATA
# title: Allow rental
# description: Grants rental permission when the member meets all requirements.
# authors:
#   - store manager <manager@emptybox.example.com>
allow if {
  input.member.membership.active
  not input.member.membership.fees_outstanding
  input.film.available_copies > 0
  input.member.age >= to_number(input.film.rating)
}
```

Every line in the block must start with a comment marker (`#`) at column 1. The YAML begins on the line after `# METADATA`, and you can't add any blank lines; otherwise, the policy engine will think the YAML has finished. Malformed YAML is a compile-time parse error. That's deliberate: annotations stay machine-readable as you edit them, the same way invalid Rego fails fast instead of silently drifting.

If you want a normal comment between the block and the rule, put a blank line after the YAML first so OPA does not merge the comment into the annotation:

```
# METADATA
# title: Allow rental

# This is a regular comment, safe because there's a blank line above.
package rental

allow if input.member.membership.active
```

Try this example in your code editor. You can see the validation in action by deliberately breaking the YAML (try removing the `:` after `title`).

You can use `opa check` to perform validation against the folder.

```
opa check rental
```

When the YAML isn't valid, you'll get an error:

> rego_parse_error: yaml: line 2: mapping values are not allowed in this context

Annotation fields

There's a fixed set of supported annotation fields. The ones below are what you will reach for first when documenting EmptyBox's rules or wiring tooling around them.

"title"

A short, human-readable name for the annotated rule or package. This is what you might print on an internal audit sheet next to `deny`:

```
# METADATA
# title: Check membership is active
membership_valid if {
  input.member.membership.active
  not input.member.membership.fees_outstanding
}
```

Titles show up in `opa inspect` output and in documentation generated from policies. For Alex's case, a title like "Check membership is active" tells the next engineer which counter question the rule answers without reading the whole body.

"description"

A longer explanation of purpose, inputs, and behavior. You can use multi-line YAML for this, using the `|` block style:

```
# METADATA
# title: Deny underage member
# description: |
#   Rejects rental requests where the member's age is below the minimum
#   required for the film's age rating. applies to all rating categories
#   (U, PG, 12, 15, 18). premier members bypass this check.
deny contains msg if {
  input.member.membership.tier != "premier"
  input.member.age < to_number(input.film.rating)
  msg := $"member is {input.member.age} but film is rated {input.film.rating}"
}
```

With Alex (16) and *Pulp Fiction* (rated 18), evaluating `deny` still produces the runtime message from the rule body. The description documents *why* that rule exists for operators and for `opa inspect`; it does not replace the message Alex sees unless you copy it into the output yourself (you'll see how to do this a bit later).

"authors"

Who maintains this rule or package. You can use plain strings or structured name and email properties:

```
# METADATA
# authors:
#   - store manager <manager@emptybox.example.com>
#   - name: policy team
#     email: policy@emptybox.example.com
```

The author is the person to ask when you have a question or want to make a change.

"related_resources"

Links to runbooks, BBFC rating guidance, or internal policy pages:

```
# METADATA
# related_resources:
#   - ref: https://emptybox.example.com/membership/late-fees
#     description: late fee schedule and waiver policy
#   - ref: https://www.bbfc.co.uk/what-we-do/classification-guidelines
#     description: BBFC classification guidelines
```

The age-denial rule above is easier to defend in a review when `related_resources` points at the same BBFC definitions the employee uses at the counter.

"custom"

You can add other information you need to a custom section. You might include severity, a work item ID, or a remediation hint for the user to fix the policy violation.:

```
# METADATA
# custom:
#   severity: HIGH
#   category: age-restriction
#   ticket: REN-1234
#   auto_remediation: false
deny contains msg if {
  input.member.membership.tier != "premier"
  input.member.age < to_number(input.film.rating)
  msg := $"member is {input.member.age} but film is rated {input.film.rating}"
}
```

Custom keys are visible at runtime through `rego.metadata.rule()` (see below). That's how you keep severity and category in one place with the rule, rather than in a separate spreadsheet keyed by rule name.

"entrypoint"

Marks a rule or package as an *entrypoint*, which is a rule the policy engine must retain when building a bundle, even if nothing else references it.

```
# METADATA
# entrypoint: true
allow if {
  input.member.membership.active
  not input.member.membership.fees_outstanding
  input.film.available_copies > 0
  input.member.age >= to_number(input.film.rating)
}
```

`opa build` uses entrypoint annotations to decide which rules to include. Without these annotations, the process may prune unreferenced rules. You can pass entrypoints on the command line, too, but annotations keep the intent right next to the rule.

Annotation scope

An annotation block always has a *scope*: which rules it applies to. You can set `scope` explicitly, or OPA infers it from what the block sits above.

"rule" scope

When the block immediately precedes a single rule definition, it applies to that rule only. This is the default when no `scope` is given, but the example below makes it explicit:

```
# METADATA
# title: Check member age against film rating
# scope: rule
age_permitted if input.member.age >= to_number(input.film.rating)
```

"document" scope

A `document`-scoped annotation applies to every definition of the same rule name in the package, including across files. That matches how OPA merges multiple `allow` heads into one logical decision:

```
# METADATA
# scope: document
# title: Allow rental
# description: Fires if any of the allow conditions are met.
allow if {
  input.member.membership.active
  not input.member.membership.fees_outstanding
  input.film.available_copies > 0
  input.member.age >= to_number(input.film.rating)
}

allow if {
  input.member.membership.active
  not input.member.membership.fees_outstanding
  input.film.available_copies > 0
  input.member.membership.tier == "premier"
}
```

The metadata describes both `allow` definitions together, the standard and premier versions, not only the first block in the file.

You may declare `document` scope only once per rule name per package. Declaring it in two files for a rule with the same name is an error.

"package" scope

A `package`-scoped annotation applies to all rules in the package, across all files. Place it above the `package` declaration:

```
# METADATA
# scope: package
# title: Video rental authorization policy
# description: Defines allow and deny rules for
#   the EmptyBox central rental system.
# authors:
#   - policy team <policy@emptybox.example.com>
package rental
```

The sample rental policy uses this pattern at the top of `package rental` for schemas and ownership. Like `document` scope, `package` scope may appear only once per package across the whole project.

"subpackages" scope

A `subpackages`-scoped annotation applies to the annotated package and every package whose path extends it:

```
# METADATA
# scope: subpackages
# organizations:
#   - EmptyBox central
package rental
```

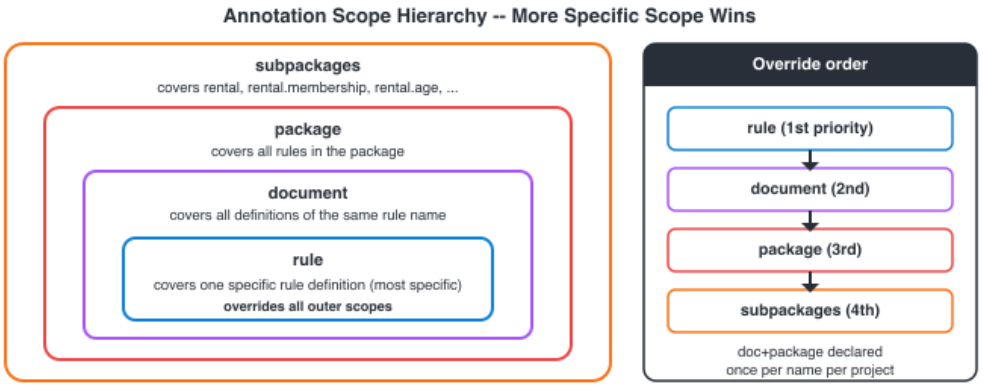
This would cover `rental`, `rental.membership`, `rental.age`, `rental.inventory`, and any other path under `rental`. This is useful for org-wide defaults you don't want to repeat in every file.

Scope override order

When several annotations apply to the same rule, for example, both package-level authors and rule-level title, the more specific scope wins:

```
rule > document > package > subpackages
```

A `rule`-scoped title overrides a `document`-scoped one, which overrides `package`, which overrides `subpackages`.



A rule-scoped title overrides document, which overrides package, which overrides subpackages -- the innermost scope always wins.

The "entrypoint" + scope interaction

If you set `entrypoint: true` on a rule without an explicit `scope`, the policy engine sets scope to `document` automatically. Combining `entrypoint: true` with `scope: rule` is an error: entrypoints always target the whole rule document (all definitions of that name), not a single head.

Accessing annotations at runtime

The EmptyBox Rentals employee doesn't have access to the policy code, so it can be useful to extract metadata for use in messages you create during evaluation. There are built-in functions to help you do that.

"rego.metadata.rule()"

You can use `rego.metadata.rule()` to fetch the metadata for the current rule. You can run the example below in the pre-populated playground (**oc.to/rego-playground**).

```
package rentals

# METADATA
# title: Deny underage member
# description: Member does not meet the film's age requirement.
# custom:
#   severity: HIGH
#   category: age-restriction
deny contains violation if {
  input.member.membership.tier != "premier"
  input.member.age < to_number(input.film.rating)
  meta      := rego.metadata.rule()
  violation := {
    "message":  meta.description,
    "severity": meta.custom.severity,
    "category": meta.custom.category,
  }
}
```

When you evaluate this policy, you'll get back a response with the metadata you retrieved:

```
{
  "deny": [
    {
      "category": "age-restriction",
      "message": "Member does not meet the film's age requirement.",
      "severity": "HIGH"
    }
  ]
}
```

"rego.metadata.chain()"

When you need more information than the current rule provides, you can request the whole chain of annotations using `rego.metadata.chain()`. It returns an array of annotation objects for the rule and its ancestors in the package hierarchy, with the most specific information first:

```
# METADATA
# scope: package
# authors:
#   - name: John Empty-Box
package rentals

# METADATA
# scope: document
# authors:
#   - name: Sarah Empty-Box
deny contains violation if {
  input.member.membership.tier != "premier"
  input.member.age < to_number(input.film.rating)
  chain := rego.metadata.chain()
  authors := [author |
    some entry in chain
      some author in entry.annotations.authors
  ]
  violation := {
    "authors": authors
  }
}
```

When you evaluate this policy, you'll get a list of authors from the metadata at multiple levels:

```
{
  "deny": [{
    "authors": [{
      "name": "Sarah Empty-Box"
    },{
      "name": "John Empty-Box"
    }]
  }]
}
```

You can walk the chain to combine information, or to find information at a higher level that doesn't exist in the current rule.

Viewing annotations with "opa inspect"

You can validate the documented rules and validate the YAML as part of your pre-shipping checks:

```
opa inspect -a ./rental/
```

This produces a summary of namespaces and annotations, like the example below:

```
┌──────────────────────┬──────────────────────────────────────┐
│      NAMESPACE       │                FILE                  │
├──────────────────────┼──────────────────────────────────────┤
│ data.rental          │ samples/rental/policy.rego           │
│ data.rental.inventory│ samples/rental/inventory/policy.rego │
│ ...                  │ ...                                  │
└──────────────────────┴──────────────────────────────────────┘

ANNOTATIONS:
rental
======

Package:  rental
Location: samples/rental/policy.rego:5
Scope: package

Schemas:
 input: schema.input

rental.allow
============

Canonical rental decision for EmptyBox; true only when every precondition passes

Package:  rental
Rule:     allow
Location: samples/rental/policy.rego:20
Scope: document
Entrypoint: true

...
```

A practical annotation pattern

Here is how package scope, per-rule titles, `custom` fields, and `rego.metadata.rule()` fit together so each denial carries its own documentation—no external lookup table keyed by rule name:

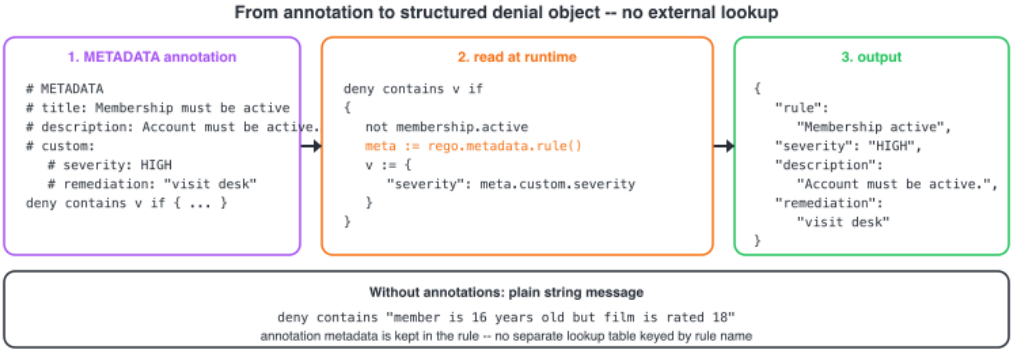
At the package level:

```
# METADATA
# scope: package
# title: Rental authorization policy
# authors:
#   - policy team <policy@emptybox.example.com>
package rental
```

And at the rule level:

```
# METADATA
# title: Membership must be active
# description: The member's account must be in active status to rent films.
# custom:
#   severity: HIGH
#   remediation: "visit the customer service desk to reactivate your membership"
deny contains v if {
  not input.member.membership.active
  meta := rego.metadata.rule()
  v := {
    "rule":        meta.title,
    "description": meta.description,
    "severity":    meta.custom.severity,
    "remediation": meta.custom.remediation,
  }
}

# Continued...
```

```
# ...continued

# METADATA
# title: No outstanding late fees
# description: Members must settle outstanding fees above the threshold.
# custom:
#   severity: MEDIUM
#   remediation: "pay outstanding fees at the front desk"
deny contains v if {
  input.member.late_fees >= data.store.max_late_fees_before_block
  meta := rego.metadata.rule()
  v := {
    "rule":        meta.title,
    "description": meta.description,
    "severity":    meta.custom.severity,
    "remediation": meta.custom.remediation,
  }
}
```



custom fields in the annotation travel with the rule; rego.metadata.rule() reads them at evaluation time to build a rich, self-describing violation object.

In the playground, set Alex's `membership.active` to false and evaluate the policy to see the following result:

```
{
  "deny": [{
    "description": "The account must be in active status to rent films.",
    "remediation": "visit customer services to reactivate your membership",
    "rule": "Membership must be active",
    "severity": "HIGH"
  }]
}
```

What comes next

Annotations document and label the decisions your policy makes. In the next chapter, you'll learn about *schemas*, which describe what valid`input` and `data` looks like. You can use schemas to catch structural mistakes when you compile, long before Alex's request ever reaches evaluation.

Key concepts

- Annotations live in `# METADATA` comment blocks immediately above the rule or package they describe. The content is YAML and must be valid or OPA raises a parse error.
- Key fields: `title`, `description`, `authors`, `related_resources`, `custom`, and `entrypoint`.
- Scope controls which rules an annotation applies to: `rule` (default), `document` (all definitions of the same name), `package` (all rules in the package), `subpackages` (recursive). More specific scopes override less specific ones.
- `rego.metadata.rule()` returns the current rule's annotations at runtime. `rego.metadata.chain()` returns the full ancestor chain.
- Embedding severity, category, and remediation in `custom` annotations and reading them at runtime produces self-describing violation objects.
- `opa inspect -a` lists all annotations in a policy or directory.

CHAPTER 15

Schemas and type checking

Annotations provide information about the intent behind rules, who owns them, and where the entry points are. You can go even further with schemas, which describe object shapes and pull errors to compile-time through type-checking. This chapter walks through schema files and inline annotations for EmptyBox's rental documents, how scope and overrides work, and where the type checker still has gaps.

The problem schemas solve

Suppose a tired engineer adds a late-fee denial rule and mistypes `membership`:

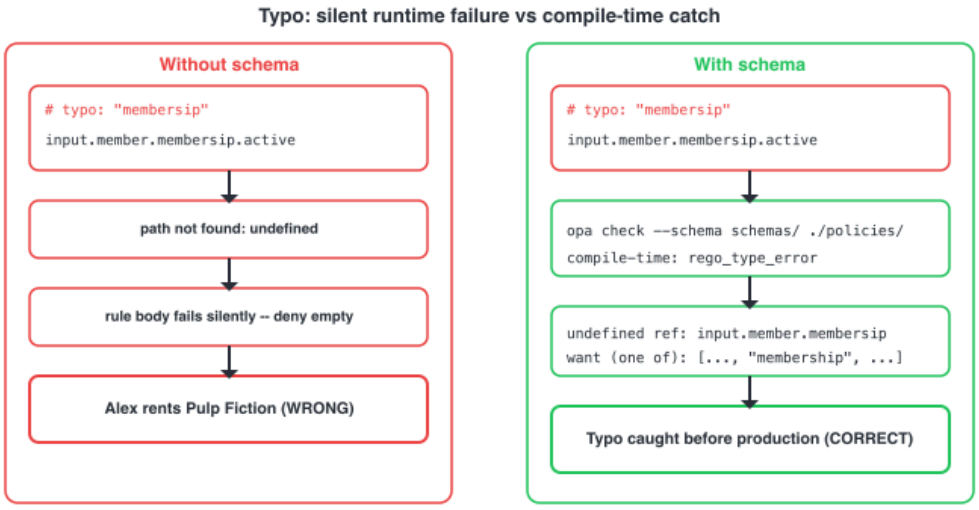
```
package rental

deny contains msg if {
  # typo: "membersip" should be "membership"
  input.member.membersip.active
  input.member.age < to_number(input.film.rating)
  msg := "member is too young for this film"
}
```

Run this in the pre-populated playground (**oc.to/rego-playground**). The expected result is that Alex is denied the rental with the message "member is too young for this film", but instead the request gets the green light. Huh? What's happened here is there's a typo. Instead of "membership", the policy has "membersip", so the condition doesn't hold (because there is no "membersip" property on the member). Mistakes like this means Alex will go home with *Pulp Fiction*, and you'll have some angry parents to deal with tomorrow morning when EmptyBox Rentals opens.

A schema would catch this issue and report the typo:

> 1 error occurred: rental.rego:5: rego_type_error: undefined ref: input.member.membersip
>
> have: "membersip"
>
> want (one of): ["age" "id" "late_fees" "membership" "name" "rentals"]

The schema turns a silent runtime failure into a loud compile-time error. That's a neat feature in most languages, but for policies it turns out to be superbly important.



Without a schema the typo produces silent undefined and permits the rental; with a schema OPA catches it at compile time before evaluation.

Attaching an external schema file

The `opa eval` and `opa check` commands accept a `--schema` flag pointing to a JSON Schema file or directory.

Single file associates one schema with `input` globally, for all rules in all packages:

```
opa eval \
 --data rental.rego \
  --input input.json \
 --schema schemas/input.json \
  'data.rental.allow'
```

With this flag, every reference to `input` in any rule is type-checked against `input.json`. Access a field not defined in the schema, and OPA reports a type error at check time, preventing policy mistakes and runtime errors.

Directory loads multiple schema files and relies on policy annotations to associate each file with the correct path. Name the schema for `input input.json` inside the directory; other files are referenced by name via annotations (covered below):

```
opa eval \
 --data rental.rego \
  --input input.json \
 --schema schemas/ \
  'data.rental.allow'
```

You can use `opa check` as part of your Continuous Integration (CI) process to validate policies without evaluating them.

```
opa check --schema schemas/ ./policies/
```

That command checks all `.rego` files in `./policies/` against the schemas in `schemas/` and exits with a non-zero status if anything fails. A typo like `membersip` fails the build, and your compliance team retains their joyful demeanor.

A schema for the rental input

Chapter 1 introduced Alex's input as a single JSON document. Here is a JSON schema that defines the input's shape:

```
{
  "type": "object",
  "properties": {
    "member": {
      "type": "object",
      "properties": {
        "id":       { "type": "string" },
        "name":     { "type": "string" },
        "age":      { "type": "number" },
        "membership": {
          "type": "object",
          "properties": {
            "tier":             { "type": "string" },
            "active":           { "type": "boolean" },
            "joined":           { "type": "string", "format": "date" },
            "fees_outstanding": { "type": "boolean" }
          }
        },
        "rentals": {
          "type": "object",
          "properties": {
            "current_count": { "type": "number" },
            "max_allowed":   { "type": "number" },
            "items": {
              "type": "array",
              "items": {
                "type": "object",
                "properties": {
                  "film_id":  { "type": "string" },
                  "title":    { "type": "string" },
                  "due_date": { "type": "string", "format": "date" },
                  "overdue":  { "type": "boolean" }
                }
              }
            }
          }
        },

# Continued...
```

```
# ...continued

        "late_fees": { "type": "number" }
      }
    },
    "film": {
      "type": "object",
      "properties": {
        "id":                 { "type": "string" },
        "title":              { "type": "string" },
        "director":           { "type": "string" },
        "year":               { "type": "number" },
        "rating":             { "type": "string" },
        "genres":             { "type": "array", "items": { "type": "string" } },
        "available_copies":   { "type": "number" },
        "rental_price":       { "type": "number" },
        "rental_period_days": { "type": "number" }
      }
    },
    "request": {
      "type": "object",
      "properties": {
        "action":    { "type": "string" },
        "store":     { "type": "string" },
        "timestamp": { "type": "string", "format": "date-time" }
      }
    }
  }
}
```

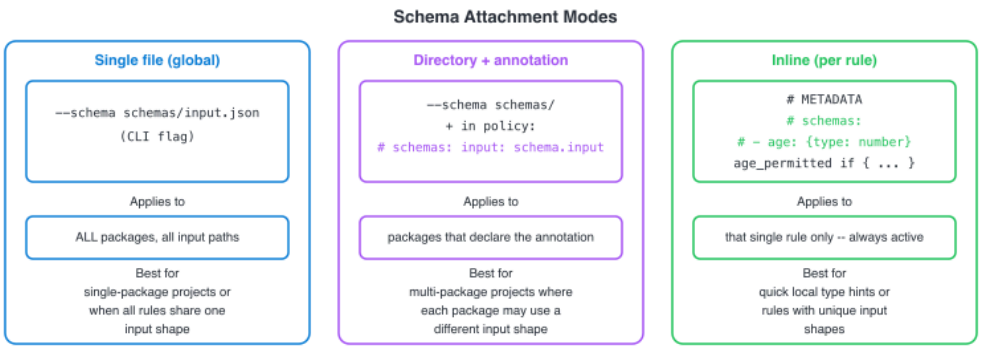
With this schema attached, OPA knows that `input.member.age` is a number, `input.film.rating` is a string, and `input.member.rentals.items` is an array. If you reference a path that's not in the schema, like`input.member.membersip`, and `opa check` fails with the error you saw above. The checker is documenting the same shape Alex's JSON already uses; your job is to keep policy paths aligned with it.

Inline schema annotations

External files suit the full rental input. For a single rule that only needs a few types, you can embed schema fragments in the policy using the `schemas` annotation field from Chapter 14. Inline schemas are always evaluated, even if you don't pass `--schema` on the command line:

```
# METADATA
# schemas:
#   - input.member.age: {type: number}
#   - input.film.rating: {type: string}
age_permitted if {
  input.member.age >= to_number(input.film.rating)
}
```

With the schemas annotation, the policy engine knows `input.member.age` is a number and `input.film.rating` is a string. These are the fields the comparison Alex's age check depends on. Compare `input.member.age` to a boolean, and you get a type error instead of a confusing runtime result. Inline schemas accept any valid JSON Schema structure as YAML. Use them for small, local constraints; keep the full Alex input document in an external file.



Use package-scope annotations with a schema directory for most projects; add inline schemas only for one-off local type hints.

Schema annotations with external files

When you load a schema directory, annotations in the policy map document paths to files inside that directory. The `schema` prefix is the root of the directory; trailing components are the relative path without the `.json` extension:

```
# METADATA
# schemas:
#   - input: schema.input
#   - data.store: schema["store-config"]
allow if {
  input.member.membership.active
  input.member.late_fees < data.store.max_late_fees_before_block
}
```

- `input` is typed against `schemas/input.json` (Alex's counter document).
- `data.store` is typed against `schemas/store-config.json` (store limits, like max late fees).

`schema.store-config` would be invalid Rego as hyphens are not allowed in dot-notation identifiers. Filenames with hyphens need the bracket form: `schema["store-config"]`, or you could avoid hyphens in your schema file paths. Paths under `data` that have no schema annotation remain type `Any`. The policy engine doesn't restrict them. Schema annotations are additive: declare what you know; the checker validates only those paths.

Controlling schema scope

Schema annotations follow the same scope rules as other metadata from Chapter 14. A `document`-scoped schema applies to all rules with that name; a `package`-scoped one applies to every rule in the package.

In EmptyBox's rental policy, a single input shape for the whole package is cleaner than repeating schemas for every rule:

```
# METADATA
# scope: package
# schemas:
#   - input: schema.input
package rental
```

If different rules genuinely expect different input shapes, `rule`-scoped annotations attach a different schema per rule:

```
package rental

# METADATA
# scope: rule
# schemas:
#   - input: schema["input-v1"]
allow_v1 if {
  input.version == "v1"
  input.member.membership.active
}

# METADATA
# scope: rule
# schemas:
#   - input: schema["input-v2"]
allow_v2 if {
  input.version == "v2"
  input.customer.subscription.enabled
}
```

`allow_v1` is checked against the v1 member shape; `allow_v2` against a different customer model. Each rule gets the schema that matches what it actually reads.

Schema overriding

Sometimes a shared schema deliberately under-specifies a field. For example, `input.member.rentals.items` might be a generic array of objects. *Schema overriding* refines the type at a specific path for one rule without replacing the whole document schema.

Alex still has an overdue *Lion King* on his account. A rule that flags severely late items might need `days_late` on each rental item, even though the base input schema does not list that field:

```
# METADATA
# scope: rule
# schemas:
#   - input: schema.input
#   - input.member.rentals.items: schema["rental-item-detail"]
overdue_items contains item if {
  some item in input.member.rentals.items
  item.overdue
  item.days_late > 7 # days_late is in rental-item-detail but not in input.json
}
```

The second annotation overrides `input.member.rentals.items` for this rule only. OPA merges `rental-item-detail` at that path, so `item.days_late` is a known number here. Overrides apply in order: later annotations refine earlier ones; fields from the outer schema that the override does not mention are preserved.

Remote references in JSON schema

JSON Schema supports `$ref` to pull in definitions from another file by URL, which is useful if EmptyBox centralizes film metadata:

```
{
  "type": "object",
  "properties": {
    "film": {
      "$ref": "https://schemas.emptybox.example.com/v1/film.json"
    }
  }
}
```

The policy engine fetches remote references by default during type checking. To control which hosts it may contact, pass a capabilities file with an `allow_net` key:

```
{
  "builtins": [ "..." ],
  "allow_net": ["schemas.emptybox.example.com"]
}
```

Set `allow_net` to an empty array to deactivate all network access during schema checking. You'd use this in an air-gapped CI environment where outbound connections are blocked.

Limitations

The schema type checker is powerful but not exhaustive. Several JSON Schema features are not yet supported:

- `additionalProperties` and `additionalItems`: schemas are treated as if they allow extra properties even when set to `false`
- `pattern` properties for objects
- `contains` for arrays
- `oneOf` and `not`
- `enum`
- `if` / `then` / `else`

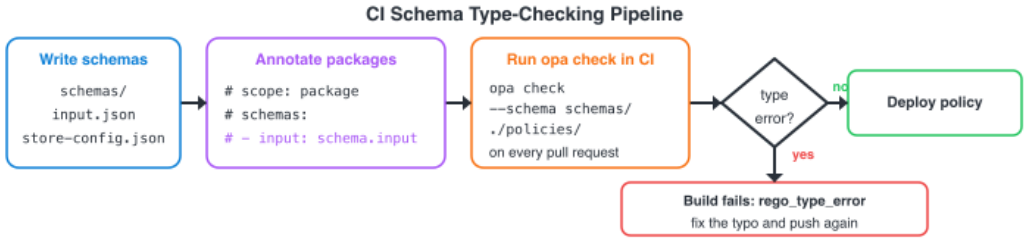
So the checker will not flag every access to an undeclared property when `additionalProperties` is false, and it cannot enforce enum constraints. These are enforcement gaps, not reasons to skip schemas. In practice, schemas excel at *structural* mistakes, like wrong field names or treating a string as an object. They are less useful for *value* mistakes (a field exists but holds the wrong data). Use them as a first line of defense for typos like `membersip`; use tests in Chapter 16 to prove Alex is denied for the right business reasons.

A practical workflow

For the rental policy project, a workflow that matches how the samples in this book are laid out:

1. Write a JSON schema for `input` (as above) and for any `data` documents your rules read. Place them in `schemas/` next to the policies.
2. Add a `package`-scoped schema annotation to each policy package, pointing at the right files.
3. Run `opa check --schema schemas/ ./policies/` in CI on every pull request.
4. Reserve inline schemas for small, one-off type hints on individual rules.

That catches the most common policy bug at EmptyBox (a mistyped field name) before it can fail in production. With Alex's document alone listing membership tiers, rental items, film ratings, and request metadata, having the compiler flag misspellings is worth the upfront schema maintenance.



Write schemas, annotate packages, run opa check on every PR -
- a type error fails the build before the policy reaches production.

Designing good schemas

You should think about schemas early because you can make policy files simpler and more broadly applicable by designing your schemas at an appropriate level of abstraction. For example, if you want to write policies for deployments, having a schema that can represent deployments in a single shape regardless of tech stacks and infrastructure choices means a single policy could evaluate all deployments. Without thought given to schema design, you'd end up with different input shapes and policies that only work for a single tech stack.

What comes next

You have now covered the complete Rego language. Part V shifts to making policies production-ready. The next chapter covers testing, including running unit tests with `opa test`, using `with` to control inputs, measuring coverage, and structuring a policy project to keep Alex's scenarios honest as the rules evolve.

Key concepts

- Without schemas, typos in field references produce silent, undefined values rather than errors. Schemas turn these into compile-time type errors.
- The `--schema` flag on `opa eval` and `opa check` loads external JSON Schema files. A single file is applied to `input` globally; a directory is mapped via annotations.
- Inline schema annotations (`schemas` field in `# METADATA`) embed schema definitions directly in the policy. They are always evaluated without requiring `--schema`.
- Schema annotations follow the same scope rules as other annotations. `package` scope applies to all rules; `rule` scope applies to one.
- Schema overriding refines the type of a specific path within an existing schema. Order matters, and the general annotation must precede the specific one.
- The type checker does not support `additionalProperties`, `enum`, `oneOf`, `not`, or `if/then/else`. It is best suited for catching structural navigation errors, not value constraints.
- Run `opa check --schema schemas/ ./policies/` in CI as a standard quality gate.

Part V

Scaling up from working to production-ready policies

CHAPTER 16

Testing policies

You now know all the tools you need to write policies, organize them, and annotate them with metadata and schemas. The next step is to create policies with legendary robustness by adding tests you write in Rego itself. Policies are likely to be among the most important parts of your system, so it's worth having proof that they work. Automated tests also provide ongoing evidence that the policies function as expected. Though this chapter comes late in your learning journey, it's best to write tests early for your production policies. Let's get to work.

The "test_" convention

In Rego, tests look just like rules. To make it clear they're tests, the convention is to begin test rule names with `test_`. If the rule body evaluates to a truthy value, the test passes. If the body is false or undefined, the test fails. That's as simple as it gets. There's no separate test framework or domain-specific language to learn. You test Rego rules with other Rego rules named `test_something` and you prove your policy works.

Your first test case will handle age and film rating checks. If Alex is 18, he should be allowed to rent an 18-rated film, but if he's 17, he shouldn't. We use 17 here, rather than Alex's real age of 16, because it's best to place the tests on the edge of a rule. It catches a common kind of error, which is where the logic is off by one. This commonly happens when someone uses the wrong comparison operator, like`<` when they meant to use `<=`.

```
package rental_test
import data.rental

test_adult_member_can_rent_18_film if {
  rental.allow with input as {
    "member": {
      "id": "MBR-0001",
      "age": 18,
      "membership": {"active": true, "fees_outstanding": false},
      "rentals": {"current_count": 0, "max_allowed": 1},
    },
    "film": {"id": "VHS-0001", "rating": "18", "available_copies": 1},
    "request": {"action": "rent", "timestamp": "1995-11-09T18:30:00Z"},
  }
}

test_underage_member_is_denied if {
  not rental.allow with input as {
    "member": {
      "id": "MBR-0001",
      "age": 17,
      "membership": {"active": true, "fees_outstanding": false},
      "rentals": {"current_count": 0, "max_allowed": 1},
    },
    "film": {"id": "VHS-0001", "rating": "18", "available_copies": 1},
    "request": {"action": "rent", "timestamp": "1995-11-09T18:30:00Z"},
  }
}
```

You can run the tests on the command line to get the result:

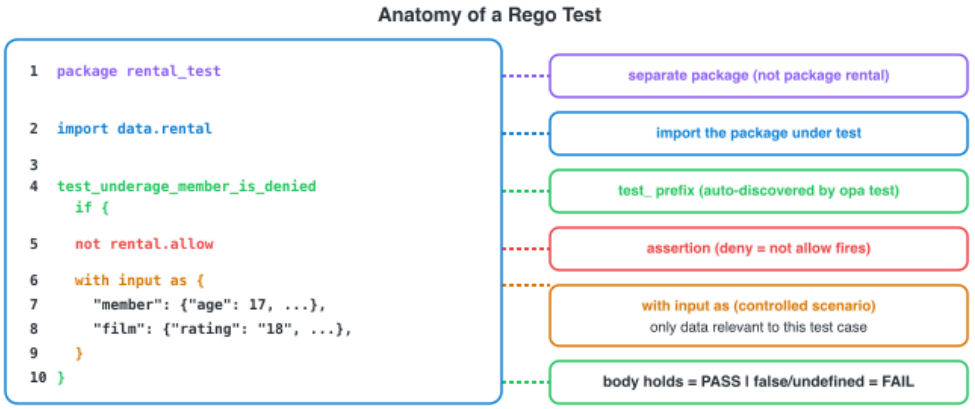
```
opa test -v \
  samples/rental/ratings \
  samples/rental/lib \
  samples/rental/chapter14_test_001.rego
```

Both tests will pass and a summary of the tests and the overall result will be provided:

```
samples/rental/chapter14_test_001.rego:11:
data.rental_test.test_adult_member_can_rent_18_film: PASS (508.375µs)
samples/rental/chapter14_test_001.rego:18:
data.rental_test.test_underage_member_is_denied: PASS (570.916µs)
--------------------------------------------------------------------------------
PASS: 2/2
```

There are a few conventions worth noting:

- The test package is `rental_test`, separate from the `rental` package under test.
- The test imports the package under test, so `rental.allow` is available without writing `data.rental.allow`.
- Each test uses `with` to supply a controlled `input`, which is the standard way to replay a specific counter scenario.



Tests are ordinary Rego rules in a separate package; test_ prefix marks them for opa test; with input as supplies the controlled scenario.

Reusing test data

When you have several tests that use similar test data, you can reduce noise in your tests and make the difference more apparent by creating a base input and chaining the `with` commands to pass the part that differs. In this example, the only test data shown in the test rule is the age, the rest of the data's shape comes from `base_input`. This is useful if your test object is complex and you only want to change one thing.

```
package rental_test

import data.rental.ratings

base_input := {
  "member": {"membership": {}},
  "film": {"rating": "18"},
}

test_adult_member_can_rent_18_film if {
  ratings.permitted with input as base_input with input.member.age as 18
}

test_underage_member_is_denied if {
  not ratings.permitted with input as base_input with input.member.age as 17
}
```

Failing tests

A test fails when its rule body does not hold. If `rental.allow` is `true` for Alex at age 17, then `not rental.allow` is `false`, the test rule evaluates to `false`, and the policy engine reports a failure.

Sometimes you need more than allow/deny. You might need to test the exact message the employee (or terminal) would see:

```
test_underage_member_denial_message if {
  msgs := ratings.deny with input as base_input with input.member.age as 17
  count(msgs) == 1
  "member is 17 years old but film is rated 18" in msgs
}
```

When this test passes, `rental.deny` contains exactly one message and it matches the string the policy promises. If the wording drifts or a second denial sneaks in, the test fails, even when `allow` still behaves correctly. You can use the `with` keyword to provide input, or data, or both.

"test_" rules are unordered

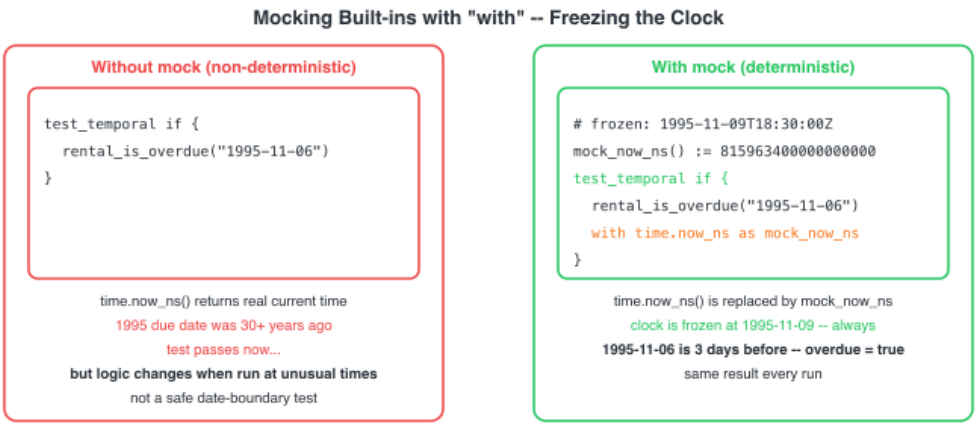
Like all Rego rules, test rules have no fixed execution order. `opa test` runs them all and reports each result independently. There is no setup or teardown: each test is self-contained. That avoids shared state between cases and keeps failures easy to read. For each broken scenario, there will be one failed test name.

Mocking built-in functions

Opening-hours rules depend on when the request arrives. You do not want tests to pass when you run them at 2 pm but fail if you run them at 10 pm. When dates appear in the input, you can supply hard-coded values to make sure the result is deterministic. If the policy depends on `time.now_ns()`, you'll need to replace the built-in function with a constant for the duration of the test, so you can control the time it provides.

```
package rental_test

# 1995-11-09T18:30:00Z in nanoseconds
mock_now_ns() := 815963400000000000

rental_is_overdue(due_date) if {
  now_ns := time.now_ns()
  due_ns := time.parse_rfc3339_ns(due_date)
  now_ns > due_ns
}

test_temporal if {
  rental_is_overdue("1995-11-06T00:00:00Z")
  with time.now_ns as mock_now_ns
}
```

You can replace any built-in with a mock when you need to freeze time or behavior to make a test stable.



with time.now_ns as mock_now_ns replaces the real built-in for that expression only -- the frozen clock makes date-boundary tests reproducible.

Running tests with "opa test"

The command line `opa test` needs to know where to find your policies and tests:

```
opa test ./policies/ ./tests/
```

You can also specify the location of schemas, fixtures, and data (and at this point, I usually place this in a make file so I can call `make test` on the command line):

```
opa test -v --schema ./samples/schemas/ \
  ./samples/rental \
  ./samples/emptybox \
  ./samples/fixtures/data.json
```

OPA loads every `.rego` file in those paths and runs every rule whose name starts with `test_`. A clean run looks like:

```
PASS: 12/12
```

A failure names the package and rule, then summarizes the run:

```
FAIL: data.rental_test.test_underage_member_is_denied (1.2ms)
PASS: data.rental_test.test_adult_member_can_rent_18_film (0.8ms)
---
FAIL: 1/2
```

You can add `-v` to list every test, including passes. This is useful when you are checking that a subset still runs:

```
opa test -v ./policies/
```

For your CI builds, a non-zero exit on failure is all you need to gate a merge:

```
opa test ./policies/ || exit 1
```

Filtering tests

When you need to debug a single scenario, you can pass a flag to run only matching tests:

```
opa test -r "test_underage" ./policies/
```

Only matching tests will run, which may be a single test or several tests if your filter matches. Your test naming conventions can be very useful when you need to run subsets of tests. For example, if you named tests after domain areas, you'd be able to run them in sets easily by providing a partial filter of `test_inventory_` or `test_membership_`. If you don't see a test count, it means no tests matched your filter.

Benchmarking

`opa test --bench` runs tests as benchmarks and reports the time and memory use. You can use this to catch policies that slow down on large inputs. Most rental-sized suites do not need it for day-to-day work.

Coverage reporting

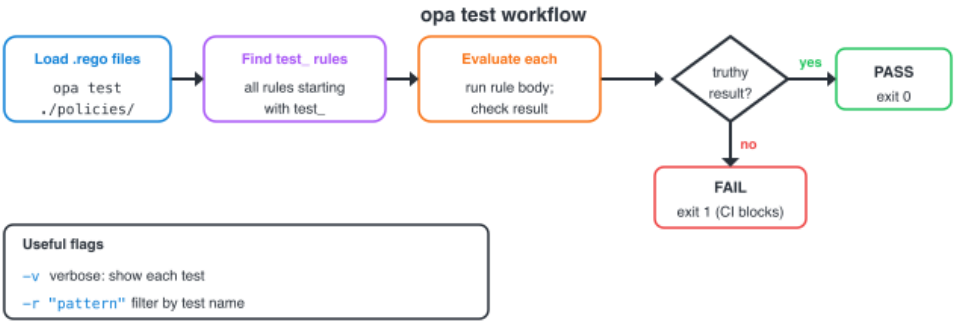
A suite of passing tests gives you confidence in the scenarios you wrote tests for. But how do you know you haven't missed a rule or scenario? That's what test coverage is for. Coverage shows which lines of policy your tests actually exercised:

```
opa test --coverage ./policies/
```

The result is JSON. Pipe it through `jq` for a readable summary:

```
opa test --coverage ./policies/ | jq '.coverage'
```

The full report lists files and the percentage of expressions evaluated by the test suite. A line that never appears is either dead code or a gap. Coverage is a map, not a guarantee. One hundred percent coverage does not mean the policy is correct. It does mean you can spot rules with no tests at all and the places silent bugs like Chapter 15's typo tend to hide.



opa test loads all .rego files, finds every test_ rule, evaluates each, and exits non-zero on any failure -- plug it into CI with opa test ./policies/ || exit 1.

Structuring a policy project for testability

Tests reward policy structure as much as policy authors. Three habits keep EmptyBox's rental tree easy to verify as it grows.

Keep test files alongside policy files

Place tests next to the rules they cover, using the `_test.rego` suffix:

```
policies/
├── rental/
│   ├── allow.rego
│   ├── allow_test.rego          # tests for allow.rego
│   └── deny.rego
├── rental/membership/
│   ├── policy.rego
│   └── policy_test.rego
└── rental/lib/
 ├── ids.rego
 └── ids_test.rego
```

When `allow.rego` and `allow_test.rego` sit in the same folder, a missing test file is obvious. It also means you don't have to hunt across the tree to find how membership is verified.

Extract logic into testable helper rules

One monolithic `allow` rule that checks membership, age, stock, and hours in a single body is hard to test in pieces. Split the decision into helpers; test each helper with minimal input. This is one of the benefits of the code organization skills you picked up in Chapter 13: modular code is more testable, and tests against each domain policy require less data and provide better documentation for your policies.

Test edge cases explicitly

Every rule deserves at least two tests: the case where it should hold, and the case where it should not. For partial rules that build sets, also test the empty case. When a rule sits at a boundary, it's worth testing either side of that boundary, like you did with the age check.

What comes next

Your rental policy declares what should be true and you validate it every time the test suite runs. The next chapter turns to patterns and practices that keep large projects maintainable. You'll learn about allow/deny shapes, helper design, input normalization, linting, and structural choices.

Key concepts

- Any rule beginning with `test_` is a test case. It passes if it evaluates to a truthy value and fails if it evaluates to `false` or `undefined`.
- `with` is the primary testing tool, and you can use it to supply known `input`, override paths under `data`, and mock built-in functions for deterministic results.
- `opa test ./policies/` runs all tests and exits with a non-zero status on failure, making it CI-friendly. Add `-v` for verbose output and `-r` to filter by name.
- `opa test --coverage` reports which policy lines were exercised. Uncovered lines are gaps, not guarantees.
- Keep test files adjacent to the policies they test, named with the `_test.rego` suffix.
- Extract logic into helper rules: small, single-purpose helpers are easier to test independently than monolithic rules.
- Always test both the firing case and the non-firing case. For partial rules, also test that the result is empty when it should be.

CHAPTER 17

Patterns and best practices

You're almost there. You nailed the basics, added some structure, used annotations and schemas, and tested your policies. These are the skills of a Rego expert.

To make it all scale for managing complex policies and collaborating with a team of other Rego heroes, you need robust patterns and practices. This chapter will lay a practical layer on top of your Rego language skills that will help you work as well on day 1,000 of your policies as you do on day one.

The allow/deny pattern

The employee at EmptyBox Rentals rarely stops at a yes-or-no answer. A denial comes with a reason, whether that's inactive membership, outstanding fees, or no available copies of a film.

Authorization policies in production work the same way. Integrators want a final decision *and* enough detail for logs, support tickets, and user-facing displays. The most common architecture for that shape uses two partial set rules, `allow` and `deny`, combined at the top:

```
package rental

default allow := false

# Explicit deny overrides any allow
allow if {
  count(deny) == 0
  count(permit) > 0
}

# Continued...
```

```
# ...continued

# Collect reasons to deny
deny contains msg if {
  not input.member.membership.active
  msg := "membership is not active"
}

deny contains msg if {
  input.member.membership.fees_outstanding
  msg := sprintf("outstanding late fees of £%.2f", [input.member.late_fees])
}

deny contains msg if {
  input.film.available_copies < 1
  msg := $"no copies of {input.film.title} available"
}

deny contains msg if {
  input.member.age < to_number(input.film.rating)
  input.member.membership.tier != "premier"
  msg := sprintf(
    "member is %v, film requires age %v (or premier tier)",
    [input.member.age, input.film.rating]
  )
}

# Collect reasons to permit
permit contains "member meets age requirement" if {
  input.member.age >= to_number(input.film.rating)
}

permit contains "premier member bypasses age restriction" if {
  input.member.membership.tier == "premier"
}
```

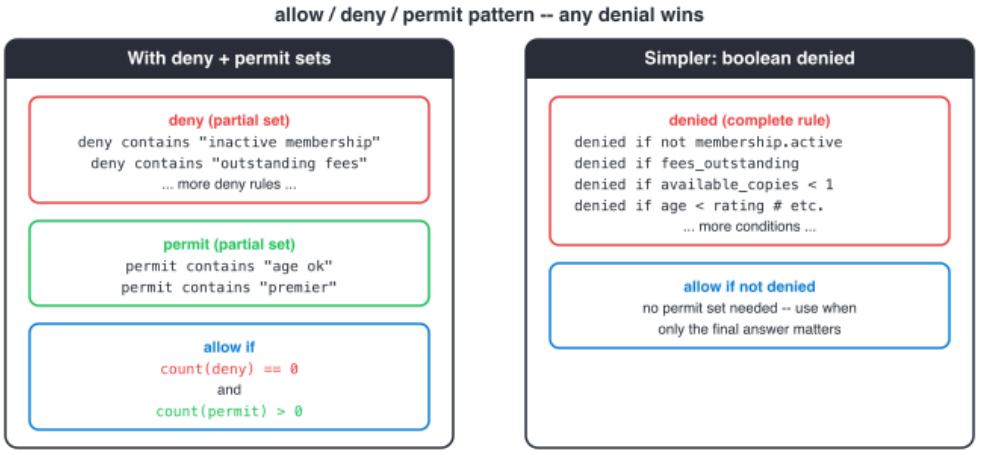
Run this in the pre-populated playground (**oc.to/rego-playground**). The policy engine denies the original request because Alex isn't old enough to rent *Pulp Fiction*. If you change Alex's age in the input panel to 18, the policy engine allows the request. Crucially, this pattern means that the presence of *any* deny message will prevent the rental, regardless of whether there are any allow messages. Rentals are only allowed when there are no deny messages and at least one allow message.

The top-level `allow` rule encodes the policy you want at the register: **any denial wins**, and **at least one permit must exist**. The individual `deny` and `permit` rules are additive, so you can add a new check by adding a rule, rather than by editing existing logic.

When you only need the final answer and never surface reasons, a slimmer variant keeps the same deny-wins spirit without a `permit` set:

```
package rental

default allow := false

allow if not denied

denied if not input.member.membership.active
denied if input.member.membership.fees_outstanding
denied if input.film.available_copies < 1
denied if {
  input.member.membership.tier != "premier"
  input.member.age < to_number(input.film.rating)
}
```

You can use either of these forms to create robust and maintainable policies. Opt for the deny/permit sets shape when you need to explain the reasons for a decision, and the boolean `denied` form when only the result matters to the caller.



Any denial wins regardless of permits; use deny+permit sets when callers need reasons, boolean denied when only the result matters.

Avoid silent defaults

Silent defaults become serious trip-hazards at scale. Adding `default allow := false` makes sure you get a result when nothing else matches. When you don't set a default, you open a crack that something will eventually fall through, because at some point, `allow` will be `undefined` when no rules fire, and that's not the same as `false`. Some OPA integrations treat undefined the same as false, but many don't. A silent default in areas like authorization could become a dangerous attack vector.

Helper rules

Helper rules extract reusable logic from rule bodies and give it a name. At EmptyBox Rentals, the employee runs a mental checklist before handing over the tape. "The membership is good, stock is on hand, the age is okay, and outstanding fees are acceptable." Helpers let your policy read the same way.

Name what you mean

Compare a single `allow` rule that inlines every condition with one that names them:

Without helpers:

```
package rental

allow if {
  input.member.membership.active
  not input.member.membership.fees_outstanding
  input.film.available_copies > 0
  input.member.age >= to_number(input.film.rating)
  input.member.late_fees < data.store.max_late_fees_before_block
}
```

With helpers:

```
package rental

allow if {
  membership_valid
  film_available
  age_permitted
  fees_acceptable
}

membership_valid if {
  input.member.membership.active
  not input.member.membership.fees_outstanding
}

film_available if input.film.available_copies > 0

age_permitted if input.member.age >= to_number(input.film.rating)
age_permitted if input.member.membership.tier == "premier"

fees_acceptable if {
  input.member.late_fees < data.store.max_late_fees_before_block
}
```

The version with helpers is like a newspaper. The headlines are easy to scan, and you can read the details if you're interested in finding out more. A new policy maintainer will be able to understand the conditions for `allow` without having to read every line of code. Your tests will also be more readable, as you can test each helper independently, reducing the amount of input and data required and simplifying test data and mocks.

Keep helpers focused

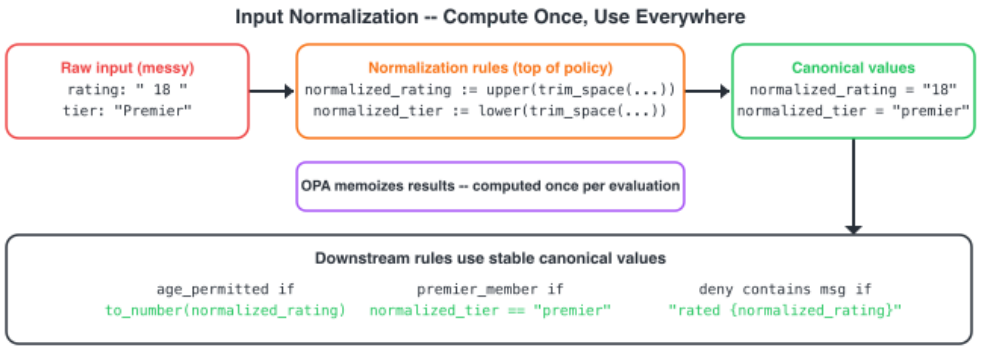
A helper should answer one question. If the name needs "and" to describe it, like `membership_valid_and_fees_paid`, split it into two helpers instead.

Input normalization

In real life, input is messy. Fields can be missing, strings may have inconsistent casing, and types sometimes need to be parsed. Repeating normalization in every rule opens opportunities for inconsistent handling of values, so normalize once at the top and write downstream rules against the stable values:

```
package rental

# Normalize input once, use throughout
normalized_rating := upper(trim_space(input.film.rating))
normalized_tier   := lower(trim_space(input.member.membership.tier))

age_permitted if {
  min_age := to_number(normalized_rating)
  input.member.age >= min_age
}

premier_member if normalized_tier == "premier"

deny contains msg if {
  not age_permitted
  not premier_member
  msg := $"member is {input.member.age} but film is rated {normalized_rating}"
}
```

Normalization rules run once per evaluation. The policy engine memoizes the results, so you don't pay a per-rule string-trimming tax.



Normalize once at the top; OPA memoizes the result so all downstream rules see the same canonical value without repeated string trimming.

Use "object.get" for optional fields

When a field might be absent from *input*, referencing it directly can leave downstream rules undefined in ways that are hard to trace. `object.get` makes the default explicit:

```
package rental

# Default to 0.00 if late_fees is not present
late_fees := object.get(input.member, "late_fees", 0.00)

# Default to "classic" if tier is not present
member_tier := object.get(input.member.membership, "tier", "classic")
```

Every rule that uses `late_fees` or `member_tier` now agrees that "missing" means £0.00 and `"classic"`, instead of each rule failing quietly on its own.

Avoiding the common pitfalls

Several patterns compile and still encode the wrong decision. They are worth recognizing before they reach production.

The "!=" in a loop bug

Chapter 8 showed that `!=` inside a loop does not mean "no element matches." It succeeds once for *each* element that fails the comparison. Alex still has two rentals out: *Speed* (current) and *The Lion King* (overdue):

```
# WRONG: fires if any rental item is not overdue (i.e., if Speed is current)
deny contains "has overdue rentals" if {
  some item in input.member.rentals.items
  item.overdue != true
}

# RIGHT: fires if any rental item IS overdue
deny contains "has overdue rentals" if {
  some item in input.member.rentals.items
  item.overdue
}
```

The wrong rule fires because *Speed* has `overdue != true`. The right rule fires because *The Lion King* is overdue. Prefer positive membership checks in loops, not negated comparisons.

When you need a universal condition, deny only if *every* item is overdue, use the `every` keyword:

```
# Correct: fires only if every item is overdue
deny contains "all rentals overdue" if {
  every item in input.member.rentals.items {
    item.overdue
  }
}
```

With Alex's mix of current and overdue tapes, that rule does not fire; it would fire only if every line on his account were overdue.

Confusing "{}" with "set()"

An empty object literal and an empty set look alike in Rego: `{}`. The policy engine treats a bare `{}` as an empty object. To build an empty set, use `set()`:

```
# This is an empty object, not a set
x := {}

# This is an empty set
y := set()
```

This bites when you compare a partial set rule to empty. `count(deny) == 0` is safe as `count` works on sets. `deny == {}` is always false when `deny` is a set, even an empty one. Stick to `count(deny) == 0` when you mean "no denials."

Long "else" chains

Chapter 11 introduced `else` and warned against long chains. More than three steps usually means the logic wants a lookup table:

```
# Fragile: order-dependent, hard to test middle cases
max_rentals := 10 if input.member.membership.tier == "premier"
max_rentals := 5  if input.member.membership.tier == "plus"
max_rentals := 3  if input.member.membership.tier == "classic"
max_rentals      else := 0

# Better: lookup table, easy to extend
tier_limits := {
  "premier": 10,
  "plus":    5,
  "classic": 3,
}

max_rentals := tier_limits[input.member.membership.tier]
max_rentals := 0 if not input.member.membership.tier in object.keys(tier_limits)
```

The table form is easier to test, easier to extend (add a tier with one line), and makes the mapping visible to whoever maintains store policy.

Tightly coupled packages

A common mistake is reaching from a library package back into a policy package. Chapter 13's layering, where lib is at the bottom and entry points are at the top, avoids circular imports and keeps packages testable on their own. When you feel tempted to have `rental.lib` import something from `rental.membership`, the logic probably belongs in a different layer or a shared lib module both can import.

The Regal linter

Regal is the official linter for Rego. It flags style problems, deprecated syntax, and patterns that work but obscure intent, including issues `opa check` will not catch. Running it in CI is how EmptyBox Rental's policy stays consistent as more authors touch the same files:

```
regal lint ./policies/
```

`regal` ships with rules covering:

- Use of deprecated syntax (e.g., `p[x] { ... }` instead of `p contains x if { ... }`)
- Missing `default` declarations for complete rules
- Unification (`=`) where assignment (`:=`) was probably intended
- Overly complex rule bodies that should be split into helpers
- Missing `import rego.v1` for new-style syntax

You can tune the severity per rule. A `.regal/config.yaml` in the project root controls that:

```
rules:
  style:
    todo-comment:
      level: ignore     # don't flag TODO comments
  idiomatic:
    use-assignment-operator:
      level: error      # treat as a hard error
```

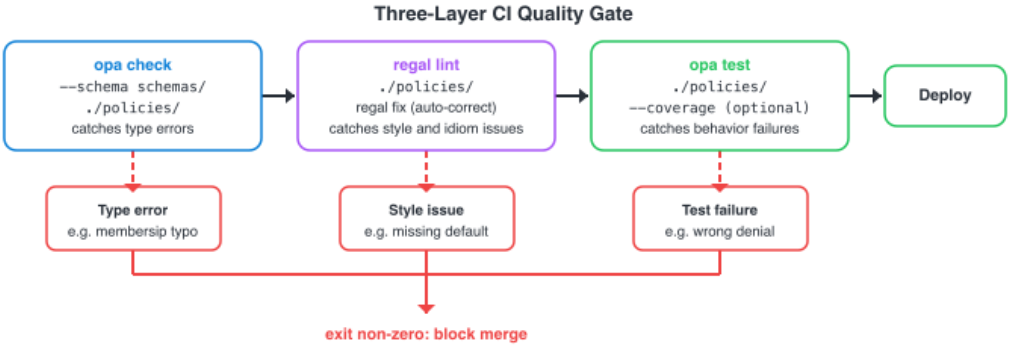
You can use `regal fix` to automatically correct many of the linter's findings. This is useful to run on a new codebase to clear mechanical noise before human review.

Using Regal alongside "opa check"

Regal and `opa check` are complementary.

- `opa check --schema schemas/ ./policies/` catches *type errors*; structural mistakes that would fail at runtime against your schemas.
- `regal lint ./policies/` catches *style and idiom issues*; code that runs but is unclear, fragile, or inconsistent.

A solid CI pipeline runs both before `opa test`.



opa check catches type errors, regal lint catches style issues, opa test catches behavior failures -- all three gate the merge.

What comes next

That's a lot of patterns, practices, and pitfalls. You're all set to be successful with policy. The final piece of the puzzle concerns how you deploy OPA and connect it to the systems you want it to govern. Change into your pajamas and brush your teeth as you're almost finished.

Key concepts

- The allow/deny pattern separates permit conditions from deny conditions. Explicit denials win. Always declare `default allow := false`.
- Helper rules name what you mean. Prefer small, focused helpers over large rule bodies. Named helpers are easier to audit and test.
- Normalize input once at the top of the policy. Use `object.get` to handle missing fields with explicit defaults.
- Avoid `!= x` inside loops and check for the positive condition instead. Use `every` for universal checks. Remember that `{}` is an empty object, not an empty set; use `count(deny) == 0` rather than `deny == {}`.
- Prefer lookup tables over long `else` chains for tier- or category-based mappings.
- `regal lint` catches style and idiom issues that `opa check` misses. Run both in CI.

CHAPTER 18

What comes next

At the end of each chapter, we shared "what comes next", but we're nearing the end of the book, so this is a much bigger "what comes next". As you've worked your way through the examples in this book, you've used the Rego playground or the OPA CLI on your machine to run everything. In production, policy and data ship to a long-running OPA process. Other systems will come over an HTTP connection to ask: "Given this input, what is data.rental. allow?" This closing chapter maps the world of OPA-as-a-service. It tackles bundles, config, decision logs, and integrations.

OPA-as-a-service

The playground and CLI for OPA let you quickly try stuff out, but OPA in production is a long-lived process. It loads policies at startup and spins up an HTTP API for other systems to call to make policy queries. Those systems might be the EmptyBox Rentals point-of-sale device, a deployment process, or a service running on Kubernetes. Anything that can communicate over HTTP can call the OPA server.

Start OPA as a server with:

```
opa run --server --addr :8181
```

The command binds an HTTP server on port 8181. Any client that can POST JSON can ask the same questions you have been asking locally.

The REST API

The heart of the integration is the Data API. To evaluate a query, POST to `/v1/data/<path>` with an `input` document in the body that has the same shape Alex's transaction would carry from the register:

```
curl -X POST http://localhost:8181/v1/data/rental/allow \
  -H 'Content-Type: application/json' \
  -d '{
    "input": {
      "member": {
        "id": "MBR-0042",
        "name": "Alex Murphy",
        "age": 16,
        "membership": {
          "tier": "classic",
          "active": true,
          "fees_outstanding": false
        },
        "late_fees": 1.50
      },
      "film": {
        "id": "VHS-2209",
        "title": "Pulp Fiction",
        "rating": "18",
        "available_copies": 1
      },
      "request": {
        "action": "rent",
        "store": "emptybox-central",
        "timestamp": "1995-11-09T18:30:00Z"
      }
    }
  }'
```

OPA responds with a JSON document:

```
{
  "result": false
}
```

If you're following the patterns and practices from the previous chapter, you'll always provide a return value for a request. An empty response would indicate that no policies were loaded. Calling systems should pay some attention to how they handle empty responses, as in most cases, they shouldn't be taken as an indication of success.

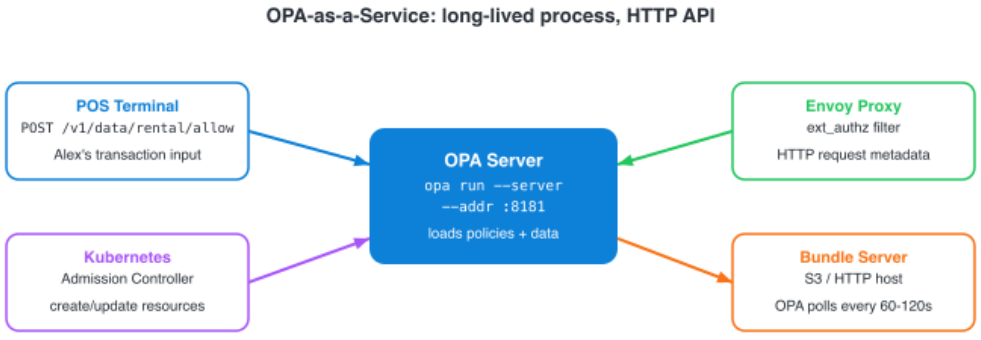
You can also change policy and data at runtime without restarting the server. POST policies through the Policy API:

```
curl -X PUT http://localhost:8181/v1/policies/rental \
 -H 'Content-Type: text/plain' \
  --data-binary @rental.rego
```

Load static configuration through the Data API:

```
curl -X PUT http://localhost:8181/v1/data/store \
  -H 'Content-Type: application/json' \
  -d '{
    "opening_hour": 10,
    "closing_hour": 21,
    "max_late_fees_before_block": 5.00
  }'
```

For a single store and a handful of policies, that is often enough: point the terminal at OPA, send member and film details, act on `result`. Once you have many instances or frequent policy updates, you will want bundles instead of hand-loading files.



Any client that can POST JSON can query OPA; OPA answers allow/deny decisions; bundles update policies atomically without restarting. POS terminals, Kubernetes admission, and Envoy all reduce to: build input, POST to /v1/data/..., act on result.

Bundles

Pushing individual `.rego` files and JSON documents through the API works for experiments and small deployments. Production fleets typically use *bundles*, which are tarballs of policies and data that OPA downloads, verifies, and applies in one atomic step. This ensures every register in the chain runs the same rental rules at the same revision.

A bundle directory has a simple layout:

```
bundle/
├── .manifest          # version, roots, metadata
├── policies/
│   ├── rental.rego
│   ├── rental/
│   │   ├── membership.rego
│   │   ├── age.rego
│   │   └── inventory.rego
└── data.json          # store config and membership limits
```

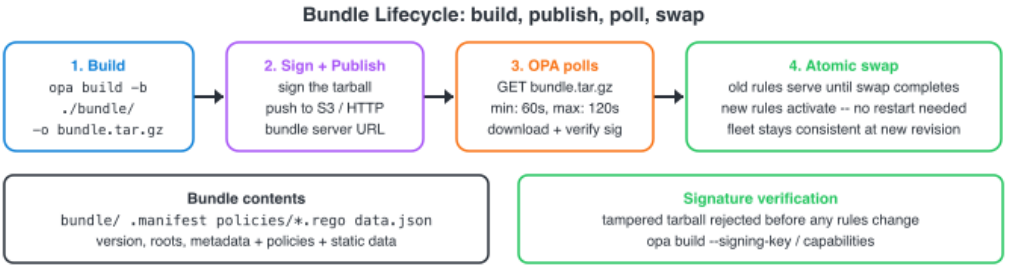
You build it with the `opa build` command:

```
opa build -b ./bundle/ -o rental-bundle.tar.gz
```

You can then point OPA at a URL (HTTP server, S3 storage, or any host that serves the bundle format):

```
# opa-config.yaml
bundles:
  rental:
    resource: https://bundles.emptybox.example.com/rental/bundle.tar.gz
    polling:
      min_delay_seconds: 60
      max_delay_seconds: 120
```

OPA polls on that interval and swaps policy only after the new bundle is fully downloaded and verified. The previous rules keep answering queries until the swap completes, so you don't need to perform a coordinated restart across a fleet of OPA instances. That's the usual distribution path for production. You can sign your bundles and have OPA verify the signature before applying it, so a tampered tarball cannot replace your rental policy on the wire.



Bundles are the standard production distribution: build, sign, publish, poll, and swap -- old rules answer queries until the new bundle is fully verified.

Decision logs

Every query can leave an audit trail. A *decision log* is a structured record of the input, the result, the query path, and timing metadata. It's what you'd want if a manager asked why Alex was turned away last Saturday evening.

You can log to the console during development:

```
decision_logs:
  console: true    # log to stdout (useful for development)
```

And you can ship logs to a remote collector in production:

```
decision_logs:
  service: logging-service
  reporting:
    min_delay_seconds: 5
    max_delay_seconds: 10

services:
  logging-service:
    url: https://logs.emptybox.example.com
```

A single rental decision might look like:

```
{
  "decision_id": "abc123",
  "input": {
    "member": {"id": "MBR-0042", "name": "Alex Murphy", "age": 16},
    "film": {"id": "VHS-2209", "title": "Pulp Fiction", "rating": "18"},
    "request": {"action": "rent", "timestamp": "1995-11-09T18:30:00Z"}
 },
  "result": false,
  "path": "data.rental.allow",
  "timestamp": "1995-11-09T18:30:01Z",
  "metrics": {"timer_rego_query_eval_ns": 45000}
}
```

You can mask fields before they are written to prevent sensitive details from being added to plain-text log storage. The policy decision should be auditable, but the sensitive information shouldn't be part of that trail.

The status API

Operators need to know whether each OPA instance is healthy and current. The Status API reports bundle revisions, whether the last download succeeded, and plugin state:

```
curl http://localhost:8181/v1/status
```

The response includes:

- Which bundles are loaded and their current revision
- Whether the last bundle download succeeded or failed
- Plugin state

Wire this into Kubernetes liveness probes or a monitoring dashboard when OPA runs as a sidecar or a deployment, so you get an alert when stores are evaluating against stale or missing policy, not when a clerk discovers it at the counter.

Integrations

OPA is built to sit in front of many kinds of decisions. The rental counter is one shape; the integrations below are where most teams meet OPA in the wild. Each one still reduces to the same question: build an `input` document, evaluate a rule path, act on the result.

Kubernetes

The most common deployment is as a Kubernetes admission controller. The API server sends create and update requests to OPA before persisting them; OPA admits or rejects the resource.

Two main options:

OPA with kube-mgmt: OPA runs as a sidecar; a companion process (`kube-mgmt`) mirrors cluster objects into OPA's `data` tree so policies can query live cluster state.

Gatekeeper: A Kubernetes operator built around OPA. It introduces `ConstraintTemplate` (parameterized Rego) and `Constraint` (a template instance with specific parameters). Platform teams ship templates; application teams instantiate constraints, which is closer to how you might ship rental policy modules once and let each store set limits.

A `ConstraintTemplate` holds the Rego; a `Constraint` binds it to resource types and parameters. The split is deliberate: one policy definition and many scoped deployments.

Envoy

Envoy can delegate authorization to OPA through the External Authorization API. On each request, Envoy calls OPA; the Envoy plugin maps the request into `input` and returns allow or deny before traffic reaches the upstream service. That pattern fits well with service meshes: authorization stays in one place rather than spreading across every microservice.

Terraform

Before `terraform apply`, you can evaluate the plan as policy input. `terraform show -json` produces machine-readable JSON and Rego checks for security violations, disallowed resource types, or cost guardrails.

Conftest wraps that workflow:

```
terraform plan -out tfplan
terraform show -json tfplan | conftest test -
```

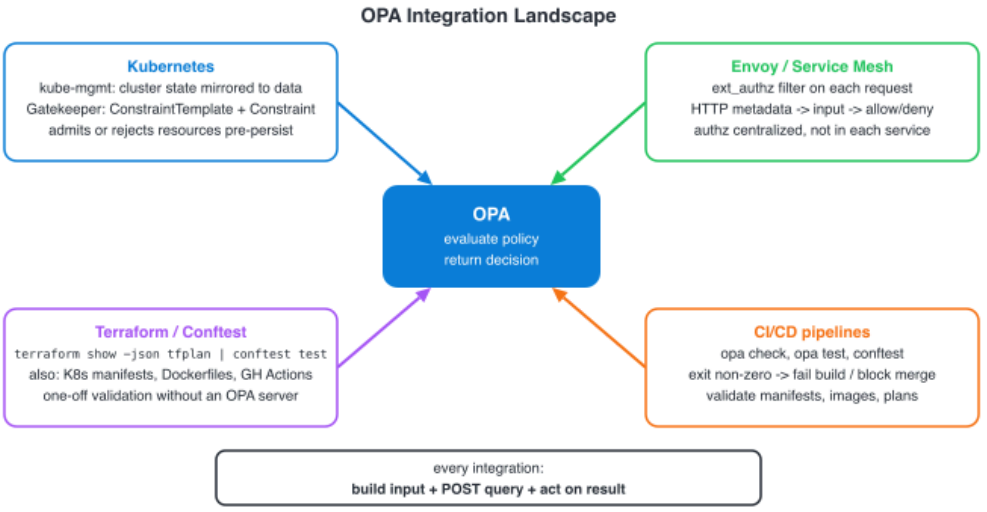
Conftest is a general policy runner on top of OPA. It accepts Terraform plans, Kubernetes manifests, Dockerfiles, GitHub Actions workflows, and other structured config. It works anywhere you want Rego validation without standing up a full OPA server for a one-off check.

CI/CD pipelines

Pipelines are a natural home for policy gates:

- Validate Kubernetes manifests before deployment
- Ensure container images come from approved registries
- Block Terraform plans that violate security or cost rules
- Enforce naming and labeling conventions

`opa check`, `opa test`, and `conftest` exit non-zero on failure, so a failing policy fails the build or blocks a merge, which is the same discipline you used for Alex's rental tests, applied to infrastructure changes.



All four integration patterns reduce to the same contract: build an input document, evaluate a rule path, act on the decision.

The Rego playground

When you want to try a built-in, a comprehension, or a corner of the language without touching a repo, use the playground at **play.openpolicyagent.org**. Write policy, set `input` and `data`, evaluate, and share a URL. It's handy for questions, bug reports, or a quick sanity check before you paste a snippet into your project.

The playground is the fastest feedback loop for language experiments. Local OPA remains the right tool when you need the REPL, the test runner, or Regal.

Further learning

Official documentation: **openpolicyagent.org/docs** is the authoritative reference. The Policy Reference lists every built-in with types and examples; the How-to Guides walk through specific integrations in depth.

The Rego v1 migration guide: Rego v1 (from OPA 0.59 onward) makes several formerly optional forms mandatory. The `import rego.v1` declaration enables the syntax used throughout this book. If you inherit policies that use bracket partial rules (`p[x] { ... }` instead of `p contains x if { ... }`), the migration guide maps old forms to new ones.

Regal: The Rego linter at **github.com/StyraInc/regal** is actively maintained. Its release notes are a practical way to pick up new idioms and catch issues that tests might miss.

The OPA Slack community: The `#opa-users` channel on the CNCF Slack workspace is active and welcoming. You can ask about integrations, odd evaluation behavior, or performance tuning.

Conftest: **conftest.dev** is worth bookmarking for configuration validation. The docs include worked examples for Kubernetes, Terraform, Dockerfiles, and other formats.

Closing thoughts

You started in 1995 at EmptyBox Central with Alex Murphy, a 16-year-old with an overdue tape, £1.50 in late fees, and the empty *Pulp Fiction* box on the counter. A simple yes-or-no question turned into membership checks, age ratings, overdue detection, fee thresholds, stock checks, and opening hours. That's policy complex enough to need every feature this book introduced.

You now know how Rego evaluates rules declaratively, how undefined differs from false, how sets and comprehensions collect reasons, how to test policies thoroughly, and how to structure a project that stays readable. The patterns and practices are not Rego trivia; they're how you keep policy logic as clear and verifiable as application logic. A policy nobody can read is a policy nobody can trust.

OPA and its ecosystem keep moving: new integrations, a stronger type checker, and more lint rules. The language core is stable; what you have learned here transfers wherever OPA shows up next. The video store closed long ago, as we always knew it would. Your work wasn't wasted, though, because the Rego you wrote for the check-out desk still runs within the EmptyBox streaming systems.

The appendices that follow are reference material. There are operator tables, a built-in cheat sheet, a glossary, and a setup guide. Use them when you need a quick reminder; they support the understanding you have already built. Turn the page when you are ready, or open the playground and point it at Alex one more time.

Key concepts

- OPA runs as a long-lived HTTP server in production. The REST API accepts `input` and returns policy decisions. You can load policies and data at startup or update them at runtime.
- Bundles are the standard distribution mechanism for production: tarballs of policies and data that OPA downloads, verifies, and applies atomically.
- Decision logs record policy decisions for audit and debugging; masking redacts sensitive fields. The Status API reports bundle health and plugin state.
- Key integrations: Kubernetes (kube-mgmt or Gatekeeper), Envoy (External Authorization), Terraform (plan evaluation via Conftest), and CI/CD gates (`opa check`, `opa test`, `conftest`).
- The OPA playground is the fastest way to experiment with Rego interactively; local OPA is for tests, the REPL, and linting.

Appendices

Useful references

APPENDIX A

Operators and keywords quick reference

Assignment and equality

Operator	Name	Description	Example
`:=`	Unification / assignment	Binds a new local variable. Fails if the variable is already bound to a different value.	`x := 42`
`==`	Equality comparison	Tests that two values are equal. Does not bind variables.	`input.role == "admin"`
`=`	Unification	Unifies two terms. Binds unbound variables; fails if bound variables disagree.	`[x, y] = [1, 2]`
`!=`	Inequality	True if two values are not equal.	`input.action != "delete"`

Comparison operators

Operator	Description	Notes
`<`	Less than	Numbers and strings
`<=`	Less than or equal	Numbers and strings
`>`	Greater than	Numbers and strings
`>=`	Greater than or equal	Numbers and strings

All comparison operators require both sides to be defined. If either side is undefined, the whole expression is undefined (not `false`).

Arithmetic operators

Operator	Description	Example
`+`	Addition	`x := a + b`
`-`	Subtraction	`x := a - b`
`*`	Multiplication	`x := a * b`
`/`	Division (float)	`x := a / b`
`%`	Modulo	`x := a % b`

Logical operators

Keyword	Description	Example
`not`	Negation as failure. True when the expression is undefined or false.	`not input.user.suspended`

There is no `and` keyword. Logical AND is expressed by placing conditions on separate lines in a rule body. There is no `or` keyword. Logical OR is expressed by writing multiple rule definitions.

Membership and iteration

Operator / Keyword	Description	Example
`in`	Tests membership or iterates over a collection	`"admin" in input.roles`
`some x in collection`	Existential quantification — binds `x` to each element	`some role in input.roles`
`some x, y in collection`	Two-variable iteration over object key-value pairs	`some k, v in input.labels`
`every x in collection { ... }`	Universal quantification — true if body holds for all elements	`every role in input.roles { role != "root" }`
`_`	Wildcard / anonymous variable — discards the binding	`input.items[_].name`

Set operators

Operator	Description	Example
`|`	Set union	`a | b`
`&`	Set intersection	`a & b`
`-`	Set difference	`a - b`

Reference notation

Notation	Description	Example
`.field`	Dot notation — access a known field name	`input.user.role`
`["field"]`	Bracket notation — same as dot, required for names with special characters	`input["content-type"]`
`[variable]`	Variable key — iterates over all keys/indices, binding each to the variable	`input.users[i].name`
`[_]`	Wildcard key — iterates without binding	`input.users[_].name`

Keywords

Declaration keywords

Keyword	Description
`package`	Declares the namespace for all rules in the module
`import`	Brings a package, rule, or path into local scope
`as`	Creates an alias in an import statement

Rule keywords

Keyword	Description
`default`	Provides a fallback value for a complete rule when all other definitions are undefined
`if`	Separates the rule head from its body (optional but recommended)
`contains`	Marks a rule as a partial set contributor
`else`	Chains rule definitions in priority order; OPA stops at the first match
`with`	Overrides `input`, a `data` path, or a built-in function for the duration of one expression

Comprehension delimiters

Syntax	Type	Description
`[ term | body ]`	Array comprehension	Produces an array
`{ term | body }`	Set comprehension	Produces a set
`{ key: value | body }`	Object comprehension	Produces an object

Special values

Value	Description
`true`	Boolean true
`false`	Boolean false
`null`	JSON null
`undefined`	Not a value — the absence of a value. A rule that does not fire produces undefined.

String literal forms

Form	Description	Example
`"..."`	Standard string — backslash escapes apply	`"hello\nworld"`
`` `...` ``	Raw string — no escape processing, useful for regex patterns	`` `^\d+$` ``
`$"..."`	String interpolation — `{expression}` is evaluated and embedded	`$"Hello {input.user.name}"`

Annotation syntax

```
# METADATA
# Field: value
# Field:
#   - list item
```

Must appear immediately above the rule or package declaration, with no blank lines inside the block.

Annotation scope values

Scope	Applies to
`rule`	The single rule immediately below (default when preceding a rule)
`document`	All rules with the same name in the same package, across all files
`package`	All rules in the package, across all files
`subpackages`	The package and all packages whose path starts with the same prefix

APPENDIX B

Built-in function cheat sheet

This appendix lists the most commonly used built-in functions by category. The full reference, including type signatures for every function, is at **openpolicyagent.org/docs/policy-reference**.

Strings

Function	Returns	Description
`startswith(str, prefix)`	boolean	True if `str` begins with `prefix`
`endswith(str, suffix)`	boolean	True if `str` ends with `suffix`
`contains(str, substr)`	boolean	True if `substr` appears in `str`
`lower(str)`	string	Converts to lowercase
`upper(str)`	string	Converts to uppercase
`trim(str, cutset)`	string	Removes leading/trailing characters found in `cutset`
`trim_space(str)`	string	Removes leading/trailing whitespace
`trim_prefix(str, prefix)`	string	Removes `prefix` if present
`trim_suffix(str, suffix)`	string	Removes `suffix` if present

Function	Returns	Description
`replace(str, old, new)`	string	Replaces all occurrences of `old` with `new`
`split(str, delim)`	array	Splits `str` on `delim`, returns array of strings
`concat(delim, coll)`	string	Joins array or set elements with `delim`
`sprintf(fmt, args)`	string	C-style format string with `args` array
`count(str)`	number	Number of characters (Unicode code points) in the string
`indexof(str, substr)`	number	Index of first occurrence of `substr`, or -1
`substring(str, start, len)`	string	Substring starting at `start` with length `len`
`strings.any_prefix_match(search, prefixes)`	boolean	True if `search` starts with any string in `prefixes`
`strings.any_suffix_match(search, suffixes)`	boolean	True if `search` ends with any string in `suffixes`

Regular expressions

Function	Returns	Description
`regex.match(pattern, str)`	boolean	True if `str` matches `pattern`
`regex.is_valid(pattern)`	boolean	True if `pattern` is a valid regex
`regex.replace(str, pattern, value)`	string	Replaces matches of `pattern` in `str` with `value`
`regex.find_all_string_submatch_n(pattern, str, n)`	array	Returns up to `n` match groups; -1 for all
`regex.split(pattern, str)`	array	Splits `str` on `pattern`
`regex.globs_match(glob1, glob2)`	boolean	True if the two glob patterns match the same set of strings

Use raw strings (backticks) for patterns to avoid double-escaping: `regex.match(`^\d+$`, str)`

Numbers

Function	Returns	Description
`abs(n)`	number	Absolute value
`ceil(n)`	number	Round up to nearest integer
`floor(n)`	number	Round down to nearest integer
`round(n)`	number	Round to nearest integer
`div(a, b)`	number	Integer division
`rem(a, b)`	number	Integer remainder
`numbers.range(a, b)`	array	Inclusive integer range `[a, a+1, ..., b]`
`to_number(x)`	number	Converts string or boolean to number

Aggregates

Function	Returns	Description
`count(coll)`	number	Number of elements in array, set, or object; characters in string
`sum(coll)`	number	Sum of numeric elements in array or set
`product(coll)`	number	Product of numeric elements in array or set
`min(coll)`	any	Smallest value in array or set
`max(coll)`	any	Largest value in array or set
`sort(coll)`	array	Sorted array (accepts array or set)
`all(coll)`	boolean	True if all elements are truthy
`any(coll)`	boolean	True if at least one element is truthy

Arrays

Function	Returns	Description
`array.concat(a, b)`	array	Concatenation of two arrays
`array.slice(arr, start, end)`	array	Sub-array from `start` (inclusive) to `end` (exclusive)
`array.reverse(arr)`	array	Reversed copy of the array

Objects

Function	Returns	Description
`object.get(obj, key, default)`	any	Value at `key`, or `default` if absent. Safe access.
`object.keys(obj)`	set	Set of all keys
`object.values(obj)`	array	Array of all values
`object.union(a, b)`	object	Merged object; `b`'s values win on key conflicts
`object.union_n(arr)`	object	Merges an array of objects
`object.remove(obj, keys)`	object	New object without the specified keys
`object.filter(obj, keys)`	object	New object with only the specified keys
`object.subset(super, sub)`	boolean	True if `sub` is a subset of `super`
`json.filter(obj, paths)`	object	New object keeping only specified paths
`json.remove(obj, paths)`	object	New object with specified paths removed

Sets

Set operators are used as infix expressions, not function calls:

Operator	Description	Example
`a | b`	Union	`{1,2} | {2,3}` → `{1,2,3}`
`a & b`	Intersection	`{1,2} & {2,3}` → `{2}`
`a - b`	Difference	`{1,2} - {2,3}` → `{1}`

Set built-in functions:

Function	Returns	Description
`intersection(sets)`	set	Intersection of an array of sets
`union(sets)`	set	Union of an array of sets

Type checking

Function	Returns	Description
`is_string(x)`	boolean	True if `x` is a string
`is_number(x)`	boolean	True if `x` is a number
`is_boolean(x)`	boolean	True if `x` is a boolean
`is_null(x)`	boolean	True if `x` is null
`is_array(x)`	boolean	True if `x` is an array
`is_object(x)`	boolean	True if `x` is an object
`is_set(x)`	boolean	True if `x` is a set
`type_name(x)`	string	Returns `"string"`, `"number"`, `"boolean"`, `"null"`, `"array"`, `"object"`, or `"set"`

Encoding and decoding

Function	Returns	Description
`json.marshal(val)`	string	Encodes Rego value as JSON string
`json.unmarshal(str)`	any	Parses JSON string to Rego value
`json.is_valid(str)`	boolean	True if `str` is valid JSON
`json.match_schema(doc, schema)`	boolean	True if `doc` matches the JSON Schema
`yaml.marshal(val)`	string	Encodes Rego value as YAML string
`yaml.unmarshal(str)`	any	Parses YAML string to Rego value
`yaml.is_valid(str)`	boolean	True if `str` is valid YAML
`base64.encode(str)`	string	Base64-encodes a string
`base64.decode(str)`	string	Decodes a base64 string
`base64url.encode(str)`	string	URL-safe base64 encode
`base64url.decode(str)`	string	URL-safe base64 decode
`urlquery.encode(str)`	string	Percent-encodes a string for use in a URL query
`urlquery.decode(str)`	string	Decodes a percent-encoded string
`urlquery.encode_object(obj)`	string	Encodes an object as a URL query string
`hex.encode(str)`	string	Hex-encodes a string
`hex.decode(str)`	string	Decodes a hex string

JWT

Function	Returns	Description
`io.jwt.decode(token)`	`[header, payload, sig]`	Decodes without verifying signature
`io.jwt.decode_verify(token, constraints)`	`[valid, header, payload]`	Decodes and verifies signature and claims
`io.jwt.verify_hs256(token, secret)`	boolean	Verifies HMAC-SHA256 signature
`io.jwt.verify_hs384(token, secret)`	boolean	Verifies HMAC-SHA384 signature
`io.jwt.verify_hs512(token, secret)`	boolean	Verifies HMAC-SHA512 signature
`io.jwt.verify_rs256(token, cert)`	boolean	Verifies RSA-SHA256 signature
`io.jwt.verify_es256(token, cert)`	boolean	Verifies ECDSA-SHA256 signature
`io.jwt.encode_sign(header, payload, key)`	string	Creates and signs a JWT

Time

All time functions work in nanoseconds (Unix epoch). 1 second = 1,000,000,000 nanoseconds.

Function	Returns	Description
`time.now_ns()`	number	Current time as Unix nanoseconds
`time.parse_rfc3339_ns(str)`	number	Parses RFC3339 string to nanoseconds
`time.parse_ns(layout, str)`	number	Parses string using Go time layout
`time.format(ns)`	string	Formats nanoseconds as RFC3339 string
`time.date(ns)`	`[year, month, day]`	Extracts calendar date (UTC)
`time.clock(ns)`	`[hour, min, sec]`	Extracts time of day (UTC)
`time.weekday(ns)`	string	Day name: `"Monday"`, `"Tuesday"`, etc.
`time.diff(ns1, ns2)`	`[years, months, days, hours, mins, secs]`	Difference between two timestamps
`time.add_date(ns, years, months, days)`	number	Adds a duration to a timestamp

Cryptography

Function	Returns	Description
`crypto.md5(str)`	string	MD5 hex digest
`crypto.sha1(str)`	string	SHA-1 hex digest
`crypto.sha256(str)`	string	SHA-256 hex digest
`crypto.hmac.md5(str, key)`	string	HMAC-MD5 hex digest
`crypto.hmac.sha1(str, key)`	string	HMAC-SHA1 hex digest
`crypto.hmac.sha256(str, key)`	string	HMAC-SHA256 hex digest
`crypto.hmac.sha512(str, key)`	string	HMAC-SHA512 hex digest
`crypto.x509.parse_certificates(pem)`	array	Parses PEM-encoded X.509 certificates

Networking

Function	Returns	Description
`net.cidr_contains(cidr, addr)`	boolean	True if `addr` is within the CIDR range
`net.cidr_intersects(cidr1, cidr2)`	boolean	True if the two CIDR ranges overlap
`net.cidr_expand(cidr)`	set	All IP addresses in the CIDR range
`net.lookup_ip_addr(name)`	set	DNS lookup — returns set of IP addresses
`http.send(request)`	object	Makes an HTTP request (use carefully; prefer mocking in tests)

OPA and metadata

Function	Returns	Description
`rego.metadata.rule()`	object	Annotation metadata for the current rule
`rego.metadata.chain()`	array	Annotation metadata chain from rule to subpackages scope
`opa.runtime()`	object	Information about the running OPA instance (version, labels, etc.)

Tracing and debugging

Function	Description
`print(values...)`	Prints values to stderr during evaluation. For debugging only — has no effect on the result. Requires `--instrument` flag or `strict-builtin-errors` mode to be visible in some contexts.

Notes on runtime errors

By default, built-in functions that encounter errors at runtime (dividing by zero, decoding malformed base64, etc.) produce `undefined` rather than raising an exception. The expression containing them fails silently and the rule does not fire.

To make runtime errors halt evaluation, use the `--strict-builtin-errors` flag with `opa eval`:

```
opa eval --strict-builtin-errors \
  --data policy.rego \
  --input input.json \
  'data.example.allow'
```

APPENDIX C
Glossary

annotation Structured documentation embedded in a policy file using `# METADATA` comment blocks. Annotations are parsed by OPA at compile time and can be read at runtime via `rego.metadata.rule()` and `rego.metadata.chain()`. Fields include `title`, `description`, `authors`, `custom`, and `entrypoint`. See Chapter 14.

base document A document loaded into OPA from an external source, for example a JSON or YAML file, an API response, or a bundle. Base documents are accessible under the `data` tree. Contrast with *virtual document*.

binding The association of a variable with a value during unification or iteration. Once a variable is bound, it cannot be rebound to a different value within the same scope. This is what makes Rego variables behave differently from variables in imperative languages.

body The part of a rule that contains the conditions. All conditions in a rule body must be true (not undefined) for the rule to fire. Conditions in a body are connected by logical AND. See Chapters 2 and 4.

bundle A tarball containing policies and data that OPA can download, verify, and apply atomically. Bundles are the standard distribution mechanism for production OPA deployments. Build bundles with `opa build`; configure OPA to fetch them with a bundle configuration block. See Chapter 18.

complete rule A rule that defines a single value. When evaluated, a complete rule either produces its value or is undefined. Multiple complete rule definitions with the same name must all agree on the value (they are alternatives via logical OR, but the value they produce must match). Contrast with *partial rule*. See Chapter 5.

comprehension An expression that constructs a collection by iterating over a body. The three forms are array comprehension `[term | body]`, set comprehension `{term | body}`, and object comprehension `{key: value | body}`. See Chapter 7.

data document The document rooted at `data` in OPA's virtual document model. It contains both base documents (loaded from external sources) and virtual documents (produced by rules). Any rule in package `foo.bar` contributes to `data.foo.bar`.

decision log A record of every policy decision OPA makes, including the input, the result, and metadata. Decision logs are used for audit, compliance, and debugging. See Chapter 18.

default A keyword that provides a fallback value for a complete rule when all other definitions produce undefined. The default value must be a literal (scalar or composite), not a reference. See Chapter 11.

defined A rule, expression, or reference is defined when it produces a value. Contrast with *undefined*.

document In OPA, a document is any JSON value accessible in the policy evaluation context. The two root documents are `input` (the query-time data provided by the caller) and `data` (policies and base documents loaded into OPA). See Chapter 1.

else A keyword that chains rule definitions in priority order. OPA evaluates each definition in order and stops at the first that produces a value. Use sparingly — long `else` chains are hard to reason about. See Chapter 11.

entrypoint A rule or package marked as an entrypoint via the `entrypoint: true` annotation. `opa build` uses entrypoints to determine which rules to include in a compiled bundle. See Chapter 14.

existential quantification A claim that at least one element in a collection satisfies some condition. In Rego, expressed with `some x in collection { ... }` or with the `some` keyword in combination with a rule body that binds the variable. Contrast with *universal quantification*. See Chapter 9.

expression Any term that OPA can evaluate to a value or undefined. Expressions include references (`input.user.role`), function calls (`startswith(...)`), comparisons (`x == y`), and comprehensions.

function A rule that takes arguments and returns a value. Defined with `f(arg) := value if { ... }`. Functions can have multiple definitions for conditional dispatch (pattern matching on arguments). See Chapter 10.

ground term A term that contains no unbound variables. A ground term has a fully determined value. `"admin"` and `{"role": "admin"}` are ground terms. `x` is not a ground term until it is bound.

head The part of a rule before the body — the rule name, any parameters, and the value (for complete and function rules). For a rule `allow if { ... }`, the head is `allow`. See Chapter 5.

incremental rule A rule with multiple definitions that contribute values to the same collection. For sets: each definition that fires contributes its term to the set. For objects: each definition contributes a key-value pair. Incremental rules are how logical OR is expressed in Rego. See Chapter 5.

input document The document rooted at `input`. It contains the query-time data provided by whatever system is calling OPA — the context for the current policy evaluation. Each query supplies a fresh `input`.

module A single `.rego` file. Every module has exactly one `package` declaration, followed by zero or more `import` statements and zero or more rule definitions. Multiple modules can declare the same package — their rules are merged. See Chapter 13.

namespace The hierarchical path created by a `package` declaration. Rules in `package authz.api` live in the `data.authz.api` namespace. See Chapter 13.

negation The `not` keyword in Rego implements *negation as failure*: `not expr` is true when `expr` is undefined or false. This is different from logical negation; it applies to the definedness of an expression. See Chapter 8.

negation as failure The semantics of `not` in Rego (and Datalog generally). `not expr` succeeds when `expr` fails to produce a value. This means `not expr` does not require `expr` to be `false`; it requires `expr` to be undefined or false. See Chapter 8.

package A namespace declaration that places all rules in a module under a dotted path. The package `foo.bar.baz` makes all rules accessible under `data.foo.bar.baz`. Multiple modules can share a package. See Chapter 13.

partial rule A rule that contributes to a collection rather than defining a single complete value. Partial set rules contribute elements to a set; partial object rules contribute key-value pairs to an object. Contrast with *complete rule*. See Chapter 5.

policy In OPA, a policy is a set of rules that express conditions under which decisions should be made. Policies are written in Rego and loaded into OPA for evaluation.

reference A path into a document. `input.user.role` is a reference. References use dot notation for known field names and bracket notation for computed keys or names with special characters. A reference to an absent field is undefined. See Chapter 4.

Rego The policy language used by OPA. Pronounced "ray-go." Rego is a declarative, logic-based language descended from Datalog. Policies are expressed as rules that define virtual documents.

rule The fundamental unit of Rego policy. A rule has a head (name and value) and optionally a body (conditions). If the body conditions all hold, the rule fires and the head value is produced. See Chapter 5.

safety A property of rules and expressions. An expression is safe if every variable referenced in it is either bound by an earlier expression in the same rule body or appears in the rule head. OPA enforces safety at compile time — a rule with an unsafe variable is an error. See Chapter 8.

schema A JSON Schema definition attached to `input` or `data` to enable static type checking. With a schema, OPA can catch structural errors (field name typos, wrong navigation paths) at compile time via `opa check`. See Chapter 15.

set An unordered collection of unique values. Created with `{1, 2, 3}` syntax or `set()` for the empty set (bare `{}` is an empty object). Set operations: union (`|`), intersection (`&`), difference (`-`). See Chapter 3.

type error An error reported by OPA's static type checker when a policy accesses a field or uses a value in a way that contradicts an attached schema. Type errors are caught at compile time by `opa check`, before the policy runs. See Chapter 15.

undefined The absence of a value. A rule that fires no definitions is undefined. A reference to an absent field is undefined. In most cases, undefined propagates — any expression that depends on an undefined value is itself undefined. `not undefined_expr` is `true`. See Chapters 2 and 8.

unification The `=` operator. Unification either compares two bound values for equality or binds an unbound variable to the value of the other side. Used for pattern matching and destructuring. See Chapter 6.

universal quantification A claim that all elements in a collection satisfy some condition. In Rego, expressed with `every x in collection { ... }`. The `every` keyword was introduced in OPA 0.38. Contrast with *existential quantification*. See Chapter 9.

variable A named placeholder for a value within a rule. In Rego, variables are local to the rule and cannot be reassigned — once bound, a variable's value is fixed for the lifetime of that rule evaluation. Variables beginning with `_` (including the bare `_`) are anonymous and can appear multiple times without the constraint of shared binding. See Chapter 4.

virtual document A document produced by evaluating a rule, as opposed to a base document loaded from an external source. `data.authz.allow` is a virtual document if `allow` is a rule in `package authz`. Virtual documents are computed on demand during evaluation. Contrast with *base document*.

with A keyword that overrides the value of `input`, a path under `data`, or a built-in function for the duration of a single expression. The primary tool for writing tests with controlled inputs. See Chapter 11.

APPENDIX D

Setting up OPA (local and CI)

Installing OPA locally

macOS

Install OPA with Homebrew:

```
brew install opa
```

Verify the installation:

```
opa version
```

Linux

Download the latest binary directly from GitHub:

```
curl -L -o opa \
  https://openpolicyagent.org/downloads/latest/opa_linux_amd64_static
chmod +x opa
sudo mv opa /usr/local/bin/
```

Or, for a specific version (replace `v0.68.0` with the version you want):

```
OPA_URL="https://github.com/open-policy-agent/opa/releases/\
download/v0.68.0/opa_linux_amd64_static"
curl -L -o opa "$OPA_URL"
chmod +x opa
sudo mv opa /usr/local/bin/
```

Windows

Download the `.exe` from the **OPA releases page**, rename it to `opa.exe`, and add it to a directory on your `PATH`.

Or, with Scoop:

```
scoop install opa
```

Docker

OPA is available as a Docker image:

```
docker run --rm openpolicyagent/opa:latest version
```

For local development, mount your policy directory:

```
docker run --rm \
  -v $(pwd)/policies:/policies \
  openpolicyagent/opa:latest \
  eval --data /policies --input input.json 'data.authz.allow'
```

Installing regal

`regal` is the official Rego linter. Install it separately from OPA.

macOS

```
brew install styrainc/packages/regal
```

Linux

```
curl -L -o regal \
  https://github.com/StyraInc/regal/releases/latest/download/regal_Linux_x86_64
chmod +x regal
sudo mv regal /usr/local/bin/
```

Verify

```
regal version
```

Project structure

A minimal project structure for getting started:

```
my-policy-project/
├── .regal/
│   └── config.yaml      # regal linter configuration (optional)
├── schemas/
│   └── input.json       # JSON Schema for your input document
├── policies/
│   ├── authz.rego
│   └── authz_test.rego
└── Makefile             # optional convenience targets
```

A sample `Makefile`:

```
.PHONY: check lint test

check:
  opa check --schema schemas/ ./policies/

lint:
  regal lint ./policies/

test:
  opa test -v ./policies/

all: check lint test
```

Run the full quality gate with:

```
make all
```

The OPA playground

No installation needed. Visit **play.openpolicyagent.org** in any browser. The playground provides:

- A policy editor with syntax highlighting
- An input editor for your JSON input document
- A data editor for `data` documents
- A query field
- Instant evaluation results

The playground is the fastest way to experiment with Rego without setting up a local environment. You can share policies via URL, which makes it useful for asking questions on the CNCF Slack or filing bug reports.

Running OPA interactively

The `opa run` command starts an interactive REPL (read-eval-print loop). This is useful for exploring how expressions evaluate without writing a full policy file:

```
opa run
```

Inside the REPL:

```
> input := {"user": {"role": "admin"}}
Rule 'input' defined in package repl. Type 'show' to see rules.
> input.user.role
"admin"
> "admin" in {"admin", "editor"}
true
> exit
```

Load policy files and data into the REPL at startup:

```
opa run policies/authz.rego data/roles.json
```

Running OPA as a server

Start OPA as an HTTP server on port 8181:

```
opa run --server --addr :8181
```

Load policies at startup:

```
opa run --server --addr :8181 --data ./policies/
```

Load policies and a configuration file:

```
opa run --server --addr :8181 --config-file opa-config.yaml
```

A minimal `opa-config.yaml` for bundle-based deployment:

```
bundles:
  authz:
    resource: http://localhost:9000/bundle.tar.gz
    polling:
      min_delay_seconds: 30
      max_delay_seconds: 60

decision_logs:
  console: true
```

Test that the server is running:

```
curl http://localhost:8181/health
```

CI/CD integration

GitHub actions

A complete workflow that checks, lints, and tests your policies on every pull request:

```yaml
name: Policy CI

on:
  push:
    branches: [main]
  pull_request:
    branches: [main]

jobs:
  policy-checks:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4

      - name: Install OPA
        run: |
          URL="https://openpolicyagent.org/downloads/latest"
          curl -L -o opa "$URL/opa_linux_amd64_static"
          chmod +x opa
          sudo mv opa /usr/local/bin/

      - name: Install regal
        run: |
          URL="https://github.com/StyraInc/regal/releases/latest/download"
          curl -L -o regal "$URL/regal_Linux_x86_64"
          chmod +x regal
          sudo mv regal /usr/local/bin/

      - name: Type check policies
        run: opa check --schema schemas/ ./policies/

      - name: Lint policies
        run: regal lint ./policies/

      - name: Run tests
        run: opa test -v ./policies/
```

GitLab CI

```
policy-checks:
  image: openpolicyagent/opa:latest
  stage: test
  script:
    - opa check --schema schemas/ ./policies/
    - opa test -v ./policies/
```

Note: The official OPA Docker image does not include `regal`. For the full quality gate in GitLab CI, use a base image with OPA pre-installed and add `regal` in the script, or build a custom image.

Exit codes

All three tools exit with non-zero status on failure:

Command	Non-zero exit when
`opa check`	Any type errors found
`regal lint`	Any lint violations at `error` level
`opa test`	Any test failures, or no tests found

This makes them reliable CI gates — the pipeline fails automatically when policies have problems.

Keeping OPA up to date

OPA releases frequently. Subscribe to release notifications on GitHub at **github.com/open-policy-agent/opa** to stay informed about new features and bug fixes. When upgrading, check the release notes for breaking changes. OPA maintains strong backward compatibility within major versions, but built-in function behavior and evaluation semantics occasionally change in minor releases.

The **OPA changelog** is the authoritative record of changes between versions.

Authors

John Bristowe

John Bristowe is a Principal Developer Advocate at Octopus Deploy, where he creates practitioner-focused content on Continuous Delivery, deployment governance, and Platform Engineering. He previously worked at Microsoft and Progress/Telerik.

Steve Fenton

Steve Fenton is a Principal DevEx Researcher at Octopus Deploy, a DORA Community Guide, and a 8-time Microsoft MVP with more than two decades of experience in software delivery. He has written books on TypeScript (Apress, InfoQ), Octopus Deploy, and Web Operations.

Matt Allford

Matt Allford is a Developer Advocate at Octopus Deploy with over 15 years of experience across infrastructure, development, and operations. With a passion for education and content creation, he has authored courses at Pluralsight and presented at conferences across Australia and America.

Acknowledgements

We want to thank our families for supporting the work we do. It's not unusual to find us traveling to events, joining community calls and webinars at strange times of the day, or lying on the floor staring at the carpet while we think deeply about our work. Having folks around us who understand all these oddities is a great boost to what we do. So, thank you Fiona, Liam, Ewan, Pumpkin, Beckie, Lily, Luna, Carissa, Chloe, Sophie, and Isla. Thanks are also due to Octopus and the leaders who let three people disappear into a book about Rego and, astonishingly, waved us off with encouragement rather than concern. Writing about a policy language is a strange hill to die on, and we appreciate being allowed to die on it. We may have already done so halfway up.

To our fellow Octonauts, thank you for your knowledge and support. Thank you also for building something we're genuinely proud to stand behind. It's much easier to write convincingly about a product you believe in and use every day. We'd also like to thank Hedy Lamarr, Grace Hopper, and Ada Lovelace, three women who did rather more for computing than most of the people usually credited for it, and who did it while being underestimated at every turn. If you're looking for the real origin story of this industry, it runs through them, not through a garage in California.

And finally, thanks to Ug the caveman. Ug gave us all a great many things, chief among them ancestry, but historians rarely mention his other contribution: the binary system. As the story goes, Ug set three logs on the fire, announced proudly that he'd set eleven ablaze, and when challenged, calmly explained he was correct, because in binary, "11" is three. Nobody was impressed at the time. It took several thousand years and quite a lot of silicon before anyone understood what he was on about, and we still don't think he did either.

Octopus Deploy
Level 4, 199 Grey St
South Brisbane, QLD 4101, Australia

Email: sales@octopus.com

Phone: +1 512-823-0256

octopus.com