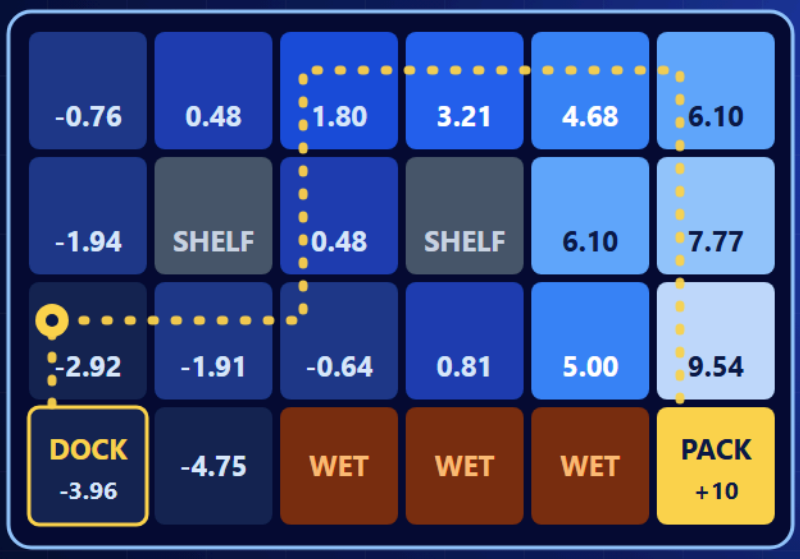





```
obs, reward, terminated, truncated, info = env.step(a)
target = reward + gamma * Q[obs].max() * (not terminated)
Q[s, a] += alpha * (target - Q[s, a])
# learn from every step, one move at a time
```

```
loss = -torch.min(ratio * adv, ratio.clamp(0.8, 1.2) * adv).mean()
```

A COMPLETE, PRACTICAL GUIDE

Reinforcement Learning from Zero

Build agents that learn by trial and error with Python, Gymnasium, and PyTorch

Bandits · MDPs · Dynamic Programming · Q-Learning · SARSA
DQN · REINFORCE · Actor-Critic · PPO · Reward Design · Safety

TABULAR · DEEP RL · POLICY GRADIENTS

THENEXBOOKS

Reinforcement Learning from Zero

Build Agents That Learn by Trial and Error with Python, Gymnasium, and PyTorch
From bandits and Q-learning to DQN and PPO — built step by step on one running example

First Edition

THENEXBOOKS

Reinforcement Learning from Zero
Build Agents That Learn by Trial and Error with Python, Gymnasium, and PyTorch

First Edition, 2026

Copyright © 2026 thenexbooks (thenexbooks.gumroad.com). All rights reserved.

No part of this publication may be reproduced, distributed, or transmitted in any form or by any means without the prior written permission of the publisher, except for brief quotations in reviews and certain other non-commercial uses permitted by copyright law. Code examples may be used freely in your own projects.

The information in this book is provided "as is" without warranty of any kind. While every precaution has been taken in its preparation, the publisher assumes no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein. Python libraries change over time; function names, defaults, and behavior can differ between versions. Training results depend on random seeds, library versions, and hardware, so your numbers may differ slightly from those printed here. Reinforcement learning policies must never replace certified safety systems in physical machines; always follow the safety standards and regulations that apply to your equipment.

Python is a trademark of the Python Software Foundation. PyTorch is a trademark of The Linux Foundation. Gymnasium is a project of the Farama Foundation. NumPy, Matplotlib, and other names are trademarks of their respective owners. Their use in this book is for identification only and does not imply endorsement. This book is independent and is not affiliated with, sponsored by, or endorsed by any of these organizations.

Thornbury Fulfillment, the fulfillment center used as the running example in this book, is fictional, as are its robot Tally, its staff, its warehouse layout, and all data. Any resemblance to real companies or people is coincidental.

Preface

Most programs do exactly what we tell them. Reinforcement learning is different: instead of writing the rules, we describe what we want — reach the packing station, never drive onto a wet floor, do not waste time — and let a program discover the rules by trying things and seeing what happens. The same idea has taught computers to play games at superhuman level, to control simulated robots, and to make large language models more helpful.

It is also a field with a reputation for being hard. Papers are dense with notation, tutorials jump straight to neural networks, and training runs fail silently. This book takes the opposite path. We start from zero, explain every idea in plain words before writing it as a formula, turn every formula into short Python code, and run that code on a problem small enough that we can check the answers by hand. Only when the foundations are solid do we move on to deep reinforcement learning with PyTorch.

Everything in the book is hands-on. You will build a custom Gymnasium environment, solve it exactly with dynamic programming, then train agents on it with Monte Carlo methods, SARSA, Q-learning, deep Q-networks, REINFORCE, actor-critic methods, and Proximal Policy Optimization written from scratch. Because we know the exact optimal answer, you will always be able to tell whether an algorithm really learned — a luxury most RL tutorials never give you.

Who This Book Is For

- **Python developers** who are curious about reinforcement learning and want to understand it properly, not just call a library.
- **Data scientists and machine learning engineers** who know supervised learning and want to add sequential decision-making to their toolkit.
- **Students** who find textbook notation intimidating and want runnable code next to every idea.
- **Engineers in robotics, operations, games, and AI products** who need to judge when reinforcement learning is the right tool, and how to use it safely when it is.

You should know basic Python: functions, classes, lists, dictionaries, and installing packages. You do not need prior machine learning experience. The mathematics is introduced gently, and Chapter 12 teaches the PyTorch you need.

What You Will Learn

- The core ideas of reinforcement learning: agents, environments, rewards, returns, policies, and value functions.
- Exploration versus exploitation with multi-armed bandits: ε-greedy, optimistic starts, UCB, and Thompson sampling.

- Markov decision processes and the Bellman equations, and how to solve them exactly with policy iteration and value iteration.
- Model-free learning: Monte Carlo methods, TD(0), SARSA, Q-learning, Expected SARSA, Double Q-learning, n-step methods, and Dyna-Q.
- Function approximation and deep reinforcement learning in PyTorch: DQN with replay buffers and target networks, Double DQN, and debugging techniques.
- Policy gradient methods: REINFORCE, baselines, actor-critic, and PPO from scratch, plus continuous actions with Gaussian policies.
- How to build and test custom Gymnasium environments, use wrappers and vector environments, and evaluate agents honestly across seeds.
- Reward design and shaping, exploration with sparse rewards, imitation learning, offline RL, and safe reinforcement learning.
- How reinforcement learning from human feedback shapes language models, and how to move a policy from simulation toward deployment.

How This Book Is Organized

Part I — Foundations (Chapters 1–5) introduces reinforcement learning, the toolkit, bandits, Markov decision processes, and the custom warehouse environment. **Part II — Planning and Tabular Learning** (Chapters 6–10) covers the Bellman equations, dynamic programming, Monte Carlo methods, and temporal-difference learning. **Part III — Function Approximation and Deep Q-Learning** (Chapters 11–15) moves from tables to features and neural networks, builds DQN, and shows how to evaluate agents. **Part IV — Policy Gradient Methods** (Chapters 16–20) covers REINFORCE, actor-critic, PPO, continuous actions, and practical training. **Part V — Designing RL Systems** (Chapters 21–24) covers reward design, exploration, model-based methods, and learning from logged data. **Part VI — RL in the Real World** (Chapters 25–28) covers RL for language models, safety, deployment, and a complete case study. The appendices provide a cheat sheet, templates and reference tables, a production checklist, and a glossary.

Each chapter ends with exercises and a **Key Takeaways** box for quick review.

The Running Example: Thornbury Fulfillment

Throughout the book we work with **Thornbury Fulfillment**, a fictional e-commerce fulfillment center that is piloting a small autonomous mobile robot called **Tally**. Tally carries order totes from the loading dock to the packing station through Aisle Block C, a small area with shelves, a top aisle, a middle aisle, and a lane next to the dock that is often wet after cleaning. Every move costs time, bumping a shelf costs more, and driving onto the wet floor is a safety incident. The floor is also slightly slick, so Tally does not always move where it intends to.

We model Aisle Block C as a 4 × 6 grid, small enough to solve exactly and to print every value on one page, yet rich enough to show the real lessons of reinforcement learning: a slippery floor changes the best route, some algorithms learn risky shortcuts while others learn caution, a badly designed reward makes the robot circle forever, and learning from logged data can go wrong when the logs never show the rare accident. Marisol Ferrante, the operations director, sets the goals; Kwame Asante, the robotics and ML engineer, writes the code; and Imogen Beale, the safety and compliance lead, makes sure no shortcut puts people at risk.

Versions and Conventions

The code uses **Python 3.12**, **NumPy**, **Gymnasium 1.x** (the maintained successor of OpenAI Gym), and the CPU build of a recent version of **PyTorch**. All training runs are deliberately small: every example finishes in a few minutes or less on an ordinary laptop CPU, with fixed random seeds so you can reproduce the printed results. Results can still differ slightly between library versions and machines. Where a full-size run would behave differently, the text says so. A GPU is never required.

> **TIP**
> A recommended practice or a technique that makes your work easier.

> **NOTE**
> Background information, version differences, or a useful aside.

> **WARNING**
> A common mistake or something that can cause wrong results, wasted training time, or safety problems.

Code appears in `monospace`; long lines are wrapped to fit the page. Formulas are written in plain text, such as `G_t = R_t+1 + γ G_t+1`, and every symbol is explained when it first appears and collected in Appendix A. Grids are printed with the dock at the bottom left and the packing station at the bottom right.

Contents

Preface

PART I — FOUNDATIONS

1 What Reinforcement Learning Is: Agents, Rewards, and Learning by Trial and Error

2 Your RL Toolkit: Python, NumPy, Gymnasium, and PyTorch

3 Multi-Armed Bandits: Exploration versus Exploitation

4 Markov Decision Processes: States, Actions, Rewards, and Returns

5 Building the Thornbury Warehouse: A Custom Gymnasium Environment

PART II — PLANNING AND TABULAR LEARNING

6 Bellman Equations: Values, Q-Values, and Optimality

7 Dynamic Programming: Policy Evaluation, Policy Iteration, and Value Iteration

8 Monte Carlo Methods: Learning from Complete Episodes

9 Temporal-Difference Learning: TD(0), SARSA, and Q-Learning

10 Beyond One Step: Expected SARSA, Double Q-Learning, and n-Step Returns

PART III — FUNCTION APPROXIMATION AND DEEP Q-LEARNING

11 Function Approximation: From Tables to Features

12 PyTorch for Reinforcement Learning: Networks, Losses, and Optimizers

13 Deep Q-Networks: Replay Buffers and Target Networks

14 Making DQN Work: Double DQN, Dueling Networks, Prioritized Replay, and Debugging

15 Evaluating Agents Honestly: Seeds, Learning Curves, and Fair Comparisons

PART IV — POLICY GRADIENT METHODS

16 Policy Gradients and REINFORCE

17 Baselines and Actor-Critic Methods

18 Proximal Policy Optimization from Scratch

19 Continuous Actions: Gaussian Policies, DDPG, TD3, and SAC

20 Practical Training: Vectorized Environments, Normalization, Wrappers, and Hyperparameters

PART V — DESIGNING RL SYSTEMS

21 Reward Design and Shaping: Getting the Behavior You Asked For

22 Exploration, Sparse Rewards, and Curriculum Learning

23 Model-Based RL and Planning: Dyna-Q and Search

24 Learning from Logged Data: Imitation and Offline RL

PART VI — RL IN THE REAL WORLD

25 RL for Language Models: RLHF, Reward Models, and Preference Optimization

26 Safe Reinforcement Learning: Constraints, Shielding, and Safety Testing

27 From Simulation to Deployment: Sim-to-Real, Libraries, Serving, and Monitoring

28 Case Study: Taking Tally from Gridworld to a Pilot Rollout

APPENDICES

A Reinforcement Learning Cheat Sheet

B Templates and Reference

C Production Checklist

D Glossary

PART I

Foundations

What reinforcement learning is and when to use it, the Python tools we use throughout the book, the exploration-versus-exploitation problem in its simplest form, the mathematics of Markov decision processes in plain language, and the custom Gymnasium environment for Thornbury's warehouse robot.

1 · What Reinforcement Learning Is

2 · Your RL Toolkit

3 · Multi-Armed Bandits

4 · Markov Decision Processes

5 · Building the Thornbury Warehouse

CHAPTER 1 · PART I — FOUNDATIONS

What Reinforcement Learning Is: Agents, Rewards, and Learning by Trial and Error

Reinforcement learning is the branch of machine learning in which a program learns what to do by trying things and receiving rewards. In this chapter we meet the core vocabulary, the warehouse robot that will carry us through the whole book, and the first hand-played episode whose numbers we will meet again and again.

Learning from Consequences

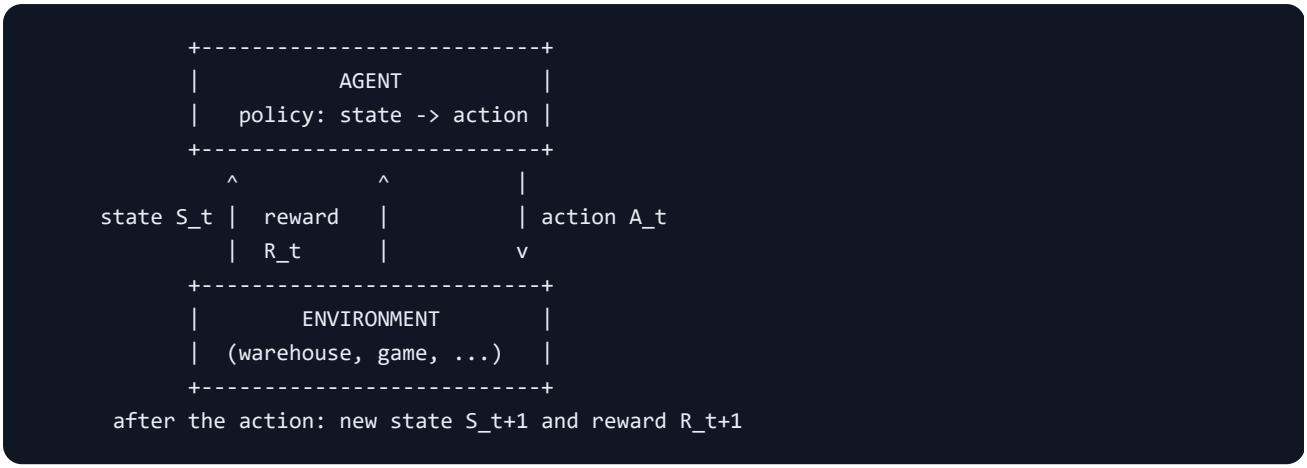
Think about how a child learns to ride a bicycle. Nobody hands the child a spreadsheet with the correct handlebar angle for every possible lean and speed. Instead, the child tries, wobbles, falls, tries a little differently, and slowly discovers which movements keep the bicycle upright. The feedback is not a list of correct answers. It is a signal that arrives after the fact: staying up feels good, falling down hurts. Over many attempts, the child turns that thin stream of feedback into a skill.

Reinforcement learning (RL) is the study of programs that learn in the same way. A program, which we call the **agent**, interacts with a world, which we call the **environment**. At each moment the agent sees some information about the environment, chooses an action, and receives a number called a **reward** that says how good the immediate result was. The agent's job is not to maximize the next reward but the total reward it collects over time. That last point is what makes the problem interesting: an action that looks good now can lead to trouble later, and an action that costs something now can open the door to a large payoff.

Three ideas separate reinforcement learning from ordinary programming. First, nobody writes down the rules of good behavior; we only describe what we want through rewards. Second, the agent learns from its own experience, so the data it sees depends on the choices it makes. Third, consequences are delayed: the reward for a good decision may arrive many steps after the decision itself, and the agent has to work out which earlier choices deserve the credit. RL researchers call this last challenge the **credit assignment problem**, and much of the machinery in this book exists to solve it.

The Agent–Environment Loop

Every reinforcement learning problem, from a board game to a robot, fits into the same loop. The agent observes the current **state** of the environment, picks an **action**, and the environment answers with a reward and a new state. Then the loop repeats. We write the state at time step `t` as `S_t`, the action as `A_t`, and the reward that follows it as `R_t+1`. The reward carries the index `t+1` because it arrives together with the next state, after the action has been taken.



Running the loop produces a sequence of states, actions, and rewards called a **trajectory**:

```
S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, ...
```

Many tasks have a natural ending. A game is won or lost, a delivery arrives or fails. Each run from a starting state to an ending is called an **episode**. When an episode ends, the environment is reset and a new episode begins. The sum of the rewards collected during an episode is called the **return**; Chapter 4 refines this idea with discounting, which makes rewards that arrive sooner count a little more than rewards that arrive later.

The agent's way of choosing actions is called its **policy**, written with the Greek letter `π` (pi). A policy can be as simple as a lookup table ("in this state, go up") or as complicated as a neural network with millions of parameters. It can also be random: "in this state, go up with probability 0.9 and right with probability 0.1." Learning, in RL, means improving the policy so that it collects more reward on average. Everything else, including value functions, Q-tables, replay buffers, and neural networks, is a tool for improving the policy.

The core vocabulary

Term	Plain meaning	Symbol
Agent	The learner and decision maker	—
Environment	Everything the agent interacts with	—
State	The situation the agent is in	`S_t`, `s`
Action	A choice the agent can make	`A_t`, `a`
Reward	A number that scores one step	`R_t+1`
Policy	The rule for choosing actions	`π(a|s)`
Episode	One run from start to end	—
Return	Total (discounted) reward from a time step	`G_t`

We use these symbols the same way in every chapter, following the conventions of the standard textbook in the field, Sutton and Barto's *Reinforcement Learning: An Introduction* (2nd edition, 2018). If you read that book later, the notation will feel familiar.

> **NOTE**
>
> In this book, the word *agent* always means a reinforcement learning agent: a program that chooses actions to maximize reward in an environment. Today the same word is also used for software built around a large language model that calls tools and APIs. Those "LLM agents" are a different topic. They may use RL during training (Chapter 25 explains how), but an LLM agent and an RL agent are not the same thing.

How RL Differs from Other Kinds of Machine Learning

If you have met machine learning before, you have probably seen **supervised learning**. There, we collect examples with correct answers, called labels, and train a model to reproduce them: photos labeled "cat" or "dog", emails labeled "spam" or "not spam", house descriptions paired with sale prices. The model is told the right answer for every example. In **unsupervised learning** there are no labels at all; the goal is to find structure, such as clusters of similar customers.

Reinforcement learning sits apart from both. There is a signal, so it is not unsupervised, but the signal is not the right answer. A reward of -1 tells Tally's control program that a move cost time; it does not say which move would have been better. The agent has to discover that by trying alternatives. The differences are worth spelling out, because they explain why RL algorithms look the way they do.

Question	Supervised learning	Reinforcement learning
What feedback arrives?	The correct answer	A score for what you did
Where does data come from?	A fixed dataset	The agent's own actions
When does feedback arrive?	Immediately, per example	Often many steps later
Do decisions affect later inputs?	No	Yes
Main extra challenge	Generalization	Exploration and credit assignment

The second row deserves attention. Because an RL agent collects its own data, a poor early policy produces poor data, and it can get stuck: if the agent never tries a particular action, it never learns that the action is good. The tension between using what you already know (**exploitation**) and trying something new to learn more (**exploration**) is the oldest problem in the field. Chapter 3 studies it in its purest form, with no states at all, before we add the complications of sequential decisions.

Where Reinforcement Learning Is Used

RL has a few famous successes. A deep Q-network learned to play many classic Atari video games directly from screen pixels (Mnih et al., 2013 and 2015). AlphaGo and its successor AlphaZero combined reinforcement learning with tree search to beat top human players at Go, and AlphaZero also mastered chess and shogi (Silver et al., 2016 and 2017–2018). Reinforcement learning from human feedback, or RLHF (Christiano et al., 2017; Ouyang et al., 2022), became a standard step in making large language models follow instructions more helpfully. We meet each of these ideas later in the book, at a size you can run on a laptop.

Beyond headlines, RL appears wherever a system makes a sequence of decisions and gets measurable feedback:

- **Games and simulations**, where an agent can play millions of rounds safely and cheaply.
- **Robotics**, usually trained in simulation first and then transferred to hardware with care.
- **Recommendation and personalization**, often in the simpler "bandit" form covered in Chapter 3, where the system picks which item to show and learns from clicks.
- **Operations**: routing, scheduling, inventory, and resource allocation, where each decision changes the situation for the next one.
- **Fine-tuning language models** with human or AI preferences, covered conceptually in Chapter 25.

When RL is a bad fit

An honest introduction should also say when *not* to use reinforcement learning. RL is a powerful but expensive tool, and many problems that sound like RL problems are better solved another way.

- **You already have labeled answers.** If experts can tell you the right action for each situation and you have many such examples, supervised learning is simpler, faster, and easier to debug. Chapter 24 shows this idea, called behavior cloning, as a strong baseline.
- **Mistakes are costly and there is no simulator.** RL learns by making mistakes. If every mistake damages equipment, hurts someone, or loses real money, and you cannot build a realistic simulator to make those mistakes in, trial and error is not acceptable.
- **The problem has a known exact solution.** Shortest paths on a known map, linear programs, and many scheduling problems have classic algorithms that are proven correct. Chapter 7 shows that when we know the rules of our small warehouse exactly, we can compute the best policy without any learning at all.
- **You cannot write down what "good" means.** RL optimizes the reward you give it, not the goal you had in mind. If you cannot express the goal as a reward, the agent will optimize something else. Chapter 21 shows a robot that learns to circle forever because of one badly chosen bonus.

> **WARNING**
>
> Reinforcement learning has a reputation for being finicky, and the reputation is earned. Training can fail silently: the code runs, the numbers change, and the agent learns nothing useful. Results can depend on random seeds, small hyperparameter changes, and subtle bugs in the environment. Throughout this book we use a problem small enough that we can compute the exact best answer, so you will always be able to check whether an algorithm really worked. Keep that habit when you move to bigger problems: always compare against a simple baseline you trust.

Meet Thornbury Fulfillment

Abstract definitions only go so far, so the whole book follows one running example. **Thornbury Fulfillment** is a fictional mid-sized e-commerce fulfillment center. Orders arrive, items are picked from shelves, put into plastic order totes, and carried to a packing station where they are boxed and shipped. Thornbury is piloting a small autonomous mobile robot (AMR) named **Tally**. Tally carries one order tote at a time from the loading dock to the packing station through a section of the building called **Aisle Block C**.

Three people care about how Tally behaves:

- **Marisol Ferrante**, the operations director, owns the business goals: deliveries should be fast, and there should be zero incidents.
- **Kwame Asante**, the robotics and ML engineer, writes the code. The code you type in this book is, in the story, Kwame's code.
- **Imogen Beale**, the safety and compliance lead, reviews how rewards are designed, how the wet-floor rule is enforced, and what must be true before Tally is allowed to run in a new area.

Aisle Block C is small. It has a top aisle, two shelf units with a short middle aisle between them, a lower aisle in front of the shelves (row 2), and a bottom lane that runs past the loading dock. The bottom lane is the trouble spot: cleaning crews mop it, and parts of it are often wet. Driving onto a wet floor is a safety incident, because a loaded robot can skid into people or racks. To make the problem manageable, Kwame models Aisle Block C as a grid of 4 rows and 6 columns:

```
col:  0 1 2 3 4 5
r0    . . . . . .
r1    . # . # . .
r2    . . . . . .
r3    S . W W W G

.  floor        #  shelf (impassable)
S  loading dock (start)
G  packing station (goal)
W  wet floor (spill)
```

Tally starts at the loading dock `S` in the bottom-left corner, row 3, column 0. The packing station `G` is in the bottom-right corner, row 3, column 5. The three cells between them on the bottom row are wet. Two shelf units, marked `#`, block cells (1, 1) and (1, 3). In each step Tally can try to move **up**, **right**, **down**, or **left**.

Each move has a consequence that Kwame turns into a reward, after agreeing the numbers with Marisol and Imogen:

Event	Reward	What happens
Ordinary move	-1	Tally moves one cell
Bump into a shelf or the outer wall	-2	Tally stays where it is
Reach the packing station	+10	The episode ends (success)
Drive onto the wet floor	-50	The episode ends (safety incident)

The -1 per move expresses Marisol's wish for speed: every step costs time and battery, so shorter routes are better. The -2 for bumping says that bumping is worse than moving, because it wastes time and wears the robot. The +10 rewards a completed delivery. The -50 for the wet floor is Imogen's number, and it is deliberately large: a single incident should outweigh any time saved by a shortcut. When Tally reaches the goal or the wet floor, that special reward replaces the ordinary -1, so entering the packing station gives exactly +10.

There is one more detail, and it turns out to matter a great deal. The floor of Aisle Block C is slightly slick. When Tally tries to move in some direction, it succeeds 90% of the time. The other 10% of the time it slips sideways: 5% of the time to one side of the intended direction and 5% to the other. Tally might aim to go right and drift down instead. Episodes are also capped at 100 steps, so a robot that wanders aimlessly is eventually stopped. Chapter 4 describes all of this precisely, and Chapter 5 turns it into a real Gymnasium environment called `ThornburyGrid-v0`.

> **WARNING**
>
> Tally is a teaching abstraction, not a design for a real robot. Real mobile robots in workplaces rely on certified safety controllers, speed limits, obstacle sensors, and emergency stops that work regardless of what any learned policy decides. A reinforcement learning policy can choose routes; it never replaces those safety layers. We return to this point in Chapters 26, 27, and 28.

Playing an Episode by Hand

Before writing any learning algorithm, let us play the game ourselves. Imagine the floor is dry and nothing ever slips, so every move goes exactly where we intend. Which route should Tally take?

The most direct idea is to go along the bottom row: right, right, right, right, right. But the second move would enter cell (3, 2), which is wet. That ends the episode with a -50 incident. So instead Tally can go up one cell to row 2, drive right along row 2 beside the wet lane, and drop down into the packing station at the end. We call this the **wet-lane route**, because it hugs the wet lane. It takes 7 moves. A more cautious route crosses row 2 only as far as column 2, climbs the middle aisle between the shelves to the top aisle, travels along the top aisle, far from the spill, then comes down the right-hand column. We call it the **safe route**; it takes 11 moves.

```
Wet-lane route (7 moves)          Safe route (11 moves)
. . . . . .                       . . * * * *
. # . # . .                       . # * # . *
* * * * * *                       * * * . . *
S . W W W G                       S . W W W G
```

We can score both routes with a few lines of plain Python. The listing below does not use any library; it simply encodes the grid, the four moves, and the reward table, then plays a fixed list of actions. Create a folder for the book's code (Chapter 2 sets up the full project), save the file there, and run it.

hand_played.py

```
"""Chapter 1: play two routes through Aisle Block C by hand (no slipping)."""
MAP = ("......",
       ".#.#..",
       "......",
       "S.WWWG")
MOVES = {"up": (-1, 0), "right": (0, 1), "down": (1, 0), "left": (0, -1)}


def step(pos, action):
    """Return (new_pos, reward, done) for one move on a dry, non-slippery floor."""
    r, c = pos[0] + MOVES[action][0], pos[1] + MOVES[action][1]
    if not (0 <= r < 4 and 0 <= c < 6) or MAP[r][c] == "#":
        return pos, -2, False          # bump: Tally stays where it is
    if MAP[r][c] == "G":
        return (r, c), 10, True        # packing station reached
    if MAP[r][c] == "W":
        return (r, c), -50, True       # wet floor: safety incident
    return (r, c), -1, False           # ordinary move


def play(name, actions):
    pos, total = (3, 0), 0
    print(f"--- {name} ---")
    for t, action in enumerate(actions, start=1):
        pos, reward, done = step(pos, action)
        total += reward
        print(f"t={t:2d} {action:5s} -> {pos} reward {reward:+d}  total {total:+d}")
        if done:
            break
    print(f"return {total:+d}\n")
```

```
play("wet-lane route", ["up", "right", "right", "right", "right", "right", "down"])
play("safe route", ["up", "right", "right", "up", "up", "right", "right", "right",
                    "down", "down", "down"])
play("careless route", ["right", "right"])
```

Run it with `py -3.12 hand_played.py` on Windows or `python3 hand_played.py` on macOS and Linux. You should see exactly this:

Output

```
--- wet-lane route ---
t= 1 up    -> (2, 0) reward -1  total -1
t= 2 right -> (2, 1) reward -1  total -2
t= 3 right -> (2, 2) reward -1  total -3
t= 4 right -> (2, 3) reward -1  total -4
t= 5 right -> (2, 4) reward -1  total -5
t= 6 right -> (2, 5) reward -1  total -6
t= 7 down  -> (3, 5) reward +10  total +4
return +4

--- safe route ---
t= 1 up    -> (2, 0) reward -1  total -1
t= 2 right -> (2, 1) reward -1  total -2
...
t=10 down  -> (2, 5) reward -1  total -10
t=11 down  -> (3, 5) reward +10  total +0
return +0

--- careless route ---
t= 1 right -> (3, 1) reward -1  total -1
t= 2 right -> (3, 2) reward -50  total -51
return -51
```

The numbers tell a clear story. On a dry, non-slippery floor, the wet-lane route earns a return of +4: six ordinary moves at -1 each, then +10 for the delivery. The safe route earns 0, because it spends four extra moves. The careless route that drives straight along the bottom row hits the wet floor on its second move and earns -51. If the floor never slips, the wet-lane route is clearly the best of the three, and in fact no route can do better, because 7 moves is the shortest path that avoids the spill.

The slippery twist

Now remember the slick floor. On the wet-lane route, Tally drives right five times along row 2. Each of those moves has a 5% chance of slipping downward, and three of them start from cells directly above the spill, at columns 2, 3, and 4. A slip on any of those three moves puts Tally on the wet floor with a -50 incident. The chance that all three risky moves succeed is about 0.95 × 0.95 × 0.95, or roughly 86%, so roughly one delivery in seven would end in an incident. Three moves of extra caution start to look cheap.

Which route is actually best once slipping is included? Answering that by hand is hard: we would need to consider every possible slip at every step, including slips that push Tally somewhere unexpected and force it to recover. Here is the first thing reinforcement learning will do for us. In Chapter 7 we compute the exact answer for both floors and find that the slippery floor really does change the best route: with no slipping the optimal policy takes the 7-move wet-lane route, and with a 10% slip it takes the 11-move safe route through the top aisle. In Chapters 8 to 10 we watch learning algorithms discover an equally long safe route on their own, purely from experience, and in Chapter 9 we find that two famous algorithms disagree about it in an instructive way.

> **TIP**
>
> Playing a few episodes by hand before writing any learning code is one of the most useful habits in RL. It forces you to check that the rewards say what you mean, it gives you a baseline score to beat, and it often exposes bugs in the environment, such as a wall that is not where you think it is.

What the Agent Needs to Learn

When we played by hand, we used knowledge Tally does not have. We could see the whole map, we knew the reward table, and we could reason about routes before taking a single step. A learning agent starts with none of this. It only sees its current cell, takes an action, and receives a number. From that stream of numbers it must work out that moving onto certain cells is disastrous, that the packing station is worth reaching, and that some routes are safer than others.

Learning methods differ in what they build from experience. Some learn a **value** for each state: roughly, "how much total reward can I expect from here?" A cell next to the packing station should have a high value; a cell next to the spill, a lower one. Others learn a value for each state and action pair, called a **Q-value**: "how much total reward can I expect if I take this action here and behave well afterward?" Once you know the Q-values, choosing an action is easy: pick the one with the highest Q-value. Still other methods skip values and adjust the policy directly, nudging up the probability of actions that led to good outcomes. All three families appear in this book, and each is useful in different situations.

A useful mental picture is a map with a number written in each cell. After learning, Tally's value map might show high numbers near the packing station and along the top aisle, and low numbers near the wet floor. Following the numbers uphill leads to the goal. Chapter 6 shows the real value map for Aisle Block C, computed exactly, and Chapter 7 shows how to compute it.

Exploration in the warehouse

A learning agent faces the exploration problem from the very first episode. Suppose Tally's first few trips happen to go up and around the top aisle and reach the packing station. A naive agent might decide this is the best route and never try anything else. It would never discover that, on a dry day, the route along row 2 is shorter. The opposite failure is just as real: an agent that explores too much keeps

trying moves it already knows are bad, such as driving onto the wet floor. In simulation that costs nothing but time. On a real warehouse floor it would be unacceptable, which is one reason real robot learning happens in simulators first, and one reason Imogen insists on safety layers that do not depend on learning at all.

> **NOTE**
>
> The grid is tiny on purpose. With 24 cells and 4 actions, every value fits on one printed page, every algorithm runs in seconds or a couple of minutes on a laptop CPU, and we can compute the exact optimal behavior to check our learners against. Real problems are far larger, but the concepts transfer directly. Chapter 11 discusses what changes when the table no longer fits, and Chapters 12 to 18 replace the table with neural networks.

The Road Ahead

The book is organized in six parts, each building on the previous one.

1. **Part I — Foundations (Chapters 1–5).** The vocabulary, the toolkit (Python, NumPy, Gymnasium, PyTorch), the exploration problem in its simplest form (multi-armed bandits), the mathematical framework of Markov decision processes, and the Thornbury warehouse as a real Gymnasium environment.
2. **Part II — Planning and Tabular Learning (Chapters 6–10).** Bellman equations, exact solutions with dynamic programming, and learning from experience with Monte Carlo methods, SARSA, Q-learning, and their relatives. Here we discover how a slippery floor changes the best route.
3. **Part III — Function Approximation and Deep Q-Learning (Chapters 11–15).** Moving from tables to features and neural networks, building a deep Q-network step by step, making it work reliably, and evaluating agents honestly across random seeds.
4. **Part IV — Policy Gradient Methods (Chapters 16–20).** Learning the policy directly with REINFORCE, actor-critic methods, and PPO, then extending to continuous actions and practical training techniques.
5. **Part V — Designing RL Systems (Chapters 21–24).** Reward design and its pitfalls, exploration with sparse rewards, model-based planning, and learning from logged data.
6. **Part VI — RL in the Real World (Chapters 25–28).** RL for language models, safe RL, moving from simulation to deployment, and a final case study that takes Tally all the way to a pilot rollout plan.

Every chapter has runnable code with its real output. Many chapters end by comparing a learned agent with the exact optimal policy from Chapter 7, which gives us an answer key few RL tutorials can offer. When a result could depend on your machine or library versions, we say so; small differences in the last digits are normal.

How to read this book

Read the chapters in order, at least through Part II, because each one reuses the code of the previous ones. Chapters 4 and 5 define the exact environment that every later result depends on, so read them carefully. Type the code when you can: typing makes you notice where a reward is added or when an episode ends, which is exactly where RL bugs hide. Each formula is first described in words, then written in plain text, then turned into Python; if a formula looks intimidating, read the words and the code first.

Being Honest about RL

We close this chapter with a few honest expectations. Reinforcement learning is exciting, but it is also a field where it is easy to fool yourself. A learning curve that goes up can hide an agent that learned a risky shortcut. A result from one random seed can look impressive and fail to reproduce on the next. An agent can score well on the reward you wrote while doing something you never intended. This book shows each of these traps happening on Tally's small grid, where we can see exactly why.

The habits that protect you are simple to state and worth practicing from the start: keep an exact or trusted baseline when you can, evaluate with fixed seeds over many episodes rather than eyeballing one run, look at what the agent actually does and not just at its score, and treat the reward function as a specification that needs review just like code. Marisol will ask how often Tally reaches the packing station. Imogen will ask how often it ends up on the wet floor. Learning to answer both questions with real numbers is as important as learning any algorithm.

Try It Yourself

1. Run `hand_played.py` and confirm the three returns: +4, 0, and -51.
2. Add a fourth route that bumps into the shelf at (1, 1) on the way, for example by going up twice from the dock, then right. Predict its return before running it, then check your prediction.
3. Find another route to the packing station that avoids the wet floor and takes exactly 9 moves. Is there more than one? What return does it earn?
4. Change the wet-floor reward from -50 to -5 in `step()`. Does any route that drives over the wet floor now look attractive? (Remember that entering a wet cell ends the episode.) What does this tell you about why Imogen chose a large penalty?
5. Using the 5% sideways slip described in this chapter, estimate by hand the probability that the wet-lane route slips onto the wet floor at least once. Which of its moves are risky, and which are not?
6. Write down, in one or two sentences each, how a supervised learning approach to Tally's routing would work and what data it would need. Why might Thornbury not have that data?