Foundations of Reinforcement Learning with Applications in Finance

Ashwin Rao, Tikhon Jelvis

Programming and Design

Programming is creative work with few constraints: imagine something and you can probably build it — in *many* different ways. Liberating and gratifying, but also challenging. Just like starting a novel from a blank page or a painting from a blank canvas, a new program is so open that it's a bit intimidating. Where do you start? What will the system look like? How will you get it *right*? How do you split your problem up? How do you prevent your code from evolving into a complete mess?

There's no easy answer. Programming is inherently iterative — we rarely get the right design at first, but we can always edit code and refactor over time. But iteration itself is not enough; just like a painter needs technique and composition, a programmer needs patterns and design.

Existing teaching resources tend to deemphasize programming techniques and design. Theory-heavy and algorithms-heavy books show models and algorithms as self-contained procedures written in pseudocode, without the broader context (and corresponding design considerations) of a real codebase. Newer AI/ML materials sometimes take a different tack and provide real code examples using industry-strength frameworks, but rarely touch on software design questions.

In this book, we take a third approach. Starting *from scratch*, we build a Python framework that reflects the key ideas and algorithms we cover in this book. The abstractions we define map to the key concepts we introduce; how we structure the code maps to the relationships between those concepts.

Unlike the pseudocode approach, we do not implement algorithms in a vacuum; rather, each algorithm builds on abstractions introduced earlier in the book. By starting from scratch (rather than using an existing ML framework) we keep the code reasonably simple, without needing to worry about specific examples going out of date. We can focus on the concepts important to this book while teaching programming and design *in situ*, demonstrating an intentional approach to code design.

0.1 Code Design

How can we take a complex domain like reinforcement learning and turn it into code that is easy to understand, debug and extend? How can we split this problem into manageable pieces? How do those pieces interact?

There is no simple single answer to these questions. No two programming challenges are identical and the same challenge has many reasonable solutions. A solid design will not be completely clear up-front; it helps to have a clear direction in mind, but expect to revisit specific decisions over time. That's exactly what we did with the code for this book: we had a vision for a Python Reinforcement Learning framework that matched the topics we present, but as we wrote more and more of the book, we revised the framework code as we came up with better ideas or found new requirements our previous design did not cover.

We might have no easy answers, but we do have patterns and principles that — in our experience — consistently produce quality code. Taken together, these ideas form a phi-

losophy of code design oriented around defining and combining **abstractions** that reflect how we think about our domain. Since code itself can point to specific design ideas and capabilities, there's a feedback loop: expanding the programming abstractions we've designed can help us find new algorithms and functionality, improving our understanding of the domain.

Just what *is* an abstraction? An appropriately abstract question! An abstraction is a "compound idea": a single concept that combines multiple separate ideas into one. We can combine ideas along two axes:

- We can *compose* different concepts together, thinking about how they behave as one unit. A car engine has thousands of parts that interact in complex ways, but we can think about it as a single object for most purposes.
- We can *unify* different concepts by identifying how they are similar. Different breeds of dogs might look totally different, but we can think of all of them as dogs.

The human mind can only handle so many distinct ideas at a time — we have an inherently limited working memory. A rather simplified model is that we only have a handful of "slots" in working memory and we simply can't track more independent thoughts at the same time. The way we overcome this limitation is by coming up with *new* ideas (new *abstractions*) that combine multiple concepts into one.

We want to organize code around abstractions for the same reason that we use abstractions to understand more complex ideas. How do you understand code? Do you run the program in your head? That's a natural starting point and it works for simple programs but it quickly becomes difficult and then impossible. A computer doesn't have working-memory limitations and can run *billions* of instructions a second that we can't possibly keep up with. The computer doesn't need structure or abstraction in the code it runs, but we need it to have any hope of writing or understanding anything beyond the simplest of programs. Abstractions in our code group information and logic so that we can think about rich concepts rather than tracking every single bit of information and every single instruction separately.

The details may differ, but designing code around abstractions that correspond to a solid mental model of the domain works well in any area and with any programming language. It might take some extra up-front thought but, done well, this style of design pays dividends. Our goal is to write code that makes life easier *for ourselves*; this helps for everything from "one-off" experimental code through software engineering efforts with large teams.

0.2 Environment Setup

You can follow along with all of the examples in this book by getting a copy of the RL framework from GitHub[1] and setting up a dedicated Python 3 environment for the code.

The Python code depends on several Python libraries. Once you have a copy of the code repository, you can create an environment with the right libraries by running a few shell commands.

First, move to the directory with the codebase:

```
cd rl-book
```

[1] If you are not familiar with Git and GitHub, look through GitHub's Getting Started documentation.

Then, create and activate a Python virtual environment[2]:

```
python3 -m venv .venv
source .venv/bin/activate
```

You only need to create the environment once, but you will need to activate it every time that you want to work on the code from a new shell. Once the environment is activated, you can install the right versions of each Python dependency:

```
pip install -r requirements.txt
```

To access the framework itself, you need to install it in editable mode (`-e`):

```
pip install -e .
```

Once the environment is set up, you can confirm that it works by running the frameworks automated tests:

```
python -m unittest discover
```

If everything installed correctly, you should see an "OK" message on the last line of the output after running this command.

0.3 Classes and Interfaces

What does designing clean abstractions actually entail? There are always two parts to answering this question:

1. Understanding the domain concept that you are modeling.
2. Figuring out how to express that concept with features and patterns provided by your programming language.

Let's jump into an extended example to see exactly what this means. One of the key building blocks for Reinforcement Learning — all of statistics and machine learning, really — is Probability. How are we going to handle uncertainty and randomness in our code?

One approach would be to keep Probability implicit. Whenever we have a random variable, we could call a function and get a random result (technically referred to as a *sample*). If we were writing a Monopoly game with two six-sided dice, we would define it like this:

```
from random import randint

def six_sided()
    return randint(1, 6)

def roll_dice():
    return six_sided() + six_sided()
```

[2] A Python "virtual environment" is a way to manage Python dependencies on a per-project basis. Having a different environment for different Python projects lets each project have its own version of Python libraries, which avoids problems when one project needs an older version of a library and another project needs a newer version.

This works, but it's pretty limited. We can't do anything except get one outcome at a time. More importantly, this only captures a slice of how we *think* about Probability: there's *randomness* but we never even mentioned probability distributions (referred to as simply *distribution*s for the rest of this chapter). We have outcomes and we have a function we can call repeatedly, but there's no way to tell that function apart from a function that has nothing to do with Probability but just happens to return an integer.

How can we write code to get the expected value of a distribution? If we have a parametric distribution (eg: a distribution like Poisson or Gaussian, characterized by parameters), can we get the parameters out if we need them?

Since distributions are implicit in the code, the *intentions* of the code aren't clear and it is hard to write code that generalizes over distributions. Distributions are absolutely crucial for machine learning, so this is not a great starting point.

0.3.1 A Distribution Interface

To address these problems, let's define an abstraction for probability distributions.

How do we represent a distribution in code? What can we *do* with distributions? That depends on exactly what kind of distribution we're working with. If we know something about the structure of a distribution — perhaps it's a Poisson distribution where $\lambda = 5$, perhaps it's an empirical distribution with set probabilities for each outcome — we could do quite a bit: produce an exact Probability Distribution Function (PDF) or Cumulative Distribution Function (CDF), calculate expectations and do various operations efficiently. But that isn't the case for all the distributions we work with! What if the distribution comes from a complicated simulation? At the extreme, we might not be able to do anything except draw samples from the distribution.

Sampling is the least common denominator. We can sample distributions where we don't know enough to do anything else, and we can sample distributions where we know the exact form and parameters. Any abstraction we start with for a probability distribution needs to cover sampling, and any abstraction that requires more than just sampling will not let us handle all the distributions we care about.

In Python, we can express this idea with a class:

```
from abc import ABC, abstractmethod

class Distribution(ABC):
    @abstractmethod
    def sample(self):
        pass
```

This class defines an **interface**: a definition of what we require for something to qualify as a distribution. Any kind of distribution we implement in the future will be able to, at minimum, generate samples; when we write functions that sample distributions, they can require their inputs to inherit from `Distribution`.

The class itself does not actually implement `sample`. `Distribution` captures the *abstract concept* of distributions that we can sample, but we would need to specify a specific distribution to actually sample anything. To reflect this in Python, we've made `Distribution` an **abstract base class** (ABC), with `sample` as an *abstract* method—a method without an implementation. Abstract classes and abstract methods are features that Python provides to help us define interfaces for abstractions. We can define the `Distribution` class to structure the rest of our probability distribution code before we define any specific distributions.

0.3.2 A Concrete Distribution

Now that we have an interface, what do we do with it? An interface can be approached from two sides:

- Something that **requires** the interface. This will be code that uses operations specified in the interface and work with *any* value that satisfies those requirements.
- Something that **provides** the interface. This will be some value that supports the operations specified in the interface.

If we have some code that requires an interface and some other code that satisfies the interface, we know that we can put the two together and get something that works — even if the two sides were written without any knowledge or reference to each other. The interface manages how the two sides interact.

To use our `Distribution` class, we can start by providing a **concrete class**[3] that implements the interface. Let's say that we wanted to model dice — perhaps for a game of D&D or Monopoly. We could do this by defining a `Die` class that represents an n-sided die and inherits from `Distribution`:

```
import random

class Die(Distribution):
    def __init__(self, sides):
        self.sides = sides

    def sample(self):
        return random.randint(1, self.sides)

six_sided = Die(6)

def roll_dice():
    return six_sided.sample() + six_sided.sample()
```

This version of `roll_dice` has exactly the same behavior as `roll_dice` in the previous section, but it took a bunch of extra code to get there. What was the point?

The key difference is that we now have a value that represents the *distribution* of rolling a die, not just the outcome of a roll. The code is easier to understand — when we come across a `Die` object, the meaning and intention behind it is clear — and it gives us a place to add additional die-specific functionality. For example, it would be useful for debugging if we could print not just the *outcome* of rolling a die but the die itself — otherwise, how would we know if we rolled a die with the right number of sides for the given situation?

If we were using a function to represent our die, printing it would not be useful:

```
>>> print(six_sided)
<function six_sided at 0x7f00ea3e3040>
```

That said, the `Die` class we've defined so far isn't much better:

```
>>> print(Die(6))
<__main__.Die object at 0x7ff6bcadc190>
```

With a class — and unlike a function — we can fix this. Python lets us change some of the built-in behavior of objects by overriding special methods. To change how the class is printed, we can override `__repr__`:[4]

[3] In this context, a concrete class is any class that is not an abstract class. More generally, "concrete" is the opposite of "abstract" — when an abstraction can represent multiple more specific concepts, we call any of the specific concepts "concrete."

[4] Our definition of `__repr__` used a Python feature called an "f-string." Introduced in Python 3.6, f-strings make it easier to inject Python values into strings. By putting an `f` in front of a string literal, we can include a Python value in a string: `f"{1 + 1}"` gives us the string `"2"`.

```
class Die(Distribution):
    ...
    def __repr__(self):
        return f"Die(sides={self.sides})"
```

Much better:

```
>>> print(Die(6))
Die(sides=6)
```

This seems small but makes debugging *much* easier, especially as the codebase gets larger and more complex.

Dataclasses

The `Die` class we wrote is intentionally simple. Our die is defined by a single property: the number of sides it has. The `__init__` method takes the number of sides as an input and puts it into an *attribute* of the class; once a `Die` object is created, there is no reason to change this value — if we need a die with a different number of sides, we can just create a new object. Abstractions do not have to be complex to be useful.

Unfortunately, some of the default behavior of Python classes isn't well-suited to simple classes. We've already seen that we need to override `__repr__` to get useful behavior, but that's not the only default that's inconvenient. Python's default way to compare objects for equality — the `__eq__` method — uses the `is` operator, which means it compares objects *by identity*. This makes sense for classes in general which can change over time, but it is a poor fit for simple abstraction like `Die`. Two `Die` objects with the same number of sides have the same behavior and represent the same probability distribution, but with the default version of `__eq__`, two `Die` objects declared separately will never be equal:

```
>>> six_sided = Die(6)
>>> six_sided == six_sided
True
>>> six_sided == Die(6)
False
>>> Die(6) == Die(6)
False
```

This behavior is inconvenient and confusing, the sort of edge-case that leads to hard-to-spot bugs. Just like we overrode `__repr__`, we can fix this by overriding `__eq__`:

```
def __eq__(self, other):
    return self.sides == other.sides
```

This fixes the weird behavior we saw earlier:

```
>>> Die(6) == Die(6)
True
```

However, this simple implementation will lead to errors if we use `==` to compare a `Die` with a non-`Die` value:

```
>>> Die(6) == None
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File ".../rl/chapter1/probability.py", line 18, in __eq__
    return self.sides == other.sides
AttributeError: 'NoneType' object has no attribute 'sides'
```

We generally won't be comparing values of different types with == — for `None`, `Die(6) is None` would be more idiomatic — but the usual expectation in Python is that == on different types will return `False` rather than raising an exception. We can fix by explicitly checking the type of `other`:

```
def __eq__(self, other):
    if isinstance(other, Die):
        return self.sides == other.sides
    return False
```

```
>>> Die(6) == None
False
```

Most of the classes we will define in the rest of the book follow this same pattern — they're defined by a small number of parameters, all that `__init__` does is set a few attributes and they need custom `__repr__` and `__eq__` methods. Manually defining `__init__`, `__repr__` and `__eq__` for every single class isn't *too* bad — the definitions are entirely systematic — but it carries some real costs:

- Extra code without important content makes it harder to *read* and *navigate* through a codebase.
- It's easy for mistakes to sneak in. For example, if you add an attribute to a class but forget to add it to its `__eq__` method, you won't get an error — == will just ignore that attribute. Unless you have tests that explicitly check how == handles your new attribute, this oversight can sneak through and lead to weird behavior in code that uses your class.
- Frankly, writing these methods by hand is just *tedious*.

Luckily, Python 3.7 introduced a feature that fixes all of these problems: **dataclasses**. The `dataclasses` module provides a decorator[5] that lets us write a class that behaves like `Die` without needing to manually implement `__init__`, `__repr__` or `__eq__`. We still have access to "normal" class features like inheritance (`(Distribution)`) and custom methods (`sample`):

```
from dataclasses import dataclass

@dataclass
class Die(Distribution):
    sides: int

    def sample(self):
        return random.randint(1, self.sides)
```

This version of `Die` has the exact behavior we want in a way that's easier to write and — more importantly — *far* easier to read. For comparison, here's the code we would have needed *without* dataclasses:

```
class Die(Distribution):
    def __init__(self, sides):
        self.sides = sides

    def __repr__(self):
        return f"Die(sides={self.sides})"
```

[5] Python decorators are modifiers that can be applied to class, function and method definitions. A decorator is written *above* the definition that it applies to, starting with a `@` symbol. Examples include `abstractmethod` — which we saw earlier — and `dataclass`.

```
    def __eq__(self, other):
        if isinstance(other, Die):
            return self.sides == other.sides
        return False
    def sample(self):
        return random.randint(1, self.sides)
```

As you can imagine, the difference would be even starker for classes with more attributes!

Dataclasses provide such a useful foundation for classes in Python that the *majority* of the classes we define in this book are dataclasses — we use dataclasses unless we have a *specific* reason not to.

Immutability

Once we've created a `Die` object, it does not make sense to change its number of sides — if we need a distribution for a different die, we can create a new object instead. If we change the `sides` of a `Die` object in one part of our code, it will also change in every other part of the codebase that uses that object, in ways that are hard to track. Even if the change made sense in one place, chances are it is not expected in other parts of the code. Changing state can create invisible connections between seemingly separate parts of the codebase which becomes hard to mentally track. A sure recipe for bugs!

Normally, we avoid this kind of problem in Python purely by convention: nothing *stops* us from changing `sides` on a `Die` object, but we know not to do that. This is doable, but hardly ideal; just like it is better to rely on seatbelts rather than pure driver skill, it is better to have the language prevent us from doing the wrong thing than relying on pure convention. Normal Python classes don't have a convenient way to stop attributes from changing, but luckily dataclasses do:

```
@dataclass(frozen=True)
class Die(Distribution):
    ...
```

With `frozen=True`, attempting to change `sides` will raise an exception:

```
>>> d = Die(6)
>>> d.sides = 10
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<string>", line 4, in __setattr__
dataclasses.FrozenInstanceError: cannot assign to field 'sides'
```

An object that we cannot change is called **immutable**. Instead of changing the object *in place*, we can return a fresh copy with the attribute changed; `dataclasses` provides a `replace` function that makes this easy:

```
>>> import dataclasses
>>> d6 = Die(6)
>>> d20 = dataclasses.replace(d6, sides=20)
>>> d20
Die(sides=20)
```

This example is a bit convoluted — with such a simple object, we would just write `d20 = Die(20)` — but `dataclasses.replace` becomes a lot more useful with more complex objects that have multiple attributes.

Returning a fresh copy of data rather than modifying in place is a common pattern in Python libraries. For example, the majority of Pandas operations — like `drop` or `fillna` — return a *copy* of the dataframe rather than modifying the dataframe in place. These methods have an `inplace` argument as an option, but this leads to enough confusing behavior that the Pandas team is currently deliberating on deprecating `inplace` altogether.

Apart from helping prevent odd behavior and bugs, `frozen=True` has an important bonus: we can use immutable objects as dictionary keys and set elements. Without `frozen=True`, we would get a `TypeError` because non-frozen dataclasses do not implement `__hash__`:

```
>>> d = Die(6)
>>> {d : "abc"}
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'Die'
>>> {d}
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'Die'
```

With `frozen=True`, dictionaries and sets work as expected:

```
>>> d = Die(6)
>>> {d : "abc"}
{Die(sides=6): 'abc'}
>>> {d}
{Die(sides=6)}
```

Immutable dataclass objects act like plain data — not too different from strings and ints. In this book, we follow the same practice with `frozen=True` as we do with dataclasses in general: we set `frozen=True` unless there is a specific reason not to.

0.3.3 Checking Types

A die has to have an int number of sides — `0.5` sides or `"foo"` sides simply doesn't make sense. Python will not stop us from *trying* `Die("foo")`, but we would get a `TypeError` if we tried sampling it:

```
>>> foo = Die("foo")
>>> foo.sample()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File ".../rl/chapter1/probability.py", line 37, in sample
    return random.randint(1, self.sides)
  File ".../lib/python3.8/random.py", line 248, in randint
    return self.randrange(a, b+1)
TypeError: can only concatenate str (not "int") to str
```

The types of an object's attributes are a useful indicator of how the object should be used. Python's dataclasses let us use **type annotations** (also known as "type hints") to specify the type of each attribute:

```
@dataclass(frozen=True)
class Die(Distribution):
    sides: int
```

In normal Python, these type annotations exist primarily for documentation — a user can see the types of each attribute at a glance, but the language does not raise an error when an object is created with the wrong types in an attribute . External tools — Integrated Development Environments (IDEs) and typecheckers — can catch type mismatches in annotated Python code without running the code. With a type-aware editor, `Die("foo")` would be underlined with an error message:

> Argument of type `"Literal['foo']"` cannot be assigned to parameter `"sides"` of type `"int"` in function `"__init__"`
>
> `"Literal['foo']"` is incompatible with `"int"` `[reportGeneralTypeIssues]`

This particular message comes from **pyright** running over the language server protocol (LSP), but Python has a number of different typecheckers available[6].

Instead of needing to call `sample` to see an error — which we then have to carefully read to track back to the source of the mistake — the mistake is highlighted for us without even needing to run the code.

Static Typing

Being able to find type mismatches *without running code* is called **static typing**. Some languages — like Java and C++ — require *all* code to be statically typed; Python does not. In fact, Python started out as a **dynamically typed** language with no type annotations. Type errors would only come up when the code containing the error was run.

Python is still *primarily* a dynamically typed language — type annotations are optional in most places and there is no built-in checking for annotations. In the `Die("foo")` example, we only got an error when we ran code that passed `sides` into a function that *required* an `int` (`random.randint`). We can get static checking with external tools, but even then it remains *optional* — even statically checked Python code runs dynamic type checks, and we can freely mix statically checked and "normal" Python. Optional static typing on top of a dynamically typed languages is called **gradual typing** because we can incrementally add static types to an existing dynamically typed codebase.

Dataclass attributes are not the only place where knowing types is useful; it would also be handy for function parameters, return values and variables. Python supports *optional* annotations on all of these; dataclasses are the only language construct where annotations are *required*. To help mix annotated and unannotated code, typecheckers will report mismatches in code that is explicitly annotated, but will usually not try to guess types for unannotated code.

How would we add type annotations to our example code? So far, we've defined two classes:

---

[6] Python has a number of external typecheckers, including:

- **mypy**
- **pyright**
- **pytype**
- **pyre**

The PyCharm IDE also has a propriety typchecker built-in.

These tools can be run from the command line or integrated into editors. Different checkers *mostly* overlap in functionality and coverage, but have slight differences in the sort of errors they detect and the style of error messages they generate.

- `Distribution`, an abstract class defining interfaces for probability distributions in general
- `Die`, a concrete class for the distribution of an n-sided die

We've already annotated the `sides` in `Die` has to be an `int`. We also know that the *outcome* of a die roll is an `int`. We can annotate this by adding `-> int` after `def sample(...)`:

```
@dataclass(frozen=True)
class Die(Distribution):
    sides: int

    def sample(self) -> int:
        return random.randint(1, self.sides)
```

Other kinds of concrete distributions would have other sorts of outcomes. A coin flip would either be `"heads"` or `"tails"`; a normal distribution would produce a `float`. Type annotations are particularly important when writing code for any kind of mathematical modeling because we ensure that the type specifications in our code correspond clearly to the precise specification of sets in our mathematical (notational) description.

0.3.4 Type Variables

Annotating `sample` for the `Die` class was straightforward: the outcome of a die roll is always a number, so `sample` always returns `int`. But what type would `sample` in general have? The `Distribution` class defines an interface for *any* distribution, which means it needs to cover *any* type of outcomes. For our first version of the `Distribution` class, we didn't annotate anything for `sample`:

```
class Distribution(ABC):
    @abstractmethod
    def sample(self):
        pass
```

This works — annotations are optional — but it can get confusing: some code we write will work for any kind of distribution, some code needs distributions that return numbers, other code will need something else... In every instance `sample` better return *something*, but that isn't explicitly annotated. When we leave out annotations our code will still work, but our editor or IDE will not catch as many mistakes.

The difficulty here is that different kinds of distributions — different *implementations of the* `Distribution` *interface* — will return different types from `sample`. To deal with this, we need **type variables**: variables that stand in for *some* type that can be different in different contexts. Type variables are also known as "generics" because they let us write classes that generically work for any type.

To add annotations to the abstract `Distribution` class, we will need to define a type variable for the outcomes of the distribution, then tell Python that `Distribution` is "generic" in that type:

```
from typing import Generic, TypeVar

# A type variable named "A"
A = TypeVar("A")

# Distribution is "generic in A"
class Distribution(ABC, Generic[A]):

    # Sampling must produce a value of type A
```

```
    @abstractmethod
    def sample(self) -> A:
        pass
```

In this code, we've defined a type variable `A`[7] and specified that `Distribution` uses `A` by inheriting from `Generic[A]`. We can now write type annotations for distributions *with specific types of outcomes*: for example, `Die` would be an instance of `Distribution[int]` since the outcome of a die roll is always an `int`. We can make this explicit in the class definition:

```
class Die(Distribution[int]):
    ...
```

This lets us write specialized functions that only work with certain kinds of distributions. Let's say we wanted to write a function that approximated the expected value of a distribution by sampling repeatedly and calculating the mean. This function works for distributions that have numeric outcomes — `float` or `int` — but not other kinds of distributions (How would we calculate an average for a random name?). We can annotate this explicitly by using `Distribution[float]`:[8]

```
import statistics

def expected_value(d: Distribution[float], n: int = 100) -> float:
    return statistics.mean(d.sample() for _ in range(n))
```

With this function:

- `expected_value(Die(6))` would be fine
- `expected_value(RandomName())` (where `RandomName` is a `Distribution[str]`) would be a type error

Using `expected_value` on a distribution with non-numeric outcomes would raise a type error at runtime. Having this highlighted in the editor can save us time — we see the mistake right away, rather than waiting for tests to run — and will catch the problem even if our test suite doesn't.

0.3.5 Functionality

So far, we've covered two abstractions for working with probability distributions:

- `Distribution`: an abstract class that defines the *interface* for probability distributions
- `Die`: a distribution for rolling fair n-sided dice

This is an illustrative example, but it doesn't let us do much. If all we needed were n-sided dice, a separate `Distribution` class would be overkill. Abstractions are a means for managing complexity, but any abstraction we define also adds some complexity to a codebase itself — it's one more concept for a programmer to learn and understand. It's always worth considering whether the added complexity from defining and using an abstraction is worth the benefit. How does the abstraction help us understand the code? What kind of mistakes does it prevent — and what kind of mistakes does it *encourage*? What kind of

[7] Traditionally, type variables have one-letter capitalized names — although it's perfectly fine to use full words if that would make the code clearer.

[8] The `float` type in Python *also* covers `int`, so we can pass a `Distribution[int]` anywhere that a `Distribution[float]` is required.

added functionality does it give us? If we don't have sufficiently solid answers to these questions, we should consider leaving the abstraction out.

If all we cared about were dice, `Distribution` wouldn't carry its weight. Reinforcement Learning, though, involves both a wide range of specific distributions — any given Reinforcement Learning problem can have domain-specific distributions — as well as algorithms that need to work for all of these problems. This gives us two reasons to define a `Distribution` abstraction: `Distribution` will *unify* different applications of Reinforcement Learning and will *generalize* our Reinforcement Learning code to work in different contexts. By programming against a general interface like `Distribution`, our algorithms will be able to work for the different applications we present in the book — and even work for applications we weren't thinking about when we designed the code.

One of the practical advantages of defining general-purpose abstractions in our code is that it gives us a place to add functionality that will work for *any* instance of the abstraction. For example, one of the most common operations for a probability distribution that we can sample is drawing n samples. Of course, we could just write a loop every time we needed to do this:

```
samples = []
for _ in range(100):
    samples += [distribution.sample()]
```

This code is *fine*, but it's not *great*. A `for` loop in Python might be doing pretty much *anything*; it's used for repeating pretty much anything. It's hard to understand what a loop is doing at a glance, so we'd have to carefully read the code to see that it's putting 100 samples in a list. Since this is such a common operation, we can add a method for it instead:

```
class Distribution(ABC, Generic[A]):
    ...
    def sample_n(self, n: int) -> Sequence[A]:
        return [self.sample() for _ in range(n)]
```

The implementation here is different — it's using a **list comprehension**[9] rather than a normal `for` loop — but it's accomplishing the same thing. The more important distinction happens when we *use* the method; instead of needing a `for` loop or list comprehension each time, we can just write:

[9] List comprehensions are a Python feature to build lists by looping over something. The simplest pattern is the same as writing a `for` loop:

```
foo = [expr for x in xs]

# is the same as:
foo = []
for x in xs:
    foo += [expr]
```

List comprehensions can combine multiple lists, acting like nested `for` loops:

```
>>> [(x, y) for x in range(3) for y in range(2)]
[(0, 0), (0, 1), (1, 0), (1, 1), (2, 0), (2, 1)]
```

They can also have `if` clauses to only keep elements that match a condition:

```
>>> [x for x in range(10) if x % 2 == 0]
[0, 2, 4, 6, 8]
```

Some combination of `for` and `if` clauses can let us build surprisingly complicated lists! Comprehensions will often be easier to read than loops—a loop could be doing *anything*, a comprehension is always creating a list—but it's always a judgement call. A couple of nested for loops might be easier to read than a sufficiently convoluted comprehension!

```
samples = distribution.sample_n(100)
```

The meaning of this line of code — and the programmer's intention behind it — are immediately clear at a glance.

Of course, this example is pretty limited. The list comprehension to build a list with 100 samples is a bit more complicated than just calling `sample_n(100)`, but not by much — it's still perfectly readable at a glance. This pattern of implementing general-purpose functions on our abstractions becomes a lot more useful as the functions themselves become more complicated.

However, there is another advantage to defining methods like `sample_n`: some kinds of distributions might have more efficient or more accurate ways to implement the same logic. If that's the case, we would override `sample_n` to use the better implementation. Code that uses `sample_n` would automatically benefit; code that used a loop or comprehension instead would not. For example, this happens if we implement a distribution by wrapping a function from `numpy`'s `random` module:

```
import numpy as np

@dataclass
class Gaussian(Distribution[float]):
    μ: float
    σ: float

    def sample(self) -> float:
        return np.random.normal(loc=self.μ, scale=self.σ)

    def sample_n(self, n: int) -> Sequence[float]:
        return np.random.normal(loc=self.μ, scale=self.σ, size=n)
```

`numpy` is optimized for array operations, which means that there is an up-front cost to calling `numpy.random.normal` *the first time*, but it can quickly generate additional samples after that. The performance impact is significant[10]:

```
>>> d = Gaussian(μ=0, σ=1)
>>> timeit.timeit(lambda: [d.sample() for _ in range(100)])
293.33819171399955
>>> timeit.timeit(lambda: d.sample_n(100))
5.566364272999635
```

That's a 53× difference!

The code for `Distribution` and several concrete classes implementing the `Distribution` interface (including `Gaussian`) is in the file rl/distribution.py.

0.4 Abstracting over Computation

So far, we've seen how we can build up a programming model for our domain by defining interfaces (like `Distribution`) and writing classes (like `Die`) that implement these interfaces. Classes and interfaces give us a way to model the "things" in our domain, but, in an area like Reinforcement Learning, "things" aren't enough: we also want some way to abstract over the *actions* that we're taking or the computation that we're performing.

Classes do give us one way to model behavior: **methods**. A common analogy is that objects act as "nouns" and methods act as "verbs" — methods are the actions we can take

[10] This code uses the `timeit` module from Python's standard library, which provides a simple way to benchmark small bits of Python code. By default, it measures how long it takes to execute 1,000,000 iterations of the function in seconds, so the two examples here took 0.293ms and 0.006ms respectively.

with an object. This is a useful capability that lets us abstract over doing the same kind of action on different sorts of objects. Our `sample_n` method, for example, with it's default implementation, gives us two things:

1. If we implement a new type of distribution with a custom sample method, we get `sample_n` for free for that distribution.
2. If we implement a new type of distribution which has a way to get n samples faster than calling sample n times, we can override the method to use the faster algorithm.

If we made `sample_n` a normal function we could get 1. but not 2.; if we left `sample_n` as an abstract method, we'd get 2. but not 1.. Having a non-abstract method on the abstract class gives us the best of both worlds.

So if methods are our "verbs," what else do we need? While methods abstract over actions, they do this *indirectly* — we can talk about objects as standalone values, but we can only use methods *on* objects, with no way to talk about computation itself. Stretching the metaphor with grammar, it's like having verbs without infinitives or gerunds — we'd be able to talk about "somebody skiing," but not about "skiing" itself or somebody "planning to ski!"

In this world, "nouns" (objects) are **first-class citizens** but "verbs" (methods) aren't. What it takes to be a "first-class" value in a programming language is a fuzzy concept; a reasonable litmus test is whether we can pass a value to a function or store it in a data structure. We can do this with objects, but it's not clear what this would mean for methods.

0.4.1 First-Class Functions

If we didn't have a first-class way to talk about actions (as opposed to objects), one way we could work around this would be to represent functions *as* objects with a single method. We'd be able to pass them around just like normal values and, when we needed to actually perform the action or computation, we'd just call the object's method. This solution shows us that it makes sense to have a first-class way to work with actions, but it requires an extra layer of abstraction (an object just to have a single method) which doesn't add anything substantial on its own while making our intentions less clear.

Luckily, we don't have to resort to a one-method object pattern in Python because Python has **first-class functions**: functions are already values that we can pass around and store, without needing a separate wrapper object.

First-class functions give us a new way to abstract over computation. Methods let us talk about the same kind of behavior for different objects; first-class functions let us do something *with* different actions. A simple example might be repeating the same action `n` times. Without an abstraction, we might do this with a `for` loop:

```
for _ in range(10):
    do_something()
```

Instead of writing a loop each time, we could factor this logic into a function that took `n` *and* `do_something` as arguments:

```
def repeat(action: Callable, n: int):
    for _ in range(n):
        action()

repeat(do_something, 10)
```

`repeat` takes `action` and `n` as arguments, then calls `action` `n` times. `action` has the type `Callable` which, in Python, covers functions as well as any other objects you can call with the `f()` syntax. We can also specify the return type and arguments a `Callable` should have; if we wanted the type of a function that took an `int` and a `str` as input and returned a `bool`, we would write `Callable[[int, str], bool]`.

The version with the `repeat` function makes our intentions clear in the code. A `for` loop can do many different things, while `repeat` will always just repeat. It's not a big difference in this case — the `for` loop version is sufficiently easy to read that it's not a big impediment — but it becomes more important with complex or domain-specific logic.

Let's take a look at the `expected_value` function we defined earlier:

```
def expected_value(d: Distribution[float], n: int) -> float:
    return statistics.mean(d.sample() for _ in range(n))
```

We had to restrict this function to `Distribution[float]` because it only makes sense to take an expectation of a numeric outcome. But what if we have something else like a coin flip? We would still like some way to understand the expectation of the distribution, but to make that meaningful we'd need to have some mapping from outcomes (`"heads"` or `"tails"`) to numbers (For example, we could say `"heads"` is `1` and `"tails"` is `0`.). We could do this by taking our `Coin` distribution, converting it to a `Distribution[float]` and calling `expected_value` on that, but this might be inefficient and it's certainly awkward. An alternative would be to provide the mapping as an argument to the `expected_value` function:

```
def expected_value(
  d: Distribution[A],
  f: Callable[[A], float],
  n: int
) -> float:
    return statistics.mean(f(d.sample()) for _ in range(n))
```

The implementation of `expected_value` has barely changed — it's the same `mean` calculation as previously, except we apply `f` to each outcome. This small change, however, has made the function far more flexible: we can now call `expected_value` on *any* sort of `Distribution`, not just `Distribution[float]`.

Going back to our coin example, we could use it like this:

```
def payoff(coin: Coin) -> float:
    return 1.0 if coin == "heads" else 0.0

expected_value(coin_flip, payoff)
```

The `payoff` function maps outcomes to numbers and then we calculate the expected value using that mapping.

We'll see first-class functions used in a number of places throughout the book; the key idea to remember is that *functions are values* that we can pass around or store just like any other object.

Lambdas

`payoff` itself is a pretty reasonable function: it has a clear name and works as a standalone concept. Often, though, we want to use a first-class function in some specific context where giving the function a name is not needed or even distracting. Even in cases with reasonable

names like `payoff`, it might not be worth introducing an extra named function if it will only be used in one place.

Luckily, Python gives us an alternative: `lambda`. Lambdas are function literals. We can write `3.0` and get a number without giving it a name, and we can write a `lambda` expression to get a function without giving it a name. Here's the same example as with the `payoff` function but using a `lambda` instead:

```
expected_value(coin_flip, lambda coin: 1.0 if coin == "heads" else 0.0)
```

The `lambda` expression here behaves exactly the same as `def payoff` did in the earlier version. Note how the lambda as a single expression with no `return` — if you ever need multiple statements in a function, you'll have to use a `def` instead of a `lambda`. In practice, lambdas are great for functions whose bodies are *short* expressions, but anything that's too long or complicated will read better as a standalone function `def`.

0.4.2 Iterative Algorithms

First-class functions give us an abstraction over *individual* computations: we can pass functions around, give them inputs and get outputs, but the computation between the input and the output is a complete black box. The caller of the function has no control over what happens inside the function. This limitation can be a real problem!

One common scenario in Reinforcement Learning — and other areas in numerical programming — is algorithms that *iteratively converge* to the correct result. We can run the algorithm repeatedly to get more and more accurate results, but the improvements with each iteration get progressively smaller. For example, we can approximate the square root of a by starting with some initial guess x_0 and repeatedly calculating x_{n+1}:

$$x_{n+1} = \frac{x_n + \frac{a}{x_n}}{2}$$

At each iteration, x_{n+1} gets closer to the right answer by smaller and smaller steps. At some point the change from x_n to x_{n+1} becomes small enough that we decide to stop. In Python, this logic might look something like this:

```
def sqrt(a: float) -> float:
    x = a / 2 # initial guess
    while abs(x_n - x) > 0.01:
        x_n = (x + (a / x)) / 2
    return x_n
```

The hard coded `0.01` in the `while` loop should be suspicious. How do we know that `0.01` is the right stopping condition? How do we decide when to stop at all?

The trick with this question is that we *can't* know when to stop when we're implementing a general-purpose function because the level of precision we need will depend on what the result is used for! It's the *caller* of the function that knows when to stop, not the *author*.

The first improvement we can make is to turn the `0.01` into an extra parameter:

```
def sqrt(a: float, threshold: float) -> float:
    x = a / 2 # initial guess
    while abs(x_n - x) > threshold:
        x_n = (x + (a / x)) / 2
    return x_n
```

This is a definite improvement over a literal `0.01`, but it's still limited. We've provided an extra parameter for how the function behaves, but the control is still fundamentally with the function. The caller of the function might want to stop before the method converges if it's taking too many iterations or too much time, but there's no way to do that by changing the `threshold` parameter. We could provide additional parameters for all of these, but we'd quickly end up with the logic for how to stop iteration requiring a lot more code and complexity than the iterative algorithm itself! Even that wouldn't be enough; if the function isn't behaving as expected in some specific application, we might want to print out intermediate values or graph the convergence over time — so should we include additional control parameters *for that*?

Then what do we do when we have n other iterative algorithms? Do we copy-paste the same stopping logic and parameters into each one? We'd end up with a lot of redundant code!

Iterators and Generators

This friction points to a conceptual distinction that our code does not support: *what happens at each iteration* is logically separate from *how we do the iteration*, but the two are fully coupled in our implementation. We need some way to abstract over iteration in some way that lets us separate *producing* values iteratively from *consuming* them.

Luckily, Python provides powerful facilities for doing exactly this: **iterators** and **generators**. Iterators give us a way of *consuming* values and generators give us a way of *producing* values.

You might not realize it, but chances are your Python code uses iterators all the time. Python's `for` loop uses an iterator under the hood to get each value it's looping over — this is how `for` loops work for lists, dictionaries, sets, ranges and even custom types. Try it out:

```
for x in [3, 2, 1]: print(x)
for x in {3, 2, 1}: print(x)
for x in range(3): print(x)
```

Note how the iterator for the set (`{3, 2, 1}`) prints `1 2 3` rather than `3 2 1` — sets do not preserve the order in which elements are added, so they iterate over elements in some kind of internally defined order instead.

When we iterate over a dictionary, we will print the *keys* rather than the *values* because that is the default iterator. To get values or key-value pairs we'd need to use the `values` and `items` methods respectively, each of which returns a different kind of iterator over the dictionary.

```
d = {'a': 1, 'b': 2, 'c': 3}
for k in d: print(k)
for v in d.values(): print(v)
for k, v in d.items(): print(k, v)
```

In each of these three cases we're still looping over the same dictionary, we just get a different view each time—iterators give us the flexibility of iterating over the same structure in different ways.

Iterators aren't just for loops: they give us a first-class abstraction for iteration. We can pass them into functions; for example, Python's `list` function can convert any iterator into a list. This is handy when we want to see the elements of specialized iterators if the iterator itself does not print out its values:

```
>>> range(5)
range(0, 5)
>>> list(range(5))
[0, 1, 2, 3, 4]
```

Since iterators are first-class values, we can also write general-purpose iterator functions. The Python standard library has a set of operations like this in the `itertools` module; for example, `itertools.takewhile` lets us stop iterating as soon as some condition stops holding:

```
>>> elements = [1, 3, 2, 5, 3]
>>> list(itertools.takewhile(lambda x: x < 5, elements))
[1, 3, 2]
```

Note how we converted the result of `takewhile` into a list — without that, we'd see that `takewhile` returns some kind of opaque internal object that implements that iterator specifically. This works fine — we can use the `takewhile` object anywhere we could use any other iterator — but it looks a bit odd in the Python interpreter:

```
>>> itertools.takewhile(lambda x: x < 5, elements)
<itertools.takewhile object at 0x7f8e3baefb00>
```

Now that we've seen a few examples of how we can *use* iterators, how do we define our own? In the most general sense, a Python `Iterator` is any object that implements a `__next__()` method, but implementing iterators this way is pretty awkward. Luckily, Python has a more convenient way to create an iterator by creating a *generator* using the `yield` keyword. `yield` acts similar to `return` from a function, except instead of stopping the function altogether, it outputs the yielded value to an iterator and pauses the function until the yielded element is consumed by the caller.

This is a bit of an abstract description, so let's look at how this would apply to our `sqrt` function. Instead of looping and stopping based on some condition, we'll write a version of `sqrt` that returns an iterator with each iteration of the algorithm as a value:

```
def sqrt(a: float) -> Iterator[float]:
    x = a / 2 # initial guess
    while True:
        x = (x + (a / x)) / 2
        yield x
```

With this version, we update `x` at each iteration and then `yield` the updated value. Instead of getting a single value, the caller of the function gets an iterator that contains an infinite number of iterations; it is up to the caller to decide how many iterations to evaluate and when to stop. The `sqrt` function itself has an infinite loop, but this isn't a problem because execution of the function pauses at each `yield` which lets the caller of the function stop it whenever they want.

To do 10 iterations of the `sqrt` algorithm, we could use `itertools.islice`:

```
>>> iterations = list(itertools.islice(sqrt(25), 10))
>>> iterations[-1]
5.0
```

A fixed number of iterations can be useful for exploration, but we probably want the threshold-based convergence logic we had earlier. Since we now have a first-class abstraction for iteration, we can write a general-purpose `converge` function that takes an iterator and returns a version of that same iterator that stops as soon as two values are sufficiently close. Python 3.10 and later provides `itertools.pairwise` which makes the code pretty simple:

```
def converge(values: Iterator[float], threshold: float) -> Iterator[float]:
        for a, b in itertools.pairwise(values):
            yield a

            if abs(a - b) < threshold:
                break
```

For older versions of Python, we'd have to implement our version of `pairwise` as well:

```
def pairwise(values: Iterator[A]) -> Iterator[Tuple[A, A]]:
    a = next(values, None)
    if a is None:
        return

    for b in values:
        yield (a, b)
        a = b
```

Both of these follow a common pattern with iterators: each function takes an iterator as an input *and returns an iterator as an output*. This doesn't always have to be the case, but we get a major advantage when it is: iterator → iterator operations *compose*. We can get relatively complex behavior by starting with an iterator (like our `sqrt` example) then applying *multiple* operations to it. For example, somebody calling `sqrt` might want to converge at some threshold but, just in case the algorithm gets stuck for some reason, also have a hard stop at 10,000 iterations. We don't need to write a new version of `sqrt` or even `converge` to do this; instead, we can use `converge` *with* `itertools.islice`:

```
results = converge(sqrt(n), 0.001)
capped_results = itertools.islice(results, 10000)
```

This is a powerful programming style because we can write and test each operation—`sqrt`, `converge`, `islice` — in isolation and get complex behavior by combining them in the right way. If we were writing the same logic *without* iterators, we would need a single loop that calculated each step of `sqrt`, checked for convergence *and* kept a counter to stop after 10,000 steps — and we'd need to replicate this pattern for every single such algorithm!

Iterators and generators will come up all throughout this book because they provide a programming abstraction for *processes*, making them a great foundation for the *mathematical* processes that underlie Reinforcement Learning.

0.5 Key Takeaways

- The code in this book is designed not only to illustrate the topics and algorithms covered in each chapter, but also to demonstrate some general principles of code design. Paying attention to code design gives us a codebase that's easier to understand, debug and extend.
- Code design is centered around abstractions. An abstraction is a code construct that combines separate pieces of information or unifies distinct concepts into one. Programming languages like Python provide different tools for expressing abstractions, but the goal is always the same: make the code easier to understand.
- Classes and interfaces (abstract classes in Python) define abstractions over data. Multiple implementations of the same concept can have the same interface (like `Distribution`), and code written using that interface will work for objects of *any* compatible class. Traditional Python classes have some inconvenient limitations which can be avoided by using dataclasses (classes with the `@dataclass` annotation).

- Python has type annotations which are required for dataclasses but are also useful for describing interfaces in functions and methods. Additional tools like `mypy` or PyCharm can use these type annotations to catch errors without needing to run the code.
- Functions are first-class values in Python, meaning that they can be stored in variables and passed to other functions as arguments. Classes abstract over data; functions abstract over computation.
- Iterators abstract over *iteration*: computations that happen in sequence, producing a value after each iteration. Reinforcement learning focuses primarily on iterative algorithms, so iterators become one of the key abstractions for working with different reinforcement learning algorithms.