Chapter 1
Free Sample

This free sample includes the introduction (p. 2) and the chapter on refactoring code (p. 5). The full version is available as an ebook or in print. See https://www.logicforprogrammers.com for more.

Chapter 2
Intro: How to Ride a Bike

Consider these two questions:

1. If I start a build at 3:05 PM and it takes 12 minutes to complete, when will the build be finished?
2. If I have the conditional `if(sensor_offline || sensor_inactive)`, and I know for sure that `sensor_offline` is true, does the value of `sensor_inactive` matter?

To answer the first question, we need to know how to manipulate numbers. The mathematics of numbers is called **arithmetic**. Arithmetic shows us how to multiply two numbers, add fractions, and compare the size of two numbers.

To answer the second question, we need to know how to manipulate Booleans. The mathematics of Booleans is called **logic**. Logic shows us how to simplify a Boolean expression, combine sets, and compare the strength of two statements.

Both arithmetic and logic appear everywhere in software engineering. The one key difference? Everybody learns arithmetic in elementary school while almost no one is formally taught logic. Most programmers pick up a little logic by osmosis, but even that rarely exposes us to more than just the basics.

This creates an opportunity: because logic is the single most useful math topic a programmer can learn, doing so helps us immensely in our jobs. But how are we supposed to learn it? There are plenty of books available written for philosophers, mathematicians, and computer scientists—who all have far more need for the theory than the practice. There are no books on logic meant for the self-studying programmer looking for practical skills useful in day-to-day work. It's as if nobody will teach us how to ride a bicycle, only how to build one.

This book teaches you how to ride the bicycle. After we develop a strong foundation of logic basics, we'll learn how to apply them to various everyday software problems—like testing code, designing a database, or working out customer requirements. By the end of this book, you'll have a greater understanding of all the ways software uses logic (implicitly or not), and you will be comfortable manipulating logical expressions to create the software you need.

2.1 Design Philosophy

This book is meant specifically for programmers with little to no familiarity with formal math. In all cases, I opted for accessibility and ease-of-use over precision or rigor. This is a technical how-to, not a textbook.

Notation

Math shares many operations in common with programming but uses different representations, such as writing AND as $\wedge$ instead of **&&**. I'll use programming terminology wherever possible. In cases where math symbols don't have common programming analogs (such as $\forall$), I have opted to use an explicit English equivalent (such as **all**). I included an appendix that maps conventional programming symbols to math symbols.

Math distinguishes between **predicates** and **functions**. We'll detail the differences next chapter, but as a notation, all predicates will be **TitleCase** and all functions will be **snake_case**.

Last comes the question of array indexing. Does the array **arr** start at **arr[1]** or **arr[0]**? There's no universal programming convention, as different languages make different choices. I would use the mathematician's convention except that doesn't exist either: different branches of mathematics make different choices too! So I'll default to 0-based indexing unless demonstrating a tool or language which uses 1-based indexing.

Code Samples

Some of the code samples in this book are for a specific language, in which case I'll introduce and mark the language. Otherwise, code samples are in pseudocode or Python; I've done my best to make them clear to programmers who don't know Python. As an example:

```
def f(l: list):
  if l:
    print("l is nonempty")
  else:
    print("l is empty")
```

This is based on Python's "truthiness" rules, which is plenty clear to a Pythonista! To make sure it's clear to *everybody*, I'll use more verbose expressions like `l != []` or `len(l) > 0`. The same goes for every other language covered: I'm opting for beginner-friendliness over following the recommended style.

All code blocks larger than a few lines are extracted from complete, runnable programs. These are all available to download at https://logicforprogrammers.com/code.

2.2 How to Read This Book

I recommend first reading the chapter chapter_predicates, and then moving to whichever technique looks most interesting. Techniques chapters are independent except when otherwise noted, in which case backreferences are provided.

The first six technique chapters, starting with Refactoring Code (p. 5), focus on how logic applies to everyday software. The last four, starting with chapter_data_modeling, cover special logic-based tools that unlock powerful new solutions to difficult software problems.

The book covers applications in many different fields, each of which could support a book on its own. I only show the most basic applications of each tool, but each chapter ends with a Learn More section to help you go deeper.

I've provided additional online material at https://logicforprogrammers.com/assets. This includes all code samples, version history, errata, and extra-credit topics that didn't smoothly fit in the book.

Exercises

Provided exercises are provided to help you check your knowledge and further develop your skills. All exercises are optional and have solutions in the back of the book. Some of the exercises have multiple possible solutions. Your answer can be correct even if it differs from the listed solution!

Some questions involve writing short snippets of code. In these cases, use whatever language you like.

Enough introduction. Let's learn some logic!

Chapter 3
Refactoring Code

Refactoring code is a universal programming task and knowing more tricks is always handy. This chapter will cover some of the ways we can directly use logic to simplify code. As programmers most often use logic to manipulate Boolean expressions, this chapter will focus on simplifying conditional expressions. We'll also look at some applications of using set types in code. All code samples are either code I have personally encountered or are samples of production code I found on GitHub.

3.1 Conditionals

A refactoring is a code change that does not change the code's behavior. In the last chapter, we learned about rewrite rules, which let us change a logical expression without changing its behavior. We regularly use logical expressions in our conditionals, so applying a rewrite rule counts as a logical refactoring. Take the expression:

```
if !((x && y) || !x) {do_thing()}
```

Different programming communities use different words for the same concepts. For this book I'll say that this full expression is a **conditional**, the Boolean expression that follows `if` is the **condition** and `do_thing()` inside our braces is the **body**. We can simplify the condition in the conditional like this:

Step	Expression	Rule
0	!((x && y) || !x)	Initial value
1	!((x || !x) && (!x || y))	Distribution
2	!(T && (!x || y))	x || !x is always true
3	!(!x || y)	true && y == y
4	!!x && !y	De Morgan's law
5	x && !y	Double negation

Each step in the chain uses a single logical rule. As long as we don't make a mistake in applying the rule, we don't change the value of the expression. We can be confident that our refactored code has the same behavior.

> **Tip**
>
> This is usually *much* faster to do with pen and paper.

Most of the time, we don't write out every single step along with the name of the applied rule, since the next steps are obvious in our heads. I *know* I can rewrite `(x && y) || !x` as `!x || y`, in the same way I *know* that four times three is twelve. But we can always fall back on the rewrite rules if we get confused or have to deal with something messy.

> **Tip**
>
> Some equations can be simplified automatically with tools like sympy[1].

Nested Conditionals

In a really gnarly control flow, we sometimes get conditionals inside the bodies of other conditionals:

```
if (P || !Q) {
  # body 1
} else {
  if (Q && R) {
    # body 2
  }
}
```

A useful trick is to recognize a simple rule: in the body of an `if(P)`, the condition `P` is true, whereas in the else branch, the condition `P` is false. So the else branch body is effectively this:

```
if (!(P || !Q) && Q && R) {
  # body 2
}
```

[1] https://www.sympy.org/en/index.html

We can further simplify it like this:

Step	Expression	Rule
0	`!(P || !Q) && Q && R`	Init
1	`!P && Q && Q && R`	De Morgan's law
2	`!P && Q && R`	`Q && Q == Q`
3	`!(P || !Q) && R`	De Morgan's law

Step 3 has the same expression as step 0 except without the `Q` from the middle. In other words, checking whether `Q` is true in the second `if` is unnecessary; we already know that it's true because we're in the top conditional's `else` branch! The code snippet simplifies to:

```
if (P || !Q) {
  body1
} else {
  if R { body2 }
}
```

For more sophisticated refactoring of conditional expressions, we need to pull in quantifiers.

3.2 Refactoring with Quantifiers

GitHub has lots of code like this:

```
def has_offline_server(servers):
    for s in servers:
        if s.getStatus() == "offline":
            return True
    return False
```

Consider what this does. It searches through a list to see if any element of the list satisfies a property. Doesn't that look like one of our quantifiers?

```
HasOfflineServer = some s in servers: IsOffline(s)
```

There's a difference in that **has_offline_server** iterates through list while **HasOfflineServer** quantifies over a set. But programming languages are willing to meet us halfway by providing built-in quantifier functions that work on lists. In Python, these are **all(bool_list)** and **any**. Using quantifiers in **has_offline_server** simplifies the code considerably:

```
def has_offline_server(servers):
    return any(s.getStatus() == "offline" for s in servers)
```

> **Exercise 1. Your language's quantifiers** Solution on p. 17
>
> Find the quantifiers in your language of choice. One of them should be **all** and one should be **some**.
>
> Bonus: does your language have any non-standard quantifiers, like "exactly one" or "none"?

Simplifying Quantifiers

Once we start using quantifiers, we can use quantifier rewrite rules to simplify expressions further. This can be more of an art than a science. Let's practice with this anonymized block of Python code that comes from a large public project.

```
if not all(P(x) for x in l) or any(not Q(x) for x in l):
  do_thing()
else:
  do_other_thing()
```

In formal logic, the condition is **!(all x: P(x)) || some x: !Q(x)**. I can use unscoped quantifiers for brevity and because I don't care about the type of **x**; this simplification should work regardless.

My first heuristic is to try to reduce the total number of quantifiers used. Based on the quantifier distribution rules, I know that **some** distributes over **||**, so I'll use the duality rule to turn the **!all x: P** into **some x: !P**.

Step	Expression	Rule
0	**!(all x: P(x)) || some x: !Q(x)**	Init
1	**some x: !P(x) || some x: !Q(x)**	Duality
2	**some x: !P(x) || !Q(x)**	Distribution

This is simpler mathematically and more efficient programmatically, as we only iterate over the list once instead of twice. Stopping here would be absolutely fine. I find it useful, though, to continue experimenting with rewrites. I see De Morgan's law can be used here, so I'll try that change and see where it takes us.

Step	Expression	Rule
0	some x: !P(x) || !Q(x)	Init
1	some x: !(P(x) && Q(x))	De Morgan's law
2	!all x: P(x) && Q(x)	Duality

This doesn't seem to me like a significant improvement over where we were. Even so, there was no cost to trying and it gives us good practice. This also opens up one more possible refactor: if P then Q else R is the same as if !P then R else Q. This lets us remove the top-level NOT:

```
# OLD
if not all(P(x) for x in l) or any(not Q(x) for x in l):
  do_thing()
else:
  do_other_thing()

# NEW
if all(P(x) and Q(x) for x in l):
  do_other_thing()
else:
  do_thing()
```

This last refactoring could be a step too far. Programmers tend to think of the if as the normal case and the else as the exceptional case, and by switching the two, we may have changed how they understand the code. Sometimes a more complex control flow can better match the reader's intuition. **We must always apply our best judgement as a software engineer.**

> **Exercise 2** **Solution on p. 17**
>
> I once saw some code that used the same predicate in two quantifiers:
>
> ```
> return any(P(x) for x in l) and all(P(x) for x in l)
> ```
>
> Is the any necessary? Rewrite this to use only one quantifier.

Rewriting predicates

In the previous example, we treated the predicates P(x) and Q(x) as opaque. If we can modify the predicates then we can often simplify code even further. One more anonymized example:

```
if not all(
     not a.chunks
     or len(a.chunks[0]) == df.npartitions
     for df in dfs):
  raise_error()
```

The logical representation of this conditional is !all df in dfs: (!P(df) || Q(df)). First, we can start by treating the predicates as opaque and rewrite the abstract expression:

Step	Expression	Rule
0	!all df in dfs: (!P(df) || Q(df))	Init
1	some df in dfs: !(!P(df) || Q(df))	Duality
2	some df in dfs: P(df) && !Q(df)	De Morgan's law

This corresponds to this code change:

```
# v1
if not all(
      not a.chunks
      or len(a.chunks[0]) == df.npartitions
      for df in dfs):

# v2
if any(
    a.chunks
    and not len(a.chunks[0]) == df.npartitions
    for df in dfs):
```

Looking at the code directly, though, reveals some useful details hidden by our opaque predicates. The first is that P(df) = a.chunks, which doesn't depend on df at all. There's no reason to keep it in the quantifier, and indeed we can extract it outside:

Step	Expression	Rule
0	some df in dfs: P(df) && !Q(df)	Init
1	some df in dfs: P && !Q(df)	P doesn't depend on df
2	P && some df in dfs: !Q(df)	Constant extraction

```
# v2
if any(
    a.chunks
    and not len(a.chunks[0]) == df.npartitions
    for df in dfs):

# v3
if a.chunks
   and any(
       not len(a.chunks[0]) == df.npartitions
       for df in dfs):
```

The second detail is that Q(df) compares some property of df to len(a.chunks[0]), which is effectively a constant. If we say that Q(df) = (c == df.prop), we can simplify !Q(df) into c != np(df).

Step	Expression	Rule
0	P && some df in dfs: !Q(df)	Init
1	P && some df in dfs: !(c == np(df))	Definition of Q
2	P && some df in dfs: c != np(df)	Negation

The overall code change is:

```
# v1
if not all(
     not a.chunks
     or len(a.chunks[0]) == df.npartitions
     for df in dfs):
  raise_error()

# v4
if a.chunks
   and any(len(a.chunks[0]) != df.npartitions for df in dfs):
  raise_error()
```

In general, we can define R(x) = !Q(x) to hide a negation. This only makes sense if we can find a suitable, easily understandable R(x) that doesn't muddle things. In this example, we did that by replacing the infix operator == with !=. Another way is by defining new abstract functions, like replacing !correct_password(p) with wrong_password(p).

> **Tip**
>
> In these examples, we converted the program to a logical formula and did all our rewriting before converting back. In practice it can be easier to switch between the representations: rewrite the logic, convert back to code, rewrite the code, convert back to logic, etc.

> **Exercise 3. Simplification** Solution on p. 17
>
> Simplify the expression !(x > 1 && x <= 10).

3.3 Using sets

Our last refactoring tool is a little different. Many languages have a built-in data type for sets, which can be a fantastic tool for simplifying code.

Say that we're modeling a simple social network, where every user has a list of their connections. Further, all connections are bidirectional: if I'm connected to you, then you're connected to me. We want to write a function that, for a given user, finds all other users "one hop away", or everyone that is connected to the input's connections. For simplicity we'll assume that all users are represented by string ids and that conn_list maps each user to the list of users they are connected to.

```
### Python
def get_with_lists(user, conn_list):
    # conn_list looks like {"alice": ["bob", "carol"], ...}
    out = []
    for c in conn_list[user]:
        for u in conn_list[c]:
            if u != user and u not in out:
                out.append(u)
    return out
```

The output should be a collection of *unique* users. Since list don't guarantee uniqueness by default, we have to make sure our function does not add duplicates. This forces us to iterate through **conn_list[u]** one element at a time, checking that the connection isn't already in **out** before we append it. On the other hand, if **g.members** instead returned a *set*, our function is simpler:

```
def get_with_sets(user, conn_set):
    out = set()
    for c in conn_set[user]:
        out |= conn_set[c] #like += but with set union
    out -= {user} #same but with set difference
    return out
```

Our uniqueness constraint is instead directly enforced by the type itself; we can just union the entire set of connections to **out**. Since sets can't encode ordered or duplicate data, language implementers can make some set operations more efficient than corresponding list operations. I provided a Python benchmark in the chapter 3 code samples[2]: as the connection graph grows larger, the set-based approach becomes orders of magnitude faster. This holds even when we include the time taken to construct a set representation from a list representation!

On top of these benefits, sets also provide a useful signal to other programmers reading our code. When I see a codebase that uses both sets and lists, I can be confident it uses sets for unique unordered data and lists for data that *must* be ordered or duplicated.

This is our first application of the ability-guarantee tradeoff. Sets can represent a strict subset of what lists can represent, but we can be certain that our set doesn't have duplicate data. That certainty is why we can make some set operations so much faster. We'll see more tradeoffs in future chapters.

> **Tip**
>
> A couple of common data types can represent more than sets while giving more guarantees than lists. A **bag** is a collection with duplicates but no ordering. In some language libraries these are also called **multisets** or **counters**. An **ordered set** has ordering but no duplicates. I am not familiar with any other terms for this data type.

[2] https://logicforprogrammers.com/code/3

3.4 Programs are not math

Logic gives us lots of ways to rewrite conditionals. Unfortunately, we can't always use them: programming languages don't always follow the same rules as mathematics.

Sets and Lists

Logic primarily uses *sets* while programming languages use *lists*. Many programming languages don't even have sets, and those that do often place restrictions on what can go into a set and/or determine membership by reference and not value. This can lead to some surprising behavior:

```
// javascript
var s = new Set();
s.add([1]);
s.add([1]); // different identity
console.log(s); // prints Set([1], [1])
console.log(s.has([1])); // prints false
```

For this reason, I recommend getting a good sense of your language's set semantics before using sets as an optimization.

Emulating implication

In math, we can rewrite `!x || y` into `x => y`. We can't do this while refactoring because almost no programming language has an implication operator. This can be annoying when trying to use implication inside of a quantifier:

```
# Not valid python
if all((l != [] => x in l) for l in lists):
```

We can sometimes use the trick of replacing an implication with a filter, as seen in the next exercise.

Exercise 4. Implication via filtering Solution on p. 17

Explain why these two logical expressions are equivalent:

```
all x in set: P(x) => Q(x)      # (a)
all x in {x in set: P(x)}: Q(x) # (b)
```

Conditionals and Side Effects

One last important difference: in many programming languages, evaluating a Boolean expression may produce a **side effect**, or change to the program state. If we don't account for the side effect, a correct-looking refactor might introduce a bug! Take, for example, this conditional:

```
# f(x) is some function that returns a boolean
if x == [] and f(x) and x != []:
    print(1)
else:
    print(2)
```

On first look, this looks trivially replaceable with `print(2)`, because `P && Q && !P` is always false. But what evaluating `f(x)` modifies `x`? Then the value of `x` in `x == []` is different than the value of `x` in `x != []`, and the expression can be true after all!

On top of this, languages often do **short-circuit evaluation**: in the expression `x = f() && g()`, `g()` is only evaluated if `f()` returns true. If `f()` returns false, the program sets `x` to false without running `g()` at all. This means swapping the operands to `x = g() && f()` might change the behavior of the program, as it will change which side effects actually happen.

We can only safely apply rewrite rules to boolean expressions that don't have any side effects. If even one clause can change state, then even replacing `f() || !f()` with `true` carries the risk of a program change.

3.5 Summary

- Logic provides us with rewrite rules we can use to refactor Boolean expressions.
- Many languages support quantifiers, which let us further simplify code.

- Set types can, in some circumstances, be clearer and more efficient than list types.
- Be careful: not all logical refactorings are supported by all languages, and some language features like side effects can make rewrites unsafe.

No refactor is complete until we have thoroughly tested that the behavior is the same. In the next chapter, we'll learn a logic-based technique to test refactorings and code more broadly.

Chapter 4
Exercise Solutions

Solution to Exercise 1 on p. 8. Your language's quantifiers

Python and Haskell use all() and any(). JavaScript uses Array.every() and Array.some(). C++ has std::all_of() and std::any_of, and also has std::none_of. Ruby has all?, any?, none?, *and* one?, which is only true if exactly one element passes the predicate.

Solution to Exercise 2 on p. 9

In all x in set: P(x) && some x in set: P(x), the only thing that the some is doing is checking that the set is nonempty, as that's the only case where all x can be true and some x can be false. So, we can rewrite the code to not have that:

```
return l != [] and all(P(x) for x in l)
```

Solution to Exercise 3 on p. 12. Simplification

```
x <= 1 || x > 10
```

Solution to Exercise 4 on p. 15. Implication via filtering

In all x in set: P(x) => Q(x) we only check Q(x) on the elements that also satisfy P(x). So, we just need to show that the same is true for all x in {x in set: P(x)}: Q(x).

- If P(x) is true, we leave it in the set and then have to check Q(x).
- If P(x) is false, we throw it out of the set and never check Q(x).

Then (b) only checks Q(x) on elements that satisfy P(x), so the two are equivalent. This speaks to a broader relationship between implication and subsets: if all x in set: P(x) => Q(x), then {x in set: P(x)} is a subset of {x in set: Q(x)}.