Introduction

> *When given the bug report, the developer responded, shaking his head vigorously, "Oh. That's not a bug."*
> *"What do you mean, 'that's not a bug'? It crashes."*
> *"Look," shrugged the developer. "It can crash or it can corrupt your data. Take your pick."*
>
> —Gerald Weinberg, "Perfect Software: And Other Illusions About Testing"

Programming is fun, even if it can sometimes feel a bit painful, like pounding nails into your head—and if it does, don't worry, it's not just you. We all feel that way from time to time, and it's not because we're bad programmers.

It's usually because we're trying to understand or untangle some complicated code that isn't working properly, and that's one of the hardest jobs a programmer will ever do. How do you avoid getting into a tangle like this in the first place?

Another not-so-fun thing is building large and complex projects from scratch, when we're running around trying to design the thing, figure out the requirements, and write the code, all without turning it into a big pile of spaghetti. How the heck do you do that?

But software engineers have learned a trick or two over the years. We've developed a way of programming that helps us organise our thinking about the product, guides us through the tricky early stages, leads us towards good designs, gives us confidence in the correctness of what we're doing, catches bugs before users see them, and makes it easier and safer for us to make changes in the future.

It's a way of programming that's more enjoyable, less stressful, and more productive than any other that I've come across. It also produces much better results. I'll share some of these insider secrets with you in this book. Let's get started!

1. Programming with confidence

> *It seemed that for any piece of software I wrote, after a couple of years I started hating it, because it became increasingly brittle and terrifying.*
>
> *Looking back in the rear-view, I'm thinking I was reacting to the experience, common with untested code, of small changes unexpectedly causing large breakages for reasons that are hard to understand.*
>
> —Tim Bray, "Testing in the Twenties"

When you launch yourself on a software engineering career, or even just a new project, what goes through your mind? What are your hopes and dreams for the software you're going to write? And when you look back on it after a few years, how will you feel about it?

There are lots of qualities we associate with good software, but undoubtedly the most important is that it be *correct*. If it doesn't do what it's supposed to, then almost nothing else about it matters.

Self-testing code

How do we know that the software we write is correct? And, even if it starts out that way, how do we know that the minor changes we make to it aren't introducing bugs?

One thing that can help give us confidence about the correctness of software is to write *tests* for it. While tests are useful whenever we write them, it turns out that they're especially useful when we write them *first*. Why?

The most important reason to write tests first is that, to do that, we need to have a clear idea of how the program should behave, from the user's point of view. There's some thinking involved in that, and the best time to do it is before we've written any code.

Why? Because trying to write code before we have a clear idea of what it should do is simply a waste of time. It's almost bound to be wrong in important ways. We're also likely to end up with a design which might be convenient from the point of view of the *implementer*, but that doesn't necessarily suit the needs of users at all.

Working test-first encourages us to develop the system in small *increments*, which helps prevent us from heading too far down the wrong path. Focusing on small, simple chunks of user-visible behaviour also means that everything we do to the program is about making it more valuable to users.

Tests can also guide us toward a good design, partly because they give us some experience of using our own APIs, and partly because breaking a big program up into small, independent, well-specified modules makes it much easier to understand and work on.

What we aim to end up with is *self-testing code*:

> *You have self-testing code when you can run a series of automated tests against the code base and be confident that, should the tests pass, your code is free of any substantial defects.*
>
> *One way I think of it is that as well as building your software system, you simultaneously build a bug detector that's able to detect any faults inside the system. Should anyone in the team accidentally introduce a bug, the detector goes off.*
>
> —Martin Fowler, "Self-Testing Code"

This isn't just because well-tested code is more reliable, though that's important too. The real power of tests is that they make developers happier, less stressed, and more productive as a result.

> *Tests are the Programmer's Stone, transmuting fear into boredom. "No, I didn't break anything. The tests are all still green." The more stress I feel, the more I run the tests. Running the tests immediately gives me a good feeling and reduces the number of errors I make, which further reduces the stress I feel.*
>
> —Kent Beck, "Test-Driven Development by Example"

The adventure begins

Let's see what writing a function test-first looks like in Go. Suppose we're writing an old-fashioned text adventure game, like Zork, and we want the player to see something like this:

> *Attic*
> The attics, full of low beams and awkward angles, begin here in a relatively tidy area which extends north, south and east. You can see here a battery, a key, and a tourist map.

Adventure games usually contain lots of different locations and items, but one thing that's common to every location is that we'd like to be able to list its contents in the form of a sentence:

```
You can see here a battery, a key, and a tourist map.
```

Suppose we're storing these items as strings, something like this:

```
a battery
a key
a tourist map
```

How can we take a bunch of strings like this and list them in a sentence, separated by commas, and with a concluding "and"? It sounds like a job for a function; let's call it `ListItems`.

What kind of test could we write for such a function? You might like to pause and think about this a little.

One way would be to call the function with some specific *inputs* (like the strings in our example), and see what it returns. We can predict what it should return when it's working properly, so we can compare that prediction against the actual result.

Here's one way to write that in Go, using the built-in `testing` package:

```
func TestListItems_GivesCorrectResultForInput(t *testing.T) {
    t.Parallel()
    input := []string{
        "a battery",
        "a key",
        "a tourist map",
    }
    want := "You can see here a battery, a key, and a tourist map."
    got := game.ListItems(input)
    if want != got {
        t.Errorf("want %q, got %q", want, got)
```

```
    }
}
```

(Listing game/1)

Don't worry too much about the details for now; we'll deal with them later. The gist of this test is as follows:

1. We call the function `game.ListItems` with our test inputs.
2. We check the result against the expected string.
3. If they're not the same, we call `t.Errorf`, which causes the test to fail.

Note that we've written this code as though the `game.ListItems` function already *exists*. It doesn't. This test is, at the moment, an exercise in imagination. It's saying *if* this function existed, here's what we think it should return, given this input.

But it's also interesting that we've nevertheless made a number of important design decisions as an unavoidable part of writing this test. First, we have to *call* the function, so we've decided its name (`ListItems`), and what package it's part of (`game`).

We've also decided that its parameter is a slice of strings, and (implicitly) that it returns a single result that is a string. Finally, we've encoded the exact behaviour of the function into the test (at least, for the given inputs), by specifying exactly what the function should produce as a result.

> *The original description of test-driven development was in an ancient book about programming. It said you take the input tape, manually type in the output tape you expect, then program until the actual output tape matches the expected output.*
>
> *When describing this to older programmers, I often hear, "Of course. How else could you program?"*
>
> —Kent Beck

Naming something and deciding its inputs, outputs, and behaviour are usually the hardest decisions to make about any software component, so even though we haven't yet written a single line of code for `ListItems`, we've actually done some pretty important *thinking* about it.

And the mere fact of writing the test has also had a significant influence on the *design* of `ListItems`, even if it's not very visible. For example, if we'd just gone ahead and written `ListItems` first, we might well have made it print the result to the terminal. That's fine for the real game, but it would be difficult to test.

Testing a function like `TestItems` requires *decoupling* it from some specific output device, and making it instead a *pure function*: that is, a function whose result is deterministic, depends on nothing but its inputs, and has no side-effects.

Functions that behave like this tend to make a system easier to understand and reason about, and it turns out that there's a deep synergy between *testability* and good design, which we'll return to later in this book.

Verifying the test

So what's the next step? Should we go ahead and implement `ListItems` now and make sure the test passes? We'll do that in a moment, but there's a step we need to take first. We need some feedback on whether the *test* itself is correct. How could we get that?

It's helpful to think about ways the test could be *wrong*, and see if we can work out how to catch them. Well, one major way the test could be wrong is that it might not fail when it's supposed to.

Tests in Go pass by default, unless you explicitly make them fail, so a test function with no code at all would always pass, no matter what:

```
func TestAlwaysPasses(t *testing.T) {}
```

That test is so obviously useless that we don't need to say any more. But there are more subtle ways to accidentally write a useless test. For example, suppose we mistakenly wrote something like this:

```
if want != want {
    t.Errorf("want %q, got %q", want, got)
}
```

A value always equals itself, so this `if` statement will never be true, and the test will never fail. We might spot this just by looking at the code, but then again we might not.

I've noticed that when I teach Go to my students, this is a concept that often gives them trouble. They can readily imagine that the function itself might be wrong. But it's not so easy for them to encompass the idea that the *test* could be wrong. Sadly, this is something that happens all too often, even in the best-written programs.

Until you've seen the test fail as expected, you don't really have a test.

So we can't be *sure* that the test doesn't contain logic bugs unless we've seen it fail when it's supposed to. When *should* the test fail, then? When `ListItems` returns the wrong result. Could we arrange that? Certainly we could.

That's the next step, then: write just enough code for `ListItems` to return the wrong result, and verify that the test fails in that case. If it doesn't, we'll know we have a problem with the test that needs fixing.

Writing an incorrect function doesn't sound too difficult, and something like this would be fine:

```
func ListItems(items []string) string {
    return ""
}
```

Almost everything here is dictated by the decisions we already made in the test: the function name, its parameter type, its result type. And all of these need to be there in

order for us to call this function, even if we're only going to implement enough of it to return the wrong answer.

The only real choice we need to make here, then, is what actual result to return, remembering that we want it to be *incorrect*.

What's the simplest incorrect string that we could return given the test inputs? Just the empty string, perhaps. Any other string would also be fine, provided it's not the one the test expects, but an empty string is the easiest to type.

Running tests with `go test`

Let's run the test and check that it does fail as we expect it to:

```
go test
```

```
--- FAIL: TestListItems_GivesCorrectResultForInput (0.00s)
    game_test.go:18: want "You can see here a battery, a key, and
    a tourist map.", got ""
FAIL
exit status 1
FAIL    game    0.345s
```

Reassuring. We *know* the function doesn't produce the correct result yet, so we expected the test to detect this, and it did.

If, on the other hand, the test had *passed* at this stage, or perhaps failed with some different error, we would know there was a problem. But it seems to be fine, so now we can go ahead and implement `ListItems` for real.

Here's one rough first attempt:

```
func ListItems(items []string) string {
    result := "You can see here"
    result += strings.Join(items, ", ")
    result += "."
    return result
}
```

(Listing game/1)

I really didn't think too hard about this, and I'm sure it shows. That's all right, because we're not aiming to produce elegant, readable, or efficient code at this stage. Trying to write code from scratch that's both correct *and* elegant is pretty hard. Let's not stack the odds against ourselves by trying to multi-task here.

In fact, the only thing we care about right *now* is getting the code correct. Once we have that, we can always tidy it up later. On the other hand, there's no point trying to beautify code that doesn't work yet.

> *The goal right now is not to get the perfect answer but to pass the test. We'll make our sacrifice at the altar of truth and beauty later.*
>
> —Kent Beck, "Test-Driven Development by Example"

Let's see how it performs against the test:

```
--- FAIL: TestListItems_GivesCorrectResultForInput (0.00s)
    game_test.go:18: want "You can see here a battery, a key, and
    a tourist map.", got "You can see herea battery, a key, a
    tourist map."
```

Well, that looks *close*, but clearly not exactly right. In fact, we can improve the test a little bit here, to give us a more helpful failure message.

Using `cmp.Diff` to compare results

Since part of the result is correct, but part isn't, we'd actually like the test to report the *difference* between `want` and `got`, not just print both of them out.

There's a useful third-party package for this, `go-cmp`. We can use its `Diff` function to print just the differences between the two strings. Here's what that looks like in the test:

```
func TestListItems_GivesCorrectResultForInput(t *testing.T) {
    t.Parallel()
    input := []string{
        "a battery",
        "a key",
        "a tourist map",
    }
    want := "You can see here a battery, a key, and a tourist map."
    got := game.ListItems(input)
    if want != got {
        t.Error(cmp.Diff(want, got))
    }
}
```

(Listing game/2)

Here's the result:

```
--- FAIL: TestListItems_GivesCorrectResultForInput (0.00s)
    game_test.go:20:   strings.Join({
                "You can see here",
        -       " ",
                "a battery, a key,",
        -       " and",
```

```
            " a tourist map.",
        }, "")
```

When two strings differ, `cmp.Diff` shows which parts are the same, which parts are only in the first string, and which are only in the second string.

According to this output, the first part of the two strings is the same:

```
"You can see here",
```

But now comes some text that's only in the first string (`want`). It's preceded by a minus sign, to indicate that it's missing from the second string, and the exact text is just a space, shown in quotes:

```
-       " ",
```

So that's one thing that's wrong with `ListItems`, as detected by the test. It's not including a space between the word "here" and the first item.

The next part, though, `ListItems` got right, because it's the same in both `want` and `got`:

```
"a battery, a key,",
```

Unfortunately, there's something else present in `want` that is missing from `got`:

```
-       " and",
```

We forgot to include the final "and" before the last item. The two strings are otherwise identical at the end:

```
" a tourist map.",
```

You can see why it's helpful to show the *difference* between `want` and `got`: instead of a simple pass/fail test, we can see how close we're getting to the correct result. And if the result were very long, the diff would make it easy to pick out which parts of it weren't what we expected.

Let's make some tweaks to `ListItems` now to address the problems we detected:

```
func ListItems(items []string) string {
    result := "You can see here "
    result += strings.Join(items[:len(items)-1], ", ")
    result += ", and "
    result += items[len(items)-1]
    result += "."
    return result
}
```

(Listing game/3)

A bit ugly, but who cares? As we saw earlier, we're not trying to write beautiful code at this point, only correct code. This approach has been aptly named "Shameless Green":

> *The most immediately apparent quality of Shameless Green code is how very simple it is. There's nothing tricky here. The code is gratifyingly easy to comprehend. Not only that, despite its lack of complexity this solution does extremely well.*
>
> —Sandi Metz & Katrina Owen, "99 Bottles of OOP: A Practical Guide to Object-Oriented Design"

In other words, shameless green code passes the tests in the simplest, quickest, and most easily understandable way possible. That kind of solution may not be the *best*, as we've said, but it may well be good enough, at least for now. If we suddenly had to drop everything and ship right now, we *could* grit our teeth and ship this.

So does `ListItems` work now? Tests point to yes:

```
go test
```

```
PASS
ok      game    0.160s
```

The test is passing, which means that `ListItems` is behaving correctly. That is to say, it's doing what we asked of it, which is to format a list of three items in a pleasing way.

New behaviour? New test.

Are we asking *enough* of `ListItems` with this test? Will it be useful in the actual game code? If the player is in a room with exactly three items, we can have some confidence that `ListItems` will format them the right way. And four or more items will probably be fine too.

What about just two items, though? From looking at the code, I'm not sure. It *might* work, or it might do something silly. Thinking about the case of *one* item, though, I can see right away that the result won't make sense.

The result of formatting a slice of *no* items clearly won't make sense either. So what should we do? We could add some code to `ListItems` to handle these cases, and that's what many programmers would do in this situation.

But hold up. If we go ahead and make that change, then how will we know that we got it right? We can at least have some confidence that we won't break the formatting for three or more items, since the test would start failing if that happened. But we won't have any way to know if our new code correctly formats two, one, or zero items.

We started out by saying we have a specific job that we want `ListItems` to do, and we defined it carefully in advance by writing the test. `ListItems` now does that job, since it passes the test.

If we're now deciding that, on reflection, we want `ListItems` to do *more*, then that's perfectly all right. We're allowed to have new ideas while we're programming: indeed, it would be a shame if we didn't.

But let's adopt the rule "new behaviour, new test". Every time we think of a new behaviour we want, we have to write a test for it, or at least extend an existing passing test so that it fails for the case we're interested in.

That way, we'll be forced to get our ideas absolutely clear before we start coding, just like with the first version of `ListItems`. And we'll also know when we've written *enough* code, because the test will start passing.

This is another point that I've found my students sometimes have difficulty with. Often, the more experienced a programmer they are, the more trouble it gives them. They're so used to just going ahead and writing code to solve the problem that it's hard for them to insert an extra step in the process: writing a new *test*.

Even when they've written a function test-first to start with, the temptation is then to start extending the behaviour of that function, without pausing to extend the test. In that case, just saying "New behaviour, new test" is usually enough to jog their memory. But it can take a while to thoroughly establish this new habit, so if you have trouble at first, you're not alone. Stick at it.

Test cases

We could write some new test functions, one for each case that we want to check, but that seems a bit wasteful. After all, each test is going to do exactly the same thing: call `ListItems` with some input, and check the result against expectations.

Any time we want to do the same operation repeatedly, just with different data each time, we can express this idea using a *loop*. In Go, we usually use the `range` operator to loop over some slice of data.

What data would make sense here? Well, this is clearly a slice of test *cases*, so what's the best data structure to use for each case?

Each case here consists of two pieces of data: the strings to pass to `ListItems`, and the expected result. Or, to put it another way, `input` and `want`, just like we have in our existing test.

One of the nice things about Go is that any time we want to group some related bits of data into a single value like this, we can just define some arbitrary `struct` type for it. Let's call it `testCase`:

```
func TestListItems_GivesCorrectResultForInput(t *testing.T) {
    type testCase struct {
        input []string
        want  string
    }
    ...
```

(Listing game/4)

How can we refactor our existing test to use the new `testCase` struct type? Well, let's start by creating a slice of `testCase` values with just one element: the three-item case we already have.

```
...
cases := []testCase{
    {
        input: []string{
            "a battery",
            "a key",
            "a tourist map",
        },
        want:
        "You can see here a battery, a key, and a tourist map.",
    },
}
...
```

(Listing game/4)

What's next? We need to loop over this slice of cases using `range`, and for each case, we want to pass its `input` value to `ListItems` and compare the result with its `want` value.

```
...
for _, tc := range cases {
    got := game.ListItems(tc.input)
    if tc.want != got {
        t.Error(cmp.Diff(tc.want, got))
    }
}
```

(Listing game/4)

This looks very similar to the test we started with, except that most of the test body has moved inside this loop. That makes sense, because we're doing exactly the same thing in the test, but now we can do it repeatedly for multiple *cases*.

This is commonly called a *table test*, because it checks the behaviour of the system given a table of different inputs and expected results. Here's what it looks like when we put it all together:

```
func TestListItems_GivesCorrectResultForInput(t *testing.T) {
    type testCase struct {
        input []string
        want  string
```

```
    }
    cases := []testCase{
        {
            input: []string{
                "a battery",
                "a key",
                "a tourist map",
            },
            want:
            "You can see here a battery, a key, and a tourist map.",
        },
    }
    for _, tc := range cases {
        got := game.ListItems(tc.input)
        if tc.want != got {
            t.Error(cmp.Diff(tc.want, got))
        }
    }
}
```

(Listing game/4)

First, let's make sure we didn't get anything wrong in this refactoring. The test should still pass, since it's still only testing our original three-item case:

```
PASS
ok      game    0.222s
```

Great. Now comes the payoff: we can easily add more cases, by inserting extra elements in the `cases` slice.

Adding cases one at a time

What new test cases should we add at this stage? We could add lots of cases at once, but since we feel pretty sure they'll all fail, there's no point in that.

Instead, let's treat each case as describing a new behaviour, and tackle one of them at a time. For example, there's a certain way the system should behave when given *two* inputs instead of three, and it's distinct from the three-item case. We'll need some special logic for it.

So let's add a single new case that supplies two items:

```
{
```

```
    input: []string{
        "a battery",
        "a key",
    },
    want: "You can see here a battery and a key.",
},
```

(Listing game/5)

The value of `want` is up to us, of course: what we want to happen in this case is a product design decision. This is what I've decided *I* want, with my game designer hat on, so let's see what `ListItems` actually does:

```
--- FAIL: TestListItems_GivesCorrectResultForInput (0.00s)
    game_test.go:36:   strings.Join({
            "You can see here a battery",
        +   ",",
            " and a key.",
          }, "")
```

Not bad, but not perfect. It's inserting a comma after "battery" that shouldn't be there.

Now let's try to fix that. For three or more items, we'll always want the comma, and for two, one, or zero items, we won't. So the quickest way to get this test to pass is probably to add a special case to `ListItems`, when `len(items)` is less than 3:

```
func ListItems(items []string) string {
    result := "You can see here "
    if len(items) < 3 {
        return result + items[0] + " and " + items[1] + "."
    }
    result += strings.Join(items[:len(items)-1], ", ")
    result += ", and "
    result += items[len(items)-1]
    result += "."
    return result
}
```

(Listing game/5)

Again, this isn't particularly elegant, nor does it need to be. We just need to write the minimum code to pass the current failing test case. In particular, we don't need to worry about trying to pass test cases we don't *have* yet, even if we plan to add them later:

Add one case at a time, and make it pass before adding the next.

The test passes for the two cases we've defined, so now let's add the one-item case:

```
{
    input: []string{
        "a battery",
    },
    want: "You can see a battery here.",
},
```

(Listing game/6)

Note the slightly different word order: "you can see here a battery" would sound a little odd.

Let's see if this passes:

```
--- FAIL: TestListItems_GivesCorrectResultForInput (0.00s)
panic: runtime error: index out of range [1] with length 1
[recovered]
```

Oh dear. `ListItems` is now panicking, so that's even worse than simply failing. In the immortal words of Wolfgang Pauli, it's "not even wrong".

Quelling a panic

Panics in Go are accompanied by a stack trace, so we can work our way through it to see which line of code is the problem. It's this one:

```
return result + items[0] + " and " + items[1] + "."
```

This is being executed in the case where there's only one item (`items[0]`), so we definitely can't refer to `items[1]`: it doesn't exist. Hence the panic.

Let's treat the one-item list as another special case:

```
func ListItems(items []string) string {
    result := "You can see here "
    if len(items) == 1 {
        return "You can see " + items[0] + " here."
    }
    if len(items) < 3 {
        return result + items[0] + " and " + items[1] + "."
    }
    result += strings.Join(items[:len(items)-1], ", ")
    result += ", and "
    result += items[len(items)-1]
```

```
    result += "."
    return result
}
```

(Listing game/6)

This eliminates the panic, and the test now passes for this case.

Let's keep going, and add the zero items case. What should we expect `ListItems` to return?

```
{
    input: []string{},
    want:  "",
},
```

(Listing game/7)

Just the empty string seems reasonable. We could have it respond "You see nothing here", but it would be a bit weird to get that message every time you enter a location that happens to have no items in it, which would probably be true for most locations.

Running this test case panics again:

```
--- FAIL: TestListItems_GivesCorrectResultForInput (0.00s)
panic: runtime error: index out of range [0] with
length 0 [recovered]
```

And we can guess what the problem is without following the stack trace: if `items` is empty, then we can't even refer to `items[0]`. Another special case:

```
if len(items) == 0 {
    return ""
}
```

(Listing game/7)

This passes.

Refactoring

We saw earlier that it doesn't matter how elegant the code for `ListItems` looks, if it doesn't do the right thing. So now that it *does* do the right thing, we're in a good place, because we have options. If we had to ship right now, this moment, we could actually do that. We wouldn't be delighted about it, because the code is hard to read and maintain, but *users* don't care about that. What *they* care about is whether it solves their problem.

But maybe we don't have to ship right now. Whatever extra time we have in hand, we can now use to refactor this correct code to make it nicer. And, while there's always a risk of making mistakes or introducing bugs when refactoring, we have a safety net: the test.

The definition of "refactoring", by the way, is changing code without changing its *behaviour* in any relevant way. Since the test defines all the behaviour we consider relevant, we can change the code with complete freedom, relying on the test to tell us the moment the code starts *behaving* differently.

Since we have four different code paths, depending on the number of input items, we can more elegantly write that as a `switch` statement with four cases:

```
func ListItems(items []string) string {
    switch len(items) {
    case 0:
        return ""
    case 1:
        return "You can see " + items[0] + " here."
    case 2:
        return "You can see here " + items[0] + " and " +
            items[1] + "."
    default:
        return "You can see here " +
            strings.Join(items[:len(items)-1], ", ") +
            ", and " + items[len(items)-1] + "."
    }
}
```

(Listing game/8)

Did we break anything or change any behaviour? No, because the test still passes. Could we have written `ListItems` from the start using a `switch` statement, saving this refactoring step? Of course, but we've ended up here anyway, just by a different route.

In fact, all good programs go through at least a few cycles of refactoring. We shouldn't even try to write the final program in a single attempt. Instead, we'll get much better results by aiming for *correct* code first, then iterating on it a few times to make it clear, readable, and easy to maintain.

> *Writing is basically an iterative process. It is a rare writer who dashes out a finished piece; most of us work in circles, returning again and again to the same piece of prose, adding or deleting words, phrases, and sentences, changing the order of thoughts, and elaborating a single sentence into pages of text.*
>
> —Dale Dougherty & Tim O'Reilly, "Unix Text Processing"

Well, that was easy

No doubt there's more refactoring we could do here, but I think you get the point. We've developed some correct, reasonably readable code, with accompanying tests that completely define the behaviour users care about.

And we did it without ever having to really *think* too hard. There were no brain-busting problems to solve; we didn't have to invent any complicated algorithms previously unknown to computer science, or that you might be tested on in some job interview. We didn't use any advanced features of Go, just basic strings, slices, and loops.

That's a good thing. If code is hard to write, it'll be hard to understand, and even harder to debug. So we *want* the code to be easy to write.

> *Everyone knows that debugging is twice as hard as writing a program in the first place. So if you're as clever as you can be when you write it, how will you ever debug it?*
>
> —Brian Kernigan & P.J. Plauger, "The Elements of Programming Style"

If we find that writing the code is hard, we'll reduce the scope of our ambition so that we're solving some simpler problem instead. We'll keep simplifying the problem in this way until the code becomes easy and obvious. Then we can gradually build back up to the real problem we started with.

The key point is that we're always writing *code,* always testing our evolving ideas against a running program, instead of getting trapped into doing "big design up front": that never works.

> *My way of writing code is, you sculpt it, you get something as good as you can, and everything's subject to change, always, as you learn. But you climb this ladder of learning about your problem. Every problem's unique, so you have to learn about each problem, and you do something and get a better vantage point. And from that vantage point you can decide to throw it out. Code is cheap. But often it tells you what to do next.*
>
> —Andy Herzfeld, quoted in Scott Rosenberg's "Dreaming in Code"

At every stage in this process, we had the confidence that comes from having a good test. It caught our occasional missteps, showed us from moment to moment how close we were getting to the correct solution, and what still remained to be done.

As we generated new ideas during development, it was easy to add them as new test cases, and we only had to do a very little work to make them pass. Once we had all the cases passing, we were able to confidently refactor the entire function, without worrying about breaking anything or introducing subtle bugs.

If you're not used to this way of programming with tests, it might all seem a bit long-winded for a relatively simple function. But the process is a lot quicker to *do* than it is to explain step by step.

In a real-life programming situation, it would probably take me a couple of minutes to write this test, two or three minutes to get it passing, and maybe another couple of minutes to refactor the code. Let's generously say ten minutes is a reasonable time to build something like `ListItems` using this workflow.

Could we do it faster if we didn't bother about the test? Perhaps. But *significantly* faster? I doubt it. To write a working `ListItems` from scratch that handles all these cases would take me a good ten minutes, I think. Indeed, it would actually be harder work, because I'd have to *think* very carefully at each stage and painstakingly work out whether the code is going to produce the correct result.

Even then, I wouldn't be completely confident that it was correct without seeing it *run* a few times, so I'd have to go on and write some throwaway code just to call `ListItems` with some inputs and print the result. And it probably *wouldn't* be correct, however carefully and slowly I worked. I would probably have missed out a space, or something. So I'd have to go back and fix that.

> *It will feel slow at first. The difference is, when we're done, we're really done. Since we're getting closer to really done, the apparent slowness is an illusion. We're eliminating most of that long, painful test-and-fix finish that wears on long after we were supposed to be done.*
>
> —Ron Jeffries, "The Nature of Software Development"

In other words, the test-first workflow isn't slow at all, once you're familiar with it. It's quick, enjoyable, and productive, and the result is correct, readable, self-testing code.

Watching an experienced developer build great software fast, guided by tests, can be a transformative experience:

> *I grew a reporting framework once over the course of a few hours, and observers were absolutely certain it was a trick. I must have started with the resulting framework in mind. No, sorry. I've just been test driving development long enough that I can recover from most of my mistakes faster than you can recognize I've made them.*
>
> —Kent Beck, "Test-Driven Development by Example"

Sounds good, now what?

Maybe you're intrigued, or inspired, by the idea of programming with confidence, guided by tests. But maybe you still have a few questions. For example:

- How does Go's `testing` package work? How do we write tests to communicate intent? How can we test error handling? How do we come up with useful test data? What about the test cases we didn't think of?
- What if there are bugs still lurking, even when the tests are passing? How can we test things that seem untestable, like concurrency, or user interaction? What about command-line tools?

- How can we deal with code that has too many dependencies, and modules that are too tightly coupled? Can we test code that talks to external services like databases and network APIs? What about testing code that relies on time? And are mock objects a good idea? How about assertions?
- What should we do when the codebase has no tests at all? For example, legacy systems? Can we refactor and test them in safe, stress-free ways? What if we get in trouble with the boss for writing tests? What if the existing tests are no good? How can we improve testing through code review?
- How should we deal with tests that are optimistic, persnickety, over-precise, redundant, flaky, failing, or just slow? And how can tests help us improve the design of the system overall?

Well, I'm glad you asked. Enjoy the rest of the book.