THE POWER OF GO TOOLS

“Superb and well written”
—Lee Gibson

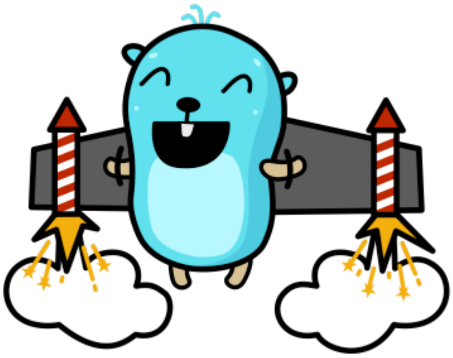
BUILDING DELIGHTFUL SOFTWARE IN GO

JOHN ARUNDEL

7. Commands

This is the Unix philosophy: Write programs that do one thing and do it well. Write programs to work together. Write programs to handle text streams, because that is a universal interface.

—Doug McIlroy, quoted in Peter Salus's "A Quarter Century of Unix"



In previous chapters we've used Go to create executable commands that users can run on their systems, such as `words` or `lines`. That's great, but what if we wanted to run such a command from within a Go program itself? What would that look like?

The `exec` package

For example, suppose we want to run the command `ls` to list the files in the current directory. When we ask the shell to run that command, it makes a call to the operating system kernel to start a new process. There's more work involved in this than you might think.

What even is a process?

The kernel first allocates some memory to store the process's context and other housekeeping information about it. Then it *loads* the specific executable file we requested (something like `/bin/ls`), which means copying its bytes from disk into memory. Finally it starts *executing* the machine code instructions at the beginning of the program.

So the kernel-level API for running a command needs to take at least the path to the executable, and in practice there are a few more arguments.

On Unix-like systems, there's some *environment* (a set of key-value pairs storing information the program might need) for the process, possibly some *command-line arguments*, and also some *file descriptors*: byte streams allowing the program to read input and write output in a standard way.

This sounds like a lot of paperwork, and while we *could* do it (using the `os.StartProcess` standard library call), we'd prefer not to. So it's good to know that there's a higher-level API provided by the `os/exec` package.

Let's see how that works in a Go program with our `ls` example:

```go
package main

import (
    "os/exec"
)

func main() {
    cmd := exec.Command("/bin/ls")
    cmd.Run()
}
```

First we call `exec.Command` with the path to some executable (`/bin/ls`). If the `ls` executable is in a different place on your system, you'll need to use a different path here (use `whereis ls` to find it).

This returns a value of type `*exec.Cmd`, and we assign it to the variable `cmd`. Note that nothing has actually *happened* so far: we haven't started a new process yet, just created the abstraction we'll use to control it.

Calling the command's `Run` method is what actually causes the kernel to create a new process, load the `ls` executable, and run it. If we run this Go program, then, we should see just one file listed: the `main.go` file containing its source code. Let's try.

```
go run main.go
```

Hmm. Nothing happened. In fact, the `ls` command worked perfectly well, but we have no way of knowing that, because we didn't see its output.

Managing command output

We said earlier that the Unix process model includes input and output file descriptors: the equivalent of `os.Stdin` and `os.Stdout` (and `os.Stderr`) in Go. And an `exec.Cmd` has fields for these to be attached, but we didn't attach anything, so the output of `ls` went... nowhere.

We'll need to attach something to `cmd.Stdout` in order to see that output. If we attach `os.Stdout`, for example, we should see it printed to the terminal:

```go
func main() {
    cmd := exec.Command("/bin/ls")
    cmd.Stdout = os.Stdout
    cmd.Run()
}
```

```
go run main.go
```

```
main.go
```

That's more like it! In practice, when we're running external commands from a Go program, we usually want to capture that output somehow: for example, in a `bytes.Buffer`. But this is enough to prove the concept.

Can we do more? How about passing arguments to the command, for example?

It turns out that `exec.Command` is variadic: it takes any number of strings after the first, and interprets them as arguments to be passed to the command.

```go
func main() {
    cmd := exec.Command("/bin/ls", "-l", "main.go")
    cmd.Stdout = os.Stdout
    cmd.Run()
}
```

Here's the output:

```
-rw-r--r--  1 john  staff  127 30 Sep 15:46 main.go
```

When not to use `exec`

Although it makes a neat demo, we wouldn't actually want to run the `ls` command from a Go program: that functionality is much better implemented in Go itself, and we've seen how to do it in the chapter on filesystems.

Similarly, there's no benefit in *shelling out* (that is, invoking a subprocess) to run standard file-management commands such as `mkdir`, `touch`, `rm`, and so on. All of these things are either already available in the standard library, or easy to write in Go. Don't shell out to an external command whose functionality is easily replicated in native Go code. That just adds an unnecessary dependency.

Suppose we wanted to do something a bit more advanced, though, such as triggering a Kubernetes deployment with `kubectl`, or creating a cloud virtual machine with the `gcloud`, `aws`, or `az` commands. Would this be a good case for running external commands from Go?

Not really. Apart from the dependency issue that we already discussed, complicated programs like `kubectl` tend to change over time. New flags and verbs are added, old ones deprecated, and the behaviour of the program can change radically.

A new version of `kubectl` could easily cause your Go program to stop working properly. And since we usually want the output from such tools, we have to *parse* it to get the information we need, and that's equally fragile.

Tiny changes in the output format of external commands can completely break programs that rely on them, and this kind of thing isn't easy to test automatically.

A better approach is to look for a Go package that provides the same, or similar functionality to the command-line tool. Often, the CLI tool itself will use that package to implement what it does.

For example, this is the complete `main` function for the `kubectl` tool, which I'm sure we will all agree does a lot:

```go
func main() {
    command := cmd.NewDefaultKubectlCommand()
    if err := cli.RunNoErrOutput(command); err != nil {
        // Pretty-print the error and exit with an error.
        util.CheckErr(err)
    }
}
```

(`kubectl.go`)

It doesn't look like there's much code here, but that's the point: this minimal `main` function just calls into the accompanying `cli` package to do the actual work. And if `kubectl` can use that package, so could we.

All the important machinery of `kubectl`, then, is in Go packages that we can import and use in our own programs, which is sensible. Anything the `kubectl` binary can do, we can do too, using pure Go, and without fragile dependencies on external commands.

We've worked hard throughout this book to design programs with a "minimal `main`", and now that we're looking at other people's programs from the user's point of view, we can see why that's so valuable. `main` can't be imported into any program, so it shouldn't do anything that might be useful to others.

When to use `exec`

There are legitimate use cases for running external commands from Go. For example, the *purpose* of the Go program might be to run such a command, perhaps to automate some process that's currently done manually.

It may not be practical or desirable to completely replace this command with a Go program, or at least not yet, so the compromise option is to execute it from Go for the time being.

And there isn't always a convenient Go SDK or package that we can use to get the same functionality that the command provides. For example, the command might be written in C, or use a C shared library. If this is the only implementation of the necessary behaviour that exists, then we have no choice but to use it, or write our own.

While we can use the `cgo` interoperability layer to call C functions from Go, that's always a last resort, and greatly complicates our programs. In such a case, it would be better to use `exec` to run some command instead, despite the disadvantages that we've already discussed.

Migrating from shell scripts to Go

It's sometimes said that the internet is held together with shell scripts and duct tape, and it's certainly true that a great many organisations depend on some rather fragile and ancient (and almost certainly buggy) shell scripts. While the shell is a great tool for one-shot tasks and rapid prototyping, it's not ideal for developing robust, maintainable software over the long term.

Why use Go to run commands?

It's possible to write and maintain relatively robust shell programs, but it's uphill work. Just use Go instead. Go is a much better language for this job, so many important programs that were once made of duct tape are now being migrated to Go.

When the program is a script that simply executes a bunch of Unix commands, for example, the first step on the migration path may well be to replace it with a Go program that uses `exec` to run the same bunch of commands.

That may not sound like a big improvement, but we can do a lot from this position. We can write unit tests, and use Go for things like text processing, instead of relatively awkward `grep`, `awk`, and `sed` commands.

I yield to no one in my admiration of things like `awk`, and even Perl: they're terrific at doing what they do. What they *don't* necessarily do well is communicate data to and from other tools, except in the very simple byte-oriented way provided by the Unix API.

In other words, we can pipe lines of text in and out of tools, but we can't easily do complex filtering, logic, or manipulation. That's a job for a programming language. Luckily, we have one.

A command wrapper in Go

When we need the behaviour provided by proprietary or other closed-source programs, especially if they're only available as executable binaries, we may have no alternative but to use `exec`.

The `pmset` command

For example, suppose we want to get the current battery charge status on a Mac. We can do this easily on the command line with the OS-specific `pmset` command:

```
pmset -g ps
```

```
Now drawing from 'AC Power'
 -InternalBattery-0 (id=10879075) 98%; charging;
    0:42 remaining present: true
```

Getting this information from a Go program is not so straightforward, though. How does `pmset` do it? Looking at the C source code for this command, we find that it gets the data by making a system call to the macOS kernel.

While we *could* make the same system call from a Go program, this is likely to involve a lot of paperwork; probably more than we'd care to do just to get the battery status.

Although we don't want to use external commands when it's easy to replicate their functionality in Go, that's not the case here. Running the `pmset` command makes our Go program much simpler than it would be if we tried to implement the same behaviour ourselves.

Since the `pmset` command is part of macOS, it's fairly safe to assume we'll be able to execute it on any Mac. To put it another way, executing the command is no *less* portable or reliable than making the equivalent system call. And the output doesn't look too difficult to parse.

What can we test?

Let's see how to approach writing a `pmset` *wrapper* in Go. This will be a Go package that users can import and call some function to get the current battery status, without having to worry about how that actually works under the hood.

If you're not using macOS, just substitute some equivalent command, such as `acpi` on Linux, or `powercfg` on Windows, and make the appropriate tweaks to the code as you go. The logic will be roughly the same whatever command you're using.

Go ahead and create a new folder for this project (you might call it `battery`, for example). Now, what's the first test we should write?

Suppose what we want is some Go function that will give us the current charge status of the battery. You might find it difficult to think of any test that you could write for this: how can you know in advance what the correct battery charge status will *be*?

Any function that returns dynamic information based on some external conditions can't be tested by comparing its result directly against expectation. We need to think harder.

Throughout this book we've been asking the question: *what behaviour are we really testing?* So what's the answer here?

It's tempting to answer "The function correctly gets the battery status", but that's wrong. The "getting the battery status" behaviour isn't actually *in* our code; it's in the `pmset` command. So what does *our* code do, then?

Breaking down behaviour into chunks

The behaviour we're really testing in this case is in two parts:

1. We execute the `pmset` command with the correct arguments
2. We correctly parse the `pmset` output to get the battery status

When you frame it this way, it's easier to see how to test these two chunks of behaviour, isn't it? The "output" of the first one is simply a string representing the appropriate `pmset` command line.

The "input" of the second behaviour is some string representing the kind of text that `pmset` prints out when you run it, and its output is whatever battery information we manage to parse from it.

In between these two steps there's some hidden stuff that's outside our code: basically, everything `pmset` does. And we're not interested in testing `pmset`, so we can focus entirely on the two things *our* code does.

In this case, the first behaviour is trivial, since the required command line is always the same: it's just `pmset -g ps`. No need to test that we can produce this string. We can imagine situations where this could be more complicated, and would need testing, but let's consider this one solved for now.

An otherwise hard-to-test function can often be broken down into sub-behaviours, each of which can be refactored out to a testable function. We can then trivially *compose* these behaviours into a single function that users can call.

Parsing command output

The second behaviour needs a little more thought. First, we'll need some test input. The best test data is always the *real* data, so let's use `pmset` to generate it, and *pipe* the output to a file in the `testdata` folder.

```
mkdir testdata
```

```
pmset -g ps >testdata/pmset.txt
```

As we saw earlier, the result will be something like this:

```
Now drawing from 'AC Power'
 -InternalBattery-0 (id=10879075) 98%; charging; 0:42
remaining present: true
```

So that's the input to our parsing function; what output do we want? Well, there are several useful pieces of information we *could* glean from this text, such as the AC power status, the battery ID, its charge percentage, and the charging time remaining.

That's too much fun for one chapter. Let's just focus on the charge percentage for now: if we can extract that, presumably we can extract the other things the same way. We'd like to avoid being tempted into solving more problems than we strictly need to.

Without worrying yet about exactly *how* we're going to get the charge number, we can certainly write a test for the function that does it. Take a minute and try to sketch out a suitable test.

GOAL: Write a test for the "parse `pmset` output for battery charge percentage" behaviour. Use the real output from `pmset -g ps` as your test data.

HINT: First, what input will the function need to take? Well, what we'll *have* is a string, since the output from `pmset` is plain text, so a string parameter sounds reasonable.

What about outputs? The most useful thing would be a *number* representing the battery charge percentage, so we could have the function return an `int`. `pmset` doesn't report fractional percentages, so `int` will be fine.

But, thinking ahead a little, we can see that there might be other information we'll want to extract from the data in future. Can we leave room in our API for this?

We'd like to avoid changing the signature of our function, as this would break any user code that calls it. Instead, it'll be convenient to have some struct type representing all the information we want to know about the battery status. We can always add new fields to it, without making breaking changes to our public API.

So let's define a `want` variable of this struct type, whose only field (so far) is the charge percentage. To be unambiguous, then, let's name it `ChargePercent`.

And the rest should be straightforward: call the function, and compare the result we got against `want` in our now-familiar way.

Testing the parsing function

SOLUTION: Here's my attempt at this:

```go
func TestParsePmsetOutput_GetsChargePercent(t *testing.T) {
    t.Parallel()
    data, err := os.ReadFile("testdata/pmset.txt")
    if err != nil {
        t.Fatal(err)
    }
    want := battery.Status{
        ChargePercent: 98,
    }
    got, err := battery.ParsePmsetOutput(string(data))
    if err != nil {
        t.Fatal(err)
    }
    if !cmp.Equal(want, got) {
        t.Error(cmp.Diff(want, got))
    }
}
```

(Listing battery/1)

This looks pretty much like most of the other tests we've written, which makes it straightforward to understand. We read the test data, call our parse function, and compare `want` against `got`.

What's `want`? Our `pmset` output text shows a charge percentage of 98, so that's the `ChargePercent` we should expect.

`ParsePmsetOutput` needs to return an `error` as well as the `Status` struct, because clearly parsing *can* fail: we might not be able to make sense of the input if its format has changed, for example.

Parsing command output

Let's write a null implementation of `ParsePmsetOutput` and check that we get the expected test failure:

```
-       ChargePercent: 98,
+       ChargePercent: 0,
```

Good. This time, the test was the easy bit, and the *implementation* might be more challenging. Let's see.

GOAL: Implement `ParsePmsetOutput`.

Clarifying the problem

HINT: This needs a little thought. Let's clarify the problem statement. We're expecting input similar to our test data:

```
Now drawing from 'AC Power'
 -InternalBattery-0 (id=10879075) 98%; charging; 0:42
remaining present: true
```

And we want to be able to extract the charge percentage: in this case, the value 98. How could we do that?

It's not a case of simple string matching, unfortunately. We don't know the value in advance, so we can't search for it.

The value isn't always at the same byte position in the text, either, so we can't just read bytes 63-64, or something like that. We can't even rely on counting words to get to the text we want, since presumably the part about "drawing from 'AC Power' " could change to something else.

What could we do?

SOLUTION: Some programmers, when confronted with a problem like this, think "I know, I'll use regular expressions!" (Now they have two problems.)

What kind of regular expression would work here? Well, the thing we want is a *sequence of digits*, but there's more than one such sequence in our example. Can we narrow it down a bit more?

Writing a regular expression

On closer inspection, it seems that there's only *one* digit sequence followed by a % character, so we can try to match that.

Here's a regular expression that will match one or more digits followed by a %:

```
[0-9]+%
```

What can we do with that in Go? Here's an example:

```
r := regexp.MustCompile("[0-9]+%")
result := r.FindString("the charge is 98%")
```

Our real string is different, but this simpler version will make it easier to see what's going on. First, we *compile* the regular expression, which is like a mini-program, to get some value `r` that we can use. Then, we call its `FindString` method with our input text.

`FindString` returns the first string that completely matches the specified regular expression, so in this case that's:

```
98%
```

Great, but we only want the 98, not the trailing %. We need to *match* the % character, to make sure we've got the right digit sequence. But once we *have* it, we then want to extract only the digits from it.

We can do this by adding parentheses to our regular expression, to create a *capturing group*:

```
([0-9]+)%
```

If this expression matches a chunk of text, then we'll be able to extract just the contents of the capturing group: the part inside parentheses. Here's what that looks like:

```
r := regexp.MustCompile("([0-9]+)%")
matches := r.FindStringSubmatch("the charge is 98%")
```

Whereas `FindString` only returns the entire matched string, `FindStringSubmatch` returns a slice of strings. The first element of this slice is the entire matched string, as before, but the second is the contents of the capturing group.

If we had more groups in the regular expression, then there would be extra elements in this slice, each holding the contents of the corresponding group. But as it happens, we only have one group, so the `matches` slice contains exactly two elements:

```
["98%", "98"]
```

The second element is the one we want, and we can use the index expression `matches[1]` to get it. Once we have that string, we feel confident that we can use something like `strconv.Atoi` to turn it into an integer.

Now we have a plan. So how can we put all this together to implement `ParsePm-setOutput`?

Using the regexp

It happens that the `MustCompile` operation is relatively expensive, compared to the actual string matching. But compilation only needs to be done once per program run, so we can do it at package level, in a `var` statement.

If the call to `MustCompile` were instead inside our parsing function, then the regexp would be recompiled every time the function is called, which is unnecessary. Once we *have* the compiled value, we can use it to match against any number of input strings relatively cheaply.

Here's my version of the function, then:

```
var pmsetOutput = regexp.MustCompile("([0-9]+)%")

func ParsePmsetOutput(text string) (Status, error) {
    matches := pmsetOutput.FindStringSubmatch(text)
    if len(matches) < 2 {
        return Status{}, fmt.Errorf("failed to parse pmset \
            output: %q", text)
    }
    charge, err := strconv.Atoi(matches[1])
    if err != nil {
        return Status{}, fmt.Errorf("failed to parse charge \
            percentage: %q", matches[1])
    }
    return Status{
        ChargePercent: charge,
    }, nil
}
```

(Listing battery/1)

Having used the compiled `pmsetOutput` regexp to test the input for matches, we now need to check whether or not it actually matched. If it did, we should have the contents of the capturing group, so there will be at least two elements in `matches`. We can check `len(matches)` and return an error if this isn't the case.

Note that we include the output that we couldn't parse: just saying "failed to parse" would be no help, either to the user or the developer. Including the unparsable text in the error message takes only a few more keystrokes, but makes a big difference to the usefulness of the error.

Now we know that we successfully extracted the string of digits we need, we can call `strconv.Atoi` to turn it into an integer value. This returns `error`, because not all strings represent valid integers. We feel *pretty* sure that ours will, but again, let's not take chances. So we check that error too.

Finally, we have the `int` value we need, so we can construct a suitable `Status` literal and set its `ChargePercent` field to the value we calculated.

Let's try the test again:

```
PASS
```

Mischief managed! We can now feel confident that, given some genuine `pmset` output, we can extract the battery status from it.

We can turn our attention, then, to the other part of the task: *getting* that output in the first place. We can imagine some function `GetPmsetOutput` that runs `pmset` using `exec` and returns its output as a string.

But there's a problem. If we're not allowed to execute external commands in a unit test, how can we possibly write a unit test for such a function?

We can't, clearly. But unit tests aren't the only kind of tests we can write.

Integration tests

A unit test, as the name implies, tests some *unit* of your code, such as a function, usually in isolation. Its job is to verify that the function's *logic* is correct, so it tries to avoid using any external dependencies, such as commands.

The job of an *integration* test, on the other hand, is to test what happens when we *do* use those external dependencies. It validates our assumptions about how they work: for example, that we execute the right command in the right way.

Why isolate integration tests?

Why make a distinction between these two kinds of tests? Well, unit tests need to be fast and lightweight, because we run them very often, and the only time they should fail is when something's wrong with our code.

We don't need to run integration tests so often, though, because the only way they could break is if something external changed: the `pmset` command was updated or removed, for example.

Unit tests check a program's behaviour given certain assumptions about external dependencies. Integration tests check those assumptions are still *correct*. So it doesn't matter if integration tests are relatively slow, or use external dependencies, as long as we can figure out a way to avoid running them every time we run unit tests.

Build tags

One common way to control the execution of integration tests is to use a *build tag*.

A build tag is a special comment in a Go file that prevents it from being compiled unless that tag is defined. There are some *predefined* build tags: for example, the tag `windows` is defined if we're building on Windows. Similarly, `darwin` and `linux` indicate macOS and Linux.

Here's what specifying a build tag looks like in a Go file. It needs to be the first line in the file, and be followed by a blank line before the `package` declaration:

```go
//go:build darwin

package ...
```

We could use this mechanism to provide OS-specific implementations of a certain piece of code, for example. We would write the macOS version in a Go file protected by the `darwin` build tag, the Linux version protected by `linux`, and so on.

But we can also define arbitrary build tags of our own. Suppose we put the following at the beginning of a new Go file:

```go
//go:build integration

package battery_test
```

Now this file will be ignored by the Go tools unless the `integration` tag is defined, and under normal circumstances it won't be, because it's not one of the predefined tags. If the file contains tests, they won't be run by the `go test` command. If it contains implementations, they won't be built by `go build`, and so on.

So suppose we write some test for `GetPmsetOutput` and we put it in its own Go file, protected by the `integration` build tag. It won't be run by `go test`, which is what we want, but then how *do* we run it when we want to?

The answer is that you can supply arbitrary build tags to the `go test` command, so we would run something like this:

```
go test -tags=integration
```

Now that tag is defined, so the file will become visible to Go, and the test will be run. But if we don't supply that tag on the command line, it won't.

Testing the command runner

What integration test should we write? We know we're going to call `GetPmsetOutput`, and that that function will execute the `pmset` command. Fine. So what can we test about it?

Well, another way to phrase that question is to ask "What could fail?" Clearly, running the command could fail for a number of reasons, most likely that the command doesn't exist or has a different path.

If we ran this test on some non-macOS machine, for example, it would presumably fail. So that's one thing we can check. In that case we'd get an error something like this:

```
fork/exec /usr/bin/pmset: no such file or directory
```

If this happens in the test, we could do some work to make the test failure friendlier. For example, we could check the value of `runtime.GOOS`, which tells us what operating system Go thinks it's running on. If it's something other than `darwin`, we could report a failure like "This test will only work when run on macOS and when `/usr/bin/pmset` is available."

That's getting a bit fancy, though, so for now we'll just skip the test, using `t.Skip`, if we can't run the `pmset` command for whatever reason.

There's one other sanity check we need. Even if `pmset` is available, on machines without batteries (a Mac mini, for example), it won't report anything useful to us. So if we don't find the string `InternalBattery` in the output from `pmset`, we should also skip this test.

Now it's reasonably safe to call the function under test, so what should we test about it? `GetPmsetOutput` returns a string, so we could test that the string isn't empty, but that doesn't prove much. Can we do more? We don't know in advance what the string actually *is*, so we can't simply compare it against a string literal.

But there's something else we *can* do with it. We can pass it to `ParsePmsetOutput`.

After all, if we executed the pmset command correctly, then its output *should* be parseable without error, shouldn't it?

So we can check the error value, but is there anything useful we can test about the resulting Status struct? Not really, it turns out.

We *could* look at the ChargePercent field and see if it has the default value 0, indicating it hasn't been set. But the battery might really *be* 0% charged, in which case we'd get a bogus test failure for code that's actually correct.

In any case, we don't actually need to check the result at all. If ParsePmsetOutput doesn't return error, then the result is valid by definition. And we know that function works, because it has its *own* test.

We're not *testing* the parsing here, just using it to check that pmset returned something parseable. If this is indeed the case, then we feel we can't have messed up *too* badly in executing it. So we've done enough for now.

Here's the test, then:

```go
//go:build integration

package battery_test

import (
	"testing"

	"github.com/bitfield/battery"
)

func TestGetPmsetOutput_CapturesCmdOutput(t *testing.T) {
	t.Parallel()
	data, err := exec.Command("/usr/bin/pmset", "-g", "ps").
		CombinedOutput()
	if err != nil {
		t.Skipf("unable to run 'pmset' command: %v", err)
	}
	if !bytes.Contains(data, []byte("InternalBattery")) {
		t.Skip("no battery fitted")
	}
	text, err := battery.GetPmsetOutput()
	if err != nil {
		t.Fatal(err)
	}
```

```
    status, err := battery.ParsePmsetOutput(text)
    if err != nil {
        t.Fatal(err)
    }
    t.Logf("Charge: %d%%", status.ChargePercent)
}
```

(Listing battery/1)

Even if there's no error from parsing the pmset output, it would be nice to *know* what value is actually being parsed, so we pass that to t.Logf. The output from this won't be printed unless the test fails, or unless we run go test with the -v flag to enable verbose mode.

Running the command

If we supply a null implementation of GetPmsetOutput, we should be able to run this test and see it fail:

```
go test -tags=integration
```

```
--- FAIL: TestGetPmsetOutput_CapturesCmdOutput (0.00s)
    battery_integration_test.go:19: failed to
        parse pmset output: ""
```

Capturing output

Good. Now, how do we implement GetPmsetOutput? You already know how to use exec.Command to create a command object, and Run to run it. How can we get its output as a Go value?

In our ls example, we set the Stdout field on the command to send its output somewhere. We could use something like a strings.Builder to capture it, but there's an easier way.

The command object has an Output method that runs the command and returns its output as a string. In fact, it'll be handy to get the standard error stream, too.

We can get both standard output and standard error in one string, by calling CombinedOutput. Here's what that looks like:

```
func GetPmsetOutput() (string, error) {
    data, err := exec.Command("/usr/bin/pmset -g ps").
        CombinedOutput()
    if err != nil {
        return "", err
    }
    return string(data), nil
}
```

(Listing battery/1)

This should pass the test, so let's try:

```
go test -tags=integration
```

```
fork/exec /usr/bin/pmset -g ps: no such file or directory
```

Oops. The command line is correct, as a single string, but that's not actually how you *pass* it to `exec.Command`.

It needs to be broken up into individual strings, one for each command-line argument:

```
data, err := exec.Command("/usr/bin/pmset", "-g", "ps").
    CombinedOutput()
```

Good thing we wrote that integration test, or we might not have spotted that bug until we ran the program for real.

Let's try again:

```
go test -tags=integration -v
```

```
...
    battery_integration_test.go:21: Charge: 100%
--- PASS: TestGetPmsetOutput_CapturesCmdOutput (0.02s)
```

Not only does this pass, but thanks to the `-v` flag, we can spy on the charge percentage value we passed to `t.Logf`:

```
Charge: 100%
```

There's more we could do with this package, of course, but it's a good start. Some programmer out there will be glad that we provided a straightforward way for them to get the battery status in Go code. Otherwise, they'd have to do what we did: work out what command to run, exactly how to invoke it, parse its output, and so on.

When to import a third-party package

It's worth saying a word or two about what to do when you have some problem that isn't trivially solved by the standard library, like the battery status example. This happens a lot, so how should we think about the right way to proceed?

The first thing would be to look for some existing third-party package (on `pkg.go.dev`, for example). If there's already a pure Go package that does exactly what we need, then that's wonderful—providing its licence allows us to use it, naturally.

It sometimes happens, though, that the best we can find is something that's *close* to what we want, but not a perfect match. We may need to do a fair bit of paperwork to get the package to solve our problem, and at that point it's worth questioning whether we should import the package at all.

If we can import a package to save writing ten lines of Go code, but it takes ten lines to *use* the package, then there's no real benefit from importing it. It might be better just to copy the piece of code that we need directly into our program.

If importing some package makes your program simpler, in other words, then import it. If it doesn't, don't.

We need to keep in mind that imports aren't free. Every extra dependency for your program complicates the code, slows the build process, and adds to the list of things that could break your build for one reason or another.

Maybe the package pushed a breaking upgrade, or introduced a critical bug, or maybe it was just deleted altogether. Every import is a potential point of failure.

Even if we're copying code rather than importing it, we're still introducing a foreign object into our program that could contain bugs. If it doesn't have its own tests that we can copy, we should cover it with tests.

We can't just take the attitude that if something is on GitHub or StackOverflow, it must be fine. That's very much not the case, as a brief inspection of those sources will confirm.

Go adoption is growing rapidly, after all, so statistically most Go programmers must be relative beginners. By extension, most Go code we find in the wild won't be all that good.

That doesn't mean we should never use "found code": it just means we shouldn't assume it's necessarily a model of good Go style, or even that it works at all.

Going further

If you feel excited and inspired to write some interesting Go programs that execute external commands, or even if you don't, here's one suggestion.

- Write a command in Go that times how long it takes to execute some *other* command. For example, you could use it like this:

  ```
  howlong sleep 1
  (time: 1.007s)

  howlong backup.sh
  ...
  (time: 1h14m2s)
  ```

Make sure the substantive functionality is part of an importable package, so other people can use your code in their own programs.

If you get stuck, take a sneak peek at my suggested solution in listing `howlong/1`.