Introduction

Writing good tests is not trivial, but in many situations a lot of ground can be covered with table-driven tests: Each table entry is a complete test case with inputs and expected results, and sometimes with additional information such as a test name to make the test output easily readable. If you ever find yourself using copy and paste when writing a test, think about whether refactoring into a table-driven test or pulling the copied code out into a helper function might be a better option.

Given a table of test cases, the actual test simply iterates through all table entries and for each entry performs the necessary tests. The test code is written once and amortized over all table entries, so it makes sense to write a careful test with good error messages.

Table driven testing is not a tool, package or anything else, it's just a way and perspective to write cleaner tests.

Example of a table driven test

Here is a good example from the testing code for the `fmt` package (https://pkg.go.dev/fmt/):

```
var flagtests = []struct {
    in  string
    out string
}{
    {"%a", "[%a]"},
    {"%-a", "[%-a]"},
    {"%+a", "[%+a]"},
    {"%#a", "[%#a]"},
    {"% a", "[% a]"},
    {"%0a", "[%0a]"},
    {"%1.2a", "[%1.2a]"},
    {"%-1.2a", "[%-1.2a]"},
    {"%+1.2a", "[%+1.2a]"},
    {"%-+1.2a", "[%+-1.2a]"},
    {"%-+1.2abc", "[%+-1.2a]bc"},
    {"%-1.2abc", "[%-1.2a]bc"},
}
func TestFlagParser(t *testing.T) {
    var flagprinter flagPrinter
    for _, tt := range flagtests {
        t.Run(tt.in, func(t *testing.T) {
            s := Sprintf(tt.in, &flagprinter)
            if s != tt.out {
                t.Errorf("got %q, want %q", s, tt.out)
            }
        })
    }
}
```

Note the detailed error message provided with `t.Errorf` : its result and expected result are provided; the input is the subtest name. When the test fails it is immediately obvious which test failed and why, even without having to read the test code.

A `t.Errorf` call is not an assertion. The test continues even after an error is logged. For example, when testing something with integer input, it is worth knowing that the function fails for all inputs, or only for odd inputs, or for powers of two.

Parallel Testing

Parallelizing table tests is simple, but requires precision to avoid bugs. Please note closely the three changes below, especially the re-declaration of `tt`

```
package main

import (
    "testing"
)

func TestTLog(t *testing.T) {
    t.Parallel() // marks TLog as capable of running in parallel with other tests
    tests := []struct {
        name string
    }{
        {"test 1"},
        {"test 2"},
        {"test 3"},
        {"test 4"},
    }
    for _, tt := range tests {
        tt := tt // NOTE: https://github.com/golang/go/wiki/CommonMistakes#using-goroutines-on-loop-iterator-variables
        t.Run(tt.name, func(t *testing.T) {
            t.Parallel() // marks each test case as capable of running in parallel with each other
            t.Log(tt.name)
        })
    }
}
```

References

- https://go.dev/doc/code#Testing
- https://go.dev/doc/faq#assertions
- https://go.dev/doc/faq#testing_framework
- https://pkg.go.dev/testing/