Go Test Comments

This page is a supplement to Go Code Review Comments, but is targeted specifically to test code.

Please discuss changes before editing this page, even *minor* ones. Many people have opinions and this is not the place for edit wars.

Assert Libraries

Avoid the use of 'assert' libraries to help your tests. Go developers arriving from xUnit frameworks often want to write code like:

```
assert.isNotNil(t, "obj", obj)
assert.stringEq(t, "obj.Type", obj.Type, "blogPost")
assert.intEq(t, "obj.Comments", obj.Comments, 2)
assert.stringNotEq(t, "obj.Body", obj.Body, "")
```

but this either stops the test early (if assert calls `t.Fatalf` or `panic`) or omits interesting information about what the test got right. It also forces the assert package to create a whole new sub-language instead of reusing the existing programming language (Go itself). Go has good support for printing structures, so a better way to write this code is:

```
if obj == nil || obj.Type != "blogPost" || obj.Comments != 2 || obj.Body == "" {
    t.Errorf("AddPost() = %+v", obj)
}
```

Assert libraries make it too easy to write imprecise tests and inevitably end up duplicating features already in the language, like expression evaluation, comparisons, sometimes even more. Strive to write tests that are precise both about what went wrong and what went right, and make use of Go itself instead of creating a mini-language inside Go.

Choose Human-Readable Subtest Names

When you use `t.Run` to create a subtest, the first argument is used as a descriptive name for the test. To ensure that test results are legible to humans reading the logs, choose subtest names that will remain useful and readable after escaping. (The test runner replaces spaces with underscores, and it escapes non-printing characters).

To identify the inputs, use `t.Log` in the body of the subtest or include them in the test's failure messages, where they won't be escaped by the test runner.

Compare Full Structures

If your function returns a struct, don't write test code that performs an individual comparison for each field of the struct. Instead, construct the struct that you're

expecting your function to return, and compare in one shot using diffs or deep comparisons. The same rule applies to arrays and maps.

If your struct needs to be compared for approximate equality or some other kind of semantic equality, or it contains fields that cannot be compared for equality (e.g. if one of the fields is an `io.Reader`), tweaking a `cmp.Diff` or `cmp.Equal` comparison with cmpopts options such as `cmpopts.IgnoreInterfaces` may meet your needs (example); otherwise, this technique just won't work, so do whatever works.

If your function returns multiple return values, you don't need to wrap those in a struct before comparing them. Just compare the return values individually and print them.

Compare Stable Results

Avoid comparing results that may inherently depend on output stability of some external package that you do not control. Instead, the test should compare on semantically relevant information that is stable and resistent to changes in your dependencies. For functionality that returns a formatted string or serialized bytes, it is generally not safe to assume that the output is stable.

For example, `json.Marshal` makes no guarantee about the exact bytes that it may emit. It has the freedom to change (and has changed in the past) the output. Tests that perform string equality on the exact JSON string may break if the `json` package changes how it serializes the bytes. Instead, a more robust test would parse the contents of the JSON string and ensure that it is semantically equivalent to some expected data structure.

Equality Comparison and Diffs

The `==` operator evaluates equality using the language-defined comparisons. Values it can compare include numeric, string, and pointer values and structs with fields of those values. In particular, it determines two pointers to be equal only if they point to the same variable.

Use the cmp package. Use `cmp.Equal` for equality comparison and `cmp.Diff` to obtain a human-readable diff between objects.

Although the `cmp` package is not part of the Go standard library, it is maintained by the Go team and should produce stable results across Go version updates. It is user-configurable and should serve most comparison needs.

You will find older code using the standard `reflect.DeepEqual` function to compare complex structures. Prefer `cmp` for new code, and consider updating older code to use `cmp` where practical. `reflect.DeepEqual` is sensitive to changes in unexported fields and other implementation details.

NOTE: The `cmp` package can also be used with protocol buffer messages, by

including the `cmp.Comparer(proto.Equal)` option when comparing protocol buffer messages.

Got before Want

Test outputs should output the actual value that the function returned before printing the value that was expected. A usual format for printing test outputs is "`YourFunc(%v) = %v, want %v`".

For diffs, directionality is less apparent, and thus it is important to include a key to aid in interpreting the failure. See Print Diffs.

Whichever order you use in your failure messages, you should explicitly indicate the ordering as a part of the failure message, because existing code is inconsistent about the ordering.

Identify the Function

In most tests, failure messages should include the name of the function that failed, even though it seems obvious from the name of the test function.

Prefer:

```
t.Errorf("YourFunc(%v) = %v, want %v", in, got, want)
```

and not:

```
t.Errorf("got %v, want %v", got, want)
```

Identify the Input

In most tests, your test failure messages should include the function inputs if they are short. If the relevant properties of the inputs are not obvious (for example, because the inputs are large or opaque), you should name your test cases with a description of what's being tested, and print the description as part of your error message.

Do not use the index of the test in the test table as a substitute for naming your tests or printing the inputs. Nobody wants to go through your test table and count the entries in order to figure out which test case is failing.

Keep Going

Even after your test cases encounter a failure, they should keep going for as long as possible in order to print out all of the failed checks in a single run. This way, someone who's fixing the failing test doesn't have to play whac-a-mole, fixing one bug and then re-running the test to find the next bug.

On a practical level, prefer calling `t.Error` over `t.Fatal`. When comparing several different properties of a function's output, use `t.Error` for each of those comparisions.

`t.Fatal` is usually only appropriate when some piece of test setup fails, without which you cannot run the test at all. In a table-driven test, `t.Fatal` is appropriate for failures that set up the whole test function before the test loop. Failures that affect a single entry in the test table, which make it impossible to continue with that entry, should be reported as follows:

- If you're not using `t.Run` subtests, you should use `t.Error` followed by a `continue` statement to move on to the next table entry.
- If you're using subtests (and you're inside a call to `t.Run`), then `t.Fatal` ends the current subtest and allows your test case to progress to the next subtest, so use `t.Fatal`.

Mark Test Helpers

A test helper is a function that performs a setup or teardown task, such as constructing an input message, that does not depend on the code under test.

If you pass a `*testing.T`, call `t.Helper` to attribute failures in the test helper to the line where the helper is called.

```
func TestSomeFunction(t *testing.T) {
  golden := readFile(t, "testdata/golden.txt")
  // ...
}

func readFile(t *testing.T, filename string) string {
  t.Helper()

  contents, err := ioutil.ReadFile(filename)
  if err != nil {
    t.Fatal(err)
  }

  return string(contents)
}
```

Do not use this pattern when it obscures the connection between a test failure and the conditions that led to it. Specifically, `t.Helper` should not be used to implement assert libraries.

Print Diffs

If your function returns large output then it can be hard for someone reading the failure message to find the differences when your test fails. Instead of printing

both the returned value and the wanted value, make a diff.

Add some text to your failure message explaining the direction of the diff.

Something like "`diff -want +got`" is good when you're using the `cmp` package (if you pass `(want, got)` to the function), because the `-` and `+` that you add to your format string will match the `+` and `-` that actually appear at the beginning of the diff lines.

The diff will span multiple lines, so you should print a newline before you print the diff.

Table-Driven Tests vs Multiple Test Functions

Table-driven tests should be used whenever many different test cases can be tested using similar testing logic, for example when testing whether the actual output of a function is equal to the expected output [example], or when testing whether the outputs of a function always conform to the same set of invariants.

When some test cases need to be checked using different logic from other test cases, it is more appropriate to write multiple test functions. The logic of your test code can get difficult to understand when every entry in a table has to be subjected to multiple kinds of conditional logic to do the right kind of output check for the right kind of input. If they have different logic but identical setup, a sequence of subtests within a single test function might also make sense.

You can combine table-driven tests with multiple test functions. For example, if you're testing that a function's non-error output exactly matches the expected output, and you're also testing that the function returns some non-nil error when it gets invalid input, then the clearest unit tests can be achieved by writing two separate table-driven test functions — one for normal non-error outputs, and one for error outputs.

Test Error Semantics

When a unit test performs string comparisons or uses `reflect.DeepEqual` to check that particular kinds of errors are returned for particular inputs, you may find that your tests are fragile if you have to reword any of those error messages in the future. Since this has the potential to turn your unit test into a change detector, don't use string comparison to check what type of error your function returns.

It's OK to use string comparisons to check that error messages coming from the package under test satisfy some property, for example, that it includes the parameter name.

If you care about testing the exact type of error that your functions return, you should separate the error string intended for human eyes from the structure

that is exposed for programmatic use. In this case, you should avoid using `fmt.Errorf`, which tends to destroy semantic error information.

Many people who write APIs don't care exactly what kinds of errors their API returns for different inputs. If your API is like this, then it is sufficient to create error messages using `fmt.Errorf`, and then in the unit test, test only whether the error was non-nil when you expected an error.