Tests are a critical framework tool for stability and education. Tests fulfill the following roles:

- Automatically protect against regressions
- Define an executable specification that engrains original intent
- Educate other developers about why and how to use an API

To facilitate the above roles, there are some practices to keep in mind when writing tests:

Name tests based on the behavior being tested

It is common to find tests that are simply named after the object under test rather than the behavior under test. For example, a developer might find tests that look like the following:

```
// Bad test name
test('ListView', () {...});

// Bad test name
test('RenderViewport', () {...});

// etc.
```

The above test names do not communicate anything useful to the developer reading the tests. The developer probably already knows which object is being tested, so these names are no better than an empty string. Instead, each test should succinctly declare the behavior under test and/or the expected results:

```
// Better test name
test('Shrink-wrapped ListView resizes to match its content height', () {...});

// Better test name
test('RenderViewport applies its offset to all child Slivers', () {...});

// etc.
```

One behavior per test

It is common to find single tests that exercise multiple behaviors:

```
// Single test that tests multiple behaviors
test('SliverGeometry', () {
  expect(const SliverGeometry().debugAssertIsValid(), isTrue);
  expect(() {
    const SliverGeometry(layoutExtent: 10.0, paintExtent: 9.0).debugAssertIsValid();
  }, throwsFlutterError);
  expect(() {
    const SliverGeometry(paintExtent: 9.0, maxPaintExtent:
8.0).debugAssertIsValid();
  }, throwsFlutterError);
});
```

Don't test multiple behaviors in a single test. When multiple behaviors exist within a test then reported test failures become misleading. Is one thing broken, or many things? Is one method broken, or multiple methods, or are entire object interactions broken? When multiple behaviors exist in a single test then a broken test requires a developer to carefully analyze the code within the test just to get a feel for the magnitude of the bug. Add to this fact that the name of the test probably does not explain everything the test is doing, because the test is doing multiple things. Instead, include exactly one behavior per test:

```
test('SliverGeometry with no arguments is valid', () {
  expect(const SliverGeometry().debugAssertIsValid(), isTrue);
});

test('SliverGeometry throws error when layoutExtent exceeds paintExtent', () {
  expect(() {
    const SliverGeometry(layoutExtent: 10.0, paintExtent: 9.0).debugAssertIsValid();
  }, throwsFlutterError);
});

test('SliverGeometry throws error when maxPaintExtent is less than paintExtent', ()
{
  expect(() {
    const SliverGeometry(paintExtent: 9.0, maxPaintExtent:
8.0).debugAssertIsValid();
  }, throwsFlutterError);
});
```

What constitutes a single behavior? The answer to that question can vary. In most circumstances, only a single method should be invoked on the object under test, and it should be invoked only one time. However, that heuristic is not always correct. There can be cases where multiple method calls represent a singular behavior and a single set of expectations. Developers should use their discretion in this regard. That said, a higher number of shorter tests will tend to be a safer bet than a smaller number of longer tests.

Only include relevant details in a test

Tests often involve some amount of setup before the behavior-under-test can be executed. Some of these setup details are critical to the behavior-under-test, but many of these details are completely irrelevant.

Including irrelevant details in a test can only confuse the issue in the mind of the developer reading the test. The developer may become confused about which parts are relevant to him/her, and which are not. In such a scenario, the developer is likely to copy and paste the entire test without fully understanding what the code is doing, which is a recipe for disaster in a production app.

```
// Notice how much of the widget setup has nothing to do with the behavior-under-
test...
testWidgets('Title Section is empty, Button section is not empty.', (WidgetTester
tester) async {
  const double textScaleFactor = 1.0;
  final ScrollController scrollController = new ScrollController(keepScrollOffset:
true);
  await tester.pumpWidget(
    new MaterialApp(home: new Material(
      child: new Center(
        child: new Builder(builder: (BuildContext context) {
          return new RaisedButton(
            onPressed: () {
              showDialog<Null>(
                context: context,
                builder: (BuildContext context) {
                  return new MediaQuery(
                    data: MediaQuery.of(context).copyWith(textScaleFactor:
textScaleFactor),
                    child: new CupertinoAlertDialog(
                      actions: const <Widget>[
                        const CupertinoDialogAction(
                          child: const Text('One'),
                        ),
                        const CupertinoDialogAction(
                          child: const Text('Two'),
                        ),
                      ],
                      actionScrollController: scrollController,
                    ),
                  );
                },
              );
            },
            child: const Text('Go'),
          );
        }),
      ),
    )),
  );

  await tester.tap(find.text('Go'));

  await tester.pump();
  await tester.pump(const Duration(seconds: 1));

  // Check that the title/message section is not displayed
  expect(scrollController.offset, 0.0);
  expect(tester.getTopLeft(find.widgetWithText(CupertinoDialogAction, 'One')).dy,
equals(283.5));

  // Check that the button's vertical size is the same.
  expect(tester.getSize(find.widgetWithText(CupertinoDialogAction, 'One')).height,
    equals(tester.getSize(find.widgetWithText(CupertinoDialogAction,
'Two')).height));
});
```

Strive to write tests that only communicate details that are relevant to the behavior-under-test. This can be accomplished by extracting unrelated setup into other methods that are clearly named by their purpose:

```
// Now the unrelated details are factored out...
testWidgets('Title Section is empty, Button section is not empty.', (WidgetTester
tester) async {
  final ScrollController scrollController = new ScrollController(keepScrollOffset:
true);
  await _launchDialog(
    dialog: CupertinoAlertDialog(
      actions: const <Widget>[
        const CupertinoDialogAction(
          child: const Text('One'),
        ),
        const CupertinoDialogAction(
          child: const Text('Two'),
        ),
      ],
      actionScrollController: scrollController,
    ),
  );

  // Check that the title/message section is not displayed
  expect(scrollController.offset, 0.0);
  expect(tester.getTopLeft(find.widgetWithText(CupertinoDialogAction, 'One')).dy,
equals(283.5));

  // Check that the button's vertical size is the same.
  expect(tester.getSize(find.widgetWithText(CupertinoDialogAction, 'One')).height,
    equals(tester.getSize(find.widgetWithText(CupertinoDialogAction,
'Two')).height));
});
```

Optimize tests for comprehension

To illustrate the way that a small adjustment can make a test much more comprehensible to another developer, consider this test from above:

```
testWidgets('Title Section is empty, Button section is not empty.', (WidgetTester
tester) async {
  final ScrollController scrollController = new ScrollController(keepScrollOffset:
true);
  await _launchDialog(
    dialog: CupertinoAlertDialog(
      actions: const <Widget>[
        const CupertinoDialogAction(
          child: const Text('One'),
        ),
        const CupertinoDialogAction(
          child: const Text('Two'),
        ),
      ],
      actionScrollController: scrollController,
    ),
  );

  // Check that the title/message section is not displayed
  expect(scrollController.offset, 0.0);
  expect(tester.getTopLeft(find.widgetWithText(CupertinoDialogAction, 'One')).dy,
equals(283.5));

  // Check that the button's vertical size is the same.
  expect(tester.getSize(find.widgetWithText(CupertinoDialogAction, 'One')).height,
    equals(tester.getSize(find.widgetWithText(CupertinoDialogAction,
'Two')).height));
});
```

This test was presented as an improvement because it factored out unrelated details. But we can make this test better by slightly adjusting how we decompose the individual lines in the test:

```
testWidgets('Title Section is empty, Button section is not empty.', (WidgetTester
tester) async {
  final ScrollController scrollController = new ScrollController(keepScrollOffset:
true);

  final Widget dialog = CupertinoAlertDialog(
    actions: const <Widget>[
      const CupertinoDialogAction(
        child: const Text('One'),
      ),
      const CupertinoDialogAction(
        child: const Text('Two'),
      ),
    ],
    actionScrollController: scrollController,
  );

  await _launchDialog(dialog);

  // Check that the title/message section is not displayed
  expect(scrollController.offset, 0.0);
  expect(tester.getTopLeft(find.widgetWithText(CupertinoDialogAction, 'One')).dy,
equals(283.5));

  // Check that the button's vertical size is the same.
  expect(tester.getSize(find.widgetWithText(CupertinoDialogAction, 'One')).height,
    equals(tester.getSize(find.widgetWithText(CupertinoDialogAction,
'Two')).height));
});
```

By extracting the declaration of the `CupertinoAlertDialog` into its own statement, we have clearly disambiguated between the dialog we're here to test vs. the action of launching the dialog for testing purposes.

When writing tests, think about the developer who will read this test 6 months from now and ask if there is anything you can do to help that developer comprehend what the test is doing, and why.