Engineer

Unit test

an automated check of one small piece of code in isolation

What it is

A unit test is a small automated check of one piece of code on its own: it calls a function, a class or a small module with chosen inputs and asserts that what comes back is what should come back. Such tests are usually designed to run in the same process as the code they check, finish quickly and need no network, database or browser. Those are useful resource and isolation choices, not guarantees that follow from a test's narrow scope alone. The practice of writing them is usually called unit testing, and a project's collection of them is its unit test suite.

What counts as a unit is less settled than the name suggests. In the chapter on unit testing in *Software Engineering at Google*, Erik Kuefler uses the term for tests of relatively narrow scope, such as a single class or method, and adds that a unit may be a single function or reach as far as several related packages. The useful boundary is the one other people call: a unit is whatever offers an interface someone outside it relies on, and its tests use that interface the way its callers do.

The modern frameworks share an ancestry. Kent Beck's paper *Simple Smalltalk Testing: With Patterns* described a small framework built from fixtures, test cases, checks and suites, and advised that developers write their unit tests themselves, one for each class. Python's `unittest` says in its documentation that it was originally inspired by JUnit, and it keeps the same four ideas: a test fixture (the preparation a test needs and the cleanup after it), a test case (the individual unit of testing), a test suite and a test runner. Both draw a line worth keeping: a failure is a check that got the wrong answer, while an error is something the test did not anticipate at all, such as an exception thrown from deep inside the code.

Unit tests sit at the narrow end of a range. Adam Bender's chapter on testing in the same book sorts tests by scope: narrow tests are commonly called unit tests, medium-scoped tests that cover a few components together are integration tests, and large-scoped tests that drive the whole system are end-to-end tests.

Use the concept in a decision

Put "What is a unit test?" to work in a real decision. We can help relate the concept to the system you are building or buying. One team taking an idea from a sketch to something your customers can log into, instead of handing it between three suppliers.

Explore our services Get in touch →

Test the assumptions behind the term

Use the explanation and linked work to frame questions for your project. We can examine the assumptions with you before a tool is chosen. Led a GIS map application to 10x revenue growth — 10x revenue growth, led from the map product

Led a GIS map application to 10x revenue growth Get in touch →

Why it matters

Speed is the first reason. A small, isolated unit test runs in the same process as the code it checks, so an engineer can run thousands of them in an ordinary working day, and a mistake shows up while the change that caused it is still on the screen. Google's guidance is blunt about what it allows: a small test must not sleep, perform I/O, or touch the network or the disk. Those constraints make fast, repeatable tests easier to build, and a team learns to trust a suite that gives the same answer every time.

The second reason is diagnosis. When a narrow test fails, the fault is in a small piece of code, usually the one just changed. When a test that drives the whole system fails, the fault could be anywhere between the browser and the database. That is why *Software Engineering at Google* describes a pyramid as a rough guideline: about 80 per cent of tests narrow-scoped, 15 per cent integration tests and 5 per cent end-to-end tests.

The third is change. A suite that checks behaviour through a public interface lets the code behind that interface be restructured without fear: if the tests still pass, the callers are still served. Kuefler's ideal is a test that never has to change unless the requirements change, so a pure refactoring touches no test at all.

Unit tests also document. A test named after a behaviour, with plain inputs and an expected result written out in full, shows the next reader how the code is meant to be called and what it promises, and unlike a comment it fails when it stops being true.

What it looks like in practice

Take a function that reads durations such as `1h30m` from a configuration file and turns them into seconds:

```
# durations.py
UNITS = {"h": 3600, "m": 60, "s": 1}


def to_seconds(text: str) -> int:
    """Turn "1h30m" or "45s" into a whole number of seconds."""
    total, digits = 0, ""
    for char in text:
        if char.isdigit():
            digits += char
        elif char in UNITS and digits:
            total += int(digits) * UNITS[char]
            digits = ""
        else:
            raise ValueError(f"not a duration: {text!r}")
    if digits or not text:
        raise ValueError(f"not a duration: {text!r}")
    return total
```

Its unit tests, written for pytest, check two behaviours: what it accepts and what it refuses.

```
# test_durations.py
import pytest

from durations import to_seconds


@pytest.mark.parametrize(
    "text,expected",
    [("45s", 45), ("2m", 120), ("1h30m", 5400), ("0s", 0)],
)
def test_counts_each_unit(text, expected):
    assert to_seconds(text) == expected


@pytest.mark.parametrize("text", ["", "90", "h", "1x", "1.5h"])
def test_refuses_what_it_cannot_read(text):
    with pytest.raises(ValueError):
        to_seconds(text)
```

Running `pytest` in the directory collects every file named `test_*.py` or `*_test.py`, and the two functions expand into nine cases, one per row of parameters. The pytest documentation describes each test as arrange, act, assert and cleanup; here the arrangement is the parameter row, the act is the single call, and there is nothing to clean up because nothing outside the function is touched. If someone later sets the minute to 3600 seconds by mistake, two cases fail, and the report names each by its parameters: `test_counts_each_unit[2m-120]` shows 7200 against the expected 120, and `test_counts_each_unit[1h30m-5400]` shows 111600 against 5400. Note that the expected values are worked out by hand, not copied from what the code returned on its first run.

The same checks in the standard library's `unittest` would be a subclass of `unittest.TestCase` with methods whose names start with `test`, using `assertEqual` and `assertRaises`; `python -m unittest` finds files matching `test*.py` by default. Go puts the convention into the toolchain: a function `func TestXxx(t *testing.T)` in a file ending in `_test.go` is run by `go test`.

Scale changes the bookkeeping, not the idea. The database generator held to 981 test cases, work carried out in July 2026, counts 383 test functions that become 981 cases once the parameterised ones expand, across 45 files and 7,944 lines. Its build enforces a branch-coverage floor of 92 per cent, with about 95 per cent measured, and treats warnings as failures. The page is candid about the limit of the number: two defects found later sat in code the coverage report showed as covered.

Where it goes wrong

The tests mirror the implementation. A test that reaches into private helpers, or asserts the exact sequence of internal calls, breaks every time the code is restructured even though nothing a caller sees has changed. Kuefler calls this a brittle test, and names over-reliance on mocking frameworks as the most common reason that tests of interactions go wrong. Test through the public interface, check the resulting state rather than the calls made to get there, and use real objects where they are cheap.

Tests depend on each other. One test creates a record, the next one reads it, and the suite passes only in that order. Run a single test, or the same tests in a different order, and it fails for no reason a reader can see. Beck's framework

created and threw away its test data for each test, and the `unittest` documentation asks that a test case be entirely self-contained. Give every test its own fixture and clean up after it.

Coverage becomes the goal. Coverage shows that a line ran, not that anything checked what it did. A test with no assertion raises the number and proves nothing, and *Software Engineering at Google* warns that an 80 per cent bar tends to become a ceiling, with changes landing at exactly the minimum. Treat the number as a floor and read the assertions in review.

One test per method. Matching each production method with one test method looks tidy, but the test grows every time the method gains a branch, until nobody can say what it checks. Kuefler's advice is to test behaviours instead: one behaviour per test, named after what it proves, so a failure report reads as a sentence about what broke.

The test waits for time to pass. A `sleep()` to let something finish makes the suite slower on every run and still fails on a busy machine. In a unit test, time is an input: pass in a clock or a timestamp, so the test can set the time it needs instead of waiting for it.

Further reading

- unittest: Unit testing framework — the Python standard library's framework: fixtures, test cases, suites, runners, discovery and the difference between a failure and an error.
- Anatomy of a test — pytest's explanation of arrange, act, assert and cleanup.
- Get Started — pytest's own introduction: plain `assert`, test discovery and `pytest.raises`.
- Unit Testing — Erik Kuefler's chapter of *Software Engineering at Google* on brittle tests, public APIs, state versus interactions and testing behaviours.
- Simple Smalltalk Testing: With Patterns — Kent Beck's paper on the framework, the case for writing unit tests one per class, and failures versus errors, in an archived copy.
- testing — Go's standard testing package: `TestXxx` functions, `_test.go` files and `go test`.

Glossary

Choose for your actual constraints

Describe the workflow, the existing interfaces and the users who need the change. We can define a testable delivery and a handover that lets your team maintain it.

See what it costs Get in touch →

Make the terminology practical

Ask how “What is a unit test?” applies to your organisation. A concrete example from your current system is enough to start the discussion.

Explore our services

welcome@engineer.company +45 50 10 61 84 Copenhagen, Denmark

Take this page with you

Contact card

Carefully Crafted With Love by D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

Copyright © 2025 – present · Engineer ApS · All rights reserved.

Printed from https://engineer.company/glossary/unit-test/