Godkjenn

Release 7.0.0

Austin Bingham

Oct 11, 2021

CONTENTS:

godkjenn /go:kjen/ Approve (Norwegian)

Approval testing for Python 3.

CHAPTER

ONE

INTRODUCTION

Godkjenn is a test for performing *approval testing* in Python. Approval testing is a simple yet powerful technique for testing systems that may not be amenable to more traditional kinds of tests or for which more traditional tests might not be particularly valuable. Approval testing can also be extremely useful for adding tests to legacy systems where simply capturing the behavior of the system as a whole is the first step in adding more sophisticated tests.

1.1 Principle

The principle of approval testing simple. Given a function or program that you consider correct, you store its output for a given input. This is the *accepted* or *golden* version of its output. Then, as you change your code, you reproduce the output (we call this *received* output) and compare it to the accepted version. If they match, then the test passes. Otherwise, it fails.

A test failure can mean one of two things. First, it could mean that you actually broke you program and need to fix it so that the received output matches the accepted. Second, it could mean the received output is now correct, the accepted output is now out of date, and you need to update the accepted output with the received.

As an approval testing tool, `godkjenn` aims to streamline and simplify this kind of testing.

1.2 Core elements

There are a few core elements to `godkjenn`. These are the parts of the approval testing system that are independent of any particular testing framework. Generally speaking, you won't need to work with these directly; the integration with your testing framework with hide most of the low-level details.

1.2.1 Vaults

Vaults are where the accepted outputs are stored. (The term vault is a bit of a play on words: the accepted output is "golden", and you keep gold in vaults.)

The vault abstraction defines an API for storing and retrieving accepted (and received) output.

`godkjenn` provides a simple vault, `FSVault`, that stores its data on the filesystem. Other vaults can be provided via a plugin system.

1.2.2 Verification

The core verification algorithm compares new received data with the accepted data for a given test. If there's a mismatch or if not accepted output exists, this triggers test failure and instructs the user on what to do next.

1.2.3 Diffing

When an approval test fails, `godkjenn` provides tools for viewing the differences between the accepted and received data. `godkjenn` comes with some very basic fallback diffing support, and it provides a way to run external diffing tools. You can even have it use different tools for different types of files.

CHAPTER

TWO

GODKJENN TUTORIAL

This will take you through the process of installing, setting up, and using godkjenn for approval testing in a Python project.

Note: This tutorial will use godkjenn's pytest integration. Godkjenn does not mandate the use of pytest, though currently it's the only testing framework for which godkjenn provides an integration. Integrating with other frameworks is straightforward and encouraged!

2.1 Installing godkjenn

First you need to install godkjenn. The simplest and most common way to do this is with *pip*:

```
pip install godkjenn
```

For pytest integration you'll also want to install the necessary plugin:

```
pip install "godkjenn[pytest-plugin]"
```

2.2 A first test

Now let's create our first test that uses godkjenn. Create a directory to contain the code for the rest of this tutorial. We'll refer to it as `TEST_DIR` or `$TEST_DIR`.

Once you have that directory, create the file `TEST_DIR/pytest.ini`. This can be empty; it just exists to tell pytest where the "top" of your tests is.

Next create the file `TEST_DIR/test_tutorial.py` with these contents:

```
1 def test_demo(godkjenn):
2     test_data = b'12345'
3     godkjenn.verify(test_data, mime_type='application/octet-stream')
```

This will be mostly familiar if you've used pytest: it's just a single test function with a fixture.

On line 1 we define the test function. The `godkjenn` parameter tells pytest that we want to use the `godkjenn` fixture. This fixture gives us an object that we use for *verifying* our test data.

On line 2 we simply invent some test data. Notice that it's a `bytes` object. Godkjenn ultimately requires all of its data to be `bytes`, so for this tutorial we've just created some simple data. In practice, this data would be the output from some function that you want to test.

Finally on line 3 we call `godkjenn.verify()`, passing in our `test_data`. This call will take the data we pass in and compare it to the currently-accepted "golden" output. If the data we pass in does not match the accepted output, the test is flagged as a failure. Similarly - as will happen for us - we'll get a failure if there is *no* existing accepted output.

2.2.1 Running the test

Now we can run the tests with pytest. For now just run the test from `TEST_DIR`:

```
cd $TEST_DIR
pytest .
```

You should see some output like this:

```
$ pytest .
============================================================================================================test session starts␣
↪============================================================================================================
platform darwin -- Python 3.8.0, pytest-6.2.2, py-1.10.0, pluggy-0.13.1
rootdir: /Users/abingham/repos/sixty-north/godkjenn/docs/tutorial, configfile: pytest.ini
plugins: hypothesis-6.4.0, godkjenn-2.0.1
collected 1 item

test_tutorial.py F                                                                                                  ␣
↪                                                                                                                   ␣
↪                                                    [100%]

============================================================================================================FAILURES␣
↪============================================================================================================
_____________________________________________________________________________________________________________
↪_____________________ test_demo ______________________________________________________
↪_____________________________________________________
There is no accepted data

If you wish to accept the received result, run:

    godkjenn -C . accept "test_tutorial.py::test_demo"

============================================================================================================short test summary info␣
↪============================================================================================================
FAILED test_tutorial.py::test_demo
============================================================================================================1 failed in 0.07s␣
↪============================================================================================================
```

We see that - as expected - our test failed. The report tells us that "There is no accepted data". This means that this is the first time we've run the test and haven't accepted any output for the test. We're also given instructions on how accept the output if we believe it to be correct.

This idea of of "accepting" output is central to the notion of approval testing. At some point we have to decide that our code is correct and that the output it produces is indicative of that proper functioning. For now let's assume that our data is correct.

Status

Before accepting the data, let's use the `godkjenn status` command to see the state of our approval tests:

```
$ godkjenn status

 Test ID                       Status

│ test_tutorial.py::test_demo │ initialized │
```

This tells us that we have one approval test in our system and that its state is "initialized". This means that it has some *received* data (i.e. the data from the test we just ran) but no accepted data.

Accepting the data

Since we believe that the data we passed to `godkjenn.verify()` represents the correct output from our program, we want to accept it. We can use the command provided to us in the test output:

```
$ godkjenn accept "test_tutorial.py::test_demo"
```

Now if we run `status` we see don't get any output:

```
$ godkjenn status
```

This is because all of our tests are "up-to-date", i.e. all of that have accepted data and not received data. In order to see the status of *all* test-ids, including those that are up-to-date, you can use the "-a/–show-all" options of the 'status' command:

```
$ godkjenn status --show-all

 Test ID                       Status

│ test_tutorial.py::test_demo │ up-to-date │
```

And that's it! You've created your first godkjenn approval test and accepted its output.

2.3 Accepting new data

Over time, of course, your code may change such that its correct output no longer matches your accepted output. When this happens your test will fail and you'll have to accept the new data. To see this, let's change our `test_tutorial.py` to look like this:

```
1 def test_demo(godkjenn):
2     test_data = b'1234567890'
3     godkjenn.verify(test_data, mime_type='application/octet-stream')
```

You can see on line 2 that `test_data` now has more digits. When we run our test we get a failure because of this change:

```
$ pytest test_tutorial.py
============================================================================================================================
↪test session starts␣
↪============================================================================================================================
platform darwin -- Python 3.8.0, pytest-6.2.2, py-1.10.0, pluggy-0.13.1
rootdir: /Users/abingham/repos/sixty-north/godkjenn/docs/tutorial, configfile: pytest.ini
plugins: hypothesis-6.4.0, godkjenn-2.0.1
collected 1 item

test_tutorial.py F                                                                                                      ␣
↪                                                                                                                       ␣ ␣
↪                                                                                                                       ␣ ␣
↪              [100%]

============================================================================================================================
↪FAILURES␣
↪============================================================================================================================
_____________________________________________________________________________________________________________________
↪_____________________________________________ test_demo ______________________________________
↪_____________________________________________________________________________________________________________________
↪_____________________
Received data does not match accepted

If you wish to accept the received result, run:

    godkjenn -C . accept "test_tutorial.py::test_demo"

============================================================================================================================
↪short test summary info␣
↪============================================================================================================================
FAILED test_tutorial.py::test_demo
============================================================================================================================
↪1 failed in 0.05s␣
↪============================================================================================================================
```

You can see the failure was because "Received data does not match accepted". That is, the data we're passing to `godkjenn.verify()` doesn't match the accepted data.

If we run `godkjenn status` again, we see a new status for our test:

```
$ godkjenn status

 Test ID                       Status

 test_tutorial.py::test_demo   mismatch
```

The status "mismatch" means that the received and accepted data are different.

2.3.1 Seeing the difference

Our job now is to decide if the *received* data is correct and should become the *accepted*. To make this decision it can be very helpful to see the accepted data, the received data, and the differenced between them.

To see the accepted data we can use the `godkjenn accepted` command:

```
$ godkjenn accepted "test_tutorial.py::test_demo" -
12345%
```

Similarly, we can see the received data using the `godkjenn received` command:

```
$ godkjenn received "test_tutorial.py::test_demo" -
1234567890%
```

In this case it's pretty easy to see the difference. In other cases it might be more difficult. To help with this godkjenn also lets you view the difference between the files with the `godkjenn diff` command. By default `godkjenn diff` uses a very basic diff display:

```
$ godkjenn diff "test_tutorial.py::test_demo"
WARNING:godkjenn.cli:No review tools configured. Fallback differs will be used.
---
+++
@@ -1 +1 @@
-12345
+1234567890
```

Again, in a simple case like this, this default diff output is enough to make it clear what the difference is. In more complex cases you might need more powerful tools, though, and we'll look at how to use those soon.

2.3.2 Configuring an external diff tool

The built-in diff tool in godkjenn is sufficient for simple cases, but many people have other, more sophisticated diff tools that they would prefer to use with approval testing. Godkjenn allows you to specify these tools in your configuration.

For this tutorial we're going to configure godkjenn to use Beyond Compare as its default diff tool. To do this you need to create the file ".godkjenn/config.toml". Put these contents in that file:

```
[godkjenn.differs]
default_command = "bcomp {accepted.path} {received.path}"
```

This is telling godkjenn to run the command "bcomp" (the Beyond Compare executble) to display diffs. The first argument to "bcomp" will be the path to the current *accepted* data, and the second is the path to the *received* data. With this configuration, whenever godkjenn needs to display a diff (e.g in the "diff" and "review" commands), it will use "bcomp".

If you don't have Beyond Compare installed, you can replace "bcomp" with many other commands like "diff", "vimdiff", and "p4diff".

Once you've made this change, you can run your *godkjenn diff* command again and see your configure diff tool being used.

Note: Godkjenn supports fairly sophisticated configuration of diff tools, allowing you to use different diff tools for different MIME types. See the *configuration documentation* for details.

2.3.3 Accepting the new data

We'll assume that our new data is actually correct and accept it:

```
godkjenn accept "test_tutorial.py::test_demo"
```

Once we do that we see that our status is back to "up-to-date":

```
$ godkjenn status --show-all

 Test ID                       Status

| test_tutorial.py::test_demo | up-to-date |
```

2.4 Reviewing multiple tests

The *godkjenn diff* command lets you view the difference between the accepted and received data for a single test. In many cases, though, you have several - and in some cases a great many - tests for which you need to see the diff. The *godkjenn review* command lets you view all of the diffs for 'mismatch' tests in sequence.

In effect, the `review` command calls `diff` for each test that's in the mismatch state, one after the other, using the diffing tools that you've configured.

To see `review` in action, let's first add a new test. Here's the new contents of "test_tutorial.py":

```
def test_demo(godkjenn):
    test_data = b"1234567890"
    godkjenn.verify(test_data, mime_type="application/octet-stream")


def test_second_demo(godkjenn):
    test_data = b"8675309"
    godkjenn.verify(test_data, mime_type="application/octet-stream")
```

We'll run the tests and accept the received data in order to lay a foundation for running `review`:

```
$ pytest test_tutorial.py
============================================================ test session starts␣
↪============================================================
platform darwin -- Python 3.8.0, pytest-6.2.2, py-1.10.0, pluggy-0.13.1
rootdir: /Users/abingham/repos/sixty-north/godkjenn/docs/tutorial/sandbox, configfile:␣
↪pytest.ini
plugins: hypothesis-6.4.0, godkjenn-4.0.0
collected 2 items

test_tutorial.py FF                                                              ␣
↪                                  [100%]

================================================================= FAILURES␣
↪=================================================================
_____________________________________________________________ test_demo ____________________
↪_____________________________________
```

(continues on next page)

(continued from previous page)

```
Received data does not match accepted

If you wish to accept the received result, run:

    godkjenn -C . accept "test_tutorial.py::test_demo"

_____________________________________________________ test_second_demo _____________
↪_____________________________________
There is no accepted data

If you wish to accept the received result, run:

    godkjenn -C . accept "test_tutorial.py::test_second_demo"

================================================== short test summary info␣
↪==================================================
FAILED test_tutorial.py::test_demo
FAILED test_tutorial.py::test_second_demo
==================================================== 2 failed in 0.03s␣
↪====================================================
$ godkjenn accept-all
```

Now we'll modify the tests so that each produces different output:

```
def test_demo(godkjenn):
    test_data = b"-- 1234567890 --"
    godkjenn.verify(test_data, mime_type="application/octet-stream")


def test_second_demo(godkjenn):
    test_data = b"-- 8675309 --"
    godkjenn.verify(test_data, mime_type="application/octet-stream")
```

If you run the tests again, `godkjenn status` shows that both tests are in the 'mismatch' state:

```
$ pytest test_tutorial.py
... elided ...
$ godkjenn status

 Test ID                              Status

┌ test_tutorial.py::test_second_demo ┌ mismatch ┐
┌ test_tutorial.py::test_demo        ┌ mismatch ┐
└──────────────────────────────────────────────┘
```

Now if you run `godkjenn review`, `godkjenn` will run your configured diff tool twice, once for each test, letting you view and potentially modify the accepted data.

Critically, if the *received* and *accepted* data are identical after `godkjenn` runs your diff tool (i.e. if you edit them through your diff tool), then `godkjenn` will accept the data and mark them as 'up-to-date'. This gives you a conventient way to rapidly iterate through a set of received data, verifying each in turn and rapidly updating the accepted data for each affected test.

Note: The `godkjenn review` command can be very useful if you've a collection of received data for which you need to manually verify each one. However, if you somehow know that all of the received data should be accepted, then it's even faster to use the `godkjenn accept-all` command.

CHAPTER

THREE

CONFIGURING GODKJENN

Godkjenn support some minor configuration through the file `config.toml` in the data directory. This file must contain a top-level 'godkjenn' key, for example:

```
[godkjenn]
```

Everything under this key is part of the godkjenn configuration.

3.1 Configurable options

3.1.1 differs.default_command

The `godkjenn.differs.default_command` option specifies the default tool to use for displaying diffs. The value is a template used with the `str.format()` method that is passed `Artifact` instances for the received and accepted data. The call looks like this:

```
command_template = . . . value of differs.default_command . . .
command = command_template.format(accepted=accepted_artifact, received=received.artifact)
```

As you can see, the artifacts are passed using the 'accepted' and 'received' keyword arguments.

A common template value would extract the `path` attribute from each artifacts and use those as arguments to a diff/merge tool. For example, here's a config that passes the artifact paths to vimdiff:

```
[godkjenn.differs]
default_command = "vimdiff {accepted.path} {received.path}"
```

This option is only used if there is not a MIME-type-specific tool defined in `godkjenn.differs.mime_types`.

3.1.2 differs.mime_types

The `godkjenn.differs.mime_types` option allows you to specify diff commands for specific MIME-types. It is a mapping from MIME-types to command template (as described in the `differs.default_command` option). When godkjenn needs to run a diff tool for an artifact the MIME-type of the *received* data is looked up in the `differs.mime_types` mapping, and if it's found then the value is used as the command template. If no match is found, then `differs.default_command` is used as the default.

For example, here's how you could specify that the `image-diff` tool should be used for PNGs, `vimdiff` should be used for plain text, and `bcomp` for everything else:

```
[godkjenn.differs]
default_command = "bcomp {accepted.path} {received.path}"

[godkjenn.differs.mime_types]
"image/png" = "image-diff {accepted.path} {received.path}"
"text/plain" = "vimdiff {accepted.path} {received.path}"
```

Note: If there is no `differs.mime_types` entry for an artifact, and if `differs.default_command` is not set, godkjenn fallsback to some fairly primitive built-in diffing tools. You're almost always best off configuring at least the `differs.default_command` options.

CHAPTER

FOUR

LOCATING DATA

4.1 TL;DR

Tell godkjenn where to start looking for the data directory with the `-C` option. Tell it the name of the data directory with the `-d` options. E.g. `godkjenn -C tests -d .godkjenn_dir status`.

`-C` defaults to "." (i.e. the current directory), and `-d` defaults to ".godkjenn".

4.2 Overview

Godkjenn stores data for each call to `verify()` in your tests. The data it stores includes any accepted or received data for the test, as well as any metadata required for the other data (e.g. mime types, encoding, etc.). Godkjenn wraps all of this up in the concept of a *vault*, and the vault stores this data on the filesystem.

When you run godkjenn, it needs to know where to find the vault data. It find this based on two pieces of data: a starting directory and the name of the *data directory*. By default the starting directory is whichever directory you run godkjenn from. Similarly, the default value for the *data directory* name is ".godkjenn". Godkjenn starts by looking for the *data directory* name in the starting directory. If it doesn't find it, it checks the parent of the starting directory. It continues looking up the ancestor directories until either a) it find a directory containing the data directory or b) it runs out of ancestors.

Assuming a data directory is found, godkjenn now has the information it needs to run.

4.3 An example

Suppose you had a directory structure like this:

```
my_project/
    tests/
        .godkjenn
```

Here the ".godkjenn" directory is the *data directory* that godkjenn needs to find.

The simplest mode for godkjenn is if you run it from the 'tests' directory:

```
cd my_project/tests
godkjenn status
```

Run this way, godkjenn will find the ".godkjenn" directory using its default settings. It will first start its search from the "tests" directory because that's where we're running godkjenn from. Since by default it looks for a directory called ".godkjenn", it will find it immediately.

4.3.1 Specifying a start directory

Suppose though that you wanted to start godkjenn from another directory. If we wanted to start godkjenn from the 'my_project' directory we'd need to tell it where to start looking for ".godkjenn". We can use the `-C` command line argument to do this. Here's how it looks if we start godkjenn from the 'my_project' directory:

```
cd my_project
godkjenn -C tests status
```

In this case, instead of starting the search in the "my_project" directory as it would by default, godkjenn starts the search from the 'tests' directory.

4.3.2 Using a different data directory name

While unusual, it is technically possible to use a different name for the data directory. Suppose you instead had this directory structure:

```
my_project/
    tests/
        .godkjenn_data
```

You can tell godkjenn to look for a different name with the `-d` command line option. For example, to run godkjenn from the 'my_project' directory with this structure you'd use this:

```
cd my_project
godkjenn -C tests -d .godkjenn_data status
```

Again, you won't normally need this, but it's there if you do.

4.4 Creating the data directory

Probably the most common way to create the data directory is to let godkjenn do it for you automatically. This generally happens in one of two ways. First, when you run tests using the pytest integration, it will create a ".godkjenn" directory in the pytest root directory. Pytest uses a fairly involved algorithm to determine the location of the root directory.

Another way to create the data directory is with the `pytest receive` command. If no existing data directory exists (as determined by the algorithm described above), then this command will try to create a ".godkjenn" directory in your current directory. If you use the `-C` and/or `-d` arguments to change where godkjenn starts its search, then `godkjenn receive` will create the directory there instead.

If you **don't** want `godkjenn receive` to create a directory, you can pass the "–no-init" argument.

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search