Tests that (Almost) Write Themselves

Hints for Golden Master Testing in Python

Stefan Baerisch, stefan@stbaer.com, 2020-07-20
Stefan Baerisch
stbaer.com (German only, sorry)

Software since 2005

Python since 2006

Project management / Test Management since 2010
Some Background on Golden Master Tests
Good Duck?

How would you describe an ‘acceptable’ rubber duck? Do you know it when you see it?
How would you describe an ‘acceptable’ rubber duck?
Do you know it when you see it?
The Idea behind Golden Master Testing

How to implement test if data is unknown or too complex for an assert()?
What is Golden Master Testing

1 Run
2 Change
3 Run Again
4 Compare

1a Capture Output
3a Capture Output
How does this help?

**Scenario**
- Testing a legacy system
- Testing complex data
- Testing complex data with some changes

**Action**
- Ensure that changes are visible
- Identify and review changes
- Select or filter data used for tests
Golden Master Tests are also known as:
- Characterization Tests
- Approval Tests
- Snapshot Tests

Because changes to the data are ‘approved’

In the Javascript World, especially Jest
Implementing Golden Master Tests
The Process

**assert(check(data))**

- **Data exists?**
  - Yes: Create Diffable Format
    - Compare current data to stored data
    - Differences?
      - No Differences?: Store current results, Return True
      - Differences?: Store data, Return True
    - OK?: Replace Stored Data
    - Not OK?: Change Code
  - No: No Differences?, Return True
- **Store data**
- **Return True**

**Automatic Test Process**

**Manual Approval Process**

- **Review Differences**
  - OK?:
  - Not OK?:
    - Replace Stored Data
    - Change Code
## Design Decisions

<table>
<thead>
<tr>
<th>Test Target</th>
<th>State</th>
<th>Log</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Test Size</th>
<th>Few Large</th>
<th>Many Small</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Storage Formats</th>
<th>Text</th>
<th>Json/XML</th>
<th>Custom</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Diff Approach</th>
<th>CLI $…</th>
<th>Lib</th>
<th>Custom</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>What to Ignore</th>
<th>Key Order</th>
<th>Dates</th>
<th>Random Values</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Keep it simple. Python makes it quite easy to get a text presentation of data
What to Ignore

What to ignore?

Dates / timestamps
Key order for non-ordered dicts
Random data (object ids / sequence ids)
Non-relevant data (non significant floating point data)

Needs some pre-processing, depending on use case
Test Size

Large Tests ↔ Small Tests

- easy, fast setup
- will notice most changes
- frequent reviews
- likely more work cleaning up data

- will need to break up / select data
- may miss changes
- less frequent reviews / shorter reviews
- Less work cleaning up data
How do you store results from past runs?

The format should be:

- Easy to store in VCSs
- Diffable in Python and External Tools
- Supported by Editors / Views
- Should be structured to support working with Python

If possible, use normalized JSON with linebreaks
An Example Implementation
Design Goals

Keep things simple

Compare Python Objects, no extensions
Simple comparison
Simple storage
Simple operations

jsonpickle

difflib.unified_diff

store files with tests, explicit naming of tests
check() - store or compare data
list() - show stored data and conflicts
review(name) - show differences
approve(name) - mark the current version as ok
```python
class MyTestClass:
    def __init__(self):
        self.a = "Hello"
        self.b = 1
        self.c = 2.33433
        self.d = [1, 2.3333, 3.3332]
        self.e = "2020-05-12"

test_data = MyTestClass()
changed_test_data = MyTestClass()
changed_test_data.d.pop()
changed_test_data.c += 0.5
```
Overview

```python
checker = checker.Checker(os.path.dirname(__file__))

assert checker.check(test_data, "name1")  # Save
assert checker.check(test_data, "name1")  # Compare

assert checker.check(test_data, "name2")  # Save
assert checker.check(changed_test_data, "name2")  # Compare

checker.list()  # Show state of tests / stored data

checker.review("name2")  # Show diff of “name2” test

checker.approve("name2")  # Accept the lastest version
```

Set storage location

Success

Failure
checker.list()

checker.review("name2")

checker.approve("name2")

```python
def checker(list)
    return

def name2
    now = name2.now
    last = name2.last
```
def check(self, obj, name):
    jsonpickle.set_preferred_backend('json')
    jsonpickle.set_encoder_options('json', sort_keys=True, indent=4)
    now_text = jsonpickle.encode(obj, unpicklable=False)

    last_filename = self._get_filename(name, LAST)
    if os.path.exists(last_filename):
        last_text = open(last_filename, mode="r", encoding="utf-8").read()
        diff = difflib.unified_diff(now_text.split("\n"), last_text.split("\n"))
        if len(list(diff)) != 0:
            now_filename = self._get_filename(name, NOW)
            self._write_file(now_filename, now_text)
            return False
        else:
            return True
    else:
        self._write_file(last_filename, now_text)
        return True
def list(self):
    for val in self._get_list():
        print(f"Name: {val['name']} Conflict: {val['conflict']}")

def _get_list(self):
    result = []
    cands = glob.glob(os.path.join(self.path, "*.last"))
    for cand in cands:
        name = os.path.basename(cand).replace(".last", ".").lower()
        conflict = os.path.exists(cand.replace(".last", ".now"))
        result.append({'name': name, 'conflict': conflict, 'filepath': cand})
    return result
```python
def review(self, name=None):
    read = lambda fn: open(fn, mode="r", encoding="utf-8").read().split("\n")
    for cand in self._get_entries(name, True):
        last_name = cand['filepath']
        last_cont = read(last_name)
        now_name = cand['filepath'].replace(".last", ".now")
        now_cont = read(now_name)
        diff = list(difflib.unified_diff(last_cont, now_cont,
                                          tofile=now_name, fromfile=last_name))
        print("\n".join(diff[2:]))

def _get_entries(self, name, only_conflict=False):
    cands = self._get_list()
    if name is not None:
        cands = [c for c in cands if c['name'] == name.lower()
                 and (not only_conflict or c['conflict'])]
    return cands
```
def approve(self, name=None):
    for cand in self._get_entries(name, True):
        last = cand['filepath']
        now = last.replace(".last", ".now")
        print(f"{os.path.basename(now)} => {os.path.basename(last)}")
        shutil.move(now, last)
Other Existing Libraries

github.com/syrusakbary/snapshottest
Inspired by Javascript’s Jess. Nice Integration with unittest/nose/pytest

github.com/approvals/ApprovalTests.Python
Python implementation of approval testing,
Summary

Why
Golden Master Tests work by capturing & comparing results of program executions. It helps with complex data, especially when we want to monitor for changes.

How
We store results and compare between run. On differences, we review & approve the results.

Python
Python has many modules to help us. Some existing implementations exist if we don’t want to implement it ourselves.
Thank you!