SAS2347-2018

Test-Driven Data Science: Writing Unit Tests for SASPy Python Data Processes

Stephen Siegert, SAS Institute Inc.

ABSTRACT

Code needs tested == True. We've all been there—tweak this parameter, comment another line out—but how do you maintain the expected function of your programs as they change?

Unit tests! SASPy is a Python interface module to SAS®. It enables developers and data scientists alike to leverage SAS and Python by using both APIs to build robust systems. Oh yeah, you can submit SAS code, too! This paper walks through a test-driven development (TDD) approach to leveraging a data pipeline powered by SAS with Python. The pytest framework and unittest library are covered.

INTRODUCTION

The SASPy module is a Python interface to MultiVendor Architecture™ SAS. It provides a clean application programming interface (API) to SAS procedures. Also, it allows SAS code, in raw form, to be submitted from a Python program. In this paper, a basic test-driven development (TDD) workflow to writing data process functions is provided.

The TDD approach encourages you to determine the expected behavior of your code before you actually program the logic. This paper assumes that the reader has a working environment with the required software for SASPy. Experience with Base SAS® and Python programming are required. Also, a familiarity of the pandas library is suggested because SASPy inherits class methods and attributes from the DataFrame structure. Please use the GitHub Issues tracker for the SASPy project if you experience any bugs or if you want to suggest an enhancement.

The documentation and project page is located on GitHub at https://sassoftware.github.io/saspy/.

All code used in this paper is included in Appendix A.

UNIT TESTING OVERVIEW

Unit tests are a common programming construct that help increase consistency and reduce errors in programs. It is very common for programmers to interactively program—that is, modify a portion of code until it works based on the current task at hand. This type of programming does not scale when several developers interact with the same code base. In addition, version-control systems such as Git and Subversion (SVN) encourage collaboration. A lack of testing can introduce unexpected bugs or regressions in data processes as a result of many developers working on the same code without understanding the impact of their contributions.

Choosing a testing library is a matter of preference. The two used in this paper were chosen because one is always included in a working distribution of Python (unittest). The other is well known in the community because it's a less verbose but just as powerful approach to writing tests (pytest).

- unittest is included in the Python standard library.
- pytest is a testing framework that is commonly used in the Python community.

With this information, you are now ready to step through some tests using the TDD method. Here are the steps:

1. Write the test based on a requirement.
2. Write a function to pass the test.

3. Run the tests.
4. Iterate until all tests pass.

In this paper, unittest is used in subsequent test cases. For reference, a duplicate test file for pytest is included in Appendix A to demonstrate the differences in syntax. As you'll see, the two approaches are very similar.

DATA PROCESS OVERVIEW

First, create a directory that matches the structure below:

```
/data_process
  data_process.py
  test_data_process_unittest.py
  test_data_process_pytest.py
```

You need three files for this. First, you need the data processing program data_process.py. This program holds the logic and functions that you will be testing. The other two files are for tests. These files demonstrate the tests for unittest and pytest, respectively. You write these tests first because you are approaching this from a TDD perspective! That seems daunting *and cool* at the same time. The idea is that code is written to satisfy only the given requirements, and no more. By defining the requirements of the code using test cases, you do not add extraneous code.

Create a test for each of the requirements below:

- Check the structure of the original input data.
- Sort and remove duplicate data.
- Remove an unused variable.

unittest test suites are modules that group together sets of tests. It usually makes sense to group tests into logical sets as they relate to the functionality of the code that they are testing.

Enter the follow code into the test_data_process_unittest.py file:

```
# test_data_process_unittest.py

import unittest
import data_process


class TestDataProcess(unittest.TestCase):

    def setUp(self):
        pass

    # individual tests will go here

    def tearDown(self):
        pass

if __name__ == '__main__':
    unittest.main()
```

The `setUp()` and `tearDown()` methods are unittest fixtures. These methods run immediately before and after each test, respectively. These methods are useful for specifying reusable seed or test data and removing temporary files after use. These methods are not always needed, but it helps to stub them in to visualize the test flow.

The `data_process` module is imported. This is the file that houses the functionality that you are testing! This is imported ***into*** the test file, not the other way around. This enables you to create a complete test environment that mimics real-life use.

Initialize and commit the unittest boilerplate. In the directory previously created, initialize a new Git repository from the command line (or terminal session):

```
$ git init

$ git commit -m 'set up unittest structure'
```

Git is used along the way to track your progress with commits. Using version control is a best practice and an easy habit to learn. The `–m` flag in the second command indicates that a commit message follows.

Okay, cool, right? But you still haven't written any actual tests. Let's get started.

REQUIREMENT: CHECK THE STRUCTURE OF THE ORIGINAL INPUT DATA

Adjust the `setUp()` method to define the test data that you will use on an instance attribute of the test case `TestDataProcess`. This data is a small representation of a larger hypothetical data set. Note that the data is of the list data type, containing nested dictionaries of data.

Enter the code below into the test_data_process_unittest.py file in the `setUp` method:

```
# test_data_process_unittest.py

...

def setUp(self):
  self.data = [
    {
      "id": 1,
      "name": "Leanne Graham",
      "username": "Bret",
      "email": "Sincere@april.biz",
      "address": {
        "street": "Kulas Light",
        "suite": "Apt. 556",
        "city": "Gwenborough",
        "zipcode": "92998-3874",
        "geo": {
          "lat": "-37.3159",
          "lng": "81.1496"
        }
      },
      "phone": "1-770-736-8031 x56442",
      "website": "hildegard.org",
      "company": {
        "name": "Romaguera-Crona",
        "catchPhrase": "Multi-layered client-server neural-net",
        "text": "harness real-time e-markets"
      }
    },
    {
      "id": 2,
      "name": "Ervin Howell",
      "username": "Antonette",
      "email": "Shanna@melissa.tv",
      "address": {
        "street": "Victor Plains",
        "suite": "Suite 879",
        "city": "Wisokyburgh",
        "zipcode": "90566-7771",
        "geo": {
          "lat": "-43.9509",
          "lng": "-34.4618"
        }
      },
      "phone": "010-692-6593 x09125",
      "website": "anastasia.net",
      "company": {
        "name": "Deckow-Crist",
        "catchPhrase": "Proactive didactic contingency",
        "text": "synergize scalable supply-chains"
      }
    }
  ]
```

Next, write your test. Remember to place the tests between the test methods (`setUp` and `tearDown`).

```
# test_data_process_unittest.py

from pandas.io.json import json_normalize

def test_data_structure(self):
    """
    The nested data structure should be flattened
    into a single record. We'll just use pandas
    for this.
    """
    expected_columns = ['id', 'name', 'username', 'email',
                        'address.street', 'address.suite',
                        'address.city', 'address.zipcode',
                        'address.geo.lat', 'address.geo.lng',
                        'phone', 'website', 'company.name',
                        'company.catchPhrase', 'company.text']


    dataframe = json_normalize(self.data)
    actual_columns = dataframe.columns
    self.assertEqual(set(expected_columns), set(actual_columns))
```

Now, run the tests from within the same directory:

```
$ python test_data_process_unittest.py

.
----------------------------------------------------------------------
Ran 1 test in 0.002s
```

```
OK
```

Awesome! This test is straightforward and passes. Make sure that the variables that you want on the flattened DataFrame match the output of the `json_normalize` function. Convert the Python lists into sets to do this. Also, for the first time, you use an assert method, `assertEqual`. This is one of many methods from unittest that can be used.

This tested function is imported from the `pandas.io.json` package, so you don't actually write it. As a result, be sure to include the package in your data_process.py file as an import. This is just to get your bearings and validate your process.

Commit your progress with the first test.

```
$ git add .

$ git commit -m 'add test for dataset structure and import json_normalize
function'
```

REQUIREMENT: SORT AND REMOVE DUPLICATE DATA

Your data process requires that your data does not have any duplicate records based on the Name variable. Also, the data must be sorted by the same variable. At this point, you want to convert any data objects into SAS data sets. So, use the utility functions provided by SASPy to do the conversions.

Write your next test in the test_data_process_unittest.py file:

```
# test_data_process_unittest.py

def test_remove_duplicate_data(self):
    """
    Remove duplicate data from an input dataset
    as defined by a `by variable` parameter.
    """

    data_with_duplicates = self.data + self.data
    actual = data_process.remove_duplicates(data_with_duplicates,
                                            'users', 'name').to_df()

    # convert both data objects to pandas dataframes
    expected = json_normalize(self.data)

    # compare
    assert_frame_equal(expected, actual)
```

The `assert_frame_equal` method is a testing helper provided by the pandas library for comparing DataFrame objects. Be sure to import it into your testing files.

Now, run the tests:

```
$ python test_data_process_unittest.py
.E
======================================================================
ERROR: test_remove_duplicate_data (__main__.TestDataProcess)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "test_data_process_unittest.py", line 84, in
test_remove_duplicate_data
    output_data = data_process.remove_duplicates(data_with_duplicates)
```

```
AttributeError: module 'data_process' has no attribute 'remove_duplicates'
----------------------------------------------------------------------
Ran 2 tests in 0.002s
FAILED (errors=1)
```

The output has highlighted that the `remove_duplicates` function that you used doesn't exist. Or, more simply, TDD! You wrote your test, and now you need a function to make it pass! You've now entered the territory of using SASPy. In doing so, you need to initialize the SAS session in data_process.py before you can use it.

Add the code below to data_process.py:

```
# data_process.py

import saspy
from pandas.io.json import json_normalize

# initialize our SAS session
sas = saspy.SASsession()

def remove_duplicates(data, name, sort_variable):
    """
    Sort and remove duplicates from normalized
    data using PROC SORT.  Variable name literals
    are used to account for nested data object
    labels.
    """
    dataframe = json_normalize(data)
    sas.df2sd(dataframe, table=name)
    sas.submit(
        f"""
        proc sort data={name} NODUPKEY;
            by DESCENDING '{sort_variable}'n;
        run;
        quit;
        """
    )
    return sas.sasdata(f"{name}")
```

Notice the usage of PROC SORT. Your function is wrapping Base SAS code. Now, you can verify that the correct data is returned based on the parameters that are used in the `remove_duplicates` function. You are using Python f-strings and running SAS code! Pretty cool. Also, you are using SAS name literal syntax for the Sort variable. The normalization of the nested data creates variable names that contain special characters (e.g., company.name). You want to make sure that you account for that, if possible.

f-strings are available in Python 3.6+. If you are using an earlier version, then the `str.format()` method will be of more use.

Now, run the tests again:

```
$ python test_data_process_unittest.py
..
----------------------------------------------------------------------
Ran 2 tests in 1.151s


OK
```

Awesome! The PROC SORT code is now parametrized and testable. Save a checkpoint here.

```
$ git add .

$ git commit -m 'passing test_remove_duplicate_data with corresponding
function'
```

REQUIREMENT: REMOVE AN UNUSED VARIABLE

This requirement is a standard data operation. You can continue to use a similar approach as the previous test now that SASPy is instantiated and already used in your data_process program.

Add the code below to the test_data_process_unittest.py file:

```
# test_data_process_unittest.py

def test_remove_variable(self):
    """
    Remove variable from an existing
    SAS dataset using SAS Data Step.
    """
    columns = ['id', 'email', 'address.street',
               'address.suite', 'address.city',
               'address.zipcode', 'address.geo.lat',
               'address.geo.lng', 'phone', 'website',
               'company.catchPhrase', 'company.text']


    sasdf = data_process.remove_variable('users', 'name')
    self.assertTrue('name' not in sasdf.to_df().columns)

    sasdf = data_process.remove_variable('users', 'username')
    self.assertTrue('username' not in sasdf.to_df().columns)

    sasdf = data_process.remove_variable('users', 'company.name')
    self.assertTrue('company.name' not in sasdf.to_df().columns)

    self.assertEqual(set(sasdf.to_df().columns), set(columns))

    # make sure an Error is raised if data != exist
    self.assertRaises(ValueError, remove_variable,
                      'bogus_data_name', 'name')
```

This test relies on five assertions to check that the removed variable name is not in the return data set's column list. The first three assertions drop a variable and assert that it is no longer present in the data set. The fourth assertion compares the final variable list against what is expected as defined by `columns`. The last assertion is `assertRaises`. This makes sure that a `ValueError` is thrown if the data set does not exist.

Let's give this a try.

```
$ python test_data_process_unittest.py

..E
======================================================================
ERROR: test_remove_variable (__main__.TestDataProcess)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "test_data_process_unittest.py", line 117, in test_remove_variable
    self.assertTrue('name' not in sasdf.columns)
AttributeError: 'NoneType' object has no attribute 'columns'
----------------------------------------------------------------------
Ran 3 tests in 1.153s
FAILED (errors=1)
```

The error is expected because the function does not exist. Add the code below to data_process.py:

```
# data_process.py

def remove_variable(name, variable):
    """
    Drop a dataset variable using
    Data Step.
    """
    if sas.exist(name):
        sas.submit(f"""
            data {name};
                set {name};
                drop '{variable}'n;
            run;"""
        )
        return sas.sasdata(f"{name}")


    raise ValueError('Dataset does not exist.')
```

An IF statement has been added to make sure that the data set exists before submitting the SAS DATA step code. The last assertion in your test, `assertRaises`, accounts for this. Now, let's run the tests again.

```
$ python test_data_process_unittest.py
...
----------------------------------------------------------------------
Ran 3 tests in 5.784s

OK
```

If you have been following along with the pytest tests, use this instead:

```
$ pytest -q test_data_process_pytest.py

...
3 passed in 7.62 seconds
```

Looks great! Now, commit that update.

```
$ git add .

$ git commit -m 'function to remove dataset variable and passing test'
```

CONCLUSION

Wow. That was fun!

This should give you a taste of what a TDD workflow can be using elements from both SAS and Python. But, don't stop here. Testing is a great development skill, and the possibilities are somewhat endless in how to integrate Python and SAS using SASPy.

Hopefully, this encourages you to continue to expand general-purpose programming concepts into the realm of data science and analytics.

APPENDIX A

data_process.py

```
# -*- coding: utf-8 -*-
# data_process.py

import saspy
from pandas.io.json import json_normalize

# initialize the SAS session
sas = saspy.SASsession()

def remove_duplicates(data, name, sort_variable):
    """
    Sort and deduplicate normalized data
    using PROC SORT.
    """

    dataframe = json_normalize(data)

    sas.df2sd(dataframe, table=name)

    sas.submit(
        f"""
        proc sort data={name} NODUPKEY;
            by DESCENDING '{sort_variable}'n;
        run;
        quit;
        """
    )

    return sas.sasdata(f"{name}")

def remove_variable(name, variable):
    """
    Drop a dataset variable using
    Data Step.
    """
    if sas.exist(name):
        sas.submit(f"""
            data {name};
                set {name};
                drop '{variable}'n;
            run;"""
        )
        return sas.sasdata(f"{name}")
    raise ValueError('Dataset does not exist.')
```

test_data_process_unittest.py

```
# -*- coding: utf-8 -*-
# test_data_process_unittest.py

import unittest
import data_process
from pandas.io.json import json_normalize
from pandas.util.testing import assert_frame_equal

class TestDataProcess(unittest.TestCase):

    def setUp(self):
        self.data = [
            {
                "id": 1,
                "name": "Leanne Graham",
                "username": "Bret",
                "email": "Sincere@april.biz",
                "address": {
                    "street": "Kulas Light",
                    "suite": "Apt. 556",
                    "city": "Gwenborough",
                    "zipcode": "92998-3874",
                    "geo": {
                        "lat": "-37.3159",
                        "lng": "81.1496"
                    }
                },
                "phone": "1-770-736-8031 x56442",
                "website": "hildegard.org",
                "company": {
                    "name": "Romaguera-Crona",
                    "catchPhrase": "Multi-layered client-server",
                    "text": "harness real-time e-markets"
                }
            },
            {
                "id": 2,
                "name": "Ervin Howell",
                "username": "Antonette",
                "email": "Shanna@melissa.tv",
                "address": {
                    "street": "Victor Plains",
                    "suite": "Suite 879",
                    "city": "Wisokyburgh",
                    "zipcode": "90566-7771",
                    "geo": {
                        "lat": "-43.9509",
                        "lng": "-34.4618"
                    }
                },
                "phone": "010-692-6593 x09125",
                "website": "anastasia.net",
                "company": {
                    "name": "Deckow-Crist",
                    "catchPhrase": "Proactive didactic contingency",
                    "text": "synergize scalable supply-chains"
                }
            }
        ]

    def test_data_structure(self):
        """
        The nested data structure should be flattened
        into a single record. We'll just use pandas
        for this.
        """
        expected_columns = ['id', 'name', 'username', 'email',
                            'address.street', 'address.suite',
                            'address.city', 'address.zipcode',
                            'address.geo.lat', 'address.geo.lng',
                            'phone', 'website', 'company.name',
                            'company.catchPhrase', 'company.text']

        dataframe = json_normalize(self.data)
        actual_columns = dataframe.columns
        self.assertEqual(set(expected_columns), set(actual_columns))

    def test_remove_duplicate_data(self):
        """
        Remove duplicate data from an input dataset
        as defined by a `by variable` parameter.
        """
        data_with_duplicates = self.data + self.data

        actual = data_process.remove_duplicates(data_with_duplicates,
                                                'users', 'name').to_df()

        # convert both data objects to pandas dataframes
        expected = json_normalize(self.data)

        # compare
        assert_frame_equal(expected, actual)

    def test_remove_variable(self):
        """
        Remove variable from an existing
        SAS dataset using SAS Data Step.
        """

        columns = ['id', 'email', 'address.street',
                   'address.suite', 'address.city',
                   'address.zipcode', 'address.geo.lat',
                   'address.geo.lng', 'phone', 'website',
                   'company.catchPhrase', 'company.text']

        sasdf = data_process.remove_variable('users', 'name')
        self.assertTrue('name' not in sasdf.to_df().columns)

        sasdf = data_process.remove_variable('users', 'username')
        self.assertTrue('username' not in sasdf.to_df().columns)

        sasdf = data_process.remove_variable('users', 'company.name')
        self.assertTrue('company.name' not in sasdf.to_df().columns)

        self.assertEqual(set(sasdf.to_df().columns), set(columns))

        # make sure an Error is raised if data != exist
        self.assertRaises(ValueError,
                          data_process.remove_variable,
                          'bogus_data_name', 'name')
    def tearDown(self):
        pass

if __name__ == '__main__':
    unittest.main()
```

test_data_process_pytest.py

```
# -*- coding: utf-8 -*-
# test_data_process_pytest.py

import pytest
import data_process
from pandas.io.json import json_normalize
from pandas.util.testing import assert_frame_equal

@pytest.fixture()
def set_up_data(request):
    data = [
        {
            "id": 1,
            "name": "Leanne Graham",
            "username": "Bret",
            "email": "Sincere@april.biz",
            "address": {
                "street": "Kulas Light",
                "suite": "Apt. 556",
                "city": "Gwenborough",
                "zipcode": "92998-3874",
                "geo": {
                    "lat": "-37.3159",
                    "lng": "81.1496"
                }
            },
            "phone": "1-770-736-8031 x56442",
            "website": "hildegard.org",
            "company": {
                "name": "Romaguera-Crona",
                "catchPhrase": "Multi-layered client-server",
                "text": "harness real-time e-markets"
            }
        },
        {
            "id": 2,
            "name": "Ervin Howell",
            "username": "Antonette",
            "email": "Shanna@melissa.tv",
            "address": {
                "street": "Victor Plains",
                "suite": "Suite 879",
                "city": "Wisokyburgh",
                "zipcode": "90566-7771",
                "geo": {
                    "lat": "-43.9509",
                    "lng": "-34.4618"
                }
            },
            "phone": "010-692-6593 x09125",
            "website": "anastasia.net",
            "company": {
                "name": "Deckow-Crist",
                "catchPhrase": "Proactive didactic contingency",
                "text": "synergize scalable supply-chains"
            }
        }
    ]

    def tear_down():
        pass

    request.addfinalizer(tear_down)

    return data

def test_data_structure(set_up_data):
    """
    The nested data structure should be flattened
    into a single record. We'll just use pandas
    for this.
    """
    expected_columns = ['id', 'name', 'username', 'email',
                        'address.street', 'address.suite',
                        'address.city', 'address.zipcode',
                        'address.geo.lat', 'address.geo.lng',
                        'phone', 'website', 'company.name',
                        'company.catchPhrase', 'company.text']

    dataframe = json_normalize(set_up_data)
    actual_columns = dataframe.columns
    assert set(expected_columns) == set(actual_columns)

def test_remove_duplicate_data(set_up_data):
    """
    Remove duplicate data from an input dataset
    as defined by a `by variable` parameter.
    """
    data_with_duplicates = set_up_data + set_up_data

    actual = data_process.remove_duplicates(data_with_duplicates,
                                            'users', 'name').to_df()

    # convert both data objects to pandas dataframes
    expected = json_normalize(set_up_data)

    # compare
    assert_frame_equal(expected, actual)

def test_remove_variable(set_up_data):
    """
    Remove variable from an existing
    SAS dataset using SAS Data Step.
    """
```

```
columns = ['id', 'email', 'address.street',
           'address.suite', 'address.city',
           'address.zipcode', 'address.geo.lat',
           'address.geo.lng', 'phone', 'website',
           'company.catchPhrase', 'company.text']

sasdf = data_process.remove_variable('users', 'name')
assert 'name' not in sasdf.to_df().columns

sasdf = data_process.remove_variable('users', 'username')
assert 'username' not in sasdf.to_df().columns

sasdf = data_process.remove_variable('users', 'company.name')
assert 'company.name' not in sasdf.to_df().columns

assert set(sasdf.to_df().columns) == set(columns)

# make sure an Error is raised if data != exist
with pytest.raises(ValueError):
    data_process.remove_variable('bogus_data_name', 'name')
```

REFERENCES

SAS Institute Inc. 2018. SASPy Documentation. Cary, NC: SAS Institute Inc. Available https://sassoftware.github.io/saspy.

“Fake Online REST API for Testing and Prototyping.” JSONPlaceholder. Available https://jsonplaceholder.typicode.com/users.

Python Software Foundation. 2018. “26.4.4 Organizing test code.” Available https://docs.python.org/3/library/unittest.html#organizing-tests.

Python Software Foundation. 2018. “assertEqual.” Available https://docs.python.org/3/library/unittest.html#unittest.

Python Software Foundation. 2018. “Formatted string literals.” Available https://docs.python.org/3/reference/lexical_analysis.html#f-strings.

SAS Institute. 2010. “Names in the SAS Language.” In *SAS 9.2 Language Reference: Concepts, Second Edition*. Available https://support.sas.com/documentation/cdl/en/lrcon/62955/HTML/default/viewer.htm#a000998953.htm.

SAS Institute Inc. 2010. “Length and Precision of Variables.” In SAS 9.2 Companion for Windows, Second Edition. Available http://support.sas.com/documentation/cdl/en/hostwin/63285/PDF/default/hostwin.pdf.

RECOMMENDED READING

- *SASPy Documentation*
- *unittest Documentation*
- *pytest Documentation*
- *pandas Documentation*
- *Git Documentation (https://git-scm.com/doc)*
- *Base SAS Programming Guide*
- *Base SAS Procedures Guide*

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Stephen Siegert
SAS Institute, Inc.
Stephen.Siegert@sas.com
@siegerts