Pemi Documentation

Sterling Paramore - InsideTrack, Inc.

Aug 04, 2020

GETTING STARTED

Pemi is a framework for building testable ETL processes and workflows.

CHAPTER

ONE

MOTIVATION

There are too many ETL tools. So why do we need another one? Many tools emphasize performance, or scalability, or building ETL jobs "code free" using GUI tools. One of the features often lacking is the ability to build testable data integration solutions. ETL can be exceedingly complex, and small changes to code can have large effects on output, and potentially devastating effects on the cleanliness of your data assets. Pemi was conceived with the goal of being able to build highly complex ETL workflows while maintaining testability.

This project aims to be largely agnostic to the way data is represented and manipulated. There is currently some support for working with Pandas DataFrames, in-database transformations (via SqlAlchemy) and Apache Spark DataFrames. Adding new data representations is a matter of creating a new Pemi DataSubject class.

Pemi does not orchestrate the execution of ETL jobs (for that kind of functionality, see Apache Airflow or Luigi). And as stated above, it does not force a developer to work with data using specific representations. Instead, the main role for Pemi fits in the space between manipulating data and job orchestration.

CHAPTER

TWO

INDEX

Getting Started

2.1 Install Pemi

Pemi can be installed from pip:

```
pip install pemi
```

2.2 Concepts and Features

2.2.1 Pipes

The principal abstraction in Pemi is the **Pipe**. A pipe can be composed of **Data Sources**, **Data Targets**, and other **Pipes**. When a pipe is executed, it collects data form the data sources, manipulates that data, and loads the results into the data targets. For example, here's a simple "Hello World" pipe. It takes a list of names in the form of a Pandas DataFrame and returns a Pandas DataFrame saying hello to each of them.

```
import pandas as pd

import pemi
from pemi.fields import *

class HelloNamePipe(pemi.Pipe):
    # Override the constructor to configure the pipe
    def __init__(self):
        # Make sure to call the parent constructor
        super().__init__()

        # Add a data source to our pipe - a pandas dataframe called 'input'
        self.source(
```

(continues on next page)

(continued from previous page)

```
            pemi.PdDataSubject,
            name='input',
            schema = pemi.Schema(
                name=StringField()
            )

        )

        # Add a data target to our pipe - a pandas dataframe called 'output'
        self.target(
            pemi.PdDataSubject,
            name='output'
        )

    # All pipes must define a 'flow' method that is called to execute the pipe
    def flow(self):
        self.targets['output'].df = self.sources['input'].df.copy()
        self.targets['output'].df['salutation'] = self.sources['input'].df['name'].
↪apply(
            lambda v: 'Hello ' + v
        )
```

To use the pipe, we have to create an instance of it:

```
pipe = HelloNamePipe()
```

and give some data to the source named “input”:

```
pipe.sources['input'].df = pd.DataFrame({
    'name': ['Buffy', 'Xander', 'Willow', 'Dawn']
})
```

name
Buffy
Xander
Willow
Dawn

The pipe performs the data transformation when the `flow` method is called:

```
pipe.flow()
```

The data target named “output” is then populated:

```
pipe.targets['output'].df
```

name	salutation
Buffy	Hello Buffy
Xander	Hello Xander
Willow	Hello Willow
Dawn	Hello Dawn

2.2.2 Data Subjects

Data Sources and **Data Targets** are both types of **Data Subjects**. A data subject is mostly just a reference to an object that can be used to manipulate data. In the [Pipes](#pipes) example above, we defined the data source called "input" as using the `pemi.PdDataSubject` class. This means that this data subject refers to a Pandas DataFrame object. Calling the *df* method on this data subject simply returns the Pandas DataFrame, which can be manipulated in all the ways that Pandas DataFrames can be manipulated.

Pemi supports 3 data subjects natively, but can easily be extended to support others. The 3 supported data subjects are

- `pemi.PdDataSubject` - Pandas DataFrames
- `pemi.SaDataSubject` - SQLAlchemy Engines
- `pemi.SparkDataSubject` - Apache Spark DataFrames

2.2.3 Schemas

A data subject can optionally be associated with a **Schema**. Schemas can be used to validate that the data object of the data subject conforms to the schema. This is useful when data is passed from the target of one pipe to the source of another because it ensures that downstream pipes get the data they are expecting.

For example, suppose we wanted to ensure that our data had fields called `id` and `name`. We would define a data subject like:

```
from pemi.fields import *

ds = pemi.PdDataSubject(
    schema=pemi.Schema(
        id=IntegerField(),
        name=StringField()
    )
)
```

If we provide the data subject with a dataframe that does not have a field:

```
df = pd.DataFrame({
    'name': ['Buffy', 'Xander', 'Willow']
})

ds.df = df
```

Then an error will be raised when the schema is validated (which happens automatically when data is passed between pipes, as we'll see below):

```
ds.validate_schema()
#=> MissingFieldsError: DataFrame missing expected fields: {'id'}
```

We'll also see later that defining a data subject with a schema also aids with writing tests. So while optional, defining data subjects with an associated schema is highly recommended.

Referencing data subjects in pipes

Data subjects are rarely defined outside the scope of a pipe as done in [Schemas](#schemas). Instead, they are usually defined in the constructor of a pipe as in [Pipes](#pipes). Two methods of the `pemi.Pipe` class are used to define data subjects: `source` and `target`. These methods allow one to specify the data subject class that the data subject will use, give it a name, assign a schema, and pass on any other arguments to the specific data subject class.

For example, if we were to define a pipe that was meant to use an Apache Spark dataframe as a source:

```
spark_session = ...
class MyPipe(pemi.Pipe):
    def __init__(self):
        super().__init__()

        self.source(
            pemi.SparkDataSubject,
            name='my_spark_source',
            schema=pemi.Schema(
                id=IntegerField(),
                name=StringField()
            ),
            spark=spark_session
        )
```

When `self.source` is called, it builds the data subject from the options provided and puts it in a dictionary that is associated with the pipe. The spark data frame can then be accessed from within the flow method as:

```
def flow(self):
    self.sources['my_spark_source'].df
```

2.2.4 Types of Pipes

Most user pipes will typically inherit from the main `pemi.Pipe` class. However, the topology of the pipe can classify it according to how it might be used. While the following definitions can be bent in some ways, they are useful for describing the purpose of a given pipe.

- A **Source Pipe** is a pipe that is used to extract data from some external system and convert it into a Pemi data subject. This data subject is the *target* of the *source* pipe.
- A **Target Pipe** is a pipe that is used to take a data subject and convert it into a form that can be loaded into some external system. This data subject is the *source* of the *target* pipe.
- A **Transformation Pipe** is a pipe that takes one or more data sources, transforms them, and delivers one more target sources.
- A **Job Pipe** is a pipe that is self-contained and does not specify any source or target data subjects. Instead, it is usually composed of other pipes that are connected to each other.

2.2.5 Pipe Connections

A pipe can be composed of other pipes that are each connected to each other. These connections for a directed acyclic graph (DAG). When then connections between all pipes are executed, the pipes that form the nodes of the DAG are executed in the order specified by the DAG (in parallel, when possible – parallel execution is made possible under the hood via Dask graphs). The data objects referenced by the node pipes' data subjects are passed between the pipes according.

As a minimal example showing how connections work, let's define a dummy source pipe that just generates a Pandas dataframe with some data in it:

```
class MySourcePipe(pemi.Pipe):
    def __init__(self):
        super().__init__()
```

(continues on next page)

(continued from previous page)

```
        self.target(
            pemi.PdDataSubject,
            name='main'
        )

    def flow(self):
        self.targets['main'].df = pd.DataFrame({
            'id': [1,2,3],
            'name': ['Buffy', 'Xander', 'Willow']
        })
```

And a target pipe that just prints the "salutation" field:

```
class MyTargetPipe(pemi.Pipe):
    def __init__(self):
        super().__init__()

        self.source(
            pemi.PdDataSubject,
            name='main'
        )

    def flow(self):
        for idx, row in self.sources['main'].df.iterrows():
            print(row['salutation'])
```

Now we define a job pipe that will connect the dummy source pipe to our hello world pipe and connect that to our dummy target pipe:

```
class MyJob(pemi.Pipe):
    def __init__(self):
        super().__init__()

        self.pipe(
            name='my_source_pipe',
            pipe=MySourcePipe()
        )
        self.connect('my_source_pipe', 'main').to('hello_pipe', 'input')

        self.pipe(
            name='hello_pipe',
            pipe=HelloNamePipe()
        )
        self.connect('hello_pipe', 'output').to('my_target_pipe', 'main')

        self.pipe(
            name='my_target_pipe',
            pipe=MyTargetPipe()
        )

    def flow(self):
        self.connections.flow()
```

In the flow method we call `self.connections.flow()`. This calls the `flow` method of each pipe defined in the connections graph and transfers data between them, in the order specified by the DAG.

The job pipe can be executed by calling its `flow` method:

```
MyJob().flow()
# => Hello Buffy
# => Hello Xander
# => Hello Willow
```

Furthermore, if you're running this in a Jupyter notebook, you can see a graph of the connections by running:

```
import pemi.dot
pemi.dot.graph(MyJob())
```

Referencing pipes in pipes

Referencing pipes within pipes works the same way as for data sources and targets. For example, if we wanted to run the `MyJob` job pipe and then look at the source of the "hello_pipe":

```
job  = MyJob()
job.flow()
job.pipes['hello_pipe'].sources['input'].df
```

2.3 Tests

Testing is an essential component of software development that is often neglected in the data and ETL world. Pemi was designed to fill that gap, and facilitate writing expressive data transformation tests. Pemi tests run on top of the popular Python testing framework Pytest.

The concepts involved in testing Pemi pipes include

- A **Scenario** describes the transformation that is being tested (a Pemi pipe), and the data sources and targets that are the subject of the test. Scenarios are composed of one more **Cases**.
- A **Case** is a set of **Conditions** and **Expectations** that describe how the pipe is supposed to function.
- A **Condition** describes how the data for a particular case is to be initialized – e.g., "when the field 'name' has the value 'Xander'".
- An **Expectation** describes the expected result, after the pipe has been executed – e.g., "then the field 'salutation' has the value 'Hello Xander'".

To see these concepts play out in an example, let's write a simple test for our `HelloNamePipe`. In this README, we'll talk through it in stages, but the full example can be found in tests/test_readme.py.

To aid with grouping cases into distinct scenarios, scenarios are defined using a context-manager pattern. So if we want to build a scenario called "Testing HelloNamePipe", we set that up like:

```
import pemi.testing as pt

with pt.Scenario(
    name='Testing HelloNamePipe',
    pipe=HelloNamePipe(),
    factories={
        'scooby': KeyFactory
    },
    sources={
        'input': lambda pipe: pipe.sources['input']
    },
```

(continues on next page)

(continued from previous page)

```
    targets={
        'output': lambda pipe: pipe.targets['output']
    },
    target_case_collectors={
        'output': pt.CaseCollector(subject_field='id', factory='scooby', factory_
↪field='scooby_id')
    }
) as scenario:
    #.... cases will go here ....
```

Let's quickly review the parameters of `Scenario` (see *testing* for more details):

- `name` - The name of the scenario.
- `pipe` - An instance of the pipe to be tested.
- `factories` - A dictionary containing key factories (more on this below).
- `sources`/`targets` - These are the sources/targets that will be the subject of testing. Defined as a dictionary, the keys are a short-hand for referencing the specific data subjects indicated in the values.
- `target_case_collectors` - Every target needs to have a case collector. The case collector links the field in a particular target to the field in the factory in which it was generated.

With testing, we're specifying how a data transformation is supposed to behave under certain conditions. Typically, we're focused on how subtle variations in the values of fields in the sources affect the values of fields in the targets. Each of these subtle variations defines a **Case** that was mentioned above. Now, it would be possible to have the tester execute the pipe for every case that needed to be tested. However, this could result in exceedingly slow tests, particularly when the overhead of loading data and executing a process is high (like it is for in-database transformations, and even more so for Apache Spark). Therefore, Pemi testing was built to only execute the pipe once for each scenario, regardless of how many cases are defined. This can only work if the records of the targets can be associated with a particular case in which the conditions and expectations are defined.

This brings us to **Factories and Case Collectors**. A Factory is a way of generating data used for testing. Pemi uses Factory Boy to generate keys for data records so that a record in a source table can be connected to a target record. In the HelloNamePipe example, we can define a Factory Boy key factory as:

```
class KeyFactory(factory.Factory):
    class Meta:
        model = dict
    id = factory.Sequence('scooby-{}'.format)
```

Every time a new instance of `KeyFactory` is created, the `id` column is given a new value:

```
>>> KeyFactory() #=> {'id': 'scooby-0'}
>>> KeyFactory() #=> {'id': 'scooby-1'}
>>> KeyFactory() #=> {'id': 'scooby-2'}
```

In the arguments of the `Scenario` class above, we see that a factory with the name of `scooby` is defined to use the `KeyFactory` class. Internally, the `Scenario` instance uses this factory to generate records, and keeps track of the **case** in which the factory instances were created. When the expectations of a case are asserted, the testing suite collects all of the ids in a target field and groups the target records into the specific cases. In the example above, the target case collector named `output` specifies that the target field `id` is generated from the `id` field via the factory called `scooby`. Note that there is no need for the field names to be the same. We'll see more about how this works below.

2.3.1 Column-Oriented Tests

With the hard part out of the way, we can now define our first test case. Cases are also defined using a context-manager pattern. To test that the salutations are behaving correctly we could write:

```
with scenario.case('Populating salutation') as case:
    case.when(
        pt.when.source_conforms_to_schema(
            scenario.sources['input'],
            {'id': scenario.factories['scooby']['id']}
        ),
        pt.when.source_field_has_value(scenario.sources['input'], 'name', 'Dawn')
    ).then(
        pt.then.target_field_has_value(scenario.targets['output'], 'salutation',
↪'Hello Dawn')
    )
```

The conditions set up the data to be tested:

- `pt.when.source_conforms_to_schema` - loads dummy data into the source called 'input', and uses the schema to determine the valid values that can be used. It also specifies that the `id` field on the source should come from the `id` field of the scenario's factory called `scooby`.
- `pt.when.source_field_has_value` - sets up the `name` field of the source data to have the value `Dawn`.

The expectations are then:

- `pt.then.target_field_has_value` - the target field `salutations` on the output has the value `Hello Dawn`. If we were to modify this value to be `Goodbye Dawn, don't let any vampires bite you neck`, then the test would fail.

This style of testing is referred to as "Column-Oriented" because we're only focused on the values of particular columns. We do not care about how the individual records are ordered or related to one another.

2.3.2 Row-Oriented Tests

Column-oriented tests are not always sufficient to describe data transformations. Sometimes we care about how rows are related. For example, we might need to describe how to drop duplicate records, or how to join two data sources together. To that end, we can write "Row-Oriented" tests. While the example we are working with here doesn't have any row operations, we can still write a test case that highlights how it can work.

```
with scenario.case('Dealing with many records') as case:
    ex_input = pemi.data.Table(
        '''
        | id       | name  |
        | -        | -     |
        | {sid[1]} | Spike |
        | {sid[2]} | Angel |
        '''.format(
            sid=scenario.factories['scooby']['id']
        )
    )

    ex_output = pemi.data.Table(
        '''
        | id       | salutation  |
        | -        | -           |
```

(continues on next page)

(continued from previous page)

```
        | {sid[1]} | Hello Spike |
        | {sid[2]} | Hello Angel |
        '''.format(
            sid=scenario.factories['scooby']['id']
        )
    )

    case.when(
        pt.when.example_for_source(scenario.sources['input'], ex_input)
    ).then(
        pt.then.target_matches_example(scenario.targets['output'], ex_output)
    )
```

In this case, we set up two data tables to show how the output records are related to the input records. Using examples built with `pemi.data.Table`, we can focus the test case on just those fields that we care about. If we had a source that had 80 fields in it, we would only need to define those that we care about for this particular test. Pemi will use the schema defined for that source to fill in the other fields with dummy data.

In this example, we use `scenario.factories['scooby']['id']` to generate ids for each record that will ensure that the ids created when defining the source data can be tied to records that are output in the target data. In `ex_input`, `{sid[1]}` will evaluate to some value generated by the factory (e.g., `scooby-9` or `scooby-12`, etc.). However, when `{sid[1]}` is referenced in the `ex_output`, it will use the same value that was generated for the `ex_input`.

A complete version of this test can be found in tests/test_readme.py.

2.3.3 Running Tests

Pemi tests require that the pytest package be installed in your project. Furthermore, you'll need to tell pytest that you want to use pemi tests by added the following to your `conftest.py`:

```
import pemi
pytest_plugins = ['pemi.pytest']
```

2.4 Tutorial

- TODO: Build an example repo that uses Pemi and has a full job
- TODO: Build a full integration job based off of the CSV job
- TODO: Guide for writing your own data subjects
- TODO: Guide for writing custom test conditions
- TODO: Row-focused tests vs column-focused tests
- TODO: Managing schemas as data flows through multiple pipes

2.5 Roadmap

- Future
  - SQLAlchemy/Spark as plugins

  - Streaming - I would like to be able to support streaming data subjects (like Kafka).
  - Auto-documentation - The testing framework should be able to support building documentation by collecting test scenario and case definitions. Documents could be built to look something like Gherkin.

Reference

- *pipe*
- *data_subject*
- *schema*
- *testing*

2.6 pipe

class pemi.pipe.**Pipe**(*, *name='self'*, ***params*)

A pipe is a parameterized collection of sources and targets which can be executed (flow).

Parameters

- **name** (*str*) – Assign a name to the pipe
- ****params** – Additional keyword parameters

name

The name of the pipe.

Type str

sources

A dictionary where the keys are the names of source data subjects and the values are instances of a data subject class.

Type dict

targets

A dictionary where the keys are the names of target data subjects and the values are instances of a data subject class.

Type dict

pipes

A dictionary referencing nested pipes where the keys are the names of the nested pipes and the values are the nested pipe instances. All pipes come with at least one nested pipe called 'self'.

Type dict

connections

Pemi connection object.

Type PipeConnection

connect(*from_pipe_name*, *from_subject_name*)

Connect one nested pipe to another

Parameters

- **from_pipe_name** (*str*) – Name of the nested pipe that contains the source of the connection.

- **from_subject_name** (*str*) – Name of the data subject in the nested pipe that contains the source of the connection. This data subject needs to be a *target* of the pipe referenced by *from_pipe_name*.

Returns the PipeConnection object.

Return type PipeConnection

Example

Connecting the target of one pipe to the source of another:

```
class MyPipe(pemi.Pipe):
    def __init__(self):
        super().__init__()

        self.pipe(
            name='get_awesome_data',
            pipe=GetAwesomeDataPipe()
        )
        self.connect('get_awesome_data', 'main').to('load_awesome_data', 'main
↪')

        self.pipe(
            name='load_awesome_data',
            pipe=LoadAwesomeDataPipe()
        )
```

flow()

Execute this pipe. This method is meant to be defined in a child class.

Example

A simple hello-world pipe:

```
class MyPipe(pemi.Pipe):
    def flow(self):
        print('hello world')
```

```
>>> MyPipe().flow()
'hello world'
```

from_pickle(*picklepipe=None*)

Recursively load all data subjects in all nested pipes from a pickled bytes object created by *to_pickle*.

Parameters **picklepipe** – The bytes object created by *to_pickle*

Returns

Return type self

Example

De-pickling a pickled pipe:

```
my_pipe = MyPipe()
pickled = my_pipe.to_pickle()

my_other_pipe = MyPipe().from_pickle(pickled)
```

pipe (*name*, *pipe*)

Defines a named pipe nested in this pipe.

Parameters

- **name** (*str*) – Name of the nested pipe.
- **pipe** (*pipe*) – The nested pipe instance.

Example

Creating a target:

```
class MyPipe(pemi.Pipe):
    def __init__(self):
        super().__init__()

        self.pipe(
            name='nested',
            pipe=pemi.Pipe()
        )
```

```
>>> MyPipe().pipes.keys()
['self', 'nested']
```

source (*subject_class*, *name*, *schema=None*, ***kwargs*)

Define a source data subject for this pipe.

Parameters

- **subject_class** (*class*) – The `DataSubject` class this source uses.
- **name** (*str*) – Name of this data subject.
- **schema** (*schema*) – `Schema` associated with this source.

Example

Creating a source:

```
class MyPipe(pemi.Pipe):
    def __init__(self):
        super().__init__()

        self.source(
            pemi.PdDataSubject,
            name='main'
        )
```

```
>>> MyPipe().sources.keys()
['main']
```

target (*subject_class*, *name*, *schema=None*, ***kwargs*)
Define a target data subject for this pipe.

Parameters

- **subject_class** (*class*) – The `DataSubject` class this target uses.
- **name** (*str*) – Name of this data subject.
- **schema** (*schema*) – `Schema` associated with this target.

Example

Creating a target:

```
class MyPipe(pemi.Pipe):
    def __init__(self):
        super().__init__()

        self.target(
            pemi.PdDataSubject,
            name='main'
        )
```

```
>>> MyPipe().targets.keys()
['main']
```

to_pickle (*picklepipe=None*)
Recursively pickle all of the data subjects in this and all nested pipes

Parameters **picklepipe** – A pickled representation of a pipe. Only used for recursion not meant to be set by user.

Returns A bytes object containing the pickled pipe.

Return type bytes

Example

Pickling a pipe:

```
>>> MyPipe.to_pickle()
b'.. <bytes> ..'
```

2.7 data_subject

class pemi.data_subject.**DataSubject** (*schema=None*, *name=None*, *pipe=None*)
A data subject is mostly just a schema and a generic data object Actually, it's mostly just a schema that knows which pipe it belongs to (if any) and can be converted from and to a pandas dataframe (really only needed for testing to work)

2.8 schema

class pemi.schema.**Schema**(**args*, ***kwargs*)
A schema is a thing.

metapply(*elem*, *func*)
Allows one to create/modify metadata elements using a function

Parameters

- **elem** (*str*) – Name of the metadata element to create or modify
- **func** (*func*) – Function that accepts a single `pemi.Field` argument and returns the value of the metadata element indicated by `elem`

Returns A new `pemi.Schema` with the updated metadata

Return type pemi.Schema

Example

Suppose we wanted to add some metadata to a schema that will be used to construct a SQL statement:

```
pemi.schema.Schema(
    id=StringField(),
    name=StringField()
).metapply(
    'sql',
    lambda field: 'students.{} AS student_{}'.format(field.name, field.name)
)
```

select(*func*)
Returns a new schema with the fields selected via a function (func) of the field

2.8.1 Fields

class pemi.fields.**Field**(*name=None*, ***metadata*)
A field is a thing that is inherited

class pemi.fields.**StringField**(*name=None*, ***metadata*)

class pemi.fields.**IntegerField**(*name=None*, ***metadata*)

class pemi.fields.**FloatField**(*name=None*, ***metadata*)

class pemi.fields.**DateField**(*name=None*, ***metadata*)

class pemi.fields.**DateTimeField**(*name=None*, ***metadata*)

class pemi.fields.**BooleanField**(*name=None*, ***metadata*)

class pemi.fields.**DecimalField**(*name=None*, ***metadata*)

class pemi.fields.**JsonField**(*name=None*, ***metadata*)

2.9 testing

Testing is described with examples in *Tests*.

class pemi.testing.**Scenario**(*name*, *pipe*, *factories*, *sources*, *targets*, *target_case_collectors*, *flow='flow'*, *selector=None*, *usefixtures=None*)

A **Scenario** describes the transformation that is being tested (a Pemi pipe), and the data sources and targets that are the subject of the test. Scenarios are composed of one more **Cases**.

Parameters

- **name** (*str*) – The name of a scenario. Multiple scenarios may be present in a file, but the names of each scenario must be unique.
- **pipe** (*pemi.Pipe*) – The Pemi pipe that is the main subject of the test. Test data will be provided to the sources of the pipe (defined below), and the pipe will be executed. Note that the pipe is only executed once per scenario.
- **flow** (*str*) – The name of the method used to execute the pipe (default: *flow*).
- **factories** (*dict*) – A dictionary where the keys are the names of factories and the values are FactoryBoy factories that will be used to generate unique keys.
- **sources** (*dict*) – A dictionary where the keys are the names of sources that will be the subjects of testing. The values are methods that accept the pipe referenced in the **pipe** argument above and return the data subject that will be used as a source.
- **targets** (*dict*) – A dictionary where the keys are the names of targets that will be the subjects of testing. The values are methods that accept the pipe referenced in the **pipe** argument above and return the data subject that will be used as a target.
- **target_case_collectors** (*dict*) – A dictionary where the keys are the names of the targets that will be the subjects of testing. The values are `CaseCollector` objects that tie a field in the scenario's target to the field in a given factory. Every named target needs to have a case collector.
- **selector** (*str*) – A string representing a regular expression. Any case names that **do not** match this regex will be excluded from testing.
- **usefixtures** (*str*) – Name of a Pytest fixture to use for the scenario. Often used for database setup/teardown options.

class pemi.testing.**Case**(*name*, *scenario*)

A **Case** is a set of **Conditions** and **Expectations** that describe how the pipe is supposed to function.

Parameters

- **name** (*str*) – The name of the case. The names of cases within a scenario must be unique.
- **scenario** (pemi.testing.Scenario) – The scenario object that this case is associated with.

expect_exception(*exception*)

Used to indicate that the test case is expected to fail with exception `exception`. If the test case raises this exception, then it will pass. If it does not raise the exception, then it will fail.

then(**funcs*)

Accepts a list of functions that are used to test the result data for a specific case. Each of the functions should accept one argument, which is the case object. See pemi.testing.then for examples.

when (**funcs*)
Accepts a list of functions that are used to set up the data for a specific case. Each of the functions should accept one argument, which is the case object. See pemi.testing.when for examples.

class pemi.testing.**when**
Contains methods used to set up conditions for a testing case.

static example_for_source (*source*, *table*)
Set specific rows and columns to specific values.

Parameters

- **source** (*scenario.sources[]*) – The scenario source data subject.
- **table** (*pemi.data.Table*) – Pemi data table to use for specifying data.

Example

Given a Pemi data table, specify rows and columns for the source *main*:

```
case.when(
    when.example_for_source(
        scenario.sources['main'],
        pemi.data.Table(
            '''
            | id       | name  |
            | -        | -     |
            | {sid[1]} | Spike |
            | {sid[2]} | Angel |
            '''.format(
                sid=scenario.factories['vampires']['id']
            )
        )
    )
)
```

static source_conforms_to_schema (*source*, *key_factories=None*)
Creates 3 records and fills out data for a data source subject that conforms to the data types specified by the data subject's schema.

Parameters

- **source** (*scenario.sources[]*) – The scenario source data subject.
- **key_factories** (*dict*) – A dictionary where the keys are the names of fields
- **the values are the field value generator originating from a scenario** (*and*) –
- **factory** (*key*) –

Example

For the source subject 'main', this will generate faked data that conforms to the schema defined for main. It will also populate the `id` field with values generated from the `id` field in the `vampires` factory:

```
case.when(
    when.source_conforms_to_schema(
        scenario.sources['main'],
```

(continues on next page)

(continued from previous page)

```
        {'id': scenario.factories['vampires']['id']}
    )
)
```

static source_field_has_value(*source*, *field*, *value*)

Sets the value of a specific field to a specific value.

Parameters

- **source** (*scenario.sources[]*) – The scenario source data subject.
- **field** (*str*) – Name of field.
- **value** (*str, iter*) – Value to set for the field.

Examples

Set the value of the field `name` to the string value `Buffy` in the scenario source `main`:

```
case.when(
    when.source_field_has_value(scenario.sources['main'], 'name', 'Buffy')
)
```

static source_fields_have_values(*source*, *mapping*)

Sets the value of a multiples fields to a specific values.

Parameters

- **source** (*scenario.sources[]*) – The scenario source data subject.
- **mapping** (*dict*) – Dictionary where the keys are the names of fields and the values are the values those fields are to be set to.

Examples

Set the value of the field `name` to the string value `Buffy` and the value of the field `vampires_slain` to `133` in the scenario source `main`:

```
case.when(
    when.source_fields_have_values(
        scenario.sources['main'],
        {
            'name': 'Buffy',
            'vampires_slain': 133
        }
    )
)
```

class pemi.testing.**then**

Contains methods used to test that actual outcome is equal to expected outcome.

static field_is_copied(*source*, *source_field*, *target*, *target_field*, *by=None*, *source_by=None*, *target_by=None*)

Asserts that a field value is copied from the source to the target.

Parameters

- **source** (*scenario.sources[]*) – The scenario source data subject.

- **source_field** (*str*) – The name of the source field.
- **target** (*scenario.targets[]*) – The scenario target data subject.
- **target_field** (*str*) – The name of the target field.
- **by** (*list*) – A list of field names to sort the data by before performing the comparison.
- **source_by** (*list*) – A list of field names to sort the source data by before performing the comparison (uses `by` if not given).
- **target_by** (*list*) – A list of field names to sort the target data by before performing the comparison (uses `by` if not given).

Examples

Asserts that the value of the source field `name` is copied to the target field `slayer_name`:

```
case.then(
    then.field_is_copied(
        scenario.sources['main'], 'name',
        scenario.targets['main'], 'slayer_name',
        by=['id']
    )
)
```

static fields_are_copied(*source*, *target*, *mapping*, *by=None*, *source_by=None*, *target_by=None*)

Asserts that various field values are copied from the source to the target.

Parameters

- **source** (*scenario.sources[]*) – The scenario source data subject.
- **target** (*scenario.targets[]*) – The scenario target data subject.
- **mapping** (*list*) – A list of tuples. Each tuple contains the source field name and target field name, in that order.
- **by** (*list*) – A list of field names to sort the data by before performing the comparison.
- **source_by** (*list*) – A list of field names to sort the source data by before performing the comparison (uses `by` if not given).
- **target_by** (*list*) – A list of field names to sort the target data by before performing the comparison (uses `by` if not given).

Examples

Asserts that the value of the source field `name` is copied to the target field `slayer_name` and `num` is copied to `vampires_slain`:

```
case.then(
    then.fields_are_copied(
        scenario.sources['main'],
        scenario.targets['main'],
        [
            ('name', 'slayer_name'),
            ('num', 'vampires_slain')
```

(continues on next page)

(continued from previous page)

```
        ],
        by=['id']
    )
)
```

static target_does_not_have_fields(*target*, *fields*)

Asserts that the target does not have certain fields.

Parameters

- **target** (*scenario.targets[]*) – The scenario target data subject.
- **fields** (*list*) – List of field names that should not be on the target.

Examples

Asserts that the scenario target `main` does not have the fields `sparkle_factor` or `is_werewolf`:

```
case.then(
    then.target_does_not_have_fields(
        scenario.targets['main'],
        ['sparkle_factor', 'is_werewolf']
    )
)
```

static target_field_has_value(*target*, *field*, *value*)

Asserts that a specific field has a specific value.

Parameters

- **target** (*scenario.targets[]*) – The scenario target data subject.
- **field** (*str*) – Name of field.
- **value** (*str*) – Value of the field that is expected.

Examples

Asserts that the value of the field `name` is set to the string value `Buffy` in the scenario target `main`:

```
case.then(
    then.target_field_has_value(scenario.targets['main'], 'name', 'Buffy')
)
```

static target_fields_have_values(*target*, *mapping*)

Asserts that multiple fields have specific values.

Parameters

- **target** (*scenario.targets[]*) – The scenario target data subject.
- **mapping** (*dict*) – Dictionary where the keys are the names of fields and the values are the expected values those fields.

Examples

Asserts that the value of the field `name` is the string value `Buffy` and the value of the field `vampires_slain` is `133` in the scenario target `main`:

```
case.then(
    then.target_fields_have_values(
        scenario.targets['main'],
        {
            'name': 'Buffy',
            'vampires_slain': 133
        }
    )
)
```

static target_has_fields(*target*, *fields*, *only=False*)

Asserts that the target has certain fields.

Parameters

- **target** (*scenario.targets[]*) – The scenario target data subject.
- **fields** (*list*) – List of field names that should not be on the target.
- **only** (*bool*) – Specifies whether the target should only have the fields listed. Raises an exception if there are additional fields.

Examples

Asserts that the scenario target `main` only has the fields `name` and `vampires_slain`:

```
case.then(
    then.target_has_fields(
        scenario.targets['main'],
        ['name', 'vampires_slain'],
        only=True
    )
)
```

static target_has_n_records(*target*, *expected_n*)

Asserts that the target has a specific number of records.

Parameters

- **target** (*scenario.targets[]*) – The scenario target data subject.
- **expected_n** (*int*) – The number of records expected.

Examples

Asserts that the scenario target `main` has 3 records:

```
case.then(then.target_has_n_records(scenario.targets['main'], 3)
```

static target_is_empty(*target*)

Asserts that the target has no records.

Parameters **target** (*scenario.targets[]*) – The scenario target data subject.

Examples

Asserts that the scenario target `errors` does not have any records:

```
case.then(then.target_is_empty(scenario.targets['errors'])
```

static target_matches_example(*target*, *expected_table*, *by=None*, *query=None*)

Asserts that a given target matches an example data table

Parameters

- **target** (*scenario.targets[]*) – The scenario target data subject.
- **expected_table** (*pemi.data.Table*) – Expected result data. If the table has fewer columns than the pipe generates, those extra columns are not considered in the comparison.
- **by** (*list*) – A list of field names to sort the result data by before performing the comparison.
- **query** (*string*) – A pandas query string that can be used to filter down target records prior to comparison

Examples

Asserts that the scenario target `main` conforms to the expected data:

```
case.then(
    then.target_matches_example(
        scenario.targets['main'],
        pemi.data.Table(
            '''
            | id       | name  |
            | -        | -     |
            | {sid[1]} | Spike |
            | {sid[2]} | Angel |
            '''.format(
                sid=scenario.factories['vampires']['id']
            )
        ),
        by=['id'] #esp important if the ids are generated randomly
    )
)
```

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

p

INDEX

W