pywrangler Documentation

Release 0.0.post0.dev312+g6638949

mansenfranzen

Jun 16, 2020

Content

New to pywrangler: If you are completely new to pywrangler and you want to know more about its motivation, scope and future directions, please start *here*.

Getting started: If you are already familiar with what pywrangler is about, you may dive into how to install and how to use it via the *user guide*.

Contribute: If you want to contribute to pywrangler or you want to know more about pywrangler's design decisions and architecture, please take a look at the *developer guide*.

CHAPTER 1

Meet pywrangler

1.1 Motivation

1.1.1 Problem formulation

The pydata ecosystem provides a rich set of tools (e.g pandas, dask, vaex, modin and pyspark) to handle most data wrangling tasks with ease. When dealing with data on a daily basis, however, one often encounters problems which go beyond the common dataframe API usage. They typically require a combination of multiple transformations and aggregations in order to achieve the desired outcome. For example, extracting intervals with given start and end values from raw time series is out of scope for native dataframe functionality.

1.1.2 Mission

pywrangler accomplishes such requirements with care while exposing so called **data wranglers**. A data wrangler serves a specific use case just like the one mentioned above. It takes one or more input dataframes, applies a computation which is usually built on top of existing dataframe API, and returns one or more output dataframes.

1.2 Scope

pywrangler can be seen as a **transform** part in common ETL pipelines. It is not concerned with data extraction or data loading. It assumes that data is already available in tabular format.

pywrangler addresses data transformations which are not covered by standard dataframe API. Such transformations are usually more complex and require careful testing. Hence, a major focus lies on **extensive testing** of provided implementations.

Apart from testing, a sophisticated documentation of how an algorithm works will increase user compliance. Accordingly, every data wrangler is supposed to provide a comprehensive and **visual step-by-step guide**.

pywrangler is not committed to a single computation backend like pandas. Instead, data wranglers are defined abstractly and need to be implemented concretely in regard to the specific computation backend. In general, all python

related computation engines are supported if corresponding implementations are provided (e.g pandas, dask, vaex, modin and pyspark). pywrangler attempts to provide at least one implementation for **smallish data** (single node computation e.g. pandas) and **largish data** (distributed computation e.g. pyspark).

Moreover, one computation engine may have several implementations with varying trade-offs, too. In order to identify trade-offs, pywrangler aims to offer **benchmarking utilities** to compare different implementations in regard to cpu and memory usage.

To make pywranlger **integrate** well with standard data wrangling workflows, data wranglers confirm to the scikit-learn API as a common standard for data transformation pipelines.

1.2.1 Goals

- describe data transformations independent of computation engine
- define data transformation requirements through extensive tests
- visually document implementation logic of data transformations step by step
- provide concrete implementations for small and big data computation engines
- add benchmarking utilities to compare different implementations to identify tradeoffs
- follow scikit-learn API for easy integration for data pipelines

1.2.2 Non-Goals

- always support all computation engines for a single data transformation
- handle *extract* and *load* stages of ETL pipelines

1.3 Rationale

Computation engines may come and go. Currently (June 2020), pandas is still very popular for single node computation. Vaex slowly catches up. Pyspark and dask are both very popular in the realm of distributed computation engines. There will be new engines in the future, perhaps pandas 2 or a computation engine originating from the Apache Arrow project as outlined here.

In any case, what remains is the careful description of data transformations. More importantly, computation backend independent tests manifest the requirements of data transformations. Moreover, such a specification may also be easily ported to languages other than python like R or Scala. This is probably the major lasting value of pywrangler.

1.4 Future directions

What has been totally neglected so far by the pywrangler project (as of June 2020), is the importance of SQL. Due to its declarative nature, SQL offers a computation engine independent way to formulate data transformations. They are applicable to any computation engine that supports sql. Therefore, one major goal is to add a sql backend that produces the required sql code to perform a specific data transformation.

CHAPTER 2

User guide

2.1 Installation

2.2 Tutorial

2.3 How To

CHAPTER 3

Developer guide

3.1 Setting up developer environment

3.1.1 Create and activate environment

First, create a separate virtual environment for pywrangler using the tool of your choice (e.g. conda):

```
conda create -n pywrangler_dev python=3.6
```

Next, make sure to activate your environment or to explicitly use the python interpreter of your newly created environment for the following commands:

```
source activate pywrangler_dev
```

3.1.2 Clone and install pywrangler

Install all dependencies

To clone pywrangler's master branch into the current working directory and to install it in development mode (editable) with all dependencies, run the following command:

```
pip install -e git+https://github.com/mansenfranzen/pywrangler.git@master
↪#egg=pywrangler[all] --src ''
```

You may separate cloning and installing:

```
git clone https://github.com/mansenfranzen/pywrangler.git
cd pywrangler
pip install -e .[all]
```

Install selected dependencies

You may not want to install all dependencies because they may be irrelevant for you. If you want to install only the minimal required development dependencies to develop pyspark data wranglers, switch `[all]` with `[dev,pyspark]`:

```
pip install -e git+https://github.com/mansenfranzen/pywrangler.git@master
↪#egg=pywrangler[dev,pyspark] --src ''
```

All available dependency packages are listed in the setup.cfg under `options.extras_require`.

3.2 Running tests

pywrangler uses pytest as a testing framework and tox for providing different testing environments.

3.2.1 Using pytest

If you want to run tests within your currently activated python environment, just run pytest (assuming you are currently in pywrangler's root directory):

```
pytest
```

This will run all tests. However, you may want to run only tests which are related to pyspark:

```
pytest -m pyspark
```

Same works with `pandas` and `dask`.

3.2.2 Using tox

pywrangler specifies many different environments to be tested to ensure that it works as expected across multiple python and varying computation engine versions.

If you want to test against all environments, simply run tox:

```
tox
```

If you want to run tests within a specific environment (e.g the most current computation engines for python 3.7), you will need provide the environment abbreviation directly:

```
tox -e py37-master
```

Please refer to the tox.ini to see all available environments.

3.3 Writing tests for data wranglers

When writing tests for data wranglers, it is highly recommended to use pywrangler's *DataTestCase*. It allows a computation engine independent test case formulation with three major goals in mind:

- Unify and standardize test data formulation across different computation engines.
- Let test data be as readable and maintainable as possible.

- Make writing data centric tests easy while reducing boilerplate code.

Note: Once a test is formulated with the *DataTestCase*, you may easily convert it to any computation backend. Behind the scences, an computation engine independent dataframe called *PlainFrame* converts the provided test data to the specific test engine.

3.3.1 Example

Lets start with an easy example. Imagine a data transformation for time series which increases a counter each time it encounters a specific target signal.

Essentially, a data tranfsormation focused test case requires two things: First, the input data which needs to be processed. Second, the output data which is expected as a result of the data wrangling stage:

```
from pywrangler.util.testing import DataTestCase

class IncreaseOneTest(DataTestCase):

    def input(self):
        """Provide the data given to the data wrangler."""

        cols = ["order:int", "signal:str"]
        data = [[    1,        "noise"],
                [    2,        "target"],
                [    3,        "noise"],
                [    4,        "noise"],
                [    5,        "target"]]

        return data, cols

    def output(self):
        """Provide the data expected from the data wrangler."""

        cols = ["order:int", "signal:str", "result:int"]
        data = [[    1,        "noise",           0],
                [    2,        "target",          1],
                [    3,        "noise",           1],
                [    4,        "noise",           1],
                [    5,        "target",          2]]

        return data, cols
```

That's all you need to do in order define a data test case. As you can see, typed columns are provided along with the corresponding data in a human readable format.

Next, let's write two different implementations using pandas and pyspark and test them against the `IncreaseOneTest`:

```
import pandas as pd
from pyspark.sql import functions as F, DataFrame, Window

def transform_pandas(df: pd.DataFrame) -> pd.DataFrame:
    df = df.sort_values("order")
    result = df["signal"].eq("target").cumsum()

```

(continues on next page)

(continued from previous page)

```
    return df.assign(result=result)


def transform_pyspark(df: DataFrame) -> DataFrame:
    target = F.col("signal").eqNullSafe("target").cast("integer")
    result = F.sum(target).over(Window.orderby("order"))

    return df.withColumn(result=result)

# instantiate test case
test_case = IncreaseOneTest()

# perform test assertions for given computation backends
test_case.test.pandas(transform_pandas)
test_case.test.pyspark(transform_pyspark)
```

The single test case `IncreaseOneTest` can be used to test multiple implementations based on different computation engines.

The *DataTestCase* and *PlainFrame* offer much more functionality which is covered in the corresponding reference pages. For example, you may use *PlainFrame* to seamlessly convert between pandas and pyspark dataframes. *DataTestCase* allows to formulate mutants of the input data which should cause the test to fail (hence covering multiple distinct but similar test data scenarios within the same data test case).

Note: *DataTestCase* currently supports only single input and output data wranglers. Data wranglers requiring multiple input dataframes or computing multiple output dataframes are not supported, yet.

3.4 Building & writing docs

3.5 Design & architecture

CHAPTER 4

pywrangler

4.1 pywrangler package

4.1.1 Subpackages

pywrangler.dask package

Submodules

pywrangler.dask.base module

pywrangler.dask.benchmark module

Module contents

pywrangler.pandas package

Subpackages

pywrangler.pandas.wranglers package

Submodules

pywrangler.pandas.wranglers.interval_identifier module

This module contains implementations of the interval identifier wrangler.

class pywrangler.pandas.wranglers.interval_identifier.**NaiveIterator**(*marker_column: str, marker_start: Any, marker_end: Any = <object object>, marker_start_use_first: bool = False, marker_end_use_first: bool = True, orderby_columns: Union[str, Iterable[str], None] = None, groupby_columns: Union[str, Iterable[str], None] = None, ascending: Union[bool, Iterable[bool]] = None, result_type: str = 'enumerated', target_column_name: str = 'iids'*)

Bases: pywrangler.pandas.wranglers.interval_identifier._BaseIntervalIdentifier

Most simple, sequential implementation which iterates values while remembering the state of start and end markers.

class pywrangler.pandas.wranglers.interval_identifier.**VectorizedCumSum**(*marker_column: str, marker_start: Any, marker_end: Any = <object object>, marker_start_use_first: bool = False, marker_end_use_first: bool = True, orderby_columns: Union[str, Iterable[str], None] = None, groupby_columns: Union[str, Iterable[str], None] = None, ascending: Union[bool, Iterable[bool]] = None, result_type: str = 'enumerated', target_column_name: str = 'iids'*)

Bases: pywrangler.pandas.wranglers.interval_identifier._BaseIntervalIdentifier

Sophisticated approach using multiple, vectorized operations. Using cumulative sum allows enumeration of intervals to avoid looping.

Module contents

Submodules

pywrangler.pandas.base module

This module contains the pandas base wrangler.

class pywrangler.pandas.base.**PandasSingleNoFit**
Bases: *pywrangler.pandas.base.PandasWrangler*

Mixin class defining *fit* and *fit_transform* for all wranglers with a single data frame input and output with no fitting necessary.

fit(*df: pandas.core.frame.DataFrame*)
Do nothing and return the wrangler unchanged.

This method is just there to implement the usual API and hence work in pipelines.

Parameters **df** (*pd.DataFrame*) –

fit_transform(*df: pandas.core.frame.DataFrame*) → pandas.core.frame.DataFrame
Apply fit and transform in sequence at once.

Parameters **df** (*pd.DataFrame*) –

Returns **result**

Return type pd.DataFrame

class pywrangler.pandas.base.**PandasWrangler**
Bases: *pywrangler.base.BaseWrangler*

Pandas wrangler base class.

computation_engine

pywrangler.pandas.benchmark module

This module contains benchmarking utility for pandas wranglers.

class pywrangler.pandas.benchmark.**PandasMemoryProfiler**(*wrangler: pywrangler.pandas.base.PandasWrangler*, *repetitions: int = 5*, *interval: float = 0.01*)
Bases: *pywrangler.benchmark.MemoryProfiler*

Approximate memory usage that a pandas wrangler instance requires to execute the *fit_transform* step.

As a key metric, *ratio* is computed. It refers to the amount of memory which is required to execute the *fit_transform* step. More concretely, it estimates how much more memory is used standardized by the input memory usage (memory usage increase during function execution divided by memory usage of input dataframes). In other words, if you have a 1GB input dataframe, and the *usage_ratio* is 5, *fit_transform* needs

5GB free memory available to succeed. A *usage_ratio* of 0.5 given a 2GB input dataframe would require 1GB free memory available for computation.

Parameters

- **wrangler** (*pywrangler.wranglers.pandas.base.PandasWrangler*) – The wrangler instance to be profiled.
- **repetitions** (*int*) – The number of measurements for memory profiling.
- **interval** (*float, optional*) – Defines interval duration between consecutive memory usage measurements in seconds.

measurements
The actual profiling measurements in bytes.

Type list

best
The best measurement in bytes.

Type float

median
The median of measurements in bytes.

Type float

worst
The worst measurement in bytes.

Type float

std
The standard deviation of measurements in bytes.

Type float

runs
The number of measurements.

Type int

baseline_change
The median change in baseline memory usage across all runs in bytes.

Type float

input
Memory usage of input dataframes in bytes.

Type int

output
Memory usage of output dataframes in bytes.

Type int

ratio
The amount of memory required for computation in units of input memory usage.

Type float

profile()
Contains the actual profiling implementation.

report()
Print simple report consisting of best, median, worst, standard deviation and the number of measurements.

profile_report()
Calls profile and report in sequence.

input
Returns the memory usage of the input dataframes in bytes.

output
Returns the memory usage of the output dataframes in bytes.

profile(**dfs*, ***kwargs*)
Profiles the actual memory usage given input dataframes *dfs* which are passed to *fit_transform*.

ratio
Refers to the amount of memory which is required to execute the *fit_transform* step. More concretely, it estimates how much more memory is used standardized by the input memory usage (memory usage increase during function execution divided by memory usage of input dataframes). In other words, if you have a 1GB input dataframe, and the *usage_ratio* is 5, *fit_transform* needs 5GB free memory available to succeed. A *usage_ratio* of 0.5 given a 2GB input dataframe would require 1GB free memory available for computation.

class pywrangler.pandas.benchmark.**PandasTimeProfiler**(*wrangler: pywrangler.pandas.base.PandasWrangler*, *repetitions: Union[None, int] = None*)

Bases: *pywrangler.benchmark.TimeProfiler*

Approximate time that a pandas wrangler instance requires to execute the *fit_transform* step.

Parameters

- **wrangler** (*pywrangler.wranglers.base.BaseWrangler*) – The wrangler instance to be profiled.
- **repetitions** (*None, int, optional*) – Number of repetitions. If *None*, *timeit.Timer.autorange* will determine a sensible default.

measurements
The actual profiling measurements in seconds.

Type list

best
The best measurement in seconds.

Type float

median
The median of measurements in seconds.

Type float

worst
The worst measurement in seconds.

Type float

std
The standard deviation of measurements in seconds.

Type float

runs
The number of measurements.

Type int

profile()
Contains the actual profiling implementation.

report()
Print simple report consisting of best, median, worst, standard deviation and the number of measurements.

profile_report()
Calls profile and report in sequence.

pywrangler.pandas.util module

This module contains utility functions (e.g. validation) commonly used by pandas wranglers.

pywrangler.pandas.util.**groupby**(*df: pandas.core.frame.DataFrame, groupby_columns: Union[str, Iterable[str], None]*) → pandas.core.groupby.generic.DataFrameGroupBy

Convenient function to group by a dataframe while taking care of optional groupby columns. Always returns a *DataFrameGroupBy* object.

Parameters

- **df** (*pd.DataFrame*) – Dataframe to check against.
- **groupby_columns** (*TYPE_COLUMNS*) – Columns to be grouped by.

Returns **groupby**

Return type DataFrameGroupBy

pywrangler.pandas.util.**sort_values**(*df: pandas.core.frame.DataFrame, order_columns: Union[str, Iterable[str], None], ascending: Union[bool, Iterable[bool]]*) → pandas.core.frame.DataFrame

Convenient function to return sorted dataframe while taking care of optional order columns and order (ascending/descending).

Parameters

- **df** (*pd.DataFrame*) – Dataframe to check against.
- **order_columns** (*TYPE_COLUMNS*) – Columns to be sorted.
- **ascending** (*TYPE_ASCENDING*) – Column order.

Returns **df_sorted**

Return type pd.DataFrame

pywrangler.pandas.util.**validate_columns**(*df: pandas.core.frame.DataFrame, columns: Union[str, Iterable[str], None]*)
Check that columns exist in dataframe and raise error if otherwise.

Parameters

- **df** (*pd.DataFrame*) – Dataframe to check against.
- **columns** (*iterable[str]*) – Columns to be validated.

pywrangler.pandas.util.**validate_empty_df**(*df: pandas.core.frame.DataFrame*)

Check for empty dataframe. By definition, wranglers operate on non empty dataframe. Therefore, raise error if dataframe is empty.

Parameters **df** (*pd.DataFrame*) – Dataframe to check against.

Module contents

pywrangler.pyspark package

Subpackages

pywrangler.pyspark.wranglers package

Submodules

pywrangler.pyspark.wranglers.interval_identifier module

Module contents

Submodules

pywrangler.pyspark.base module

pywrangler.pyspark.benchmark module

pywrangler.pyspark.pipeline module

pywrangler.pyspark.testing module

pywrangler.pyspark.types module

pywrangler.pyspark.util module

Module contents

pywrangler.util package

Subpackages

pywrangler.util.testing package

Submodules

pywrangler.util.testing.datatestcase module

This module contains the DataTestCase class.

class pywrangler.util.testing.datatestcase.**DataTestCase**(*engine: Optional[str] = None*)

Bases: object

Represents a data focused test case which has 3 major goals. First, it aims to unify and standardize test data formulation across different computation engines. Second, test data should be as readable as possible and should be maintainable in pure python. Third, it intends to make writing data centric tests as easy as possible while reducing the need of test case related boilerplate code.

To accomplish these goals, (1) it provides an abstraction layer for a computation engine independent data representation via *PlainFrame*. Test data is formulated once and automatically converted into the target computation engine representation. To ensure readability (2), test data may be formulated in column or row format with pure python objects. To reduce boilerplate code (3), it provides automatic assertion test functionality for all computation engines via *EngineAsserter*. Additionally, it allows to define mutants of the input data which should cause the test to fail (hence covering multiple distinct but similar test data scenarios within the same data test case).

Every data test case implements *input* and *output* methods. They resemble the data given to a test function and the computed data expected from the corresponding test function, respectively. Since the data needs to be formulated in a computation engine independent format, the *PlainFrame* is is used. For convenience, there are multiple ways of instantiation of a *PlainFrame* as a dict or tuple.

A dict requires typed column names as keys and values as values, which resembles the column format (define values column wise): >>> result = {“col1:int”: [1,2,3], “col2:str”: [“a”, “b”, “c”]}

A tuple may be returned in 2 variants. Both represent the row format (define values row wise). The most verbose way is to include data, column names and dtypes. >>> data = [[1, “a”], >>> [2, “b”], >>> [3, “b”]] >>> columns = [“col1”, “col2”] >>> dtypes = [“int”, “str”] >>> result = (data, columns, dtypes)

Second, dtypes may be provided simultaneously with column names as typed column annotations: >>> data = [[1, “a”], [2, “b”], [3, “b”]] >>> columns = [“col1:int”, “col2:str”] >>> result = (data, columns)

In any case, you may also provide a *PlainFrame* directly.

input
Represents the data input given to a data transformation function to be tested.

It needs to be implemented by every data test case.

mutants
Mutants describe modifications to the input data which should cause the test to fail.

Mutants can be defined in various formats. You can provide a single mutant like: >>> return ValueMutant(column=”col1”, row=0, value=3)

This is identical to the dictionary notation: >>> return {(“col1”, 0): 3}

If you want to provide multiple mutations within one mutant at once, you can use the *MutantCollection* or simply rely on the dictionary notation: >>> return {(“col1”, 2): 5, (“col2”, 1): “asd”}

If you want to provide multiple mutants at once, you may provide multiple dictionaries within a list: >>> [{(“col1”, 2): 5}, {(“col1”, 2): 3}]

Overall, all subclasses of *BaseMutant* are allowed to be used. You may also mix a specialized mutant with the dictionary notation: >>> [RandomMutant(), {(“col1”, 0): 1}]

output
Represents the data output expected from data transformation function to be tested.

It needs to be implemented by every data test case.

class pywrangler.util.testing.datatestcase.**EngineTester**(*parent: pywrangler.util.testing.datatestcase.DataTestCase*)

Bases: object

Composite of *DataTestCase* which resembles a collection of engine specific assertion functions. More concretely, for each computation engine, the input data from the parent data test case is passed to the function to be tested. The result is then compared to the output data of the parent data test case. Each engine may additionally provide engine specific functionality (like repartition for pyspark).

generic_assert (*test_func: Callable, test_kwargs: Optional[Dict[str, Any]], output_func: Callable*)
Generic assertion function for all computation engines which requires a computation engine specific output generation function.

Parameters

- **test_func** (*callable*) – A function that takes a pandas dataframe as the first keyword argument.
- **test_kwargs** (*dict, optional*) – Keyword arguments which will be passed to *test_func*.
- **output_func** (*callable*) – Output generation function which is computation engine specific.

generic_assert_mutants (*func_generate_output: Callable*)
Given a computation engine specific output generation function *generate_output*, iterate all available mutants and confirm their test assertion.

Parameters **func_generate_output** (*callable*) – Computation engine specific function that creates output PlainFrame given a mutant.

Raises AssertionError is raised if a mutant is not killed.

pandas (*test_func: Callable, test_kwargs: Optional[Dict[str, Any]] = None, merge_input: Optional[bool] = False, force_dtypes: Optional[Dict[str, str]] = None*)
Assert test data input/output equality for a given test function. Input data is passed to the test function and the result is compared to output data.

Some data test cases require the test function to add new columns to the input dataframe where correct row order is mandatory. In those cases, pandas test functions may only return new columns instead of adding columns to the input dataframe (modifying the input dataframe may result in performance penalties and hence should be prevented). This is special to pandas since it provides an index containing the row order information and does not require the input dataframe to be modified. However, data test cases are formulated to include the input dataframe within the output dataframe when row order matters because other engines may not have an explicit index column (e.g. pyspark). To account for this pandas specific behaviour, *merge_input* can be activated to make the assertion behave appropriately.

Parameters

- **test_func** (*callable*) – A function that takes a pandas dataframe as the first keyword argument.
- **test_kwargs** (*dict, optional*) – Keyword arguments which will be passed to *test_func*.
- **merge_input** (*bool, optional*) – Merge input dataframe to the computed result of the test function (inner join on index).
- **force_dtypes** (*dict, optional*) – Enforce specific dtypes for the returned result of the pandas test function. This may be necessary due to float casts when NaN values are present.

Raises AssertionError is thrown if computed and expected results do not match.

pyspark (*test_func: Callable, test_kwargs: Optional[Dict[str, Any]] = None, repartition: Union[int, List[str], None] = None*)
Assert test data input/output equality for a given test function. Input data is passed to the test function and

the result is compared to output data.

Pyspark's partitioning may be explicitly varied to test against different partitioning settings via *repartition*.

Parameters

- **test_func** (*callable*) – A function that takes a pandas dataframe as the first keyword argument.
- **test_args** (*iterable, optional*) – Positional arguments which will be passed to *test_func*.
- **test_kwargs** (*dict, optional*) – Keyword arguments which will be passed to *test_func*.
- **repartition** (*int, list, optional*) – Repartition input dataframe.

Raises AssertionError is thrown if computed and expected results do not match.

class pywrangler.util.testing.datatestcase.**TestCollection**(*datatestcases: Sequence[pywrangler.util.testing.datatestcase.Data test_kwargs: Optional[Dict[str, Dict[KT, VT]]] = None*)

Bases: object

Contains one or more DataTestCases. Provides convenient functions to be testable as a group (e.g. for pytest).

testcases
List of collected DataTestCase instances.

Type List[*DataTestCase*]

test_kwargs
A dict of optional parameter configuration which could be applied to collected DataTestCase instances. Keys refer to configuration names. Values refer to dicts which in turn represent keyword arguments.

Type dict, optional

names

pytest_parametrize_kwargs(*identifier: str*) → Callable
Convenient decorator to access provided *test_kwargs* and wrap them into *pytest.mark.parametrize*.

Parameters **identifier** (*str*) – The name of the test kwargs.

Examples

In the following example, *conf1* represents an available configuration to be tested. *param1* and *param2* will be passed to the actual test function.

```
>>> kwargs= {"conf1": {"param1": 1, "param2": 2}}
>>> test_collection = TestCollection([test1, test2])
>>> @test_collection.pytest_parametrize_testcases
>>> @test_collection.pytest_parametrize_kwargs("conf1")
>>> def test_dummy(testcase, conf1):
>>>     testcase().test.pandas(some_func, test_kwargs=conf1)
```

pytest_parametrize_testcases(*arg: Union[str, Callable]*) → Callable
Convenient decorator to wrap a test function which will be parametrized with all available DataTestCases in pytest conform manner.

Decorator can be called before wrapping the test function to supply a custom parameter name or can be used directly with the default parameter name (testcase). See examples for more.

Parameters **arg** (*str, callable*) – Name of the argument that will be used within the wrapped test function if decorator gets called.

Examples

If not used with a custom parameter name, *testcase* is used by default:

```
>>> test_collection = TestCollection([test1, test2])
>>> @test_collection.pytest_parametrize_testcases
>>> def test_dummy(testcase):
>>>     testcase().test.pandas(some_func)
```

If a custom parameter name is provided, it will be used:

```
>>> test_collection = TestCollection([test1, test2])
>>> @test_collection.pytest_parametrize_testcases("customname")
>>> def test_dummy(customname):
>>>     customname().test.pandas(some_func)
```

class pywrangler.util.testing.datatestcase.**TestDataConverter**(*name*, *bases*, *nmspc*)

Bases: type

Metaclass for DataTestCase. It's main purpose is to simplify the usage of DataTestCase and to avoid boilerplate code.

Essentially, it wraps and modifies the results of the *input*, *output* and *mutants* methods of DataTestCase.

For *input* and *output*, in converts the result to PlainFrame. For *mutants*, it converts the result to BaseMutant. Additionally, methods are wrapped as properties for simple dot notation access.

pywrangler.util.testing.datatestcase.**convert_method**(*func: Callable*, *convert: Callable*) → Callable

Helper function to wrap a given function with a given converter function.

pywrangler.util.testing.mutants module

This module contains the data mutants and mutation classes.

class pywrangler.util.testing.mutants.**BaseMutant**

Bases: object

Base class for all mutants. A mutant produces one or more mutations.

classmethod **from_dict**(*raw: dict*) → Union[pywrangler.util.testing.mutants.ValueMutant, pywrangler.util.testing.mutants.MutantCollection]

Factory method to conveniently convert a raw value into a Mutant instance. This is used for easy Mutant creation in dict format to avoid boilerplate code. Essentially, the dict format understands value mutations only. The key consists of a tuple of column and row and the value represents the actual new value, as follows:

```
>>> {("col1", 1): 0}
```

is identical to

```
>>> ValueMutant(column="col1", row=1, value=0)
```

Moreover, multiple mutations may be provided:

```
>>> {("col1", 1): 0, ("col1", 2): 1}
```

will result into

```
>>> MutantCollection([ValueMutant(column="col1", row=1, value=0),
>>>                   ValueMutant(column="col1", row=2, value=1)])
```

Parameters **raw** (*dict*) – Raw value mutant definitions.

Returns **mutant**

Return type *ValueMutant*, *MutantCollection*

classmethod from_multiple_any(*raw: Union[dict, BaseMutant, List[BaseMutant], None]*) → List[pywrangler.util.testing.mutants.BaseMutant]

Factory method to conveniently convert raw values into a list of Mutant objects.

Mutants can be defined in various formats. You can provide a single mutant like: >>> return ValueMutant(column="col1", row=0, value=3)

This is identical to the dictionary notation: >>> return {("col1", 0): 3}

If you want to provide multiple mutations within one mutant at once, you can use the *MutantCollection* or simply rely on the dictionary notation: >>> return {("col1", 2): 5, ("col2", 1): "asd"}

If you want to provide multiple mutants at once, you may provide multiple dictionaries within a list: >>> [{("col1", 2): 5}, {("col1", 2): 3}]

Overall, all subclasses of *BaseMutant* are allowed to be used. You may also mix a specialized mutant with the dictionary notation: >>> [RandomMutant(), {("col1", 0): 1}]

Parameters **raw** (*TYPE_RAW_MUTANTS*) –

Returns **mutants** – List of converted mutant instances.

Return type list

generate_mutations(*df: pywrangler.util.testing.plainframe.PlainFrame*) → List[pywrangler.util.testing.mutants.Mutation]

Returns all mutations produced by a mutant given a PlainFrame. Needs to be implemented by every Mutant. This is essentially the core of every mutant.

Parameters **df** (PlainFrame) – PlainFrame to generate mutations from.

Returns **mutations** – List of Mutation instances.

Return type list

get_params() → Dict[str, Any]

Retrieve all parameters set within the __init__ method.

Returns **param_dict** – Parameter names as keys and corresponding values as values

Return type dictionary

mutate(*df: pywrangler.util.testing.plainframe.PlainFrame*) → pywrangler.util.testing.plainframe.PlainFrame

Modifies given PlainFrame with inherent mutations and returns new, modifed PlainFrame.

Parameters **df** (PlainFrame) – PlainFrame to be modified.

Returns modified

Return type *PlainFrame*

class pywrangler.util.testing.mutants.**FunctionMutant**(*func: Callable*)
Bases: *pywrangler.util.testing.mutants.BaseMutant*

Represents a Mutant which wraps a function that essentially generates mutations.

func
A function to be used as a mutation generation method.

Type callable

generate_mutations(*df: pywrangler.util.testing.plainframe.PlainFrame*) → List[pywrangler.util.testing.mutants.Mutation]
Delegates the mutation generation to a custom function to allow all possible mutation generation.

Parameters **df** (PlainFrame) – PlainFrame to generate mutations from.

Returns **mutations** – List of Mutation instances.

Return type list

class pywrangler.util.testing.mutants.**ImmutableMutation**(*column*, *row*, *value*)
Bases: tuple

column
Alias for field number 0

row
Alias for field number 1

value
Alias for field number 2

class pywrangler.util.testing.mutants.**MutantCollection**(*mutants: Sequence[T_co]*)
Bases: *pywrangler.util.testing.mutants.BaseMutant*

Represents a collection of multiple Mutant instances.

mutants
List of mutants.

Type sequence

generate_mutations(*df: pywrangler.util.testing.plainframe.PlainFrame*) → List[pywrangler.util.testing.mutants.Mutation]
Collects all mutations generated by included Mutants.

Parameters **df** (PlainFrame) – PlainFrame to generate mutations from.

Returns **mutations** – List of Mutation instances.

Return type list

class pywrangler.util.testing.mutants.**Mutation**
Bases: *pywrangler.util.testing.mutants.ImmutableMutation*

Resembles a single mutation of a dataframe which essentially represents a data modification of a single cell of a dataframe. Hence, a mutation is fully specified via three values: a column, a row and a new value.

The column is always given via label (string). The row is always given via an index (integer) because plainframe does not have labeled indices. The row index starts with 0. The new value may be of any type.

key

class pywrangler.util.testing.mutants.**RandomMutant**(*count: int = 1*, *columns: Sequence[str] = None*, *rows: Sequence[int] = None*, *seed: int = 1*)

Bases: *pywrangler.util.testing.mutants.BaseMutant*

Creates random mutations with naive values for supported dtypes of PlainFrame. Randomness is controlled via an explicit seed to allow reproducibility. Mutation generation may be narrowed to given rows or columns. The number of distinct mutations may also be specified.

count
The number of mutations to be executed.

Type int, optional

columns
Restrict mutations to provided columns, if given.

Type sequence, optional

rows
Restrict mutations to provided rows, if given.

Type sequence, optional

seed
Set the seed for the random generator.

Type int, optional

generate_mutation(*df: pywrangler.util.testing.plainframe.PlainFrame*, *column: str*, *row: int*) → pywrangler.util.testing.mutants.Mutation
Generates single mutation from given PlainFrame for a given candidate. A candidate is specified via column name and row index.

Parameters

- **df** (PlainFrame) – PlainFrame to generate mutations from.
- **column** (*str*) – Identifies relevant column of mutation.
- **row** (*int*) – Identifies relevant row of mutation.

Returns mutation

Return type *Mutation*

generate_mutations(*df: pywrangler.util.testing.plainframe.PlainFrame*) → List[pywrangler.util.testing.mutants.Mutation]
Generates population of all possible mutations and draws a sample of it.

Parameters **df** (PlainFrame) – PlainFrame to generate mutations from.

Returns **mutations** – List of Mutation instances.

Return type list

class pywrangler.util.testing.mutants.**ValueMutant**(*column: str*, *row: int*, *value: Any*)

Bases: *pywrangler.util.testing.mutants.BaseMutant*

Represents a Mutant with a single mutation.

column
Name of the column.

Type str

row
Index of the row.

Type int

value
The new value to be used.

Type Any

generate_mutations(*df: pywrangler.util.testing.plainframe.PlainFrame*) → List[pywrangler.util.testing.mutants.Mutation]
Returns a single mutation.

Parameters **df** (PlainFrame) – PlainFrame to generate mutations from.

Returns **mutations** – List of Mutation instances.

Return type list

pywrangler.util.testing.plainframe module

This module contains the PlainFrame and PlainColumn classes.

class pywrangler.util.testing.plainframe.**ConverterFromPandas**(*df: pandas.core.frame.DataFrame*)
Bases: object

Convert pandas dataframe into plain PlainFrame.

convert_series(*column: str*, *dtype: str*) → List[Union[bool, int, float, str, datetime.datetime, pywrangler.util.testing.plainframe.NullValue]]
Converts a column of pandas dataframe into PlainFrame readable format with specified dtype (np.NaN to NULL, timestamps to datetime.datetime).

Parameters

- **column** (*str*) – Identifier for column.
- **dtype** (*str*) – Dtype identifier.

Returns **values** – Converted pandas series as plain python objects.

Return type list

static force_dtype(*series: pandas.core.series.Series*, *dtype: str*) → List[Union[bool, int, float, str, datetime.datetime, pywrangler.util.testing.plainframe.NullValue]]
Attempts to convert values to provided type.

Parameters

- **series** (*pd.Series*) – Values in pandas representation.
- **dtype** (*str*) – Dtype identifier.

Returns **values** – Converted pandas series as plain python objects.

Return type list

get_forced_dtypes(*dtypes: Union[List[str], Dict[str, str]]*) → Dict[str, str]
Validate user provided *dtypes* parameter.

Parameters **dtypes** (*list, dict*) – If list is provided, each value represents a dtype and maps to one column of the dataframe in order. If dict is provided, keys refer to column names and values represent dtypes.

Returns **dtypes_forced** – Keys refer to column names and values represent dtypes.

Return type dict

get_inferred_dtypes(*dtypes_validated: Dict[str, str]*) → Dict[str, str]
Get all dtypes for columns which have not been provided, yet. Assumes that columns of dtype object are not present. Raises type error otherwise.

Parameters **dtypes_validated** (*dict*) – Represents already given column/dtype pairs. Keys refer to column names and values represent dtypes.

Returns **dtypes_inferred** – Keys refer to column names and values represent dtypes.

Return type dict

get_object_dtypes(*dtypes_validated: Dict[str, str]*) → Dict[str, str]
Inspect all columns of dtype object and ensure no mixed dtypes are present. Raises type error otherwise. Ignores columns for which dtypes are already explicitly set.

Parameters **dtypes_validated** (*dict*) – Represents already given column/dtype pairs. Keys refer to column names and values represent dtypes.

Returns **dtypes_object** – Keys refer to column names and values represent dtypes.

Return type dict

static inspect_dtype(*series: pandas.core.series.Series*) → str
Get appropriate dtype of pandas series. Checks against bool, int, float and datetime. If dtype object is encountered, raises type error.

Parameters **series** (*pd.Series*) – pandas series column identifier.

Returns **dtype** – Inferred dtype as string.

Return type str

inspect_dtype_object(*column: str*) → str
Inspect series of dtype object and ensure no mixed dtypes are present. Try to infer actual dtype after removing np.NaN distinguishing dtypes bool and str.

Parameters **column** (*str*) – Identifier for column.

Returns **dtype** – Inferred dtype as string.

Return type str

class pywrangler.util.testing.plainframe.**ConverterFromPySpark**(*df: pyspark.sql.DataFrame*)

Bases: object

Convert pyspark dataframe into PlainFrame.

TYPE_MAPPING = {'bigint': 'int', 'boolean': 'bool', 'date': 'datetime', 'double':

static convert_null(*values: Iterable[T_co]*) → list
Substitutes python *None* with NULL values.

Parameters **values** (*iterable*) –

get_column_dtypes() → Tuple[List[str], List[str]]
Get column names and corresponding dtypes.

class pywrangler.util.testing.plainframe.**ConverterToPandas**(*parent: pywrangler.util.testing.plainframe.PlainColumn*)

Bases: object

Collection of pandas conversion methods as a composite of PlainColumn. It handles pandas specifics like the missing distinction between NULL and NaN.

sanitized
Replaces any Null values with np.NaN to conform pandas' missing value convention.

class pywrangler.util.testing.plainframe.**ConverterToPySpark**(*parent: pywrangler.util.testing.plainframe.PlainColumn*)

Bases: object

Collection of pyspark conversion methods as a composite of PlainColumn. It handles spark specifics like NULL as None and proper type matching.

sanitized
Replaces Null values with None to conform pyspark missing value convention.

class pywrangler.util.testing.plainframe.**EqualityAsserter**(*parent: pywrangler.util.testing.plainframe.PlainFrame*)

Bases: object

Collection of equality assertions as a composite of PlainFrame. It contains equality tests in regard to number of rows, columns, dtypes etc.

class pywrangler.util.testing.plainframe.**NullValue**
Bases: object

Represents null values. Provides operator comparison functions to allow sorting which is required to determine row order of data tables.

class pywrangler.util.testing.plainframe.**PlainColumn**(**args*, ***kwargs*)
Bases: pywrangler.util.testing.plainframe._ImmutablePlainColumn

Represents an immutable column of a PlainFrame consisting of a name, dtype and values. Ensures type validity.

Instantiation should be performed via *from_plain* factory method which adds preprocessing steps to ensure type correctness.

In addition, it contains conversion methods for all supported computation engines.

classmethod from_plain(*name: str, dtype: str, values: Sequence[T_co]*) → pywrangler.util.testing.plainframe.PlainColumn
Factory method to instantiate PlainColumn from plain objects. Adds preprocessing steps for float and datetime types.

Parameters

- **name** (*str*) – Name of the column.
- **dtype** (*str*) – Data type of the column. Must be one of bool, int, float, str or datetime.
- **values** (*sequence*) – sequence of values

Returns plaincolumn

Return type *PlainColumn*

has_nan
Signals presence of NaN values.

has_null
Signals presence of NULL values.

modify(*modifications: Dict[int, Any]*) → pywrangler.util.testing.plainframe.PlainColumn
Modifies PlainColumn and returns new instance. Modification does not change dtype, name or the number of values. One or more values will be modified.

Parameters **modifications** (*dict*) – Dictionary containing modifications with keys representing row indicies and values representing new values.

Returns modified

Return type *PlainColumn*

to_pandas
Composite for conversion functionality to pandas.

to_pyspark
Composite for conversion functionality to pyspark.

typed_column
Return typed column annotation of PlainColumn.

class pywrangler.util.testing.plainframe.**PlainFrame**(**args*, ***kwargs*)
Bases: pywrangler.util.testing.plainframe._ImmutablePlainFrame

Resembles an immutable dataframe in plain python. Its main purpose is to represent test data that is independent of any computation engine specific characteristics. It serves as a common baseline format. However, in order to be usable for all engines, it can be converted to and from any computation engine's data representation. This allows to formulate test data in an engine independent way only once and to employ it for all computation engines simultaneously.

The main focus lies on simple but correct data representation. This includes explicit values for NULL and NaN. Each column needs to be typed. Available types are integer, boolean, string, float and datetime. For simplicity, all values will be represented as plain python types (no 3rd party). Hence, it is not intended to be used for large amounts of data due to its representation in plain python objects.

There are several limitations. No index column is supported (as in pandas). Mixed dtypes are not supported (like dtype object in pandas). No distinction is made between int32/int64 or single/double floats. Only primitive/atomic types are supported (pyspark's ArrayType or MapType are currently not supported).

Essentially, a PlainFrame consists of only 3 attributes: column names, column types and column values. In addition, it provides conversion methods for all computation engines. It does not offer any computation methods itself because it only represents data.

assert_equal
Return equality assertion composite.

columns
Return column names of PlainFrame.

data
Return data of PlainFrame row wise.

dtypes
Return dtypes of columns of PlainFrame.

classmethod **from_any**(*raw: Union[PlainFrame, dict, tuple, pandas.core.frame.DataFrame, pyspark.sql.DataFrame]*) → PlainFrame
Instantiate *PlainFrame* from any possible type supported.

Checks following scenarios: If PlainFrame is given, simply pass. If dict is given, call constructor from dict. If tuple is given, call constructor from plain. If pandas dataframe is given, call from pandas. If spark dataframe is given, call from pyspark.

Parameters **raw** (*TYPE_ANY_PF*) – Input to be converted.

Returns plainframe

Return type *PlainFrame*

classmethod from_dict(*data: collections.OrderedDict[str, Sequence]*) → PlainFrame
Instantiate *PlainFrame* from ordered dict. Assumes keys to be column names with type annotations and values to be values.

Parameters **data** (*dict*) – Keys represent typed column annotations and values represent data values.

Returns **plainframe**

Return type *PlainFrame*

classmethod from_pandas(*df: pandas.core.frame.DataFrame, dtypes: Union[List[str], Dict[str, str]] = None*) → pywrangler.util.testing.plainframe.PlainFrame
Instantiate *PlainFrame* from pandas DataFrame.

Parameters

- **df** (*pd.DataFrame*) – Dataframe to be converted.
- **dtypes** (*list, dict, optional*) – If list is provided, each value represents a dtype and maps to one column of the dataframe in given order. If dict is provided, keys refer to column names and values represent dtypes.

Returns **datatable** – Converted dataframe

Return type *PlainFrame*

classmethod from_plain(*data: Sequence[Sequence[T_co]], columns: Sequence[str], dtypes: Optional[Sequence[str]] = None, row_wise: bool = True*)
Instantiate *PlainFrame* from plain python objects. Dtypes have to be provided either via *columns* as typed column annotations or directly via *dtypes*. Typed column annotations are a convenient way to omit the *dtypes* parameter while specifying dtypes directly with the *columns* parameter.

An exmaple of a typed column annotation is as follows: >>> columns = [“col_a:int”, “col_b:str”, “col_c:float”]

Abbreviations may also be used like: >>> columns = [“col_a:i”, “col_b:s”, “col_c:f”]

For a complete abbreviation mapping, please see *TYPE_ABBR*.

Parameters

- **data** (*list*) – List of iterables representing the input data.
- **columns** (*list*) – List of strings representing the column names. Typed annotations are allowed to be used here and will be checked of *dtypes* is not provided.
- **dtypes** (*list, optional*) – List of column types.
- **row_wise** (*bool, optional*) – By default, assumes *data* is provided in row wise format. All values belonging to the same row are stored in the same array. In contrast, if *row_wise* is False, column wise alignment is assumed. In this case, all values belonging to the same column are stored in the same array.

Returns **plainframe**

Return type *PlainFrame*

classmethod from_pyspark(*df: pyspark.sql.DataFrame*) → PlainFrame
Converts pandas dataframe into TestDataTabble.

Parameters **df** (*pyspark.sql.DataFrame*) – Dataframe to be converted.

Returns **datatable** – Converted dataframe

Return type *PlainFrame*

get_column (*name: str*) → pywrangler.util.testing.plainframe.PlainColumn
Convenient access to PlainColumn via column name.

Parameters **name** (*str*) – Label identifier for columns.

Returns **column**

Return type *PlainColumn*

modify (*modifications: Dict[str, Dict[int, Any]]*) → pywrangler.util.testing.plainframe.PlainFrame
Modifies PlainFrame and returns new instance. Modification does not change dtype, name or the number of values of defined columns. One or more values of one or more columns will be modified.

Parameters **modifications** (*dict*) – Contains modifications. Keys represent column names and values represent column specific modifications.

Returns **modified**

Return type *PlainFrame*

n_cols
Returns the number columns.

n_rows
Return the number of rows.

to_dict () → collections.OrderedDict[str, tuple]
Converts PlainFrame into dictionary with key as typed columns and values as data.

Returns **table_dict**

Return type OrderedDict

to_pandas () → pandas.core.frame.DataFrame
Converts test data table into a pandas dataframe.

to_plain () → Tuple[List[List[T]], List[str], List[str]]
Converts PlainFrame into tuple with 3 values (data, columns, dtypes).

Returns

Return type data, columns, values

to_pyspark ()
Converts test data table into a pandas dataframe.

pywrangler.util.testing.util module

pywrangler.util.testing.util.**concretize_abstract_wrangler** (*abstract_class: Type[CT_co]*) → Type[CT_co]
Makes abstract wrangler classes instantiable for testing purposes by implementing abstract methods of *BaseWrangler*.

Parameters **abstract_class** (*Type*) – Class object to inherit from while overriding abstract methods.

Returns **concrete_class** – Concrete class usable for testing.

Return type Type

Module contents

Submodules

pywrangler.util.dependencies module

This module contains functionality to check optional and mandatory imports. It aims to provide useful error messages if optional dependencies are missing.

pywrangler.util.dependencies.**is_available**(**deps*) → bool
Check if given dependencies are available.

Parameters **deps** (*list*) – List of dependencies to check.

Returns **available**

Return type bool

pywrangler.util.dependencies.**raise_if_missing**(*import_name*)
Checks for available import and raises with more detailed error message if not given.

Parameters **import_name** (*str*) –

pywrangler.util.dependencies.**requires**(**deps*) → Callable
Decorator for callables to ensure that required dependencies are met. Provides more useful error message if dependency is missing.

Parameters **deps** (*list*) – List of dependencies to check.

Returns **decorated**

Return type callable

Examples

```
>>> @requires("dep1", "dep2")
>>> def func(a):
>>>     return a
```

pywrangler.util.helper module

This module contains commonly used helper functions or classes.

pywrangler.util.helper.**get_param_names**(*func: Callable*, *ignore: Optional[Iterable[str]] = None*) → List[str]
Retrieve all parameter names for given function.

Parameters

- **func** (*Callable*) – Function for which parameter names should be retrieved.
- **ignore** (*iterable, None, optional*) – Parameter names to be ignored. For example, *self* for *__init__* functions.

Returns **param_names** – List of parameter names.

Return type list

pywrangler.util.sanitizer module

This module contains common helper functions for sanity checks and conversions.

pywrangler.util.sanitizer.**ensure_iterable**(*values: Any*, *seq_type: Type[CT_co] = <class 'list'>*, *retain_none: bool = False*) → Union[List[Any], Tuple[Any], None]

For convenience, some parameters may accept a single value (string for a column name) or multiple values (list of strings for column names). Other functions always require a list or tuple of strings. Hence, this function ensures that the output is always an iterable of given *constructor* type (list or tuple) while taking care of exceptions like strings.

Parameters

- **values** (*Any*) – Input values to be converted to tuples.
- **seq_type** (*type*) – Define return container type.
- **retain_none** (*bool, optional*) – Define behaviour if None is passed. If True, returns None. If False, returns empty

Returns iterable

Return type seq_type

pywrangler.util.types module

This module contains type definitions.

Module contents

4.1.2 Submodules

4.1.3 pywrangler.base module

This module contains the BaseWrangler definition and the wrangler base classes including wrangler descriptions and parameters.

class pywrangler.base.**BaseWrangler**

Bases: abc.ABC

Defines the basic interface common to all data wranglers.

In analogy to sklearn transformers (see link below), all wranglers have to implement *fit*, *transform* and *fit_transform* methods. In addition, parameters (e.g. column names) need to be provided via the *__init__* method. Furthermore, *get_params* and *set_params* methods are required for grid search and pipeline compatibility.

The *fit* method contains optional fitting (e.g. compute mean and variance for scaling) which sets training data dependent transformation behaviour. The *transform* method includes the actual computational transformation. The *fit_transform* either applies the former methods in sequence or adds a new implementation of both with better performance. The *__init__* method should contain any logic behind parameter parsing and conversion.

In contrast to sklearn, wranglers do only accept dataframes like objects (like pandas/pyspark/dask dataframes) as inputs to *fit* and *transform*. The relevant columns and their respective meaning is provided via the *__init__*

method. In addition, wranglers may accept multiple input dataframes with different shapes. Also, the number of samples may also change between input and output (which is not allowed in sklearn). The *preserves_sample_size* indicates whether sample size (number of rows) may change during transformation.

The wrangler's employed computation engine is given via *computation_engine*.

See also:

https //scikit-learn.org/stable/developers/contributing.html

computation_engine

fit(**args*, ***kwargs*)

fit_transform(**args*, ***kwargs*)

get_params() → dict
Retrieve all wrangler parameters set within the __init__ method.

Returns **param_dict** – Parameter names as keys and corresponding values as values

Return type dictionary

preserves_sample_size

set_params(***params*)
Set wrangler parameters

Parameters **params** (*dict*) – Dictionary containing new values to be updated on wrangler. Keys have to match parameter names of wrangler.

Returns

Return type self

transform(**args*, ***kwargs*)

4.1.4 pywrangler.benchmark module

This module contains benchmarking utility.

class pywrangler.benchmark.**BaseProfiler**
Bases: object

Base class defining the interface for all profilers.

Subclasses have to implement *profile* (the actual profiling method) and *less_is_better* (defining the ranking of profiling measurements).

The private attribute *_measurements* is assumed to be set by *profile*.

measurements
The actual profiling measurements.

Type list

best
The best measurement.

Type float

median
The median of measurements.

Type float

worst
The worst measurement.

Type float

std
The standard deviation of measurements.

Type float

runs
The number of measurements.

Type int

profile()
Contains the actual profiling implementation.

report()
Print simple report consisting of best, median, worst, standard deviation and the number of measurements.

profile_report()
Calls profile and report in sequence.

best
Returns the best measurement.

less_is_better
Defines ranking of measurements.

measurements
Return measurements of profiling.

median
Returns the median of measurements.

profile(**args*, ***kwargs*)
Contains the actual profiling implementation and has to set *self._measurements*. Always returns self.

profile_report(**args*, ***kwargs*)
Calls profile and report in sequence.

report()
Print simple report consisting of best, median, worst, standard deviation and the number of measurements.

runs
Return number of measurements.

std
Returns the standard deviation of measurements.

worst
Returns the worst measurement.

class pywrangler.benchmark.**MemoryProfiler**(*func: Callable*, *repetitions: int = 5*, *interval: float = 0.01*)
Bases: *pywrangler.benchmark.BaseProfiler*

Approximate the increase in memory usage when calling a given function. Memory increase is defined as the difference between the maximum memory usage during function execution and the baseline memory usage before function execution.

In addition, compute the mean increase in baseline memory usage between repetitions which might indicate memory leakage.

Parameters

- **func** (*callable*) – Callable object to be memory profiled.
- **repetitions** (*int, optional*) – Number of repetitions.
- **interval** (*float, optional*) – Defines interval duration between consecutive memory usage measurements in seconds.

measurements
The actual profiling measurements in bytes.

Type list

best
The best measurement in bytes.

Type float

median
The median of measurements in bytes.

Type float

worst
The worst measurement in bytes.

Type float

std
The standard deviation of measurements in bytes.

Type float

runs
The number of measurements.

Type int

baseline_change
The median change in baseline memory usage across all runs in bytes.

Type float

profile()
Contains the actual profiling implementation.

report()
Print simple report consisting of best, median, worst, standard deviation and the number of measurements.

profile_report()
Calls profile and report in sequence.

Notes

The implementation is based on *memory_profiler* and is inspired by the IPython *%memit* magic which additionally calls *gc.collect()* before executing the function to get more stable results.

baseline_change
Returns the median change in baseline memory usage across all run. The baseline memory usage is defined as the memory usage before function execution.

baselines
Returns the absolute, baseline memory usages for each run in bytes. The baseline memory usage is defined as the memory usage before function execution.

less_is_better
Less memory consumption is better.

max_usages
Returns the absolute, maximum memory usages for each run in bytes.

profile(**args*, ***kwargs*)
Executes the actual memory profiling.

Parameters

- **args** (*iterable, optional*) – Optional positional arguments passed to *func*.
- **kwargs** (*mapping, optional*) – Optional keyword arguments passed to *func*.

class pywrangler.benchmark.**TimeProfiler**(*func: Callable*, *repetitions: Union[None*, *int] = None*)
Bases: *pywrangler.benchmark.BaseProfiler*

Approximate the time required to execute a function call.

By default, the number of repetitions is estimated if not set explicitly.

Parameters

- **func** (*callable*) – Callable object to be memory profiled.
- **repetitions** (*None, int, optional*) – Number of repetitions. If *None*, *timeit.Timer.autorange* will determine a sensible default.

measurements
The actual profiling measurements in seconds.

Type list

best
The best measurement in seconds.

Type float

median
The median of measurements in seconds.

Type float

worst
The worst measurement in seconds.

Type float

std
The standard deviation of measurements in seconds.

Type float

runs
The number of measurements.

Type int

profile()
Contains the actual profiling implementation.

report()
Print simple report consisting of best, median, worst, standard deviation and the number of measurements.

profile_report()
Calls profile and report in sequence.

Notes

The implementation is based on standard library's *timeit* module.

less_is_better
Less time required is better.

profile(**args*, ***kwargs*)
Executes the actual time profiling.

Parameters

- **args** (*iterable, optional*) – Optional positional arguments passed to *func*.
- **kwargs** (*mapping, optional*) – Optional keyword arguments passed to *func*.

pywrangler.benchmark.**allocate_memory**(*size: float*) → numpy.ndarray
Helper function to approximately allocate memory by creating numpy array with given size in MiB.

Numpy is used deliberately to define the used memory via dtype.

Parameters **size** (*float*) – Size in MiB to be occupied.

Returns **memory_holder**

Return type np.ndarray

4.1.5 pywrangler.exceptions module

The module contains package wide custom exceptions and warnings.

exception pywrangler.exceptions.**NotProfiledError**
Bases: ValueError, AttributeError

Exception class to raise if profiling results are acquired before calling *profile*.

This class inherits from both ValueError and AttributeError to help with exception handling

4.1.6 pywrangler.wranglers module

This module contains computation engine independent wrangler interfaces and corresponding descriptions.

class pywrangler.wranglers.**IntervalIdentifier**(*marker_column: str, marker_start: Any, marker_end: Any = <object object>, marker_start_use_first: bool = False, marker_end_use_first: bool = True, orderby_columns: Union[str, Iterable[str], None] = None, groupby_columns: Union[str, Iterable[str], None] = None, ascending: Union[bool, Iterable[bool]] = None, result_type: str = 'enumerated', target_column_name: str = 'iids'*)
Bases: *pywrangler.base.BaseWrangler*

Defines the reference interface for the interval identification wrangler.

An interval is defined as a range of values beginning with an opening marker and ending with a closing marker (e.g. the interval daylight may be defined as all events/values occurring between sunrise and sunset). Start and end marker may be identical.

The interval identification wrangler assigns ids to values such that values belonging to the same interval share the same interval id. For example, all values of the first daylight interval are assigned with id 1. All values of the second daylight interval will be assigned with id 2 and so on.

By default, values which do not belong to any valid interval, are assigned the value 0 by definition (please refer to *result_type* for different result types). If start and end marker are identical or the end marker is not provided, invalid values are only possible before the first start marker is encountered.

Due to messy data, start and end marker may occur multiple times in sequence until its counterpart is reached. Therefore, intervals may have different spans based on different task requirements. For example, the very first start or very last start marker may define the correct start of an interval. Accordingly, four intervals can be selected by setting *marker_start_use_first* and *marker_end_use_first*. The resulting intervals are as follows:

- first start / first end
- first start / last end (longest interval)
- last start / first end (shortest interval)
- last start / last end

Opening and closing markers are included in their corresponding interval.

Parameters

- **marker_column** (*str*) – Name of column which contains the opening and closing markers.
- **marker_start** (*Any*) – A value defining the start of an interval.
- **marker_end** (*Any, optional*) – A value defining the end of an interval. This value is optional. If not given, the end marker equals the start marker.
- **marker_start_use_first** (*bool*) – Identifies if the first occurring *marker_start* of an interval is used. Otherwise the last occurring *marker_start* is used. Default is False.
- **marker_end_use_first** (*bool*) – Identifies if the first occurring *marker_end* of an interval is used. Otherwise the last occurring *marker_end* is used. Default is True.
- **orderby_columns** (*str, Iterable[str], optional*) – Column names which define the order of the data (e.g. a timestamp column). Sort order can be defined with the parameter *ascending*.
- **groupby_columns** (*str, Iterable[str], optional*) – Column names which define how the data should be grouped/split into separate entities. For distributed computation engines, groupby columns should ideally reference partition keys to avoid data shuffling.
- **ascending** (*bool, Iterable[bool], optional*) – Sort ascending vs. descending. Specify list for multiple sort orders. If a list is specified, length of the list must equal length of *order_columns*. Default is True.
- **result_type** (*str, optional*) – Defines the content of the returned result. If 'raw', interval ids will be in arbitrary order with no distinction made between valid and invalid intervals. Intervals are distinguishable by interval id but the interval id may not provide any more information. If 'valid', the result is the same as 'raw' but all invalid intervals are set

to 0. If 'enumerated', the result is the same as 'valid' but interval ids increase in ascending order (as defined by order) in steps of one.

- **target_column_name** (*str, optional*) – Name of the resulting target column.

preserves_sample_size

4.1.7 Module contents

CHAPTER 5

License

CHAPTER 6

Contributors

- mansenfranzen <franz.woellert@gmail.com>
- TobiasRasbold

CHAPTER 7

Changelog

7.1 Version 0.1.0

This is the initial release of pywrangler.

- Enable raw, valid and enumerated return type for `IntervalIdentifier` (#23).
- Enable variable sequence lengths for `IntervalIdentifier` (#23).
- Add `DataTestCase` and `TestCollection` as standards for data centric test cases (#23).
- Add computation engine independent data abstraction `PlainFrame` (#23).
- Add `VectorizedCumSum` pyspark implementation for `IntervalIdentifier` wrangler (#7).
- Add benchmark utilities for pandas, spark and dask wranglers (#5).
- Add sequential `NaiveIterator` and vectorized `VectorizedCumSum` pandas implementations for `IntervalIdentifier` wrangler (#2).
- Add `PandasWrangler` (#2).
- Add `IntervalIdentifier` wrangler interface (#2).
- Add `BaseWrangler` class defining wrangler interface (#1).
- Enable `pandas` and `pyspark` testing on TravisCI (#1).

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index

K

L

M

N

O

P

R

S

T

V

W