pysemantic Documentation

Release 0.0.1

Jaidev Deshpande

June 21, 2016

Contents

Contents:

CHAPTER 1

Examples

- Introduction to PySemantic.

CHAPTER 2

Schema Configuration Reference

Every project in PySemantic can be configured via a data dictionary or a schema, which is a yaml file. This file houses the details of how PySemantic should treat a project's constituent datasets. A typical data dictionary follows the following pattern:

```
dataset_name:
  dataset_param_1: value1
  dataset_param_2: value2
  # etc
```

PySemantic reads this as a dictionary where the parameter names are keys and their values are the values in the dictionary. Thus, the schema for a whole project is a dictionary of dictionaries.

2.1 Basic Schema Configuration

Here is a list of different dataset parameters that PySemantic is sensitive to:

- `path` (Required, except when the `source` parameter is "mysql") The path to the file containing the data. Note that the path must either be absolute, or relative to the directory containing the schema. This can also be a list of files if the dataset spans multiple files. If that is the case, the path parameter can be specified as:

```
    path:
      - absolulte/path/to/file/1
      - absolulte/path/to/file/2
      # etc

    # or

    path:
      - foo/bar/baz
    # where foo is a directory in the directory that contains the schema.
```

- `demlimiter` (Optional, default: `,`) The delimiter used in the file. This has to be a character delimiter, not words like "comma" or "tab".
- `md5` (Optional) The MD5 checksum of the file to read. This necessary because sometimes we read files and after processing it, rewrite to the same path. This parameter helps keep track of whether the file is correct.
- `header`: (Optional) The header row of the file.
- `index_col`: (Optional) Name of the column that forms the index of the dataframe. This can be a single string or a list of strings. If a list is provided, the dataframe becomes multi-indexed.

- `sheetname`: (Optional) Name of the sheet containing the dataset in an MS Excel spreadsheet. This comes into play only when `path` points to an Excel file. For other types of files, this is ignored. When `path` is an Excel file and this parameter is not provided, it is assumed to be the same as the name of the dataset. For example:

```
iris:
    path: /path/to/iris.xlsx
```

The schema above assumes that the iris dataset resides in a sheet named "iris". If instead the name of the sheet is different, you can specify it as:

```
iris:
    path: /path/to/iris.xlsx
    sheetname: name_of_sheet
```

This parameter can also be a list, to enable the combination of multiple sheets into a dataframe, as follows:

```
iris:
    path: /path/to/iris.xlsx
    sheetname:
        - sheet1
        - sheet2
```

This will combine the data from sheet1 and sheet2 into a single dataframe.

- `column_names`: (Optional) Specify the names of columns to use in the loaded dataframe. This option can have multiple types of values. It can be:

    1. A list of strings to use as column names:

```
column_names:
  - column_1
  - column_2
  - column_3
```

    2. A dictionary that maps original column names to new ones:

```
column_names:
  org_colname_1: new_colname_a
  org_colname_2: new_colname_b
  org_colname_3: new_colname_c
```

    3. A Python function that translates the name of every column in the loaded dataframe:

```
column_names: !!python/name:module_name.translate_column_name
```

- `nrows`: (Optional) Method to select which rows are read from the dataset. This option, like `column_names`, can be specified in many ways. It can be:

    1. An integer (default): Number of rows to read from the file. If this option is not specified, all rows from the file are read.

```
nrows: 100
```

    2. A dictionary that recognizes specific keys:

        – `random`: A boolean that directs PySemantic to shuffle the selected rows after loading the dataset. For example, including the following lines in the schema

```
nrows:
   random: true
```

will shuffle the dataset before returning it.

- `range`: A list of two integers, which denote the first and the last index of the range of rows to be read. For example, the following lines

```
nrows:
    range:
        - 10
        - 50
```

will only select the 10th to the 50th (exclusive) rows.

- `count`: An integer that can be used in conjunction with either or both of the above options, to denote the number of rows to read from a random selection or a range.

```
nrows:
    range:
        - 10
        - 50
    count: 10
    random: true
```

The lines shown above will direct PySemantic to load 10 rows at random between the 10th and the 50th rows of a dataset.

- `shuffle`: A boolean to be used with `count` to shuffle the top `count` rows before returning the dataframe.

```
nrows:
    count: 10
    shuffle: True
```

The above schema will read the first ten rows from the dataset and shuffle them.

3. A callable which returns a logical array which has the same number of elements as the number of rows in the dataset. The output of this callable is used as a logical index for slicing the dataset. For example, suppose we wanted to extract all even numbered rows from a dataset, then we could make a callable as follows:

```
iseven = lambda x: np.remainder(x, 2) == 0
```

Suppose this function resides in a module called `foo.bar`, then we can include it in the schema as follows:

```
nrows: !!python/name:foo.bar.iseven
```

This will cause PySemantic to only load all even valued row numbers.

- `use_columns`: (Optional) The list of the columns to read from the dataset. The format for specifying this parameter is as follows:

```
use_columns:
  - column_1
  - column_2
  - column_3
```

If this parameter is not specified, all columns present in the dataset are read.

- `exclude_columns`: This option can be used to specify columns that are explicityly to be ignored. This is useful when there are large number of columns in the dataset and we only wish to exclude a few. Note that this option overrides the `use_columns` option, i.e. if a column name is present in both lists, it will be dropped.

- `na_values`: A string or a list of values that are considered as NAs by the pandas parsers, applicable to the whole dataframe.
- `converters`: A dictionary of functions to be applied to columns when loading data. Any Python callable can be added to this list. This parameter makes up the `converters` argument of Pandas parsers. The usage is as follows:

```
converters:
  col_a: !!python/name:numpy.int
```

This results in the `numpy.int` function being called on the column `col_a`

- `dtypes` (Optional) Data types of the columns to be read. Since types in Python are native objects, PySemantic expects them to be so in the schema. This can be formatted as follows:

```
dtypes:
  column_name: !!python/name:python_object
```

For example, if you have three columns named `foo`, `bar`, and `baz`, which have the types `string`, `integer` and `float` respectively, then your schema should look like:

```
dtypes:
  foo: !!python/name:__builtin__.str
  bar: !!python/name:__builtin__.int
  baz: !!python/name:__builtin__.float
```

Non-builtin types can be specified too:

```
dtypes:
  datetime_column: !!python/name:datetime.date
```

Note: You can figure out the yaml representation of a Python type by doing the following:

```
import yaml
x = type(foo) # where foo is the object who's type is to be yamlized
print yaml.dump(x)
```

- `parse_dates` (Optional) Columns containing Date/Time values can be parsed into native NumPy datetime objects. This argument can be a list, or a ditionary. If it is a dictionary of the following form:

```
parse_dates:
  output_col_name:
    - col_a
    - col_b
```

it will parse columns `col_a` and `col_b` as datetime columns, and put the result in a column named `output_col_name`. Specifying the output name is optional. You may declare the schema as a list, as follows:

```
parse_dates:
  - col_a
  - col_b
```

In this case the parser will independently parse columns `col_a` and `col_b` into datetime.

NOTE: Specifying this column will make PySemantic ignore any columns that have been declared as having the datetime type in the `dtypes` parameter.

- `pickle` (Optional) Absolute path to file which contains pickled arguments for the parser. This option can be used if readability or declaratives are not a concern. The file should contain a picked dictionary that is directly passed to the parser, i.e. if the loaded pickled data is in a dict named `data`, then parser invocation becomes `parser(**data)`.

NOTE: If any of the above options are present, they will override the corresponding arguments contained in the pickle file. In PySemantic, declarative statements have the right of way.

2.2 Column Schema Configuration

PySemantic also allows specifying rules and validators independently for each column. This can be done using the `column_rules` parameter of the dataset schema. Here is a typical format:

```
dataset_name:
  column_rules:
    column_1_name:
      # rules to be applied to the column
    column_2_name:
      # rules to be applied to the column
```

The following parameters can be supplied to any column under `column_rules`:

- `is_drop_na` ([true|false], default false) Setting this to `true` causes PySemantic to drop all NA values in the column.
- `is_drop_duplicates` ([true|false], default false) Setting this to `true` causes PySemantic to drop all duplicated values in the column.
- `unique_values`: These are the unique values that are expected in a column. The value of this parameter has to be a yaml list. Any value not found in this list will be dropped when cleaning the dataset.
- `exclude`: These are the values that are to be explicitly excluded from the column. This comes in handy when a column has too many unique values, and a handful of them have to be dropped. Note that this value has to be a list.
- `minimum`: Minimum value allowed in a column if the column holds numerical data. By default, the minimum is -np.inf. Any value less than this one is dropped.
- `maximum`: Maximum value allowed in a column if the column holds numerical data. By default, the maximum is np.inf. Any value greater than this one is dropped.
- `regex`: A regular expression that each element of the column must match, if the column holds text data. Any element of the column not matching this regex is dropped.
- `na_values`: A list of values that are considered as NAs by the pandas parsers, applicable to this column.
- `postprocessors`: A list of callables that called one by one on the columns. Any python function that accepts a series, and returns a series can be a postprocessor.

Here is a more extensive example of the usage of this schema.

```
iris:
  path: /home/username/src/pysemantic/testdata/iris.csv
  converters:
    Sepal Width: !!python/name:numpy.floor
  column_rules:
    Sepal Length:
      minimum: 2.0
    Petal Length:
      maximum: 4.0
    Petal Width:
      exclude:
        - 3.14
    Species:
      unique_values:
```

```
        - setosa
        - versicolor
    postprocessors:
        - !!python/name:module_name.foo
```

This would cause PySemantic to produce a dataframe corresponding to the Fisher iris dataset which has the following characteristics:

1. It contains no observations where the sepal length is less than 2 cm.
2. It contains no observations where the petal length is more than 4 cm.
3. The sepal width only contains integers.
4. The petal width column will not contain the specific value 3.14
5. The species column will only contain the values "setosa" and "versicolor", i.e. it will not contain the value "virginica".
6. The species column in the dataframe will be processed by the `module_name.foo` function.

2.3 DataFrame Schema Configuration

A few rules can also be enforced at the dataframe level, instead of at the level of individual columns in the dataset. Two of them are:

- `drop_duplicates` ([true|false, default true]). This behaves in the same way as `is_drop_duplicates` for series schema, with the exception that here the default is True.
- `drop_na` ([true|false, default true]). This behaves in the same way as `is_drop_na` for series schema, with the exception that here the default is True.

2.4 Reading a MySQL Table

Note: This has not yet been tested.

PySemantic can automatically create the function calls required to download a MySQL table as a dataframe - by using a wrapper around the `pandas.read_sql_table` function. The configuration parameters are as follows:

- `source`: This is simply a string saying "mysql", which lets pysemantic know that the dataset is to be downloaded from a MySQL database.
- `config`: This is a dictionary that contains the configuration required to connect to the MySQL server. The configuration must have the following elements:
    1. `hostname`: The IP address or the hostname of the machine hosting the MySQL server.
    2. `db_name`: Name of the database from which to read the table.
    3. `table_name`: Name of the table to be read.
    4. `username`: The MySQL username
    5. `password`: The MySQL password
- `chunksize`: (Integer, optional) If this is specified, Pandas returns an iterator in which every iteration contains `chunksize` rows.

CHAPTER 3

API Reference

3.1 pysemantic package

3.1.1 Submodules

3.1.2 pysemantic.cli module

3.1.3 pysemantic.custom_traits module

3.1.4 pysemantic.errors module

3.1.5 pysemantic.exporters module

3.1.6 pysemantic.loggers module

3.1.7 pysemantic.project module

3.1.8 pysemantic.utils module

3.1.9 pysemantic.validator module

3.1.10 Module contents

3.2 pysemantic

CHAPTER 4

Indices and tables

- genindex
- modindex
- search