
Trump Documentation

Release 0.0.5

Jeffrey McLarty

August 15, 2015

1	Introduction	1
2	Getting Started	3
2.1	Installation	3
2.2	Uninstall	6
2.3	Basic Usage	6
3	Object Model	13
3.1	Object-Relational Model	13
3.2	Data Flow	30
3.3	Aggregation Methods	32
4	Templating	35
4.1	Template Base Classes	35
4.2	Template Classes	36
5	Source Extensions	39
5.1	Creating & Modifying Source Extensions	39
5.2	Pre-Installed Source Extensions	41
6	User Interface	45
6.1	UI Prototype	45
	Python Module Index	53

Introduction

Trump is a framework for objectifying data, with the goal of centralizing the management of data feeds to enable quicker deployment of analytics, applications, and reporting. Munging data, common calculations, validation of data, can all be handled by Trump, upstream of any application or user requirement.

Inside the Trump framework, a symbol refers to one or more data feeds, each with their own instructions saved for retrieving data from a specific source. Once it's retrieved by Trump, depending on the attributes of the symbol, it gets munged, aggregated, checked, and cached. Downstream users are free to query the existing cache, force a re-cache, or check any property of the data prior to using it.

System Admins can systematically detect problems in advance, via common integrity checks of the data, then optionally schedule the re-cache by tag or symbol name. Users and admins have the ability to manually override problems if they exist, with a specific feed, in a way that is centralized, auditable, and backed-up efficiently.

With a focus on business processes, Trump's long run goals enable data feeds to be:

- **Prioritized**, *flexibly* - a symbol can be associated with multiple data source for a variety of reasons including redundancy, calculations, or optionality.
- **Modified**, *reliably* - a symbol's data feeds can be changed out, without any changes requiring testing to the downstream application or user.
- **Verified**, *systematically* - a variety of common data processing checks are performed as the symbol's data is cached.
- **Audited**, *quickly* - alerts and reports all become possible to assess integrity or inspect where manual over-rides have been performed.
- **Aggregated**, *intelligently* - on a symbol by symbol basis, feeds can be combined and used in an extensible number of ways.
- **Customized**, *dynamically* - extensibility is possible at the templating, munging, aggregation, and validity steps.

Getting Started

2.1 Installation

2.1.1 Step 1. Install Package

SUMMARY OF STEP 1: Clone and install trump, from github.

```
git clone https://github.com/Equitable/trump.git + cd trump + python setup.py
install
```

Note: If you use any other installation method (Eg. `python setup.py develop`), you will need to manually create your own `.cfg` files, in step 2, by renaming the `.cfg_sample` files to `cfg` files.

Note: Trump is setup to work with `pip install trump`, however the codebase and features are being worked on very quickly right now (2015Q2). The version on pypi, will be very stale, very quickly. It's best to install from the latest commit to the master branch direct from GitHub.

2.1.2 Step 2. Configure Settings

SUMMARY OF STEP 2: Put a SQLAlchemy Engine String in trump/config/trump.cfg. Comment out all other engines.

Trump needs information about a database it can use, plus there are a couple other settings you may want to tweak. You can either follow the instructions below, or pass a SQLAlchemy engine/engine-string, to both `SetupTrump()` and `SymbolManager()` everytime you use them.

The configuration file for trump is in:

```
userbase/PythonXY/site-packages/trump/config/trump.cfg
```

or

```
yourprojfolder/trump/config/trump.cfg
```

Note: A sample config file is included, by the name `trump.cfg_sample`. Depending on your installation method, you may need to copy and rename it to `trump.cfg`. `cfg` files aren't tracked by git, nor does the installation do anything other than copy and rename the file extension. `pip` and `python setup.py install` will rename them for you. `python setup.py develop` won't rename them for you, you'll have to do it yourself.

Assuming you want to use a file based sqlite database (easiest, for beginners), change:

```
engine:  sqlite:// to ;engine:  sqlite:// (notice the semi-colon, this just comments out the line)
```

and change this line:

```
;engine:  sqlite:///home/jnmclarty/Desktop/trump.db to
```

```
engine:  sqlite:///home/path/to/some/file/mytrumpfile.db (on linux) or
```

```
engine:  sqlite:///C:\path\to\some\mytrumpfile.db (on windows)
```

The folder needs to exist in advance, the file should not exist. Trump creates the file.

2.1.3 Step 3. Adjust Existing Template Settings (Optional)

SUMMARY OF STEP 3: Adjust any settings for templates you intend you use.

Trump has template settings, stored in multiple settings files, using an identical method as the config file in Step 2. `pip` or `python setup.py install` would have created some from samples. Using any other installation method, you would have to rename `cfg_sample` to `cfg` yourself.

The files are here:

```
userbase/PythonXY/site-packages/trump/templates/settings/
```

or

```
yourprojfolder/trump/templates/settings/
```

Edit `trump/templating/settings` `cfg` files, depending on the intended data sources to be used.

See the documentation section “Configuring Data Sources” for guidance.

2.1.4 Step 4. Run SetupTrump()

SUMMARY OF STEP 4: Run `trump.SetupTrump()`, to setup the tables required for Trump to work.

Running the code block below, will create all the tables required in the database provided in Step 2.

```
from trump import SetupTrump
SetupTrump()
# Or, if you skipped step 2 correctly, you could do:
SetupTrump(r'sqlite:///home/path/to/some/file/mytrumpfile.db')
```

If it all worked, you will see “Trump is installed @...”. You’re Done! Hard part is over.

You’re now ready to create a `SymbolManager`, which will help you create your first symbol.

```
from trump import SymbolManager
sm = SymbolManager()
# Or, if you skipped step 2 correctly, you could do:
sm = SymbolManager(r'sqlite:///home/path/to/some/file/mytrumpfile.db')
...
mysymbol = sm.create('MyFirstSymbol') # should run without error.
```

2.1.5 Configuring Data Sources

Data feed source template classes map to their respective `.cfg` file in the `templating/settings` directory, as discussed in Step 3.

The goal of the files is to add a small layer of security. The goal of the template classes is to reduce code during symbol creation scripts. There is nothing preventing a password from being hardcoded into a template, the same way a tablename can be added to a .cfg file. It's only a maintenance decision for the admin.

The sections of the cfg files get used by the template's in their respective classes. The section of the config files' names are then either referenced at the symbol creation point, storing .cfg file info with the symbol in the database, or leaving Trump to query the attributes at every cache, from the source .cfg file.

Trump will use parameters for a source in the following order:

1. Specified explicitly when a template is used. (Eg. table name)

```
#Assuming the template doesn't clobber the value.
myfeed = QuandlFT(authtoken='XXXXXXXX')
```

2. Specified implicitly using default value or logic derived in the template. (Eg. Database Names)

```
class QuandlFT(object):
    def __init__(authtoken = 'XXXXXXXXXX'):
        if len(authtoken) == 8:
            self.authtoken = authtoken
        else:
            self.authtoken = 'YYYYYYYYYY'
```

3. Specified implicitly using read_settings(). (Eg. database host, port)

```
class QuandlFT(object):
    def __init__(**kwargs):
        autht = read_settings('Quandl', 'userone', 'authtoken')
        self.authtoken = autht
```

4. Specified via cfg section. (Eg. authentication keys and passwords)

```
class QuandlFT(object):
    def __init__(**kwargs):
        self.meta['stype'] = 'Quandl' #cfg file name
        self.meta['sourcing_key'] = 'userone' #cfg file section
```

contents of templating/settings/Quandl.cfg:

```
[userone]
authtoken = XXXXXXXXXX
```

If the template points to a section of a config file, rather than reading in a value from a config file, (ie, #4), the info will not be stored in the database. Instead, the information will be looked up during caching from the appropriate section in the cfg file.

This means that the cfg file values can be changed post symbol creation, outside of Trump.

2.1.6 Testing the Installation

After Trump has been configured, and pointed at a database via an engine string using a config file, one can run the py.test enabled test suite. The tests require network access, but will skip certain tests without it. The testing suite makes a mess, and doesn't clean up after itself. So, be prepared to run it on a database which can be delete immediately after.

Insight into compatibility with databases other SQLite and PostGRES, are of interest to the maintainers. So, if you run the test suite on some other database, and it all works, do let us know via a GitHub issue or e-mail. If it doesn't, please let us know that as well!

2.2 Uninstall

#. Delete all tables Trump created. (There is a script, which attempts to do that for you. See `uninstall.py`. This will (attempt to) remove all tables created by Trump. The file will likely require minor changes if you use anything other than PostgreSQL, or if it hasn't been updated to reflect newer tables in Trump.) #. Delete `site-packages/trump` and all its subdirectories.

2.3 Basic Usage

These examples dramatically understate the utility of Trump's long term feature set.

2.3.1 Tesla Closing Price from Multiple Sources

Adding the Symbol

```
from trump.orm import SymbolManager
from trump.templating import QuandlFT, GoogleFinanceFT, YahooFinanceFT,
                             DateExistsVT, FeedMatchVT

sm = SymbolManager()

TSLA = sm.create(name = "TSLA",
                 description = "Tesla Closing Price USD",
                 units = '$ / share')

TSLA.add_tags(["stocks", "US"])

#Try Google First
#If Google's feed has a problem, try Quandl's backup
#If all else fails, use Yahoo's data...

# 'Close' is stored in the GoogleFinanceFT Template
TSLA.add_feed(GoogleFinanceFT("TSLA"))

TSLA.add_feed(QuandlFT("GOOG/NASDAQ_TSLA", fieldname='Close'))

# 'Close' is stored in the YahooFinanceFT Template
TSLA.add_feed(YahooFinanceFT("TSLA"))

#All three are downloaded, with every cache instruction
TSLA.cache()

# In the end, the result is one clean pandas Series representing
# TSLA's closing price, with source, munging, and validity parameters
# all stored persistently for future
# re-caching.

print TSLA.df.tail()

           TSLA
dateindex
2015-03-20  198.08
```

```

2015-03-23  199.63
2015-03-24  201.72
2015-03-25  194.30
2015-03-26  190.40

```

```
sm.finish()
```

Using the Symbol

```

from trump.orm import SymbolManager

sm = SymbolManager()

TSLA = sm.get("TSLA")

#optional
TSLA.cache()

print TSLA.df.tail()

           TSLA
dateindex
2015-03-20  198.08
2015-03-23  199.63
2015-03-24  201.72
2015-03-25  194.30
2015-03-26  190.40

sm.finish()

```

2.3.2 Data From CSV, with a frequency-specified index

Adding the Symbol

```

from trump.orm import SymbolManager

#Import the CSV Feed Template
from trump.templating import CSVFT

#Import the Forward-Fill Index Template
from trump.templating import FFillIT

sm = SymbolManager()

sym = sm.create(name = "DailyDataTurnedWeekly")

f1 = CSVFT('somedata.csv', 'ColumnName', parse_dates=0, index_col=0)

sym.add_feed(f1)

weeklyind = FFillIT('W')
sym.set_indexing(weekly)

sym.cache()

```

```
sm.finish()
```

Using the Symbol

```
from trump.orm import SymbolManager

sm = SymbolManager()

sym = sm.get("DailyDataTurnedWeekly")

#optional
oil.cache()

print sym.df.index
# <class 'pandas.tseries.index.DatetimeIndex'>
# [2010-01-03, ..., 2010-01-17]
# Length: 3, Freq: W-SUN, Timezone: None

sm.finish()
```

2.3.3 Tesla Closing Price from Two Sources, With Validity Checks

Adding the Symbol

```
from trump.orm import SymbolManager
from trump.templating import QuandlFT, GoogleFinanceFT,
                             DateExistsVT, FeedsMatchVT

sm = SymbolManager()

TSLA = sm.create(name = "TSLA",
                 description = "Tesla Closing Price USD",
                 units = '$ / share')

TSLA.add_feed(GoogleFinanceFT("TSLA"))
TSLA.add_feed(QuandlFT("GOOG/NASDAQ_TSLA", fieldname='Close'))

# Tell trump, to check the first and second feed,
# because they should be equal.

validity_settings = FeedsMatchVT(1, 2)
TSLA.add_validity(validity_settings)

# Tell trump, to make sure we have a data point for the current day
# any time we check validity.

validity_settings = DateExistsVT('today')
TSLA.add_validity(validity_settings)

# By default, the cache process checks the validity settings
# or will raise/log/warn/print/etc. based on the appropriate
# handler for validity.

# Since we're going to check validity, with a bit more
```

```
# granularity upstream/later, we can skip it during the cache process
# by setting it to False.

TSLA.cache(checkvalidty=False)

sm.finish()
```

Using the Symbol

```
from trump.orm import SymbolManager

sm = SymbolManager()

TSLA = sm.get("TSLA")

#optional
TSLA.cache()

#There are a few options, to check the data...

#Individual validity checks can be ran, with the
# settings stored persistently in the object

# Eg 1
if TSLA.check_validity('FeedsMatch'):
    #do stuff with clean data

# Eg 2
if TSLA.check_validity('DateExists'):
    #do stuff with today's data point

# Or, all the validity checks with their
# respective settings can be ran with one simple
# property:

if TSLA.isvalid:
    #do stuff with knowing both feeds match, and
    # a datapoint for today exists.
```

2.3.4 Oil from Quandl & SQL Example

Adding the Symbol

```
from trump.orm import SymbolManager
from trump.templating import QuandlFT, SQLFT

sm = SymbolManager()

oil = sm.create(name = "oil_front_month",
               description = "Crude Oil",
               units = '$ / barrel')

oil.add_tags(['commodity', 'oil', 'futures'])

f1 = QuandlFT(r"CHRIS/CME_CL2",fieldname='Settle')
```

```
f2 = SQLFT("SELECT date,data FROM test_oil_data;")

oil.add_feed(f1)
oil.add_feed(f2)

oil.cache()

print oil.df.tail()

sm.finish()
```

Using the Symbol

```
from trump.orm import SymbolManager

sm = SymbolManager()

oil = sm.get("oil_front_month")

#optional
oil.cache()

print oil.df.tail()

sm.finish()
```

2.3.5 Google Stock Price Daily Percent Change Munging

Adding the Symbol

```
from trump.orm import SymbolManager
from trump.templating import YahooFinaceFT

sm = SymbolManager()

GOOGpct = sm.create(name = "GOOGpct",
                    description = "Google Percent Change")

fdtemp = YahooFinanceFT("GOOG")

mgtemp = PctChangeMT()

GOOGpct.add_feed(fdtemp, munging=mgtemp)
```

Using the Symbol

```
from trump.orm import SymbolManager

sm = SymbolManager()

GOOG = sm.get("GOOGpct")

#optional
```

```
GOOG.cache()  
  
print GOOG.df.tail()  
  
#           GOOGpct  
# 2015-05-04  0.005354  
# 2015-05-05 -0.018455  
# 2015-05-06 -0.012396  
# 2015-05-07  0.012361  
# 2015-05-08  0.014170
```

Object Model

3.1 Object-Relational Model

Trump's persistent object model, made possible by its object-relational model (ORM), all starts with a *Symbol*, and an associated list of *Feed* objects.

An fragmented illustration of the ORM is presented in the three figures below.

Supporting objects store details persistently about error handling, sourcing, munging, and validation, so that a *Symbol* can *cache()* the data provided from the various *Feed* objects, in a single datatable or serve up a fresh *pandas.Series* at anytime. A symbol's its *Index*, can further enhance the intelligence that Trump can serve via *pandas*.

Note: Trump's template system consists of objects, which are external to the ORM. Templates are used to expedite construction of ORM objects. Nothing about any template, persists in the database. Only instantiated ORM objects would do so. Templates, should be thought of as boilerplate, or macros, to reduce Feed creation time.

3.1.1 Symbol Manager

class *SymbolManager* (*engine_or_eng_str=None, loud=False, echo=False*)

Bases: *object*

The *SymbolManager* maintains the SQLAlchemy database session, and provides access to object creation, deletion, searching, and overrides/failsafes.

Parameters

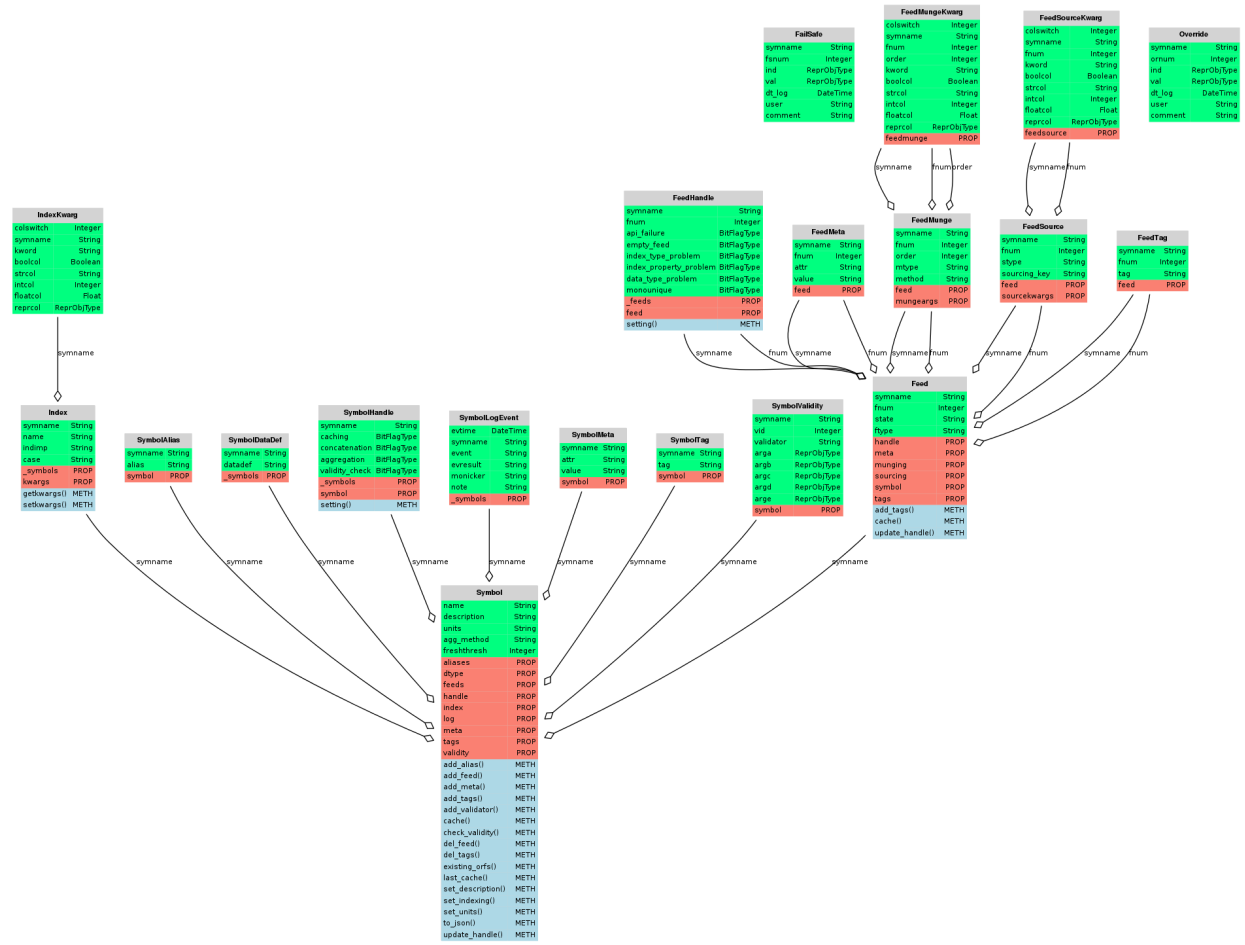
- **engine_or_eng_str** (*str or None, optional*) – Pass a SQLAlchemy engine, or a string. Without one, it will use the string provided in *trump/options/trump.cfg*. If it fails to get a value there, an in-memory SQLite session would be created.
- **loud** (*bool, optional*) – Print information such as engine string used, defaults to *False*
- **echo** (*bool, optional*) – If a new engine is created, it will pass this to its constructor, enabling SQLAlchemy's echo mode.

Returns

Return type *SymbolManager*

add_fail_safe (*symbol, ind, val, dt_log=None, user=None, comment=None*)

Appends a single indexed-value pair, to a symbol object, to be used during the final steps of the aggregation of the datatable.



Trump's ORM

Fig. 3.1: The full ORM, excludes the symbol's datatable.

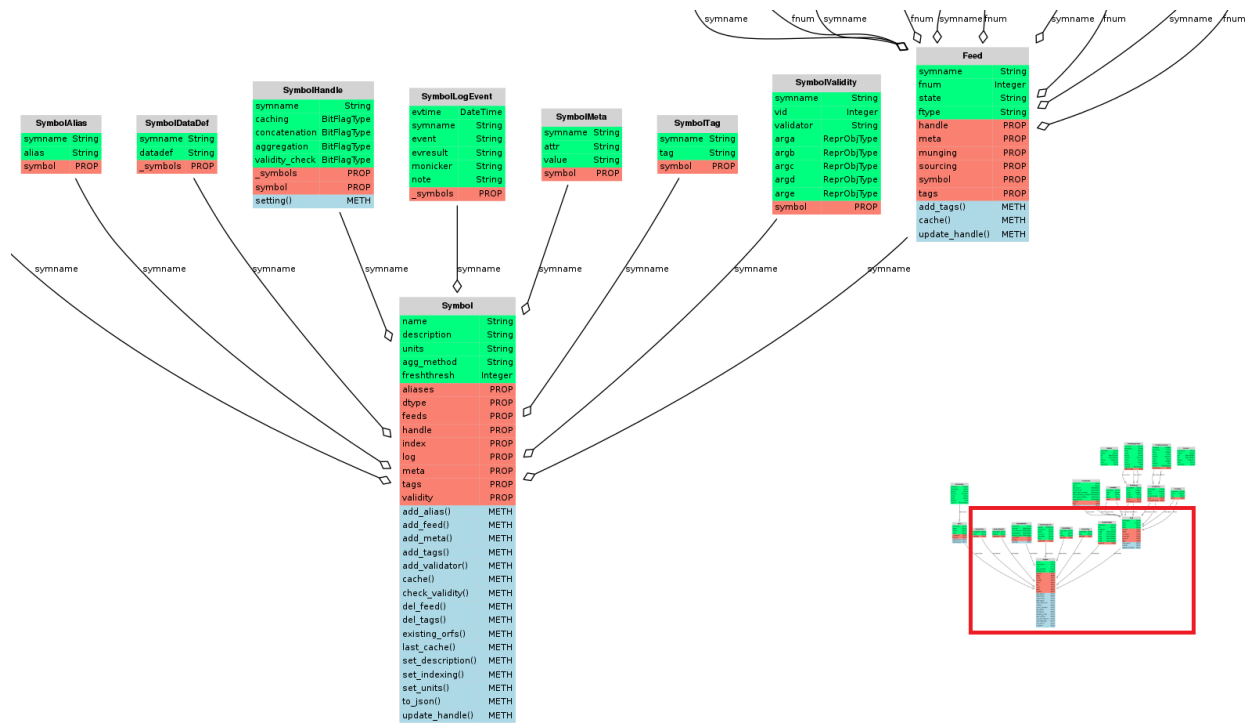


Fig. 3.2: The Symbol portion of the ORM, excludes the symbol's datatable.

With default settings FailSafes, get applied with lowest priority.

Parameters

- **symbol** (*Symbol or str*) – The Symbol to apply the fail safe
- **ind** (*obj*) – The index value where the fail safe should be applied
- **val** (*obj*) – The data value which will be used in the fail safe
- **dt_log** (*datetime*) – A log entry, for saving when this fail safe was created.
- **user** (*str*) – A string representing which user made the fail safe
- **comment** (*str*) – A string to store any notes related to this fail safe.

add_override (*symbol, ind, val, dt_log=None, user=None, comment=None*)

Appends a single indexed-value pair, to a symbol object, to be used during the final steps of the aggregation of the datatable.

With default settings Overrides, get applied with highest priority.

Parameters

- **symbol** (*Symbol or str*) – The Symbol to override
- **ind** (*obj*) – The index value where the override should be applied
- **val** (*obj*) – The data value which will be used in the override
- **dt_log** (*datetime*) – A log entry, for saving when this override was created.
- **user** (*str*) – A string representing which user made the override
- **comment** (*str*) – A string to store any notes related to this override.

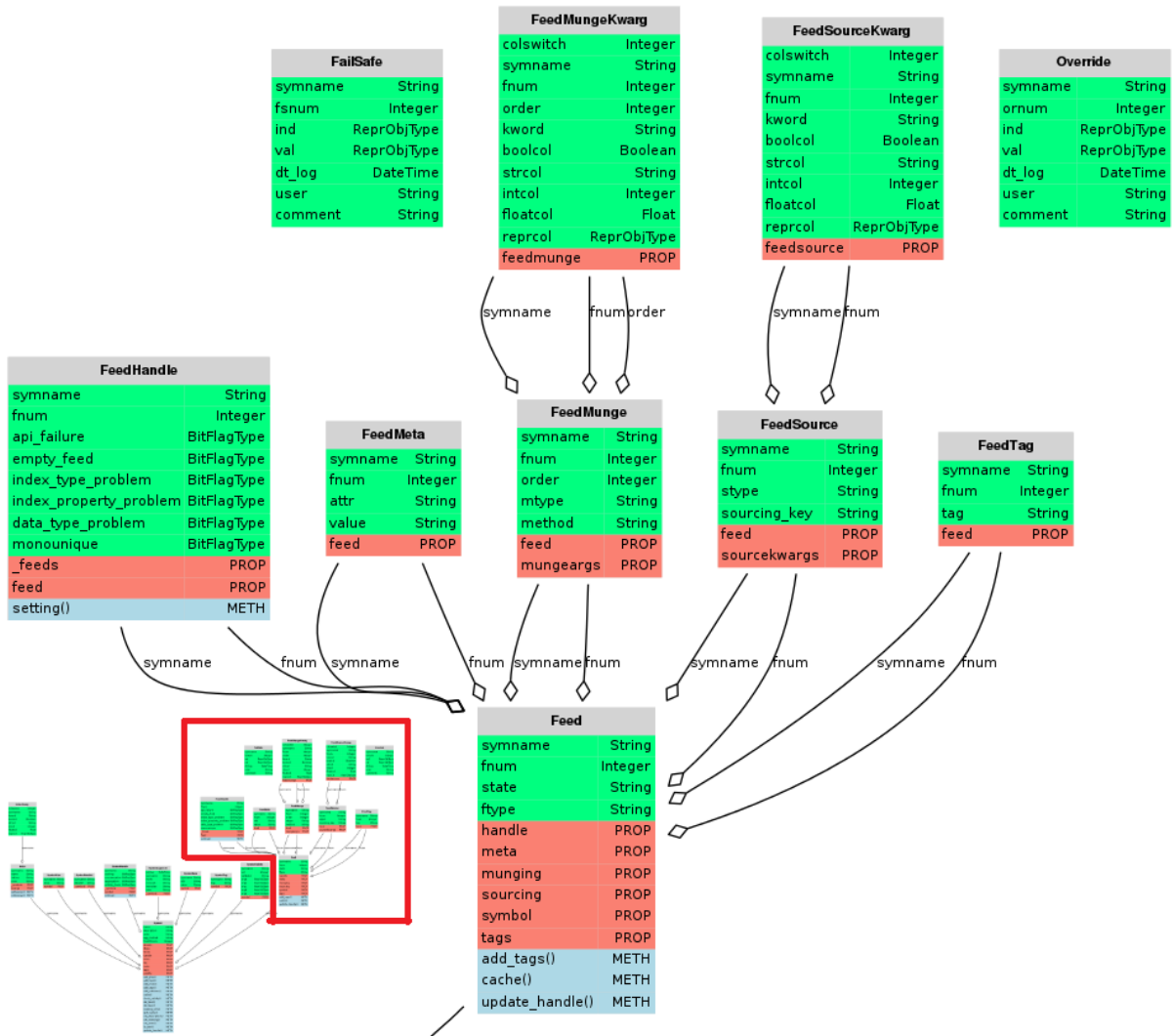


Fig. 3.3: The Feed, FailSafe & Override portion of the ORM

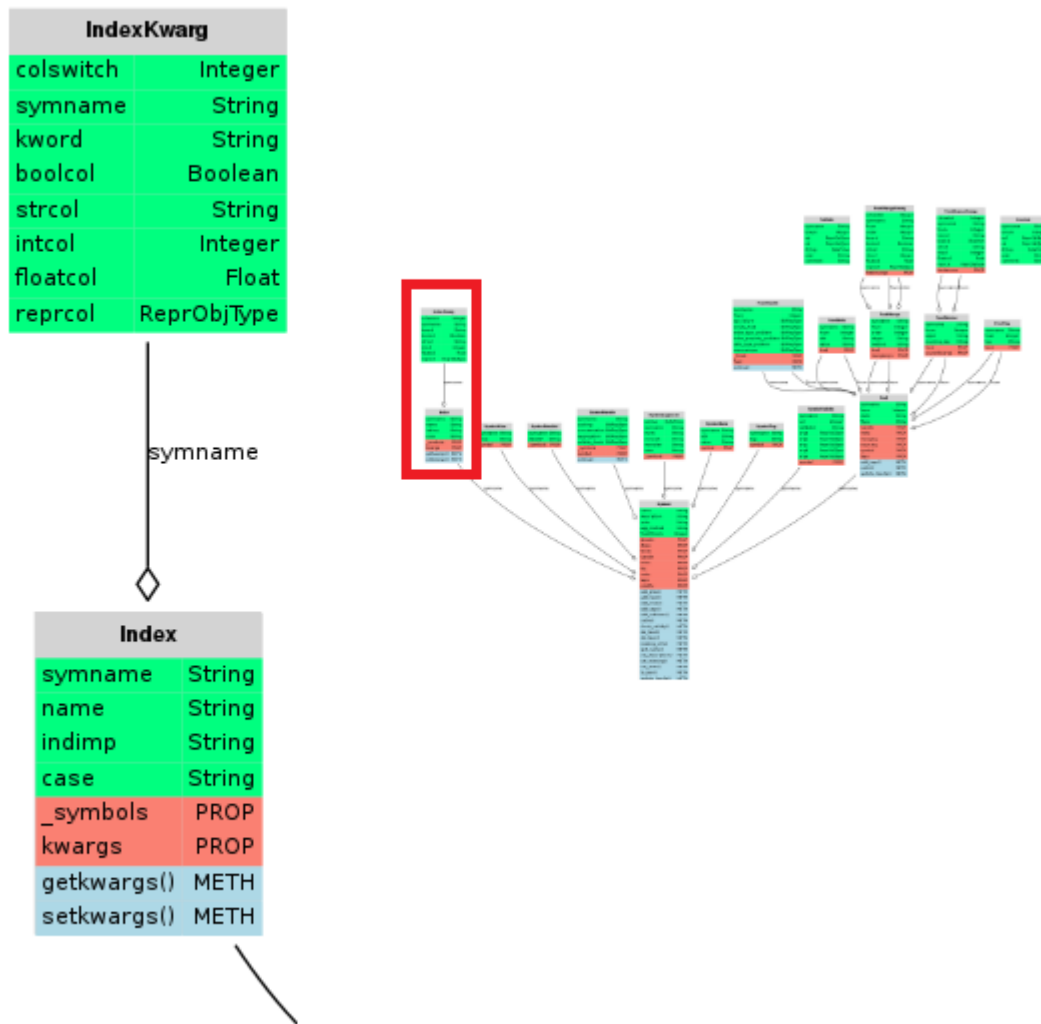


Fig. 3.4: The Index portion of the ORM.

build_view_from_tag (*tag*)

Build a view of group of Symbols based on their tag.

Parameters *tag* (*str*) – Use ‘%’ to enable SQL’s “LIKE” functionality.

Note: This function is written without SQLAlchemy, so it only tested on Postgres.

complete ()

Commits any changes to the database. In general, most of Trump API’s auto-commits or does so internally.

This is necessary when working directly with SQLAlchemy exposed attributes.

create (*name*, *description=None*, *units=None*, *agg_method='priority_fill'*, *overwrite=False*)

Create, or get if exists, a Symbol.

Parameters

- **name** (*str*) – A symbol’s name is a primary key, used across the Trump ORM.
- **description** (*str*, *optional*) – An arbitrary string, used to store user information related to the symbol.
- **units** (*str*, *optional*) – This is a string used to denote the units of the final data Series.
- **agg_method** (*str*, *optional*) – The aggregation method, used to calculate the final feed. Defaults to *priority_fill*.
- **overwrite** (*bool*, *optional*) – Set to True, to force deletion an existing symbol. defaults to False.

Returns

Return type *Symbol*

delete (*symbol*)

Deletes a Symbol.

Parameters *symbol* (*str* or *Symbol*) –

finish ()

Closes the session with the database.

Call at the end of a trump session. It also calls `SessionManager.complete()`.

get (*symbol*)

Gets a Symbol based on name, which is expected to exist.

Parameters *symbol* (*str* or *Symbol*) –

Returns

Return type *Symbol*

Raises `Exception` – If it does not exist. Use `.try_to_get()`, if the symbol may or may not exist.

search (*usrqry=None*, *name=False*, *desc=False*, *tags=False*, *meta=False*, *stronly=False*, *dolike-logic=True*)

Get a list of Symbols by searching a combination of a Symbol’s name, description, tags or meta values.

Parameters

- **usrqry** (*str*) – The string used to query. Appending ‘%’ will use SQL’s “LIKE” functionality.
- **name** (*bool*, *optional*, *default False*) – Search by symbol name.

- **desc** (*bool, optional, default False*) – Search by symbol descriptions.
- **tags** (*bool, optional, default False*) – Search by symbol tags.
- **meta** (*bool, optional, default False*) – Search within a symbol’s meta attribute’s value.
- **stronly** (*bool, optional, default True*) – Return only a list of symbol names, as opposed to the (entire) Symbol objects.
- **dolike****logic** – Append ‘%’ to either side of the string, if the string doesn’t already have % specified.

Returns

Return type List of Symbols or empty list

search_meta (*attr, value=None, stronly=False*)

Get a list of Symbols by searching a specific meta attribute, and optionally the value.

Parameters

- **attr** (*str*) – The meta attribute to query.
- **value** (*None, str or list*) – The meta attribute to query. If you pass a float, or an int, it’ll be converted to a string, prior to searching.
- **stronly** (*bool, optional, default True*) – Return only a list of symbol names, as opposed to the (entire) Symbol objects.

Returns

Return type List of Symbols or empty list

search_meta_specific (***avargs*)

Search list of Symbol objects by by querying specific meta attributes and their respective values.

Parameters avargs – The attributes and values passed as key word arguments. If more than one criteria is specified, AND logic is applied. Appending ‘%’ to values will use SQL’s “LIKE” functionality.

Example

```
>>> sm.search_meta(geography='Canada', sector='Gov%')
```

Returns

Return type List of Symbols or empty list

search_tag (*tag, symbols=True, feeds=False*)

Get a list of Symbols by searching a tag or partial tag.

Parameters

- **tag** (*str*) – The tag to search. Appending ‘%’ will use SQL’s “LIKE” functionality.
- **symbols** (*bool, optional*) – Search for Symbol’s based on their tags.
- **feeds** (*bool, optional*) – Search for Symbol’s based on their Feeds’ tags.

Returns

Return type List of Symbols or empty list

3.1.2 Conversion Manager

class **ConversionManager** (*engine_or_eng_str=None, system='FX', tag=None*)

Bases: *trump.orm.SymbolManager*

A ConversionManager handles the conversion of previously instantiated symbols, based on the object's units and the conversion manager setup. The conversion is performed adhoc, in python only usage. That is, nothing about the conversion persists in the Trump framework. Only the final series is converted.

Parameters

- **engine_or_eng_str** (*str or None*) – Pass a SQLAlchemy engine, or a string. Without one, it will use the default provided in trump/options/trump.cfg If it fails to get a value there, an in-memory SQLite session would be created.
- **system** (*str, optional*) – Uses the FX conversion system logic by default. Currently, no other systems are implemented. Eg. metric-only, imperial-metric, etc.

Other systems can be added after instantiation of the ConversionManager, but the one specified at instantiation will be used as default.

- **tag** (*str, optional*) – Tag for the set of feeds to use for conversion. Only necessary, if the conversion system relies on it. For FX, it's needed, to specify the set of feeds to use.

Other tags can be added after instantiation of the ConversionManager, but the one specified at instantiation will be used as default.

get_converted (*symbol, units='CAD', system=None, tag=None*)

Uses a Symbol's Dataframe, to build a new Dataframe, with the data converted to the new units

Parameters

- **symbol** (*str or tuple of the form (Dataframe, str)*) – String representing a symbol's name, or a dataframe with the data required to be converted. If supplying a dataframe, units must be passed.
- **units** (*str, optional*) – Specify the units to convert the symbol to, default to CAD
- **system** (*str, optional*) – If None, the default system specified at instantiation is used. System defines which conversion approach to take.
- **tag** (*str, optional*) – Tags define which set of conversion data is used. If None, the default tag specified at instantiation is used.

3.1.3 Symbols

class **Symbol** (*name, description=None, units=None, agg_method='PRIORITY_FILL', index-name='UNNAMED', indeximp='DatetimeIndexImp', freshthresh=0*)

Bases: *sqlalchemy.ext.declarative.api.Base, trump.tools.sqla.ReprMixin*

A Trump Symbol persistently objectifies indexed data

Use the SymbolManager class to create or retrieve existing symbols.

Parameters

- **name** (*str*) – The name of the symbol to be added to the database, serves as a primary key across the trump installation.
- **description** (*str, optional*) – a description of the symbol, just for notes.
- **units** (*str, optional*) – a string representing the units for the data.

- **agg_method** (*str*, default *PRIORITY_FILL*) – the method used for aggregating feeds, see `trump.aggregation.symbol_aggs.py` for the list of available options.
- **indexname** (*str*) – a proprietary name assigned to the index.
- **indeximp** (*str*) – a string representing an index implementer (one of the classes in `indexing.py`)
- **freshthresh** (*int*, default *0*) – number of minutes before the feed is considered stale.

add_feed (*feedlike*, ***kwargs*)
Add a feed to the Symbol

Parameters

- **feedlike** (*Feed* or *bFeed-like*) – The feed template, or Feed object to be added.
- **kwargs** – Munging instructions

add_tags (*tags*)
add a tag or tags to a symbol

Parameters **tags** (*str* or [*str*,]) – Tags to be added

add_validator (*val_template*)
Creates and adds a SymbolValidity object to the Symbol.

Parameters **validity_template** (*bValidity* or *bValidity-like*) – a validity template.

cache (*checkvalidity=True*, *staleonly=False*, *allowraise=True*)
Re-caches the Symbol's datatable by querying each Feed.

Parameters

- **checkvalidity** (*bool*, *optional*) – Optionally, check validity post-cache. Improve speed by turning to False.
- **staleonly** (*bool*, default *False*) – Set to True, for speed up, by looking at staleness
- **allowraise** (*bool*, default *True*) – AND with the Symbol.handle and Feed.handle's 'raise', set to False, to do a list of symbols. Note, this won't silence bugs in Trump, eg. unhandled edge cases. So, those still need to be handled by the application.

Returns

Return type *SymbolReport*

check_validity (*checks=None*, *report=True*)
Runs a Symbol's validity checks.

Parameters

- **checks** (*str*; [*str*,], *optional*) – Only run certain checks.
- **report** (*bool*, *optional*) – If set to False, the method will return only the result of the check checks (True/False). Set to True, to have a SymbolReport returned as well.

Returns

Return type Bool, or a Tuple of the form (Bool, SymbolReport)

describe
describes a Symbol, returns a string

isvalid
Quick access to the results of a check_validity report

Returns**Return type** Bool**set_indexing** (*index_template*)

Update a symbol's indexing strategy

Parameters **index_template** (*bIndex or bIndex-like*) – An index template used to overwrite all details about the symbol's current index.**to_json** ()

Returns the json representation of a Symbol object's tags, description, and meta data

update_handle (*chkpnt_settings*)

Update a symbol's handle checkpoint settings

Parameters **chkpnt_settings** (*dict*) – a dictionary where the keys are strings representing individual handle checkpoint names, for a Symbol (eg. `caching_of_feeds`, `feed_aggregation_problem`, ...) See `SymbolHandle.__table__.columns` for the current list.

The values can be either integer or BitFlags.

class SymbolTag (*tag, sym=None*)Bases: `sqlalchemy.ext.declarative.api.Base`, `trump.tools.sqla.ReprMixin`**class SymbolDataDef** (*datadef, sym=None*)Bases: `sqlalchemy.ext.declarative.api.Base`, `trump.tools.sqla.ReprMixin`

Indices

A *Symbol* object's *Index* stores the information required for Trump to cache and serve data with different types of pandas indices.

Warning: A Trump *Index* does not contain a list of hashable values, like a pandas index. It should not be confused with the datatable's index, however it is used in the creation of the datatable's index. A more appropriate name for the class might be `IndexCreationKwargs`.

class Index (*name, indimp, case=None, kwargs=None, sym=None*)Bases: `sqlalchemy.ext.declarative.api.Base`, `trump.tools.sqla.ReprMixin`**case**string used in a *IndexImplementer* switch statement.**indimp**string representing a *IndexImplementer*.**name**

string to name the index, only used when serving.

class IndexKwarg (*kword, val*)Bases: `sqlalchemy.ext.declarative.api.Base`, `trump.tools.sqla.ReprMixin`, `trump.tools.sqla.DuckTypeMixin`

Index Types

class IndexImplementer (*case, **kwargs*)Bases: `object`

IndexImplementer is the base required to implement an index of a specific type. The same instance is created at two points in the Trump dataflow:

- 1.the datatable getting cached and
- 2.the data being served.

The IndexImplementer should be idempotent, and dataframe/series agnostic.

Parameters

- **case** – str This should match a case used to switch the logic created in each subclass of IndexImplementer
- **kwargs** – dict

pytyp

alias of int

squatyp

alias of Integer

class DatetimeIndexImp (case, **kwargs)

Bases: *trump.indexing.IndexImplementer*

Implements a pandas DatetimeIndex

Cases include:

- **asis** - Cache timestamps to the database and drop any intelligence associated with the index, such as frequency. serve a Series with a DatetimeIndex, without frequency.

If the index consists of 4-digit integers, it will be treated as the year, in a date which is of the form YYYY-12-31.

- **asfreq** - Apply 'asfreq' logic prior to cache, and apply the same logic when serving.
- **date_range** - Create a new index, using pandas date_range(), at time of cache... NotImplemented yet.
- **guess** - NotImplemented yet. Attempt to guess the frequency at time of cache, and time of serve.
- **guess_post** - NotImplemented yet. Attempt to guess the frequency at time of serve, but store the cache unsaved.

In the event that case hasn't implemented the logic to handle a specific datatype, a rudimentary attempt to convert it to a DatetimeIndex is applied by inspecting the start and end, with the kwargs. passed pandas.DatetimeIndex constructor.

Parameters

- **case** – str This should match a case used to switch the logic created in each subclass of IndexImplementer
- **kwargs** – dict

pytyp

alias of datetime

squatyp

alias of DateTime

class PeriodIndexImp (case, **kwargs)

Bases: *trump.indexing.IndexImplementer*

Implements a pandas PeriodIndex

NotImplemented, yet.

sqldatyp

alias of `DateTime`

class StrIndexImp (*case*, ***kwargs*)

Bases: `trump.indexing.IndexImplementer`

Implements a pandas Index consisting of string objects.

Only method, is “asis”

Parameters

- **case** – str This should match a case used to switch the logic created in each subclass of `IndexImplementer`
- **kwargs** – dict

pytyp

alias of `str`

sqldatyp

alias of `String`

class IntIndexImp (*case*, ***kwargs*)

Bases: `trump.indexing.IndexImplementer`

Implements a pandas `Int64Index`.

Cases include:

- **asis** - attempts to pass the index through, without applying any logic. Use this, if the index is already integers, or unique and integer-like.
- **drop** - will drop the pandas index, to reset it.

Parameters

- **case** – str This should match a case used to switch the logic created in each subclass of `IndexImplementer`
- **kwargs** – dict

pytyp

alias of `int`

sqldatyp

alias of `Integer`

Validity Checking

class SymbolValidity (*symbol*, *vid*, *validator*, **args*)

Bases: `sqlalchemy.ext.declarative.api.Base`, `trump.tools.sqla.ReprMixin`

3.1.4 Feeds

class Feed (*symbol*, *ftype*, *sourcing*, *munging=None*, *meta=None*, *fnum=None*)

Bases: `sqlalchemy.ext.declarative.api.Base`, `trump.tools.sqla.ReprMixin`

The Feed object stores parameters associated with sourcing and munging a single series.

```

class FeedMeta (feed, attr, value)
    Bases: sqlalchemy.ext.declarative.api.Base, trump.tools.sqla.ReprMixin

class FeedSource (stype, sourcing_key, feed)
    Bases: sqlalchemy.ext.declarative.api.Base, trump.tools.sqla.ReprMixin

class FeedSourceKwarg (keyword, val, feedsource)
    Bases: sqlalchemy.ext.declarative.api.Base, trump.tools.sqla.ReprMixin,
    trump.tools.sqla.DuckTypeMixin

```

Feed Munging

```

class FeedMunge (order, mtype, method, feed)
    Bases: sqlalchemy.ext.declarative.api.Base, trump.tools.sqla.ReprMixin

class FeedMungeKwarg (keyword, val, feedmunge)
    Bases: sqlalchemy.ext.declarative.api.Base, trump.tools.sqla.ReprMixin,
    trump.tools.sqla.DuckTypeMixin

```

3.1.5 Centralized Data Editing

Each trump datatable comes with two extra columns beyond the feeds, index and final.

The two columns are populated by any existing *Override* and *FailSafe* objects which survive caching, and modification to feeds.

Any *Override* will get applied blindly regardless of feeds, while the *FailSafe* objects are used only when data isn't available for a specific point. Once a datapoint becomes available for a specific index in the datatable, the failsafe is ignored.

```

class Override (*args, **kwargs)
    Bases: sqlalchemy.ext.declarative.api.Base, trump.tools.sqla.ReprMixin

    An Override represents a single datapoint with an associated index value, applied to a Symbol's datatable after sourcing all the data, and will be applied after any aggregation logic

    comment
        a user field to store an arbitrary string about the override

    dt_log
        datetime that the override was created

    ind
        the repr of the object used in the Symbol's index.

    ornum
        Override number, uniquely assigned to every override

    symname
        symbol name, for the override

    user
        user name or process name that created the override

    val
        the repr of the object used as the Symbol's value.

class FailSafe (*args, **kwargs)
    Bases: sqlalchemy.ext.declarative.api.Base, trump.tools.sqla.ReprMixin

```

A FailSafe represents a single datapoint with an associated index value, applied to a Symbol's datatable after sourcing all the data, and will be applied after any aggregation logic, only where no other datapoint exists. It's a back-up datapoint, used only by Trump, when an NA exists.

Note: only datetime based indices with float-based data currently work with Overrides

comment
user field to store an arbitrary string about the FailSafe

dt_log
datetime of the FailSafe creation.

fsnum
Failsafe number, uniquely assigned to every FailSafe

ind
the repr of the object used in the Symbol's index.

symname
symbol name, for the override

user
user name or process name that created the FailSafe

val
the repr of the object used as the Symbol's value.

3.1.6 Reporting

During the cache process, information comes back from validity checks, and any exceptions. This area of Trump's code base is currently WIP, however the basic idea is that the caching of a Feed, returns a *FeedReport*. For each cached Feed, there would be one report, all of which would get aggregated up into, and combined with the symbol-level information, in a *SymbolReport*. When the SymbolManager caches one or more symbols, it aggregates SymbolReports into one big and final *TrumpReport*.

```
class FeedReport (num)
    Bases: object

    add_handlepoint (hpreport)
        Appends a HandlePointReport

    add_reportpoint (rpoint)
        Appends a ReportPoint

    asodict (handlepoints=True, reportpoints=True)
        Returns an ordered dictionary of handle/report points

class SymbolReport (name)
    Bases: object

    add_feedreport (freport)
        Appends a FeedReport

    add_handlepoint (hpreport)
        Appends a HandlePointReport

    add_reportpoint (rpoint)
        Appends a ReportPoint
```

asodict (*freports=True, handlepoints=True, reportpoints=True*)
Returns an ordered dictionary of feed, and handle/report points

class TrumpReport (*name*)

Bases: object

Each of the three levels of reports, have the appropriate aggregated results, plus collections of their own Handle-PointReport and ReportPoint objects.

class HandlePointReport (*handlepoint, trace*)

Bases: object

class ReportPoint (*reportpoint, attribute, value, extended=None*)

Bases: object

3.1.7 Error Handling

The Symbol & Feed objects have a single SymbolHandle and FeedHandle object accessed via their .handle attribute. They both work identically. The only difference is the column names that each have. Each column, aside from symname, represents a checkpoint during caching, which could cause errors external to trump. The integer stored in each column is a serialized BitFlag object, which uses bit-wise logic to save the settings associated with what to do upon an exception. What to do, mainly means deciding between various printing, logging, warning or raising options.

The Symbol's possible exception-inducing handle-points include:

- caching (of feeds)
- concatenation (of feeds)
- aggregation (of final value column)
- validity_check

class SymbolHandle (*chkpnt_settings=None, sym=None*)

Bases: sqlalchemy.ext.declarative.api.Base, trump.tools.sqla.ReprMixin

Stores instructions about how to handle exceptions thrown during specific points of Symbol caching:

```
sh = SymbolHandle({'aggregation' : BitFlag(36)}, aSymbol)
>>> sh.aggregation['email']
True
```

Parameters

- **chkpnt_settings** (*dict*) – A dictionary with keys matching names of the handle points and the values either integers or BitFlags
- **sym** (*str or Symbol*) – The Symbol that this SymbolHandle is associated with it.

The Feed's possible exception-inducing handle-points include:

- api_failure
- feed_type
- index_type_problem
- index_property_problem
- data_type_problem
- monounique

```
class FeedHandle (chkpnt_settings=None, feed=None)
    Bases: sqlalchemy.ext.declarative.api.Base, trump.tools.sqla.ReprMixin

    Stores instructions about specific handle points during Feed caching:
```

```
fh = FeedHandle({'api_failure' : BitFlag(36)}, aSymbol.feeds[0])
>>> fh.api_failure['email']
True
```

Parameters

- **chkpnt_settings** – dict A dictionary with keys matchin names of the handle points and the values either integers or BitFlags
- **feed** – Feed The feed that this FeedHandle is associated with it.

For example, if a feed source is prone to problems, set the `api_failure` to print the trace by setting the BitFlag object's 'stdout' flag to True, and the other flags to False. If there's a problem, Trump will attempt to continue, and hope that there is another feed with good data available. However, if a source should be reliably available, you may want to set the BitFlag object's 'raise' flag to True.

BitFlags

Trump stores instructions regarding how to handle exceptions in specific points of the cache process using a serializable object representing a list of boolean values called a BitFlag. There are two objects which make the BitFlag implementation work. There is the BitFlag object, which converts dictionaries and integers to bitwise logic, and then there is the BitFlagType which give SQLAlchemy the ability to create columns, and handle them appropriately, containing BitFlag objects.

```
class BitFlag (obj, defaultflags=None)
    Bases: sqlalchemy.ext.mutable.Mutable, object
```

An object used to encode and decode a boolean array as an an integer representing bitwise logic-flags.

There are 7 hardcoded flags:

- raise
- warn
- email
- dblog
- txtlog
- stdout
- report

Each can be set to True or False, with convenience either at instantiation, or key-base set operations.

Example of instatiation, setting the email and stdout flag to True:

```
BitFlag(['email', 'stdout'])
```

Example of instatiation, setting the email then, later setting stdout flag to True:

```
bf = BitFlag(['email'])
bf['stdout'] = True
```

After either running either of these, the BitFlag will have a value of:


```
>>> bf.val == 36
True

>>> print bf
raise warn EMAIL dblog txtlog STDOUT report

>>> print bf.bin_str
00100100

>>> print bf.email
True
```

...because the 3rd and 6th bit are set.

Warning: Flag state can be read from the accessors named after the flags, however, they can't be written to.

Parameters

- **(int, dict)** (*obj*,) – either the decimal form of the bitwise array, or a dictionary (complete or otherwise) of the form {flag : bool, flag : bool, ...}
- **dict** (*defaultflags*,) – a dictionary representing the default for one or more flags. Only applicable when a dictionary is passed to *obj*. It's ignored when *obj* is an integer.

__and__ (*other*)

Parameters *other* – int, BitFlag

BitFlag and integers work with the and operator using bitwise logic.

Returns BitFlag

__or__ (*other*)

Parameters *other* – int, BitFlag

BitFlag and integers work with the or operator using bitwise logic.

Returns BitFlag

asdict ()

convert the flags to a dictionary, with keys as flags.

bin

the binary equivalent

bin_str

the binary equivalent, as a string

class BitFlagType (*args, **kwargs)

Bases: sqlalchemy.sql.type_api.TypeDecorator

SQLAlchemy type definition for the BitFlag implementation. A BitFlag is a python object that wraps bitwise logic for hardcoded flags into a single integer value for quick database access and use.

The likely values assigned, will commonly be from the list below. Use Bitwise logic operators to make other combinations.

Desired Effect	BitFlag Instantiation	Description
Raise-Only	BitFlag(1)	Raise an Exception
Warn-Only	BitFlag(2)	Raise a Warning
Email-Only *	BitFlag(4)	Send an E-mail
DBLog-Only *	BitFlag(8)	Log to the Database
TxtLog-Only	BitFlag(16)	Text Log
StdOut-Only	BitFlag(32)	Standard Output Stream
Report-Only	BitFlag(64)	Report
TxtLog and StdOut	BitFlag(48)	Print & Log

- Denotes Features not implemented, yet.

The implementation is awkward, all in the name of speed. There are $(4 + 7 \times \# \text{ of Feeds})$ BitFlags, per symbol. So they are serialized into integers, rather than having $(4 + 7 \times \# \text{ of Feeds}) \times 7$ boolean database columns.

3.2 Data Flow

Trump centralizes the flow of information using two concepts:

1. Objectification - the process of persistently storing information about data.
2. Caching - the process of fetching data, saving it systematically, and serving it intelligently.

3.2.1 Objectification

The objectification happens via an addition-like process entailing the instantiation of one or more symbols. The objectification enables downstream applications to work with symbol names in order to force the caching, and be served reliable data.

There are two approaches to perform the objectification instantiation of Symbols

1. First Principles (from ORM)
2. Template Based (from Special Python Classes + ORM)

First Principles

The first principles approach to using Trump is basically direct access to the SQLAlchemy-based object-relational model. It's time consuming to develop with, but necessary to understand in order to design new intelligent templates.

Using Trump's ORM, the process is something akin to:

For Every Symbol:

1. Instantiate a new *Symbol*
2. Optionally, add some *SymbolTag*
3. Optionally, adjust the symbol's *Index* case and type
4. Optionally, adjust the symbol's *SymbolHandle* handlepoints
5. Instantiate one ore more *Feed* objects
6. For each Feed, update *FeedMeta*, *FeedSource* details
7. Optionally, adjust each feed's *FeedMunge* instructions
8. Optionally, adjust each feed's *FeedHandle* handlepoints

9. Optionally, adjust each Symbol's *SymbolValidity* instructions

Template Based

By setting up, and using Trump template classes, the two steps below replace steps 1 to 8 of the first principles approach.

For Every Kind of Symbol:

1. Create custom templates for common sources of proprietary data.

For Every Symbol:

2. Instantiate a new *Symbol* using a template containing Tag, Feed, Source, Handle, Validity settings.
3. Tweak any details uncovered by the chosen templates for the symbol, or any of it's feeds.

In practice, it's inevitable that templates will be used where possible, and do the heavy lifting of instantiation, but tweaks to each symbol would be made post-instantiation.

3.2.2 Caching

The cache process, is more than just caching, but that's the main purpose. The cache process, essentially builds a fresh datatable. In order to cache a symbol, Trump performs the following steps:

For each Feed...

1. Fetches a fresh copy of each Feed, based on the *FeedSource* parameters.
2. Munges each Feed, based on the *FeedMunge* parameters.
3. Converts the datatype using a *SymbolDataDef*

Then...

4. Concatenates the data from each feed, into a dataframe.
5. Converts the index datatype using the Symbol's *Index* parameters.
6. Two columns are appended to the dataframe, one for overrides, one for failsafes. Any which exist, are fetched.
7. An aggregation method is used to build a final series out of the data from the feeds and any overrides/failsafes.
8. The dataframe is stored in the database, in it's own table, called a datatable.
9. Optionally, any validity checks, which are set up in *SymbolValidity*, are performed.

When executed, data from each Feed is queried, and munged according to predefined instructions, on a per-feed basis. The feeds are joined together, each forming columns of a pandas Dataframe. A *IndexImplementor* corrects the index. An aggregation method converts the Dataframe into a single, final, Series. Depending on the aggregation method, any single values are overrode, and blanks get populated, based on any previously defined *Override* and *FailSafe* objects associated with the symbol being cached.

The Datatable & Aggregation

Steps #6, #7 & #8 above are easiest to understand, with a graphical look at the final product: a cached Symbol's datatable.

An example of a datatable, is in the figure below. This, is a simple table, common to anybody with SQL knowledge.

	indx	final	override_feed000	feed001	feed002	failsafe_feed999
1	2015-03-02 ...	43.88	{null}	43.88	43.88	{null}
2	2015-03-03 ...	43.28	{null}	43.28	43.28	{null}
3	2015-03-04 ...	43.06	{null}	43.06	43.06	{null}
4	2015-03-05 ...	43.11	{null}	43.11	43.11	{null}
5	2015-03-06 ...	42.37	42.37	42.36	42.36	{null}
6	2015-03-09 ...	42.85	{null}	42.85	42.85	{null}
7	2015-03-10 ...	42.03	{null}	42.03	42.03	{null}
8	2015-03-11 ...	41.98	{null}	{null}	41.98	{null}
9	2015-03-12 ...	41.02	{null}	{null}	{null}	41.02
10	2015-03-13 ...	41.38	{null}	{null}	41.38	41.44

Fig. 3.5: Example of a symbol's datatable, with two feeds of data, both with problems.

The example datatable, seen above, is one symbol with two feeds, both of which had problems. One of the feeds stopped completely on the 11th, the other had a missing datapoint. Plus, a previous problem, looks like it was manually overrode on the 6th, but then later, the feed started working again. The overrides and failsafes were applied appropriately on the 6th, and the 12th, while the failsafe on the 10th, was ignored after the feed #2 started working again.

It's easy to imagine the simple Dataframe after step #5 of the cache process. It would have a single index, then a column for every Feed. #6, appends the two columns mentioned, along with any individual datapoints. Then an aggregation method creates the 'final' column. Details about the specific aggregation method are defined at, or updated after, Symbol instantiation. Up to and including the aggregation, all operations are simply changing the dataframe of feeds, overrides, and failsafes.

After the final is calculated, the dataframe is stored until the next cache, as a table - the datatable, illustrated in the figure above. It can then be quickly checked for validity and served to applications.

3.3 Aggregation Methods

Trump currently has two types of aggregation methods:

1. Apply-Row
2. Choose-Column

As the names infer, the apply-row methods have one thing in common, they build the final data values by looking at each row of the datatable, one at a time. The choose-column methods, compare the data available in each column, then return an entire series. Row-apply methods all take a pandas Series, and return a value. Column-choose methods all take a pandas Dataframe, and return a series.

Row-apply functions are invoked using the pseudo code below:

```
df['final'] = df.apply(row_apply_method, axis=1)
```

Column-choose functions are invoked using the pseudo code below:

```
df['final'] = column_choose_method(df)
```

Both methods have access to the data in the override, and failsafe, columns so it's technically possible to create a method which overloads the behaviour of these columns. It is the responsibility of each method to implement the

override, and failsafe, logic.

3.3.1 Apply-Row Methods

Each of these methods, can be thought of as a for-loop that looks at each row of the datatable, then decides on the correct value for the final column, on a row by row basis.

The datatable, as a Dataframe, gets these methods applied. The columns are sorted prior to being passed. So, the value at index 0, is always the override datapoint, if it exists, and the value at index -1, is always the failsafe datapoint, if it exists. Everything else, that is, the feeds, are in columns 1 through n, where n is the number of feeds.

static `ApplyRow.priority_fill (adf)`

Looks at each row, and chooses the value from the highest priority (lowest #) feed, one row at a time.

static `ApplyRow.mean_fill (adf)`

Looks at each row, and calculates the mean. Honours the Trump override/failsafe logic.

static `ApplyRow.median_fill (adf)`

Looks at each row, and chooses the median. Honours the Trump override/failsafe logic.

static `ApplyRow.custom (adf)`

A custom Apply-Row Aggregator can be defined, as any function which accepts a Series, and returns any number-like object, which will get assigned to the Dataframe's 'final' column in using the pandas .apply, function.

Note: The aggregation methods are organized in the code using private mixin classes. The FeedAggregator object handles the implementation of every static method, based solely on it's name. This means that any new methods added, must be unique to either mixin.

3.3.2 Choose-Column Methods

Each of these methods, can be thought of as a for-loop that looks at each column of the datatable, then chooses the appropriate feed to use, as final. They all still apply overrides and failsafes on a row-by-row basis.

The datatable, as a Dataframe, is passed to these methods in a single call.

static `ChooseCol.most_populated (adf)`

Looks at each column, using the one with the most values Honours the Trump override/failsafe logic.

static `ChooseCol.most_recent (adf)`

Looks at each column, and chooses the feed with the most recent data point. Honours the Trump override/failsafe logic.

static `ChooseCol.custom (adf)`

A custom Choose-Column Aggregator can be defined, as any function which accepts a dataframe, and returns any Series-like object, which will get assigned to the Dataframe's 'final' column.

Note: See the note in the previous section about custom method naming.

Templating

4.1 Template Base Classes

Trump's templating system consists of pure-python objects, which can be converted into either lists, dictionaries, or ordered dictionaries, which can then be used in the generalized constructors of Trump's SQLAlchemy based ORM system.

class bTags

Bases: `trump.templating.converters._ListConverter`

Tag Templates are any object which implements a property called `as_list`, which returns a list of strings

The Base Template for Tag Templates inherits from `_listConverter`, which implements `as_list()`. `as_list()` looks at the attributes defined and set to `True`, in order to include the list of tags.

class bMunging

Bases: `trump.templating.converters._OrderedDictConverter`

Munging Templates are any object which implements a property called `as_odict`, which returns an `odict` where each key is a function in `munging_methods`, and its value is an object which represents the parameters to use on that object. This should be sufficient to pass to a Feed constructor's munging parameter, which then becomes `FeedMungingArgs` objects making up a `FeedMunge` object, of which will be the instructions associated with a specific Feed object.

class bSource

Bases: `trump.templating.converters._DictConverter`

Source Templates are any object which implements a property called `as_dict`. The keywords and values of which are sufficient to pass to a Feed constructor's source parameter, which then become `FeedSource` objects making up a source.

class bFeed

Bases: `object`

Feed objects need an `tags`, `sourcing`, `munging` and `validity` attribute defined. They must be a list, dict, `odict`, and dict, respectively.

class bValidity

Bases: `object`

Validity Templates are any object which implements an attribute named `'validator'`, and optionally some additional arguments as `arga`, `argb`, `argc`, `argd` and `arge`.

class bIndex

Bases: `object`

Index Templates are any object which implements sufficient information to fully define an `IndexImplementer` via it's name, case and associated kwargs, via three attributes called `imp_name` (string), `case` (string), `kwargs` (dict).

4.2 Template Classes

4.2.1 Tag Templates

```
class AssetTT (cls)
    Bases: trump.templating.bases.bTags
    implements groups of tags for certain asset classes

class GenericTT (tags)
    Bases: trump.templating.bases.bTags
    implements generic list of tags via boolean attributes

class SimpleTT (tags)
    Bases: trump.templating.bases.bTags
    implements a simple list of tags via a single attribute

class SimpleTT (tags)
    Bases: trump.templating.bases.bTags
    implements a simple list of tags via a single attribute
```

4.2.2 Munging Templates

```
class AbsMT
    Bases: trump.templating.bases.bMunging, trump.templating.munging_helpers.mixin_pab
    Example munging template, which implements an absolute function.

class AsFreqMT (**kwargs)
    Bases: trump.templating.bases.bMunging, trump.templating.munging_helpers.mixin_pab
    Example munging template, which implements pct_change.

class RollingMeanMT (**kwargs)
    Bases: trump.templating.bases.bMunging, trump.templating.munging_helpers.mixin_pnab
    Example munging template, which implements a rolling mean.

class FFillRollingMeanMT (**kwargs)
    Bases: trump.templating.bases.bMunging, trump.templating.munging_helpers.mixin_pab,
    trump.templating.munging_helpers.mixin_pnab
    Example munging template, which implements a ffill using the generic pandas attribute based munging, and
    then a rolling mean.

class RollingMeanFFillMT (**kwargs)
    Bases: trump.templating.bases.bMunging, trump.templating.munging_helpers.mixin_pab,
    trump.templating.munging_helpers.mixin_pnab
    Example munging template, which implements a rolling mean and a generic pandas attribute based munging
    step.
```



```
class SimpleExampleMT (periods, window)
    Bases: trump.templating.bases.bMunging, trump.templating.munging_helpers.mixin_pnab,
    trump.templating.munging_helpers.mixin_pab

    Example munging template, which has defaults to forward fill, and a minimum period argument of 5
```

```
class MultiExampleMT (pct_change_kwargs, add_kwargs)
    Bases: trump.templating.bases.bMunging, trump.templating.munging_helpers.mixin_pnab,
    trump.templating.munging_helpers.mixin_pab

    Example munging template, which implements a pct_change and add, using two sets of kwargs
```

4.2.3 Source Templates

```
class DBapiST (dsn=None, user=None, password=None, host=None, database=None, sourcing_key=None)
    Bases: trump.templating.bases.bSource, trump.templating.source_helpers.mixin_dbCon,
    trump.templating.source_helpers.mixin_dbIns

    implements the generic source information for a DBAPI 2.0 driver
```

```
class PyDataDataReaderST (data_source, name, column='Close', start='2000-01-01', end='now')
    Bases: trump.templating.bases.bSource

    implements the pydata datareaders sources
```

```
class PyDataCSVST (filepath_or_buffer, data_column, **kwargs)
    Bases: trump.templating.bases.bSource

    implements pandas.read_csv source
```

4.2.4 Feed Templates

```
class DBapiFT (table=None, indexcol=None, datacol=None, dsn=None, user=None, password=None,
               host=None, database=None, sourcing_key=None)
    Bases: trump.templating.bases.bFeed

    Feed template for DBAPI 2.0, which collects up everything it needs via parameters about the connection and
    information.
```

```
class SQLFT (command)
    Bases: trump.templating.templates.ExplicitCommandFT

    Just wrap inherit, for renaming purposes.
```

```
class QuandlFT (dataset, **kwargs)
    Bases: trump.templating.bases.bFeed

    Feed template for a Quandl data source
```

```
class QuandlSecureFT (dataset, **kwargs)
    Bases: trump.templating.templates.QuandlFT

    Feed template for a Quandl data source, authtoken left in config file.
```

```
class GoogleFinanceFT (name, column='Close', start='1995-01-01', end='now')
    Bases: trump.templating.bases.bFeed

    PyData reader feed, generalized for google finance.
```

```
class YahooFinanceFT (name, column='Close', start='1995-01-01', end='now')
```

```
    Bases: trump.templating.bases.bFeed
```

PyData reader feed, generalized for Yahoo Finance.

```
class StLouisFEDFT (name, column=None, start='1995-01-01', end='now')
```

```
    Bases: trump.templating.bases.bFeed
```

PyData reader feed, generalized for St Louis FED.

```
class CSVFT (filepath_or_buffer, data_column, **kwargs)
```

```
    Bases: trump.templating.bases.bFeed
```

Creates a feed from a CSV.

4.2.5 Index Templates

```
class FFillIT (freq='B')
```

```
    Bases: trump.templating.bases.bIndex
```

4.2.6 Validity Templates

```
class FeedsMatchVT (feed_left=1, feed_right=2, lastx=10)
```

```
    Bases: trump.templating.bases.bValidity
```

```
class DateExistsVT (date='today')
```

```
    Bases: trump.templating.bases.bValidity
```

Source Extensions

5.1 Creating & Modifying Source Extensions

This section of the docs is really only intended for those who want to write, or modify, their own source extensions. But, it can be helpful to understand how they work, even for those who don't want to write an extension.

Trump's framework enables sources of varying, dynamic, and proprietary types. A source extension is basically a generalized way of getting a pandas Series out of an existing external API. For instance examples include, the pandas datareader, a standardized DBAPI 2.0 accessible schema, a proprietary library, or something as simple as a CSV file. At a high level, each symbol's feed's source's kwargs are passed to the appropriate source extension, based on the defined source type.

When each symbol is cached, it loops through each of it's feeds. Each feed's source is queried, using four critical python lines in `orm.Feed.cache()`:

```
if stype in sources:
    self.data = sources[stype](self.ses, **kwargs)
else:
    raise Exception("Unknown Source Type : {}".format(stype))
```

The important line, is the second one. 'sources', is a dictionary loaded every time trump's `orm.py` is imported. The key's are just strings representing the "Source Type", eg. "DBAPI", "Quandl", "BBFetch" (Example of a proprietary source). The values of the sources dictionary are SourceExtension objects. The SourceExtension objects wrap modules discovered dynamically when `loader.py` scans the source extension folder. The code for the SourceExtension is below:

```
class SourceExtension(object):
    def __init__(self, mod): #instantiated only once per import of trump.orm
        self.initialized = False
        self.mod = mod
        self.renew = mod.renew
        self.Source = mod.Source
    def __call__(self, _ses, **kwargs): #called each symbol's feed's cache (in the second line above,
        if not self.initialized or self.renew:
            self.fetcher = self.Source(_ses, **kwargs)
            self.initialized = True
        return self.fetcher.getseries(_ses, **kwargs)
```

A SourceExtension is instantiated only once, when `loader.py` passes a module it discovered. The modules, are the "source extension", which are just simply python files, required to be created in a standard way. The standard can be illustrated with an example. Below, is an example csv-file source extension (which may be stale, compared to the actual csv extension).

See `trump/extensions/source` for more examples.

```
stype = 'PyDataCSV'
renew = False

class Source(object):
    def __init__(self, ses, **kwargs):
        from pandas import read_csv
        self.read_csv = read_csv

    def getseries(self, ses, **kwargs):

        col = kwargs['data_column']
        del kwargs['data_column']

        fpob = kwargs['filepath_or_buffer']
        del kwargs['filepath_or_buffer']

        df = self.read_csv(fpob, **kwargs)

        data = df[col]

        return data
```

Noticed that the two variables, `stype` & `renew`, as well as the `Source` class, are used in the `SourceExtension` instantiation.

5.1.1 Source Extension Standard Form

Any extension module needs 3 things; an `stype` variable, `renew` variable, and `Source` class.

stype (str)

`stype` is the string used in the ‘sources’ dictionary mentioned earlier, and must match the `stype` set in the corresponding `Source` template(s).

renew (boolean)

`renew` is a boolean, which determines if the `Source` object is reinstantiated on every use. For instance, one might create a source, which sets up a database connection, which stays open for the life of any script using trump’s orm, but only if that specific source is used at least once. `Renew` would be set to `False`, and the connection logic, would be put in `Source.__init__`. Alternatively, if a new connection would be required on every symbol’s cache, `renew` would be set to `True`. The tradeoffs, are speed and resource constraints. Both `__init__` and `getseries` get the same arguments. The current live trump session, and the symbol’s feed’s source `kwargs`.

Source (class)

`Source` is an an object with one other method, `getseries`, other than the constructor (`__init__`). Both take the same arguments: the trump session, and the Symbol’s Feed’s `Source`’s `kwargs`. `getseries`, returns a dataframe.

5.2 Pre-Installed Source Extensions

5.2.1 BBFetch

```
# the directory is tx-bbfetch
stype = 'BBFetch'
renew = True
```

Required kwargs:

- 'elid'
- 'bbtype' = ['COMMON', 'BULK'], then a few relevant kwargs depending on each.

Optional kwargs:

- 'duphandler' - 'sum'
- 'croptime' - boolean

5.2.2 DBAPI

```
# the directory is tx-dbapi
stype = 'DBAPI'
renew = True
```

The DBAPI driver, will use by default the same driver SQLAlchemy is using for trump. There is currently no way to change this default. It's assumed that the driver is DBAPI 2.0 compliant.

Required kwargs include:

- 'dbinstype' which must be one of 'COMMAND', 'KEYCOL', 'TWOKEYCOL'
- 'dsn', 'user', 'password', 'host', 'database', 'port'

Optional kwargs include:

- duphandler ['sum'] which just groups duplicate index values together via the sum.

Additional kwargs:

Required based on 'dbinstype' chosen:

'COMMAND' : - 'command' which is just a SQL string, where the first column becomes the index, and the second column becomes the data.

'KEYCOL' : - ['indexcol', 'datacol', 'table', 'keycol', 'key']

'TWOKEYCOL' : - ['indexcol', 'datacol', 'table', 'keyacol', 'keya', 'keybcol', 'keyb']

5.2.3 psycopg2

```
# the directory is tx-psycopg2
stype = 'psycopg2'
renew = True
```

Started extension for a Postgres-specific source.

Not fully implemented.

5.2.4 PyDataCSV

```
# the directory is tx-pydatacsv
stype = 'PyDataCSV'
renew = False
```

All kwargs are passed to panda's read_csv function.

Additional required kwargs:

- 'filepath_or_buffer' - should be an absolute path. Relative will only work, if caching is only performed by a python script which can access the relative path.
- 'data_column' - the specific column required, so to turn the dataframe into a series.

5.2.5 PyDataDataReaderST

```
# the directory is tx-pydatadatareaderst
stype = 'PyDataDataReaderST'
renew = True
```

This uses pandas.io.data.DataReader, all kwargs get passed to that.

start and end are optional, but must be of the form 'YYYY-MM-DD'.

Will default to since the beginning of available data, and run through "today".

data_column is required to be specified as well.

5.2.6 Quandl

```
# the directory is tx-quandl
stype = 'Quandl'
renew = True
```

All kwargs are passed to Quandl's API quandl.get()

An additional 'fieldname' is available to select a specific column if a specific quandl DB, doesn't support quandl's version of the same feature.

5.2.7 SQLAlchemy

```
# the directory is tx-sqlalchemy
stype = 'SQLAlchemy'
renew = True
```

a SQLAlchemy based implementation...so an engine string could be used.

Not fully implemented

5.2.8 WorldBankST

```
# the directory is tx-worldbankst
stype = 'WorldBankST'
renew = False
```

Uses `pandas.io.wb.download` to query indicators, for a specific country.

country, must be a world bank country code.

Some assumptions as implied about the indicator and the first level of the index. This may not work for all worldbank indicators.

User Interface

6.1 UI Prototype

A preliminary user interface for Trump is being prototyped.

6.1.1 Web Interface

The web UI was born out of Flask, Jinja2 and Bootstrap “hello world”.

Some screen shots, of the beginning, are below.

The screenshot displays a web application interface for searching data. At the top, a dark navigation bar contains links: Trump, Search, Tags, Status, List, and About. The main content area is titled 'Search'. It features a search input field with the text 'GDP' and a 'Search' button. Below the input, a blue button indicates 'searched: GDP'. A message states '0 SQL, 212 Fuzzy Results found.' and a 'Did SQL search' button is present. To the right, a 'Search By' dropdown menu is open, showing options: Name (checked), Description, Tags (checked), Meta, and Fuzzy (Beta). The search results are displayed as a list of items, each with a header, a date range, and tags. The first four items are visible:

Item	Header	Units	Description	Date Range	Tags
1	GDP_ABW	NoUnits	GDP (current US\$)	2014-12-31 to 1960-12-31	GDP, economics, world bank
2	GDP_AFG	NoUnits	GDP (current US\$)	2014-12-31 to 1960-12-31	GDP, economics, world bank
3	GDP_AGO	NoUnits	GDP (current US\$)	2014-12-31 to 1960-12-31	GDP, economics, world bank
4	GDP_ALB	NoUnits	GDP (current US\$)	2014-12-31 to 1960-12-31	GDP, economics, world bank

Fig. 6.1: SQL-like search, is straight forward and as expected.

The screenshot shows the Trump SymbolManager web interface for the symbol 'TSLA'. The top navigation bar includes links for Trump, Search, Tags, Status, List, and About. Below this is a secondary navigation bar with buttons for Symbol, Munging, Validity, Index, Data, Analyze, Download, Chart, Log, and Edit. The main content area displays the symbol 'TSLA' with its description 'Tesla Motor Company Common Stock' and various tags like 'Company', 'ETF', 'equity', and 'USD:us11'. It also shows the data source 'pandas.tseries.index.DatetimeIndex' and a list of recent data points. On the right, there are buttons for 'Cache' and 'SymbolManager'. Below the main content, there are two feeds: 'YahooFinanceFT' and 'GoogleFinanceFT', each with a table of data columns and a 'Feed #'. At the bottom, it shows the last completed and attempted cache times.

Fig. 6.2: An example of symbol page, for a symbol with two feeds.

And, much, much more, coming soon...

6.1.2 Search

Trump's SymbolManager object, has basic/expected SQL-enabled search functionality.

The Trump UI prototype is boosted by an ElasticSearch server with a single index consisting of symbol, tag, description, and meta data. To add a symbol to the index, use the json created from Symbol.to_json().

6.1.3 Background Caching

Trump's caching process isn't blazing fast, which means using the UI to kick off caching of one or more symbols, requires a background process in order for the web interface to stay responsive.

A very simple RabbitMQ consumer application, is included with the UI, which listens for the instruction to cache. The python pika package is required.

Trump Search Tags Status List About

Symbol Munging Validity Index Data Analyze Download Chart Log Edit

SPY Analyze @ freq W

SPDR S&P 500 ETF Trust

Index Information

```
DatetimeIndex(['1995-01-08', '1995-01-15', '1995-01-22', '1995-02-05', '1995-02-12', '1995-02-19', '1995-03-05', '1995-03-12', ..., '2015-05-10', '2015-05-17', '2015-05-24', '2015-06-07', '2015-06-14', '2015-06-21', '2015-06-28', '2015-07-05', '2015-07-12'], dtype='datetime64[ns]', length=1071, freq='W-SUN', tz=None)
```

SPY float64
dtype: object

Raw

	SPY
1995-01-08	46.046799
1995-01-15	46.734299
1995-01-22	46.546799
1995-01-29	47.109299
1995-02-05	48.031200

SPY

Fig. 6.3: View the index, data, and do common analysis. Or, download to excel/csv...

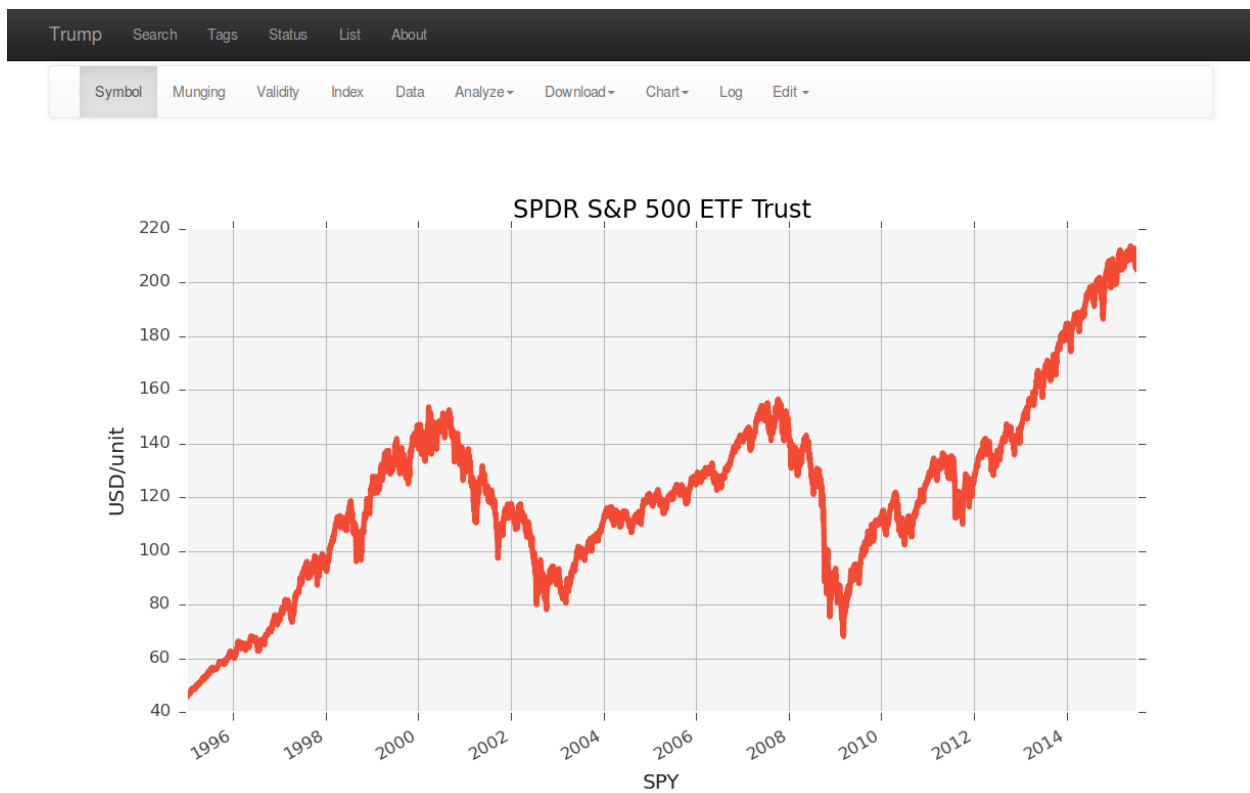


Fig. 6.4: Histograms and basic charting are available.

Trump
Search
Tags
Status
List
About

2015-07-13 00:00:00	264.0	264.0	262.160004	262.16	None
2015-07-14 00:00:00	265.649994	None	265.649994	265.65	None
2015-07-15 00:00:00	263.140015	None	263.140015	263.14	None
2015-07-16 00:00:00	266.679993	None	266.679993	266.68	None
2015-07-17 00:00:00	274.660004	None	274.660004	274.66	274.5

No Comment

Submit

Overrides

2015-07-18 23:51:48.935483 by user "None"	2015-07-13 00:00:00 264.0	Actually closed at 264, 262.12 was the halt.
--	-----------------------------	--

Fail Safes

2015-07-18 23:53:02.349267 by user "None"	2015-07-17 00:00:00 274.5	It will likely close near 274.5 today, I want to try printing a report scheduled for tomorrow.
--	-----------------------------	--

Feed Details

Feed #	YahooFinanceFT	data_column	Close	api_failure	index_property_problem
0	PyDataDataReaderST	data_source	yahoo	RAISE warn email dblog txlog stdout report	RAISE warn email dblog txlog stdout report
ON		end	now	empty_feed	data_type_problem
		name	TSLA	RAISE warn email dblog txlog stdout report	RAISE warn email dblog txlog stdout report
		start	1995-01-01	index_type_problem	monounique

Fig. 6.5: Overrides and failsafes, are what makes Trump amazing for business processes.

Trump	Search	Tags	Status	List	About
SCSSSLTIQ			Never cached	Never cached	Cache SCSS : Long Term Investments
SEESINQ			Never cached	Never cached	Cache SEE : Intangibles, Net
SENEASOETQ			Never cached	Never cached	Cache SENE : Other Equity, Total
SPY			2015 Wed Jul 15 @ 7:03 AM	2015 Wed Jul 15 @ 7:03 AM	Cache SPDR S&P 500 ETF Trust
SSFNSNIQ			Never cached	Never cached	Cache SSFN : Net Income
STESADTQ			Never cached	Never cached	Cache STE : Accumulated Depreciation, Total
STRISIATCIEIQ			Never cached	Never cached	Cache STRI : Income Available to Common Incl. Extra Items
SYRGSNIQ			Never cached	Never cached	Cache SYRG : Net Income
SYUTSONQ			Never cached	Never cached	Cache SYUT : Other, Net
TAITTRNQ			Never cached	Never cached	Cache TAIT : Total Receivables, Net
THTITAPQ			Never cached	Never cached	Cache THTI : Accounts Payable
TISITLSEQ			Never cached	Never cached	Cache TISI : Total Liabilities & Shareholders' Equity
TLXTRQ			Never cached	Never cached	Cache TLX : Revenue
TOFTOCLTQ			Never cached	Never cached	Cache TOF : Other Current liabilities, Total
TRLATCASTIQ			Never cached	Never cached	Cache TRLA : Cash and Short Term Investments
TRNOTDEEEIQ			Never cached	Never cached	Cache TRNO : Diluted EPS Excluding Extraordinary Items
TRNSTOLTATQ			Never cached	Never cached	Cache TRNS : Other Long Term Assets, Total
TSLA			2015 Sat Jul 18 @ 11:53 PM	2015 Sat Jul 18 @ 11:53 PM	Cache Tesla Motor Company Common Stock
UPIUTCLOQ			Never cached	Never cached	Cache UPI : Total Current Liabilities
UQMUIBTQ			Never cached	Never cached	Cache UQM : Income Before Tax
VENTURE_Biotechnology			Never cached	Never cached	Cache Venture Capital Investments By Industry

Fig. 6.6: The last time a symbol was attempted, and successfully cached, are available.

Trump Search Tags Status List About

Tags

Cache	Balance Sheet	300
Cache	Company	1
Cache	Consumer	1
Cache	ETF	3
Cache	FED	600
Cache	Fundamental	300
Cache	GDP	212
Cache	Monetary Policy	600
Cache	Price Index	1
Cache	Seasonally Adjusted	1
Cache	US	318
Cache	Venture Capital	18
Cache	economics	212
Cache	equity	2
Cache	iShares	1

Click "Cache" to cache all symbols tagged, or click the tag to view all symbols

Fig. 6.7: Browse and cache sets of symbols, based on tags...

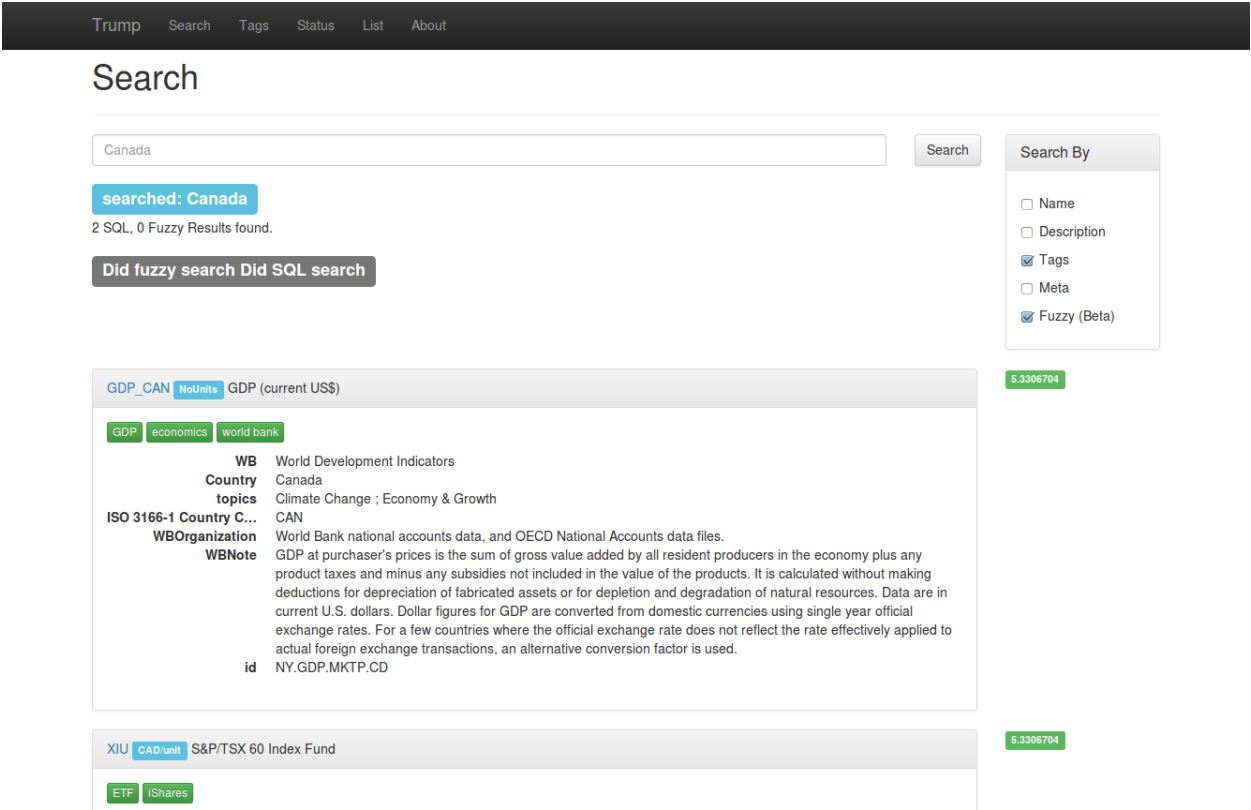


Fig. 6.8: ElasticSearch, makes searching much cooler...

t

`trump.indexing`, [22](#)

`trump.templating.bases`, [35](#)

Symbols

`__and__()` (BitFlag method), 29

`__or__()` (BitFlag method), 29

A

AbsMT (class in trump.templating), 36

`add_fail_safe()` (SymbolManager method), 13

`add_feed()` (Symbol method), 21

`add_feedreport()` (SymbolReport method), 26

`add_handlepoint()` (FeedReport method), 26

`add_handlepoint()` (SymbolReport method), 26

`add_override()` (SymbolManager method), 15

`add_reportpoint()` (FeedReport method), 26

`add_reportpoint()` (SymbolReport method), 26

`add_tags()` (Symbol method), 21

`add_validator()` (Symbol method), 21

`asdict()` (BitFlag method), 29

AsFreqMT (class in trump.templating), 36

`asodict()` (FeedReport method), 26

`asodict()` (SymbolReport method), 26

AssetTT (class in trump.templating), 36

B

bFeed (class in trump.templating), 35

bin (BitFlag attribute), 29

bin_str (BitFlag attribute), 29

bIndex (class in trump.templating), 35

BitFlag (class in trump.tools.bitflags), 28

BitFlagType (class in trump.tools.bitflags), 29

bMunging (class in trump.templating), 35

bSource (class in trump.templating), 35

bTags (class in trump.templating), 35

`build_view_from_tag()` (SymbolManager method), 15

bValidity (class in trump.templating), 35

C

`cache()` (Symbol method), 21

case (Index attribute), 22

`check_validity()` (Symbol method), 21

comment (FailSafe attribute), 26

comment (Override attribute), 25

`complete()` (SymbolManager method), 18

ConversionManager (class in trump.orm), 20

`create()` (SymbolManager method), 18

CSVFT (class in trump.templating), 38

`custom()` (ApplyRow static method), 33

`custom()` (ChooseCol static method), 33

D

DateExistsVT (class in trump.templating), 38

DatetimeIndexImp (class in trump.indexing), 23

DBapiFT (class in trump.templating), 37

DBapiST (class in trump.templating), 37

`delete()` (SymbolManager method), 18

describe (Symbol attribute), 21

dt_log (FailSafe attribute), 26

dt_log (Override attribute), 25

F

FailSafe (class in trump.orm), 25

Feed (class in trump.orm), 24

FeedHandle (class in trump.orm), 27

FeedMeta (class in trump.orm), 24

FeedMunge (class in trump.orm), 25

FeedMungeKwarg (class in trump.orm), 25

FeedReport (class in trump.reporting.objects), 26

FeedsMatchVT (class in trump.templating), 38

FeedSource (class in trump.orm), 25

FeedSourceKwarg (class in trump.orm), 25

FFillIT (class in trump.templating), 38

FFillRollingMeanMT (class in trump.templating), 36

`finish()` (SymbolManager method), 18

fsnum (FailSafe attribute), 26

G

GenericTT (class in trump.templating), 36

`get()` (SymbolManager method), 18

`get_converted()` (ConversionManager method), 20

GoogleFinanceFT (class in trump.templating), 37

H

HandlePointReport (class in trump.reporting.objects), 27

I

ind (FailSafe attribute), 26

ind (Override attribute), 25

Index (class in trump.orm), 22

IndexImplementer (class in trump.indexing), 22

IndexKwarg (class in trump.orm), 22

indimp (Index attribute), 22

IntIndexImp (class in trump.indexing), 24

isvalid (Symbol attribute), 21

M

mean_fill() (ApplyRow static method), 33

median_fill() (ApplyRow static method), 33

most_populated() (ChooseCol static method), 33

most_recent() (ChooseCol static method), 33

MultiExampleMT (class in trump.templating), 37

N

name (Index attribute), 22

O

ornum (Override attribute), 25

Override (class in trump.orm), 25

P

PeriodIndexImp (class in trump.indexing), 23

priority_fill() (ApplyRow static method), 33

PyDataCSVST (class in trump.templating), 37

PyDataDataReaderST (class in trump.templating), 37

pytyp (DatetimeIndexImp attribute), 23

pytyp (IndexImplementer attribute), 23

pytyp (IntIndexImp attribute), 24

pytyp (StrIndexImp attribute), 24

Q

QuandlFT (class in trump.templating), 37

QuandlSecureFT (class in trump.templating), 37

R

ReportPoint (class in trump.reporting.objects), 27

RollingMeanFFillMT (class in trump.templating), 36

RollingMeanMT (class in trump.templating), 36

S

search() (SymbolManager method), 18

search_meta() (SymbolManager method), 19

search_meta_specific() (SymbolManager method), 19

search_tag() (SymbolManager method), 19

set_indexing() (Symbol method), 22

SimpleExampleMT (class in trump.templating), 36

SimpleTT (class in trump.templating), 36

squatyp (DatetimeIndexImp attribute), 23

squatyp (IndexImplementer attribute), 23

squatyp (IntIndexImp attribute), 24

squatyp (PeriodIndexImp attribute), 23

squatyp (StrIndexImp attribute), 24

SQLFT (class in trump.templating), 37

StLouisFEDFT (class in trump.templating), 38

StrIndexImp (class in trump.indexing), 24

Symbol (class in trump.orm), 20

SymbolDataDef (class in trump.orm), 22

SymbolHandle (class in trump.orm), 27

SymbolManager (class in trump.orm), 13

SymbolReport (class in trump.reporting.objects), 26

SymbolTag (class in trump.orm), 22

SymbolValidity (class in trump.orm), 24

symname (FailSafe attribute), 26

symname (Override attribute), 25

T

to_json() (Symbol method), 22

trump.indexing (module), 22

trump.templating.bases (module), 35

TrumpReport (class in trump.reporting.objects), 27

U

update_handle() (Symbol method), 22

user (FailSafe attribute), 26

user (Override attribute), 25

V

val (FailSafe attribute), 26

val (Override attribute), 25

Y

YahooFinanceFT (class in trump.templating), 37