

---

# **Data Traffic Control**

**Laura Kinkead**

**Aug 31, 2021**



**CONTENTS:**

- 1 What is it? 3**
- 2 Main Features 5**
- 3 Installation 7**
  - 3.1 Getting Started . . . . . 7
  - 3.2 Supported File Formats . . . . . 13
  - 3.3 API Reference . . . . . 14
- Index 19**



*Whrrrr... Vooooosh... That's the sound of your data coming and going exactly where it belongs.*



## WHAT IS IT?

data traffic control is a python package that automates every-day interactions with your data files. Data files are the building materials we work with every day, all day. Working with them should be effortless!





## MAIN FEATURES

- Navigate your data directories with ease, without having to memorize long file paths.
- Load data files without having to think about it (or looking it up the syntax yet again).
- Be certain of how a datafile was generated, and rest easy knowing you could reproduce it.



## INSTALLATION

```
>>> pip install datatc
```

### 3.1 Getting Started

#### 3.1.1 Register a project

*datatc* will remember for you where the data for each of your projects is stored, so that you don't have to. The first time you use *datatc* with a project, register it with `DataDirectory`. You only have to do this once- *datatc* saves the information in `~/ .data_map.yaml` so you never have to memorize that long file path again.

```
>>> DataDirectory.register_project('project_name', '/path/to/project/data/dir/')
```

Example:

```
>>> DataDirectory.register_project('mridle', '/home/user/data/mridle/data')
```

Once you've registered a project, you can establish a `DataDirectory` on the fly by referencing it's name.

```
>>> dd = DataDirectory.load('mridle')
```

The `DataDirectory` object, `dd`, will be your gateway to all file discovery, load, and save operations.

---

#### 3.1.2 Explore the data directory

`DataDirectory` makes it easier to interact with your project's data directory. With `ls()`, you can print out the file structure:

```
>>> dd.ls()
raw/
  EntityViews/
    7 mixed items
  data_extracts/
    2019-11-23_Extract_3days/
      1 mixed items
    2020-01-05_Extract_3days_withHistory_v2/
      2 mixed items
```

(continues on next page)

(continued from previous page)

```
2020-02-04_Extract_3months/  
  3 mixed items  
db-connection.R  
query.sql
```

You can also print the contents of a subdirectory by navigating to it via the `[]` operators:

```
>>> dd['data_extracts'].ls(full=True)  
data_extracts/  
  2019-11-23_Extract_3days/  
    2019-11-23_Extract_3days.xlsx  
  2020-01-05_Extract_3days_withHistory_v2/  
    2020-01-05_Extract_3days_withHistory_v2.xlsx  
    2020-01-05_Extract_3days_withHistory_v2_sql.sql  
  2020-02-04_Extract_3months/  
    2020-02-04_Extract_3months.sql  
    2020-02-04_Extract_3months.xlsx  
    3_month_export.csv
```

### 3.1.3 Loading data files

To load a file, navigate the file system using `[]` operators, and then call `.load()`.

```
>>> raw_df = dd['data_extracts']['2020-02-04_Extract_3months']['2020-02-04_Extract_  
↪3months.xlsx'].load()
```

Don't worry about what format the file is in- *datatec* will intuit how to load the file. See *Supported Formats*.

#### Shortcuts for loading data files *faster*

##### latest

If you use timestamps to version your data files, you can load the latest file or subdirectory within a directory with `.latest()`:

```
>>> raw_df = dd['data_extracts'].latest()['2020-02-04_Extract_3months.xlsx'].load()
```

##### select

To help you navigate those long finicky file names, *DataDirectory* provides a `.select('hint')` method to search for files matching a substring.

```
>>> raw_df = dd['data_extracts']['2020-02-04_Extract_3months'].select('xlsx').load()
```

Combing *latest* and *select*, the file load in the previous can be reduced to the following:

```
>>> raw_df = dd['data_extracts'].latest().select('xlsx').load()
```

## Loading irregular data files

### ... my file needs special arguments to load

If your file needs special parameters to load it, specify them in `load`, and they will be passed on to the internal loading function. For example, if your csv file is actually pipe separated and has a non-default encoding, you can specify so:

```
>>> raw_df = dd['queries']['batch_query.csv'].load(sep='|', encoding='utf-16')
```

### ... my file type isn't recognized by *datatc*

If *datatc* doesn't recognize the file type, you can give it a type hint of which loader to use. For example, *datatc* doesn't have a specific interface for reading tab separated files, but if you tell it to treat it as a csv and instruct it to use tab as the separator, it will load it right up:

```
>>> raw_df = dd['queries']['batch_query.tsv'].load(data_interface_hint='csv', sep='\t')
```

### ... I want to load my file my own way

If you ever want to do your own load, and not use the build in `.load()`, you can also use `dd[...]['filename'].path` to get the path to the file for use in a separate loading operation.

---

## 3.1.4 Saving data files

To save a file, navigate with `dd` to the position in the file system where you'd like to save your file using the `[]` operators, and then call `.save(data_object, file_name)`.

For example:

```
dd['processed_data'].save(processed_df, 'processed.csv')
```

---

## 3.1.5 Working with *SelfAwareData*

*SelfAwareData* helps you remember how your datasets were generated.

You have 2 options for turning your dataset into a *SelfAwareData*:

1. Load from a file:

```
>>> my_sad = SelfAwareData.load_from_file('~/.path/to/data.csv')
```

When you establish a *SelfAwareData* from a file, it will track the file that the *SelfAwareData* originated from.

2. Create on the fly from a live data object:

```
>>> my_sad = SelfAwareData(raw_df)
```

Starting a *SelfAwareData* this way will not track how the data originated.

Your data is now accessible via `my_sad.data`.

### Transform

When you apply a transform to your dataset, use the built-in `transform` method to track the transform.

```
>>> new_sad = my_sad.transform(transform_func)
```

If you need to specify arguments to your `transform_func`, do so as keyword arguments in the `transform` function:

```
>>> new_sad = my_sad.transform(transform_func, num_bins=12)
```

---

**Note:** Note on Tracking Git Metadata: To ensure traceability, `SelfAwareData` checks that there are no uncommitted changes in the repo before running the transform. If there are uncommitted changes, `datatc` raises a `RuntimeError`. If you would like to override this check, specify `enforce_clean_git = False` in `transform()`.

---

---

**Note:** If the `transform_func` you pass to `transform()` is written in a file in a git repo, then `datatc` will include the git hash of the repo where `transform_func` is written. If the `transform_func` is not in a file (for example, is written on the fly in a notebook or in an interactive session), the user may specify the module to get a git hash from via `get_git_hash_from=module`.

---

### *SelfAwareData* objects automatically track their own metadata

By using the `SelfAwareData.transform` method, metadata about the transformation is automatically tracked, including:

- the timestamp of when the transformation was run
- the git hash of the repo where `transform_func` is located
- the code of the transform used to transform the data
- the arguments to the transform function

To access metadata, you can print the transform steps:

```
>>> new_sad.print_steps()
```

```
-----  
Step 0                2021-01-27 21:46          #f98fc21  
-----
```

```
def transform_step_1(input_df):  
    df = input_df.copy()  
    df['col_1'] = df['col_1'] * -1  
    return df  
-----
```

```
Step 1                2021-01-27 21:46          #f98fc21  
-----
```

```
def transform_step_2(input_df):  
    df = input_df.copy()  
    df['col_2'] = df['col_2']**2  
    return df
```

To access the data programmatically, use `SelfAwareData.get_info()`:

```

>>> new_sad.get_info()
[
  {
    'timestamp': '2021-01-27_21-46-52',
    'tag': '',
    'code': "def transform_step_1(input_df):\n    df = input_df.copy()\n    df['col_1
↪'] = df['col_1'] * -1\n    return df\n",
    'kwargs': {},
    'git_hash': 'f98fc21'
  },
  {
    'timestamp': '2021-01-27_21-46-55',
    'tag': '',
    'code': "def transform_step_2(input_df):\n    df = input_df.copy()\n    df['col_2
↪'] = df['col_2']**2\n    return df\n",
    'kwargs': {},
    'git_hash': 'f98fc21'
  }
]

```

## Save

There are 2 ways to save *SelfAwareData* objects.

1. If you are using *DataDirectory*, then saving your *SelfAwareData* works the same as saving any other file with *DataDirectory*.

```

>>> dd['directory'].save(sad, output_file_name)

```

2. You can also save *SelfAwareData*, independently, without using *DataDirectory*.

```

>>> sad.save(output_file_path)

```

## Load

Loading *SelfAwareData* works the same as loading any other data file with *DataDirectory*.

```

>>> sad = dd['feature_sets']['my_feature_set.csv'].load()

```

This load returns you a *SelfAwareData* object. This object contains not only the data you transformed and saved, but also the transformation function itself.

To access the data:

```

>>> sad.data

```

To view the code of the data's transformation function:

```

>>> sad.print_steps()

```

To rerun the same transformation function on a new data object:

```
>>> sad.rerun(new_df)
```

### Loading *SelfAwareData* objects without *DataDirectory*

You can also load *SelfAwareData* objects without going through *DataDirectory*:

```
>>> sad = SelfAwareData.load(file_path)
```

However, *SelfAwareData* objects are saved to the file system as directories with long names, like `sad_dir__2021-01-01_12-00__transform_1`. When you interact with *SelfAwareData* via *DataDirectory*, you can reference them like normal files (`transform_1.csv`), however, referencing them outside of *DataDirectory* is not as easy.

### Loading *SelfAwareData* objects in dependency-incomplete environments

If the *SelfAwareData* object is moved to a different environment where the dependencies for the code transform are not met, use

```
>>> sad = SelfAwareDataDirectory.load(load_function=False)
```

to avoid a `ModuleNotFoundError`.

### *SelfAwareData* Example

Here's a toy example of working with *SelfAwareData*:

```
from datatc import DataDirectory, SelfAwareData

dd = DataDirectory.load('datatc_demo')

raw_sad = SelfAwareData.load_from_file(dd['raw']['iris.csv'].path)

def petal_area(df):
    df['petal_area'] = df['petal_length'] * df['petal_width']
    return df

area_sad = raw_sad.transform(petal_area)

dd['processed'].save(area_sad, 'area.csv')
```

### View your *SelfAwareData* tree with the *datatc* web app!

You can view the tree of how your *SelfAwareData* files are related in the *datatc* web app. To start the web app, run the `datatc_app` command in your shell and provide the name of the project whose data directory you'd like to view.

```
>>> datatc_app <registered_project>
```

This project must have been previously registered to *datatc* with `DataDirectory.register_project()`. If you need a reminder of what projects you've registered with *datatc*, run `datatc_list`.



---

**Note:** To use the web app, install `datatc` with the extra app dependencies: `pip install datatc[app]`. If you're using `zsh`, you'll need to escape the square brackets with `pip install datatc\[app\]`

---

### 3.1.6 Working with File Types via *MagicDataInterface*

`MagicDataInterface` provides a one-stop shop for interacting with common file types. Just point `MagicDataInterface` to a file path and call `save()` and `load()`.

```
from datatc import MagicDataInterface

iris_df = MagicDataInterface.load('iris.csv')
config = MagicDataInterface.load('config/model_params.yaml')

MagicDataInterface.save(results, 'results/iris_results.pkl')
```

See *Supported Formats* for a list of data types that `MagicDataInterface` knows how to work with.

If you want to work with a file type that `MagicDataInterface` doesn't know about yet, you can create a *DataInterface* for it:

1. Create a `DataInterface` that subclasses from `DataInterfaceBase`, and implement the `_interface_specific_save` and `_interface_specific_load` functions.
2. Register your new *DataInterface* with `MagicDataInterface`:

```
>>> MagicDataInterface.register_data_interface(MyNewDataInterface)
```

## 3.2 Supported File Formats

`datatc` supports reading and writing the following file formats.

*Note: Most formats work out of the box, but a few require additional installations- these are noted with an asterisk and explained below.*

- CSV
- Dill
- Excel
- Parquet\*
- PDF\*
- Pickle
- Text
- YAML

### 3.2.1 Formats that Require Additional Installation

#### Parquet

To work with Parquet files, you must also install either `pyarrow` or `fastparquet`.

#### PDF

To work with PDF files, you must also install the `fitz` package.

## 3.3 API Reference

### 3.3.1 DataDirectory

```
class datatc.data_directory.DataDirectory(path, contents=None,  
                                           magic_data_interface=<datatc.data_interface.MagicDataInterfaceBase  
                                           object>)
```

Manages saving, loading, and viewing data files within a specific data path.

**latest()**

Return the latest data file or directory, as determined alphabetically.

**Return type** Union[*DataDirectory*, *DataFile*]

**classmethod list\_projects()**

List all data directories previously registered via *register\_project*.

**Return type** None

**classmethod load(*hint*)**

Shortcut for *load\_project*.

**classmethod load\_project(*hint*)**

Create a *DataDirectory* from a project hint previously registered via *register\_project*.

**ls(*full=False*)**

Print the contents of the data directory. Defaults to printing all subdirectories, but not all files.

**Parameters** *full* – Whether to print all files.

**Return type** None

**mkdir(*dir\_name*)**

Create a new directory within the current directory. :type *dir\_name*: str :param *dir\_name*: Name for the new directory

Returns: None

**classmethod register\_project(*project\_hint*, *project\_path*)**

Register a hint for a project data directory so that it can be easily reloaded via *load(*hint*)*.

**Return type** None

**save(*data*, *file\_name*, *\*\*kwargs*)**

Save a data object within the data directory.

**Parameters**

- **data** (Any) – data object to save.

- **file\_name** (str) – file name for the saved object, including file extension. The file extension is used to determine the file type and method for saving the data.
- **\*\*kwargs** – Remaining args are passed to the data interface save function.

**Return type** None

**select**(*hint*)

Return the `DataDirectory` from `self.contents` that matches the `hint`. If more than one file matches the `hint`, then select the one that file whose type matches the `hint` exactly. Otherwise raise an error and display all matches.

**Parameters** `hint` (str) – string to use to search for a file within the directory.

**Raises**

- **FileNotFoundError** – if no file can be found in the data directory that matches the `hint`.
- **ValueError** – if more than one file is found in the data directory that matches the `hint`.

**Return type** Union[`DataDirectory`, `DataFile`]

**class** `datatc.data_directory.SelfAwareDataDirectory`(*path*, *contents=None*)

Subclass of `DataDirectory` that manages interacting with the file expression of `SelfAwareData`.

**get\_info**()

Get metadata about the `SelfAwareData` object.

**Return type** Dict[str, str]

**load**(*data\_interface\_hint=None*, *load\_function=True*, *\*\*kwargs*)

Load a saved data transformer- the data and the function that generated it.

**Parameters**

- **data\_interface\_hint** (Optional[str]) – file extension indicating the data interface to use to load the file.
- **load\_function** (bool) – Whether to load the transformation function of the `SelfAwareData` object. Specify `False` if the current environment does not support the dependencies of the transformation function.
- **\*\*kwargs** – Remaining args are passed to the data interface save function.

**Return type** `SelfAwareData`

### 3.3.2 DataFile

**class** `datatc.data_directory.DataFile`(*path*, *contents=None*)

**load**(*data\_interface\_hint=None*, *\*\*kwargs*)

Load a data file.

**Parameters**

- **data\_interface\_hint** – file extension indicating the data interface to use to load the file.
- **\*\*kwargs** – Remaining args are passed to the data interface save function.

**Return type** Any

### 3.3.3 SelfAwareData

**class** `datatc.self_aware_data.SelfAwareData`(*data*, *metadata=None*)

A wrapper around a dataset that also contains the code that generated the data. *SelfAwareData* can re-run it's transformation steps on a new dataset.

**classmethod** `load`(*file\_path*, *data\_interface\_hint=None*, *\*\*kwargs*)

Load a *SelfAwareData* object.

**Parameters**

- **file\_path** (str) – Path to the SAD to load.
- **data\_interface\_hint** – Hint for which kind of data interface to use to load the data (file extension).

**Example Usage:**

```
>>> sad = SelfAwareData.load('~/project/data/sad_dir__2021-01-01_12-00__
↳standard_features.csv')
```

**Return type** *SelfAwareData*

**classmethod** `load_from_file`(*file\_path*, *\*\*kwargs*)

Create a *SelfAwareData* object with a initial *SourceFileTransformStep*

**Parameters** **file\_path** (str) – path to a standard file (not already a *SelfAwareData*)

Returns: *SelfAwareData* with a *TransformSequence* containing a *SourceFileTransformStep* pointing to *file\_path*

**Return type** *SelfAwareData*

**print\_steps**()

Print the code of the transformation steps that generated the data.

**rerun**(*data*)

Rerun the same transformation function that generated this *SelfAwareData* on a new data object.

**Parameters** **data** –

Returns:

**Return type** Any

**save**(*file\_path*, *\*\*kwargs*)

Save a *SelfAwareData* object.

**Parameters** **file\_path** (str) – Path for where to save the SAD, including file extension.

Returns: Path to the saved SAD

**Return type** Path

**transform**(*transformer\_func*, *tag=""*, *enforce\_clean\_git=True*, *get\_git\_hash\_from=None*, *\*\*kwargs*)

Transform a *SelfAwareData*, generating a new *SelfAwareData* object.

**Parameters**

- **transformer\_func** (Callable) – Transform function to apply to data.
- **tag** (str) – (optional) short description of the transform for reference

- **enforce\_clean\_git** – Whether to only allow the save to proceed if the working state of the git directory is clean.
- **get\_git\_hash\_from** (Optional[Any]) – Locally installed module from which to get git information. Use this arg if transform\_func is defined outside of a module tracked by git.

Returns: new transform directory name, for adding to contents dict.

**Return type** *SelfAwareData*

### 3.3.4 MagicDataInterface

`datatc.data_interface.MagicDataInterface`

alias of <datatc.data\_interface.MagicDataInterfaceBase object at 0x7fec18301d10>



## INDEX

### D

DataDirectory (class in `datatc.data_directory`), 14

DataFile (class in `datatc.data_directory`), 15

### G

get\_info() (`datatc.data_directory.SelfAwareDataDirectory` method), 15

### L

latest() (`datatc.data_directory.DataDirectory` method), 14

list\_projects() (`datatc.data_directory.DataDirectory` class method), 14

load() (`datatc.data_directory.DataDirectory` class method), 14

load() (`datatc.data_directory.DataFile` method), 15

load() (`datatc.data_directory.SelfAwareDataDirectory` method), 15

load() (`datatc.self_aware_data.SelfAwareData` class method), 16

load\_from\_file() (`datatc.self_aware_data.SelfAwareData` class method), 16

load\_project() (`datatc.data_directory.DataDirectory` class method), 14

ls() (`datatc.data_directory.DataDirectory` method), 14

### M

MagicDataInterface (in `datatc.data_interface` module), 17

mkdir() (`datatc.data_directory.DataDirectory` method), 14

### P

print\_steps() (`datatc.self_aware_data.SelfAwareData` method), 16

### R

register\_project() (`datatc.data_directory.DataDirectory` class method), 14

rerun() (`datatc.self_aware_data.SelfAwareData` method), 16

### S

save() (`datatc.data_directory.DataDirectory` method), 14

save() (`datatc.self_aware_data.SelfAwareData` method), 16

select() (`datatc.data_directory.DataDirectory` method), 15

SelfAwareData (class in `datatc.self_aware_data`), 16

SelfAwareDataDirectory (class in `datatc.data_directory`), 15

### T

transform() (`datatc.self_aware_data.SelfAwareData` method), 16