
Woodwork Documentation

Release 0.0.8

Alteryx, Inc.

Jan 25, 2021

CONTENTS

1 Quick Start	3
Index	87

It provides a `DataTable` object, which contains the physical, logical, and semantic data types. It can be used with [Featuretools](#), [EvalML](#), and general machine learning applications where logical and semantic typing information is important.

Woodwork provides simple interfaces for adding and updating logical and semantic typing information, as well as selecting data columns based on the types.

QUICK START

Below is an example of using a Woodwork DataTable to automatically infer the Logical Types for a data structure and select columns with specific types.

```
[1]: import woodwork as ww

data = ww.demo.load_retail(nrows=100, return_dataframe=True)

dt = ww.DataTable(data, name="retail")
dt
```

```
[1]:
```

Data Column	Physical Type	Logical Type	Semantic Tag(s)
order_product_id	Int64	Integer	['numeric']
order_id	Int64	Integer	['numeric']
product_id	category	Categorical	['category']
description	string	NaturalLanguage	[]
quantity	Int64	Integer	['numeric']
order_date	datetime64[ns]	Datetime	[]
unit_price	float64	Double	['numeric']
customer_name	string	NaturalLanguage	[]
country	string	NaturalLanguage	[]
total	float64	Double	['numeric']
cancelled	boolean	Boolean	[]

```
[2]: filtered_dt = dt.select(include=['numeric', 'Boolean'])
filtered_dt.to_dataframe().head(5)
```

```
[2]:
```

	order_product_id	order_id	quantity	unit_price	total	cancelled
0	0	536365	6	4.2075	25.245	False
1	1	536365	6	5.5935	33.561	False
2	2	536365	8	4.5375	36.300	False
3	3	536365	6	5.5935	33.561	False
4	4	536365	6	5.5935	33.561	False

1.1 Table of contents

1.1.1 Install

Woodwork is available for Python 3.6, 3.7, and 3.8. It can be installed from PyPI, conda, or from source.

PyPI

To install Woodwork from PyPI, run the following command:

```
python -m pip install woodwork
```

Woodwork allows users to install add-ons individually or all at once. In order to install all add-ons, run:

```
python -m pip install "woodwork[complete]"
```

You can use Woodwork to create Dask DataTables by running:

```
python -m pip install "woodwork[dask]"
```

You can use Woodwork to create Koalas DataTables by running:

```
python -m pip install "woodwork[koalas]"
```

Conda

To install Woodwork from conda run the following command:

```
conda install -c conda-forge woodwork
```

Note: In order to create Dask or Koalas DataTables, the following commands must be run for your library of choice prior to installing Woodwork with conda: `conda install dask` for Dask or `conda install koalas` and `conda install pyspark` for Koalas.

Source

To install Woodwork from source, clone the repository from [Github](https://github.com/alteryx/woodwork), and install the dependencies.

```
git clone https://github.com/alteryx/woodwork.git
cd woodwork
python -m pip install .
```

Dependencies

You can view a list of all Woodwork core dependencies in the `requirements.txt` file.

Optional Dependencies

Woodwork has several other dependencies that are used only for specific methods. Attempting to use one of these methods without having the necessary library installed will result in an `ImportError` with instructions on how to install the necessary dependency.

Dependency	Min Version	Notes
boto3	1.10.45	Required to read/write to URLs and S3
smart_open	1.8.4	Required to read/write to URLs and S3
pyarrow	2.0.0	Required to serialize to parquet
dask[distributed]	2.30.0	Required to use with Dask DataFrames
koalas	1.3.0	Required to use with Koalas DataFrames
pyspark	3.0.0	Required to use with Koalas DataFrames

1.1.2 Development

To make contributions to the codebase, please follow the guidelines [here](#).

1.1.3 Start

In this guide, we will walk through an example of creating a Woodwork `DataTable`, and will show how to update and remove logical types and semantic tags. We will also demonstrate how to use the typing information to select subsets of data.

Types and Tags

Woodwork relies heavily on the concepts of physical types, logical types and semantic tags. These concepts are covered in detail in *Understanding Types and Tags*, but brief definitions of each are provided here for reference:

- Physical Type: defines how the data is stored on disk or in memory
- Logical Type: defines how the data should be parsed or interpreted
- Semantic Tag(s): provides additional data about the meaning of the data or how it should be used

Let's demonstrate how to use Woodwork, starting off by creating a dataframe containing retail sales data.

```
[1]: import woodwork as ww
```

```
data = ww.demo.load_retail(nrows=100, return_dataframe=True)
data.head(5)
```

```
[1]:  order_product_id  order_id  product_id  description \
0         0         536365    85123A  WHITE HANGING HEART T-LIGHT HOLDER
1         1         536365    71053    WHITE METAL LANTERN
2         2         536365    84406B    CREAM CUPID HEARTS COAT HANGER
3         3         536365    84029G  KNITTED UNION FLAG HOT WATER BOTTLE
4         4         536365    84029E    RED WOOLLY HOTTIE WHITE HEART.
```

(continues on next page)

(continued from previous page)

```

    quantity      order_date  unit_price customer_name      country \
0         6 2010-12-01 08:26:00    4.2075  Andrea Brown  United Kingdom
1         6 2010-12-01 08:26:00    5.5935  Andrea Brown  United Kingdom
2         8 2010-12-01 08:26:00    4.5375  Andrea Brown  United Kingdom
3         6 2010-12-01 08:26:00    5.5935  Andrea Brown  United Kingdom
4         6 2010-12-01 08:26:00    5.5935  Andrea Brown  United Kingdom

    total  cancelled
0  25.245      False
1  33.561      False
2  36.300      False
3  33.561      False
4  33.561      False

```

As we can see, this is a dataframe containing several different data types, including dates, categorical values, numeric values and natural language descriptions. Let's use Woodwork to create a DataTable from this data.

Creating a DataTable

Creating a Woodwork DataTable is as simple as passing in a dataframe with the data of interest during initialization. An optional name parameter can be specified to label the DataTable.

```
[2]: dt = ww.DataTable(data, name="retail")
dt
```

```
[2]:
```

Data Column	Physical Type	Logical Type	Semantic Tag(s)
order_product_id	Int64	Integer	['numeric']
order_id	Int64	Integer	['numeric']
product_id	category	Categorical	['category']
description	string	NaturalLanguage	[]
quantity	Int64	Integer	['numeric']
order_date	datetime64[ns]	Datetime	[]
unit_price	float64	Double	['numeric']
customer_name	string	NaturalLanguage	[]
country	string	NaturalLanguage	[]
total	float64	Double	['numeric']
cancelled	boolean	Boolean	[]

Using just this simple call, Woodwork was able to infer the logical types present in our data by analyzing the dataframe dtypes as well as the information contained in the columns. In addition, Woodwork also added semantic tags to some of the columns based on the logical types that were inferred.

We can also view the typing information with the first few columns of data with the following:

```
[3]: dt.head()
```

```
[3]:
```

Data Column	order_product_id	order_id	product_id	\
Physical Type	Int64	Int64	category	
Logical Type	Integer	Integer	Categorical	
Semantic Tag(s)	['numeric']	['numeric']	['category']	
0	0	536365	85123A	
1	1	536365	71053	
2	2	536365	84406B	
3	3	536365	84029G	

(continues on next page)

(continued from previous page)

4	4	536365	84029E	
Data Column		description	quantity \	
Physical Type		string	Int64	
Logical Type		NaturalLanguage	Integer	
Semantic Tag(s)		[]	['numeric']	
0		WHITE HANGING HEART T-LIGHT HOLDER	6	
1		WHITE METAL LANTERN	6	
2		CREAM CUPID HEARTS COAT HANGER	8	
3		KNITTED UNION FLAG HOT WATER BOTTLE	6	
4		RED WOOLLY HOTTIE WHITE HEART.	6	
Data Column		order_date	unit_price	customer_name \
Physical Type		datetime64[ns]	float64	string
Logical Type		Datetime	Double	NaturalLanguage
Semantic Tag(s)		[]	['numeric']	[]
0		2010-12-01 08:26:00	4.2075	Andrea Brown
1		2010-12-01 08:26:00	5.5935	Andrea Brown
2		2010-12-01 08:26:00	4.5375	Andrea Brown
3		2010-12-01 08:26:00	5.5935	Andrea Brown
4		2010-12-01 08:26:00	5.5935	Andrea Brown
Data Column		country	total	cancelled
Physical Type		string	float64	boolean
Logical Type		NaturalLanguage	Double	Boolean
Semantic Tag(s)		[]	['numeric']	[]
0		United Kingdom	25.245	False
1		United Kingdom	33.561	False
2		United Kingdom	36.300	False
3		United Kingdom	33.561	False
4		United Kingdom	33.561	False

Updating Logical Types

If the initial inference was not to our liking, the logical type can be changed to a more appropriate value. Let's change some of the columns to a different logical type to illustrate this process. Below we will set the logical type for the quantity, customer_name and country columns to be Categorical.

```
[4]: dt = dt.set_types(logical_types={
      'quantity': 'Categorical',
      'customer_name': 'Categorical',
      'country': 'Categorical'
    })
dt
```

```
[4]:
```

	Physical Type	Logical Type	Semantic Tag(s)
Data Column			
order_product_id	Int64	Integer	['numeric']
order_id	Int64	Integer	['numeric']
product_id	category	Categorical	['category']
description	string	NaturalLanguage	[]
quantity	category	Categorical	['category']
order_date	datetime64[ns]	Datetime	[]
unit_price	float64	Double	['numeric']
customer_name	category	Categorical	['category']

(continues on next page)

(continued from previous page)

country	category	Categorical	['category']
total	float64	Double	['numeric']
cancelled	boolean	Boolean	[]

If we now inspect the information in the `types` output, we can see that the Logical type for the three columns has been updated with the `Categorical` logical type we specified.

Selecting Columns

Now that we have logical types we are happy with, we can select a subset of the columns based on their logical types. Let's select only the columns that have a logical type of `Integer` or `Double`:

```
[5]: numeric_dt = dt.select(['Integer', 'Double'])
numeric_dt
```

```
[5]:
```

Data Column	Physical Type	Logical Type	Semantic Tag(s)
order_product_id	Int64	Integer	['numeric']
order_id	Int64	Integer	['numeric']
unit_price	float64	Double	['numeric']
total	float64	Double	['numeric']

This selection process has returned a new `DataTable` containing only the columns that match the logical types we specified. After we have selected the columns we want, we can also access a dataframe containing just those columns if we need it for additional analysis.

```
[6]: numeric_dt.to_dataframe()
```

```
[6]:
```

	order_product_id	order_id	unit_price	total
0	0	536365	4.2075	25.245
1	1	536365	5.5935	33.561
2	2	536365	4.5375	36.300
3	3	536365	5.5935	33.561
4	4	536365	5.5935	33.561
..
95	95	536378	4.2075	25.245
96	96	536378	0.6930	83.160
97	97	536378	0.9075	21.780
98	98	536378	0.9075	21.780
99	99	536378	0.9075	21.780

[100 rows x 4 columns]

Note: Accessing the dataframe associated with a `DataTable` by using `dt.to_dataframe()` will return a reference to the dataframe. Modifications to the returned dataframe can cause unexpected results. If you need to modify the dataframe, you should use `dt.to_dataframe().copy()` to return a copy of the stored dataframe that can be safely modified without impacting the `DataTable` behavior.

Adding Semantic Tags

Next, let's add semantic tags to some of the columns. We will add the tag of `product_details` to the `description` column and tag the `total` column with `currency`.

```
[7]: dt = dt.set_types(semantic_tags={'description':'product_details', 'total': 'currency'})
      dt
```

[7]:	Data Column	Physical Type	Logical Type	Semantic Tag(s)
	order_product_id	Int64	Integer	['numeric']
	order_id	Int64	Integer	['numeric']
	product_id	category	Categorical	['category']
	description	string	NaturalLanguage	['product_details']
	quantity	category	Categorical	['category']
	order_date	datetime64[ns]	Datetime	[]
	unit_price	float64	Double	['numeric']
	customer_name	category	Categorical	['category']
	country	category	Categorical	['category']
	total	float64	Double	['currency', 'numeric']
	cancelled	boolean	Boolean	[]

We can also select columns based on a semantic tag. Perhaps we want to only select the columns tagged with `category`:

```
[8]: category_dt = dt.select('category')
      category_dt
```

[8]:	Data Column	Physical Type	Logical Type	Semantic Tag(s)
	product_id	category	Categorical	['category']
	quantity	category	Categorical	['category']
	customer_name	category	Categorical	['category']
	country	category	Categorical	['category']

We can also select columns using multiple semantic tags, or even a mixture of semantic tags and logical types:

```
[9]: category_numeric_dt = dt.select(['numeric', 'category'])
      category_numeric_dt
```

[9]:	Data Column	Physical Type	Logical Type	Semantic Tag(s)
	order_product_id	Int64	Integer	['numeric']
	order_id	Int64	Integer	['numeric']
	product_id	category	Categorical	['category']
	quantity	category	Categorical	['category']
	unit_price	float64	Double	['numeric']
	customer_name	category	Categorical	['category']
	country	category	Categorical	['category']
	total	float64	Double	['currency', 'numeric']

```
[10]: mixed_dt = dt.select(['Boolean', 'product_details'])
       mixed_dt
```

[10]:	Data Column	Physical Type	Logical Type	Semantic Tag(s)
	description	string	NaturalLanguage	['product_details']
	cancelled	boolean	Boolean	[]

If we wanted to select an individual column, we just need to specify the column name. We can then get access to the data in the DataColumn using the `to_series` method:

```
[11]: dc = dt['total']
      dc
[11]: <DataColumn: total (Physical Type = float64) (Logical Type = Double) (Semantic Tags =
      ↳{'currency', 'numeric'})>
```

```
[12]: dc.to_series()
[12]: 0      25.245
      1      33.561
      2      36.300
      3      33.561
      4      33.561
      ...
      95     25.245
      96     83.160
      97     21.780
      98     21.780
      99     21.780
      Name: total, Length: 100, dtype: float64
```

You can also access multiple columns by supplying a list of column names:

```
[13]: multiple_cols_dt = dt[['product_id', 'total', 'unit_price']]
      multiple_cols_dt
[13]:
```

Data Column	Physical Type	Logical Type	Semantic Tag(s)
product_id	category	Categorical	['category']
total	float64	Double	['currency', 'numeric']
unit_price	float64	Double	['numeric']

Removing Semantic Tags

We can also remove specific semantic tags from a column if they are no longer needed. Let's remove the `product_details` tag from the `description` column:

```
[14]: dt = dt.remove_semantic_tags({'description': 'product_details'})
      dt
[14]:
```

Data Column	Physical Type	Logical Type	Semantic Tag(s)
order_product_id	Int64	Integer	['numeric']
order_id	Int64	Integer	['numeric']
product_id	category	Categorical	['category']
description	string	NaturalLanguage	[]
quantity	category	Categorical	['category']
order_date	datetime64[ns]	Datetime	[]
unit_price	float64	Double	['numeric']
customer_name	category	Categorical	['category']
country	category	Categorical	['category']
total	float64	Double	['currency', 'numeric']
cancelled	boolean	Boolean	[]

Notice how the `product_details` tag has now been removed from the `description` column. If we wanted to remove all user-added semantic tags from all columns, we can also do that:

```
[15]: dt = dt.reset_semantic_tags()
dt
```

```
[15]:
```

Data Column	Physical Type	Logical Type	Semantic Tag(s)
order_product_id	Int64	Integer	['numeric']
order_id	Int64	Integer	['numeric']
product_id	category	Categorical	['category']
description	string	NaturalLanguage	[]
quantity	category	Categorical	['category']
order_date	datetime64[ns]	Datetime	[]
unit_price	float64	Double	['numeric']
customer_name	category	Categorical	['category']
country	category	Categorical	['category']
total	float64	Double	['numeric']
cancelled	boolean	Boolean	[]

Set Index and Time Index

At any point, we can designate certain columns as the DataTable's index and with the methods `set_index` and `set_time_index`. These methods can be used to assign these columns for the first time or to change the column being used as the index or time index.

Index and time index columns contain `index` and `time_index` semantic tags, respectively.

```
[16]: dt = dt.set_index('order_product_id')
dt.index
```

```
[16]: 'order_product_id'
```

```
[17]: dt = dt.set_time_index('order_date')
dt.time_index
```

```
[17]: 'order_date'
```

```
[18]: dt
```

```
[18]:
```

Data Column	Physical Type	Logical Type	Semantic Tag(s)
order_product_id	Int64	Integer	['index']
order_id	Int64	Integer	['numeric']
product_id	category	Categorical	['category']
description	string	NaturalLanguage	[]
quantity	category	Categorical	['category']
order_date	datetime64[ns]	Datetime	['time_index']
unit_price	float64	Double	['numeric']
customer_name	category	Categorical	['category']
country	category	Categorical	['category']
total	float64	Double	['numeric']
cancelled	boolean	Boolean	[]

List Logical Types

We can also retrieve all the Logical Types present in Woodwork. These can be useful for understanding the Logical Types, and how they will be interpreted.

```
[19]: from woodwork.type_sys.utils import list_logical_types
```

```
list_logical_types()
```

```
[19]:
```

	name	type_string	\
0	Boolean	boolean	
1	Categorical	categorical	
2	CountryCode	country_code	
3	Datetime	datetime	
4	Double	double	
5	EmailAddress	email_address	
6	Filepath	filepath	
7	FullName	full_name	
8	IPAddress	ip_address	
9	Integer	integer	
10	LatLong	lat_long	
11	NaturalLanguage	natural_language	
12	Ordinal	ordinal	
13	PhoneNumber	phone_number	
14	SubRegionCode	sub_region_code	
15	Timedelta	timedelta	
16	URL	url	
17	ZIPCode	zip_code	

	description	physical_type	\
0	Represents Logical Types that contain binary v...	boolean	
1	Represents Logical Types that contain unordere...	category	
2	Represents Logical Types that contain categori...	category	
3	Represents Logical Types that contain date and...	datetime64[ns]	
4	Represents Logical Types that contain positive...	float64	
5	Represents Logical Types that contain email ad...	string	
6	Represents Logical Types that specify location...	string	
7	Represents Logical Types that may contain firs...	string	
8	Represents Logical Types that contain IP addre...	string	
9	Represents Logical Types that contain positive...	Int64	
10	Represents Logical Types that contain latitude...	object	
11	Represents Logical Types that contain text or ...	string	
12	Represents Logical Types that contain ordered ...	category	
13	Represents Logical Types that contain numeric ...	string	
14	Represents Logical Types that contain codes re...	category	
15	Represents Logical Types that contain values s...	timedelta64[ns]	
16	Represents Logical Types that contain URLs, wh...	string	
17	Represents Logical Types that contain a series...	category	

	standard_tags	is_default_type	is_registered	parent_type
0	{}	True	True	None
1	{category}	True	True	None
2	{category}	True	True	Categorical
3	{}	True	True	None
4	{numeric}	True	True	None
5	{}	True	True	NaturalLanguage
6	{}	True	True	NaturalLanguage
7	{}	True	True	NaturalLanguage

(continues on next page)

(continued from previous page)

8	{ }	True	True	NaturalLanguage
9	{numeric}	True	True	None
10	{ }	True	True	None
11	{ }	True	True	None
12	{category}	True	True	Categorical
13	{ }	True	True	NaturalLanguage
14	{category}	True	True	Categorical
15	{ }	True	True	None
16	{ }	True	True	NaturalLanguage
17	{category}	True	True	Categorical

1.1.4 Guides

The guides below provide more detail on the functionality of Woodwork.

Understanding Types and Tags

Using Woodwork effectively requires a good understanding of physical types, logical types and semantic tags, concepts which are core to Woodwork. This guide provides a detailed overview of types and tags as well as how to work with them when using Woodwork.

Definitions of Types and Tags

Woodwork has been designed to allow users to easily specify additional information about data contained in a `DataTable`, while providing interfaces to access the underlying data based on this additional information. Because a single `DataTable` may store various types of data such as numbers, text or dates in different columns, the additional information is defined on a per-column basis.

There are three main ways that Woodwork stores additional information about user data:

- **Physical Type:** defines how the data is stored on disk or in memory
- **Logical Type:** defines how the data should be parsed or interpreted
- **Semantic Tag(s):** provides additional data about the meaning of the data or how it should be used

Each of these are discussed in more detail throughout this guide.

Physical Types

Physical types define how the data is stored on disk or in memory. You may also see the physical type for a column referred to as the column's `dtype`.

For example, typical Pandas dtypes often used include `object`, `int64`, `float64` and `datetime64[ns]`, although there are many more. Within Woodwork, there are seven different physical types that are used, each corresponding to a Pandas dtype. When a `DataTable` is created, the dtype of the underlying data will be converted to one of these values, if it is not already one of these types:

- `boolean`
- `category`
- `datetime64[ns]`

- float64
- Int64
- string
- timedelta64[ns]

The physical type conversion is done based on the `LogicalType` that has been specified or inferred for a given column.

Logical Types

Logical types define how data should be interpreted or parsed. Logical types provide an additional level of detail beyond the physical type. Some columns might share the same physical type, but might have different parsing requirements depending on the information that is stored in the column.

For example, email addresses and phone numbers would typically both be stored in a data column with a physical type of `string`. However, when reading and validating these two types of information, very different rules apply. For email addresses, the presence of the `@` symbol is important. For phone numbers, you may want to confirm that only a certain number of digits are present and special characters might be restricted to `+`, `-`, `(` or `)`. In this particular example Woodwork defines two different logical types to separate these parsing needs: `EmailAddress` and `PhoneNumber`.

There are many different logical types defined within Woodwork. To get a complete list of all the available logical types, you can use the `list_logical_types` function as shown below.

```
[1]: from woodwork import list_logical_types
list_logical_types()
```

```
[1]:
```

	name	type_string	\
0	Boolean	boolean	
1	Categorical	categorical	
2	CountryCode	country_code	
3	Datetime	datetime	
4	Double	double	
5	EmailAddress	email_address	
6	Filepath	filepath	
7	FullName	full_name	
8	IPAddress	ip_address	
9	Integer	integer	
10	LatLong	lat_long	
11	NaturalLanguage	natural_language	
12	Ordinal	ordinal	
13	PhoneNumber	phone_number	
14	SubRegionCode	sub_region_code	
15	Timedelta	timedelta	
16	URL	url	
17	ZIPCode	zip_code	

	description	physical_type	\
0	Represents Logical Types that contain binary v...	boolean	
1	Represents Logical Types that contain unordere...	category	
2	Represents Logical Types that contain categori...	category	
3	Represents Logical Types that contain date and...	datetime64[ns]	
4	Represents Logical Types that contain positive...	float64	
5	Represents Logical Types that contain email ad...	string	
6	Represents Logical Types that specify location...	string	
7	Represents Logical Types that may contain firs...	string	

(continues on next page)

(continued from previous page)

8	Represents Logical Types that contain IP addre...				string
9	Represents Logical Types that contain positive...				Int64
10	Represents Logical Types that contain latitude...				object
11	Represents Logical Types that contain text or ...				string
12	Represents Logical Types that contain ordered ...				category
13	Represents Logical Types that contain numeric ...				string
14	Represents Logical Types that contain codes re...				category
15	Represents Logical Types that contain values s...				timedelta64[ns]
16	Represents Logical Types that contain URLs, wh...				string
17	Represents Logical Types that contain a series...				category
	standard_tags	is_default_type	is_registered		parent_type
0	{}	True	True		None
1	{category}	True	True		None
2	{category}	True	True		Categorical
3	{}	True	True		None
4	{numeric}	True	True		None
5	{}	True	True		NaturalLanguage
6	{}	True	True		NaturalLanguage
7	{}	True	True		NaturalLanguage
8	{}	True	True		NaturalLanguage
9	{numeric}	True	True		None
10	{}	True	True		None
11	{}	True	True		None
12	{category}	True	True		Categorical
13	{}	True	True		NaturalLanguage
14	{category}	True	True		Categorical
15	{}	True	True		None
16	{}	True	True		NaturalLanguage
17	{category}	True	True		Categorical

In the table of logical types shown above, you will notice that each logical type has a specific `pandas_dtype` value associated with it. Within Woodwork, any time a logical type is set for a column, the physical type of the underlying data will be converted to the type shown in the `pandas_dtype` column. Within Woodwork, there is only one physical type associated with each logical type.

Semantic Tags

Semantic tags define additional context about the meaning of a data column. This could directly impact how the information contained in the column is interpreted. Unlike physical types and logical types, semantic tags are much less restrictive. A column may contain many semantic tags, or none at all. However, when assigning semantic tags, users should take care to not assign tags that have conflicting meanings.

As an example of how semantic tags can be useful, let's consider a data set with two date columns: a signup date and a user birth date. Both of these columns will have the same physical type (`datetime64[ns]`) and both will have the same logical type (`Datetime`). However, semantic tags could be used to differentiate these columns. For example you might want to add the `date_of_birth` semantic tag to the user birth date column to indicate this column has special meaning and could be used to compute a user's age. Computing an age from the signup date column would not make sense, so the semantic tag can be used to differentiate between what the dates in these columns really mean.

Standard Semantic Tags

As you can see from the table that was generated with the `list_logical_types` function above, Woodwork has some standard tags that are applied to certain columns by default. Woodwork will add a standard set of semantic tags to columns with LogicalTypes that fall under certain, predefined categories.

The standard tags are as follows:

- 'numeric' - The tag applied to numeric Logical Types
 - Integer
 - Double
- 'category' - The tag applied to Logical Types that represent categorical variables
 - Categorical
 - CountryCode
 - Ordinal
 - SubRegionCode
 - ZIPCode

There are also two tags that get added to index columns. If no index columns have been specified, these tags are not present:

- 'index' - on the index column, when specified
- 'time_index' on the time index column, when specified

The application of standard tags, excluding the `index` and `time_index` tags, which have special meaning, can be controlled by the user. This will be discussed in more detail in the [Working with Semantic Tags](#) section below. There are a few different semantic tags defined within Woodwork. To get a list of the standard, index, and time index tags, you can use the `list_semantic_tags` function as shown below.

```
[2]: from woodwork import list_semantic_tags
list_semantic_tags()

[2]:
```

	name	is_standard_tag	\
0	category	True	
1	numeric	True	
2	index	False	
3	time_index	False	
4	date_of_birth	False	

	valid_logical_types
0	[Categorical, CountryCode, Ordinal, SubRegionC...
1	[Double, Integer]
2	[Integer, Double, Categorical, Datetime]
3	[Datetime]
4	[Datetime]

Working with Logical Types

When creating a `DataTable`, users have the option to specify the logical types for all, some, or none of the columns in the underlying dataframe. If logical types are defined for all of the columns, these logical types will be used directly, provided the data is compatible with the specified logical type. You cannot, for example, use a logical type of `Integer` on a column that contains text values that cannot be converted to integers.

If users do not supply any logical type information during creation of the `DataTable`, Woodwork will infer the logical types based on the physical type of the column and the information contained in the columns. If the user passes information for some of the columns, the logical types will be inferred for any columns not specified.

These scenarios are illustrated below. First, we will create a simple dataframe to use for this example.

```
[3]: import pandas as pd
      from woodwork import DataTable

      data = pd.DataFrame({
          'integers': [-2, 30, 20],
          'bools': [True, False, True],
          'names': ["Jane Doe", "Bill Smith", "John Hancock"]
      })
      data
```

```
[3]:   integers  bools      names
0         -2   True   Jane Doe
1          30  False  Bill Smith
2          20   True  John Hancock
```

Now that we have created the data to use for the example, we will create a `DataTable` object, assigning logical type values for each of the columns. We can then view the types stored for each column on the `DataTable`.

```
[4]: logical_types = {
      'integers': 'Integer',
      'bools': 'Boolean',
      'names': 'FullName'
    }

      dt = DataTable(data, logical_types=logical_types)
      dt
```

```
[4]:   Physical Type Logical Type Semantic Tag(s)
Data Column
integers          Int64      Integer  ['numeric']
bools              boolean    Boolean          []
names              string    FullName          []
```

As you can see, the logical types that we specified have been assigned to each of the columns. Now let's try this by assigning only one logical type value, and letting Woodwork infer the types for the other columns:

```
[5]: logical_types = {
      'names': 'FullName',
    }

      dt = DataTable(data, logical_types=logical_types)
      dt
```

```
[5]:   Physical Type Logical Type Semantic Tag(s)
Data Column
integers          Int64      Integer  ['numeric']
```

(continues on next page)

(continued from previous page)

bools	boolean	Boolean	[]
names	string	FullName	[]

With this input, we get the same results. Woodwork used the `FullName` logical type we assigned to the `names` column, and then correctly inferred the logical types for the `integers` and `bools` columns.

Next, we will look at what happens if we do not specify any logical types.

```
[6]: dt = DataTable(data)
dt
```

	Physical Type	Logical Type	Semantic Tag(s)
Data Column			
integers	Int64	Integer	['numeric']
bools	boolean	Boolean	[]
names	category	Categorical	['category']

In this case, Woodwork correctly inferred type for the `integers` and `bools` columns, but failed to recognize the `names` column should have a logical type of `FullName`. In situations like this, Woodwork provides users the ability to change the logical type.

Let's update the logical type of the `names` column to be `FullName`:

```
[7]: dt = dt.set_types(logical_types={'names': 'FullName'})
dt
```

	Physical Type	Logical Type	Semantic Tag(s)
Data Column			
integers	Int64	Integer	['numeric']
bools	boolean	Boolean	[]
names	string	FullName	[]

If we look carefully at the output, we will see that several things happened to the `names` column. First of all, the correct `FullName` logical type is now applied. Second, the physical type of the column has been changed from `category` to `string` to match the standard physical type for the `FullName` logical type. Finally, the standard tag of `category` that was previously set for the `names` column has been removed, as it no longer applies.

When setting the `LogicalType` for a column, the type can be specified by passing a string representing the camel-case name of the `LogicalType` class as we have done above. Alternatively, users can pass the class directly instead of a string, or the snake-case name of the string. All of these would be valid values to use for setting the `FullName` Logical type: `FullName`, `"FullName"` or `"full_name"`. Note, in order to use the class name the class must first be imported.

Working with Semantic Tags

Woodwork provides several methods for working with semantic types. Users can add and remove specific tags, as well as reset the tags to their default values. In this section, we will demonstrate these methods.

Note: When calling any of the methods that modify the logical type or semantic tags for a column, a new `DataTable` object will be returned and the original `DataTable` will be left unchanged. If you need to use the modified table, be sure to assign the returned `DataTable` to a variable to allow for proper access to it.

Standard Tags

As mentioned above, by default Woodwork will apply default semantic tags to columns, based on the logical type that was specified or inferred. If this behavior is undesirable, it can be controlled by setting the parameter `use_standard_tags` to `False` when creating the `DataTable`:

```
[8]: dt = DataTable(data, use_standard_tags=False)
dt
```

```
[8]:
```

	Physical Type	Logical Type	Semantic Tag(s)
Data Column			
integers	Int64	Integer	[]
bools	boolean	Boolean	[]
names	category	Categorical	[]

As can be seen in the table above, when creating a `DataTable` with the `use_standard_tags` set to `False`, all semantic tags will be empty. The only exception to this is if the index or time index column were set, and this will be discussed in more detail below.

Now, lets create a new table with the standard tags, and specify some additional user-defined semantic tags during creation:

```
[9]: semantic_tags = {
      'bools': 'user_status',
      'names': 'legal_name'
    }
dt = DataTable(data, semantic_tags=semantic_tags)
dt
```

```
[9]:
```

	Physical Type	Logical Type	Semantic Tag(s)
Data Column			
integers	Int64	Integer	['numeric']
bools	boolean	Boolean	['user_status']
names	category	Categorical	['category', 'legal_name']

Woodwork has applied the tags we specified along with any standard tags to the columns in our `DataTable`.

After creating the table, maybe we change our mind and decide we do not like the tag of `user_status` that was applied to the `bools` column and we want to remove it. We can do that with the `remove_semantic_tags` method.

```
[10]: dt = dt.remove_semantic_tags({'bools': 'user_status'})
dt
```

```
[10]:
```

	Physical Type	Logical Type	Semantic Tag(s)
Data Column			
integers	Int64	Integer	['numeric']
bools	boolean	Boolean	[]
names	category	Categorical	['category', 'legal_name']

As you can see, the `user_status` tag has now been removed.

Multiple tags can also be added to a column at once by passing a list of tags to add, instead of a single tag. Similarly, multiple tags can be removed at once, by passing a list of tags.

```
[11]: dt = dt.add_semantic_tags({'bools': ['tag1', 'tag2']})
dt
```

```
[11]:
```

	Physical Type	Logical Type	Semantic Tag(s)
Data Column			
integers	Int64	Integer	['numeric']
bools	boolean	Boolean	['tag2', 'tag1']
names	category	Categorical	['category', 'legal_name']

```
[12]: dt = dt.remove_semantic_tags({'bools':['tag1', 'tag2']})
dt
```

```
[12]:
```

	Physical Type	Logical Type	Semantic Tag(s)
Data Column			
integers	Int64	Integer	['numeric']
bools	boolean	Boolean	[]
names	category	Categorical	['category', 'legal_name']

Finally, all tags can be reset to their default values by using the `reset_semantic_tags` methods. If `use_standard_tags` is `True`, the tags will be reset to the standard tags. Otherwise, the tags will be reset to be empty sets.

```
[13]: dt = dt.reset_semantic_tags()
dt
```

```
[13]:
```

	Physical Type	Logical Type	Semantic Tag(s)
Data Column			
integers	Int64	Integer	['numeric']
bools	boolean	Boolean	[]
names	category	Categorical	['category']

In this case, since we created our `DataTable` with the default behavior of using standard tags, calling `reset_semantic_tags` resulted in all of our semantic tags being reset to the standard tags for each column.

Index and Time Index Tags

When creating a `DataTable` users can optionally specify which column represents the index and which column represents the time index. If these columns are specified, semantic tags of `index` and `time_index` will be applied to the specified column. Behind the scenes, Woodwork is performing additional validation checks on the columns to make sure they are appropriate. For example, index columns must be unique, and time index columns must contain datetime values or values that can be converted to datetimes.

Because of the need for these validation checks, users cannot set the `index` or `time_index` tags directly on a column. In order to designate a column as the index, the `set_index` method should be used. Similarly, in order to set the time index column, the `set_time_index` method should be used. Optionally, these can be specified when initially creating the `DataTable`.

Setting or changing the index

Let's create a new sample data set that contains columns that can be used as index and time index columns by using the `index` or `time_index` parameters.

```
[14]: data = pd.DataFrame({
    'index': [0, 1, 2],
    'id': [1, 2, 3],
    'times': pd.to_datetime(['2020-09-01', '2020-09-02', '2020-09-03']),
    'numbers': [10, 20, 30]
```

(continues on next page)

(continued from previous page)

```

})

dt = DataTable(data)
dt

```

```

[14]:          Physical Type Logical Type Semantic Tag(s)
Data Column
index          Int64      Integer      ['numeric']
id             Int64      Integer      ['numeric']
times         datetime64[ns] Datetime      []
numbers       Int64      Integer      ['numeric']

```

Without specifying an index or time index column during creation of the `DataTable`, Woodwork has inferred that the `index` and `id` columns are whole numbers and the numeric semantic tag has been applied. We can now set the index column with the `set_index` method:

```

[15]: dt = dt.set_index('index')
dt

```

```

[15]:          Physical Type Logical Type Semantic Tag(s)
Data Column
index          Int64      Integer      ['index']
id             Int64      Integer      ['numeric']
times         datetime64[ns] Datetime      []
numbers       Int64      Integer      ['numeric']

```

Inspecting the types now reveals that the `index` semantic tag has been added to the `index` column, and the numeric standard tag has been removed. We can also check that the index has been set correctly by checking the value of the `DataTable.index` attribute:

```

[16]: dt.index

```

```

[16]: 'index'

```

Perhaps we wanted to change the index column to be the `id` column instead. We can do this simply with another call to `set_index`.

```

[17]: dt = dt.set_index('id')
dt

```

```

[17]:          Physical Type Logical Type Semantic Tag(s)
Data Column
index          Int64      Integer      []
id             Int64      Integer      ['index']
times         datetime64[ns] Datetime      []
numbers       Int64      Integer      ['numeric']

```

Now we can see that the `index` tag has been removed from the `index` column and added to the `id` column. Also, of note, the numeric standard tag that was originally present on the `index` column has not been added back. If this tag is desired, the user must currently add it back using the `add_semantic_tags` method.

Setting or changing the time index

Setting the time index works similarly to setting the index. We can now set the time index for the `DataTable` with the `set_time_index` method.

```
[18]: dt = dt.set_time_index('times')
      dt
```

```
[18]:
```

	Physical Type	Logical Type	Semantic Tag(s)
Data Column			
index	Int64	Integer	['numeric']
id	Int64	Integer	['index']
times	datetime64[ns]	Datetime	['time_index']
numbers	Int64	Integer	['numeric']

As you can see, after calling `set_time_index`, the `time_index` semantic tag has been added to the `times` column of the `DataTable`.

Woodwork Global Configuration Options

Woodwork contains global configuration options that can be set by users to control the behavior of certain aspects of Woodwork. This guide will provide an overview of working with these options, including viewing the current settings and updating the config values.

Viewing Config Settings

First, we will demonstrate how to display the current configuration options. Once you have imported Woodwork, you can view the options with `ww.config` as shown below.

```
[1]: import woodwork as ww
      ww.config
```

```
[1]: Woodwork Global Config Settings
-----
natural_language_threshold: 10
numeric_categorical_threshold: -1
```

The output of `ww.config` lists each of the available config variables followed by its current setting. In the output above, the `natural_language_threshold` config variable has been set to 10 and the `numeric_categorical_threshold` has been set to -1.

Updating Config Settings

The process of updating a config variable is done simply with a call to the `ww.config.set_option` function. This function requires two arguments: the name of the config variable to update, and the new value to set.

To illustrate this, we will update the `natural_language_threshold` config variable to have a value of 25 instead of the default value of 10:

```
[2]: ww.config.set_option('natural_language_threshold', 25)
      ww.config
```

```
[2]: Woodwork Global Config Settings
-----
natural_language_threshold: 25
numeric_categorical_threshold: -1
```

As you can see from the output above, the value for the `natural_language_threshold` config variable has now been updated to 25.

Get Value for a Specific Config Variable

If you need access to the value that is set for a specific config variable you can access it with the `ww.config.get_option` function, passing in the name of the config variable for which you want the value:

```
[3]: ww.config.get_option('natural_language_threshold')
[3]: 25
```

Resetting to Default Values

Finally, config variables can be reset to their default values using the `ww.config.reset_option` function, passing in the name of the variable to reset. To demonstrate this, we will reset the `natural_language_threshold` config variable to its default value:

```
[4]: ww.config.reset_option('natural_language_threshold')
ww.config
[4]: Woodwork Global Config Settings
-----
natural_language_threshold: 10
numeric_categorical_threshold: -1
```

Available Config Settings

This section provides an overview of the current config options that can be set within Woodwork.

Natural Language Threshold

The `natural_language_threshold` config variable helps control the distinction between `Categorical` and `NaturalLanguage` logical types during type inference. More specifically, this threshold represents the average string length that is used to distinguish between these two types. If the average string length in a column is greater than this threshold, the column will be inferred as a `NaturalLanguage` column, otherwise it will be inferred as a `Categorical` column. The `natural_language_threshold` config variable defaults to 10.

Numeric Categorical Threshold

Woodwork provides the option to infer numeric columns as the `Categorical` logical type if they have few enough unique values. The `numeric_categorical_threshold` config variable allows users to set the threshold of unique values below which numeric columns will be inferred as categorical. The default threshold is `-1`, meaning that numeric columns will not be inferred to be `Categorical` by default (since the fewest number of unique values a column can have is zero).

Gain Statistical Insights into Your DataTable

Woodwork provides methods on `DataTable` to allow users to utilize the typing information inherent in a `DataTable` to better understand their data.

Let's walk through how to use `describe` and `mutual_information` on a retail `DataTable` so that we can see the full capabilities of the functions.

```
[1]: import pandas as pd
      from woodwork import DataTable
      from woodwork.demo import load_retail

      dt = load_retail()
      dt
```

```
[1]:
```

Data Column	Physical Type	Logical Type	Semantic Tag(s)
order_product_id	category	Categorical	['index']
order_id	category	Categorical	['category']
product_id	category	Categorical	['category']
description	string	NaturalLanguage	[]
quantity	Int64	Integer	['numeric']
order_date	datetime64[ns]	Datetime	['time_index']
unit_price	float64	Double	['numeric']
customer_name	category	Categorical	['category']
country	category	Categorical	['category']
total	float64	Double	['numeric']
cancelled	boolean	Boolean	[]

DataTable.describe

We use `dt.describe()` to calculate statistics for the Data Columns in a `DataTable` in the format of a Pandas `DataFrame` with the relevant calculations done for each Data Column.

```
[2]: dt.describe()
```

```
[2]:
```

	order_id	product_id	description \
physical_type	category	category	string
logical_type	Categorical	Categorical	NaturalLanguage
semantic_tags	{category}	{category}	{}
count	401604	401604	401604
nunique	22190	3684	NaN
nan_count	0	0	0
mean	NaN	NaN	NaN
mode	576339	85123A	WHITE HANGING HEART T-LIGHT HOLDER
std	NaN	NaN	NaN
min	NaN	NaN	NaN

(continues on next page)

(continued from previous page)

first_quartile	NaN	NaN	NaN
second_quartile	NaN	NaN	NaN
third_quartile	NaN	NaN	NaN
max	NaN	NaN	NaN
num_true	NaN	NaN	NaN
num_false	NaN	NaN	NaN
	quantity	order_date	unit_price \
physical_type	Int64	datetime64[ns]	float64
logical_type	Integer	Datetime	Double
semantic_tags	{numeric}	{time_index}	{numeric}
count	401604	401604	401604
nunique	436	20460	620
nan_count	0	0	0
mean	12.1833	2011-07-10 12:08:23.848567552	5.73221
mode	1	2011-11-14 15:27:00	2.0625
std	250.283	NaN	115.111
min	-80995	2010-12-01 08:26:00	0
first_quartile	2	NaN	2.0625
second_quartile	5	NaN	3.2175
third_quartile	12	NaN	6.1875
max	80995	2011-12-09 12:50:00	64300.5
num_true	NaN	NaN	NaN
num_false	NaN	NaN	NaN
	customer_name	country	total cancelled
physical_type	category	category	float64 boolean
logical_type	Categorical	Categorical	Double Boolean
semantic_tags	{category}	{category}	{numeric} {}
count	401604	401604	401604 401604
nunique	4372	37	3952 NaN
nan_count	0	0	0 0
mean	NaN	NaN	34.0125 NaN
mode	Mary Dalton	United Kingdom	24.75 False
std	NaN	NaN	710.081 NaN
min	NaN	NaN	-277975 NaN
first_quartile	NaN	NaN	7.0125 NaN
second_quartile	NaN	NaN	19.305 NaN
third_quartile	NaN	NaN	32.67 NaN
max	NaN	NaN	277975 NaN
num_true	NaN	NaN	NaN 8872
num_false	NaN	NaN	NaN 392732

There are a couple things to note in the above dataframe:

- The DataTable's index, `order_product_id`, is not included
- We provide each Data Column's typing information according to Woodwork's typing system
- Any statistic that cannot be calculated for a Data Column, say `num_false` on a `Datetime`, will be filled with `NaN`.
- Null values would not get counted in any of the calculations other than `nunique`

DataTable.value_counts

We can use `dt.value_counts()` to calculate the most frequent values for each Data Column that has category as a standard tag. This returns a dictionary where each Data Column is associated with a sorted list of dictionaries. Each dictionary contains value and count.

```
[3]: dt.value_counts()
[3]: {'order_product_id': [{'value': 401603, 'count': 1},
  {'value': 133859, 'count': 1},
  {'value': 133861, 'count': 1},
  {'value': 133862, 'count': 1},
  {'value': 133863, 'count': 1},
  {'value': 133864, 'count': 1},
  {'value': 133865, 'count': 1},
  {'value': 133866, 'count': 1},
  {'value': 133867, 'count': 1},
  {'value': 133868, 'count': 1}],
  'order_id': [{'value': '576339', 'count': 542},
  {'value': '579196', 'count': 533},
  {'value': '580727', 'count': 529},
  {'value': '578270', 'count': 442},
  {'value': '573576', 'count': 435},
  {'value': '567656', 'count': 421},
  {'value': '567183', 'count': 392},
  {'value': '575607', 'count': 377},
  {'value': '571441', 'count': 364},
  {'value': '570488', 'count': 353}],
  'product_id': [{'value': '85123A', 'count': 2065},
  {'value': '22423', 'count': 1894},
  {'value': '85099B', 'count': 1659},
  {'value': '47566', 'count': 1409},
  {'value': '84879', 'count': 1405},
  {'value': '20725', 'count': 1346},
  {'value': '22720', 'count': 1224},
  {'value': 'POST', 'count': 1196},
  {'value': '22197', 'count': 1110},
  {'value': '23203', 'count': 1108}],
  'customer_name': [{'value': 'Mary Dalton', 'count': 7812},
  {'value': 'Dalton Grant', 'count': 5898},
  {'value': 'Jeremy Woods', 'count': 5128},
  {'value': 'Jasmine Salazar', 'count': 4459},
  {'value': 'James Robinson', 'count': 2759},
  {'value': 'Bryce Stewart', 'count': 2478},
  {'value': 'Vanessa Sanchez', 'count': 2085},
  {'value': 'Laura Church', 'count': 1853},
  {'value': 'Kelly Alvarado', 'count': 1667},
  {'value': 'Ashley Meyer', 'count': 1640}],
  'country': [{'value': 'United Kingdom', 'count': 356728},
  {'value': 'Germany', 'count': 9480},
  {'value': 'France', 'count': 8475},
  {'value': 'EIRE', 'count': 7475},
  {'value': 'Spain', 'count': 2528},
  {'value': 'Netherlands', 'count': 2371},
  {'value': 'Belgium', 'count': 2069},
  {'value': 'Switzerland', 'count': 1877},
  {'value': 'Portugal', 'count': 1471},
  {'value': 'Australia', 'count': 1258}]}
```

DataTable.mutual_information()

`dt.mutual_information` will calculate the mutual information between all pairs of relevant Data Columns. Certain types such as strings cannot have mutual information calculated.

The mutual information between columns A and B can be understood as the amount of knowledge we can have about column A if we have the values of column B. The more mutual information there is between A and B, the less uncertainty there is in A knowing B or vice versa.

If we call `dt.mutual_information()`, we'll see that `order_date` will be excluded from the resulting dataframe.

```
[4]: dt.mutual_information()
[4]:
```

	column_1	column_2	mutual_info
0	order_id	customer_name	0.886411
1	order_id	product_id	0.475745
2	product_id	unit_price	0.426383
3	order_id	order_date	0.391906
4	product_id	customer_name	0.361855
5	order_date	customer_name	0.187982
6	quantity	total	0.184497
7	customer_name	country	0.155593
8	product_id	total	0.152183
9	order_id	total	0.129882
10	order_id	country	0.126048
11	order_id	quantity	0.114714
12	unit_price	total	0.103210
13	customer_name	total	0.099530
14	product_id	quantity	0.088663
15	quantity	customer_name	0.085515
16	quantity	unit_price	0.082515
17	order_id	unit_price	0.077681
18	product_id	order_date	0.057175
19	total	cancelled	0.044032
20	unit_price	customer_name	0.041308
21	quantity	cancelled	0.035528
22	product_id	country	0.028569
23	country	total	0.025071
24	order_id	cancelled	0.022204
25	quantity	country	0.021515
26	order_date	country	0.010361
27	customer_name	cancelled	0.006456
28	product_id	cancelled	0.003769
29	country	cancelled	0.003607
30	order_date	unit_price	0.003180
31	order_date	total	0.002625
32	unit_price	country	0.002603
33	quantity	order_date	0.002146
34	unit_price	cancelled	0.001677
35	order_date	cancelled	0.000199

Available Parameters

`dt.mutual_information` provides two parameters for tuning the mutual information calculation.

- `num_bins` - In order to calculate mutual information on continuous data, we bin numeric data into categories. This parameter allows users to choose the number of bins with which to categorize data.
 - Defaults to using 10 bins
 - The more bins there are, the more variety a column will have. The number of bins used should accurately portray the spread of the data.
- `nrows` - If `nrows` is set at a value below the number of rows in the `DataTable`, that number of rows will be randomly sampled from the underlying data
 - Defaults to using all the available rows.
 - Decreasing the number of rows can speed up the mutual information calculation on a `DataTable` with many rows, though care should be taken that the number being sampled is large enough to accurately portray the data.

Now we'll explore changing the number of bins. Note that this will only impact numeric Data Columns `quantity` and `unit_price`. We're going to increase the number of bins from 10 to 50, only showing the impacted columns.

```
[5]: mi = dt.mutual_information()
mi[mi['column_1'].isin(['unit_price', 'quantity']) | mi['column_2'].isin(['unit_price', 'quantity'])]
```

```
[5]:
```

	column_1	column_2	mutual_info
2	product_id	unit_price	0.426383
6	quantity	total	0.184497
11	order_id	quantity	0.114714
12	unit_price	total	0.103210
14	product_id	quantity	0.088663
15	quantity	customer_name	0.085515
16	quantity	unit_price	0.082515
17	order_id	unit_price	0.077681
20	unit_price	customer_name	0.041308
21	quantity	cancelled	0.035528
25	quantity	country	0.021515
30	order_date	unit_price	0.003180
32	unit_price	country	0.002603
33	quantity	order_date	0.002146
34	unit_price	cancelled	0.001677

```
[6]: mi = dt.mutual_information(num_bins = 50)
mi[mi['column_1'].isin(['unit_price', 'quantity']) | mi['column_2'].isin(['unit_price', 'quantity'])]
```

```
[6]:
```

	column_1	column_2	mutual_info
2	product_id	unit_price	0.528865
4	unit_price	total	0.405555
7	quantity	total	0.349243
10	order_id	quantity	0.157188
13	product_id	quantity	0.143938
14	order_id	unit_price	0.140257
16	quantity	customer_name	0.113431
17	quantity	unit_price	0.105052
18	quantity	cancelled	0.081334
19	unit_price	customer_name	0.078942

(continues on next page)

(continued from previous page)

24	quantity	country	0.023758
27	order_date	unit_price	0.011905
30	unit_price	country	0.006311
31	quantity	order_date	0.004170
34	unit_price	cancelled	0.001671

Using Woodwork with Dask and Koalas DataFrames

Woodwork enables DataTables to be created from Dask DataFrames or Koalas DataFrames when working with datasets that are too large to easily fit in memory. Although creating a DataTable from a Dask or Koalas DataFrame follows the same process as one would follow when creating a DataTable from a pandas DataFrame, there are a few limitations to be aware of. This guide will provide a brief overview of creating a DataTable starting with a Dask or Koalas DataFrame, and will outline several key items to keep in mind when using this approach.

Creating DataTables with either Dask or Koalas requires the installation of the Dask or Koalas libraries respectively, which can be installed directly with either of the following commands:

```
python -m pip install "woodwork[dask]"
```

```
python -m pip install "woodwork[koalas]"
```

Dask DataTable Example

First we will create a Dask DataFrame to use in our example. Normally you would create the DataFrame directly by reading in the data from saved files, but we will create it from a demo pandas DataFrame.

```
[1]: import dask.dataframe as dd
import woodwork as ww
```

```
df_pandas = ww.demo.load_retail(nrows=1000, return_dataframe=True)
df_dask = dd.from_pandas(df_pandas, npartitions=10)
df_dask
```

```
[1]: Dask DataFrame Structure:
```

```

      order_product_id order_id product_id description quantity  order_
↳date unit_price customer_name country    total cancelled
npartitions=10
0          int64      object      object      object      int64
↳datetime64[ns]  float64      object      object float64      bool
100          ...          ...          ...          ...          ...
↳...          ...          ...          ...          ...          ...
...          ...          ...          ...          ...          ...
↳...          ...          ...          ...          ...          ...
900          ...          ...          ...          ...          ...
↳...          ...          ...          ...          ...          ...
999          ...          ...          ...          ...          ...
↳...          ...          ...          ...          ...          ...
Dask Name: from_pandas, 10 tasks
```

Now that we have a Dask DataFrame, we can use it to create a Woodwork DataTable, just as we would with a pandas DataFrame:

```
[2]: dt = ww.DataTable(df_dask, index='order_product_id')
dt
```

```
[2]:
```

Data Column	Physical Type	Logical Type	Semantic Tag(s)
order_product_id	Int64	Integer	['index']
order_id	category	Categorical	['category']
product_id	category	Categorical	['category']
description	string	NaturalLanguage	[]
quantity	Int64	Integer	['numeric']
order_date	datetime64[ns]	Datetime	[]
unit_price	float64	Double	['numeric']
customer_name	string	NaturalLanguage	[]
country	string	NaturalLanguage	[]
total	float64	Double	['numeric']
cancelled	boolean	Boolean	[]

As you can see from the output above, the DataTable was created successfully, and logical type inference was performed for all of the columns. However, this brings us to one of the key issues in working with Dask DataFrames.

In order to perform logical type inference, Woodwork needs to bring the data into memory so it can be analyzed. Currently, Woodwork reads data from the first partition of data only, and then uses this data for type inference. Depending on the complexity of the data, this could be a time consuming operation. Additionally, if the first partition is not representative of the entire dataset, the logical types for some columns may be inferred incorrectly.

Skipping or Overriding Type Inference

If this process takes too much time, or if the logical types are not inferred correctly, users have the ability to manually specify the logical types for each column. If the logical type for a column is specified, type inference for that column will be skipped. If logical types are specified for all columns, logical type inference will be skipped completely and Woodwork will not need to bring any of the data into memory when creating the DataTable.

To skip logical type inference completely, and/or to correct type inference issues, you would simply define a logical types dictionary with the correct logical type defined for each column in the dataframe. Then, pass that dictionary to the call to create the DataTable as shown below:

```
[3]: logical_types = {
    'order_product_id': 'Integer',
    'order_id': 'Categorical',
    'product_id': 'Categorical',
    'description': 'NaturalLanguage',
    'quantity': 'Integer',
    'order_date': 'Datetime',
    'unit_price': 'Double',
    'customer_name': 'FullName',
    'country': 'Categorical',
    'total': 'Double',
    'cancelled': 'Boolean',
}

dt = ww.DataTable(df_dask, index='order_product_id', logical_types=logical_types)
dt
```

```
[3]:
```

Data Column	Physical Type	Logical Type	Semantic Tag(s)
order_product_id	Int64	Integer	['index']
order_id	category	Categorical	['category']
product_id	category	Categorical	['category']
description	string	NaturalLanguage	[]

(continues on next page)

(continued from previous page)

quantity	Int64	Integer	['numeric']
order_date	datetime64[ns]	Datetime	[]
unit_price	float64	Double	['numeric']
customer_name	string	FullName	[]
country	category	Categorical	['category']
total	float64	Double	['numeric']
cancelled	boolean	Boolean	[]

Analyzing Underlying Data

There are three `DataTable` methods that also require bringing the underlying Dask `DataFrame` into memory: `describe`, `value_counts` and `mutual_information`. When called, these methods will call a `compute` operation on the `DataFrame` associated with the `DataTable` in order to calculate the desired information. This may be problematic for datasets that cannot fit in memory, so exercise caution when using these methods.

```
[4]: dt.describe(include=['numeric'])
```

```
[4]:
```

	quantity	unit_price	total
physical_type	Int64	float64	float64
logical_type	Integer	Double	Double
semantic_tags	{numeric}	{numeric}	{numeric}
count	1000	1000	1000
nunique	43	61	232
nan_count	0	0	0
mean	12.735	5.00366	40.3905
mode	1	2.0625	24.75
std	38.4016	9.73817	123.994
min	-24	0.165	-68.31
first_quartile	2	2.0625	5.709
second_quartile	4	3.34125	17.325
third_quartile	12	6.1875	33.165
max	600	272.25	2684.88
num_true	NaN	NaN	NaN
num_false	NaN	NaN	NaN

```
[5]: dt.value_counts()
```

```
[5]: {'order_id': [{'value': '536464', 'count': 81},
  {'value': '536520', 'count': 71},
  {'value': '536412', 'count': 68},
  {'value': '536401', 'count': 64},
  {'value': '536415', 'count': 59},
  {'value': '536409', 'count': 54},
  {'value': '536408', 'count': 48},
  {'value': '536381', 'count': 35},
  {'value': '536488', 'count': 34},
  {'value': '536446', 'count': 31}],
  'product_id': [{'value': '22632', 'count': 11},
  {'value': '85123A', 'count': 10},
  {'value': '22633', 'count': 10},
  {'value': '84029E', 'count': 9},
  {'value': '22961', 'count': 9},
  {'value': '22960', 'count': 7},
  {'value': '84879', 'count': 7},
  {'value': '22866', 'count': 7},
```

(continues on next page)

(continued from previous page)

```
{'value': '21212', 'count': 7},
{'value': '22197', 'count': 7}},
'country': [{'value': 'United Kingdom', 'count': 964},
{'value': 'France', 'count': 20},
{'value': 'Australia', 'count': 14},
{'value': 'Netherlands', 'count': 2}]}
```

```
[6]: dt.mutual_information().head()
```

```
[6]:
```

	column_1	column_2	mutual_info
0	order_id	order_date	0.777905
1	order_id	product_id	0.595564
2	product_id	unit_price	0.517738
3	product_id	total	0.433166
4	product_id	order_date	0.404885

Koalas DataTable Example

As we did with Dask above, we will first create a Koalas DataFrame to use in our example. Normally you would create the DataFrame directly by reading in the data from saved files, but we will create it from a demo pandas DataFrame.

```
[7]: import databricks.koalas as ks
```

```
df_koalas = ks.from_pandas(df_pandas)
df_koalas.head()
```

```
[7]:
```

	order_product_id	order_id	product_id	description	country	total
↪	quantity	order_date	unit_price	customer_name		
↪	cancelled					
0	0	536365	85123A	WHITE HANGING HEART T-LIGHT HOLDER		
↪	6	2010-12-01 08:26:00	4.2075	Andrea Brown United Kingdom	25.245	False
1	1	536365	71053	WHITE METAL LANTERN		
↪	6	2010-12-01 08:26:00	5.5935	Andrea Brown United Kingdom	33.561	False
2	2	536365	84406B	CREAM CUPID HEARTS COAT HANGER		
↪	8	2010-12-01 08:26:00	4.5375	Andrea Brown United Kingdom	36.300	False
3	3	536365	84029G	KNITTED UNION FLAG HOT WATER BOTTLE		
↪	6	2010-12-01 08:26:00	5.5935	Andrea Brown United Kingdom	33.561	False
4	4	536365	84029E	RED WOOLLY HOTTIE WHITE HEART.		
↪	6	2010-12-01 08:26:00	5.5935	Andrea Brown United Kingdom	33.561	False

Now that we have a Koalas DataFrame, we can use it to create a Woodwork DataTable, just as we would with a pandas DataFrame:

```
[8]: dt = ww.DataTable(df_koalas, index='order_product_id')
dt
```

```
[8]:
```

Data Column	Physical Type	Logical Type	Semantic Tag(s)
order_product_id	int64	Integer	['index']
order_id	object	Categorical	['category']
product_id	object	Categorical	['category']
description	object	NaturalLanguage	['']
quantity	int64	Integer	['numeric']
order_date	datetime64[ns]	Datetime	['']
unit_price	float64	Double	['numeric']

(continues on next page)

(continued from previous page)

customer_name	object	NaturalLanguage	[]
country	object	NaturalLanguage	[]
total	float64	Double	['numeric']
cancelled	bool	Boolean	[]

As you can see from the output above, the `DataTable` was created successfully, and logical type inference was performed for all of the columns.

Notes on Koalas Dtype Conversions

In the types table above, one important thing to notice is that the physical types for the Koalas `DataTable` are quite different than the physical types for the Dask `DataTable`. The reason for this is that several of the physical types that can be used with pandas and Dask cannot be used with Koalas.

When a Woodwork `DataTable` is created, the dtype of the underlying dataframe columns are converted to a set of standard dtypes, defined by the `LogicalType pandas_dtype` property. Woodwork uses several nullable dtypes such as `Int64`, `string` and `boolean` which are not compatible with Koalas. In addition the `category` dtype cannot be used with Koalas `DataFrames`.

For `LogicalTypes` that have `pandas_dtype` properties that are not compatible with Koalas, Woodwork will try to convert the column dtype, but will be unsuccessful. At that point, Woodwork will then use a backup dtype that is compatible with Koalas. The implication of this is that creating a `DataTable` from a Koalas `DataFrame` may result in dtype values that are different than the values you would get when creating the `DataTable` from an identical pandas `DataFrame`.

The backup datatypes used are as follows: `Int64`: `int64` `boolean`: `bool` `string`: `object` `category`: `object`

Since Koalas does not support the `category` dtype, any column that is inferred or specified with a logical type of `Categorical` will have its values converted to strings and stored with a dtype of `object`. This means that a categorical column containing numeric values, will be converted into the equivalent string values when the `DataTable` is created.

As Koalas does not support the nullable `boolean` dtype, Woodwork will fall back to use the compatible `bool` dtype for columns with a `Boolean` `LogicalType`. If the corresponding column contains any missing values, these missing values will be converted to `False` values when the column dtype is updated.

Finally, Koalas does not support the `timedelta64[ns]` dtype. For this, there is not a clean backup dtype, so the use of `Timedelta` `LogicalType` is not supported in Koalas `DataTables`.

Skipping or Overriding Type Inference

As with Dask, Woodwork needs to bring the data into memory so it can be analyzed for type inference. Currently, Woodwork uses data from the first 100,000 rows of data only when using a Koalas `DataFrame` as input, and then uses this data for type inference. If the first 100,000 rows are not representative of the entire dataset, the logical types for some columns may be inferred incorrectly.

To skip logical type inference completely, and/or to correct type inference issues, simply define a logical types dictionary with the correct logical type defined for each column in the dataframe:

```
[9]: logical_types = {
      'order_product_id': 'Integer',
      'order_id': 'Categorical',
      'product_id': 'Categorical',
```

(continues on next page)

(continued from previous page)

```

'description': 'NaturalLanguage',
'quantity': 'Integer',
'order_date': 'Datetime',
'unit_price': 'Double',
'customer_name': 'FullName',
'country': 'Categorical',
'total': 'Double',
'cancelled': 'Boolean',
}

dt = ww.DataTable(df_koalas, index='order_product_id', logical_types=logical_types)
dt

```

```

[9]:
      Physical Type      Logical Type Semantic Tag(s)
Data Column
order_product_id      int64          Integer      ['index']
order_id              object        Categorical ['category']
product_id           object        Categorical ['category']
description           object        NaturalLanguage []
quantity             int64          Integer      ['numeric']
order_date           datetime64[ns]  Datetime     []
unit_price           float64         Double       ['numeric']
customer_name        object        FullName     []
country              object        Categorical ['category']
total                float64         Double       ['numeric']
cancelled            bool           Boolean      []

```

Analyzing Underlying Data

As with Dask, running `describe`, `value_counts` or `mutual_information` requires bringing the data into memory to perform the analysis. When called, these methods will call a `to_pandas` operation on the DataFrame associated with the DataTable in order to calculate the desired information. This may be problematic for very large datasets, so exercise caution when using these methods.

```

[10]: dt.describe(include=['numeric'])

```

```

[10]:
      quantity unit_price      total
physical_type      int64      float64      float64
logical_type      Integer      Double      Double
semantic_tags  {numeric}  {numeric}  {numeric}
count           1000        1000        1000
nunique           43          61          232
nan_count         0           0           0
mean            12.735      5.00366     40.3905
mode             1           2.0625     24.75
std             38.4016     9.73817     123.994
min             -24         0.165     -68.31
first_quartile   2           2.0625     5.709
second_quartile  4           3.34125    17.325
third_quartile   12         6.1875     33.165
max             600         272.25    2684.88
num_true         NaN         NaN         NaN
num_false        NaN         NaN         NaN

```

```

[11]: dt.value_counts()

```

```
[11]: {'order_id': [{'value': '536464', 'count': 81},
  {'value': '536520', 'count': 71},
  {'value': '536412', 'count': 68},
  {'value': '536401', 'count': 64},
  {'value': '536415', 'count': 59},
  {'value': '536409', 'count': 54},
  {'value': '536408', 'count': 48},
  {'value': '536381', 'count': 35},
  {'value': '536488', 'count': 34},
  {'value': '536446', 'count': 31}],
'product_id': [{'value': '22632', 'count': 11},
  {'value': '85123A', 'count': 10},
  {'value': '22633', 'count': 10},
  {'value': '22961', 'count': 9},
  {'value': '84029E', 'count': 9},
  {'value': '84879', 'count': 7},
  {'value': '22866', 'count': 7},
  {'value': '22960', 'count': 7},
  {'value': '22197', 'count': 7},
  {'value': '21212', 'count': 7}],
'country': [{'value': 'United Kingdom', 'count': 964},
  {'value': 'France', 'count': 20},
  {'value': 'Australia', 'count': 14},
  {'value': 'Netherlands', 'count': 2}]}
```

```
[12]: dt.mutual_information().head()
```

```
[12]:
```

	column_1	column_2	mutual_info
0	order_id	order_date	0.777905
1	order_id	product_id	0.595564
2	product_id	unit_price	0.517738
3	product_id	total	0.433166
4	product_id	order_date	0.404885

Data Validation Limitations

When creating a DataTable several validation checks are performed to confirm that the data in the underlying dataframe is appropriate for the specified parameters. Because some of these validation steps require pulling the underlying data into memory, they are skipped when creating a DataTable from a Dask or Koalas DataFrame. This section provides an overview of the validation checks that are performed with pandas input but skipped with Dask or Koalas input.

Index Uniqueness

Normally a check is performed to verify that any column specified as the index contains no duplicate values. With Dask or Koalas input, this check is skipped and users must manually verify that any column specified as an index column contains unique values.

Data Consistency with LogicalType (Dask Only)

If users manually define the LogicalType for a column when creating the DataTable, a check is performed to verify that the data in that column is appropriate for the specified LogicalType. For example, with pandas input if the user specifies a LogicalType of `Double` for a column that contains letters such as `['a', 'b', 'c']`, an error would be raised as it is not possible to convert the letters into numeric values with the `float` dtype associated with the `Double` LogicalType.

With Dask input, no such error would be raised at the time of DataTable creation. However, behind the scenes, Woodwork will have attempted to convert the column physical type to `float`, and this conversion would be added to the Dask task graph, without raising an error. However, an error will be raised if a `compute` operation is called on the underlying DataFrame once Dask attempts to execute the conversion step. Extra care should be taken when using Dask input to make sure any specified logical types are consistent with the data in the columns to avoid this type of error.

Ordinal Order Values Check

For the `Ordinal` LogicalType, a check is typically performed to make sure that the data column does not contain any values that are not present in the defined order values. This check will not be performed with Dask or Koalas input. Users should manually verify that the defined order values are complete to avoid unexpected results.

Other Limitations

Reading from CSV Files

Woodwork provides the ability to read data directly from a CSV file into a DataTable, and during this process Woodwork creates the underlying dataframe so the user does not have to do so. The helper function used for this, `woodwork.read_csv`, will currently only read the data into a pandas DataFrame. At some point, we hope to remove this limitation and also allow data to be read into a Dask or Koalas DataFrame, but for now only pandas DataFrames can be created by this function.

Sorting DataFrame on Time Index

When creating a DataTable with a time index, Woodwork by default will sort the input dataframe first on the time and then on the index, if specified. Because sorting a distributed DataFrame is a computationally expensive operation, this sorting is performed only when creating a DataTable from a pandas DataFrame. If a sorted DataFrame is needed when using a Dask or Koalas, the user should first sort the DataFrame before creating the DataTable.

Equality of Data Tables

In order to avoid bringing a Dask DataFrame into memory, Woodwork will not consider the equality of the underlying data when checking whether a DataTable made from a Dask or Koalas DataFrame is equal to another DataTable. This means that two DataTables with identical names, columns, indices, semantic tags, and LogicalTypes but different underlying data will return `True` if at least one of them uses Dask or Koalas.

LatLong Columns

When working with the LatLong logical type, Woodwork converts all LatLong columns to a standard format of a tuple of floats for Dask dataframes and a list of floats for Koalas dataframes. In order to do this, we bring the data into memory, which may be problematic for large datasets.

Integer Column Names

Woodwork generally allows column names of any format that is supported by the underlying DataFrame; however Dask DataTables do not currently support integer column names.

Customizing Logical Types and Type Inference

The default type system in Woodwork contains many built-in LogicalTypes that will work for a wide variety of datasets. For situations in which the built-in LogicalTypes are not sufficient, Woodwork allows users to create custom LogicalTypes.

Woodwork also has a set of standard type inference functions that can help in automatically identifying correct LogicalTypes in the data. Woodwork also allows users to override these existing functions, or add new functions for inferring any custom LogicalTypes that are added.

This guide will provide an overview of how to create custom LogicalTypes as well as how to override and add new type inference functions.

Viewing Built-In Logical Types

To view all of the default LogicalTypes in Woodwork, users can use the `list_logical_types` function. If the existing types are not sufficient for your needs, you can create and register new LogicalTypes for use in creating DataTables and DataColumnms.

```
[1]: import woodwork as ww

ww.list_logical_types()

[1]:
```

	name	type_string	\
0	Boolean	boolean	
1	Categorical	categorical	
2	CountryCode	country_code	
3	Datetime	datetime	
4	Double	double	
5	EmailAddress	email_address	
6	Filepath	filepath	
7	FullName	full_name	
8	IPAddress	ip_address	
9	Integer	integer	

(continues on next page)

(continued from previous page)

10	LatLong	lat_long		
11	NaturalLanguage	natural_language		
12	Ordinal	ordinal		
13	PhoneNumber	phone_number		
14	SubRegionCode	sub_region_code		
15	Timedelta	timedelta		
16	URL	url		
17	ZIPCode	zip_code		
			description	physical_type \
0	Represents Logical Types that contain binary v...			boolean
1	Represents Logical Types that contain unordere...			category
2	Represents Logical Types that contain categori...			category
3	Represents Logical Types that contain date and...			datetime64[ns]
4	Represents Logical Types that contain positive...			float64
5	Represents Logical Types that contain email ad...			string
6	Represents Logical Types that specify location...			string
7	Represents Logical Types that may contain firs...			string
8	Represents Logical Types that contain IP addre...			string
9	Represents Logical Types that contain positive...			Int64
10	Represents Logical Types that contain latitude...			object
11	Represents Logical Types that contain text or ...			string
12	Represents Logical Types that contain ordered ...			category
13	Represents Logical Types that contain numeric ...			string
14	Represents Logical Types that contain codes re...			category
15	Represents Logical Types that contain values s...			timedelta64[ns]
16	Represents Logical Types that contain URLs, wh...			string
17	Represents Logical Types that contain a series...			category
	standard_tags	is_default_type	is_registered	parent_type
0	{}	True	True	None
1	{category}	True	True	None
2	{category}	True	True	Categorical
3	{}	True	True	None
4	{numeric}	True	True	None
5	{}	True	True	NaturalLanguage
6	{}	True	True	NaturalLanguage
7	{}	True	True	NaturalLanguage
8	{}	True	True	NaturalLanguage
9	{numeric}	True	True	None
10	{}	True	True	None
11	{}	True	True	None
12	{category}	True	True	Categorical
13	{}	True	True	NaturalLanguage
14	{category}	True	True	Categorical
15	{}	True	True	None
16	{}	True	True	NaturalLanguage
17	{category}	True	True	Categorical

Registering a New LogicalType

The first step in registering a new LogicalType is to define the class for the new type. This can be done simply by sub-classing the built-in LogicalType class. There are a few class attributes that should be set when defining this new class. Each will be discussed in more detail below.

We will work through an example for a dataset that contains UPC Codes. First let's create a new UPCCode LogicalType. For this example we will consider the UPC Code to be a type of categorical variable.

```
[2]: from woodwork.logical_types import LogicalType

class UPCCode(LogicalType):
    """Represents Logical Types that contain 12-digit UPC Codes."""
    pandas_dtype = 'category'
    backup_dtype = 'str'
    standard_tags = {'category', 'upc_code'}
```

When defining the UPCCode LogicalType class, three class attributes were set. All three of these attributes are optional, and will default to the values defined on the LogicalType class if they are not set when defining the new type.

- `pandas_dtype`: This value specifies how the data will be stored. If the column in the underlying dataframe is not already of this type, Woodwork will convert the data to this dtype. This should be specified as a string that represents a valid pandas dtype. If not specified, this will default to `'string'`.
- `backup_dtype`: This is primarily useful when working with Koalas dataframes. `backup_dtype` specifies the dtype to use if Woodwork is unable to convert to the primary dtype specified by `pandas_dtype`. In our example, we set this to `'str'` since Koalas does not currently support the `'category'` dtype.
- `standard_tags`: This is a set of semantic tags to apply to any column that is set with the specified LogicalType. If not specified, `standard_tags` will default to an empty set.
- `docstring`: Adding a docstring for the class is optional, but if specified, this text will be used for adding a description of the type in the list of available types returned by `ww.list_logical_types()`.

Note: Behind the scenes, Woodwork uses the `category` and `numeric` semantic tags to determine whether a column is categorical or numeric column, respectively. If the new LogicalType you define represents a categorical or numeric type, you should include the appropriate tag in the set of tags specified for `standard_tags`.

Now that we have created our new LogicalType, we can register it with the Woodwork type system, so we can use it. All modifications to the type system can be performed by calling the appropriate method on the `ww.type_system` object.

```
[3]: ww.type_system.add_type(UPCCode, parent='Categorical')
```

Now, if we once again list the available LogicalTypes, we will see the new type we have created has been added to the list, including the values for description, `physical_type` and `standard_tags` we specified when defining the UPCCode LogicalType.

```
[4]: ww.list_logical_types()

[4]:
```

	name	type_string	\
0	Boolean	boolean	
1	Categorical	categorical	
2	CountryCode	country_code	
3	Datetime	datetime	
4	Double	double	

(continues on next page)

(continued from previous page)

5	EmailAddress	email_address			
6	Filepath	filepath			
7	FullName	full_name			
8	IPAddress	ip_address			
9	Integer	integer			
10	LatLong	lat_long			
11	NaturalLanguage	natural_language			
12	Ordinal	ordinal			
13	PhoneNumber	phone_number			
14	SubRegionCode	sub_region_code			
15	Timedelta	timedelta			
16	UPCCode	upc_code			
17	URL	url			
18	ZIPCode	zip_code			
			description	physical_type	\
0	Represents Logical Types that contain binary v...		binary v...	boolean	
1	Represents Logical Types that contain unordere...		unordere...	category	
2	Represents Logical Types that contain categori...		categori...	category	
3	Represents Logical Types that contain date and...		date and...	datetime64[ns]	
4	Represents Logical Types that contain positive...		positive...	float64	
5	Represents Logical Types that contain email ad...		email ad...	string	
6	Represents Logical Types that specify location...		specify location...	string	
7	Represents Logical Types that may contain firs...		may contain firs...	string	
8	Represents Logical Types that contain IP addre...		IP addre...	string	
9	Represents Logical Types that contain positive...		positive...	Int64	
10	Represents Logical Types that contain latitude...		latitude...	object	
11	Represents Logical Types that contain text or ...		text or ...	string	
12	Represents Logical Types that contain ordered ...		ordered ...	category	
13	Represents Logical Types that contain numeric ...		numeric ...	string	
14	Represents Logical Types that contain codes re...		codes re...	category	
15	Represents Logical Types that contain values s...		values s...	timedelta64[ns]	
16	Represents Logical Types that contain 12-digit...		12-digit...	category	
17	Represents Logical Types that contain URLs, wh...		URLs, wh...	string	
18	Represents Logical Types that contain a series...		a series...	category	
	standard_tags	is_default_type	is_registered	parent_type	
0	{}	True	True	None	
1	{category}	True	True	None	
2	{category}	True	True	Categorical	
3	{}	True	True	None	
4	{numeric}	True	True	None	
5	{}	True	True	NaturalLanguage	
6	{}	True	True	NaturalLanguage	
7	{}	True	True	NaturalLanguage	
8	{}	True	True	NaturalLanguage	
9	{numeric}	True	True	None	
10	{}	True	True	None	
11	{}	True	True	None	
12	{category}	True	True	Categorical	
13	{}	True	True	NaturalLanguage	
14	{category}	True	True	Categorical	
15	{}	True	True	None	
16	{upc_code, category}	False	True	Categorical	
17	{}	True	True	NaturalLanguage	
18	{category}	True	True	Categorical	

Logical Type Relationships

When adding a new type to the type system, users can specify an optional parent LogicalType as we have done above. When performing type inference it is possible that a given set of data will match multiple different LogicalTypes. Woodwork uses the parent-child relationship defined when registering a type to determine which type to infer in this case.

When multiple matches are found, Woodwork will return the most specific type match found. By setting the parent type to Categorical when registering our UPCCode LogicalType, we are telling Woodwork that if a data column matches both Categorical and UPCCode during inference, the column should be considered as UPCCode as this is more specific than Categorical. Woodwork always assumes that a child type is a more specific version than the parent type.

Working with Custom LogicalTypes

Next, we will create a small sample DataFrame to demonstrate how we can use our new custom type. This sample DataFrame includes an id column, a column with valid UPC Codes, and a column that should not be considered UPC Codes because it contains non-numeric values.

```
[5]: import pandas as pd
dataframe = pd.DataFrame({
    'id': [0, 1, 2, 3],
    'code': ['012345412359', '122345712358', '012345412359', '022323413459'],
    'not_upc': ['abcdefghijkl', '122345712358', '012345412359', '022323413459']
})
```

Before we use this dataframe, let's update Woodwork's default threshold for differentiating between a NaturalLanguage and Categorical column so that Woodwork will correctly recognize our code column as a Categorical column. After setting the threshold, we can create a new DataTable and verify that Woodwork has identified our column as Categorical.

```
[6]: ww.config.set_option('natural_language_threshold', 12)
dt = ww.DataTable(dataframe)
dt
```

```
[6]:
```

Data Column	Physical Type	Logical Type	Semantic Tag(s)
id	Int64	Integer	['numeric']
code	category	Categorical	['category']
not_upc	category	Categorical	['category']

The reason Woodwork did not identify the code column to have a UPCCode LogicalType, is that we have not yet defined an inference function to use with this type. The inference function is what tells Woodwork how to match columns to specific LogicalTypes.

Even without the inference function, we can manually tell Woodwork that the code column should be of type UPCCode. This will set the physical type properly and apply the standard semantic tags we have defined

```
[7]: dt = ww.DataTable(dataframe, logical_types = {'code': 'UPCCode'})
dt
```

```
[7]:
```

Data Column	Physical Type	Logical Type	Semantic Tag(s)
id	Int64	Integer	['numeric']
code	category	UPCCode	['upc_code', 'category']
not_upc	category	Categorical	['category']

Now, let's add a new inference function and allow Woodwork to automatically set the correct type for the `code` column.

Defining Custom Inference Functions

The first step in adding an inference function for the `UPCCode` LogicalType is to define an appropriate function. Inference functions should always accept a single parameter, a `pandas.Series`. The function should return `True` if the series is a match for the LogicalType for which the function is associated, or `False` if the series is not a match.

For the `UPCCode` LogicalType, let's define a function to check that all of the values in a column are 12 character strings that contain only numbers. Note, this function is for demonstration purposes only and may not catch all cases that need to be considered for properly identifying a UPC Code.

```
[8]: def infer_upc_code(series):
      # Make sure series contains only strings:
      if not series.apply(type).eq(str).all():
          return False
      # Check that all items are 12 characters long
      if all(series.str.len() == 12):
          # Try to convert to a number
          try:
              series.astype('int')
              return True
          except:
              return False
      return False
```

After defining our new UPC Code inference function, we can add it to the Woodwork type system, so it can be used when inferring column types.

```
[9]: ww.type_system.update_inference_function('UPCCode', inference_function=infer_upc_code)
```

After updating the inference function, we can create a new datatable from the same DataFrame. In doing so, we see that Woodwork has correctly identified the `code` column to have a LogicalType of `UPCCode` and has correctly set the physical type and added the standard tags to the semantic tags for that column.

Also note, that the `not_upc` column was identified as `Categorical`. Even though this column contain 12-digit strings, some of the values contained letters, and our inference function correctly told Woodwork this was not valid for the `UPCCode` LogicalType.

```
[10]: dt = ww.DataTable(dataframe)
      dt
```

```
[10]:
```

Data Column	Physical Type	Logical Type	Semantic Tag(s)
id	Int64	Integer	['numeric']
code	category	UPCCode	['upc_code', 'category']
not_upc	category	Categorical	['category']

Overriding Default Inference Functions

Overriding the default inference functions can also be done using the `update_inference_function` type system method. Simply pass in the `LogicalType` for which you want to override the function, along with the new function to use.

For example we can also tell Woodwork to use our new `infer_upc_code` function for the built in `Categorical` `LogicalType` as well.

```
[11]: ww.type_system.update_inference_function('Categorical', inference_function=infer_upc_
      ↪code)
```

If we create a new `DataTable`, after updating the `Categorical` function, we will now see that the `not_upc` column is no longer identified as a `Categorical` column, but is rather set to the default `NaturalLanguage` `LogicalType`. This is because the letters in the first row of the `not_upc` column cause our inference function to return `False` for this column, while the default `Categorical` function will allow non-numeric values to be present.

```
[12]: dt = ww.DataTable(dataframe)
      dt
```

```
[12]:
```

	Physical Type	Logical Type	Semantic Tag(s)
Data Column			
id	Int64	Integer	['numeric']
code	category	UPCCode	['upc_code', 'category']
not_upc	string	NaturalLanguage	[]

Updating LogicalType Relationships

If you need to change the parent for a registered `LogicalType`, you can do this using the `update_relationship` method. Let's update our new `UPCCode` `LogicalType` to be a child of `NaturalLanguage` instead.

```
[13]: ww.type_system.update_relationship('UPCCode', parent='NaturalLanguage')
```

The parent for a logical type can also be set to `None` to indicate this is a root-level `LogicalType` that is not related to any other existing `LogicalType`.

```
[14]: ww.type_system.update_relationship('UPCCode', parent=None)
```

Setting the proper parent-child relationships between logical types is important. Because Woodwork will return the most specific `LogicalType` match found during inference, improper inference can occur if the relationships are not set correctly.

As an example, if we create a new `DataTable` after setting our new `UPCCode` `LogicalType` to have a parent of `None`, we will now see that the `UPC Code` column is inferred as `Categorical` instead of `UPCCode`. After setting the parent to `None`, `UPCCode` and `Categorical` are now siblings in the relationship graph, instead of having a parent-child relationship as they did previously. When Woodwork finds multiple matches on the same level in the relationship graph, the first match is returned, which in this case is `Categorical`. Without proper parent-child relationships set, Woodwork is unable to determine which `LogicalType` is most specific.

```
[15]: dt = ww.DataTable(dataframe)
      dt
```

```
[15]:
```

	Physical Type	Logical Type	Semantic Tag(s)
Data Column			
id	Int64	Integer	['numeric']

(continues on next page)

(continued from previous page)

code	category	Categorical	['category']
not_upc	string	NaturalLanguage	[]

Removing a LogicalType

If a LogicalType is no longer needed, or is unwanted, it can be removed from the type system with the `remove_type` method. If a LogicalType that has children is removed, all of the children types will have their parent set to the parent of the LogicalType that is being removed, assuming a parent was defined.

Now that we are done with our example, let's remove our custom UPCCode type and confirm it has been removed by listing the available LogicalTypes.

```
[16]: ww.type_system.remove_type('UPCCode')
      ww.list_logical_types()
```

```
[16]:
```

	name	type_string	\
0	Boolean	boolean	
1	Categorical	categorical	
2	CountryCode	country_code	
3	Datetime	datetime	
4	Double	double	
5	EmailAddress	email_address	
6	Filepath	filepath	
7	FullName	full_name	
8	IPAddress	ip_address	
9	Integer	integer	
10	LatLong	lat_long	
11	NaturalLanguage	natural_language	
12	Ordinal	ordinal	
13	PhoneNumber	phone_number	
14	SubRegionCode	sub_region_code	
15	Timedelta	timedelta	
16	UPCCode	upc_code	
17	URL	url	
18	ZIPCode	zip_code	

	description	physical_type	\
0	Represents Logical Types that contain binary v...	boolean	
1	Represents Logical Types that contain unordere...	category	
2	Represents Logical Types that contain categori...	category	
3	Represents Logical Types that contain date and...	datetime64[ns]	
4	Represents Logical Types that contain positive...	float64	
5	Represents Logical Types that contain email ad...	string	
6	Represents Logical Types that specify location...	string	
7	Represents Logical Types that may contain firs...	string	
8	Represents Logical Types that contain IP addre...	string	
9	Represents Logical Types that contain positive...	Int64	
10	Represents Logical Types that contain latitude...	object	
11	Represents Logical Types that contain text or ...	string	
12	Represents Logical Types that contain ordered ...	category	
13	Represents Logical Types that contain numeric ...	string	
14	Represents Logical Types that contain codes re...	category	
15	Represents Logical Types that contain values s...	timedelta64[ns]	
16	Represents Logical Types that contain 12-digit...	category	
17	Represents Logical Types that contain URLs, wh...	string	

(continues on next page)

(continued from previous page)

	standard_tags	is_default_type	is_registered	parent_type
0	{}	True	True	None
1	{category}	True	True	None
2	{category}	True	True	Categorical
3	{}	True	True	None
4	{numeric}	True	True	None
5	{}	True	True	NaturalLanguage
6	{}	True	True	NaturalLanguage
7	{}	True	True	NaturalLanguage
8	{}	True	True	NaturalLanguage
9	{numeric}	True	True	None
10	{}	True	True	None
11	{}	True	True	None
12	{category}	True	True	Categorical
13	{}	True	True	NaturalLanguage
14	{category}	True	True	Categorical
15	{}	True	True	None
16	{upc_code, category}	False	False	None
17	{}	True	True	NaturalLanguage
18	{category}	True	True	Categorical

Resetting Type System to Defaults

Finally, if you have multiple changes to the default Woodwork type system and would like to reset everything back to the default state, you can use the `reset_defaults` method as shown below. This will unregister any new types you have registered, reset all relationships to their default values and set all inference functions back to their default functions.

```
[17]: ww.type_system.reset_defaults()
```

Overriding Default LogicalTypes

There may be times when you would like to override Woodwork's default LogicalTypes. An example might be if the `Int64` dtype that the `Integer` LogicalType uses is incompatible with your data and you would like it to use `int64`. In this case, you would want to stop Woodwork from inferring the default `Integer` LogicalType and have a compatible Logical Type inferred instead. You may solve this issue in one of two ways.

First, you can create an entirely new LogicalType with its own name, `MyInteger`, and register it in the Type System. If you'd like to infer it in place of the normal `Integer` LogicalType, you would remove `Integer` from the type system, and use `Integer`'s default inference function for `MyInteger`. Doing this will make it such that `MyInteger` will get inferred any place that `Integer` would.

```
[18]: from woodwork.logical_types import LogicalType

class MyInteger(LogicalType):
    pandas_dtype = 'int64'
    standard_tags = {'numeric'}
int_inference_fn = ww.type_system.inference_functions[ww.logical_types.Integer]

ww.type_system.remove_type(ww.logical_types.Integer)
```

(continues on next page)

(continued from previous page)

```
ww.type_system.add_type(MyInteger, int_inference_fn)

dt = ww.DataTable(dataframe)
dt
```

```
[18]:
```

	Physical Type	Logical Type	Semantic Tag(s)
Data Column			
id	int64	MyInteger	['numeric']
code	category	Categorical	['category']
not_upc	category	Categorical	['category']

Above, we can see that the `id` column, which had previously been inferred as `Integer` is now being inferred as `MyInteger` with the `int64` physical type. In the full list of Logical Types at `ww.list_logical_types()`, `Integer` and `MyInteger` will now both be present, but `Integer`'s `is_registered` will be `False` while `MyInteger` will be set to `True` at `is_registered`.

The second option for overriding the default Logical Types allows you to create a new `LogicalType` with the same name as an existing one. This might be desirable because it will allow Woodwork to interpret the string `'Integer'` as your new Logical Type, allowing a user who might have selected `'Integer'` to not have to update their code to reference a new Logical Type like `MyInteger`.

Before adding a Logical Type whose name already exists into the `TypeSystem`, you must first unregister the default Logical Type.

In order to avoid using the same name space locally between `Integer` Logical Types, it is recommended to reference Woodwork's default Logical Type as `ww.logical_types.Integer`.

```
[19]: ww.type_system.reset_defaults()
class Integer(LogicalType):
    pandas_dtype = 'int64'
    standard_tags = {'numeric'}
int_inference_fn = ww.type_system.inference_functions[ww.logical_types.Integer]

ww.type_system.remove_type(ww.logical_types.Integer)
ww.type_system.add_type(Integer, int_inference_fn)

dt = ww.DataTable(dataframe)
display(dt)
ww.type_system.reset_defaults()
```

	Physical Type	Logical Type	Semantic Tag(s)
Data Column			
id	int64	Integer	['numeric']
code	category	Categorical	['category']
not_upc	category	Categorical	['category']

Notice how `id` now gets inferred as an `Integer` Logical Type that has `int64` as its Physical Type!

1.1.5 API Reference

DataTable

<code>DataTable(dataframe[, name, index, ...])</code>	
<code>DataTable.shape</code>	Returns a tuple representing the dimensionality of the DataTable.
<code>DataTable.add_semantic_tags(semantic_tags)</code>	Adds specified semantic tags to columns.
<code>DataTable.remove_semantic_tags(semantic_tags)</code>	Remove the semantic tags for any column names in the provided semantic_tags dictionary.
<code>DataTable.reset_semantic_tags([columns, ...])</code>	Reset the semantic tags for the specified columns to the default values and return a new DataTable.
<code>DataTable.select(include)</code>	Create a DataTable including only columns whose logical type and semantic tags are specified in the list of types and tags to include.
<code>DataTable.iloc</code>	Purely integer-location based indexing for selection by position.
<code>DataTable.set_index(index)</code>	Set the index column and return a new DataTable.
<code>DataTable.set_types([logical_types, ...])</code>	Update the logical type and semantic tags for any columns names in the provided types dictionary.
<code>DataTable.set_time_index(time_index)</code>	Set the time index column.
<code>DataTable.to_dataframe()</code>	Retrieves the DataTable's underlying dataframe.
<code>DataTable.describe([include])</code>	Calculates statistics for data contained in DataTable.
<code>DataTable.describe_dict([include])</code>	Calculates statistics for data contained in DataTable.
<code>DataTable.mutual_information([num_bins, nrows])</code>	Calculates mutual information between all pairs of columns in the DataTable that support mutual information.
<code>DataTable.mutual_information_dict(...)</code>	Calculates mutual information between all pairs of columns in the DataTable that support mutual information.
<code>DataTable.value_counts([ascending, top_n, ...])</code>	Returns a list of dictionaries with counts for the most frequent values in each column (only)
<code>DataTable.to_csv(path[, sep, encoding, ...])</code>	Write DataTable to disk in the CSV format, location specified by <i>path</i> .
<code>DataTable.to_pickle(path[, compression, ...])</code>	Write DataTable to disk in the pickle format, location specified by <i>path</i> .
<code>DataTable.to_parquet(path[, compression, ...])</code>	Write DataTable to disk in the parquet format, location specified by <i>path</i> .
<code>DataTable.rename(columns)</code>	Renames columns in a DataTable
<code>DataTable.update_dataframe(new_df[, ...])</code>	Replace the DataTable's dataframe with a new dataframe, making sure the new dataframe dtypes are updated.
<code>DataTable.head([n])</code>	Shows the first n rows of the DataTable along with typing information.
<code>DataTable.drop(columns)</code>	Drop specified columns from a DataTable.

woodwork.datatable.DataTable

```
class woodwork.datatable.DataTable(dataframe, name=None, index=None, time_index=None,
                                   semantic_tags=None, logical_types=None, table_metadata=None,
                                   column_metadata=None, use_standard_tags=True, make_index=False,
                                   column_descriptions=None, already_sorted=False)
```

```
__init__(dataframe, name=None, index=None, time_index=None, semantic_tags=None,
          logical_types=None, table_metadata=None, column_metadata=None,
          use_standard_tags=True, make_index=False, column_descriptions=None,
          already_sorted=False)
```

Create DataTable

Parameters

- **dataframe** (*pd.DataFrame*, *dd.DataFrame*, *ks.DataFrame*, *numpy.ndarray*) – Dataframe providing the data for the datatable.
- **name** (*str*, *optional*) – Name used to identify the datatable.
- **index** (*str*, *optional*) – Name of the index column in the dataframe.
- **time_index** (*str*, *optional*) – Name of the time index column in the dataframe.
- **semantic_tags** (*dict*, *optional*) – Dictionary mapping column names in the dataframe to the semantic tags for the column. The keys in the dictionary should be strings that correspond to columns in the underlying dataframe. There are two options for specifying the dictionary values: (*str*): If only one semantic tag is being set, a single string can be used as a value. (*list[str]* or *set[str]*): If multiple tags are being set, a list or set of strings can be used as the value. Semantic tags will be set to an empty set for any column not included in the dictionary.
- **logical_types** (*dict[str -> LogicalType]*, *optional*) – Dictionary mapping column names in the dataframe to the *LogicalType* for the column. *LogicalTypes* will be inferred for any columns not present in the dictionary.
- **table_metadata** (*dict[str -> json serializable]*, *optional*) – Dictionary containing extra metadata for the *DataTable*.
- **column_metadata** (*dict[str -> dict[str -> json serializable]]*, *optional*) – Dictionary mapping column names to that column's metadata dictionary.
- **use_standard_tags** (*bool*, *optional*) – If *True*, will add standard semantic tags to columns based on the inferred or specified logical type for the column. Defaults to *True*.
- **make_index** (*bool*, *optional*) – If *True*, will create a new unique, numeric index column with the name specified by *index* and will add the new index column to the supplied *DataFrame*. If *True*, the name specified in *index* cannot match an existing column name in *dataframe*. If *False*, the name is specified in *index* must match a column present in the *dataframe*. Defaults to *False*.
- **column_descriptions** (*dict[str -> str]*, *optional*) – Dictionary containing column descriptions
- **already_sorted** (*bool*, *optional*) – Indicates whether the input dataframe is already sorted on the time index. If *False*, will sort the dataframe first on the *time_index* and then on the *index* (pandas *DataFrame* only). Defaults to *False*.

Methods

<code>__init__(dataframe[, name, index, ...])</code>	Create DataTable
<code>add_semantic_tags(semantic_tags)</code>	Adds specified semantic tags to columns.
<code>describe([include])</code>	Calculates statistics for data contained in DataTable.
<code>describe_dict([include])</code>	Calculates statistics for data contained in DataTable.
<code>drop(columns)</code>	Drop specified columns from a DataTable.
<code>head([n])</code>	Shows the first n rows of the DataTable along with typing information.
<code>mutual_information([num_bins, nrows])</code>	Calculates mutual information between all pairs of columns in the DataTable that support mutual information.
<code>mutual_information_dict([num_bins, nrows])</code>	Calculates mutual information between all pairs of columns in the DataTable that support mutual information.
<code>pop(column_name)</code>	Return a DataColumn and drop it from the DataTable.
<code>remove_semantic_tags(semantic_tags)</code>	Remove the semantic tags for any column names in the provided semantic_tags dictionary.
<code>rename(columns)</code>	Renames columns in a DataTable
<code>reset_semantic_tags([columns, retain_index_tags])</code>	re-Reset the semantic tags for the specified columns to the default values and return a new DataTable.
<code>select(include)</code>	Create a DataTable including only columns whose logical type and semantic tags are specified in the list of types and tags to include.
<code>set_index(index)</code>	Set the index column and return a new DataTable.
<code>set_time_index(time_index)</code>	Set the time index column.
<code>set_types([logical_types, semantic_tags, ...])</code>	Update the logical type and semantic tags for any columns names in the provided types dictionary.
<code>to_csv(path[, sep, encoding, engine, ...])</code>	Write DataTable to disk in the CSV format, location specified by <i>path</i> .
<code>to_dataframe()</code>	Retrieves the DataTable's underlying dataframe.
<code>to_dictionary()</code>	Get a DataTable's description
<code>to_parquet(path[, compression, profile_name])</code>	Write DataTable to disk in the parquet format, location specified by <i>path</i> .
<code>to_pickle(path[, compression, profile_name])</code>	Write DataTable to disk in the pickle format, location specified by <i>path</i> .
<code>update_dataframe(new_df[, already_sorted])</code>	Replace the DataTable's dataframe with a new dataframe, making sure the new dataframe dtypes are updated.
<code>value_counts([ascending, top_n, dropna])</code>	Returns a list of dictionaries with counts for the most frequent values in each column (only

Attributes

<code>df</code>	
<code>iloc</code>	Purely integer-location based indexing for selection by position.
<code>index</code>	The index column for the table
<code>logical_types</code>	A dictionary containing logical types for each column
<code>ltypes</code>	A series listing the logical types for each column in the table
<code>physical_types</code>	A dictionary containing physical types for each column
<code>semantic_tags</code>	A dictionary containing semantic tags for each column
<code>shape</code>	Returns a tuple representing the dimensionality of the DataTable.
<code>time_index</code>	The time index column for the table
<code>types</code>	Dataframe containing the physical dtypes, logical types and semantic tags for the table

woodwork.datatable.DataTable.shape

property `DataTable.shape`

Returns a tuple representing the dimensionality of the DataTable. If Dask DataFrame, returns a Dask *Delayed* object for the number of rows.

woodwork.datatable.DataTable.add_semantic_tags

`DataTable.add_semantic_tags` (*semantic_tags*)

Adds specified semantic tags to columns. Will retain any previously set values. Replaces updated columns with new DataColumn objects and returns a new DataTable object.

Parameters `semantic_tags` (*dict[str -> str/list/set]*) – A dictionary mapping the columns in the DataTable to the tags that should be added to the column

Returns DataTable with semantic tags added

Return type woodwork.DataTable

woodwork.datatable.DataTable.remove_semantic_tags

`DataTable.remove_semantic_tags` (*semantic_tags*)

Remove the semantic tags for any column names in the provided semantic_tags dictionary. Replaces the column with a new DataColumn object and return a new DataTable object.

Parameters `semantic_tags` (*dict[str -> str/list/set]*) – A dictionary mapping the columns in the DataTable to the tags that should be removed to the column

Returns DataTable with the specified semantic tags removed

Return type woodwork.DataTable

woodwork.datatable.DataTable.reset_semantic_tags

`DataTable.reset_semantic_tags` (*columns=None, retain_index_tags=False*)

Reset the semantic tags for the specified columns to the default values and return a new `DataTable`. The default values will be either an empty set or a set of the standard tags based on the column logical type, controlled by the `use_standard_tags` property on the table. Column names can be provided as a single string, a list of strings or a set of strings. If `columns` is not specified, tags will be reset for all columns.

Parameters

- **columns** (*str/list/set*) – The columns for which the semantic tags should be reset.
- **retain_index_tags** (*bool, optional*) – If `True`, will retain any index or `time_index` semantic tag set on the column. If `False`, will clear all semantic tags. Defaults to `False`.

Returns `DataTable` with semantic tags reset to default values

Return type `woodwork.DataTable`

woodwork.datatable.DataTable.select

`DataTable.select` (*include*)

Create a `DataTable` including only columns whose logical type and semantic tags are specified in the list of types and tags to include. If no matching columns are found, an empty `DataTable` will be returned.

Parameters include (*str or LogicalType or list[str or LogicalType]*) – Logical types, semantic tags or column names to include in the `DataTable`.

Returns The subset of the original `DataTable` that contains just the logical types and semantic tags in `include`.

Return type `DataTable`

woodwork.datatable.DataTable.iloc

property `DataTable.iloc`

Purely integer-location based indexing for selection by position. `.iloc[]` is primarily integer position based (from 0 to `length-1` of the axis), but may also be used with a boolean array.

Allowed inputs are: An integer, e.g. 5. A list or array of integers, e.g. `[4, 3, 0]`. A slice object with ints, e.g. `1:7`. A boolean array. A callable function with one argument (the calling `Series`, `DataFrame` or `Panel`) and that returns valid output for indexing (one of the above). This is useful in method chains, when you don't have a reference to the calling object, but would like to base your selection on some value.

woodwork.datatable.DataTable.set_index

`DataTable.set_index` (*index*)

Set the index column and return a new `DataTable`. Adds the 'index' semantic tag to the column and clears the tag from any previously set index column. Setting a column as the index column will also cause any previously set standard tags for the column to be removed.

Parameters index (*str*) – The name of the column to set as the index

Returns `DataTable` with the specified index column set.

Return type `woodwork.DataTable`

woodwork.datatable.DataTable.set_types

`DataTable.set_types` (*logical_types=None, semantic_tags=None, retain_index_tags=True*)

Update the logical type and semantic tags for any columns names in the provided types dictionary. Replaces existing columns with new DataColumn objects and returns a new DataTable object.

Parameters

- **logical_types** (*dict[str -> str], optional*) – A dictionary defining the new logical types for the specified columns.
- **semantic_tags** (*dict[str -> str/list/set], optional*) – A dictionary defining the new semantic_tags for the specified columns.
- **retain_index_tags** (*bool, optional*) – If True, will retain any index or time_index semantic tags set on the column. If False, will replace all semantic tags. Defaults to True.

Returns DataTable with updated logical types and specified semantic tags set.

Return type woodwork.DataTable

woodwork.datatable.DataTable.set_time_index

`DataTable.set_time_index` (*time_index*)

Set the time index column. Adds the 'time_index' semantic tag to the column and clears the tag from any previously set index column

Parameters `time_index` (*str*) – The name of the column to set as the time index.

woodwork.datatable.DataTable.to_dataframe

`DataTable.to_dataframe` ()

Retrieves the DataTable's underlying dataframe.

Note: Do not modify the returned dataframe directly to avoid unexpected behavior

Returns

The underlying dataframe of the DataTable. Return type will depend on the type of dataframe used to create the DataTable.

Return type DataFrame

woodwork.datatable.DataTable.describe

`DataTable.describe` (*include=None*)

Calculates statistics for data contained in DataTable.

Parameters

- **include** (*list[str or LogicalType], optional*) – filter for what columns to include in the
- **returned. Can be a list of columns** (*statistics*) –
- **tags** (*semantic*) –
- **types** (*logical*) –

- a list (*or*) –
- any of the three. It follows the most broad specification. Favors logical types (*combining*) –
- semantic tag then column name. If no matching columns are found (*then*) –
- empty DataFrame (*an*) –
- be returned. (*will*) –

Returns A Dataframe containing statistics for the data or the subset of the original DataTable that contains the logical types, semantic tags, or column names specified in `include`.

Return type `pd.DataFrame`

`woodwork.datatable.DataTable.describe_dict`

`DataTable.describe_dict` (*include=None*)

Calculates statistics for data contained in `DataTable`.

Parameters

- **include** (*list[str or LogicalType], optional*) – filter for what columns to include in the
- **returned**. Can be a list of columns (*statistics*) –
- **tags** (*semantic*) –
- **types** (*logical*) –
- a list (*or*) –
- any of the three. It follows the most broad specification. Favors logical types (*combining*) –
- semantic tag then column name. If no matching columns are found (*then*) –
- empty DataFrame (*an*) –
- be returned. (*will*) –

Returns A dictionary with a key for each column in the data or for each column matching the logical types, semantic tags or column names specified in `include`, paired with a value containing a dictionary containing relevant statistics for that column.

Return type `dict[str -> dict]`

`woodwork.datatable.DataTable.mutual_information`

`DataTable.mutual_information` (*num_bins=10, nrows=None*)

Calculates mutual information between all pairs of columns in the `DataTable` that support mutual information. Logical Types that support mutual information are as follows: Boolean, Categorical, CountryCode, Datetime, Double, Integer, Ordinal, SubRegionCode, and ZIPCode

Parameters

- **num_bins** (*int*) – Determines number of bins to use for converting numeric features into categorical.

- **nrows** (*int*) – The number of rows to sample for when determining mutual info. If specified, samples the desired number of rows from the data. Defaults to using all rows.

Returns A Dataframe containing mutual information with columns *column_1*, *column_2*, and *mutual_info* that is sorted in decending order by mutual info. Mutual information values are between 0 (no mutual information) and 1 (perfect dependency).

Return type pd.DataFrame

woodwork.datatable.DataTable.mutual_information_dict

DataTable.**mutual_information_dict** (*num_bins=10, nrows=None*)

Calculates mutual information between all pairs of columns in the DataTable that support mutual information. Logical Types that support mutual information are as follows: Boolean, Categorical, CountryCode, Datetime, Double, Integer, Ordinal, SubRegionCode, and ZIPCode

Parameters

- **num_bins** (*int*) – Determines number of bins to use for converting numeric features into categorical.
- **nrows** (*int*) – The number of rows to sample for when determining mutual info. If specified, samples the desired number of rows from the data. Defaults to using all rows.

Returns A list containing dictionaries that have keys *column_1*, *column_2*, and *mutual_info* that is sorted in decending order by mutual info. Mutual information values are between 0 (no mutual information) and 1 (perfect dependency).

Return type list(dict)

woodwork.datatable.DataTable.value_counts

DataTable.**value_counts** (*ascending=False, top_n=10, dropna=False*)

Returns a list of dictionaries with counts for the most frequent values in each column (only for DataColumns with *category* as a standard tag).

Parameters

- **ascending** (*bool*) – Defines whether each list of values should be sorted most frequent to least frequent value (False), or least frequent to most frequent value (True). Defaults to False.
- **top_n** (*int*) – the number of top values to retrieve. Defaults to 10.
- **dropna** (*bool*) – determines whether to remove NaN values when finding frequency. Defaults to False.

Returns

a list of dictionaries for each categorical column with keys *count* and *value*.

Return type top_list (list(dict))

woodwork.datatable.DataTable.to_csv

`DataTable.to_csv` (*path*, *sep*=';', *encoding*='utf-8', *engine*='python', *compression*=None, *profile_name*=None)

Write `DataTable` to disk in the CSV format, location specified by *path*. Path could be a local path or a S3 path. If writing to S3 a tar archive of files will be written.

Parameters

- **path** (*str*) – Location on disk to write to (will be created as a directory)
- **sep** (*str*) – String of length 1. Field delimiter for the output file.
- **encoding** (*str*) – A string representing the encoding to use in the output file, defaults to 'utf-8'.
- **engine** (*str*) – Name of the engine to use. Possible values are: {'c', 'python'}.
- **compression** (*str*) – Name of the compression to use. Possible values are: {'gzip', 'bz2', 'zip', 'xz', None}.
- **profile_name** (*str*) – Name of AWS profile to use, False to use an anonymous profile, or None.

woodwork.datatable.DataTable.to_pickle

`DataTable.to_pickle` (*path*, *compression*=None, *profile_name*=None)

Write `DataTable` to disk in the pickle format, location specified by *path*. Path could be a local path or a S3 path. If writing to S3 a tar archive of files will be written.

Parameters

- **path** (*str*) – Location on disk to write to (will be created as a directory)
- **compression** (*str*) – Name of the compression to use. Possible values are: {'gzip', 'bz2', 'zip', 'xz', None}.
- **profile_name** (*str*) – Name of AWS profile to use, False to use an anonymous profile, or None.

woodwork.datatable.DataTable.to_parquet

`DataTable.to_parquet` (*path*, *compression*=None, *profile_name*=None)

Write `DataTable` to disk in the parquet format, location specified by *path*. Path could be a local path or a S3 path. If writing to S3 a tar archive of files will be written.

Note: As the engine *fastparquet* cannot handle nullable pandas dtypes, *pyarrow* will be used for serialization to parquet.

Parameters

- **path** (*str*) – location on disk to write to (will be created as a directory)
- **compression** (*str*) – Name of the compression to use. Possible values are: {'snappy', 'gzip', 'brotli', None}.

- **profile_name** (*str*) – Name of AWS profile to use, False to use an anonymous profile, or None.

woodwork.datatable.DataTable.rename

`DataTable.rename` (*columns*)

Renames columns in a DataTable

Parameters **columns** (*dict[str -> str]*) – A dictionary mapping columns whose names we'd like to change to the name to which we'd like to change them.

Returns DataTable with the specified columns renamed.

Return type woodwork.DataTable

Note: Index and time index columns cannot be renamed.

woodwork.datatable.DataTable.update_dataframe

`DataTable.update_dataframe` (*new_df, already_sorted=False*)

Replace the DataTable's dataframe with a new dataframe, making sure the new dataframe dtypes are updated. If the original DataTable was created with `make_index=True`, an index column will be added to the updated data if it is not present.

Parameters

- **new_df** (*DataFrame*) – Dataframe containing the new data. The same columns present in the original data should also be present in the new dataframe.
- **already_sorted** (*bool, optional*) – Indicates whether the input dataframe is already sorted on the time index. If False, will sort the dataframe first on the `time_index` and then on the `index` (pandas DataFrame only). Defaults to False.

woodwork.datatable.DataTable.head

`DataTable.head` (*n=5*)

Shows the first n rows of the DataTable along with typing information.

Note: This will bring data into memory for Dask or Koalas DataTables.

Parameters **n** (*int*) – number of rows to return. Defaults to 5.

Returns DataFrame with the top n rows where the column headers contain

each DataColumn's typing information.

woodwork.datatable.DataTable.drop

`DataTable.drop(columns)`

Drop specified columns from a `DataTable`.

Parameters `columns` (*str or list[str]*) – Column name or names to drop. Must be present in the `DataTable`.

Returns `DataTable` with the specified columns removed.

Return type `woodwork.DataTable`

DataColumn

<code>DataColumn(series[, logical_type, ...])</code>	
<code>DataColumn.shape</code>	Returns a tuple representing the dimensionality of the <code>DataTable</code> .
<code>DataColumn.iloc</code>	Purely integer-location based indexing for selection by position.
<code>DataColumn.add_semantic_tags(semantic_tags)</code>	Add the specified semantic tags to the column and return a new <code>DataColumn</code> object.
<code>DataColumn.remove_semantic_tags(semantic_tags)</code>	Removes specified semantic tags from column and returns a new column.
<code>DataColumn.reset_semantic_tags(...)</code>	Reset the semantic tags to the default values.
<code>DataColumn.set_logical_type(logical_type[, ...])</code>	Update the logical type for the column and return a new <code>DataColumn</code> object.
<code>DataColumn.set_semantic_tags(semantic_tags)</code>	Replace current semantic tags with new values and return a new <code>DataColumn</code> object.
<code>DataColumn.to_series()</code>	Retrieves the <code>DataColumn</code> 's underlying series.

woodwork.datacolumn.DataColumn

class `woodwork.datacolumn.DataColumn` (*series, logical_type=None, semantic_tags=None, use_standard_tags=True, name=None, description=None, metadata=None*)

__init__ (*series, logical_type=None, semantic_tags=None, use_standard_tags=True, name=None, description=None, metadata=None*)

Create a `DataColumn`.

Parameters

- **series** (*pd.Series or dd.Series or numpy.ndarray or pd.api.extensions.ExtensionArray*) – Series containing the data associated with the column.
- **logical_type** (*LogicalType, optional*) – The logical type that should be assigned to the column. If no value is provided, the `LogicalType` for the series will be inferred.
- **semantic_tags** (*str or list or set, optional*) – Semantic tags to assign to the column. Defaults to an empty set if not specified. There are two options for specifying the semantic tags: (str) If only one semantic tag is being set, a single string can be passed. (list or set) If multiple tags are being set, a list or set of strings can be passed.

- **use_standard_tags** (*bool, optional*) – If True, will add standard semantic tags to columns based on the inferred or specified logical type for the column. Defaults to True.
- **name** (*str, optional*) – Name of DataColumn. Will overwrite Series name, if it exists.
- **description** (*str, optional*) – Optional text describing the contents of the column
- **metadata** (*dict[str -> json serializable], optional*) – Metadata associated with the column.

Methods

<code>__init__(series[, logical_type, ...])</code>	Create a DataColumn.
<code>add_semantic_tags(semantic_tags)</code>	Add the specified semantic tags to the column and return a new DataColumn object.
<code>remove_semantic_tags(semantic_tags)</code>	Removes specified semantic tags from column and returns a new column.
<code>reset_semantic_tags([retain_index_tags])</code>	Reset the semantic tags to the default values.
<code>set_logical_type(logical_type[, ...])</code>	Update the logical type for the column and return a new DataColumn object.
<code>set_semantic_tags(semantic_tags[, ...])</code>	Replace current semantic tags with new values and return a new DataColumn object.
<code>to_series()</code>	Retrieves the DataColumn's underlying series.

Attributes

<code>dtype</code>	The dtype of the underlying series
<code>iloc</code>	Purely integer-location based indexing for selection by position.
<code>logical_type</code>	The logical type for the column
<code>name</code>	The name of the column
<code>semantic_tags</code>	The set of semantic tags currently assigned to the column
<code>shape</code>	Returns a tuple representing the dimensionality of the DataTable.

`woodwork.datacolumn.DataColumn.shape`

property `DataColumn.shape`

Returns a tuple representing the dimensionality of the DataTable. If Dask DataFrame, returns a Dask *Delayed* object for the number of rows.

woodwork.datacolumn.DataColumn.iloc**property** `DataColumn.iloc`

Purely integer-location based indexing for selection by position. `.iloc[]` is primarily integer position based (from 0 to `length-1` of the axis), but may also be used with a boolean array.

Allowed inputs are: An integer, e.g. 5. A list or array of integers, e.g. `[4, 3, 0]`. A slice object with ints, e.g. `1:7`. A boolean array. A callable function with one argument (the calling Series, DataFrame or Panel) and that returns valid output for indexing (one of the above). This is useful in method chains, when you don't have a reference to the calling object, but would like to base your selection on some value.

woodwork.datacolumn.DataColumn.add_semantic_tags

`DataColumn.add_semantic_tags(semantic_tags)`

Add the specified semantic tags to the column and return a new `DataColumn` object.

Parameters `semantic_tags` (*str/list/set*) – New semantic tag(s) to add to the column

Returns `DataColumn` with specified semantic tags added.

Return type `woodwork.DataColumn`

woodwork.datacolumn.DataColumn.remove_semantic_tags

`DataColumn.remove_semantic_tags(semantic_tags)`

Removes specified semantic tags from column and returns a new column.

Parameters `semantic_tags` (*str/list/set*) – Semantic tag(s) to remove from the column.

Returns `DataColumn` with specified tags removed.

Return type `woodwork.DataColumn`

woodwork.datacolumn.DataColumn.reset_semantic_tags

`DataColumn.reset_semantic_tags(retain_index_tags=False)`

Reset the semantic tags to the default values. The default values will be either an empty set or a set of the standard tags based on the column logical type, controlled by the `use_standard_tags` property.

Parameters `retain_index_tags` (*bool, optional*) – If True, any 'index' or 'time_index' tags on the column will be retained. If False, all tags will be cleared. Defaults to False.

Returns `DataColumn` with reset semantic tags.

Return type `woodwork.DataColumn`

woodwork.datacolumn.DataColumn.set_logical_type

`DataColumn.set_logical_type(logical_type, retain_index_tags=True)`

Update the logical type for the column and return a new `DataColumn` object.

Parameters

- **logical_type** (*LogicalType, str*) – The new logical type to set for the column.
- **retain_index_tags** (*bool, optional*) – If `True`, any ‘index’ or ‘time_index’ tags on the column will be retained. If `False`, all tags will be cleared. Defaults to `True`.

Returns `DataColumn` with updated logical type.

Return type `woodwork.DataColumn`

woodwork.datacolumn.DataColumn.set_semantic_tags

`DataColumn.set_semantic_tags(semantic_tags, retain_index_tags=True)`

Replace current semantic tags with new values and return a new `DataColumn` object.

Parameters

- **semantic_tags** (*str/list/set*) – New semantic tag(s) to set for column
- **retain_index_tags** (*bool, optional*) – If `True`, any ‘index’ or ‘time_index’ tags on the column will be retained. If `False`, all tags will be replaced. Defaults to `True`.

Returns `DataColumn` with specified semantic tags.

Return type `woodwork.DataColumn`

woodwork.datacolumn.DataColumn.to_series

`DataColumn.to_series()`

Retrieves the `DataColumn`’s underlying series.

Note: Do not modify the returned series directly to avoid unexpected behavior

Returns

The underlying series of the `DataColumn`. Return type will depend on the type of series used to create the `DataColumn`.

Return type `Series`

Logical Types

<code>Boolean()</code>	Represents Logical Types that contain binary values indicating true/false.
<code>Categorical([encoding])</code>	Represents Logical Types that contain unordered discrete values that fall into one of a set of possible values.
<code>CountryCode()</code>	Represents Logical Types that contain categorical information specifically used to represent countries.
<code>Datetime([datetime_format])</code>	Represents Logical Types that contain date and time information.

continues on next page

Table 7 – continued from previous page

<i>Double()</i>	Represents Logical Types that contain positive and negative numbers, some of which include a fractional component.
<i>Integer()</i>	Represents Logical Types that contain positive and negative numbers without a fractional component, including zero (0).
<i>EmailAddress()</i>	Represents Logical Types that contain email address values.
<i>Filepath()</i>	Represents Logical Types that specify locations of directories and files in a file system.
<i>FullName()</i>	Represents Logical Types that may contain first, middle and last names, including honorifics and suffixes.
<i>IPAddress()</i>	Represents Logical Types that contain IP addresses, including both IPv4 and IPv6 addresses.
<i>LatLong()</i>	Represents Logical Types that contain latitude and longitude values in decimal degrees.
<i>NaturalLanguage()</i>	Represents Logical Types that contain text or characters representing natural human language
<i>Ordinal(order)</i>	Represents Logical Types that contain ordered discrete values.
<i>PhoneNumber()</i>	Represents Logical Types that contain numeric digits and characters representing a phone number
<i>SubRegionCode()</i>	Represents Logical Types that contain codes representing a portion of a larger geographic region.
<i>Timedelta()</i>	Represents Logical Types that contain values specifying a duration of time
<i>URL()</i>	Represents Logical Types that contain URLs, which may include protocol, hostname and file name
<i>ZIPCode()</i>	Represents Logical Types that contain a series of postal codes used by the US Postal Service for representing a group of addresses.

woodwork.logical_types.Boolean

class woodwork.logical_types.**Boolean**

Represents Logical Types that contain binary values indicating true/false.

Examples

```
[True, False, True]
[0, 1, 1]
```

__init__ ()

Initialize self. See help(type(self)) for accurate signature.

Methods

<code>__init__()</code>	Initialize self.
-------------------------	------------------

Attributes

<code>backup_dtype</code>
<code>pandas_dtype</code>
<code>standard_tags</code>
<code>type_string</code>

woodwork.logical_types.Categorical

class woodwork.logical_types.**Categorical** (*encoding=None*)

Represents Logical Types that contain unordered discrete values that fall into one of a set of possible values. Has 'category' as a standard tag.

Examples

```
["red", "green", "blue"]
["produce", "dairy", "bakery"]
[3, 1, 2]
```

`__init__` (*encoding=None*)
Initialize self. See help(type(self)) for accurate signature.

Methods

<code>__init__([encoding])</code>	Initialize self.
-----------------------------------	------------------

Attributes

<code>backup_dtype</code>
<code>pandas_dtype</code>
<code>standard_tags</code>
<code>type_string</code>

woodwork.logical_types.CountryCode

class woodwork.logical_types.**CountryCode**

Represents Logical Types that contain categorical information specifically used to represent countries. Has 'category' as a standard tag.

Examples

```
["AUS", "USA", "UKR"]
["GB", "NZ", "DE"]
```

`__init__()`

Initialize self. See help(type(self)) for accurate signature.

Methods

`__init__()`

Initialize self.

Attributes

backup_dtype

pandas_dtype

standard_tags

type_string

woodwork.logical_types.Datetime

class woodwork.logical_types.**Datetime** (*datetime_format=None*)

Represents Logical Types that contain date and time information.

Parameters `datetime_format` (*str*) – Desired datetime format for data

Examples

```
["2020-09-10",
 "2020-01-10 00:00:00",
 "01/01/2000 08:30"]
```

`__init__` (*datetime_format=None*)

Initialize self. See help(type(self)) for accurate signature.

Methods

<code>__init__</code> ([datetime_format])	Initialize self.
---	------------------

Attributes

<code>backup_dtype</code>
<code>datetime_format</code>
<code>pandas_dtype</code>
<code>standard_tags</code>
<code>type_string</code>

woodwork.logical_types.Double

class woodwork.logical_types.Double

Represents Logical Types that contain positive and negative numbers, some of which include a fractional component. Includes zero (0). Has 'numeric' as a standard tag.

Examples

```
[1.2, 100.4, 3.5]
[-15.34, 100, 58.3]
```

`__init__`()
Initialize self. See help(type(self)) for accurate signature.

Methods

<code>__init__</code> ()	Initialize self.
--------------------------	------------------

Attributes

<code>backup_dtype</code>
<code>pandas_dtype</code>
<code>standard_tags</code>
<code>type_string</code>

woodwork.logical_types.Integer**class** woodwork.logical_types.**Integer**

Represents Logical Types that contain positive and negative numbers without a fractional component, including zero (0). Has 'numeric' as a standard tag.

Examples

```
[100, 35, 0]
[-54, 73, 11]
```

__init__()

Initialize self. See help(type(self)) for accurate signature.

Methods

```
__init__()
```

Initialize self.

Attributes

```
backup_dtype
```

```
pandas_dtype
```

```
standard_tags
```

```
type_string
```

woodwork.logical_types.EmailAddress**class** woodwork.logical_types.**EmailAddress**

Represents Logical Types that contain email address values.

Examples

```
["john.smith@example.com",
 "support@example.com",
 "team@example.com"]
```

__init__()

Initialize self. See help(type(self)) for accurate signature.

Methods

<code>__init__()</code>	Initialize self.
-------------------------	------------------

Attributes

<code>backup_dtype</code>
<code>pandas_dtype</code>
<code>standard_tags</code>
<code>type_string</code>

woodwork.logical_types.Filepath

class woodwork.logical_types.**Filepath**

Represents Logical Types that specify locations of directories and files in a file system.

Examples

```
["/usr/local/bin",  
 "/Users/john.smith/dev/index.html",  
 "/tmp"]
```

`__init__()`
Initialize self. See `help(type(self))` for accurate signature.

Methods

<code>__init__()</code>	Initialize self.
-------------------------	------------------

Attributes

<code>backup_dtype</code>
<code>pandas_dtype</code>
<code>standard_tags</code>
<code>type_string</code>

woodwork.logical_types.FullName**class** woodwork.logical_types.**FullName**

Represents Logical Types that may contain first, middle and last names, including honorifics and suffixes.

Examples

```
["Mr. John Doe, Jr.",
 "Doe, Mrs. Jane",
 "James Brown"]
```

__init__()

Initialize self. See help(type(self)) for accurate signature.

Methods**__init__()**

Initialize self.

Attributes

backup_dtype

pandas_dtype

standard_tags

type_string

woodwork.logical_types.IPAddress**class** woodwork.logical_types.**IPAddress**

Represents Logical Types that contain IP addresses, including both IPv4 and IPv6 addresses.

Examples

```
["172.16.254.1",
 "192.0.0.0",
 "2001:0db8:0000:0000:0000:ff00:0042:8329"]
```

__init__()

Initialize self. See help(type(self)) for accurate signature.

Methods

<code>__init__()</code>	Initialize self.
-------------------------	------------------

Attributes

<code>backup_dtype</code>
<code>pandas_dtype</code>
<code>standard_tags</code>
<code>type_string</code>

woodwork.logical_types.LatLong

class woodwork.logical_types.LatLong

Represents Logical Types that contain latitude and longitude values in decimal degrees.

Note: LatLong values will be stored with the object dtype as a tuple of floats (or a list of floats for Koalas DataTables) and must contain only two values.

Null latitude or longitude values will be stored as `np.nan`, and a fully null LatLong (`np.nan, np.nan`) will be stored as just a single `nan`.

Examples

```
[ (33.670914, -117.841501),  
  (40.423599, -86.921162),  
  (-45.031705, nan) ]
```

`__init__()`
Initialize self. See `help(type(self))` for accurate signature.

Methods

<code>__init__()</code>	Initialize self.
-------------------------	------------------

Attributes

<code>backup_dtype</code>
<code>pandas_dtype</code>
<code>standard_tags</code>
<code>type_string</code>

woodwork.logical_types.NaturalLanguage

class woodwork.logical_types.NaturalLanguage

Represents Logical Types that contain text or characters representing natural human language

Examples

```
["This is a short sentence.",
 "I like to eat pizza!",
 "When will humans go to mars?"]
```

`__init__()`

Initialize self. See help(type(self)) for accurate signature.

Methods

`__init__()`

Initialize self.

Attributes

`backup_dtype`

`pandas_dtype`

`standard_tags`

`type_string`

woodwork.logical_types.Ordinal

class woodwork.logical_types.Ordinal (*order*)

Represents Logical Types that contain ordered discrete values. Has ‘category’ as a standard tag.

Parameters `order` (*list or tuple*) – An list or tuple specifying the order of the ordinal values from low to high. The underlying series cannot contain values that are not present in the order values.

Examples

```
["first", "second", "third"]
["bronze", "silver", "gold"]
```

`__init__(order)`

Initialize self. See help(type(self)) for accurate signature.

Methods

<code>__init__(order)</code>	Initialize self.
------------------------------	------------------

Attributes

<code>backup_dtype</code>
<code>pandas_dtype</code>
<code>standard_tags</code>
<code>type_string</code>

woodwork.logical_types.PhoneNumber

class woodwork.logical_types.**PhoneNumber**

Represents Logical Types that contain numeric digits and characters representing a phone number

Examples

```
["1-(555)-123-5495",  
 "+1-555-123-5495",  
 "5551235495"]
```

`__init__()`
Initialize self. See help(type(self)) for accurate signature.

Methods

<code>__init__()</code>	Initialize self.
-------------------------	------------------

Attributes

<code>backup_dtype</code>
<code>pandas_dtype</code>
<code>standard_tags</code>
<code>type_string</code>

woodwork.logical_types.SubRegionCode

class woodwork.logical_types.**SubRegionCode**

Represents Logical Types that contain codes representing a portion of a larger geographic region. Has 'category' as a standard tag.

Examples

```
["US-CO", "US-MA", "US-CA"]
["AU-NSW", "AU-TAS", "AU-QLD"]
```

__init__()

Initialize self. See help(type(self)) for accurate signature.

Methods

__init__()

Initialize self.

Attributes

backup_dtype

pandas_dtype

standard_tags

type_string

woodwork.logical_types.Timedelta

class woodwork.logical_types.**Timedelta**

Represents Logical Types that contain values specifying a duration of time

Examples

```
[pd.Timedelta('1 days 00:00:00'),
 pd.Timedelta('-1 days +23:40:00'),
 pd.Timedelta('4 days 12:00:00')]
```

__init__()

Initialize self. See help(type(self)) for accurate signature.

Methods

<code>__init__()</code>	Initialize self.
-------------------------	------------------

Attributes

<code>backup_dtype</code>
<code>pandas_dtype</code>
<code>standard_tags</code>
<code>type_string</code>

woodwork.logical_types.URL

class woodwork.logical_types.URL

Represents Logical Types that contain URLs, which may include protocol, hostname and file name

Examples

```
["http://google.com",  
 "https://example.com/index.html",  
 "example.com"]
```

`__init__()`
Initialize self. See help(type(self)) for accurate signature.

Methods

<code>__init__()</code>	Initialize self.
-------------------------	------------------

Attributes

<code>backup_dtype</code>
<code>pandas_dtype</code>
<code>standard_tags</code>
<code>type_string</code>

woodwork.logical_types.ZIPCode**class** woodwork.logical_types.ZIPCode

Represents Logical Types that contain a series of postal codes used by the US Postal Service for representing a group of addresses. Has 'category' as a standard tag.

Examples

```
["90210"
 "60018-0123",
 "10021"]
```

__init__()

Initialize self. See help(type(self)) for accurate signature.

Methods

<code>__init__()</code>	Initialize self.
-------------------------	------------------

Attributes

<code>backup_dtype</code>
<code>pandas_dtype</code>
<code>standard_tags</code>
<code>type_string</code>

TypeSystem

<code>TypeSystem([inference_functions, ...])</code>	
<code>TypeSystem.add_type(logical_type[, ...])</code>	Add a new LogicalType to the TypeSystem, optionally specifying the corresponding inference function and a parent type.
<code>TypeSystem.infer_logical_type(series)</code>	Infer the logical type for the given series
<code>TypeSystem.remove_type(logical_type)</code>	Remove a logical type from the TypeSystem.
<code>TypeSystem.update_inference_function(...)</code>	Update the inference function for the specified Logical-Type.
<code>TypeSystem.update_relationship(logical_type, Add or update a relationship. ...)</code>	

woodwork.type_sys.type_system.TypeSystem

```
class woodwork.type_sys.type_system.TypeSystem (inference_functions=None,
                                             relationships=None,           de-
                                             fault_type=NaturalLanguage)
```

```
__init__ (inference_functions=None, relationships=None, default_type=NaturalLanguage)
```

Create a new TypeSystem object. LogicalTypes that are present in the keys of the inference_functions dictionary will be considered registered LogicalTypes.

Parameters

- **inference_functions** (*dict[LogicalType->func], optional*) – Dictionary mapping LogicalTypes to their corresponding type inference functions. If None, only the default LogicalType will be registered without an inference function.
- **relationships** (*list, optional*) – List of tuples, each with two elements, specifying parent-child relationships between logical types. The first element should be the parent LogicalType. The second element should be the child LogicalType. If not specified, will default to an empty list indicating all types should be considered root types with no children.
- **default_type** (*LogicalType, optional*) – The default LogicalType to use if no inference matches are found. If not specified, will default to the built-in NaturalLanguage LogicalType.

Methods

<code><i>__init__</i>([inference_functions, ...])</code>	Create a new TypeSystem object.
<code><i>add_type</i>(logical_type[, inference_function, ...])</code>	Add a new LogicalType to the TypeSystem, optionally specifying the corresponding inference function and a parent type.
<code><i>infer_logical_type</i>(series)</code>	Infer the logical type for the given series
<code><i>remove_type</i>(logical_type)</code>	Remove a logical type from the TypeSystem.
<code><i>reset_defaults</i>()</code>	Reset type system to the default settings that were specified at initialization.
<code><i>str_to_logical_type</i>(logical_str[, params, ...])</code>	Helper function for converting a string value to the corresponding logical type object.
<code><i>update_inference_function</i>(logical_type, ...)</code>	Update the inference function for the specified LogicalType.
<code><i>update_relationship</i>(logical_type, parent)</code>	Add or update a relationship.

Attributes

<code><i>registered_types</i></code>	Returns a list of all registered types
<code><i>root_types</i></code>	Returns a list of all registered types that do not have a parent type

woodwork.type_sys.type_system.TypeSystem.add_type

TypeSystem.**add_type** (*logical_type*, *inference_function=None*, *parent=None*)

Add a new LogicalType to the TypeSystem, optionally specifying the corresponding inference function and a parent type.

Parameters

- **logical_type** (*LogicalType*) – The new LogicalType to add.
- **inference_function** (*func*, *optional*) – The inference function to use for inferring the given LogicalType. Defaults to None. If not specified, this LogicalType will never be inferred.
- **parent** (*LogicalType*, *optional*) – The parent LogicalType, if applicable. Defaults to None. If not specified, this type will be considered a root type with no parent.

woodwork.type_sys.type_system.TypeSystem.infer_logical_type

TypeSystem.**infer_logical_type** (*series*)

Infer the logical type for the given series

Parameters **series** (*pandas.Series*) – The series for which to infer the LogicalType.

woodwork.type_sys.type_system.TypeSystem.remove_type

TypeSystem.**remove_type** (*logical_type*)

Remove a logical type from the TypeSystem. Any children of the remove type will have their parent set to the parent of the removed type.

Parameters **logical_type** (*LogicalType*) – The LogicalType to remove.

woodwork.type_sys.type_system.TypeSystem.update_inference_function

TypeSystem.**update_inference_function** (*logical_type*, *inference_function*)

Update the inference function for the specified LogicalType.

Parameters

- **logical_type** (*LogicalType*) – The LogicalType for which to update the inference function.
- **inference_function** (*func*) – The new inference function to use. Can be set to None to skip type inference for the specified LogicalType.

woodwork.type_sys.type_system.TypeSystem.update_relationship

TypeSystem.**update_relationship** (*logical_type*, *parent*)

Add or update a relationship. If the specified LogicalType exists in the relationship graph, its parent will be updated. If the specified LogicalType does not exist in relationships, the relationship will be added.

Parameters

- **logical_type** (*LogicalType*) – The LogicalType for which to update the parent value.
- **parent** (*LogicalType*) – The new parent to set for the specified LogicalType.

Utils

Type Utils

<code>list_logical_types</code>	Returns a dataframe describing all of the available Logical Types.
<code>list_semantic_tags</code>	Returns a dataframe describing all of the common semantic tags.

woodwork.type_sys.utils.list_logical_types

woodwork.type_sys.utils.**list_logical_types** ()

Returns a dataframe describing all of the available Logical Types.

Parameters None –

Returns A dataframe containing details on each LogicalType, including the corresponding physical type and any standard semantic tags.

Return type pd.DataFrame

woodwork.type_sys.utils.list_semantic_tags

woodwork.type_sys.utils.**list_semantic_tags** ()

Returns a dataframe describing all of the common semantic tags.

Parameters None –

Returns A dataframe containing details on each Semantic Tag, including the corresponding logical type(s).

Return type pd.DataFrame

General Utils

<code>read_csv</code>	Read data from the specified CSV file and return a Woodwork DataTable
-----------------------	---

woodwork.utils.read_csv

`woodwork.utils.read_csv` (*filepath=None, name=None, index=None, time_index=None, semantic_tags=None, logical_types=None, use_standard_tags=True, **kwargs*)
 Read data from the specified CSV file and return a Woodwork DataTable

Parameters

- **filepath** (*str*) – A valid string path to the file to read
- **name** (*str, optional*) – Name used to identify the datatable.
- **index** (*str, optional*) – Name of the index column in the dataframe.
- **time_index** (*str, optional*) – Name of the time index column in the dataframe.
- **semantic_tags** (*dict, optional*) – Dictionary mapping column names in the dataframe to the semantic tags for the column. The keys in the dictionary should be strings that correspond to columns in the underlying dataframe. There are two options for specifying the dictionary values: (*str*): If only one semantic tag is being set, a single string can be used as a value. (*list[str]* or *set[str]*): If multiple tags are being set, a list or set of strings can be used as the value. Semantic tags will be set to an empty set for any column not included in the dictionary.
- **logical_types** (*dict[str -> LogicalType], optional*) – Dictionary mapping column names in the dataframe to the `LogicalType` for the column. `LogicalTypes` will be inferred for any columns not present in the dictionary.
- **use_standard_tags** (*bool, optional*) – If `True`, will add standard semantic tags to columns based on the inferred or specified logical type for the column. Defaults to `True`.
- ****kwargs** – Additional keyword arguments to pass to the underlying `pandas.read_csv` function. For more information on available keywords refer to the `pandas` documentation.

Returns DataTable created from the specified CSV file

Return type `woodwork.DataTable`

Demo Data

<code>load_retail</code> (<i>[id, nrows, return_dataframe]</i>)	Load a demo retail dataset into either a <code>DataTable</code> or a <code>DataFrame</code>
---	---

woodwork.demo.load_retail

`woodwork.demo.load_retail` (*id*='demo_retail_data', *nrows*=None, *return_dataframe*=False)

Load a demo retail dataset into either a DataTable or a DataFrame

Parameters

- **id** (*str*, *optional*) – The name to assign to the DataTable, if returning a DataTable. If not returning a DataTable, this will be ignored. Defaults to `demo_retail_data`.
- **nrows** (*int*, *optional*) – The number of rows to return in the dataset. If None, will return all possible rows. Defaults to None.
- **return_dataframe** (*bool*) – If True, will return a pandas DataFrame. If False, will return a Woodwork DataTable. Defaults to False.

Returns A DataFrame or DataTable containing the demo data.

Return type `pd.DataFrame` or `ww.DataTable`

1.1.6 Release Notes

v0.0.8 January 25, 2021

- **Enhancements**
 - Add `DataTable.df` property for accessing the underlying DataFrame (#470)
 - Set index of underlying DataFrame to match DataTable index (#464)
- **Fixes**
 - Sort underlying series when sorting dataframe (#468)
 - Allow setting indices to current index without side effects (#474)
- **Changes**
 - Fix release document with Github Actions link for CI (#462)
 - Don't allow registered LogicalTypes with the same name (#477)
 - Move `str_to_logical_type` to `TypeSystem` class (#482)
 - Remove `pyarrow` from core dependencies (#508)

Thanks to the following people for contributing to this release: @gsheni, @tamargrey, @thehomebrewnerd

v0.0.7 December 14, 2020

- **Enhancements**
 - Allow for user-defined logical types and inference functions in `TypeSystem` object (#424)
 - Add `__repr__` to `DataTable` (#425)
 - Allow initializing `DataColumn` with numpy array (#430)
 - Add `drop` to `DataTable` (#434)
 - Migrate CI tests to Github Actions (#417, #441, #451)
 - Add `metadata` to `DataColumn` for user-defined metadata (#447)
- **Fixes**
 - Update `DataColumn` name when using `setitem` on column with no name (#426)

- Don't allow pickle serialization for Koalas DataFrames (#432)
- Check DataTable metadata in equality check (#449)
- Propagate all attributes of DataTable in `_new_dt_including` (#454)

- **Changes**

- Update links to use alteryx org Github URL (#423)
- Support column names of any type allowed by the underlying DataFrame (#442)
- Use `object` dtype for LatLong columns for easy access to latitude and longitude values (#414)
- Restrict dask version to prevent 2020.12.0 release from being installed (#453)
- Lower minimum requirement for numpy to 1.15.4, and set pandas minimum requirement 1.1.1 (#459)

- **Testing Changes**

- Fix missing test coverage (#436)

Thanks to the following people for contributing to this release: @gsheni, @jeff-hernandez, @tamargrey, @the-homebrewnerd

v0.0.6 November 30, 2020

- **Enhancements**

- Add support for creating DataTable from Koalas DataFrame (#327)
- Add ability to initialize DataTable with numpy array (#367)
- Add `describe_dict` method to DataTable (#405)
- Add `mutual_information_dict` method to DataTable (#404)
- Add metadata to DataTable for user-defined metadata (#392)
- Add `update_dataframe` method to DataTable to update underlying DataFrame (#407)
- Sort dataframe if `time_index` is specified, bypass sorting with `already_sorted` parameter. (#410)
- Add `description` attribute to DataColumn (#416)
- Implement `DataColumn.__len__` and `DataTable.__len__` (#415)

- **Fixes**

- Rename `data_column.py` `datacolumn.py` (#386)
- Rename `data_table.py` `datatable.py` (#387)
- Rename `get_mutual_information` `mutual_information` (#390)

- **Changes**

- Lower moto test requirement for serialization/deserialization (#376)
- Make Koalas an optional dependency installable with `woodwork[koalas]` (#378)
- Remove WholeNumber LogicalType from Woodwork (#380)
- Updates to LogicalTypes to support Koalas 1.4.0 (#393)
- Replace `set_logical_types` and `set_semantic_tags` with just `set_types` (#379)
- Remove `copy_dataframe` parameter from DataTable initialization (#398)

- Implement `DataTable.__sizeof__` to return size of the underlying dataframe (#401)
- Include Datetime columns in mutual info calculation (#399)
- Maintain column order on DataTable operations (#406)
- **Testing Changes**
 - Add pyarrow, dask, and koalas to automated dependency checks (#388)
 - Use new version of pull request Github Action (#394)
 - Improve parameterization for `test_datatable_equality` (#409)

Thanks to the following people for contributing to this release: @ctduffy, @gsheni, @tamargrey, @thhomebrewnerd

Breaking Changes

- The `DataTable.set_semantic_tags` method was removed. `DataTable.set_types` can be used instead.
- The `DataTable.set_logical_types` method was removed. `DataTable.set_types` can be used instead.
- `WholeNumber` was removed from `LogicalTypes`. Columns that were previously inferred as `WholeNumber` will now be inferred as `Integer`.
- The `DataTable.get_mutual_information` was renamed to `DataTable.mutual_information`.
- The `copy_dataframe` parameter was removed from `DataTable` initialization.

v0.0.5 November 11, 2020

- **Enhancements**
 - Add `__eq__` to `DataTable` and `DataColumn` and update `LogicalType` equality (#318)
 - Add `value_counts()` method to `DataTable` (#342)
 - Support serialization and deserialization of `DataTables` via csv, pickle, or parquet (#293)
 - Add `shape` property to `DataTable` and `DataColumn` (#358)
 - Add `iloc` method to `DataTable` and `DataColumn` (#365)
 - Add `numeric_categorical_threshold` config value to allow inferring numeric columns as `Categorical` (#363)
 - Add `rename` method to `DataTable` (#367)
- **Fixes**
 - Catch non numeric time index at validation (#332)
- **Changes**
 - Support logical type inference from a Dask `DataFrame` (#248)
 - Fix validation checks and `make_index` to work with Dask `DataFrames` (#260)
 - Skip validation of `Ordinal` order values for Dask `DataFrames` (#270)
 - Improve support for datetimes with Dask input (#286)
 - Update `DataTable.describe` to work with Dask input (#296)
 - Update `DataTable.get_mutual_information` to work with Dask input (#300)

- Modify `to_pandas` function to return DataFrame with correct index (#281)
- Rename `DataColumn.to_pandas` method to `DataColumn.to_series` (#311)
- Rename `DataTable.to_pandas` method to `DataTable.to_dataframe` (#319)
- Remove `UserWarning` when no matching columns found (#325)
- Remove `copy` parameter from `DataTable.to_dataframe` and `DataColumn.to_series` (#338)
- Allow pandas ExtensionArrays as inputs to `DataColumn` (#343)
- Move warnings to a separate exceptions file and call via `UserWarning` subclasses (#348)
- Make Dask an optional dependency installable with `woodwork[dask]` (#357)

- **Documentation Changes**

- Create a guide for using Woodwork with Dask (#304)
- Add conda install instructions (#305, #309)
- Fix README.md badge with correct link (#314)
- Simplify issue templates to make them easier to use (#339)
- Remove extra output cell in Start notebook (#341)

- **Testing Changes**

- Parameterize numeric time index tests (#288)
- Add DockerHub credentials to CI testing environment (#326)
- Fix removing files for serialization test (#350)

Thanks to the following people for contributing to this release: @ctduffy, @gsheni, @tamargrey, @thhomebrewnerd

Breaking Changes

- The `DataColumn.to_pandas` method was renamed to `DataColumn.to_series`.
- The `DataTable.to_pandas` method was renamed to `DataTable.to_dataframe`.
- `copy` is no longer a parameter of `DataTable.to_dataframe` or `DataColumn.to_series`.

v0.0.4 October 21, 2020

- **Enhancements**

- Add optional `include` parameter for `DataTable.describe()` to filter results (#228)
- Add `make_index` parameter to `DataTable.__init__` to enable optional creation of a new index column (#238)
- Add support for setting ranking order on columns with Ordinal logical type (#240)
- Add `list_semantic_tags` function and CLI to get dataframe of woodwork semantic_tags (#244)
- Add support for numeric time index on `DataTable` (#267)
- Add `pop` method to `DataTable` (#289)
- Add entry point to `setup.py` to run CLI commands (#285)

- **Fixes**

- Allow numeric datetime time indices (#282)
- **Changes**
 - Remove redundant methods `DataTable.select_ltypes` and `DataTable.select_semantic_tags` (#239)
 - Make results of `get_mutual_information` more clear by sorting and removing self calculation (#247)
 - Lower minimum scikit-learn version to 0.21.3 (#297)
- **Documentation Changes**
 - Add guide for `dt.describe` and `dt.get_mutual_information` (#245)
 - Update README.md with documentation link (#261)
 - Add footer to doc pages with Alteryx Open Source (#258)
 - Add types and tags one-sentence definitions to Understanding Types and Tags guide (#271)
 - Add issue and pull request templates (#280, #284)
- **Testing Changes**
 - Add automated process to check latest dependencies. (#268)
 - Add test for setting a time index with specified string logical type (#279)

Thanks to the following people for contributing to this release: @ctduffy, @gsheni, @tamargrey, @thehomebrewnerd

v0.0.3 October 9, 2020

- **Enhancements**
 - Implement `setitem` on `DataTable` to create/overwrite an existing `DataColumn` (#165)
 - Add `to_pandas` method to `DataColumn` to access the underlying series (#169)
 - Add `list_logical_types` function and CLI to get dataframe of woodwork `LogicalTypes` (#172)
 - Add `describe` method to `DataTable` to generate statistics for the underlying data (#181)
 - Add optional `return_dataframe` parameter to `load_retail` to return either `DataFrame` or `DataTable` (#189)
 - Add `get_mutual_information` method to `DataTable` to generate mutual information between columns (#203)
 - Add `read_csv` function to create `DataTable` directly from CSV file (#222)
- **Fixes**
 - Fix bug causing incorrect values for quartiles in `DataTable.describe` method (#187)
 - Fix bug in `DataTable.describe` that could cause an error if certain semantic tags were applied improperly (#190)
 - Fix bug with instantiated `LogicalTypes` breaking when used with `issubclass` (#231)
- **Changes**
 - Remove unnecessary `add_standard_tags` attribute from `DataTable` (#171)
 - Remove standard tags from index column and do not return stats for index column from `DataTable.describe` (#196)

- Update `DataColumn.set_semantic_tags` and `DataColumn.add_semantic_tags` to return new objects (#205)
- Update various `DataTable` methods to return new objects rather than modifying in place (#210)
- Move `datetime_format` to `Datetime LogicalType` (#216)
- Do not calculate mutual info with index column in `DataTable.get_mutual_information` (#221)
- Move setting of underlying physical types from `DataTable` to `DataColumn` (#233)

- **Documentation Changes**

- Remove unused code from `sphinx conf.py`, update with Github URL (#160, #163)
- Update README and docs with new Woodwork logo, with better code snippets (#161, #159)
- Add `DataTable` and `DataColumn` to API Reference (#162)
- Add docstrings to `LogicalType` classes (#168)
- Add Woodwork image to index, clear outputs of Jupyter notebook in docs (#173)
- Update `contributing.md`, `release.md` with all instructions (#176)
- Add section for setting index and time index to start notebook (#179)
- Rename `changelog` to Release Notes (#193)
- Add section for standard tags to start notebook (#188)
- Add Understanding Types and Tags user guide (#201)
- Add missing docstring to `list_logical_types` (#202)
- Add Woodwork Global Configuration Options guide (#215)

- **Testing Changes**

- Add tests that confirm dtypes are as expected after `DataTable` init (#152)
- Remove unused `none_df` test fixture (#224)
- Add test for `LogicalType.__str__` method (#225)

Thanks to the following people for contributing to this release: @gsheni, @tamargrey, @thehomebrewnerd

v0.0.2 September 28, 2020

- **Fixes**

- Fix formatting issue when printing global config variables (#138)

- **Changes**

- Change `add_standard_tags` to `use_standard_Tags` to better describe behavior (#149)
- Change access of underlying dataframe to be through `to_pandas` with `._dataframe` field on class (#146)
- Remove `replace_none` parameter to `DataTables` (#146)

- **Documentation Changes**

- Add working code example to README and create Using Woodwork page (#103)

Thanks to the following people for contributing to this release: @gsheni, @tamargrey, @thehomebrewnerd

v0.1.0 September 24, 2020

- Add `natural_language_threshold` global config option used for Categorical/NaturalLanguage type inference (#135)
- Add global config options and add `datetime_format` option for type inference (#134)
- Fix bug with Integer and WholeNumber inference in column with `pd.NA` values (#133)
- Add `DataTable.ltypes` property to return series of logical types (#131)
- Add ability to create new datatable from specified columns with `dt[[columns]]` (#127)
- Handle setting and tagging of index and time index columns (#125)
- Add combined tag and ltype selection (#124)
- Add changelog, and update changelog check to CI (#123)
- Implement `reset_semantic_tags` (#118)
- Implement `DataTable.getitem` (#119)
- Add `remove_semantic_tags` method (#117)
- Add semantic tag selection (#106)
- Add github action, rename to woodwork (#113)
- Add license to `setup.py` (#112)
- Reset semantic tags on logical type change (#107)
- Add standard numeric and category tags (#100)
- Change `semantic_types` to `semantic_tags`, a set of strings (#100)
- Update dataframe dtypes based on logical types (#94)
- Add `select_logical_types` to `DataTable` (#96)
- Add pygments to `dev-requirements.txt` (#97)
- Add replacing `None` with `np.nan` in `DataTable` init (#87)
- Refactor `DataColumn` to make `semantic_types` and `logical_type` private (#86)
- Add `pandas_dtype` to each Logical Type, and remove `dtype` attribute on `DataColumn` (#85)
- Add `set_semantic_types` methods on both `DataTable` and `DataColumn` (#75)
- Support passing camel case or snake case strings for setting logical types (#74)
- Improve flexibility when setting semantic types (#72)
- Add Whole Number Inference of Logical Types (#66)
- Add `dtypes` property to `DataTables` and `repr` for `DataColumn` (#61)
- Allow specification of semantic types during `DataTable` creation (#69)
- Implements `set_logical_types` on `DataTable` (#65)
- Add init files to tests to fix code coverage (#60)
- Add `AutoAssign` bot (#59)
- Add logical types validation in `DataTables` (#49)
- Fix `working_directory` in CI (#57)
- Add `infer_logical_types` for `DataColumn` (#45)

- Fix README library name, and code coverage badge (#56, #56)
- Add code coverage (#51)
- Improve and refactor the validation checks during initialization of a DataTable (#40)
- Add dataframe attribute to DataTable (#39)
- Update README with minor usage details (#37)
- Add License (#34)
- Rename from datatables to datatables (#4)
- Add Logical Types, DataTable, DataColumn (#3)
- Add Makefile, setup.py, requirements.txt (#2)
- Initial Release (#1)

Thanks to the following people for contributing to this release: [@gsheni](#), [@tamargrey](#), [@thehomebrewnerd](#)

Symbols

- `__init__()` (*woodwork.datacolumn.DataColumn method*), 57
 - `__init__()` (*woodwork.datatable.DataTable method*), 48
 - `__init__()` (*woodwork.logical_types.Boolean method*), 61
 - `__init__()` (*woodwork.logical_types.Categorical method*), 62
 - `__init__()` (*woodwork.logical_types.CountryCode method*), 63
 - `__init__()` (*woodwork.logical_types.Datetime method*), 63
 - `__init__()` (*woodwork.logical_types.Double method*), 64
 - `__init__()` (*woodwork.logical_types.EmailAddress method*), 65
 - `__init__()` (*woodwork.logical_types.Filepath method*), 66
 - `__init__()` (*woodwork.logical_types.FullName method*), 67
 - `__init__()` (*woodwork.logical_types.IPAddress method*), 67
 - `__init__()` (*woodwork.logical_types.Integer method*), 65
 - `__init__()` (*woodwork.logical_types.LatLong method*), 68
 - `__init__()` (*woodwork.logical_types.NaturalLanguage method*), 69
 - `__init__()` (*woodwork.logical_types.Ordinal method*), 69
 - `__init__()` (*woodwork.logical_types.PhoneNumber method*), 70
 - `__init__()` (*woodwork.logical_types.SubRegionCode method*), 71
 - `__init__()` (*woodwork.logical_types.Timedelta method*), 71
 - `__init__()` (*woodwork.logical_types.URL method*), 72
 - `__init__()` (*woodwork.logical_types.ZIPCode method*), 73
 - `__init__()` (*woodwork.type_sys.type_system.TypeSystem method*), 74
- ## A
- `add_semantic_tags()` (*woodwork.datacolumn.DataColumn method*), 59
 - `add_semantic_tags()` (*woodwork.datatable.DataTable method*), 50
 - `add_type()` (*woodwork.type_sys.type_system.TypeSystem method*), 75
- ## B
- Boolean (*class in woodwork.logical_types*), 61
- ## C
- Categorical (*class in woodwork.logical_types*), 62
 - CountryCode (*class in woodwork.logical_types*), 63
- ## D
- DataColumn (*class in woodwork.datacolumn*), 57
 - DataTable (*class in woodwork.datatable*), 48
 - Datetime (*class in woodwork.logical_types*), 63
 - `describe()` (*woodwork.datatable.DataTable method*), 52
 - `describe_dict()` (*woodwork.datatable.DataTable method*), 53
 - Double (*class in woodwork.logical_types*), 64
 - `drop()` (*woodwork.datatable.DataTable method*), 57
- ## E
- EmailAddress (*class in woodwork.logical_types*), 65
- ## F
- Filepath (*class in woodwork.logical_types*), 66
 - FullName (*class in woodwork.logical_types*), 67
- ## H
- `head()` (*woodwork.datatable.DataTable method*), 56
- ## I
- `iiloc()` (*woodwork.datacolumn.DataColumn property*), 59

- `iloc()` (*woodwork.datatable.DataTable* property), 51
- `infer_logical_type()` (*woodwork.type_sys.type_system.TypeSystem* method), 75
- `Integer` (*class in woodwork.logical_types*), 65
- `IPAddress` (*class in woodwork.logical_types*), 67
- ## L
- `LatLng` (*class in woodwork.logical_types*), 68
- `list_logical_types()` (*in module woodwork.type_sys.utils*), 76
- `list_semantic_tags()` (*in module woodwork.type_sys.utils*), 76
- `load_retail()` (*in module woodwork.demo*), 78
- ## M
- `mutual_information()` (*woodwork.datatable.DataTable* method), 53
- `mutual_information_dict()` (*woodwork.datatable.DataTable* method), 54
- ## N
- `NaturalLanguage` (*class in woodwork.logical_types*), 69
- ## O
- `Ordinal` (*class in woodwork.logical_types*), 69
- ## P
- `PhoneNumber` (*class in woodwork.logical_types*), 70
- ## R
- `read_csv()` (*in module woodwork.utils*), 77
- `remove_semantic_tags()` (*woodwork.datacolumn.DataColumn* method), 59
- `remove_semantic_tags()` (*woodwork.datatable.DataTable* method), 50
- `remove_type()` (*woodwork.type_sys.type_system.TypeSystem* method), 75
- `rename()` (*woodwork.datatable.DataTable* method), 56
- `reset_semantic_tags()` (*woodwork.datacolumn.DataColumn* method), 59
- `reset_semantic_tags()` (*woodwork.datatable.DataTable* method), 51
- ## S
- `select()` (*woodwork.datatable.DataTable* method), 51
- `set_index()` (*woodwork.datatable.DataTable* method), 51
- `set_logical_type()` (*woodwork.datacolumn.DataColumn* method), 60
- `set_semantic_tags()` (*woodwork.datacolumn.DataColumn* method), 60
- `set_time_index()` (*woodwork.datatable.DataTable* method), 52
- `set_types()` (*woodwork.datatable.DataTable* method), 52
- `shape()` (*woodwork.datacolumn.DataColumn* property), 58
- `shape()` (*woodwork.datatable.DataTable* property), 50
- `SubRegionCode` (*class in woodwork.logical_types*), 71
- ## T
- `Timedelta` (*class in woodwork.logical_types*), 71
- `to_csv()` (*woodwork.datatable.DataTable* method), 55
- `to_dataframe()` (*woodwork.datatable.DataTable* method), 52
- `to_parquet()` (*woodwork.datatable.DataTable* method), 55
- `to_pickle()` (*woodwork.datatable.DataTable* method), 55
- `to_series()` (*woodwork.datacolumn.DataColumn* method), 60
- `TypeSystem` (*class in woodwork.type_sys.type_system*), 74
- ## U
- `update_dataframe()` (*woodwork.datatable.DataTable* method), 56
- `update_inference_function()` (*woodwork.type_sys.type_system.TypeSystem* method), 75
- `update_relationship()` (*woodwork.type_sys.type_system.TypeSystem* method), 76
- `URL` (*class in woodwork.logical_types*), 72
- ## V
- `value_counts()` (*woodwork.datatable.DataTable* method), 54
- ## Z
- `ZIPCode` (*class in woodwork.logical_types*), 73