XFrames Documentation

Release 0.1

Charles Hayden

Jun 05, 2017

Contents

Contents:

The XFrames library provides powerful and convenient abstractions used for operating on large datasets. It is implemented in python and runs on Spark.

The main classes that make up XFrames are:

CHAPTER 1

XFrame

class xframes.**XFrame**(*data=None*, *format='auto'*, *impl=None*, *verbose=False*)

A tabular, column-mutable dataframe object that can scale to big data. XFrame is able to hold data that are much larger than the machine's main memory. The data in XFrame is stored row-wise in a Spark RDD. Each row of the RDD is a list, whose elements correspond to the values in each column. The column names and types are stored in the XFrame instance, and give the mapping to the row list.

__init__(*data=None*, *format='auto'*, *impl=None*, *verbose=False*)

Construct a new XFrame from a url, a pandas.DataFrame or a Spark RDD or DataFrame.

An XFrame can be constructed from the following data formats: * csv file (comma separated value) * xframe directory archive (A directory where an XFrame was saved previously) * a spark RDD plus the column names and types * a spark.DataFrame * general text file (with csv parsing options, See `read_csv()`) * parquet file * a Python dictionary * pandas.DataFrame * JSON * Apache Avro

and from the following sources:

- your local file system
- the XFrame Server's file system
- HDFS
- Hive
- Amazon S3
- HTTP(S)

Only basic examples of construction are covered here. For more information and examples, please see the *User Guide*.

XFrames are immutable except for assignments to a column.

Parameters **data** : array | pandas.DataFrame | spark.rdd | spark.DataFrame | string | dict, optional

The actual interpretation of this field is dependent on the *format* parameter. If *data* is an array, Pandas DataFrame or Spark RDD, the contents are stored in the XFrame. If *data* is an object supporting iteritems, then is is handled like a dictionary. If *data* is an

object supporting iteration, then the values are iterated to form the XFrame. If *data* is a string, it is interpreted as a file. Files can be read from local file system or urls (hdfs://, s3://, or other Hadoop-supported file systems). To read files from s3, you must set the AWS_ACCESS_KEY_ID and AWS_SECRET_ACCESS_KEY environment variables, even if the file is publicly accessible.

format : string, optional

Format of the data. The default, "auto" will automatically infer the input data format. The inference rules are simple: If the data is an array or a dataframe, it is associated with 'array' and 'dataframe' respectively. If the data is a string, it is interpreted as a file, and the file extension is used to infer the file format. The explicit options are:

- "auto"
- "array"
- "dict"
- "xarray"
- "pandas.dataframe"
- "csv"
- "tsv"
- "psv"
- "parquet"
- "rdd"
- "spark.dataframe"
- "hive"
- "xframe"

verbose : bool, optional

If True, print the progress while reading a file.

See also:

xframes.XFrame.read_csv Create a new XFrame from a csv file. Preferred for text and CSV formats, because it has a lot more options for controlling the parser.

xframes.XFrame.read_parquet Read an XFrame from a parquet file.

xframes.XFrame.from_rdd Create a new XFrame from a Spark RDD or Spark DataFrame. Column names and types can be specified if a spark RDD is given; otherwise they are taken from the DataFrame.

xframes.XFrame.save Save an XFrame in a file for later use within XFrames or Spark.

xframes.XFrame.load Load an XFrame from a file. The filename extension is used to determine the file format.

xframes.XFrame.set_trace Controls entry and exit tracing.

xframes.XFrame.spark_context Returns the spark context.

xframes.XFrame.spark_sql_context Returns the spark sql context.

Notes

The following functionality is currently not implemented.

- pack_columns data types except list, array, and dict
- groupby quantile

Examples

Create an XFrame from a Python dictionary.

```
>>> from xframes import XFrame
>>> sf = XFrame({'id':[1,2,3], 'val':['A','B','C']})
>>> sf
Columns:
    id  int
    val str
Rows: 3
Data:
      id  val
   0  1   A
   1  2   B
   2  3   C
```

Create an XFrame from a remote CSV file.

```
>>> url = 'http://testdatasets.s3-website-us-west-2.amazonaws.com/users.csv.gz
↪'
>>> xf = XFrame.read_csv(url,
...     delimiter=',', header=True, comment_char="#",
...     column_type_hints={'user_id': int})
```

__getitem__(*key*)

This provides XFrame "indexing", for examle xf['column_name']. The type of the index determine what the construct does: electing a column, doing a logical filter, or returning one or more rows from the XFrame.

This method does things based on the type of *key*.
If *key* is:

- str Calls *select_column* on *key* to return a single column as an XArray.
- XArray Performs a logical filter. Expects given XArray to be the same length as all columns in current XFrame. Every row corresponding with an entry in the given XArray that is equivalent to False is filtered from the result.
- int Returns a single row of the XFrame (the 'key'th one) as a dictionary.
- slice Returns an XFrame including only the sliced rows.

Examples

```
>>> xf = xframes.XFrame({'id': [4, 6, 8], 'val': ['D', 'F', 'H']})
>>> xf
```

add_column(*col*, *name=''*)

Add a column to this XFrame. The length of the new column must match the length of the existing

XFrame. This operation returns a new XFrame with the additional columns. If no *name* is given, a default name is chosen.

Parameters **col** : *XArray*

The 'column' of data to add.

name : string, optional

The name of the column. If no name is given, a default name is chosen.

Returns *XFrame*

A new XFrame with the new column.

See also:

xframes.XFrame.add_columns Adds multiple columns.

Examples

```
>>> xf = xframes.XFrame({'id': [1, 2, 3], 'val': ['A', 'B', 'C']})
>>> xa = xframes.XArray(['cat', 'dog', 'fossa'])
>>> # This line is equivalant to `xf['species'] = xa`
>>> xf2 = xf.add_column(xa, name='species')
>>> xf2
+----+-----+---------+
| id | val | species |
+----+-----+---------+
| 1  |  A  |   cat   |
| 2  |  B  |   dog   |
| 3  |  C  |  fossa  |
+----+-----+---------+
[3 rows x 3 columns]
```

add_columns(*cols*, *names=None*)

Adds multiple columns to this XFrame. The length of the new columns must match the length of the existing XFrame. This operation returns a new XFrame with the additional columns.

Parameters **cols** : *XArray* or list of *XArray* or XFrame

The columns to add. If *cols* is an XFrame, all columns in it are added.

names : string or list of string, optional

If cols is an XArray, then the name of the column. If no name is given, a default name is chosen. If cols is a list of *XArray*, then a list of column names. All names must be specified. *Namelist* is ignored if *cols* is an XFrame. If there are columns with duplicate names, they will be made unambiguous by adding .1 to the second copy.

Returns *XFrame*

The XFrame with additional columns.

See also:

xframes.XFrame.add_column Adds one column

Examples

```
>>> xf = xframes.XFrame({'id': [1, 2, 3], 'val': ['A', 'B', 'C']})
>>> xa = xframes.XArray(['cat', 'dog', 'fossa'])
>>> # This line is equivalant to `xf['species'] = xa`
>>> xf2 = xf.add_columns(xa, names='species')
>>> xf2
+----+-----+---------+
| id | val | species |
+----+-----+---------+
| 1  |  A  |   cat   |
| 2  |  B  |   dog   |
| 3  |  C  |  fossa  |
+----+-----+---------+
[3 rows x 3 columns]
```

```
>>> xf = xframes.XFrame({'id': [1, 2, 3], 'val': ['A', 'B', 'C']})
>>> xf2 = xframes.XFrame({'species': ['cat', 'dog', 'horse'],
...                       'age': [3, 5, 9]})
>>> xf3 = xf.add_columns(xf2)
>>> xf3
+----+-----+-----+---------+
| id | val | age | species |
+----+-----+-----+---------+
| 1  |  A  |  3  |   cat   |
| 2  |  B  |  5  |   dog   |
| 3  |  C  |  9  |  horse  |
+----+-----+-----+---------+
[3 rows x 4 columns]
```

add_row_number(*column_name='id'*, *start=0*)

Returns a new XFrame with a new column that numbers each row sequentially. By default the count starts at 0, but this can be changed to a positive or negative number. The new column will be named with the given column name. An error will be raised if the given column name already exists in the XFrame.

Parameters **column_name** : str, optional

The name of the new column that will hold the row numbers.

start : int, optional

The number used to start the row number count.

Returns *XFrame*

The new XFrame with a column name

Notes

The range of numbers is constrained by a signed 64-bit integer, so beware of overflow if you think the results in the row number column will be greater than 9 quintillion.

Examples

```
>>> xf = xframes.XFrame({'a': [1, None, None], 'b': ['a', 'b', None]})
>>> xf.add_row_number()
+----+------+------+
| id |  a   |  b   |
+----+------+------+
| 0  |  1   |  a   |
| 1  | None |  b   |
| 2  | None | None |
+----+------+------+
[3 rows x 3 columns]
```

append (*other*)

Add the rows of an XFrame to the end of this XFrame.

Both XFrame must have the same set of columns with the same column names and column types.

Parameters **other** : *XFrame*

Another XFrame whose rows are appended to the current XFrame.

Returns *XFrame*

The result XFrame from the append operation.

Examples

```
>>> xf = xframes.XFrame({'id': [4, 6, 8], 'val': ['D', 'F', 'H']})
>>> xf2 = xframes.XFrame({'id': [1, 2, 3], 'val': ['A', 'B', 'C']})
>>> xf = xf.append(xf2)
+----+-----+
| 4  |  D  |
| 6  |  F  |
| 8  |  H  |
| 1  |  A  |
| 2  |  B  |
| 3  |  C  |
+----+-----+
[6 rows x 2 columns]
```

apply (*fn*, *dtype=None*, *use_columns=None*, *seed=None*)

Transform each row to an XArray according to a specified function. Returns a new XArray of *dtype* where each element in this XArray is transformed by *fn(x)* where *x* is a single row in the XFrame represented as a dictionary. The *fn* should return exactly one value which can be cast into type *dtype*. If *dtype* is not specified, the first 100 rows of the XFrame are used to make a guess of the target data type.

Parameters **fn** : function

The function to transform each row of the XFrame. The return type should be convertible to *dtype* if *dtype* is not None.

dtype : data type, optional

The *dtype* of the new XArray. If None, the first 100 elements of the array are used to guess the target data type.

use_columns : str | list[str], optional

The column or list of columns to be supplied in the row passed to the function. If not given, all columns wll be used to build the row.

seed : int, optional

Used as the seed if a random number generator is included in *fn*.

Returns *XArray*

The XArray transformed by fn. Each element of the XArray is of type *dtype*

Examples

Concatenate strings from several columns:

```
>>> xf = xframes.XFrame({'user_id': [1, 2, 3], 'movie_id': [3, 3, 6],
                         'rating': [4, 5, 1]})
>>> xf.apply(lambda x: str(x['user_id']) + str(x['movie_id']) + str(x['rating
↪']))
dtype: str
Rows: 3
['134', '235', '361']
```

column_names()

The name of each column in the XFrame.

Returns list[string]

Column names of the XFrame.

See also:

xframes.XFrame.rename Renames the columns.

column_types()

The type of each column in the XFrame.

Returns list[type]

Column types of the XFrame.

See also:

xframes.XFrame.dtype This is a synonym for column_types.

detect_type(*column_name*)

If the column is of string type, and the values can safely be cast to int or float, then return the type to be cast to. Uses the entire column to detect the type.

Parameters **column_name** : str

The name of the column to cast.

Returns type

int or float: The column can be cast to this type.

str: The column cannot be cast to one of the types above.

Examples

```
>>> xf = xpatterns.XFrame({'value': ['1', '2', '3']})
>>> xf.detect_type('value')
```

detect_type_and_cast (*column_name*)

If the column is of string type, and the values can all be interpreted as integer or float values, then cast the column to the numerical type. Otherwise, returns a copy of the XFrame.

Parameters **column_name** : str

The name of the column to cast.

Examples

```
>>> xf = xpatterns.XFrame({'value': ['1', '2', '3']})
>>> xf.detect_type_and_cast('value')
```

dropna (*columns=None*, *how='any'*)

Remove missing values from an XFrame. A missing value is either None or NaN. If *how* is 'any', a row will be removed if any of the columns in the *columns* parameter contains at least one missing value. If *how* is 'all', a row will be removed if all of the columns in the *columns* parameter are missing values.

If the *columns* parameter is not specified, the default is to consider all columns when searching for missing values.

Parameters **columns** : list or str, optional

The columns to use when looking for missing values. By default, all columns are used.

how : {'any', 'all'}, optional

Specifies whether a row should be dropped if at least one column has missing values, or if all columns have missing values. 'any' is default.

Returns *XFrame*

XFrame with missing values removed (according to the given rules).

See also:

xframes.XFrame.dropna_split Drops missing rows from the XFrame and returns them.

Examples

Drop all missing values.

```
>>> xf = xframes.XFrame({'a': [1, None, None], 'b': ['a', 'b', None]})
>>> xf.dropna()
+---+---+
| a | b |
+---+---+
| 1 | a |
+---+---+
[1 rows x 2 columns]
```

Drop rows where every value is missing.

```
>>> xf.dropna(any="all")
+------+---+
|  a   | b |
+------+---+
|  1   | a |
| None | b |
+------+---+
[2 rows x 2 columns]
```

Drop rows where column 'a' has a missing value.

```
>>> xf.dropna('a', any="all")
+---+---+
| a | b |
+---+---+
| 1 | a |
+---+---+
[1 rows x 2 columns]
```

dropna_split(*columns=None*, *how='any'*)

Split rows with missing values from this XFrame. This function has the same functionality as *dropna()*, but returns a tuple of two XFrames. The first item is the expected output from *dropna()*, and the second item contains all the rows filtered out by the *dropna* algorithm.

Parameters **columns** : list or str, optional

The columns to use when looking for missing values. By default, all columns are used.

how : {'any', 'all'}, optional

Specifies whether a row should be dropped if at least one column has missing values, or if all columns have missing values. 'any' is default.

Returns (*XFrame*, *XFrame*)

(XFrame with missing values removed, XFrame with the removed missing values)

See also:

xframes.XFrame.dropna

Examples

```
>>> xf = xframes.XFrame({'a': [1, None, None], 'b': ['a', 'b', None]})
>>> good, bad = xf.dropna_split()
>>> good
+---+---+
| a | b |
+---+---+
| 1 | a |
+---+---+
[1 rows x 2 columns]
```

```
>>> bad
+------+------+
|  a   |  b   |
+------+------+
| None |  b   |
```

```
| None | None |
+------+------+
[2 rows x 2 columns]
```

dtype()
The type of each column in the XFrame.

Returns list[type]

Column types of the XFrame.

See also:

xframes.XFrame.column_types This is a synonym for dtype.

dump_debug_info()
Print information about the Spark RDD associated with this XFrame.

classmethod empty(*column_names*, *column_types*)
Create an empty XFrame.

Creates an empty XFrame, with column names and column types.

Parameters **column_names** : list[str]

The column names.

column_types : list[type]

The column types.

Returns *XFrame*

An empty XFrame with the given column names and types.

fillna(*column*, *value*)
Fill all missing values with a given value in a given column. If the *value* is not the same type as the values in *column*, this method attempts to convert the value to the original column's type. If this fails, an error is raised.

Parameters **column** : str

The name of the column to modify.

value : type convertible to XArray's type

The value used to replace all missing values.

Returns *XFrame*

A new XFrame with the specified value in place of missing values.

See also:

xframes.XFrame.dropna

Examples

```
>>> xf = xframes.XFrame({'a':[1, None, None],
...                      'b':['13.1', '17.2', None]})
>>> xf = xf.fillna('a', 0)
>>> xf
```

```
+---+------+
| a |  b   |
+---+------+
| 1 | 13.1 |
| 0 | 17.2 |
| 0 | None |
+---+------+
[3 rows x 2 columns]
```

filterby(*values*, *column_name*, *exclude=False*)

Filter an XFrame by values inside an iterable object. Result is an XFrame that only includes (or excludes) the rows that have a column with the given *column_name* which holds one of the values in the given *values* XArray. If *values* is not an XArray, we attempt to convert it to one before filtering.

Parameters **values** : *XArray* | list |tuple | set | iterable | numpy.ndarray | pandas.Series | str | function

The values to use to filter the XFrame. The resulting XFrame will only include rows that have one of these values in the given column. If this is f function, it is called on each row and is passed the value in the column given by 'column_name'. The result includes rows where the function returns True.

column_name : str | None

The column of the XFrame to match with the given *values*. This can only be None if the values argument is a function. In this case, the function is passed the whole row.

exclude : bool

If True, the result XFrame will contain all rows EXCEPT those that have one of *values* in *column_name*.

Returns *XFrame*

The filtered XFrame.

Examples

```
>>> xf = xframes.XFrame({'id': [1, 2, 3, 4],
...                      'animal_type': ['dog', 'cat', 'cow', 'horse'],
...                      'name': ['bob', 'jim', 'jimbob', 'bobjim']})
>>> household_pets = ['cat', 'hamster', 'dog', 'fish', 'bird', 'snake']
>>> xf.filterby(household_pets, 'animal_type')
+-------------+----+------+
| animal_type | id | name |
+-------------+----+------+
|     dog     | 1  | bob  |
|     cat     | 2  | jim  |
+-------------+----+------+
[2 rows x 3 columns]
>>> xf.filterby(household_pets, 'animal_type', exclude=True)
+-------------+----+--------+
| animal_type | id |  name  |
+-------------+----+--------+
|    horse    | 4  | bobjim |
|     cow     | 3  | jimbob |
+-------------+----+--------+
[2 rows x 3 columns]
```

flat_map (*column_names*, *fn*, *column_types='auto'*, *use_columns=None*, *seed=None*)

Map each row of the XFrame to multiple rows in a new XFrame via a function.

The output of *fn* must have type `list[list[...]]`. Each inner list will be a single row in the new output, and the collection of these rows within the outer list make up the data for the output XFrame. All rows must have the same length and the same order of types to make sure the result columns are homogeneously typed. For example, if the first element emitted into the outer list by *fn* is `[43, 2.3, 'string']`, then all other elements emitted into the outer list must be a list with three elements, where the first is an *int*, second is a *float*, and third is a *string*. If *column_types* is not specified, the first 10 rows of the XFrame are used to determine the column types of the returned XFrame.

Parameters **column_names** : list[str]

The column names for the returned XFrame.

fn : function

The function that maps each of the XFrame rows into multiple rows, returning `list[list[...]]`. All output rows must have the same length and order of types. The function is passed a dictionary of column name: value for each row.

column_types : list[type], optional

The column types of the output XFrame. Default value will be automatically inferred by running *fn* on the first 10 rows of the output.

use_columns : str | list[str], optional

The column or list of columns to be supplied in the row passed to the function. If not given, all columns wll be used to build the row.

seed : int, optional

Used as the seed if a random number generator is included in *fn*.

Returns *XFrame*

A new XFrame containing the results of the `flat_map` of the original XFrame.

Examples

Repeat each row according to the value in the 'number' column.

```
>>> xf = xframes.XFrame({'letter': ['a', 'b', 'c'],
...                      'number': [1, 2, 3]})
>>> xf.flat_map(['number', 'letter'],
...             lambda x: [list(x.itervalues()) for _ in range(0, x['number
↪'])])
+--------+--------+
| number | letter |
+--------+--------+
|   1    |   a    |
|   2    |   b    |
|   2    |   b    |
|   3    |   c    |
|   3    |   c    |
|   3    |   c    |
+--------+--------+
[6 rows x 2 columns]
```

foreach(*row_fn*, *init_fn=None*, *final_fn=None*, *use_columns=None*, *seed=None*)

Apply the given function to each row of a XFrame. This is intended to be used for functions with side effects.

Rows are processed in groups. Each group is processed sequentially in one execution context. An initial funciton, if given, is executed forst for each group. Its results are passed to each row function. The row function receives the row data as a dictionary of column name: column value.

Parameters **row_fn** : function

The function to be applied to each row of the XFrame. Any value that is returned is ignored. The row_fn takes two parameters: row and init. The row is a dictionary of column-name: column_value. The init value is returned by init_fn.

init_fn : function, optional

The function to be applied before row_fn is called. The rows are processed in groups: init_fn is called once for each group. If no init_fn is supplied, the row_fn is passed None as its second parameter. Init_fn takes no parameters.

final_fn : function, optional

The function to be applied after all row_fn calls are made. Final_fn takes one parameter, the value returned by the init_fn.

use_columns : str | list[str], optional

The column or list of columns to be supplied in the row passed to the function. If not given, all columns wll be used to build the row.

seed : int, optional

Used as the seed if a random number generator is included in *fn*.

Examples

Send rows to an external sink.

```
>>> xf = xframes.XFrame({'user_id': [1, 2, 3], 'movie_id': [3, 3, 6],
                         'rating': [4, 5, 1]})
>>> xf.foreach(lambda row, ini: send(row['user_id'], row['movie_id'], row[
↪'rating']))
```

Send rows to an external sink with modification.

```
>>> xf = xframes.XFrame({'user_id': [1, 2, 3], 'movie_id': [3, 3, 6],
                         'rating': [4, 5, 1]})
>>> xf.foreach(lambda row, bias: send(row['user_id'], row['movie_id'], row[
↪'rating'] + bias),
               lambda: 10)
```

classmethod **from_rdd**(*rdd*, *column_names=None*, *column_types=None*)

Create a XFrame from a spark RDD or spark DataFrame. The data should be: * an RDD of tuples * Each tuple should be of the same length. * Each "column" should be of a uniform type.

Parameters **rdd: spark.RDD or spark.DataFrame**

Data used to populate the XFrame

column_names : list of string, optional

The column names to use. Ignored for Spark DataFrames.

column_types : list of type, optional

The column types to use. Ignored for Spark DataFrames.

Returns *XFrame*

See also:

to_rdd Converts to a Spark RDD.

classmethod from_xarray(*arry*, *name*)

Constructs a one column XFrame from an XArray and a column name.

Parameters **arry** : *XArray*

The XArray that will become an XFrame of one column.

name: str

The column name.

Returns out: *XFrame*

Returns an XFrame with one column, containing the values in arry and with the given name.

Examples

Create an XFrame from an XArray.

```
>>> print XFrame.from_xarray(XArray([1, 2, 3]), 'name')
```

name |

1 |

2 |

3 |

groupby(*key_columns*, *operations=None*, **args*)

Perform a group on the *key_columns* followed by aggregations on the columns listed in *operations*.

The *operations* parameter is a dictionary that indicates which aggregation operators to use and which columns to use them on. The available operators are SUM, MAX, MIN, COUNT, MEAN, VARIANCE, STD, CONCAT, SELECT_ONE, ARGMIN, ARGMAX, and QUANTILE. See *aggregate* for more detail on the aggregators.

Parameters **key_columns** : string | list[string]

Column(s) to group by. Key columns can be of any type other than dictionary.

operations : dict, list, optional

Dictionary of columns and aggregation operations. Each key is a output column name and each value is an aggregator. This can also be a list of aggregators, in which case column names will be automatically assigned.

***args**

All other remaining arguments will be interpreted in the same way as the operations argument.

Returns **out_xf** : *XFrame*

A new XFrame, with a column for each groupby column and each aggregation operation.

See also:

xframes.aggregate

Examples

Suppose we have an XFrame with movie ratings by many users.

```
>>> import xframes.aggregate as agg
>>> url = 'http://atg-testdata/rating.csv'
>>> xf = xframes.XFrame.read_csv(url)
>>> xf
+---------+----------+--------+
| user_id | movie_id | rating |
+---------+----------+--------+
|  25904  |   1663   |   3    |
|  25907  |   1663   |   3    |
|  25923  |   1663   |   3    |
|  25924  |   1663   |   3    |
|  25928  |   1663   |   2    |
|  25933  |   1663   |   4    |
|  25934  |   1663   |   4    |
|  25935  |   1663   |   4    |
|  25936  |   1663   |   5    |
|  25937  |   1663   |   2    |
|   ...   |   ...    |  ...   |
+---------+----------+--------+
[10000 rows x 3 columns]
```

Compute the number of occurrences of each user.

```
>>> user_count = xf.groupby('user_id',
...                         {'count': agg.COUNT()})
>>> user_count
+---------+-------+
| user_id | count |
+---------+-------+
|  62361  |   1   |
|  30727  |   1   |
|  40111  |   1   |
|  50513  |   1   |
|  35140  |   1   |
```

```
|  42352  |   1   |
|  29667  |   1   |
|  46242  |   1   |
|  58310  |   1   |
|  64614  |   1   |
|   ...   |  ...  |
+---------+-------+
[9852 rows x 2 columns]
```

Compute the mean and standard deviation of ratings per user.

```
>>> user_rating_stats = xf.groupby('user_id',
...                                {
...                                    'mean_rating': agg.MEAN('rating'),
...                                    'std_rating': agg.STD('rating')
...                                })
>>> user_rating_stats
+---------+-------------+------------+
| user_id | mean_rating | std_rating |
+---------+-------------+------------+
|  62361  |     5.0     |    0.0     |
|  30727  |     4.0     |    0.0     |
|  40111  |     2.0     |    0.0     |
|  50513  |     4.0     |    0.0     |
|  35140  |     4.0     |    0.0     |
|  42352  |     5.0     |    0.0     |
|  29667  |     4.0     |    0.0     |
|  46242  |     5.0     |    0.0     |
|  58310  |     2.0     |    0.0     |
|  64614  |     2.0     |    0.0     |
|   ...   |     ...     |    ...     |
+---------+-------------+------------+
[9852 rows x 3 columns]
```

Compute the movie with the minimum rating per user.

```
>>> chosen_movies = xf.groupby('user_id',
...                            {
...                                'worst_movies': agg.ARGMIN('rating','movie_
↪id')
...                            })
>>> chosen_movies
+---------+-------------+
| user_id | worst_movies |
+---------+-------------+
|  62361  |     1663    |
|  30727  |     1663    |
|  40111  |     1663    |
|  50513  |     1663    |
|  35140  |     1663    |
|  42352  |     1663    |
|  29667  |     1663    |
|  46242  |     1663    |
|  58310  |     1663    |
|  64614  |     1663    |
|   ...   |     ...     |
+---------+-------------+
[9852 rows x 2 columns]
```

Compute the movie with the max rating per user and also the movie with the maximum imdb-ranking per user.

```
>>> xf['imdb-ranking'] = xf['rating'] * 10
>>> chosen_movies = xf.groupby('user_id',
...          {('max_rating_movie','max_imdb_ranking_movie'):
...             agg.ARGMAX(('rating','imdb-ranking'),'movie_id')})
>>> chosen_movies
+---------+------------------+-------------------------+
| user_id | max_rating_movie | max_imdb_ranking_movie |
+---------+------------------+-------------------------+
|  62361  |       1663       |          16630          |
|  30727  |       1663       |          16630          |
|  40111  |       1663       |          16630          |
|  50513  |       1663       |          16630          |
|  35140  |       1663       |          16630          |
|  42352  |       1663       |          16630          |
|  29667  |       1663       |          16630          |
|  46242  |       1663       |          16630          |
|  58310  |       1663       |          16630          |
|  64614  |       1663       |          16630          |
|   ...   |       ...        |           ...           |
+---------+------------------+-------------------------+
[9852 rows x 3 columns]
```

Compute the movie with the max rating per user.

```
>>> chosen_movies = xf.groupby('user_id',
...          {'best_movies': agg.ARGMAX('rating','movie')})
```

Compute the movie with the max rating per user and also the movie with the maximum imdb-ranking per user.

```
>>> chosen_movies = xf.groupby('user_id',
...         {('max_rating_movie','max_imdb_ranking_movie'):
...                              agg.ARGMAX(('rating','imdb-ranking'),'movie
↪')})
```

Compute the count, mean, and standard deviation of ratings per (user, time), automatically assigning output column names.

```
>>> xf['time'] = xf.apply(lambda x: (x['user_id'] + x['movie_id']) % 11 +␣
↪2000)
>>> user_rating_stats = xf.groupby(['user_id', 'time'],
...                                [agg.COUNT(),
...                                 agg.MEAN('rating'),
...                                 agg.STDV('rating')])
>>> user_rating_stats
+------+---------+-------+---------------+----------------+
| time | user_id | Count | Avg of rating | Stdv of rating |
+------+---------+-------+---------------+----------------+
| 2006 |  61285  |   1   |      4.0      |      0.0       |
| 2000 |  36078  |   1   |      4.0      |      0.0       |
| 2003 |  47158  |   1   |      3.0      |      0.0       |
| 2007 |  34446  |   1   |      3.0      |      0.0       |
| 2010 |  47990  |   1   |      3.0      |      0.0       |
| 2003 |  42120  |   1   |      5.0      |      0.0       |
| 2007 |  44940  |   1   |      4.0      |      0.0       |
```

```
| 2008 |  58240  |   1   |      4.0      |      0.0       |
| 2002 |   102   |   1   |      1.0      |      0.0       |
| 2009 |  52708  |   1   |      3.0      |      0.0       |
| ...  |   ...   |  ...  |      ...      |      ...       |
+------+---------+-------+---------------+----------------+
[10000 rows x 5 columns]
```

The groupby function can take a variable length list of aggregation specifiers so if we want the count and the 0.25 and 0.75 quantiles of ratings:

```
>>> user_rating_stats = xf.groupby(['user_id', 'time'], agg.COUNT(),
...                                {'rating_quantiles': agg.QUANTILE('rating',
↪[0.25, 0.75])})
>>> user_rating_stats
+------+---------+-------+------------------------+
| time | user_id | Count |    rating_quantiles    |
+------+---------+-------+------------------------+
| 2006 |  61285  |   1   | array('d', [4.0, 4.0]) |
| 2000 |  36078  |   1   | array('d', [4.0, 4.0]) |
| 2003 |  47158  |   1   | array('d', [3.0, 3.0]) |
| 2007 |  34446  |   1   | array('d', [3.0, 3.0]) |
| 2010 |  47990  |   1   | array('d', [3.0, 3.0]) |
| 2003 |  42120  |   1   | array('d', [5.0, 5.0]) |
| 2007 |  44940  |   1   | array('d', [4.0, 4.0]) |
| 2008 |  58240  |   1   | array('d', [4.0, 4.0]) |
| 2002 |   102   |   1   | array('d', [1.0, 1.0]) |
| 2009 |  52708  |   1   | array('d', [3.0, 3.0]) |
| ...  |   ...   |  ...  |          ...           |
+------+---------+-------+------------------------+
[10000 rows x 4 columns]
```

To put all items a user rated into one list value by their star rating:

```
>>> user_rating_stats = xf.groupby(["user_id", "rating"],
...                                {"rated_movie_ids": agg.CONCAT("movie_id")}
↪)
>>> user_rating_stats
+--------+---------+----------------------+
| rating | user_id |     rated_movie_ids  |
+--------+---------+----------------------+
|   3    |  31434  | array('d', [1663.0]) |
|   5    |  25944  | array('d', [1663.0]) |
|   4    |  38827  | array('d', [1663.0]) |
|   4    |  51437  | array('d', [1663.0]) |
|   4    |  42549  | array('d', [1663.0]) |
|   4    |  49532  | array('d', [1663.0]) |
|   3    |  26124  | array('d', [1663.0]) |
|   4    |  46336  | array('d', [1663.0]) |
|   4    |  52133  | array('d', [1663.0]) |
|   5    |  62361  | array('d', [1663.0]) |
|  ...   |   ...   |          ...         |
+--------+---------+----------------------+
[9952 rows x 3 columns]
```

To put all items and rating of a given user together into a dictionary value:

```
>>> user_rating_stats = xf.groupby("user_id",
...                                {"movie_rating": agg.CONCAT("movie_id",
↪"rating")})
```

```
>>> user_rating_stats
+---------+--------------+
| user_id | movie_rating |
+---------+--------------+
|  62361  |  {1663: 5}   |
|  30727  |  {1663: 4}   |
|  40111  |  {1663: 2}   |
|  50513  |  {1663: 4}   |
|  35140  |  {1663: 4}   |
|  42352  |  {1663: 5}   |
|  29667  |  {1663: 4}   |
|  46242  |  {1663: 5}   |
|  58310  |  {1663: 2}   |
|  64614  |  {1663: 2}   |
|   ...   |     ...      |
+---------+--------------+
[9852 rows x 2 columns]
```

head(*n=10*)

The first n rows of the XFrame.

Parameters **n** : int, optional

The number of rows to fetch.

Returns *XFrame*

A new XFrame which contains the first n rows of the current XFrame

See also:

xframes.XFrame.tail Returns the last part of the XFrame.

xframes.XFrame.print_rows Prints the XFrame.

join(*right*, *on=None*, *how='inner'*)

Merge two XFrames. Merges the current (left) XFrame with the given (right) XFrame using a SQL-style equi-join operation by columns.

Parameters **right** : *XFrame*

The XFrame to join.

on : str | list | dict, optional

The column name(s) representing the set of join keys. Each row that has the same value in this set of columns will be merged together.

- If *on* is not given, the join keyd are all columns in the left and right XFrames that have the same name
- If a string is given, this is interpreted as a join using one column, where both XFrames have the same column name.
- If a list is given, this is interpreted as a join using one or more column names, where each column name given exists in both XFrames.
- If a dict is given, each dict key is taken as a column name in the left XFrame, and each dict value is taken as the column name in right XFrame that will be joined together. e.g. {'left_column_name':'right_column_name'}.

how : {'inner', 'left', 'right', 'outer', 'full'}, optional

The type of join to perform. 'inner' is default.

- inner: Equivalent to a SQL inner join. Result consists of the rows from the two frames whose join key values match exactly, merged together into one XFrame.
- left: Equivalent to a SQL left outer join. Result is the union between the result of an inner join and the rest of the rows from the left XFrame, merged with missing values.
- right: Equivalent to a SQL right outer join. Result is the union between the result of an inner join and the rest of the rows from the right XFrame, merged with missing values.
- full: Equivalent to a SQL full outer join. Result is the union between the result of a left outer join and a right outer join.
- cartesian: Cartesian product of left and right tables, with columns from each. There is no common column matching: the resulting number of rows is the product of the row counts of the left and right XFrames.

Returns *XFrame*

The joined XFrames.

Examples

```
>>> animals = xframes.XFrame({'id': [1, 2, 3, 4],
...                           'name': ['dog', 'cat', 'sheep', 'cow']})
>>> sounds = xframes.XFrame({'id': [1, 3, 4, 5],
...                          'sound': ['woof', 'baa', 'moo', 'oink']})
>>> animals.join(sounds, how='inner')
+----+-------+-------+
| id |  name | sound |
+----+-------+-------+
| 1  |  dog  |  woof |
| 3  | sheep |  baa  |
| 4  |  cow  |  moo  |
+----+-------+-------+
[3 rows x 3 columns]
```

```
>>> animals.join(sounds, on='id', how='left')
+----+-------+-------+
| id |  name | sound |
+----+-------+-------+
| 1  |  dog  |  woof |
| 3  | sheep |  baa  |
| 4  |  cow  |  moo  |
| 2  |  cat  |  None |
+----+-------+-------+
[4 rows x 3 columns]
```

```
>>> animals.join(sounds, on=['id'], how='right')
+----+-------+-------+
| id |  name | sound |
+----+-------+-------+
| 1  |  dog  |  woof |
| 3  | sheep |  baa  |
| 4  |  cow  |  moo  |
| 5  |  None |  oink |
```

```
+----+-------+-------+
[4 rows x 3 columns]
```

```
>>> animals.join(sounds, on={'id':'id'}, how='full')
+----+-------+-------+
| id |  name | sound |
+----+-------+-------+
| 1  |  dog  |  woof |
| 3  | sheep |  baa  |
| 4  |  cow  |  moo  |
| 5  |  None |  oink |
| 2  |  cat  |  None |
+----+-------+-------+
[5 rows x 3 columns]
```

lineage()

The table lineage: the files that went into building this table.

Returns dict

- **key 'table': set[filename]** The files that were used to build the XArray
- **key 'column': dict{column_name: set[filename]}** The set of files that were used to build each column

classmethod load(*filename*)

Load an XFrame. The filename extension is used to determine the format automatically. This function is particularly useful for XFrames previously saved in binary format. For CSV imports the *read_csv()* function provides greater control. If the XFrame is in binary format, *filename* is actually a directory, created when the XFrame is saved.

Parameters **filename** : string

Location of the file to load. Can be a local path or a remote URL.

Returns *XFrame*

See also:

xframes.XFrame.save Saves the XFrame to a file.

xframes.XFrame.read_csv Allows more control over csv parsing.

Examples

```
>>> sf = xframes.XFrame({'id':[1,2,3], 'val':['A','B','C']})
>>> sf.save('my_xframe')        # 'my_xframe' is a directory
>>> sf_loaded = xframes.XFrame.load('my_xframe')
```

num_columns()

The number of columns in this XFrame.

Returns int

Number of columns in the XFrame.

See also:

xframes.XFrame.num_rows Returns the number of rows.

num_rows()

The number of rows in this XFrame.

Returns int

Number of rows in the XFrame.

See also:

xframes.XFrame.num_columns Returns the number of columns.

pack_columns(*columns=None*, *column_prefix=None*, *dtype=<type 'list'>*, *fill_na=None*, *remove_prefix=True*, *new_column_name=None*)

Pack two or more columns of the current XFrame into one single column. The result is a new XFrame with the unaffected columns from the original XFrame plus the newly created column.

The list of columns that are packed is chosen through either the *columns* or *column_prefix* parameter. Only one of the parameters is allowed to be provided: *columns* explicitly specifies the list of columns to pack, while *column_prefix* specifies that all columns that have the given prefix are to be packed.

The type of the resulting column is decided by the *dtype* parameter. Allowed values for *dtype* are dict, array.array list, and tuple:

- dict: pack to a dictionary XArray where column name becomes dictionary key and column value becomes dictionary value
- array.array: pack all values from the packing columns into an array
- list: pack all values from the packing columns into a list.
- tuple: pack all values from the packing columns into a tuple.

Parameters **columns** : list[str], optional

A list of column names to be packed. There needs to have at least two columns to pack. If omitted and *column_prefix* is not specified, all columns from current XFrame are packed. This parameter is mutually exclusive with the *column_prefix* parameter.

column_prefix : str, optional

Pack all columns with the given *column_prefix*. This parameter is mutually exclusive with the *columns* parameter.

dtype : dict | array.array | list | tuple, optional

The resulting packed column type. If not provided, dtype is list.

fill_na : value, optional

Value to fill into packed column if missing value is encountered. If packing to dictionary, *fill_na* is only applicable to dictionary values; missing keys are not replaced.

remove_prefix : bool, optional

If True and *column_prefix* is specified, the dictionary key will be constructed by removing the prefix from the column name. This option is only applicable when packing to dict type.

new_column_name : str, optional

Packed column name. If not given and *column_prefix* is given, then the prefix will be used as the new column name, otherwise name is generated automatically.

Returns *XFrame*

An XFrame that contains columns that are not packed, plus the newly packed column.

See also:

xframes.XFrame.unpack

Notes

- There must be at least two columns to pack.
- If packing to dictionary, a missing key is always dropped. Missing values are dropped if *fill_na* is not provided, otherwise, missing value is replaced by *fill_na*. If packing to list or array, missing values will be kept. If *fill_na* is provided, the missing value is replaced with *fill_na* value.

Examples

Suppose 'xf' is an an XFrame that maintains business category information.

```
>>> xf = xframes.XFrame({'business': range(1, 5),
...                      'category.retail': [1, None, 1, None],
...                      'category.food': [1, 1, None, None],
...                      'category.service': [None, 1, 1, None],
...                      'category.shop': [1, 1, None, 1]})
```

```
>>> xf
+----------+----------------+---------------+-----------------+-------------
↪--+
| business | category.retail | category.food | category.service | category.
↪shop |
+----------+----------------+---------------+-----------------+-------------
↪--+
|    1     |        1       |       1       |       None       |       1     ␣
↪   |
|    2     |       None     |       1       |        1         |       1     ␣
↪   |
|    3     |        1       |      None     |        1         |      None   ␣
↪   |
|    4     |       None     |       1       |       None       |       1     ␣
↪   |
+----------+----------------+---------------+-----------------+-------------
↪--+
[4 rows x 5 columns]
```

To pack all category columns into a list:

```
>>> xf.pack_columns(column_prefix='category')
+----------+--------------------+
| business |         X2         |
+----------+--------------------+
|    1     |  [1, 1, None, 1]   |
|    2     |  [None, 1, 1, 1]   |
|    3     | [1, None, 1, None] |
|    4     | [None, 1, None, 1] |
+----------+--------------------+
[4 rows x 2 columns]
```

To pack all category columns into a dictionary, with new column name:

```
>>> xf.pack_columns(column_prefix='category', dtype=dict,
...                 new_column_name='category')
+----------+-------------------------------+
| business |            category           |
+----------+-------------------------------+
|    1     | {'food': 1, 'shop': 1, 're ... |
|    2     | {'food': 1, 'shop': 1, 'se ... |
|    3     |  {'retail': 1, 'service': 1}   |
|    4     |     {'food': 1, 'shop': 1}     |
+----------+-------------------------------+
[4 rows x 2 columns]
```

To keep column prefix in the resulting dict key:

```
>>> xf.pack_columns(column_prefix='category', dtype=dict,
...                 remove_prefix=False)
+----------+-------------------------------+
| business |               X2              |
+----------+-------------------------------+
|    1     | {'category.retail': 1, 'ca ... |
|    2     | {'category.food': 1, 'cate ... |
|    3     | {'category.retail': 1, 'ca ... |
|    4     | {'category.food': 1, 'cate ... |
+----------+-------------------------------+
[4 rows x 2 columns]
```

To explicitly pack a set of columns:

```
>>> xf.pack_columns(columns = ['business', 'category.retail',
...                            'category.food', 'category.service',
...                            'category.shop'])
+-----------------------+
|           X1          |
+-----------------------+
|   [1, 1, 1, None, 1]  |
|   [2, None, 1, 1, 1]  |
| [3, 1, None, 1, None] |
| [4, None, 1, None, 1] |
+-----------------------+
[4 rows x 1 columns]
```

To pack all columns with name starting with 'category' into an array type, and with missing value replaced with 0:

```
>>> xf.pack_columns(column_prefix="category", dtype=array.array,
...                 fill_na=0)
+----------+-------------------------------+
| business |               X2              |
+----------+-------------------------------+
|    1     | array('d', [1.0, 1.0, 0.0, ... |
|    2     | array('d', [0.0, 1.0, 1.0, ... |
|    3     | array('d', [1.0, 0.0, 1.0, ... |
|    4     | array('d', [0.0, 1.0, 0.0, ... |
+----------+-------------------------------+
[4 rows x 2 columns]
```

persist (*persist_flag*)

Persist or unpersist the underlying data storage object.

Persisting makes a copy of the object on the disk, so that it does not have to be recomputed in times of low memory. Unpersisting frees up this space.

Parameters **persist_flag** : boolean

If True, peersist the object. If False, unpersist it.

print_rows(*num_rows=10*, *num_columns=40*, *max_column_width=30*, *max_row_width=None*, *wrap_text=False*, *max_wrap_rows=2*, *footer=True*)

Print the first rows and columns of the XFrame in human readable format.

Parameters **num_rows** : int, optional

Number of rows to print.

num_columns : int, optional

Number of columns to print.

max_column_width : int, optional

Maximum width of a column. Columns use fewer characters if possible.

max_row_width : int, optional

Maximum width of a printed row. Columns beyond this width wrap to a new line. *max_row_width* is automatically reset to be the larger of itself and *max_column_width*.

wrap_text : boolean, optional

Wrap the text within a cell. Defaults to False.

max_wrap_rows : int, optional

When wrapping is in effect, the maximum number of resulting rows for each cell before truncation takes place.

footer : bool, optional

True to pinrt a footer.

See also:

xframes.XFrame.head Returns the first part of a XFrame.

xframes.XFrame.tail Returns the last part of an XFrame.

random_split(*fraction*, *seed=None*)

Randomly split the rows of an XFrame into two XFrames. The first XFrame contains *M* rows, sampled uniformly (without replacement) from the original XFrame. *M* is approximately the fraction times the original number of rows. The second XFrame contains the remaining rows of the original XFrame.

Parameters **fraction** : float

Approximate fraction of the rows to fetch for the first returned XFrame. Must be between 0 and 1.

seed : int, optional

Seed for the random number generator used to split.

Returns tuple [*XFrame*]

Two new XFrame.

Examples

Suppose we have an XFrame with 6,145 rows and we want to randomly split it into training and testing datasets with about a 70%/30% split.

```
>>> xf = xframes.XFrame({'id': range(1024)})
>>> xf_train, xf_test = xf.random_split(.9, seed=5)
>>> print len(xf_test), len(xf_train)
102 922
```

range(*key*)

Extracts and returns rows of the XFrame.

Parameters **key: int or slice**

If *key* is:

- int Returns a single row of the XFrame (the 'key'th one) as a dictionary.
- slice Returns an XFrame including only the sliced rows.

Returns dict or *XFrame*

The specified row of the XFrame or an XFrame containing the specified rows.

classmethod **read_csv**(*url*, *delimiter=','*, *header=True*, *error_bad_lines=False*, *comment_char=''*, *escape_char='\\'*, *double_quote=True*, *quote_char='"'*, *skip_initial_space=True*, *column_type_hints=None*, *na_values=None*, *nrows=None*, *verbose=False*)

Constructs an XFrame from a CSV file or a path to multiple CSVs.

Parameters **url** : string

Location of the CSV file or directory to load. If URL is a directory or a "glob" pattern, all matching files will be loaded.

delimiter : string, optional

This describes the delimiter used for parsing csv files. Must be a single character.

header : bool, optional

If true, uses the first row as the column names. Otherwise use the default column names : 'X1, X2, ...'.

error_bad_lines : bool

If true, will fail upon encountering a bad line. If false, will continue parsing skipping lines which fail to parse correctly. A sample of the first 10 encountered bad lines will be printed.

comment_char : string, optional

The character which denotes that the remainder of the line is a comment.

escape_char : string, optional

Character which begins a C escape sequence

double_quote : bool, optional

If True, two consecutive quotes in a string are parsed to a single quote.

quote_char : string, optional

Character sequence that indicates a quote.

skip_initial_space : bool, optional

Ignore extra spaces at the start of a field

column_type_hints : None, type, list[type], dict[string, type], optional

This provides type hints for each column. By default, this method attempts to detect the type of each column automatically.

Supported types are int, float, str, list, dict, and array.array.

- If a single type is provided, the type will be applied to all columns. For instance, column_type_hints=float will force all columns to be parsed as float.
- If a list of types is provided, the types applies to each column in order, e.g.[int, float, str] will parse the first column as int, second as float and third as string.
- If a dictionary of column name to type is provided, each type value in the dictionary is applied to the key it belongs to. For instance {'user':int} will hint that the column called "user" should be parsed as an integer, and the rest will default to string.

na_values : str | list of str, optional

A string or list of strings to be interpreted as missing values.

nrows : int, optional

If set, only this many rows will be read from the file.

verbose : bool, optional

If True, print the progress while reading files.

Returns *XFrame*

See also:

xframes.XFrame.read_csv_with_errors Allows more control over errors.

xframes.XFrame The constructor can read csv files, but is not configurable.

Examples

Read a regular csv file, with all default options, automatically determine types:

```
>>> url = 'http://s3.amazonaws.com/gl-testdata/rating_data_example.csv'
>>> xf = xframes.XFrame.read_csv(url)
>>> xf
Columns:
  user_id int
  movie_id  int
  rating  int
Rows: 10000
+---------+----------+--------+
| user_id | movie_id | rating |
+---------+----------+--------+
|  25904  |   1663   |   3    |
|  25907  |   1663   |   3    |
|  25923  |   1663   |   3    |
|  25924  |   1663   |   3    |
|  25928  |   1663   |   2    |
```

```
|   ...    |   ...    |  ...   |
+---------+----------+--------+
[10000 rows x 3 columns]
```

Read only the first 100 lines of the csv file:

```
>>> xf = xframes.XFrame.read_csv(url, nrows=100)
>>> xf
Columns:
  user_id int
  movie_id  int
  rating  int
Rows: 100
+---------+----------+--------+
| user_id | movie_id | rating |
+---------+----------+--------+
|  25904  |   1663   |   3    |
|  25907  |   1663   |   3    |
|  25923  |   1663   |   3    |
|  25924  |   1663   |   3    |
|  25928  |   1663   |   2    |
|   ...   |   ...    |  ...   |
+---------+----------+--------+
[100 rows x 3 columns]
```

Read all columns as str type

```
>>> xf = xframes.XFrame.read_csv(url, column_type_hints=str)
>>> xf
Columns:
  user_id  str
  movie_id  str
  rating  str
Rows: 10000
+---------+----------+--------+
| user_id | movie_id | rating |
+---------+----------+--------+
|  25904  |   1663   |   3    |
|  25907  |   1663   |   3    |
|  25923  |   1663   |   3    |
|  25924  |   1663   |   3    |
|  25928  |   1663   |   2    |
|   ...   |   ...    |  ...   |
+---------+----------+--------+
[10000 rows x 3 columns]
```

Specify types for a subset of columns and leave the rest to be str.

```
>>> xf = xframes.XFrame.read_csv(url,
...                              column_type_hints={
...                              'user_id':int, 'rating':float
...                              })
>>> xf
Columns:
  user_id str
  movie_id  str
  rating  float
Rows: 10000
```

```
+---------+----------+--------+
| user_id | movie_id | rating |
+---------+----------+--------+
|  25904  |   1663   |  3.0   |
|  25907  |   1663   |  3.0   |
|  25923  |   1663   |  3.0   |
|  25924  |   1663   |  3.0   |
|  25928  |   1663   |  2.0   |
|   ...   |   ...    |  ...   |
+---------+----------+--------+
[10000 rows x 3 columns]
```

Not treat first line as header:

```
>>> xf = xframes.XFrame.read_csv(url, header=False)
>>> xf
Columns:
  X1  str
  X2  str
  X3  str
Rows: 10001
+---------+----------+--------+
|    X1   |    X2    |   X3   |
+---------+----------+--------+
| user_id | movie_id | rating |
|  25904  |   1663   |   3    |
|  25907  |   1663   |   3    |
|  25923  |   1663   |   3    |
|  25924  |   1663   |   3    |
|  25928  |   1663   |   2    |
|   ...   |   ...    |  ...   |
+---------+----------+--------+
[10001 rows x 3 columns]
```

Treat '3' as missing value:

```
>>> xf = xframes.XFrame.read_csv(url, na_values=['3'], column_type_hints=str)
>>> xf
Columns:
  user_id str
  movie_id  str
  rating  str
Rows: 10000
+---------+----------+--------+
| user_id | movie_id | rating |
+---------+----------+--------+
|  25904  |   1663   |  None  |
|  25907  |   1663   |  None  |
|  25923  |   1663   |  None  |
|  25924  |   1663   |  None  |
|  25928  |   1663   |   2    |
|   ...   |   ...    |  ...   |
+---------+----------+--------+
[10000 rows x 3 columns]
```

Throw error on parse failure:

```
>>> bad_url = 'https://s3.amazonaws.com/gl-testdata/bad_csv_example.csv'
>>> xf = xframes.XFrame.read_csv(bad_url, error_bad_lines=True)
RuntimeError: Runtime Exception. Unable to parse line "x,y,z,a,b,c"
Set error_bad_lines=False to skip bad lines
```

classmethod **read_csv_with_errors**(*url*, *delimiter=','*, *' '*, *header=True*, *comment_char=''*, *escape_char='\\'*, *double_quote=True*, *quote_char='"'*, *skip_initial_space=True*, *column_type_hints=None*, *na_values=None*, *nrows=None*, *verbose=False*)

Constructs an XFrame from a CSV file or a path to multiple CSVs, and returns a pair containing the XFrame and a dict of error type to XArray indicating for each type, what are the incorrectly parsed lines encountered.

The kinds of errors that are detected are:

- width – The row has the wrong number of columns.
- **header – The first row in the file did not parse correctly. This row is used to** determine the table width, so the rest of the file is not processed. The result is an empty XFrame.
- **csv – The csv parser raised a csv.Error or a SystemError exception.** This can be caused by having an unacceptable character, such as a null byte, in the input, or by serious system errors. This presence of this error indicates that processing has been interrupted, so all remaining data in the input file is not processed.

Parameters **url** : string

Location of the CSV file or directory to load. If URL is a directory or a "glob" pattern, all matching files will be loaded.

delimiter : string, optional

This describes the delimiter used for parsing csv files. Must be a single character. Files with double delimiters such as "||" should specify delimiter='|' and should drop columns with empty heading and data.

header : bool, optional

If true, uses the first row as the column names. Otherwise use the default column names: 'X.1, X.2, ...'.

comment_char : string, optional

The character which denotes that the remainder of the line is a comment. The line must contain valid data preceding the commant.

escape_char : string, optional

Character which begins a C escape sequence

double_quote : bool, optional

If True, two consecutive quotes in a string are parsed to a single quote.

quote_char : string, optional

Character sequence that indicates a quote.

skip_initial_space : bool, optional

Ignore extra spaces at the start of a field

column_type_hints : None, type, list[type], dict{string: type}, optional

This provides type hints for each column. By default, this method attempts to detect the type of each column automatically.

Supported types are int, float, str, list, dict, and array.array.

- If a single type is provided, the type will be applied to all columns. For instance, column_type_hints=float will force all columns to be parsed as float.
- If a list of types is provided, the types applies to each column in order, e.g.[int, float, str] will parse the first column as int, second as float and third as string.
- If a dictionary of column name to type is provided, each type value in the dictionary is applied to the key it belongs to. For instance {'user':int} will hint that the column called "user" should be parsed as an integer, and the rest will default to string.

na_values : str | list of str, optional

A string or list of strings to be interpreted as missing values.

nrows : int, optional

If set, only this many rows will be read from the file.

verbose : bool, optional

If True, print the progress while reading files.

Returns tuple

The first element is the XFrame with good data. The second element is a dictionary of filenames to XArrays indicating for each file, what are the incorrectly parsed lines encountered.

See also:

xframes.XFrame.read_csv Reads csv without error controls.

xframes.XFrame The constructor can read csv files, but is not configurable.

Examples

```
>>> bad_url = 'https://s3.amazonaws.com/gl-testdata/bad_csv_example.csv'
>>> (xf, bad_lines) = xframes.XFrame.read_csv_with_errors(bad_url)
>>> xf
+---------+----------+--------+
| user_id | movie_id | rating |
+---------+----------+--------+
|  25904  |   1663   |   3    |
|  25907  |   1663   |   3    |
|  25923  |   1663   |   3    |
|  25924  |   1663   |   3    |
|  25928  |   1663   |   2    |
|   ...   |   ...    |  ...   |
+---------+----------+--------+
[98 rows x 3 columns]
```

```
>>> bad_lines
{'https://s3.amazonaws.com/gl-testdata/bad_csv_example.csv': dtype: str
```

```
Rows: 1
['x,y,z,a,b,c']}
```

classmethod read_parquet (*url*)

Constructs an XFrame from a parquet file.

Parameters **url** : string

Location of the parquet file to load.

Returns *XFrame*

See also:

xframes.XFrame The constructor can read parquet files.

classmethod read_text (*path*, *delimiter=None*, *nrows=None*, *verbose=False*)

Constructs an XFrame from a text file or a path to multiple text files.

Parameters **path** : string

Location of the text file or directory to load. If 'path' is a directory or a "glob" pattern, all matching files will be loaded.

delimiter : string, optional

This describes the delimiter used for separating records. Must be a single character. Defaults to newline.

nrows : int, optional

If set, only this many rows will be read from the file.

verbose : bool, optional

If True, print the progress while reading files.

Returns *XFrame*

Examples

Read a regular text file, with default options.

```
>>> path = 'http://s3.amazonaws.com/gl-testdata/rating_data_example.csv'
>>> xf = xframes.XFrame.read_text(path)
>>> xf
+-------
| text |
+---------+
|  25904  |
|  25907  |
|  25923  |
|  25924  |
|  25928  |
|   ...   |
+---------+
[10000 rows x 1 column]
```

Read only the first 100 lines of the text file:

```
>>> xf = xframes.XFrame.read_text(path, nrows=100)
>>> xf
Rows: 100
+---------+
|  25904  |
|  25907  |
|  25923  |
|  25924  |
|  25928  |
|   ...   |
+---------+
[100 rows x 1 columns]
```

Read using a given delimiter.

```
>>> xf = xframes.XFrame.read_text(path, delimiter='.')
>>> xf
Rows: 250
+---------+
|  25904  |
|  25907  |
|  25923  |
|  25924  |
|  25928  |
|   ...   |
+---------+
[250 rows x 1 columns]
```

remove_column(*name*)

Remove one or more columns from this XFrame. This operation returns a new XFrame with the given column or columns removed.

Parameters **name** : string or list or iterable

The name of the column to remove. If a list or iterable is given, all the named columns are removed.

Returns *XFrame*

A new XFrame with given column or columns removed.

Examples

```
>>> xf = xframes.XFrame({'id': [1, 2, 3], 'val': ['A', 'B', 'C']})
>>> xf2 = xf.remove_column('val')
>>> xf2
+----+
| id |
+----+
| 1  |
| 2  |
| 3  |
+----+
[3 rows x 1 columns]
```

```
>>> xf = xframes.XFrame({'id': [1, 2, 3], 'val1': ['A', 'B', 'C'], 'val2':␣
↪[10, 11, 12]})
>>> xf2 = xf.remove_column(['val1', 'val2'])
>>> xf2
+----+
| id |
+----+
| 1  |
| 2  |
| 3  |
+----+
[3 rows x 1 columns]
```

remove_columns(*column_names*)

Removes one or more columns from this XFrame. This operation returns a new XFrame with the given columns removed.

Parameters **column_names** : list or iterable

A list or iterable of the column names.

Returns *XFrame*

A new XFrame with given columns removed.

Examples

```
>>> xf = xframes.XFrame({'id': [1, 2, 3], 'val1': ['A', 'B', 'C'], 'val2':␣
↪[10, 11, 12]})
>>> xf2 = xf.remove_columns(['val1', 'val2'])
>>> xf2
+----+
| id |
+----+
| 1  |
| 2  |
| 3  |
+----+
[3 rows x 1 columns]
```

rename(*names*)

Rename the given columns. *Names* can be a dict specifying the old and new names. This changes the names of the columns given as the keys and replaces them with the names given as the values. Alternatively, *names* can be a list of the new column names. In this case it must be the same length as the number of columns. This operation returns a new XFrame with the given columns renamed.

Parameters **names** : dict [string, string] | list [string]

Dictionary of [old_name, new_name] or list of new names

Returns *XFrame*

A new XFrame with columns renamed.

See also:

xframes.XFrame.column_names

Examples

```
>>> xf = XFrame({'X.1': ['Alice','Bob'],
...              'X.2': ['123 Fake Street','456 Fake Street']})
>>> xf2 = xf.rename({'X.1': 'name', 'X.2':'address'})
>>> xf2
+-------+-----------------+
|  name |     address     |
+-------+-----------------+
| Alice | 123 Fake Street |
|  Bob  | 456 Fake Street |
+-------+-----------------+
[2 rows x 2 columns]
```

reorder_columns (*column_names*)

Reorder the columns in the table. This operation returns a new XFrame with the given columns reordered.

Parameters **column_names** : list of string

Names of the columns in desired order.

Returns *XFrame*

A new XFrame with reordered columns.

See also:

xframes.XFrame.select_columns Returns a subset of the columns but does not change the column order.

Examples

```
>>> xf = xframes.XFrame({'id': [1, 2, 3], 'val': ['A', 'B', 'C']})
>>> xf2 = xf.reorder_columns(['val', 'id'])
>>> xf2
+-----+-----+
| val | id  |
+-----+-----+
|  A  |  1  |
|  B  |  2  |
|  C  |  3  |
+----+------+
[3 rows x 2 columns]
```

replace_column (*name*, *col*)

Replace a column in this XFrame. The length of the new column must match the length of the existing XFrame. This operation returns a new XFrame with the replacement column.

Parameters **name** : string

The name of the column.

col : *XArray*

The 'column' to add.

Returns *XFrame*

A new XFrame with specified column replaced.

Examples

```
>>> xf = xframes.XFrame({'id': [1, 2, 3], 'val': ['A', 'B', 'C']})
>>> xa = xframes.XArray(['cat', 'dog', 'horse'])
>>> xf2 = xf.replace_column('val', xa)
>>> xf2
+----+---------+
| id | species |
+----+---------+
| 1  |   cat   |
| 2  |   dog   |
| 3  |  horse  |
+----+---------+
[3 rows x 2 columns]
```

sample (*fraction*, *max_partitions=None*, *seed=None*)

Sample the current XFrame's rows.

Parameters **fraction** : float

Approximate fraction of the rows to fetch. Must be between 0 and 1. The number of rows returned is approximately the fraction times the number of rows.

max_partitions : int, optional

After sampling, coalesce to this number of partition. If not given, do not perform this step.

seed : int, optional

Seed for the random number generator used to sample.

Returns *XFrame*

A new XFrame containing sampled rows of the current XFrame.

Examples

Suppose we have an XFrame with 6,145 rows.

```
>>> import random
>>> xf = XFrame({'id': range(0, 6145)})
```

Retrieve about 30% of the XFrame rows with repeatable results by setting the random seed.

```
>>> len(xf.sample(.3, seed=5))
1783
```

save (*filename*, *format=None*)

Save the XFrame to a file system for later use.

Parameters **filename** : string

The location to save the XFrame. Either a local directory or a remote URL. If the format is 'binary', a directory will be created at the location which will contain the XFrame.

format : {'binary', 'csv', 'tsv', 'parquet', json}, optional

Format in which to save the XFrame. Binary saved XFrames can be loaded much faster and without any format conversion losses. If not given, will try to infer the format from filename given. If file name ends with 'csv' or '.csv.gz', then save as 'csv' format. If the file ends with 'json', then save as json file. If the file ends with 'parquet', then save as parquet file. Otherwise save as 'binary' format.

See also:

xframes.XFrame.load, xframes.XFrame.XFrame

Examples

```
>>> # Save the xframe into binary format
>>> xf.save('data/training_data_xframe')
```

```
>>> # Save the xframe into csv format
>>> xf.save('data/training_data.csv', format='csv')
```

select_column(*column_name*)

Return an XArray that corresponds with the given column name. Throws an exception if the column name is something other than a string or if the column name is not found.

Subscripting an XFrame by a column name is equivalent to this function.

Parameters **column_name** : str

The column name.

Returns *XArray*

The XArray that is referred by *column_name*.

See also:

xframes.XFrame.select_columns Returns multiple columns.

Examples

```
>>> xf = xframes.XFrame({'user_id': [1,2,3],
...                      'user_name': ['alice', 'bob', 'charlie']})
>>> # This line is equivalent to `sa = xf['user_name']`
>>> sa = xf.select_column('user_name')
>>> sa
dtype: str
Rows: 3
['alice', 'bob', 'charlie']
```

select_columns(*keylist*)

Get XFrame composed only of the columns referred to in the given list of keys. Throws an exception if ANY of the keys are not in this XFrame or if *keylist* is anything other than a list of strings.

Parameters **keylist** : list[str]

The list of column names.

Returns *XFrame*

A new XFrame that is made up of the columns referred to in *keylist* from the current XFrame. The order of the columns is preserved.

See also:

xframes.XFrame.select_column Returns a single column.

Examples

```
>>> xf = xframes.XFrame({'user_id': [1,2,3],
...                      'user_name': ['alice', 'bob', 'charlie'],
...                      'zipcode': [98101, 98102, 98103]
...                     })
>>> # This line is equivalent to `xf2 = xf[['user_id', 'zipcode']]`
>>> xf2 = xf.select_columns(['user_id', 'zipcode'])
>>> xf2
+---------+---------+
| user_id | zipcode |
+---------+---------+
|    1    |  98101  |
|    2    |  98102  |
|    3    |  98103  |
+---------+---------+
[3 rows x 2 columns]
```

select_rows (*xa*)
Selects rows of the XFrame where the XArray evaluates to True.

Parameters **xa** : *XArray*

Must be the same length as the XFrame. The filter values.

Returns *XFrame*

A new XFrame which contains the rows of the XFrame where the XArray is True. The truth test is the same as in python, so non-zero values are considered true.

classmethod **set_footer_strs** (*footer_strs*)
Set the footer printed beneath tables.

Parameters **footer_strs** : list

A list of strings. Each string is a separate line, printed beneath a table. This footer is used when the length of the table is known. To disable printing the footer, pass an empty list.

classmethod **set_html_max_row_width** (*width*)
Set the maximum display width for displaying in HTML.

Parameters **width** : int

The maximum width of the table when printing in html.

classmethod **set_lazy_footer_strs** (*footer_strs*)
Set the footer printed beneath tables when the length is unknown.

Parameters **footer_strs** : list

A list of strings. Each string is a separate line, printed beneath a table. This footer is used when the length of the table is not known because the XFrame has not been evaluated. To disable printing the footer, pass an empty list.

classmethod **set_max_row_width**(*width*)

Set the maximum display width for printing.

Parameters **width** : int

The maximum width of the table when printing.

shape

The shape of the XFrame, in a tuple. The first entry is the number of rows, the second is the number of columns.

Examples

```
>>> xf = xframes.XFrame({'id':[1,2,3], 'val':['A','B','C']})
>>> xf.shape
(3, 2)
```

sort(*sort_columns*, *ascending=True*)

Sort current XFrame by the given columns, using the given sort order. Only columns that are type of str, int and float can be sorted.

Parameters **sort_columns** : str | list of str | list of (str, bool) pairs

Names of columns to be sorted. The result will be sorted first by first column, followed by second column, and so on. All columns will be sorted in the same order as governed by the *ascending* parameter. To control the sort ordering for each column individually, *sort_columns* must be a list of (str, bool) pairs. Given this case, the first value is the column name and the second value is a boolean indicating whether the sort order is ascending.

ascending : bool, optional

Sort all columns in the given order.

Returns *XFrame*

A new XFrame that is sorted according to given sort criteria

See also:

xframes.XFrame.topk

Examples

Suppose 'xf' is an xframe that has three columns 'a', 'b', 'c'. To sort by column 'a', ascending:

```
>>> xf = xframes.XFrame({'a':[1,3,2,1],
...                      'b':['a','c','b','b'],
...                      'c':['x','y','z','y']})
>>> xf
+---+---+---+
| a | b | c |
+---+---+---+
| 1 | a | x |
| 3 | c | y |
| 2 | b | z |
| 1 | b | y |
+---+---+---+
[4 rows x 3 columns]
```

```
>>> xf.sort('a')
+---+---+---+
| a | b | c |
+---+---+---+
| 1 | a | x |
| 1 | b | y |
| 2 | b | z |
| 3 | c | y |
+---+---+---+
[4 rows x 3 columns]
```

To sort by column 'a', descending:

```
>>> xf.sort('a', ascending = False)
+---+---+---+
| a | b | c |
+---+---+---+
| 3 | c | y |
| 2 | b | z |
| 1 | a | x |
| 1 | b | y |
+---+---+---+
[4 rows x 3 columns]
```

To sort by column 'a' and 'b', all ascending:

```
>>> xf.sort(['a', 'b'])
+---+---+---+
| a | b | c |
+---+---+---+
| 1 | a | x |
| 1 | b | y |
| 2 | b | z |
| 3 | c | y |
+---+---+---+
[4 rows x 3 columns]
```

To sort by column 'a' ascending, and then by column 'c' descending:

```
>>> xf.sort([('a', True), ('c', False)])
+---+---+---+
| a | b | c |
+---+---+---+
| 1 | b | y |
| 1 | a | x |
| 2 | b | z |
| 3 | c | y |
+---+---+---+
[4 rows x 3 columns]
```

split_datetime (*expand_column*, *column_name_prefix=None*, *limit=None*)

Splits a datetime column of XFrame to multiple columns, with each value in a separate column. Returns a new XFrame with the expanded column replaced with a list of new columns. The expanded column must be of datetime.datetime type.

For more details regarding name generation and other, refer to `xframes.XArray.expand()`

Parameters **expand_column** : str

Name of the unpacked column.

column_name_prefix : str, optional

If provided, expanded column names would start with the given prefix. If not provided, the default value is the name of the expanded column.

limit : list[str], optional

Limits the set of datetime elements to expand. Elements are 'year','month','day','hour','minute', and 'second'.

Returns *XFrame*

A new XFrame that contains rest of columns from original XFrame with the given column replaced with a collection of expanded columns.

Examples

```
>>> xf
Columns:
    id   int
    submission  datetime.datetime
Rows: 2
Data:
    +----+-----------------------------------------------------------+
    | id |                 submission                                |
    +----+-----------------------------------------------------------+
    | 1  | datetime.datetime(2011, 1, 21, 7, 17, 21)                 |
    | 2  | datetime.datetime(2011, 1, 21, 5, 43, 21)                 |
    +----+-----------------------------------------------------------+
```

```
>>> xf.split_datetime('submission',limit=['hour','minute'])
Columns:
    id  int
    submission.hour int
    submission.minute int
Rows: 2
Data:
+----+-----------------+-------------------+
| id | submission.hour | submission.minute |
+----+-----------------+-------------------+
| 1  |        7        |        17         |
| 2  |        5        |        43         |
+----+-----------------+-------------------+
```

sql (*sql_statement*, *table_name='xframe'*)

Executes the given sql statement over the data in the table. Returns a new XFrame with the results.

Parameters **sql_statement** : str

The statement to execute.

The statement is executed by the Spark Sql query processor. See the SparkSql documentation for details. XFrame column names and types are translated to Spark for query processing.

table_name : str, optional

The table name to create, referred to in the sql statement. Defaulst to 'xframe'.

Returns XFrame

The new XFrame with the results.

Examples

```
>>> xf = xframes.XFrame({'id': [1, 2, 3], 'val': ['a', 'b', 'c']})
>>> xf.sql("SELECT * FROM xframe WHERE id > 1"
+----+--------+
| id |  val   |
+----+--------+
| 2  |   'b'  |
| 3  |   'c'  |
+----+-----  -+
[3 rows x 2 columns]
```

stack(*column_name*, *new_column_name=None*, *drop_na=False*)

Convert a "wide" column of an XFrame to one or two "tall" columns by stacking all values.

The stack works only for columns of dict, list, or array type. If the column is dict type, two new columns are created as a result of stacking: one column holds the key and another column holds the value. The rest of the columns are repeated for each key/value pair.

If the column is array or list type, one new column is created as a result of stacking. With each row holds one element of the array or list value, and the rest columns from the same original row repeated.

The new XFrame includes the newly created column and all columns other than the one that is stacked.

Parameters **column_name** : str

The column to stack. This column must be of dict/list/array type

new_column_name : str | list of str, optional

The new column name(s). If original column is list/array type, new_column_name must a string. If original column is dict type, new_column_name must be a list of two strings. If not given, column names are generated automatically.

drop_na : boolean, optional

If True, missing values and empty list/array/dict are all dropped from the resulting column(s). If False, missing values are maintained in stacked column(s).

Returns *XFrame*

A new XFrame that contains newly stacked column(s) plus columns in original XFrame other than the stacked column.

See also:

xframes.XFrame.unstack Undo the effect of stack.

Examples

Suppose 'xf' is an XFrame that contains a column of dict type:

```
>>> xf = xframes.XFrame({'topic':[1,2,3,4],
...                      'words': [{'a':3, 'cat':2},
...                                {'a':1, 'the':2},
...                                {'the':1, 'dog':3},
...                                {}]
...                     })
+-------+----------------------+
| topic |        words         |
+-------+----------------------+
|   1   |  {'a': 3, 'cat': 2}  |
|   2   |  {'a': 1, 'the': 2}  |
|   3   | {'the': 1, 'dog': 3} |
|   4   |          {}          |
+-------+----------------------+
[4 rows x 2 columns]
```

Stack would stack all keys in one column and all values in another column:

```
>>> xf.stack('words', new_column_name=['word', 'count'])
+-------+------+-------+
| topic | word | count |
+-------+------+-------+
|   1   |  a   |   3   |
|   1   | cat  |   2   |
|   2   |  a   |   1   |
|   2   | the  |   2   |
|   3   | the  |   1   |
|   3   | dog  |   3   |
|   4   | None |  None |
+-------+------+-------+
[7 rows x 3 columns]
```

Observe that since topic 4 had no words, an empty row is inserted. To drop that row, set `dropna=True` in the parameters to stack.

Suppose 'xf' is an XFrame that contains a user and his/her friends, where 'friends' columns is an array type. Stack on 'friends' column would create a user/friend list for each user/friend pair:

```
>>> xf = xframes.XFrame({'topic':[1,2,3],
...                      'friends':[[2,3,4], [5,6],
...                                 [4,5,10,None]]
...                     })
>>> xf
+------+------------------+
| user |     friends      |
+------+------------------+
|  1   |     [2, 3, 4]    |
|  2   |      [5, 6]      |
|  3   | [4, 5, 10, None] |
+------+------------------+
[3 rows x 2 columns]
```

```
>>> xf.stack('friends', new_column_name='friend')
+------+--------+
| user | friend |
+------+--------+
|  1   |  2     |
|  1   |  3     |
```

```
|  1   |  4     |
|  2   |  5     |
|  2   |  6     |
|  3   |  4     |
|  3   |  5     |
|  3   |  10    |
|  3   |  None  |
+------+--------+
[9 rows x 2 columns]
```

swap_columns(*column_1*, *column_2*)

Swap the columns with the given names. This operation returns a new XFrame with the given columns swapped.

Parameters **column_1** : string

Name of column to swap

column_2 : string

Name of other column to swap

Returns *XFrame*

A new XFrame with specified columns swapped.

Examples

```
>>> xf = xframes.XFrame({'id': [1, 2, 3], 'val': ['A', 'B', 'C']})
>>> xf2 = xf.swap_columns('id', 'val')
>>> xf2
+-----+-----+
| val | id  |
+-----+-----+
|  A  |  1  |
|  B  |  2  |
|  C  |  3  |
+----+-----+
[3 rows x 2 columns]
```

tail(*n=10*)

The last n rows of the XFrame.

Parameters **n** : int, optional

The number of rows to fetch.

Returns *XFrame*

A new XFrame which contains the last n rows of the current XFrame.

See also:

xframes.XFrame.head Returns the first part of the XFrame.

xframes.XFrame.print_rows Prints the XFrame.

to_pandas_dataframe()

Convert this XFrame to pandas.DataFrame.

This operation will construct a pandas.DataFrame in memory. Care must be taken when size of the returned object is big.

Returns pandas.DataFrame

The dataframe which contains all rows of XFrame.

to_rdd()

Convert the current XFrame to a Spark RDD. The RDD consists of tuples containing the column data. No conversion is necessary: the internal RDD is returned.

Returns spark.RDD

The spark RDD that is used to represent the XFrame.

See also:

from_rdd Converts from a Spark RDD.

to_spark_dataframe(*table_name=None*, *column_names=None*, *column_type_hints=None*, *number_of_partitions=None*)

Convert the current XFrame to a Spark DataFrame.

Parameters **table_name** : str, optional

If given, give this name to the temporary table.

column_names : list, optional

A list of the column names to assign. Defaults to the names in the table, edited to fit the Dataframe restrictions.

column_type_hints : dict, optional

Column types must be supplied when creating a DataFrame. These hints specify these types, If hints are not given, the column types are derived from the XFrame column types. The column types in DataFrames are more restricted in XFrames.

XFrames attempts to supply the correct column types, but cannot always determine the correct settings. The caller can supply hints to ensure the desired settings, but the caller is still responsible for making sure the values in the XFrame are consistent with these settings. * Integers: In DataFrames integers must fit in 64 bits. In python, large integers can be larger. If an XFrame contains such integers, it will fail to store as a DataFrame. The column can be converted to strings in this case.

- Lists must be of a uniform type in a DataFrame. The caller must convert

lists to meet this requirement, and must provide a hint specifying the element type.

- Dictionaries must have a uniform key and value type.

The caller must convert dictionaries to meet this requirement and must provide a hint specifying the key and value types.

Hints are given as a dictionary of column_name: column_hint. Any column without a hint is handled using the XFrame column type. For simple types, hints are just type names (as strings): int, long float, bool, datetime, or str. For lists, hints are "list[<type>]" where <type> is one of the simple types. For dictionaries, hints are "dict{<key_type>:<value_type>}" where key_type and value_type is one of the simple types.

number_of_partitions : int, optional

The number of partitions to create.

Returns spark.DataFrame

The converted spark dataframe.

topk(*column_name*, *k=10*, *reverse=False*)

Get k rows according to the largest values in the given column. Result is sorted by *column_name* in the given order (default is descending). When *k* is small, *topk* is more efficient than *sort*.

Parameters **column_name** : string

The column to sort on

k : int, optional

The number of rows to return

reverse : bool, optional

If True, return the top k rows in ascending order, otherwise, in descending order.

Returns *XFrame*

An XFrame containing the top k rows sorted by column_name.

See also:

xframes.XFrame.sort

Examples

```
>>> xf = xframes.XFrame({'id': range(1000)})
>>> xf['value'] = -xf['id']
>>> xf.topk('id', k=3)
+--------+--------+
|   id   |  value |
+--------+--------+
|   999  |  -999  |
|   998  |  -998  |
|   997  |  -997  |
+--------+--------+
[3 rows x 2 columns]
```

```
>>> xf.topk('value', k=3)
+--------+--------+
|   id   |  value |
+--------+--------+
|   1    |  -1    |
|   2    |  -2    |
|   3    |  -3    |
+--------+--------+
[3 rows x 2 columns]
```

transform_col(*col*, *fn=None*, *dtype=None*, *use_columns=None*, *seed=None*)

Transform a single column according to a specified function. The remaining columns are not modified. The type of the transformed column types becomes *dtype*, with the new value being the result of *fn(x)*, where *x* is a single row in the XFrame represented as a dictionary. The *fn* should return exactly one value which can be cast into type *dtype*. If *dtype* is not specified, the first 100 rows of the XFrame are used to make a guess of the target data type.

Parameters **col** : string

The name of the column to transform.

fn : function, optional

The function to transform each row of the XFrame. The return type should be convertible to *dtype* if *dtype* is not None. If the function is not given, an identity function is used.

dtype : dtype, optional

The column data type of the new XArray. If None, the first 100 elements of the array are used to guess the target data type.

use_columns : str | list[str], optional

The column or list of columns to be supplied in the row passed to the function. If not given, all columns wll be used to build the row.

seed : int, optional

Used as the seed if a random number generator is included in *fn*.

Returns *XFrame*

An XFrame with the given column transformed by the function and cast to the given type.

Examples

Translate values in a column:

```
>>> xf = xframes.XFrame({'user_id': [1, 2, 3], 'movie_id': [3, 3, 6],
                         'rating': [4, 5, 1]})
>>> xf.transform_col('rating', lambda row: row['rating'] * 2)
```

Cast values in a column to a different type

```
>>> xf = xframes.XFrame({'user_id': [1, 2, 3], 'movie_id': [3, 3, 6],
                         'rating': [4, 5, 1]})
>>> xf.transform_col('user_id', dtype=str)
```

transform_cols(*cols*, *fn=None*, *dtypes=None*, *use_columns=None*, *seed=None*)

Transform multiple columns according to a specified function. The remaining columns are not modified. The type of the transformed column types are given by *dtypes*, with the new values being the result of *fn(x)* where *x* is a single row in the XFrame represented as a dictionary. The *fn* should return a value for each element of cols, which can be cast into the corresponding *dtype*. If *dtypes* is not specified, the first 100 rows of the XFrame are used to make a guess of the target data types.

Parameters **cols** : list [str]

The names of the column to transform.

fn : function, optional

The function to transform each row of the XFrame. The return value should be a list of values, one for each column of cols. each type should be convertible to the corresponding *dtype* if *dtype* is not None. If the function is not given, an identity function is used.

dtypes : list[type], optional

The data types of the new columns. There must be one data type for each column in cols. If not supplied, the first 100 elements of the array are used to guess the target data types.

use_columns : str | list[str], optional

The column or list of columns to be supplied in the row passed to the function. If not given, all columns wll be used to build the row.

seed : int, optional

Used as the seed if a random number generator is included in *fn*.

Returns *XFrame*

An XFrame with the given columns transformed by the function and cast to the given types.

Examples

Translate values in a column:

```
>>> xf = xframes.XFrame({'user_id': [1, 2, 3], 'movie_id': [3, 3, 6],
                         'rating': [4, 5, 1]})
>>> xf.transform_col(['movie_id', 'rating'], lambda row: [row['movie_id'] +␣
↪1, row['rating'] * 2])
```

Cast types in several columns:

```
>>> xf = xframes.XFrame({'user_id': [1, 2, 3], 'movie_id': [3, 3, 6],
                         'rating': [4, 5, 1]})
>>> xf.transform_col(['movie_id', 'rating'], dtype=[str, str])
```

unique()

Remove duplicate rows of the XFrame. Will not necessarily preserve the order of the given XFrame in the new XFrame.

Returns *XFrame*

A new XFrame that contains the unique rows of the current XFrame.

Raises **TypeError**

If any column in the XFrame is a dictionary type.

See also:

xframes.XFrame.unique

Examples

```
>>> xf = xframes.XFrame({'id':[1,2,3,3,4], 'value':[1,2,3,3,4]})
>>> xf
+----+-------+
| id | value |
+----+-------+
| 1  |   1   |
| 2  |   2   |
| 3  |   3   |        | 3  |   3   |
```

```
| 4  |   4   |
+----+-------+
[5 rows x 2 columns]
```

```
>>> xf.unique()
+----+-------+
| id | value |
+----+-------+
| 2  |   2   |
| 4  |   4   |
| 3  |   3   |
| 1  |   1   |
+----+-------+
[4 rows x 2 columns]
```

unpack(*unpack_column*, *column_name_prefix=None*, *column_types=None*, *na_value=None*, *limit=None*)

Expand one column of this XFrame to multiple columns with each value in a separate column. Returns a new XFrame with the unpacked column replaced with a list of new columns. The column must be of list, tuple, array, or dict type.

For more details regarding name generation, missing value handling and other, refer to the XArray version of *unpack()*.

Parameters **unpack_column** : str

Name of the unpacked column

column_name_prefix : str, optional

If provided, unpacked column names would start with the given prefix. If not provided, default value is the name of the unpacked column.

column_types : [type], optional

Column types for the unpacked columns. If not provided, column types are automatically inferred from first 100 rows. For array type, default column types are float. If provided, column_types also restricts how many columns to unpack.

na_value : flexible_type, optional

If provided, convert all values that are equal to "na_value" to missing value (None).

limit : list[str] | list[int], optional

Control unpacking only a subset of list/array/dict value. For dictionary XArray, *limit* is a list of dictionary keys to restrict. For list/array XArray, *limit* is a list of integers that are indexes into the list/array value.

Returns *XFrame*

A new XFrame that contains rest of columns from original XFrame with the given column replaced with a collection of unpacked columns.

See also:

xframes.XFrame.pack_columns The opposite of unpack.

Examples

```
>>> xf = xframes.XFrame({'id': [1,2,3],
...                      'wc': [{'a': 1}, {'b': 2}, {'a': 1, 'b': 2}]})
+----+------------------+
| id |        wc        |
+----+------------------+
| 1  |     {'a': 1}     |
| 2  |     {'b': 2}     |
| 3  | {'a': 1, 'b': 2} |
+----+------------------+
[3 rows x 2 columns]
```

```
>>> xf.unpack('wc')
+----+------+------+
| id | wc.a | wc.b |
+----+------+------+
| 1  |  1   | None |
| 2  | None |  2   |
| 3  |  1   |  2   |
+----+------+------+
[3 rows x 3 columns]
```

To not have prefix in the generated column name:

```
>>> xf.unpack('wc', column_name_prefix="")
+----+------+------+
| id |  a   |  b   |
+----+------+------+
| 1  |  1   | None |
| 2  | None |  2   |
| 3  |  1   |  2   |
+----+------+------+
[3 rows x 3 columns]
```

To limit subset of keys to unpack:

```
>>> xf.unpack('wc', limit=['b'])
+----+------+
| id | wc.b |
+----+------+
| 1  | None |
| 2  |  2   |
| 3  |  2   |
+----+------+
[3 rows x 3 columns]
```

To unpack an array column:

```
>>> xf = xframes.XFrame({'id': [1,2,3],
...                      'friends': [array.array('d', [1.0, 2.0, 3.0]),
...                                  array.array('d', [2.0, 3.0, 4.0]),
...                                  array.array('d', [3.0, 4.0, 5.0])]})
>>> xf
+----+-----------------------------+
| id |            friends          |
+----+-----------------------------+
```

```
| 1  | array('d', [1.0, 2.0, 3.0]) |
| 2  | array('d', [2.0, 3.0, 4.0]) |
| 3  | array('d', [3.0, 4.0, 5.0]) |
+----+-----------------------------+
[3 rows x 2 columns]
```

```
>>> xf.unpack('friends')
+----+-----------+-----------+-----------+
| id | friends.0 | friends.1 | friends.2 |
+----+-----------+-----------+-----------+
| 1  |    1.0    |    2.0    |    3.0    |
| 2  |    2.0    |    3.0    |    4.0    |
| 3  |    3.0    |    4.0    |    5.0    |
+----+-----------+-----------+-----------+
[3 rows x 4 columns]
```

unstack(*column*, *new_column_name=None*)

Concatenate values from one or two columns into one column, grouping by all other columns. The resulting column could be of type list, array or dictionary. If *column* is a numeric column, the result will be of array.array type. If *column* is a non-numeric column, the new column will be of list type. If *column* is a list of two columns, the new column will be of dict type where the keys are taken from the first column in the list.

Parameters **column** : str | [str, str]

The column(s) that is(are) to be concatenated. If str, then collapsed column type is either array or list. If [str, str], then collapsed column type is dict

new_column_name : str, optional

New column name. If not given, a name is generated automatically.

Returns *XFrame*

A new XFrame containing the grouped columns as well as the new column.

See also:

xframes.XFrame.stack The inverse of unstack.

xframes.XFrame.groupby `Unstack` is a special version of `groupby` that uses the *CONCAT* aggregator

Notes

- There is no guarantee the resulting XFrame maintains the same order as the original XFrame.
- Missing values are maintained during unstack.
- When unstacking into a dictionary, if there is more than one instance of a given key for a particular group, an arbitrary value is selected.

Examples

```
>>> xf = xframes.XFrame({'count':[4, 2, 1, 1, 2, None],
...                      'topic':['cat', 'cat', 'dog', 'elephant', 'elephant
↪', 'fish'],
```

```
...                       'word':['a', 'c', 'c', 'a', 'b', None]})
>>> xf.unstack(column=['word', 'count'], new_column_name='words')
+---------+------------------+
|  topic  |      words       |
+---------+------------------+
| elephant | {'a': 1, 'b': 2} |
|   dog   |     {'c': 1}     |
|   cat   | {'a': 4, 'c': 2} |
|   fish  |       None       |
+---------+------------------+
[4 rows x 2 columns]
```

```
>>> xf = xframes.XFrame({'friend': [2, 3, 4, 5, 6, 4, 5, 2, 3],
...                      'user': [1, 1, 1, 2, 2, 2, 3, 4, 4]})
>>> xf.unstack('friend', new_column_name='friends')
+------+-----------------------------+
| user |           friends           |
+------+-----------------------------+
|  3   |      array('d', [5.0])      |
|  1   | array('d', [2.0, 4.0, 3.0]) |
|  2   | array('d', [5.0, 6.0, 4.0]) |
|  4   |    array('d', [2.0, 3.0])   |
+------+-----------------------------+
[4 rows x 2 columns]
```

width()

Diagnostic: the number of elements in each tuple of the RDD.

CHAPTER 2

XArray

class xframes.**XArray**(*data=None*, *dtype=None*, *ignore_cast_failure=False*, *impl=None*)

An immutable, homogeneously typed array object backed by Spark RDD.

XArray is able to hold data that are much larger than the machine's main memory. It fully supports missing values and random access (although random access is inefficient). The data backing an XArray is located on the cluster hosting Spark.

__init__(*data=None*, *dtype=None*, *ignore_cast_failure=False*, *impl=None*)

Construct a new XArray. The source of data includes: list, numpy.ndarray, pandas.Series, and urls.

Parameters **data** : list | numpy.ndarray | pandas.Series | string

The input data. If this is a list, numpy.ndarray, or pandas.Series, the data in the list is converted and stored in an XArray. Alternatively if this is a string, it is interpreted as a path (or url) to a text file. Each line of the text file is loaded as a separate row. If *data* is a directory where an XArray was previously saved, this is loaded as an XArray read directly out of that directory.

dtype : {int, float, str, list, array.array, dict, datetime.datetime}, optional

The data type of the XArray. If not specified, we attempt to infer it from the input. If it is a numpy array or a Pandas series, the data type of the array or series is used. If it is a list, the data type is inferred from the inner list. If it is a URL or path to a text file, we default the data type to str.

ignore_cast_failure : bool, optional

If True, ignores casting failures but warns when elements cannot be cast into the specified data type.

See also:

xframes.XArray.from_const Constructs an XArray of a given size with a const value.

xframes.XArray.from_sequence Constructs an XArray by generating a sequence of consecutive numbers.

xframes.XArray.from_rdd Create a new XArray from a Spark RDD or Spark DataFrame.

xframes.XArray.set_trace Controls entry and exit tracing.

xframes.XArray.spark_context Returns the spark context.

xframes.XArray.spark_sql_context Returns the spark sql context.

xframes.XArray.hive_context Returns the spark hive context.

Notes

•If *data* is pandas.Series, the index will be ignored.

The following functionality is currently not implemented:

- numpy.ndarray as row data
- pandas.Series data
- count_words, count_ngrams
- sketch sub_sketch_keys

Examples

```
>>> xa = XArray(data=[1,2,3,4,5], dtype=int)
>>> xa = XArray('s3://testdatasets/a_to_z.txt.gz')
>>> xa = XArray([[1,2,3], [3,4,5]])
>>> xa = XArray(data=[{'a':1, 'b': 2}, {'b':2, 'c': 1}])
>>> xa = XArray(data=[datetime.datetime(2011, 10, 20, 9, 30, 10)])
```

all()

Return True if every element of the XArray evaluates to True.

For numeric XArrays zeros and missing values (None) evaluate to False, while all non-zero, non-missing values evaluate to True. For string, list, and dictionary XArrays, empty values (zero length strings, lists or dictionaries) or missing values (None) evaluate to False. All other values evaluate to True.

Returns True on an empty XArray.

Returns bool

See also:

xframes.XArray.any

Examples

```
>>> xframes.XArray([1, None]).all()
False
>>> xframes.XArray([1, 0]).all()
False
>>> xframes.XArray([1, 2]).all()
True
>>> xframes.XArray(["hello", "world"]).all()
True
>>> xframes.XArray(["hello", ""]).all()
False
```

```
>>> xframes.XArray([]).all()
True
```

any()

Return True if any element of the XArray evaluates to True.

For numeric XArrays any non-zero value evaluates to True. For string, list, and dictionary XArrays, any element of non-zero length evaluates to True.

Returns False on an empty XArray.

Returns bool

See also:

xframes.XArray.all

Examples

```
>>> xframes.XArray([1, None]).any()
True
>>> xframes.XArray([1, 0]).any()
True
>>> xframes.XArray([0, 0]).any()
False
>>> xframes.XArray(["hello", "world"]).any()
True
>>> xframes.XArray(["hello", ""]).any()
True
>>> xframes.XArray(["", ""]).any()
False
>>> xframes.XArray([]).any()
False
```

append(*other*)

Append an XArray to the current XArray. Creates a new XArray with the rows from both XArrays. Both XArrays must be of the same data type.

Parameters **other** : *XArray*

Another XArray whose rows are appended to current XArray.

Returns *XArray*

A new XArray that contains rows from both XArrays, with rows from the other XArray coming after all rows from the current XArray.

See also:

xframes.XFrame.append Appends XFrames

Examples

```
>>> xa = xframes.XArray([1, 2, 3])
>>> xa2 = xframes.XArray([4, 5, 6])
>>> xa.append(xa2)
dtype: int
```

```
Rows: 6
[1, 2, 3, 4, 5, 6]
```

apply (*fn*, *dtype=None*, *skip_undefined=True*, *seed=None*)

Transform each element of the XArray by a given function.

The result XArray is of type *dtype*. *fn* should be a function that returns exactly one value which can be cast into the type specified by *dtype*. If *dtype* is not specified, the first 100 elements of the XArray are used to make a guess about the data type.

Parameters **fn** : function

The function to transform each element. Must return exactly one value which can be cast into the type specified by *dtype*.

dtype : {int, float, str, list, array.array, dict}, optional

The data type of the new XArray. If not supplied, the first 100 elements of the array are used to guess the target data type.

skip_undefined : bool, optional

If True, will not apply *fn* to any missing values.

seed : int, optional

Used as the seed if a random number generator is included in *fn*.

Returns *XArray*

The XArray transformed by *fn*. Each element of the XArray is of type *dtype*.

See also:

xframes.XFrame.apply Applies a function to a column of an XFrame. Note that the functions differ in these two cases: on an XArray the function receives one value, on an XFrame it receives a dict of the column name/value pairs.

Examples

```
>>> xa = xframes.XArray([1,2,3])
>>> xa.apply(lambda x: x*2)
dtype: int
Rows: 3
[2, 4, 6]
```

astype (*dtype*, *undefined_on_failure=False*)

Create a new XArray with all values cast to the given type. Throws an exception if the types are not castable to the given type.

Parameters **dtype** : {int, float, str, list, array.array, dict, datetime.datetime}

The type to cast the elements to in XArray

undefined_on_failure: bool, optional

If set to True, runtime cast failures will be emitted as missing values rather than failing.

Returns *XArray* of dtype

The XArray converted to the type *dtype*.

Notes

- The string parsing techniques used to handle conversion to dictionary and list types are quite generic and permit a variety of interesting formats to be interpreted. For instance, a JSON string can usually be interpreted as a list or a dictionary type. See the examples below.
- For datetime-to-string and string-to-datetime conversions, use xa.datetime_to_str() and xa.str_to_datetime() functions.

Examples

```
>>> xa = xframes.XArray(['1','2','3','4'])
>>> xa.astype(int)
dtype: int
Rows: 4
[1, 2, 3, 4]
```

Given an XArray of strings that look like dicts, convert to a dictionary type:

```
>>> xa = xframes.XArray(['{1:2 3:4}', '{a:b c:d}'])
>>> xa.astype(dict)
dtype: dict
Rows: 2
[{1: 2, 3: 4}, {'a': 'b', 'c': 'd'}]
```

clip (*lower=None*, *upper=None*)

Create a new XArray with each value clipped to be within the given bounds.

In this case, "clipped" means that values below the lower bound will be set to the lower bound value. Values above the upper bound will be set to the upper bound value. This function can operate on XArrays of numeric type as well as array type, in which case each individual element in each array is clipped. By default *lower* and *upper* are set to `None` which indicates the respective bound should be ignored. The method fails if invoked on an XArray of non-numeric type.

Parameters **lower** : int, optional

The lower bound used to clip. Ignored if equal to `None` (the default).

upper : int, optional

The upper bound used to clip. Ignored if equal to `None` (the default).

Returns *XArray*

See also:

xframes.XArray.clip_lower, *xframes.XArray.clip_upper*

Examples

```
>>> xa = xframes.XArray([1,2,3])
>>> xa.clip(2,2)
dtype: int
Rows: 3
[2, 2, 2]
```

clip_lower (*threshold*)

Create new XArray with all values clipped to the given lower bound. This function can operate on numeric arrays, as well as vector arrays, in which case each individual element in each vector is clipped. Throws an exception if the XArray is empty or the types are non-numeric.

Parameters **threshold** : float

The lower bound used to clip values.

Returns *XArray*

See also:

xframes.XArray.clip, *xframes.XArray.clip_upper*

Examples

```
>>> xa = xframes.XArray([1,2,3])
>>> xa.clip_lower(2)
dtype: int
Rows: 3
[2, 2, 3]
```

clip_upper (*threshold*)

Create new XArray with all values clipped to the given upper bound. This function can operate on numeric arrays, as well as vector arrays, in which case each individual element in each vector is clipped.

Parameters **threshold** : float

The upper bound used to clip values.

Returns *XArray*

See also:

xframes.XArray.clip, *xframes.XArray.clip_lower*

Examples

```
>>> xa = xframes.XArray([1,2,3])
>>> xa.clip_upper(2)
dtype: int
Rows: 3
[1, 2, 2]
```

countna ()

Count the number of missing values in the XArray.

A missing value is represented in a float XArray as 'NaN' or None. A missing value in other types of XArrays is None.

Returns int

The count of missing values.

datetime_to_str (*str_format='%Y-%m-%dT%H:%M:%S%ZP'*)

Create a new XArray with all the values cast to str. The string format is specified by the 'str_format' parameter.

Parameters **str_format** : str

The format to output the string. Default format is "%Y-%m-%dT%H:%M:%S%ZP".

Returns *XArray* of str

The XArray converted to the type 'str'.

See also:

xframes.XArray.str_to_datetime

Examples

```
>>> dt = datetime.datetime(2011, 10, 20, 9, 30, 10, tzinfo=GMT(-5))
>>> xa = xframes.XArray([dt])
>>> xa.datetime_to_str('%e %b %Y %T %ZP')
dtype: str
Rows: 1
[20 Oct 2011 09:30:10 GMT-05:00]
```

dict_has_all_keys(*keys*)

Create a boolean XArray by checking the keys of an XArray of dictionaries.

An element of the output XArray is True if the corresponding input element's dictionary has all of the given keys. Fails on XArrays whose data type is not `dict`.

Parameters **keys** : list

A list of key values to check each dictionary against.

Returns *XArray*

An XArray of int type, where each element indicates whether the input XArray element contains all keys in the input list.

See also:

xframes.XArray.dict_has_any_keys

Examples

```
>>> xa = xframes.XArray([{"this":1, "is":5, "dog":7},
                         {"this": 2, "are": 1, "cat": 5}])
>>> xa.dict_has_all_keys(["is", "this"])
dtype: int
Rows: 2
[1, 0]
```

dict_has_any_keys(*keys*)

Create a boolean XArray by checking the keys of an XArray of dictionaries. An element of the output XArray is True if the corresponding input element's dictionary has any of the given keys. Fails on XArrays whose data type is not `dict`.

Parameters **keys** : list

A list of key values to check each dictionary against.

Returns *XArray*

A XArray of int type, where each element indicates whether the input XArray element contains any key in the input list.

See also:

xframes.XArray.dict_has_all_keys

Examples

```
>>> xa = xframes.XArray([{"this":1, "is":5, "dog":7}, {"animal":1},
                         {"this": 2, "are": 1, "cat": 5}])
>>> xa.dict_has_any_keys(["is", "this", "are"])
dtype: int
Rows: 3
[1, 1, 0]
```

dict_keys()

Create an XArray that contains all the keys from each dictionary element as a list. Fails on XArrays whose data type is not `dict`.

Returns *XArray*

A XArray of list type, where each element is a list of keys from the input XArray element.

See also:

xframes.XArray.dict_values

Examples

```
>>> xa = xframes.XArray([{"this":1, "is":5, "dog":7},
                         {"this": 2, "are": 1, "cat": 5}])
>>> xa.dict_keys()
dtype: list
Rows: 2
[['this', 'is', 'dog'], ['this', 'are', 'cat']]
```

dict_trim_by_keys(*keys*, *exclude=True*)

Filter an XArray of dictionary type by the given keys. By default, all keys that are in the provided list in *keys* are *excluded* from the returned XArray.

Parameters **keys** : list

A collection of keys to trim down the elements in the XArray.

exclude : bool, optional

If True, all keys that are in the input key list are removed. If False, only keys that are in the input key list are retained.

Returns *XArray*

A XArray of dictionary type, with each dictionary element trimmed according to the input criteria.

See also:

xframes.XArray.dict_trim_by_values

Examples

```
>>> xa = xframes.XArray([{"this":1, "is":1, "dog":2},
                         {"this": 2, "are": 2, "cat": 1}])
>>> xa.dict_trim_by_keys(["this", "is", "and", "are"], exclude=True)
dtype: dict
Rows: 2
[{'dog': 2}, {'cat': 1}]
```

dict_trim_by_values(*lower=None*, *upper=None*)

Filter dictionary values to a given range (inclusive). Trimming is only performed on values which can be compared to the bound values. Fails on XArrays whose data type is not `dict`.

Parameters **lower** : int or long or float, optional

The lowest dictionary value that would be retained in the result. If not given, lower bound is not applied.

upper : int or long or float, optional

The highest dictionary value that would be retained in the result. If not given, upper bound is not applied.

Returns *XArray*

An XArray of dictionary type, with each dict element trimmed according to the input criteria.

See also:

xframes.XArray.dict_trim_by_keys

Examples

```
>>> xa = xframes.XArray([{"this":1, "is":5, "dog":7},
                         {"this": 2, "are": 1, "cat": 5}])
>>> xa.dict_trim_by_values(2,5)
dtype: dict
Rows: 2
[{'is': 5}, {'this': 2, 'cat': 5}]
```

```
>>> xa.dict_trim_by_values(upper=5)
dtype: dict
Rows: 2
[{'this': 1, 'is': 5}, {'this': 2, 'are': 1, 'cat': 5}]
```

dict_values()

Create an XArray that contains all the values from each dictionary element as a list. Fails on XArrays whose data type is not `dict`.

Returns *XArray*

A XArray of list type, where each element is a list of values from the input XArray element.

See also:

xframes.XArray.dict_keys

Examples

```
>>> xa = xframes.XArray([{"this":1, "is":5, "dog":7},
                         {"this": 2, "are": 1, "cat": 5}])
>>> xa.dict_values()
dtype: list
Rows: 2
[[1, 5, 7], [2, 1, 5]]
```

dropna()
Create new XArray containing only the non-missing values of the XArray.

A missing value is represented in a float XArray as 'NaN' on None. A missing value in other types of XArrays is None.

Returns *XArray*

The new XArray with missing values removed.

dtype()
The data type of the XArray.

Returns type

The type of the XArray.

Examples

```
>>> xa = XArray(['The quick brown fox jumps over the lazy dog.'])
>>> xa.dtype()
str
>>> xa = XArray(range(10))
>>> xa.dtype()
int
```

dump_debug_info()
Print information about the Spark RDD associated with this XArray.

fillna(*value*)
Create new XArray with all missing values (None or NaN) filled in with the given value.

The size of the new XArray will be the same as the original XArray. If the given value is not the same type as the values in the XArray, *fillna* will attempt to convert the value to the original XArray's type. If this fails, an error will be raised.

Parameters **value** : type convertible to XArray's type

The value used to replace all missing values.

Returns *XArray*

A new XArray with all missing values filled.

filter(*fn*, *skip_undefined=True*, *seed=None*)
Filter this XArray by a function.

Returns a new XArray filtered by a function. If *fn* evaluates an element to true, this element is copied to the new XArray. If not, it isn't. Throws an exception if the return type of *fn* is not castable to a boolean value.

Parameters **fn** : function

Function that filters the XArray. Must evaluate to bool or int.

skip_undefined : bool, optional

If True, will not apply fn to any undefined values.

seed : int, optional

Used as the seed if a random number generator is included in fn.

Returns *XArray*

The XArray filtered by fn. Each element of the XArray is of type int.

Examples

```
>>> xa = xframes.XArray([1,2,3])
>>> xa.filter(lambda x: x < 3)
dtype: int
Rows: 2
[1, 2]
```

flat_map (*fn=None*, *dtype=None*, *skip_undefined=True*, *seed=None*)

Transform each element of the XArray by a given function, which must return a list.

Each item in the result XArray is made up of a list element. The result XArray is of type *dtype*. *fn* should be a function that returns a list of values which can be cast into the type specified by *dtype*. If *dtype* is not specified, the first 100 elements of the XArray are used to make a guess about the data type.

Parameters **fn** : function

The function to transform each element. Must return a list of values which can be cast into the type specified by *dtype*.

dtype : {None, int, float, str, list, array.array, dict}, optional

The data type of the new XArray. If None, the first 100 elements of the array are used to guess the target data type.

skip_undefined : bool, optional

If True, will not apply *fn* to any undefined values.

seed : int, optional

Used as the seed if a random number generator is included in *fn*.

Returns *XArray*

The XArray transformed by *fn* and flattened. Each element of the XArray is of type *dtype*.

See also:

xframes.XFrame.flat_map

Examples

```
>>> xa = xframes.XArray([[1], [1, 2], [1, 2, 3]])
>>> xa.apply(lambda x: x*2)
dtype: int
```

```
Rows: 3
[2, 2, 4, 2, 4, 6]
```

classmethod **from_const**(*value*, *size*)

Constructs an XArray of size with a const value.

Parameters **value** : [int | float | str | array.array | datetime.datetime | list | dict]

The value to fill the XArray.

size : int

The size of the XArray. Must be positive.

Examples

Construct an XArray consisting of 10 zeroes:

```
>>> xframes.XArray.from_const(0, 10)
```

classmethod **from_rdd**(*rdd*, *dtype*, *lineage=None*)

Convert a Spark RDD into an XArray

Parameters **rdd** : pyspark.rdd.RDD

The Spark RDD containing the XArray values.

dtype : type

The values in *rdd* should have the data type *dtype*.

lineage: dict, optional

The lineage to apply to the rdd.

Returns class:.*XArray*

This incorporates the given RDD.

classmethod **from_sequence**(*start*, *stop=None*)

Constructs an XArray by generating a sequence of consecutive numbers.

Parameters **start** : int

If *stop* is not given, the sequence consists of numbers 0 .. *start*-1. Otherwise, the sequence starts with *start*.

stop : int, optional

If given, the sequence consists of the numbers *start*, *start'+1* ... *'end*-1. The sequence will not contain this value.

Examples

```
>>> from_sequence(1000)
Construct an XArray of integer values from 0 to 999
```

This is equivalent, but more efficient than: >>> XArray(range(1000))

```
>>> from_sequence(10, 1000)
Construct an XArray of integer values from 10 to 999
```

This is equivalent, but more efficient than: >>> XArray(range(10, 1000))

head(*n=10*)

Returns an XArray which contains the first n rows of this XArray.

Parameters **n** : int

The number of rows to fetch.

Returns *XArray*

A new XArray which contains the first n rows of the current XArray.

Examples

```
>>> XArray(range(10)).head(5)
dtype: int
Rows: 5
[0, 1, 2, 3, 4]
```

impl()

Get the impl. For internal use.

item_length()

Length of each element in the current XArray.

Only works on XArrays of string, dict, array, or list type. If a given element is a missing value, then the output elements is also a missing value. This function is equivalent to the following but more performant:

xa_item_len = xa.apply(lambda x: len(x) if x is not None else None)

Returns *XArray*

A new XArray, each element in the XArray is the len of the corresponding items in original XArray.

Examples

```
>>> xa = XArray([
...  {"is_restaurant": 1, "is_electronics": 0},
...  {"is_restaurant": 1, "is_retail": 1, "is_electronics": 0},
...  {"is_restaurant": 0, "is_retail": 1, "is_electronics": 0},
...  {"is_restaurant": 0},
...  {"is_restaurant": 1, "is_electronics": 1},
...  None])
>>> xa.item_length()
dtype: int
Rows: 6
[2, 3, 3, 1, 2, None]
```

lineage()

The lineage: the files that went into building this array.

Returns dict

- **key 'table': set[filename]** The files that were used to build the XArray
- **key 'column': dict{column_name: set[filename]}** The set of files that were used to build each column

max()
Get maximum numeric value in XArray.

Returns None on an empty XArray. Raises an exception if called on an XArray with non-numeric type.

Returns type of XArray

Maximum value of XArray

See also:

xframes.XArray.min

Examples

```
>>> xframes.XArray([14, 62, 83, 72, 77, 96, 5, 25, 69, 66]).max()
96
```

mean()
Mean of all the values in the XArray.

Returns None on an empty XArray. Raises an exception if called on an XArray with non-numeric type.

Returns float

Mean of all values in XArray.

min()
Get minimum numeric value in XArray.

Returns None on an empty XArray. Raises an exception if called on an XArray with non-numeric type.

Returns type of XArray

Minimum value of XArray

See also:

xframes.XArray.max

Examples

```
>>> xframes.XArray([14, 62, 83, 72, 77, 96, 5, 25, 69, 66]).min()
```

nnz()
Number of non-zero elements in the XArray.

Returns int

Number of non-zero elements.

num_missing()
Number of missing elements in the XArray.

Returns int

Number of missing values.

classmethod **read_text** (*path*, *delimiter=None*, *nrows=None*, *verbose=False*)
Constructs an XArray from a text file or a path to multiple text files.

Parameters **path** : string

Location of the text file or directory to load. If 'path' is a directory or a "glob" pattern, all matching files will be loaded.

delimiter : string, optional

This describes the delimiter used for separating records. Must be a single character. Defaults to newline.

nrows : int, optional

If set, only this many rows will be read from the file.

verbose : bool, optional

If True, print the progress while reading files.

Returns *XArray*

Examples

Read a regular text file, with default options.

```
>>> path = 'http://s3.amazonaws.com/gl-testdata/rating_data_example.csv'
>>> xa = xframes.XArray.read_text(path)
>>> xa
[25904, 25907, 25923, 25924, 25928,  ... ]
```

Read only the first 100 lines of the text file:

```
>>> xa = xframes.XArray.read_text(path, nrows=100)
>>> xa
[25904, 25907, 25923, 25924, 25928,  ... ]
```

sample (*fraction*, *max_partitions=None*, *seed=None*)
Create an XArray which contains a subsample of the current XArray.

Parameters **fraction** : float

The fraction of the rows to fetch. Must be between 0 and 1.

max_partitions : int, optional

After sampling, coalesce to this number of partition. If not given, do not perform this step.

seed : int

The random seed for the random number generator.

Returns *XArray*

The new XArray which contains the subsampled rows.

Examples

```
>>> xa = xframes.XArray(range(10))
>>> xa.sample(.3)
dtype: int
Rows: 3
[2, 6, 9]
```

save(*filename*, *format=None*)
Saves the XArray to file.

The saved XArray will be in a directory named with the *filename* parameter.

Parameters **filename** : string

A local path or a remote URL. If format is 'text', it will be saved as a text file. If format is 'binary', a directory will be created at the location which will contain the XArray.

format : {'binary', 'text', 'csv'}, optional

Format in which to save the XFrame. Binary saved XArrays can be loaded much faster and without any format conversion losses. The values 'text' and 'csv' are synonymous: Each XArray row will be written as a single line in an output text file. If not given, will try to infer the format from filename given. If file name ends with 'csv', or 'txt', then save as 'csv' format, otherwise save as 'binary' format.

size()
The size of the XArray.

sketch_summary(*sub_sketch_keys=None*)
Summary statistics that can be calculated with one pass over the XArray.

Returns a *Sketch* object which can be further queried for many descriptive statistics over this XArray. Many of the statistics are approximate. See the *Sketch* documentation for more detail.

Parameters **sub_sketch_keys: int | str | list of int | list of str, optional**

For XArray of dict type, also constructs sketches for a given set of keys, For XArray of array type, also constructs sketches for the given indexes. The sub sketches may be queried using: *element_sub_sketch()* Defaults to None in which case no subsketches will be constructed.

Returns *Sketch*

Sketch object that contains descriptive statistics for this XArray. Many of the statistics are approximate.

sort(*ascending=True*)
Sort all values in this XArray.

Sort only works for xarray of type str, int and float, otherwise TypeError will be raised. Creates a new, sorted XArray.

Parameters **ascending: boolean, optional**

If True, the xarray values are sorted in ascending order, otherwise, descending order.

Returns *XArray*

The sorted XArray.

Examples

```
>>> xa = XArray([3,2,1])
>>> xa.sort()
dtype: int
Rows: 3
[1, 2, 3]
```

split_datetime(*column_name_prefix='X'*, *limit=None*)

Splits an XArray of datetime type to multiple columns, return a new XFrame that contains expanded columns. A XArray of datetime will be split by default into an XFrame of 6 columns, one for each year/month/day/hour/minute/second element.

column naming: When splitting a XArray of datetime type, new columns are named: prefix.year, prefix.month, etc. The prefix is set by the parameter "column_name_prefix" and defaults to 'X'. If column_name_prefix is None or empty, then no prefix is used.

Parameters **column_name_prefix: str, optional**

If provided, expanded column names would start with the given prefix. Defaults to "X".

limit: str, list[str], optional

Limits the set of datetime elements to expand. Elements may be 'year','month','day','hour','minute', and 'second'.

Returns *XFrame*

A new XFrame that contains all expanded columns

Examples

To expand only day and year elements of a datetime XArray

```
>>> xa = XArray(
   [datetime.datetime(2011, 1, 21, 7, 7, 21),
    datetime.datetime(2010, 2, 5, 7, 8, 21)])
```

```
>>> xa.split_datetime(column_name_prefix=None,limit=['day','year'])
   Columns:
       day   int
       year  int
   Rows: 2
   Data:
   +-------+--------+
   |  day  |  year  |
   +-------+--------+
   |   21  |  2011  |
   |   5   |  2010  |
   +-------+--------+
   [2 rows x 2 columns]
```

std(*ddof=0*)

Standard deviation of all the values in the XArray.

Returns None on an empty XArray. Raises an exception if called on an XArray with non-numeric type or if *ddof* >= length of XArray.

Parameters **ddof** : int, optional

"delta degrees of freedom" in the variance calculation.

Returns float

The standard deviation of all the values.

str_to_datetime(*str_format=None*)

Create a new XArray whose column type is datetime. The string format is specified by the 'str_format' parameter.

Parameters **str_format** : str, optional

The string format of the input XArray. If not given, dateutil parser is used.

Returns *XArray* of datetime.datetime

The XArray converted to the type 'datetime'.

See also:

xframes.XArray.datetime_to_str

Examples

```
>>> xa = xframes.XArray(['20-Oct-2011 09:30:10 GMT-05:30'])
>>> xa.str_to_datetime('%d-%b-%Y %H:%M:%S %ZP')
dtype: datetime.datetime
Rows: 1
datetime.datetime(2011, 10, 20, 9, 30, 10)
```

```
>>> xa = xframes.XArray(['Aug 23, 2015'])
>>> xa.str_to_datetime()
dtype: datetime.datetime
Rows: 1
datetime.datetime(2015, 8, 23, 0, 0, 0)
```

sum()

Sum of all values in this XArray.

Raises an exception if called on an XArray of strings. If the XArray contains numeric arrays (list or array.array) and all the lists or arrays are the same length, the sum over all the arrays will be returned. If the XArray contains dictionaries whose values are numeric, then the sum of values whose keys appear in every row. Returns None on an empty XArray. For large values, this may overflow without warning.

Returns type of XArray

Sum of all values in XArray

tail(*n=10*)

Creates an XArray that contains the last n elements in the given XArray.

Parameters **n** : int

The number of elements.

Returns *XArray*

A new XArray which contains the last n rows of the current XArray.

to_rdd(*number_of_partitions=4*)

Convert the current XArray to the Spark RDD.

Parameters **number_of_partitions: int, optional**

The number of partitions to create in the rdd. Defaults to 4.

Returns out: RDD

The internal RDD used to stores XArray instances.

topk_index(*topk=10*, *reverse=False*)

Create an XArray indicating which elements are in the top k.

Entries are '1' if the corresponding element in the current XArray is a part of the top k elements, and '0' if that corresponding element is not. Order is descending by default.

Parameters **topk** : int

The number of elements to determine if 'top'

reverse: bool

If True, return the topk elements in ascending order

Returns *XArray* of int

Notes

This is used internally by XFrame's topk function.

unique()

Get all unique values in the current XArray.

Will not necessarily preserve the order of the given XArray in the new XArray. Raises a TypeError if the XArray is of dictionary type.

Returns *XArray*

A new XArray that contains the unique values of the current XArray.

See also:

xframes.XFrame.unique Unique rows in XFrames.

unpack(*column_name_prefix='X'*, *column_types=None*, *na_value=None*, *limit=None*)

Convert an XFrame of list, array, or dict type to an XFrame with multiple columns.

unpack expands an XArray using the values of each list/array/dict as elements in a new XFrame of multiple columns. For example, an XArray of lists each of length 4 will be expanded into an XFrame of 4 columns, one for each list element. An XArray of lists/tuples/arrays of varying size will be expand to a number of columns equal to the longest list/array. An XArray of dictionaries will be expanded into as many columns as there are keys.

When unpacking an XArray of list or array type, new columns are named: *column_name_prefix*.0, *column_name_prefix*.1, etc. If unpacking a column of dict type, unpacked columns are named *column_name_prefix*.key1, *column_name_prefix*.key2, etc.

When unpacking an XArray of list or dictionary types, missing values in the original element remain as missing values in the resultant columns. If the *na_value* parameter is specified, all values equal to this given value are also replaced with missing values. In an XArray of array.array type, NaN is interpreted as a missing value.

xframes.XFrame.pack_columns() is the reverse effect of unpack

Parameters **column_name_prefix: str, optional**

If provided, unpacked column names would start with the given prefix.

column_types: list[type], optional

Column types for the unpacked columns. If not provided, column types are automatically inferred from first 100 rows. Defaults to None.

na_value: optional

Convert all values that are equal to *na_value* to missing value if specified.

limit: list, optional

Limits the set of list/array/dict keys to unpack. For list/array XArrays, 'limit' must contain integer indices. For dict XArray, 'limit' must contain dictionary keys.

Returns *XFrame*

A new XFrame that contains all unpacked columns

Examples

To unpack a dict XArray

```
>>> xa = XArray([{ 'word': 'a',      'count': 1},
...              { 'word': 'cat',    'count': 2},
...              { 'word': 'is',     'count': 3},
...              { 'word': 'coming','count': 4}])
```

Normal case of unpacking XArray of type dict:

```
>>> xa.unpack(column_name_prefix=None)
Columns:
    count   int
    word    str

Rows: 4

Data:
+-------+--------+
| count |  word  |
+-------+--------+
|   1   |   a    |
|   2   |  cat   |
|   3   |   is   |
|   4   | coming |
+-------+--------+
[4 rows x 2 columns]
```

Unpack only keys with 'word':

```
>>> xa.unpack(limit=['word'])
Columns:
    X.word  str

Rows: 4

Data:
+--------+
```

```
| X.word |
+--------+
|   a    |
|  cat   |
|   is   |
| coming |
+--------+
[4 rows x 1 columns]
```

```
>>> xa2 = XArray([
...                [1, 0, 1],
...                [1, 1, 1],
...                [0, 1]])
```

Convert all zeros to missing values:

```
>>> xa2.unpack(column_types=[int, int, int], na_value=0)
Columns:
    X.0     int
    X.1     int
    X.2     int

Rows: 3

Data:
+------+------+------+
| X.0  | X.1  | X.2  |
+------+------+------+
|  1   | None |  1   |
|  1   |  1   |  1   |
| None |  1   | None |
+------+------+------+
[3 rows x 3 columns]
```

var (*ddof=0*)

Variance of all the values in the XArray.

Returns None on an empty XArray. Raises an exception if called on an XArray with non-numeric type or if *ddof* >= length of XArray.

Parameters **ddof** : int, optional

"delta degrees of freedom" in the variance calculation.

Returns float

Variance of all values in XArray.

vector_slice (*start*, *end=None*)

If this XArray contains vectors or recursive types, this returns a new XArray containing each individual vector sliced, between start and end, exclusive.

Parameters **start** : int

The start position of the slice.

end : int, optional.

The end position of the slice. Note that the end position is NOT included in the slice. Thus a g.vector_slice(1,3) will extract entries in position 1 and 2.

Returns *XArray*

Each individual vector sliced according to the arguments.

Examples

If g is a vector of floats:

```
>>> g = XArray([[1,2,3],[2,3,4]])
>>> g
dtype: array
Rows: 2
[array('d', [1.0, 2.0, 3.0]), array('d', [2.0, 3.0, 4.0])]
```

```
>>> g.vector_slice(0) # extracts the first element of each vector
dtype: float
Rows: 2
[1.0, 2.0]
```

```
>>> g.vector_slice(0, 2) # extracts the first two elements of each vector
dtype: array.array
Rows: 2
[array('d', [1.0, 2.0]), array('d', [2.0, 3.0])]
```

If a vector cannot be sliced, the result will be None:

```
>>> g = XArray([[1],[1,2],[1,2,3]])
>>> g
dtype: array.array
Rows: 3
[array('d', [1.0]), array('d', [1.0, 2.0]), array('d', [1.0, 2.0, 3.0])]
```

```
>>> g.vector_slice(2)
dtype: float
Rows: 3
[None, None, 3.0]
```

```
>>> g.vector_slice(0,2)
dtype: list
Rows: 3
[None, array('d', [1.0, 2.0]), array('d', [1.0, 2.0])]
```

If g is a vector of mixed types (float, int, str, array, list, etc.):

```
>>> g = XArray([['a',1,1.0],['b',2,2.0]])
>>> g
dtype: list
Rows: 2
[['a', 1, 1.0], ['b', 2, 2.0]]
```

```
>>> g.vector_slice(0) # extracts the first element of each vector
dtype: list
Rows: 2
[['a'], ['b']]
```

CHAPTER 3

XStream

class xframes.**XStream**(*impl=None*, *verbose=False*)

Provides for streams of XFrames.

An XStream represents a time sequence of XFrames. These are usually read from a live sources, and are processed in batches at a selectable interval.

XStream objects encapsulate the logic associated with the stream. The interface includes a number of class methods that act as factory methods, connecting up to external systems are returning an XStream.

XStream also includes a number of transformers, taking one or two XStreams and transforming them into another XStream.

Finally, XStream includes a number of sinks, that print, save, or expose the stream to external systems.

XFrame instances are created immediately (and can be used in Jupyter notebooks without restrictions). But data does not flow through the streams until the application calls "start". This data flow happens in another thread, so your program gets control back immediately after calling "start".

Methods that print data (such as print_frames) do not produce output until data starts flowing. Their output goes to stdout, along with anything that you main thread is doing, which works well in a notebook environment.

As with the other parts of XFrames (and Spark) many of the operators take functional arguments, containing the actions to be applied to the data structures. These functions run in a worker environment, not on the main thread (they run in another process, generally on another machine). Thus you will not see anythin that you write to stdout or stderr from these functions. If you know where to look, you can find this output in the Spark worker log files.

__init__(*impl=None*, *verbose=False*)

Construct an XStream. You rarely construct an XStream directly, but through the factory methods.

Parameters **verbose** : bool, optional

If True, print progress messages.

See also:

xframes.XStream.create_from_text_files Create an XStream from text files.

xframes.XStream.create_from_socket_stream Create an XStream from data received over a socket.

xframes.XStream.create_from_kafka_topic Create from a kafka topic.

add_column(*col*, *name=''*)

Add a column to every XFrame in this XStream. The length of the new column must match the length of the existing XFrame. This operation returns new XFrames with the additional columns. If no *name* is given, a default name is chosen.

Parameters **col** : XArray

The 'column' of data to add.

name : string, optional

The name of the column. If no name is given, a default name is chosen.

Returns *XStream* of *XFrame*

A new XStream of XFrame with the new column.

See also:

xframes.XFrame.add_column Corresponding function on individual frame.

add_columns(*cols*, *namelist=None*)

Adds multiple columns to this XFrame. The length of the new columns must match the length of the existing XFrame. This operation returns a new XFrame with the additional columns.

Parameters **cols** : list of XArray or XFrame

The columns to add. If *cols* is an XFrame, all columns in it are added.

namelist : list of string, optional

A list of column names. All names must be specified. *Namelist* is ignored if *cols* is an XFrame. If there are columns with duplicate names, they will be made unambiguous by adding .1 to the second copy.

Returns *XStream* of *XFrame*

The XStream with additional columns.

See also:

xframes.XFrame.add_columns Corresponding function on individual frame.

apply(*fn*, *dtype*)

Transform each XFrame in an XStream to an XArray according to a specified function. Returns a XStream of XArray of *dtype* where each element in this XArray is transformed by *fn(x)* where *x* is a single row in the xframe represented as a dictionary. The *fn* should return exactly one value which can be cast into type *dtype*.

Parameters **fn** : function

The function to transform each row of the XFrame. The return type should be convertible to *dtype* if *dtype* is not None.

dtype : data type

The *dtype* of the new XArray. If None, the first 100 elements of the array are used to guess the target data type.

Returns *XStream* of *XFrame*

The stream of XFrame transformed by fn. Each element of the XArray is of type *dtype*

static await_termination(*timeout=None*)
Wait for streaming execution to stop.

Parameters **timeout** : int, optional

The maximum time to wait, in seconds. If not given, wait indefinitely.

Returns **status** : boolean

True if the stream has stopped. False if the given timeout has expired and the timeout expired.

column_names()
The name of each column in the XStream.

Returns list[string]

Column names of the XStream.

See also:

xframes.XFrame.column_names Corresponding function on individual frame.

column_types()
The type of each column in the XFrame.

Returns list[type]

Column types of the XFrame.

See also:

xframes.XFrame.column_types Corresponding function on individual frame.

count_distinct(*col*)
Counts the number of different values in a column of each XFrame in the stream.

Returns *XStream* of *XFrame*

Returns a new XStream consisting of one-row XFrames. Each XFrame has one column, "count" containing the number of rows in each consittuent XFrame.

static create_from_kafka_topic(*topics*, *kafka_servers=None*, *kafka_params=None*)
Create XStream (stream of XFrames) from one or more kafka topics.

Records will be read from a kafka topic or topics. Each read delivers a group of messages, as controlled by the consumer params. These records are converted into an XFrame using the ingest function, and are processed sequentially.

Parameters **topics** : string | list

A single topic name, or a list of topic names. These are kafka topics that are used to get data from kafka.

kafka_servers : string | list, optional

A single kafka server or a list of kafka servers. Each server is of the form server-name:port. If no server is given, the server "localhost:9002" is used.

kafka_params : dict, optional

A dictionary of param name - value pairs. These are passed to kafka as consumer configuration parameters.. See kafka documentation http://kafka.apache.org/documentation.html#newconsumerconfigs for more details on kafka consumer configuration params. If no kafka params are supplied, the list of kafka servers specified in this function is passed as the "bootstrap.servers" param.

Returns *XStream*

An XStream (of XFrames) made up or rows read from the socket.

static create_from_socket_stream(*hostname*, *port*)

Create XStream (stream of XFrames) from text gathered from a socket.

Parameters **hostname** : str

The data hostname.

port : str

The port to connect to.

Returns *XStream*

An XStream (of XFrames) made up or rows read from the socket.

static create_from_text_files(*directory_path*)

Create XStream (stream of XFrames) from text gathered files in a directory.

Monitors the directory. As new files are added, they are read into XFrames and introduced to the stream.

Parameters **directory_path** : str

The directory where files are stored.

Returns *XStream*

An XStream (of XFrames) made up or rows read from files in the directory.

dtype()

The type of each column in the XFrame.

Returns list[type]

Column types of the XFrame.

See also:

xframes.XFrame.dtype Corresponding function on individual frame.

dump_debug_info()

Print information about the Spark RDD associated with this XFrame.

See also:

xframes.XFrame.dump_debug_info Corresponding function on individual frame.

filterby(*values*, *col_name*, *exclude=False*)

Filter an XStream by values inside an iterable object. Result is an XStream that only includes (or excludes) the rows that have a column with the given *column_name* which holds one of the values in the given *values* XArray. If *values* is not an XArray, we attempt to convert it to one before filtering.

Parameters **values** : XArray | list |tuple | set | iterable | numpy.ndarray | pandas.Series | str | function

The values to use to filter the XFrame. The resulting XFrame will only include rows that have one of these values in the given column. If this is f function, it is called on each row and is passed the value in the column given by 'column_name'. The result includes rows where the function returns True.

col_name : str | None

The column of the XFrame to match with the given *values*. This can only be None if the values argument is a function. In this case, the function is passed the whole row.

exclude : bool

If True, the result XFrame will contain all rows EXCEPT those that have one of *values* in *column_name*.

Returns *XStream* of *XFrame*

The filtered XStream.

See also:

xframes.XFrame.filterby Corresponding function on individual frame.

flat_map(*column_names*, *fn*, *column_types='auto'*)

Map each row of each XFrame to multiple rows in a new XFrame via a function.

The output of *fn* must have type `list[list[...]]`. Each inner list will be a single row in the new output, and the collection of these rows within the outer list make up the data for the output XFrame. All rows must have the same length and the same order of types to make sure the result columns are homogeneously typed. For example, if the first element emitted into the outer list by *fn* is `[43, 2.3, 'string']`, then all other elements emitted into the outer list must be a list with three elements, where the first is an *int*, second is a *float*, and third is a *string*. If *column_types* is not specified, the first 10 rows of the XFrame are used to determine the column types of the returned XFrame.

Parameters **column_names** : list[str]

The column names for the returned XFrame.

fn : function

The function that maps each of the xframe rows into multiple rows, returning `list[list[...]]`. All output rows must have the same length and order of types. The function is passed a dictionary of column name: value for each row.

column_types : list[type]

The column types of the output XFrame.

Returns *XStream*

A new XStream containing the results of the `flat_map` of the XFrames in the XStream.

See also:

xframes.XFrame.flat_map Corresponding function on individual frame.

classmethod from_dstream(*dstream*, *col_names*, *column_types*)

Create a XStream from a spark DStream. The data should be:

Parameters **dstream** : spark.DStream

Data used to populate the XStream

col_names : list of string

The column names to use.

column_types : list of type

The column types to use.

Returns *XStream*

See also:

from_rdd Converts from a Spark RDD. Corresponding function on individual frame.

groupby(*key_columns*, *operations=None*, **args*)

Perform a group on the *key_columns* followed by aggregations on the columns listed in *operations*.

The *operations* parameter is a dictionary that indicates which aggregation operators to use and which columns to use them on. The available operators are SUM, MAX, MIN, COUNT, MEAN, VARIANCE, STD, CONCAT, SELECT_ONE, ARGMIN, ARGMAX, and QUANTILE. See *aggregate* for more detail on the aggregators.

Parameters **key_columns** : string | list[string]

Column(s) to group by. Key columns can be of any type other than dictionary.

operations : dict, list, optional

Dictionary of columns and aggregation operations. Each key is a output column name and each value is an aggregator. This can also be a list of aggregators, in which case column names will be automatically assigned.

***args**

All other remaining arguments will be interpreted in the same way as the operations argument.

Returns *XStream*

An XStream (of XFrames) made up or rows read from the socket.

A new XFrame, with a column for each groupby column and each aggregation operation.

See also:

xframes.XFrame.groupby Corresponding function on individual frame.

lineage()

The table lineage: the files that went into building this table.

Returns dict

- **key 'table': set[filename]** The files that were used to build the XArray
- **key 'column': dict{col_name: set[filename]}** The set of files that were used to build each column

See also:

xframes.XFrame.lineage Corresponding function on individual frame.

num_columns()
The number of columns in this XFrame.

Returns int

Number of columns in the XFrame.

See also:

xframes.XFrame.num_rows Returns the number of rows.

num_rows()
Counts the rows in each XFrame in the stream.

Returns stream of XFrames

Returns a new XStream consisting of one-row XFrames. Each XFrame has one column, "count" containing the number of rows in each consittuent XFrame.

See also:

xframes.XFrame.num_rows Corresponding function on individual frame.

print_frames(*label=None*, *num_rows=10*, *num_columns=40*, *max_column_width=30*, *max_row_width=None*, *wrap_text=False*, *max_wrap_rows=2*, *footer=False*)
Print the first rows and columns of each XFrame in the XStream in human readable format.

Parameters **num_rows** : int, optional

Number of rows to print.

num_columns : int, optional

Number of columns to print.

max_column_width : int, optional

Maximum width of a column. Columns use fewer characters if possible.

max_row_width : int, optional

Maximum width of a printed row. Columns beyond this width wrap to a new line. *max_row_width* is automatically reset to be the larger of itself and *max_column_width*.

wrap_text : boolean, optional

Wrap the text within a cell. Defaults to False.

max_wrap_rows : int, optional

When wrapping is in effect, the maximum number of resulting rows for each cell before truncation takes place.

footer : bool, optional

True to pinrt a footer.

See also:

xframes.XFrame.print_rows Corresponding function on individual frame.

process_frames(*frame_fn*, *init_fn=None*, *final_fn=None*)

Process the XFrames in an XStream using a given frame processing function.

This is an output operation, and forces the XFrames to be evaluated, for their side effects.

Parameters **frame_fn** : function

This function is called on each XFrame in the XStream. This function receives two parameters: a frame and an initiali value. The initial value is the return value resulting from calling the init_fn. The frame_fn need not return a value: the function is called for its side effects only.

init_fn : function, optional

The init_fn is a parameterless function, used to set up the environment for the frame function. Its value is passed to each invocation of the frame function. If no init_fn is passed, then each frame function will receive None as its second argument.

The rows are processed in parallel in groups on one or more worker machines. For each group, init_fn is called once, and its return value is passed to each row_fn. It could be used, for instance, to open a file or socket that is used by each of the row functions.

final_fn : function, optional

The final_fn is called after each group is processed. It is a function of one parameter, the return value of the initial function.

See also:

xframes.XStream.process_rows Processes individual rows and return a result.

process_rows(*row_fn*, *init_fn=None*, *final_fn=None*)

Process the rows in an XStream of XFrames using a given row processing function.

This is an output operation, and forces the XFrames to be evaluated.

Parameters **row_fn** : function

This function is called on each row of each XFrame. This function receives two parameters: a row and an initiali value. The row is in the form of a dictionary of column-name: column_value pairs. The initial value is the return value resulting from calling the init_fn. The row_fn need not return a value: the function is called for its side effects only.

init_fn : function, optional

The init_fn is a parameterless function, used to set up the environment for the row function. Its value is passed to each invocation of the row function. If no init_fn is passed, then each row function will receive None as its second argument.

The rows are processed in parallel in groups on one or more worker machines. For each group, init_fn is called once, and its return value is passed to each row_fn. It could be used, for instance, to open a file or socket that is used by each of the row functions.

final_fn : function, optional

The final_fn is called after each group is processed. It is a function of one parameter, the return value of the initial function.

Returns *XStream* of *XFrame*

XStream of XFrames that have been processed by the row function.

See also:

xframes.XStream.process_frames Processes whole frames for their side effects only.

remove_column(*name*)

Remove a column or columns from this XFrame. This operation returns a new XFrame with the given column removed.

Parameters **name** : string or list or iterable

The name of the column to remove. If a list or iterable is given, all the named columns are removed.

Returns *XStream* of *XFrame*

XStream of XFrames with given column removed.

See also:

xframes.XFrame.remove_column Corresponding function on individual frame.

remove_columns(*column_names*)

Removes one or more columns from this XFrame. This operation returns a new XFrame with the given columns removed.

Parameters **column_names** : list or iterable

A list or iterable of the column names.

Returns *XStream* of *XFrame*

XStream of XFrames with given columns removed.

See also:

xframes.XFrame.remove_columns Corresponding function on individual frame.

rename(*names*)

Rename the given columns. *Names* can be a dict specifying the old and new names. This changes the names of the columns given as the keys and replaces them with the names given as the values. Alternatively, *names* can be a list of the new column names. In this case it must be the same length as the number of columns. This operation returns a new XFrame with the given columns renamed.

Parameters **names** : dict [string, string] | list [string]

Dictionary of [old_name, new_name] or list of new names

Returns *XStream* of *XFrame*

XStream of XFrames with columns renamed.

See also:

xframes.XFrame.rename Corresponding function on individual frame.

reorder_columns(*column_names*)

Reorder the columns in the table. This operation returns a new XFrame with the given columns reordered.

Parameters **column_names** : list of string

Names of the columns in desired order.

Returns *XStream* of *XFrame*

XStream of XFrames with reordered columns.

See also:

xframes.XFrame.reorder_columns Corresponding function on individual frame.

replace_column(*name*, *col*)

Replace a column in this XFrame. The length of the new column must match the length of the existing XFrame. This operation returns a new XFrame with the replacement column.

Parameters **name** : string

The name of the column.

col : XArray

The 'column' to add.

Returns *XStream* of *XFrame*

A new XStream of XFrames with specified column replaced.

See also:

xframes.XFrame.replace_column Corresponding function on individual frame.

save(*prefix*, *suffix=None*)

Save the XStream to a set of files in the file system.

This is an output operation, and forces the XFrames to be evaluated.

Parameters **prefix** : string

The base location to save each XFrame in the XStream. The filename of each files will be made as follows: prefix-TIME-IN-MS.suffix. The prefix should be either a local directory or a remote URL.

suffix : string, optional

The filename suffix. Defaults to no suffix.

See also:

xframes.XFrame.save Corresponding function on individual frame.

select_column(*column_name*)

Return an XStream of XArray that corresponds with the given column name. Throws an exception if the column name is something other than a string or if the column name is not found.

Subscripting an XStream by a column name is equivalent to this function.

Parameters **column_name** : str

The column name.

Returns *XStream* of *XFrame*

The XStream of XArray that is referred by *column_name*.

See also:

xframes.XFrame.select_column Corresponding function on individual frame.

select_columns(*keylist*)
Get XFrame composed only of the columns referred to in the given list of keys. Throws an exception if ANY of the keys are not in this XFrame or if *keylist* is anything other than a list of strings.

Parameters **keylist** : list[str]

The list of column names.

Returns *XStream* of *XFrame*

A new XStream that is made up of XFrames of the columns referred to in *keylist* from each XFrame. The order of the columns is preserved.

See also:

xframes.XFrame.select_columns Corresponding function on individual frame.

select_rows(*xa*)
Selects rows of the XFrame where the XArray evaluates to True.

Parameters **xa** : *XArray*

Must be the same length as the XFrame. The filter values.

Returns *XFrame*

A new XFrame which contains the rows of the XFrame where the XArray is True. The truth test is the same as in python, so non-zero values are considered true.

static **set_checkpoint**(*checkpoint_dir*)
Set the checkpoint director for storing state.

Parameters **checkpoint_dir** : string

Path to a directory for storing checkpoints

static **start**()
Start the streaming pipeline running.

It will continue to run, processing XFrames, until stopped.

static **stop**(*stop_spark_context=True*, *stop_gracefully=False*)
Stop the streaming pipeline.

Parameters **stop_spark_context** : boolean, optional

If True, also stop the streaming context. This releases resources, but it can not be started again. If False, then streaming may be started again. Defaults to True.

stop_gracefully : boolean, optional

If True, stops gracefully by letting all operations in progress finish before stopping. Defaults to false.

swap_columns(*column_1*, *column_2*)
Swap the columns with the given names. This operation returns a new XFrame with the given columns swapped.

Parameters **column_1** : string

Name of column to swap

column_2 : string

Name of other column to swap

Returns *XStream* of *XFrame*

XStream of XFrames with specified columns swapped.

See also:

xframes.XFrame.swap_columns Corresponding function on individual frame.

to_dstream()

Convert the current XStream to a Spark DStream. The RDD contained in the DStream consists of tuples containing the column data. No conversion is necessary: the internal DStream is returned.

Returns spark.DStream

The spark DStream that is used to represent the XStream.

See also:

xframes.XFrame.to_rdd Converts to a Spark RDD. Corresponding function on individual frame.

transform_col(*col*, *fn*, *dtype*)

Transform a single column according to a specified function. The remaining columns are not modified. The type of the transformed column types becomes *dtype*, with the new value being the result of *fn(x)*, where *x* is a single row in the XFrame represented as a dictionary. The *fn* should return exactly one value which can be cast into type *dtype*.

Parameters **col** : string

The name of the column to transform.

fn : function, optional

The function to transform each row of the XFrame. The return type should be convertible to *dtype* If the function is not given, an identity function is used.

dtype : dtype, optional

The column data type of the new XArray. If None, the first 100 elements of the array are used to guess the target data type.

Returns *XStream* of *XFrame*

An XStream with the given column transformed by the function and cast to the given type.

update_state(*fn*, *col_name*, *state_column_names*, *state_column_types*)

Update state for an XStream by using the state key in a given column.

The state is a key-value store. The key is made up of the values in the given column. For each XFrame in the XStream, all the rows with a given key are passed to the supplied function, which computes a new state.

Parameters **fn** : function

The given function is supplied with a list of rows in each XFrame that have the same value in the given column (the key), along with the current state. It returns the new state for that key. The function is: fn(rows, old_state) and returns new_state.

col_name : str | None

The column of the XStream to match supplies the state key.

Returns An XStream made up of XFrames representing the state.

CHAPTER 4

Aggregate

xframes.aggregate.**ARGMAX**(*agg_column*, *out_column*)
Builtin arg maximum aggregator for groupby.

Examples

Get the movie with maximum rating per user.

```
>>> xf.groupby("user",
                {'best_movie':aggregate.ARGMAX('rating','movie')})
```

xframes.aggregate.**ARGMIN**(*agg_column*, *out_column*)
Builtin arg minimum aggregator for groupby.

Examples

Get the movie with minimum rating per user.

```
>>> xf.groupby("user",
                {'best_movie':aggregate.ARGMIN('rating','movie')})
```

xframes.aggregate.**CONCAT**(*src_column*, *dict_value_column=None*)
Builtin aggregator that combines values from one or two columns in one group into either a dictionary value, list value or array value.

Examples

To combine values from two columns that belong to one group into one dictionary value:

```
>>> xf.groupby(["document"],
               {"word_count": aggregate.CONCAT("word", "count")})
```

To combine values from one column that belong to one group into a list value:

```
>>> xf.groupby(["user"],
                {"friends": aggregate.CONCAT("friend")})
```

xframes.aggregate.**COUNT**()
Builtin count aggregator for groupby

Examples

Get the number of occurrences of each user.

```
>>> xf.groupby("user",
                {'count':aggregate.COUNT()})
```

xframes.aggregate.**MAX**(*src_column*)
Builtin maximum aggregator for groupby

Examples

Get the maximum rating of each user.

```
>>> xf.groupby("user",
                {'rating_max':aggregate.MAX('rating')})
```

xframes.aggregate.**MEAN**(*src_column*)
Builtin average aggregator for groupby.

Examples

Get the average rating of each user.

```
>>> xf.groupby("user",
                {'rating_mean':aggregate.MEAN('rating')})
```

xframes.aggregate.**MIN**(*src_column*)
Builtin minimum aggregator for groupby

Examples

Get the minimum rating of each user.

```
>>> xf.groupby("user",
                {'rating_min':aggregate.MIN('rating')})
```

xframes.aggregate.**QUANTILE**(*src_column*, **args*)
Builtin approximate quantile aggregator for groupby. Accepts as an argument, one or more of a list of quantiles to query.

Examples

To extract the median

```
>>> xf.groupby("user",
                {'rating_quantiles': aggregate.QUANTILE('rating', 0.5)})
```

To extract a few quantiles

```
>>> xf.groupby("user",
                {'rating_quantiles': aggregate.QUANTILE('rating', [0.25,0.5,
↪0.75])})
```

Or equivalently

```
>>> xf.groupby("user",
                {'rating_quantiles': aggregate.QUANTILE('rating', 0.25,0.5,0.
↪75)})
```

The returned quantiles are guaranteed to have 0.5% accuracy. That is to say, if the requested quantile is 0.50, the resultant quantile value may be between 0.495 and 0.505 of the true quantile.

xframes.aggregate.**SELECT_ONE**(*src_column*)

Builtin aggregator for groupby which selects one row in the group.

Examples

Get one rating row from a user.

```
>>> xf.groupby("user", {'rating':aggregate.SELECT_ONE('rating')})
```

If multiple columns are selected, they are guaranteed to come from the same row. For instance: >>> xf.groupby("user", {'rating':aggregate.SELECT_ONE('rating'), 'item':aggregate.SELECT_ONE('item')})

The selected 'rating' and 'item' value for each user will come from the same row in the XFrame.

xframes.aggregate.**STDV**(*src_column*)

Builtin standard deviation aggregator for groupby.

Examples

Get the rating standard deviation of each user.

```
>>> xf.groupby("user",
                {'rating_stdv':aggregate.STDV('rating')})
```

xframes.aggregate.**SUM**(*src_column*)

Builtin sum aggregator for groupby

Examples

Get the sum of the rating column for each user. >>> xf.groupby("user", {'rating_sum':aggregate.SUM('rating')})

xframes.aggregate.**VALUES**(*src_column*)

Builtin aggregator that combines distinct values from one column in one group into a list value.

Examples

To combine values from one column that belong to one group into a list value:

```
>>> xf.groupby(["user"],
                {"friends": aggregate.VALUES("friend")})
```

xframes.aggregate.**VALUES_COUNT**(*src_column*)

Builtin aggregator that combines distinct values from one column in one group into a dictionary value of unique values and their counts.

Examples

To combine values from one column that belong to one group into a dictionary of friend: count values:

```
>>> xf.groupby(["user"],
   {"friends": aggregate.VALUES_COUNT("friend")})
```

xframes.aggregate.**VARIANCE**(*src_column*)

Builtin variance aggregator for groupby.

Examples

Get the rating variance of each user.

```
>>> xf.groupby("user",
             {'rating_var':aggregate.VARIANCE('rating')})
```

CHAPTER 5

Custom Aggregator

class xframes.aggregator_property_set.**AggregatorPropertySet**(*agg_function*, *output_type*, *default_column_name*, *num_args*)

Store aggregator properties for one aggregator.

XFrames comes with an assortment of aggregators, which are used in conjunction with xframes.groupby. You can write your own aggregators if you need to.

This class is used to define custom aggregators. We will explain how by describing the builtin SUM function.

You create a function that you use in the groupby call. This function takes arguments, for instance the column name. It returns an instance of AggregatorPropertySet and also a list of the arguments passed to it. Let's examine SUM:

```
def SUM(src_column):
    return AggregatorPropertySet(agg_sum, int, 'sum', 1), [src_column]
```

In this example, SUM takes one argument, the name of the column to sum over. This argument is placed into an array and returned.

The AggregatorPropertyset instance tells groupby how to do its work. This includes the function to use to perform the aggregation (agg_sum), the type of the column to be created (int), the default name of the column if no name is given ('sum') and the number of arguments expected (1).

The agg_sum function does the actual aggregation. A simplified version of agg_sum is given below as an example.

This function takes arguments:

rows : A collection of rows. Each row is a dictionary, in the form passed to xframes.apply.

cols : A list of the arguments. These are the values returned as the second member of SUM.

Then the function (agg_sum) computes and returns the aggregated value.

Here is the code for agg_sum:

```
def agg_sum(rows, cols):
    src_col = cols[0]
    total = 0
    for row in rows:
        val = row[src_col]
        if not _is_missing(val):
            total += val
    return total
```

If you call SUM('some-col') in a groupby statement, then SUM returns the AggregatorPropertySet as shown above, and ['some-col'].

Then the groupby command is executed. For every distinct value of the grouped variable, it creates a row iterator and passes it to agg_sum, along with ['some-col']. The number of rows may be very large, so they are not all computed and passed as a list, but are provided by an iterator.

Then agg_sum extracts the desired value from each row (row[src_col]) and sums up the values.

The function agg_sum is executed in a spark worker node, as with the function used in xframes.apply, and the same restrictions apply.

__init__(*agg_function*, *output_type*, *default_column_name*, *num_args*)
Create a new instance.

Parameters **agg_function: func(rows, cols)**

The agregator function. This is given a pyspark resultIterable produced by rdd.groupByKey and containing the rows matching a single group. It's responsibility is to compute and return the aggregate value for the group.

output_type: type or int

If a type is given, use that type as the output column type. If an integer is given, then the output type is the same as the input type of the column indexed by the integer.

default_column_name: str

The name of the aggregate column, if not supplied explicitly.

num_args : int

The number of arguments to the agg_function.

CHAPTER 6

Sketch

class `xframes.`**`Sketch`**(*array=None*, *sub_sketch_keys=[]*, *impl=None*)

The Sketch object contains a sketch of a single XArray (a column of an SFrame). Using a sketch representation of an XArray, many approximate and exact statistics can be computed very quickly.

To construct a Sketch object, the following methods are equivalent:

```
>>> my_xarray = xframes.XArray([1,2,3,4,5])
>>> sketch_ctor = xframes.Sketch(my_xarray)
>>> sketch_factory = my_xarray.sketch_summary()
```

Typically, the XArray is a column of an XFrame:

```
>>> my_sframe =  xframes.XFrame({'column1': [1,2,3]})
>>> sketch_ctor = xframes.Sketch(my_sframe['column1'])
>>> sketch_factory = my_sframe['column1'].sketch_summary()
```

The sketch computation is fast, with complexity approximately linear in the length of the XArray. After the Sketch is computed, all queryable functions are performed nearly instantly.

A sketch can compute the following information depending on the dtype of the XArray:

For numeric columns, the following information is provided exactly:

- length (*`size()`*)
- number of missing Values (*`num_undefined()`*)
- minimum value (*`min()`*)
- maximum value (*`max()`*)
- mean (*`mean()`*)
- variance (*`var()`*)
- standard deviation (*`std()`*)

And the following information is provided approximately:

- number of unique values (*num_unique()*)
- quantiles (*quantile()*)
- frequent items (*frequent_items()*)
- frequency count for any value (*frequency_count()*)

For non-numeric columns(str), the following information is provided exactly:

- length (*size()*)
- number of missing values (*num_undefined()*)

And the following information is provided approximately:

- number of unique Values (*num_unique()*)
- frequent items (*frequent_items()*)
- frequency count of any value (*frequency_count()*)

For XArray of type list or array, there is a sub sketch for all sub elements. The sub sketch flattens all list/array values and then computes sketch summary over flattened values.

Element sub sketch may be retrieved through:

- element_summary(*element_summary()*)

For XArray of type dict, there are sub sketches for both dict key and value.

The sub sketch may be retrieved through:

- dict_key_summary(*dict_key_summary()*)
- dict_value_summary(*dict_value_summary()*)

For XArray of type dict, user can also pass in a list of dictionary keys to sketch_summary function, this would generate one sub sketch for each key. For example:

```
>>> sa = xframes.XArray([{'a':1, 'b':2}, {'a':3}])
>>> sketch = sa.sketch_summary(sub_sketch_keys=["a", "b"])
```

Then the sub summary may be retrieved by:

```
>>> sketch.element_sub_sketch()
```

or to get subset keys:

```
>>> sketch.element_sub_sketch(["a"])
```

Similarly, for XArray of type vector(array), user can also pass in a list of integers which is the index into the vector to get sub sketch For example:

```
>>> sa = xframes.XArray([[100,200,300,400,500], [100,200,300], [400,500]])
>>> sketch = sa.sketch_summary(sub_sketch_keys=[1,3,5])
```

Then the sub summary may be retrieved by:

```
>>> sketch.element_sub_sketch()
```

Or:

```
>>> sketch.element_sub_sketch([1,3])
```

for subset of keys.

Please see the individual function documentation for detail about each of these statistics.

Parameters **array** : XArray

Array to generate sketch summary.

References

- Wikipedia. Streaming algorithms.
- Charikar, et al. (2002) Finding frequent items in data streams.
- Cormode, G. and Muthukrishnan, S. (2004) An Improved Data Stream Summary: The Count-Min Sketch and its Applications.

__init__ (*array*)

Construct a new Sketch from an XArray.

Parameters **array** : XArray

Array to sketch.

sub_sketch_keys : list

The list of sub sketch to calculate, for XArray of dictionary type. key needs to be a string, for XArray of vector(array) type, the key needs to be positive integer

avg_length ()

Returns the average length of the values in the xarray. Returns 0 on an empty array.

The length of a value in a numeric array is 1. The length of a list or dictionary value is the length of the list or dict. The length of a string value is the string lenth.

Returns **out** : float

The average length of the values. Returns 0 if the XArray is empty.

Raises **RuntimeError**

If the xarray is a non-numeric type.

dict_key_summary ()

Returns the sketch summary for all dictionary keys. This is only valid for sketch object from an XArray of dict type. Dictionary keys are converted to strings and then do the sketch summary.

Examples

```
>>> sa = xframes.XArray([{'I':1, 'love': 2}, {'nature':3, 'beauty':4}])
>>> sa.sketch_summary().dict_key_summary()
+------------------+-------+----------+
|       item       | value | is exact |
+------------------+-------+----------+
|      Length      |   4   |   Yes    |
| # Missing Values |   0   |   Yes    |
| # unique values  |   4   |    No    |
+------------------+-------+----------+
Most frequent items:
+-------+---+------+--------+--------+
```

```
| value | I | love | beauty | nature |
+-------+---+------+--------+--------+
| count | 1 |  1   |   1    |   1    |
+-------+---+------+--------+--------+
```

dict_value_summary()

Returns the sketch summary for all dictionary values. This is only valid for sketch object from an XArray of dict type.

Type of value summary is inferred from first set of values.

Examples

```
>>> sa = xframes.XArray([{'I':1, 'love': 2}, {'nature':3, 'beauty':4}])
>>> sa.sketch_summary().dict_value_summary()
+--------------------+---------------+----------+
|        item        |     value     | is exact |
+--------------------+---------------+----------+
|       Length       |       4       |   Yes    |
|        Min         |      1.0      |   Yes    |
|        Max         |      4.0      |   Yes    |
|        Mean        |      2.5      |   Yes    |
|        Sum         |      10.0     |   Yes    |
|      Variance      |      1.25     |   Yes    |
| Standard Deviation | 1.11803398875 |   Yes    |
|  # Missing Values  |       0       |   Yes    |
|  # unique values   |       4       |    No    |
+--------------------+---------------+----------+
Most frequent items:
+-------+-----+-----+-----+-----+
| value | 1.0 | 2.0 | 3.0 | 4.0 |
+-------+-----+-----+-----+-----+
| count |  1  |  1  |  1  |  1  |
+-------+-----+-----+-----+-----+
Quantiles:
+-----+-----+-----+-----+-----+-----+-----+-----+------+
|  0% |  1% |  5% | 25% | 50% | 75% | 95% | 99% | 100% |
+-----+-----+-----+-----+-----+-----+-----+-----+------+
| 1.0 | 1.0 | 1.0 | 2.0 | 3.0 | 4.0 | 4.0 | 4.0 | 4.0  |
+-----+-----+-----+-----+-----+-----+-----+-----+------+
```

element_length_summary()

Returns the sketch summary for the element length. This is only valid for a sketch constructed XArray of type list/array/dict, raises Runtime exception otherwise.

Returns **out** : Sketch

An new sketch object regarding the element length of the current XArray

Examples

```
>>> sa = xframes.XArray([[j for j in range(i)] for i in range(1,1000)])
>>> sa.sketch_summary().element_length_summary()
+--------------------+---------------+----------+
|        item        |     value     | is exact |
```

```
+--------------------+---------------+----------+
|       Length       |      999      |   Yes    |
|        Min         |      1.0      |   Yes    |
|        Max         |     999.0     |   Yes    |
|        Mean        |     500.0     |   Yes    |
|        Sum         |    499500.0   |   Yes    |
|      Variance      | 83166.6666667 |   Yes    |
| Standard Deviation | 288.386314978 |   Yes    |
|  # Missing Values  |       0       |   Yes    |
|  # unique values   |      992      |    No    |
+--------------------+---------------+----------+
Most frequent items:
+-------+---+---+---+---+---+---+---+---+---+----+
| value | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
+-------+---+---+---+---+---+---+---+---+---+----+
| count | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1  |
+-------+---+---+---+---+---+---+---+---+---+----+
Quantiles:
+-----+------+------+-------+-------+-------+-------+-------+-------+
|  0% |  1%  |  5%  |  25%  |  50%  |  75%  |  95%  |  99%  |  100% |
+-----+------+------+-------+-------+-------+-------+-------+-------+
| 1.0 | 10.0 | 50.0 | 250.0 | 500.0 | 750.0 | 950.0 | 990.0 | 999.0 |
+-----+------+------+-------+-------+-------+-------+-------+-------+
```

element_sub_sketch(*keys=None*)

Returns the sketch summary for the given set of keys. This is only applicable for sketch summary created from XArray of xarray or dict type. For dict XArray, the keys are the keys in dict value. For array Xarray, the keys are indexes into the array value.

The keys must be passed into original sketch_summary() call in order to be able to be retrieved later

Parameters **keys** : list of str | str | list of int | int

The list of dictionary keys or array index to get sub sketch from. if not given, then retrieve all sub sketches that are available

Returns A dictionary that maps from the key(index) to the actual sketch summary

for that key(index)

Examples

```
>>> sa = xframes.XArray([{'a':1, 'b':2}, {'a':4, 'd':1}])
>>> s = sa.sketch_summary(sub_sketch_keys=['a','b'])
>>> s.element_sub_sketch(['a'])
{'a':
 +--------------------+-------+----------+
 |        item        | value | is exact |
 +--------------------+-------+----------+
 |       Length       |   2   |   Yes    |
 |        Min         |  1.0  |   Yes    |
 |        Max         |  4.0  |   Yes    |
 |        Mean        |  2.5  |   Yes    |
 |        Sum         |  5.0  |   Yes    |
 |      Variance      |  2.25 |   Yes    |
 | Standard Deviation |  1.5  |   Yes    |
 |  # Missing Values  |   0   |   Yes    |
 |  # unique values   |   2   |    No    |
```

```
+--------------------+-------+----------+
Most frequent items:
+-------+-----+-----+
| value | 1.0 | 4.0 |
+-------+-----+-----+
| count |  1  |  1  |
+-------+-----+-----+
Quantiles:
+-----+-----+-----+-----+-----+-----+-----+-----+------+
|  0% |  1% |  5% | 25% | 50% | 75% | 95% | 99% | 100% |
+-----+-----+-----+-----+-----+-----+-----+-----+------+
| 1.0 | 1.0 | 1.0 | 1.0 | 4.0 | 4.0 | 4.0 | 4.0 | 4.0  |
+-----+-----+-----+-----+-----+-----+-----+-----+------+}
```

element_summary()

Returns the sketch summary for all element values. This is only valid for sketch object created from XArray of list or vector(array) type. For XArray of list type, all list values are treated as string for sketch summary. For XArray of vector type, the sketch summary is on FLOAT type.

Examples

```
>>> sa = xframes.XArray([[1,2,3], [4,5]])
>>> sa.sketch_summary().element_summary()
+--------------------+---------------+----------+
|        item        |     value     | is exact |
+--------------------+---------------+----------+
|       Length       |       5       |   Yes    |
|        Min         |      1.0      |   Yes    |
|        Max         |      5.0      |   Yes    |
|        Mean        |      3.0      |   Yes    |
|        Sum         |      15.0     |   Yes    |
|      Variance      |      2.0      |   Yes    |
| Standard Deviation | 1.41421356237 |   Yes    |
|  # Missing Values  |       0       |   Yes    |
|  # unique values   |       5       |    No    |
+--------------------+---------------+----------+
Most frequent items:
+-------+-----+-----+-----+-----+-----+
| value | 1.0 | 2.0 | 3.0 | 4.0 | 5.0 |
+-------+-----+-----+-----+-----+-----+
| count |  1  |  1  |  1  |  1  |  1  |
+-------+-----+-----+-----+-----+-----+
Quantiles:
+-----+-----+-----+-----+-----+-----+-----+-----+------+
|  0% |  1% |  5% | 25% | 50% | 75% | 95% | 99% | 100% |
+-----+-----+-----+-----+-----+-----+-----+-----+------+
| 1.0 | 1.0 | 1.0 | 2.0 | 3.0 | 4.0 | 5.0 | 5.0 | 5.0  |
+-----+-----+-----+-----+-----+-----+-----+-----+------+
```

frequency_count(*element*)

Returns a sketched estimate of the number of occurrences of a given element. This estimate is based on the count sketch. The element type must be of the same type as the input XArray. Throws an exception if element is of the incorrect type.

Parameters **element** : val

An element of the same type as the XArray.

Returns **out** : int

An estimate of the number of occurrences of the element.

Raises **RuntimeError**

Throws an exception if element is of the incorrect type.

frequent_items()

Returns a sketched estimate of the most frequent elements in the XArray based on the SpaceSaving sketch. It is only guaranteed that all elements which appear in more than 0.01% rows of the array will appear in the set of returned elements. However, other elements may also appear in the result. The item counts are estimated using the CountSketch.

Missing values are not taken into account when copmuting frequent items.

If this function returns no elements, it means that all elements appear with less than 0.01% occurrence.

Returns **out** : dict

A dictionary mapping items and their estimated occurrence frequencies.

max()

Returns the maximum value in the XArray. Returns *nan* on an empty array. Throws an exception if called on an XArray with non-numeric type.

Returns **out** : type of XArray

Maximum value of XArray. Returns *nan* if the XArray is empty.

Raises **RuntimeError**

Throws an exception if the XArray is a non-numeric type.

mean()

Returns the mean of the values in the XArray. Returns 0 on an empty array. Throws an exception if called on an XArray with non-numeric type.

Returns **out** : float

Mean of all values in XArray. Returns 0 if the xarray is empty.

Raises **RuntimeError**

If the xarray is a non-numeric type.

min()

Returns the minimum value in the XArray. Returns *nan* on an empty array. Throws an exception if called on an XArray with non-numeric type.

Returns **out** : type of XArray

Minimum value of XArray. Returns *nan* if the xarray is empty.

Raises **RuntimeError**

If the xarray is a non-numeric type.

num_undefined()

Returns the the number of undefined elements in the XArray. Return 0 on an empty XArray.

Returns **out** : int

The number of missing values in the XArray.

num_unique()
Returns a sketched estimate of the number of unique values in the XArray based on the Hyperloglog sketch.

Returns out : float

An estimate of the number of unique values in the XArray.

quantile(*quantile_val*)
Returns a sketched estimate of the value at a particular quantile between 0.0 and 1.0. The quantile is guaranteed to be accurate within 1%: meaning that if you ask for the 0.55 quantile, the returned value is guaranteed to be between the true 0.54 quantile and the true 0.56 quantile. The quantiles are only defined for numeric arrays and this function will raise an exception if called on a sketch constructed for a non-numeric column.

Parameters quantile_val : float

A value between 0.0 and 1.0 inclusive. Values below 0.0 will be interpreted as 0.0. Values above 1.0 will be interpreted as 1.0.

Returns out : float | str

An estimate of the value at a quantile.

Raises RuntimeError

If the xarray is a non-numeric type.

set_frequency_sketch_parms(*num_items=None*, *epsilon=None*, *delta=None*)
Set the frequency sketch accuracy settings.

Parameters num_items: int, optional

The number "most frequent" values that are tracked.

epsilon: float (0 .. 1.0), optional

The precision of the result

delta: float (0 .. 1.0), optional

The probability that the precision specified above is not achieved.

set_quantile_accumulator_parms(*num_levels=None*, *epsilon=None*, *delta=None*)
Set the quantile accumulator accuracy settings.

Parameters num_levels: int, optional

The number of levels of hash map.

epsilon: float (0 .. 1.0), optional

The precision of the result

delta: float (0 .. 1.0), optional

The probability that the precision specified above is not achieved.

size()
Returns the size of the input XArray.

Returns out : int

The number of elements of the input XArray.

std()

Returns the standard deviation of the values in the XArray. Returns 0 on an empty array. Throws an exception if called on an XArray with non-numeric type.

Returns **out** : float

The standard deviation of all the values. Returns 0 if the xarray is empty.

Raises **RuntimeError**

If the xarray is a non-numeric type.

sum()

Returns the sum of all the values in the XArray. Returns 0 on an empty array. Throws an exception if called on an xarray with non-numeric type. Will overflow without warning.

Returns **out** : type of XArray

Sum of all values in XArray. Returns 0 if the XArray is empty.

Raises **RuntimeError**

If the xarray is a non-numeric type.

tf_idf()

Returns a tf-idf analysis of each document in a collection.

If the elements in the column are documents in string form, then a simple splitter is used to create a list of words.

If the elemenst are already in list form, then the list elements are used as the terms. These are usually strings, but could be numeric instead.

Returns **out** : XArray of dict

For each document, a dictionary mapping terms to their tf_idf score.

var()

Returns the variance of the values in the xarray. Returns 0 on an empty array. Throws an exception if called on an XArray with non-numeric type.

Returns **out** : float

The variance of all the values. Returns 0 if the XArray is empty.

Raises **RuntimeError**

If the xarray is a non-numeric type.

CHAPTER 7

XPlot

class xframes.**XPlot**(*axes=None*, *alpha=None*)

Plotting library for XFrames.

Creates simple data plots.

Parameters **axes** : list, optional

The size of the axes. Should be a four-element list. [x_origin, y_origin, x_length, y_length] Defaults to [0.0, 0.0, 1.5, 1.0]

alpha : float, optional

The opacity of the plot.

__init__(*axes=None*, *alpha=None*)

Create a plotting object.

Parameters **axes** : list, optional

The size of the axes. Should be a four-element list. [x_origin, y_origin, x_length, y_length] Defaults to [0.0, 0.0, 1.5, 1.0]

alpha : float, optional

The opacity of the plot.

col_info(*column*, *column_name=None*, *table_name=None*, *title=None*, *topk=None*, *bins=None*, *cut-off=False*)

Print column summary information.

The number of the most frequent values is shown. If the column to summarize is numerical or datetime, then a histogram is also shown.

Parameters **column** : XArray

The column to summarize.

column_name : str

The column name.

table_name : str, optional

The table name; used to labeling only. The table that us used for the data is given in the constructor.

title : str, optional

The plot title.

topk: int, optional

The number of frequent items to show.

bins : int, optional

The number of bins in a histogram.

cutoff : float, optional

The number to use as an upper cutoff, if the plot is a histogram.

Examples

(Need examples)

frequent_values(*column*, *k=15*, *title=None*, *append_counts_to_label=False*, *normalize=False*, *xlabel=None*, *ylabel=None*, *epsilon=None*, *delta=None*, *num_items=None*)

Plots the number of occurances of specific values in a column.

The most frequent values are plotted.

Parameters **column** : XArray

The column to plot. The number of distinct occurrances of each value is calculated and plotted.

k : int, optional

The number of different values to plot. Defaults to 15.

title : str, optional

A plot title.

append_counts_to_label : boolean, optional

If true, append the bar count to the label

normalize : bool, optional

If true, plot percentages instead of counts. Defaults to False.

xlabel : str, optional

A label for the X axis.

ylabel : str, optional

A label for the Y axis.

epsilon : float, optional

Governs accuracy of frequency counter.

delta : float, optional

Governs accuracy of frequency counter.

num_items : float, optional

Governs accuracy of frequency counter.

Returns list of tuples

List of (value, count) for the most frequent "k" values

Examples

(Need examples)

histogram(*column*, *title=None*, *bins=None*, *sketch=None*, *xlabel=None*, *ylabel=None*, *lower_cutoff=0.0*, *upper_cutoff=1.0*, *lower_bound=None*, *upper_bound=None*)

Plot a histogram.

All values greater than the cutoff (given as a quantile) are set equal to the cutoff.

Parameters **column** : XArray

A column to display.

title : str, optional

A plot title.

bins : int, optional

The number of bins to use. Defaults to 50.

sketch : Sketch, optional

The column sketch. If this is available, then it saves time not to recompute it.

xlabel : str, optional

A label for the X axis.

ylabel : str, optional

A label for the Y axis.

lower_cutoff : float, optional

This is a quantile value, between 0 and 1. Values below this cutoff are placed in the first bin. Defaults to 0.

upper_cutoff : float, optional

This is a quantile value, between 0 and 1. Values above this cutoff are placed in the last bin. Defaults to 1.0.

lower_bound : float, optional

Values below this bound are placed in the first bin.

upper_bound : float, optional

Values below this bound are placed in the last bin.

bins : int, optional

The number of bins to use. Defaults to 50.

Examples

(Need examples)

top_values (*xf*, *x_col*, *y_col*, *k=15*, *title=None*, *xlabel=None*, *ylabel=None*)
Plot the top values of a column of data.

Parameters **xf** : XFrame

An XFrame containing the columns to be plotted.

x_col : str

A column name: the top values in this column are plotted. These values must be numerical.

y_col : str

A column name: the values in this colum will be used to label the corresponding values in the x column.

k : int, optional

The number of values to plot. Defaults to 15.

title : str, optional

A plot title.

xlabel : str, optional

A label for the X axis.

ylabel : str, optional

A label for the Y axis.

Examples

(Come up with an example)

CHAPTER 8

Spark Context

Provides functions to create and maintain the spark context.

class xframes.spark_context.**SparkInitContext**

Spark Context initialization.

This may be used to initialize the spark context with the supplied values. If this mechanism is not used, then the spark context will be initialized using the config file the first time a context is needed.

static set(*context*)

Sets the spark context parameters, and then creates a context. If the spark context has already been created, then this will have no effect.

Parameters **context** : dict

Dictionary of property/value pairs. These are passed to spark as config parameters. If a config file is present, these parameters will override the parameters there.

Notes

The following values are the most commonly used. They will be given default values if none are supplied in a configuration file. Other values can be found in the spark configuration documentation.

spark.master [str, optional] The url of the spark cluster to use. To use the local spark, give 'local'. To use a spark cluster with its master on a specific IP address, give the IP address or the hostname as in the following examples:

spark.master=spark://my_spark_host:7077

spark.master=mesos://my_mesos_host:5050

app.name [str, optional] The app name is used on the job monitoring server, and for logging.

spark.cores.max [str, optional] The maximum number of cores to use for execution.

spark.executor.memory [str, optional] The amount of main memory to allocate to executors. For example, '2g'.

CHAPTER 9

Indices and Tables

- genindex
- modindex
- search

Python Module Index

x

Index

Symbols

A

C

D

E

T

U

V

W

X