graphlab.SFrame

class `graphlab.` `SFrame` ***(data=list(), format='auto')***

A tabular, column-mutable dataframe object that can scale to big data. The data in SFrame is stored column-wise on the GraphLab Server side, and is stored on persistent storage (e.g. disk) to avoid being constrained by memory size. Each column in an SFrame is a size-immutable `SArray`, but SFrames are mutable in that columns can be added and subtracted with ease. An SFrame essentially acts as an ordered dict of SArrays.

Currently, we support constructing an SFrame from the following data formats:

- csv file (comma separated value)
- sframe directory archive (A directory where an sframe was saved previously)
- general text file (with csv parsing options, See `read_csv()`)
- a Python dictionary
- pandas.DataFrame
- JSON
- Apache Avro
- PySpark RDD

and from the following sources:

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

Only basic examples of construction are covered here. For more information and examples, please see the User Guide, API Translator, How-Tos, and data science Gallery.

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

The actual interpretation of this field is dependent on the `format` parameter. If `data` is an array or Pandas DataFrame, the contents are stored in the SFrame. If `data` is a string, it is interpreted as a file. Files can be read from local file system or urls (local://, hdfs://, s3://, http://).

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"
- "sarray"
- "dataframe"
- "csv"
- "tsv"
- "sframe".

See also

`read_csv`

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

`save`

Save an SFrame for later use.

Notes

- When reading from HDFS on Linux we must guess the location of your java installation. By default, we will use the location pointed to by the JAVA_HOME environment variable. If this is not set, we check many common installation paths. You may use two environment variables to override this behavior. GRAPHLAB_JAVA_HOME allows you to specify a specific java installation and overrides JAVA_HOME. GRAPHLAB_LIBJVM_DIRECTORY overrides all and expects the exact directory that your preferred libjvm.so file is located. Use this ONLY if you'd like to use a non-standard JVM.

Examples

```
>>> import graphlab
>>> from graphlab import SFrame
```

Construction

Construct an SFrame from a dataframe and transfers the dataframe object across the network.

```
>>> df = pandas.DataFrame()
>>> sf = SFrame(data=df)
```

Construct an SFrame from a local csv file (only works for local server).

```
>>> sf = SFrame(data='~/mydata/foo.csv')
```

Construct an SFrame from a csv file on Amazon S3. This requires the environment variables: *AWS_ACCESS_KEY_ID* and *AWS_SECRET_ACCESS_KEY* to be set before the python session started. Alternatively, you can use `graphlab.aws.set_credentials()` to set the credentials after python is started and `graphlab.aws.get_credentials()` to verify these environment variables.

```
>>> sf = SFrame(data='s3://mybucket/foo.csv')
```

Read from HDFS using a specific java installation (environment variable only applies when using Linux)

```
>>> import os
>>> os.environ['GRAPHLAB_JAVA_HOME'] = '/my/path/to/java'
>>> from graphlab import SFrame
>>> sf = SFrame("hdfs://mycluster.example.com:8020/user/myname/coolfile.txt")
```

An SFrame can be constructed from a dictionary of values or SArrays:

```
>>> sf = gl.SFrame({'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
```

Or equivalently:

```
>>> ids = SArray([1,2,3])
>>> vals = SArray(['A','B','C'])
>>> sf = SFrame({'id':ids,'val':vals})
```

It can also be constructed from an array of SArrays in which case column names are automatically assigned.

```
>>> ids = SArray([1,2,3])
>>> vals = SArray(['A','B','C'])
>>> sf = SFrame([ids, vals])
>>> sf
Columns:
    X1 int
    X2 str
Rows: 3
Data:
   X1  X2
0  1   A
1  2   B
2  3   C
```

If the SFrame is constructed from a list of values, an SFrame of a single column is constructed.

```
>>> sf = SFrame([1,2,3])
>>> sf
Columns:
    X1 int
Rows: 3
Data:
   X1
0  1
1  2
2  3
```

Parsing

The `graphlab.SFrame.read_csv()` is quite powerful and, can be used to import a variety of row-based formats.

First, some simple cases:

```
>>> !cat ratings.csv
user_id,movie_id,rating
10210,1,1
10213,2,5
10217,2,2
10102,1,3
10109,3,4
10117,5,2
10122,2,4
10114,1,5
10125,1,1
>>> gl.SFrame.read_csv('ratings.csv')
Columns:
  user_id   int
  movie_id  int
  rating    int
Rows: 9
Data:
+---------+----------+--------+
| user_id | movie_id | rating |
+---------+----------+--------+
|  10210  |    1     |   1    |
|  10213  |    2     |   5    |
|  10217  |    2     |   2    |
|  10102  |    1     |   3    |
|  10109  |    3     |   4    |
|  10117  |    5     |   2    |
|  10122  |    2     |   4    |
|  10114  |    1     |   5    |
|  10125  |    1     |   1    |
+---------+----------+--------+
[9 rows x 3 columns]
```

Delimiters can be specified, if "," is not the delimiter, for instance space ' ' in this case. Only single character delimiters are supported.

```
>>> !cat ratings.csv
user_id movie_id rating
10210 1 1
10213 2 5
10217 2 2
10102 1 3
10109 3 4
10117 5 2
10122 2 4
10114 1 5
10125 1 1
>>> gl.SFrame.read_csv('ratings.csv', delimiter=' ')
```

By default, "NA" or a missing element are interpreted as missing values.

```
>>> !cat ratings2.csv
user,movie,rating
"tom",,1
harry,5,
jack,2,2
bill,,
>>> gl.SFrame.read_csv('ratings2.csv')
Columns:
  user  str
  movie int
  rating    int
Rows: 4
Data:
+---------+-------+--------+
|   user  | movie | rating |
+---------+-------+--------+
|   tom   |  None |   1    |
|  harry  |   5   |  None  |
|   jack  |   2   |   2    |
| missing |  None |  None  |
+---------+-------+--------+
[4 rows x 3 columns]
```

Furthermore due to the dictionary types and list types, can handle parsing of JSON-like formats.

```
>>> !cat ratings3.csv
business, categories, ratings
"Restaurant 1", [1 4 9 10], {"funny":5, "cool":2}
"Restaurant 2", [], {"happy":2, "sad":2}
"Restaurant 3", [2, 11, 12], {}
>>> gl.SFrame.read_csv('ratings3.csv')
Columns:
business    str
categories  array
ratings dict
Rows: 3
Data:
+--------------+--------------------------------+-------------------------+
|   business   |           categories           |         ratings         |
+--------------+--------------------------------+-------------------------+
| Restaurant 1 | array('d', [1.0, 4.0, 9.0, ... | {'funny': 5, 'cool': 2} |
| Restaurant 2 |           array('d')           |  {'sad': 2, 'happy': 2} |
| Restaurant 3 | array('d', [2.0, 11.0, 12.0])  |            {}           |
+--------------+--------------------------------+-------------------------+
[3 rows x 3 columns]
```

The list and dictionary parsers are quite flexible and can absorb a variety of purely formatted inputs. Also, note that the list and dictionary types are recursive, allowing for arbitrary values to be contained.

All these are valid lists:

```
>>> !cat interesting_lists.csv
list
[]
[1,2,3]
[1;2,3]
[1 2 3]
[{a:b}]
["c",d, e]
[[a]]
>>> gl.SFrame.read_csv('interesting_lists.csv')
Columns:
  list  list
Rows: 7
Data:
+-----------------+
|       list      |
+-----------------+
|        []       |
|    [1, 2, 3]    |
|    [1, 2, 3]    |
|    [1, 2, 3]    |
|   [{'a': 'b'}]  |
| ['c', 'd', 'e'] |
|     [['a']]     |
+-----------------+
[7 rows x 1 columns]
```

All these are valid dicts:

```
>>> !cat interesting_dicts.csv
dict
{"classic":1,"dict":1}
{space:1 seperated:1}
{emptyvalue:}
{}
{:}
{recursive1:[{a:b}]}
{:[{:[a]}]}
>>> gl.SFrame.read_csv('interesting_dicts.csv')
Columns:
  dict  dict
Rows: 7
Data:
+------------------------------+
|             dict             |
+------------------------------+
|  {'dict': 1, 'classic': 1}   |
| {'seperated': 1, 'space': 1} |
|     {'emptyvalue': None}     |
|              {}              |
|         {None: None}         |
| {'recursive1': [{'a': 'b'}]} |
| {None: [{None: array('d')}]} |
+------------------------------+
[7 rows x 1 columns]
```

Saving

Save and load the sframe in native format.

```
>>> sf.save('mysframedir')
>>> sf2 = graphlab.load_sframe('mysframedir')
```

Column Manipulation

An SFrame is composed of a collection of columns of SArrays, and individual SArrays can be extracted easily. For instance given an SFrame:

```
>>> sf = SFrame({'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
```

The “id” column can be extracted using:

```
>>> sf["id"]
dtype: int
Rows: 3
[1, 2, 3]
```

And can be deleted using:

```
>>> del sf["id"]
```

Multiple columns can be selected by passing a list of column names:

```
>>> sf = SFrame({'id':[1,2,3],'val':['A','B','C'],'val2':[5,6,7]})
>>> sf
Columns:
    id   int
    val  str
    val2 int
Rows: 3
Data:
   id  val val2
0  1   A   5
1  2   B   6
2  3   C   7
>>> sf2 = sf[['id','val']]
>>> sf2
Columns:
    id  int
    val str
Rows: 3
Data:
   id  val
0  1   A
1  2   B
2  3   C
```

You can also select columns using types or a list of types:

```
>>> sf2 = sf[int]
>>> sf2
Columns:
    id   int
    val2 int
Rows: 3
Data:
   id  val2
0  1   5
1  2   6
2  3   7
```

Or a mix of types and names:

```
>>> sf2 = sf[['id', str]]
>>> sf2
Columns:
    id  int
    val str
Rows: 3
Data:
   id  val
0  1   A
1  2   B
2  3   C
```

The same mechanism can be used to re-order columns:

```
>>> sf = SFrame({'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
>>> sf[['val','id']]
>>> sf
Columns:
    val str
    id  int
Rows: 3
Data:
   val id
0  A   1
1  B   2
2  C   3
```

Element Access and Slicing

SFrames can be accessed by integer keys just like a regular python list. Such operations may not be fast on large datasets so looping over an SFrame should be avoided.

```
>>> sf = SFrame({'id':[1,2,3],'val':['A','B','C']})
>>> sf[0]
{'id': 1, 'val': 'A'}
>>> sf[2]
{'id': 3, 'val': 'C'}
>>> sf[5]
IndexError: SFrame index out of range
```

Negative indices can be used to access elements from the tail of the array

```
>>> sf[-1] # returns the last element
{'id': 3, 'val': 'C'}
>>> sf[-2] # returns the second to last element
{'id': 2, 'val': 'B'}
```

The SFrame also supports the full range of python slicing operators:

```
>>> sf[1000:] # Returns an SFrame containing rows 1000 to the end
>>> sf[:1000] # Returns an SFrame containing rows 0 to row 999 inclusive
>>> sf[0:1000:2] # Returns an SFrame containing rows 0 to row 1000 in steps of 2
>>> sf[-100:] # Returns an SFrame containing last 100 rows
>>> sf[-100:len(sf):2] # Returns an SFrame containing last 100 rows in steps of 2
```

Logical Filter

An SFrame can be filtered using

```
>>> sframe[binary_filter]
```

where sframe is an SFrame and binary_filter is an SArray of the same length. The result is a new SFrame which contains only rows of the SFrame where its matching row in the binary_filter is non zero.

This permits the use of boolean operators that can be used to perform logical filtering operations. For instance, given an SFrame

```
>>> sf
Columns:
    id  int
    val str
Rows: 3
Data:
   id  val
0  1   A
1  2   B
2  3   C
```

```
>>> sf[(sf['id'] >= 1) & (sf['id'] <= 2)]
Columns:
    id  int
    val str
Rows: 3
Data:
   id  val
0  1   A
1  2   B
```

See `SArray` for more details on the use of the logical filter.

This can also be used more generally to provide filtering capability which is otherwise not expressible with simple boolean functions. For instance:

```
>>> sf[sf['id'].apply(lambda x: math.log(x) <= 1)]
Columns:
    id  int
    val str
Rows: 3
Data:
   id  val
0  1   A
1  2   B
```

Or alternatively:

```
>>> sf[sf.apply(lambda x: math.log(x['id']) <= 1)]
```

Create an SFrame from a Python dictionary.

```
>>> from graphlab import SFrame
>>> sf = SFrame({'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
```

Methods

`SFrame.add_column` (data[, name])	Add a column to this SFrame.
`SFrame.add_columns` (data[, namelist])	Adds multiple columns to this S
`SFrame.add_row_number` ([column_name, start])	Returns a new SFrame with a n
`SFrame.append` (other)	Add the rows of an SFrame to t
`SFrame.apply` (fn[, dtype, seed])	Transform each row to an `SArr`
`SFrame.column_names` ()	The name of each column in th
`SFrame.column_types` ()	The type of each column in the
`SFrame.copy` ()	Returns a shallow copy of the s
`SFrame.dropna` ([columns, how])	Remove missing values from ar
`SFrame.dropna_split` ([columns, how])	Split rows with missing values f
`SFrame.dtype` ()	The type of each column.
`SFrame.export_csv` (filename[, delimiter, ...])	Writes an SFrame to a CSV file.
`SFrame.export_json` (filename[, orient])	Writes an SFrame to a CSV file.
`SFrame.fillna` (column, value)	Fill all missing values with a giv
`SFrame.filter_by` (values, column_name[, exclude])	Filter an SFrame by values insid
`SFrame.flat_map` (column_names, fn[, ...])	Map each row of the SFrame to
`SFrame.from_odbc` (db, sql[, verbose])	Convert a table or query from a
`SFrame.from_rdd` (rdd, cur_sc)	Convert a Spark RDD into an SF
`SFrame.groupby` (key_columns, operations, *args)	Perform a group on the key_co
`SFrame.head` ([n])	The first n rows of the SFrame.

`SFrame.join` (right[, on, how])	Merge two SFrames.
`SFrame.num_cols` ()	The number of columns in this
`SFrame.num_columns` ()	The number of columns in this
`SFrame.num_rows` ()	The number of rows in this SFr
`SFrame.pack_columns` ([columns, ...])	Pack columns of the current SF
`SFrame.print_rows` ([num_rows, num_columns, ...])	Print the first M rows and N col
`SFrame.random_split` (fraction[, seed])	Randomly split the rows of an S
`SFrame.read_csv` (url[, delimiter, header, ...])	Constructs an SFrame from a C
`SFrame.read_csv_with_errors` (url[, ...])	Constructs an SFrame from a C
`SFrame.read_json` (url[, orient])	Reads a JSON file representing
`SFrame.remove_column` (name)	Remove a column from this SF
`SFrame.remove_columns` (column_names)	Remove one or more columns
`SFrame.rename` (names)	Rename the given columns.
`SFrame.sample` (fraction[, seed])	Sample the current SFrame's ro
`SFrame.save` (filename[, format])	Save the SFrame to a file syster
`SFrame.select_column` (key)	Get a reference to the `SArray` t
`SFrame.select_columns` (keylist)	Selects all columns where the r
`SFrame.show` ([columns, view, x, y])	Visualize the SFrame with Grap
`SFrame.sort` (sort_columns[, ascending])	Sort current SFrame by the give
`SFrame.split_datetime` (expand_column[, ...])	Splits a datetime column of SFr
`SFrame.stack` (column_name[, new_column_name, ...])	Convert a "wide" column of an
`SFrame.swap_columns` (column_1, column_2)	Swap the columns with the giv
`SFrame.tail` ([n])	The last n rows of the SFrame.
`SFrame.to_dataframe` ()	Convert this SFrame to pandas.
`SFrame.to_odbc` (db, table_name[, ...])	Convert an SFrame to a table ir
`SFrame.to_rdd` (sc[, number_of_partitions])	Convert the current SFrame to
`SFrame.to_spark_dataframe` (sc, sql[, ...])	Convert the current SFrame to
`SFrame.topk` (column_name[, k, reverse])	Get top k rows according to th
`SFrame.unique` ()	Remove duplicate rows of the S

`SFrame.unpack`(unpack_column[, ...])	Expand one column of this SFr
`SFrame.unstack`(column[, new_column_name])	Concatenate values from one c

Attributes

`SFrame.shape`	The shape of the SFrame, in a tuple.