

METHODS	TYPES	FUNCTORS
append_column( )	enum class drop_policy { }	struct AutoCorrVisitor { }
append_index( )	enum class fill_policy { }	struct BetaVisitor { }
bucketize( )	enum class io_format { }	struct CorrVisitor { }
bucketize_async( )	enum class join_policy { }	struct CovVisitor { }
create_column( )	enum class nan_policy { }	struct CumMaxVisitor { }
drop_missing( )	enum class return_policy { }	struct CumMinVisitor { }
fill_missing( )	enum class shift_policy { }	struct CumProdVisitor { }
gen_datetime_index( )	enum class sort_state { }	struct CumSumVisitor { }
get_col_unique_values( )	enum class time_frequency { }	struct DotProdVisitor { }
get_column( )	struct BadRange { }	struct GroupbySum { }
get_data_by_idx( )	struct ColNotFound { }	struct KthValueVisitor { }
get_data_by_loc( )	struct DataFrameError { }	struct MaxVisitor { }
get_index( )	struct InconsistentData { }	struct MeanVisitor { }
get_row( )	struct Index2D { }	struct MedianVisitor { }
get_view_by_idx( )	struct NotFeasible { }	struct MinVisitor { }
get_view_by_loc( )	struct NotImplemented { }	struct ModeVisitor { }
groupby( )		struct NLargestVisitor { }
groupby_async( )		struct NSmallestVisitor { }
is_equal( )		struct ProdVisitor { }
join_by_index( )		struct ReturnVisitor { }
load_column( )		struct SLRegressionVisitor { }
load_data( )		struct StatsVisitor { }
load_index( )		struct StdVisitor { }
make_consistent( )		struct SumVisitor { }
modify_by_idx( )		struct TrackingErrorVisitor { }
multi_visit( )		
read( )		
read_async( )		
remove_column( )		
remove_data_by_idx( )		
remove_data_by_loc( )		
rename_column( )		
replace( )		
rotate( )		
self_bucketize( )		
self_rotate( )		
self_shift( )		
shift( )		
single_act_visit( )		
sort( )		
sort_async( )		
transpose( )		
value_counts( )		
visit( )		
write( )		
write_async( )		

## MOTIVATION

Although Pandas has a spot-on interface and it is full of useful functionalities, it lacks performance and scalability. For example, it is hard to decipher high-frequency intraday data such as Options data or S&P500 constituents tick-by-tick data using Pandas.

Another issue I have encountered often is the research is done using Python, because it has such tools as Pandas, but the execution in production is in C++ for its efficiency, reliability and scalability. Therefore, there is this translation, or sometimes a bridge, between research and executions.

Also, in this day and age, C++ needs a heterogeneous data container.

Mainly because of these factors, I implemented the C++ DataFrame.

*This library could still have more functionalities compared with Pandas. I welcome all contributions from people with expertise, interest, and time to do it. I will add more functionalities from time to time, but currently I don't have much free time.*

**Views** were recently added. This is a very interesting/useful concept that even Pandas doesn't have it currently. A view is a slice of a DataFrame that is a reference to the original DataFrame. It appears exactly the same as a DataFrame, but if you modify any data in the view, the original DataFrame will also be modified.

There are certain things you cannot do in views. For example, you cannot add to delete columns, extend the index column, ...

For more understanding, look at this document further and/or the test files.

**Visitors** are the mechanism to run statistical algorithms. Most of DataFrame statistical algorithms are in "visitors". Visitor is the mechanism by which DataFrame passes data points to your algorithm. You can add your own algorithms to a visitor functor and extend DataFrame easily. There are two kinds of visit mechanisms in DataFrame:

- 1) Regular visit (`visit()`). In this case DataFrame passes the given column(s) data points one-by-one to the visitor functor. This is convenient for algorithms that can operate per data point, such as correlation, variance etc.
- 2) Single-action visit (`single_act_visit()`). In this case a reference to the given column(s) are passed to the visitor functor at once. This is necessary for algorithms that need the whole data together, such as return, median, etc.

See this document and *dataframe\_tester.cc* for more examples and documentation.

## CODE STRUCTURE

The DataFrame library is almost a header-only library with a few small source files exceptions, *HeteroVector.cc* and *HeteroView.cc*. Also there is *DateTime.cc*.

Starting from the root directory;

*include* directory contains most of the code. It includes *.h* and *.tcc* files. The latter are C++ template code files. The main header file is *DataFrame.h*. It contains the entire DataFrame object and its interface. There are comprehensive comments for each interface call in that file. The rest of the files there will show you how the sausage is made.

*src* directory has a few source files, as explained above, make-files, and test program source files. The main test source file is *dataframe\_tester.cc*. It contains test cases for all functionalities of DataFrame. It is not in a very organized structure. I plan to make the test cases more organized.

## BUILD INSTRUCTIONS

### USING PLAIN MAKE AND MAKE-FILES

Go to the root of the repository, where license file is, and execute *build\_all.sh*. This will build the library and test executables for Linux flavors.

### USING CMAKE

Please see README file. Thanks to [@justinjk007](#), you should be able to build this in Linux, Windows, Mac, and more

## EXAMPLE

This library is based on a heterogeneous vector. The heterogeneity is achieved by using static STL or STL-like vectors. Since C++ is a strongly typed language, you still have to know your column types per container at compile time. You can add more columns with different types at any time to your container, but when analyzing the data at any given time you must know the column types.

This is an example of how to create a DataFrame, load data, and run an operation on it:

```
// Defines a DataFrame with unsigned long index type that used std::vector
using MyDataFrame = DataFrame<unsigned long, std::vector>;

MyDataFrame df;
std::vector<int> intvec = { 1, 2, 3, 4, 5 };
std::vector<double> dblvec = { 1.2345, 2.2345, 3.2345, 4.2345, 5.2345 };
std::vector<double> dblvec2 = { 0.998, 0.3456, 0.056, 0.15678, 0.00345,
                                0.923, 0.06743, 0.1 };
std::vector<std::string> strvec = { "Some string", "some string 2", "some string 3",
                                     "some string 4", "some string 5" };
std::vector<unsigned long> ulgvec = { 1UL, 2UL, 3UL, 4UL, 5UL, 8UL, 7UL, 6UL };
std::vector<unsigned long> xulgvec = ulgvec;

// This is only one way of loading data into the DataFrame. There are
// many different ways of doing it. Please see DataFrame.h and dataframe_tester.cc
int rc = df.load_data(std::move(ulgvec),
                     std::make_pair("int_col", intvec),
                     std::make_pair("dbl_col", dblvec),
                     std::make_pair("dbl_col_2", dblvec2),
                     std::make_pair("str_col", strvec),
                     std::make_pair("ul_col", xulgvec));

// Sort the Frame by index
df.sort<MyDataFrame::IndexType, int, double, std::string>();
// Sort the Frame by column "dbl_col_2"
df.sort<double, int, double, std::string>("dbl_col_2");

// A functor to calculate mean, variance, skew, kurtosis, defined in
// DataFrameVisitors.h file
StatsVisitor<double> stats_visitor;

// Calculate the stats on column "dbl_col"
df.visit<double>("dbl_col", stats_visitor);
```

### View Example:

```
std::vector<unsigned long> idx =
{ 123450, 123451, 123452, 123450, 123455, 123450, 123449 };
std::vector<double> d1 = { 1, 2, 3, 4, 5, 6, 7 };
std::vector<double> d2 = { 8, 9, 10, 11, 12, 13, 14 };
std::vector<double> d3 = { 15, 16, 17, 18, 19, 20, 21 };
std::vector<double> d4 = { 22, 23, 24, 25 };
std::vector<std::string> s1 = { "11", "22", "33", "xx", "yy", "gg", "string" };
MyDataFrame df;

df.load_data(std::move(idx),
             std::make_pair("col_1", d1),
             std::make_pair("col_2", d2),
             std::make_pair("col_3", d3),
             std::make_pair("col_4", d4),
             std::make_pair("col_str", s1));

using MyDataFrameView = DataFrameView<unsigned long>;

MyDataFrameView dfv = df.get_view_by_loc<double, std::string>({ 3, 6 });
```

```
dfv.get_column<double>("col_3")[0] = 88.0;
std::cout << "After changing a value on view: "
    << dfv.get_column<double>("col_3")[0]
    << " == " << df.get_column<double>("col_3")[3]
    << std::endl;
```

For more code examples see file *dataframe\_testr.cc*

## TYPES

```
using size_type = typename std::vector<DataVec>::size_type;
```

size\_type is the size type

---

```
using IndexType = I;
```

IndexType is the type of the index column

---

```
using IndexVecType = std::vector<I>;
```

IndexVecType is the type of the vector containing the index column

---

```
enum class nan_policy : bool {  
    pad_with_nans = true,  
    dont_pad_with_nans = false  
};
```

Enumerated type of Boolean type to specify whether data should be padded with NaN or not

---

```
enum class sort_state : bool {  
    sorted = true,  
    not_sorted = false  
};
```

Enumerated type of Boolean type to specify whether data is currently sorted or not

---

```
template<typename T>  
struct Index2D {  
    T begin {};  
    T end {};  
};
```

It represents a range with begin and end within a continuous memory space

---

```
enum class shift_policy : unsigned char {  
    down = 1, // Shift/rotate the content of all columns down,  
              // keep index unchanged  
    up = 2,   // Shift/rotate the content of all columns up,  
              // keep index unchanged  
};
```

This policy is relative to a tabular data structure  
There is no right or left shift (like Pandas), because columns in DataFrame have no ordering. They can only be accessed by name

---

```
enum class fill_policy : unsigned char {  
    value = 1,  
    fill_forward = 2,
```

```

    fill_backward = 3,
    linear_interpolate = 4, // Using the index as X coordinate
    linear_extrapolate = 5 // Using the index as X coordinate
};

```

This policy determines how to fill missing values in the DataFrame

*value*: Fill all the missing values, in a given column, with the given value.

*fill\_forward*: Fill the missing values, in a given column, with the last valid value before the missing value

*fill\_backward*: Fill the missing values, in a given column, with the first valid value after the missing value

*linear\_interpolate*:

*linear\_extrapolate*:

Use the index column as X coordinate and the given column as Y coordinate

And do interpolation/extrapolation as follows:

$$Y = Y1 + \frac{X - X1}{X2 - X1} * (Y2 - Y1)$$


---

```

enum class drop_policy : unsigned char {
    all = 1,          // Remove row if all columns are nan
    any = 2,          // Remove row if any column is nan
    threshold = 3     // Remove row if threshold number of columns are nan
};

```

This policy specifies what rows to drop/remove based on missing column data

*all*: Drop the row if all columns are missing

*any*: Drop the row if any column is missing

*threshold*: Drop the column if threshold number of columns are missing

---

```

enum class io_format : unsigned char {
    csv = 1,
};

```

This specifies the I/O format for reading and writing to/from files, streams, etc.

Currently only CSV format is supported. The CSV format is as follows:

-- Any empty line or any line started with # will be ignored

-- A data line has the following format:

<column name>:<number of data points>:<\<type\>>:data,data,...

An example line would look like this:

price:1001:<double>:23.456,24.56,...

---

```

enum class time_frequency : unsigned char {
    annual = 1,
    monthly = 2,
    weekly = 3,
    daily = 4,
    hourly = 5,
    minutely = 6,
};

```



```

secondly = 7,
millisecondly = 8,
// microsecondly = 9,
// nanosecondly = 10
};

```

This enum specifies time frequency for index generation and otherwise. The names are self-explanatory.

---

```

enum class return_policy : unsigned char {
    log = 1,
    percentage = 2,
    monetary = 3,
};

```

This policy specifies the type of return to be calculated

*log*:  $\log(\text{present} / \text{past})$

*percentage*:  $(\text{present} - \text{past}) / \text{past}$

*monetary*:  $\text{present} - \text{past}$

---

```

template<typename T, typename U>
struct type_declare;

```

```

template<typename U>
struct type_declare<HeteroVector, U> { using type = std::vector<U>; };

```

```

template<typename U>
struct type_declare<HeteroView, U> { using type = VectorView<U>; };

```

This is a spoofy way to declare a type at compile time dynamically. Here it is used in declaring a few different data structures depending whether we are a *DataFrame* or *DataFrameView*

---

```

template<typename I, typename H>
class DataFrame;

```

```

template<typename I>
using StdDataFrame = DataFrame<I, HeteroVector>;

```

```

template<typename I>
using DataFrameView = DataFrame<I, HeteroView>;

```

*DataFrame* is a class that has; An index column of type *I* (timestamp, although it doesn't have to be time), and many other columns of different types. The storage used throughout is `std::vector`.

*DataFrames* could be instantiated in two different modes:

*StdDataFrame* is the standard fully functional data-frame.

*DataFrameView* is a referenced to a slice of another data-frame. Most of the functionalities of *StdDataFrame* is also available on the *DataFrameView*. But some functionalities such as adding/removing columns etc. are not allowable on

views. If you change any of the data in a *DataFrameView* the corresponding data in the original *StdDataFrame* will also be changed.

---

## METHODS

In the following methods, “I” stands for the Index type and “H” stands for a Heterogenous vector type:

```
template<typename T>  
std::vector<T> &create_column(const char *name);
```

It creates an empty column named “name”

*T*: Type of the column

Returns a reference to the vector for that column

---

```
void remove_column(const char *name);
```

It removes a column named name.

The actual data vector is not deleted, but the column is dropped from DataFrame

---

```
void rename_column (const char *from, const char *to);
```

It renames column named from to to. If column from does not exists, it throws an exception

---

```
template<typename ... Ts>  
size_type &load_data(IndexVecType &&indices, Ts ... args);
```

This is the most generalized load function. It creates and loads an index and a variable number of columns. The index vector and all column vectors are "moved" to DataFrame.

*Ts*: The list of types for columns in args

*indices*: A vector of indices (timestamps) of type IndexType;

*args*: A variable list of arguments consisting of

std::pair(<const char \*name, std::vector<T> &&data>).

Each pair represents a column data and its name

Returns number of items loaded

---

```
template<typename ITR>  
size_type load_index(const ITR &begin, const ITR &end);
```

It copies the data from iterators begin to end into the index column

*ITR*: Type of the iterator

Returns number of items loaded

```
size_type load_index(IndexVecType &&idx);
```

It moves the idx vector into the index column.

Returns number of items loaded

---

```
static std::vector<I>
```

```

gen_datetime_index(const char *start_datetime,
                   const char *end_datetime,
                   time_frequency t_freq,
                   long increment = 1,
                   DT_TIME_ZONE tz = DT_TIME_ZONE::LOCAL);

```

This static method generates a date/time-based index vector that could be fed directly to one of the load methods. Depending on the specified frequency, it generates specific timestamps (see below).

It returns a vector of I timestamps.

Currently I could be any built-in numeric type or DateTime

*start\_datetime, end\_datetime*: They are the start/end date/times of requested timestamps.

They must be in the following format:

MM/DD/YYYY [HH[:MM[:SS[.MMM]]]]

*t\_freq*: Specifies the timestamp frequency. Depending on the frequency, and I type specific timestamps are generated as follows:

- I type of DateTime always generates timestamps of DateTime.
- Annual, monthly, weekly, and daily frequencies generates YYYYMMDD timestamps.
- Hourly, minutely, and secondly frequencies generates epoch timestamps (64 bit).
- Millisecondly frequency generates nano-second since epoch timestamps (128 bit).

*increment*: Increment in the units of the frequency

*tz*: Time-zone of generated timestamps

NOTE: It is the responsibility of the programmer to make sure I type is big enough to contain the frequency.

---

```

template<typename T, typename ITR>
size_type load_column(const char *name,
                     Index2D<const ITR &> range,
                     nan_policy padding = nan_policy::pad_with_nans);

```

It copies the data from iterators begin to end to the named column. If column does not exist, it will be created. If the column exist, it will be over written.

*T*: Type of data being copied

*ITR*: Type of the iterator

*name*: Name of the column

*range*: The begin and end iterators for data

*padding*: If true, it pads the data column with nan if it is shorter than the index column.

Returns number of items loaded

```

template<typename T>

```

```
size_type
load_column(const char *name,
            std::vector<T> &&data,
            nan_policy padding = nan_policy::pad_with_nans);
```

```
template<typename T>
size_type
load_column(const char *name,
            const std::vector<T> &data,
            nan_policy padding = nan_policy::pad_with_nans);
```

It moves or copies (depending on the version) the data to the named column in DataFrame. If column does not exist, it will be created. If the column exist, it will be over written.

*T*: Type of data being moved

*name*: Name of the column

*padding*: If true, it pads the data column with nan,  
if it is shorter than the index column.

Returns number of items loaded

---

```
size_type append_index(const IndexType &val);
```

It appends val to the end of the index column.

Returns number of items loaded

```
template<typename ITR>
size_type append_index(Index2D<const ITR &> range);
```

It appends the range begin to end to the end of the index column

*ITR*: Type of the iterator

*range*: The begin and end iterators for data

Returns number of items loaded

---

```
template<typename T>
size_type append_column(const char *name,
                        const T &val,
                        nan_policy padding = nan_policy::pad_with_nans);
```

It appends val to the end of the named data column. If data column doesn't exist, it throws an exception.

*T*: Type of the named data column

*name*: Name of the column

*padding*: If true, it pads the data column with nan,  
if it is shorter than the index column.

Returns number of items loaded

```
template<typename T, typename ITR>
size_type append_column(const char *name,
                        Index2D<const ITR &> range,
                        nan_policy padding = nan_policy::pad_with_nans);
```

It appends the range begin to end to the end of the named data column. If data column doesn't exist, it throws an exception.

*T*: Type of the named data column

*ITR*: Type of the iterator

*name*: Name of the column

*range*: The begin and end iterators for data

*padding*: If true, it pads the data column with nan,  
if it is shorter than the index column.

Returns number of items loaded

---

```
template<typename ... types>
void remove_data_by_idx (Index2D<I> range);
```

It removes the data rows from index begin to index end.

DataFrame must be sorted by index or behavior is undefined.

This function first calls make\_consistent() that may add nan values to data columns.

*types*: List all the types of all data columns.

A type should be specified in the list only once.

*range*: The begin and end iterators for index specified with index values

---

```
template<typename ... types>
void remove_data_by_loc (Index2D<int> range);
```

It removes the data rows from location begin to location end within range.

This function supports Python-like negative indexing. That is why the range type is int.

This function first calls make\_consistent() that may add nan values to data columns.

*types*: List all the types of all data columns.

A type should be specified in the list only once.

*range*: The begin and end iterators for data

---

```
template<typename T, size_t N>
void fill_missing(const std::array<const char *, N> col_names,
                 fill_policy policy,
                 const std::array<T, N> values = { },
                 int limit = -1);
```

It fills all the "missing values" with the given values, and/or using the given method (See fill\_policy above). Missing is determined by being NaN for types

that have NaN. For types without NaN (e.g. string), default value is considered missing value.

*T*: Type of the column(s) in *col\_names* array

*N*: Size of *col\_names* and values array

*col\_names*: An array of names specifying the columns to fill.

*policy*: Specifies the method to use to fill the missing values.

For example; forward fill, values, etc.

*values*: If the policy is "values", use these values to fill the missing holes. Each value corresponds to the same index in the *col\_names* array.

*limit*: Specifies how many values to fill. Default is -1 meaning fill all missing values.

---

`template<typename ... types>`

`void drop_missing(drop_policy policy, size_type threshold = 0);`

It removes a row if any or all or some of the columns are NaN, based on drop policy

*types*: List all the types of all data columns.

A type should be specified in the list only once.

*threshold*: If drop policy is threshold, it specifies the numbers of NaN columns before removing the row.

---

`template<typename T, size_t N>`

`size_type replace(const char *col_name,  
                  const std::array<T, N> old_values,  
                  const std::array<T, N> new_values,  
                  int limit = -1);`

It iterates over the column named *col\_name* (or index, if *col\_name* == "INDEX") and replaces all values in *old\_values* with the corresponding values in *new\_values* up to the limit. If limit is omitted, all values will be replaced.

It returns number of items replaced.

*T*: Type on column *col\_name*. If this is index it would be the same as *I*.

*N*: Size of *old\_values* and *new\_values* arrays

*col\_name*: Name of the column

*old\_array*: An array of values to be replaced in *col\_name* column

*new\_array*: An array of values to replace the *old\_values* in *col\_name* column

*limit*: Limit of how many items to replace. Default is to replace all.

`template<typename T, size_t N>`

`std::future<size_type>`

`replace_async(const char *col_name,  
              const std::array<T, N> old_values,  
              const std::array<T, N> new_values,`

```
int limit = -1);
```

Same as `replace()` above, but executed asynchronously

**NOTE:** multiple instances of `replace_async()` maybe executed for different columns at the same time with no problem.

```
template<typename T, typename F>
void replace(const char *col_name, F &functor);
```

This is similar to `replace()` above but it lets a functor replace the values in the named column. The functor is passed every value of the column along with a const reference of the corresponding index value.

Unlike the `replace` version above, this `replace` can only work on data columns. It will not work on index column.

The functor must have the following interface at minimum:

```
bool operator() (const IndexType &ts, T &value);
```

A false return from the above operator method stops the iteration through named column values.

*T*: Type on column `col_name`. If this is index it would be the same as `I`.

*F*: The functor type

*col\_name*: Name of the column

*functor*: An instance of the functor

```
template<typename T, typename F>
std::future<void> replace_async(const char *col_name, F &functor);
```

Same as `replace()` above, but executed asynchronously

**NOTE:** multiple instances of `replace_async()` maybe executed for different columns at the same time with no problem.

---

```
template<typename ... types>
void make_consistent ();
```

Make all data columns the same length as the index. If any data column is shorter than the index column, it will be padded by nan.

This is also called by `sort()`, before sorting

---

```
template<typename T, typename ... types>
void sort(const char *by_name = nullptr);
```

Sort the DataFrame by the named column. By default, it sorts by index (i.e. `by_name == nullptr`). Sort first calls `make_consistent()` that may add nan values to data columns. nan values make sorting nondeterministic.

*T*: Type of the `by_name` column. You always of the specify this type, even if it is being sorted to the default index

*types*: List all the types of all data columns.

A type should be specified in the list only once.

```
template<typename T, typename ... types>
```



`std::future<void> sort_async (const char *by_name = nullptr);`  
Same as `sort()` above, but executed asynchronously

---

`template<typename F, typename T, typename ... types>`

`DataFrame groupby (F &&func,`

`const char *gb_col_name = nullptr,`

`sort_state already_sorted = sort_state::not_sorted) const;`

Groupby copies the DataFrame into a temp DataFrame and sorts the temp df by `gb_col_name` before performing groupby. If `gb_col_name` is null, it groups by index.

F: type functor to be applied to columns to group by

T: type of the groupby column. In case if index, it is type of index

types: List of the types of all data columns.

A type should be specified in the list only once.

func: The functor to do the groupby. Specs for the functor is in a separate doc.

already\_sorted: If the DataFrame is already sorted by `gb_col_name`, this will save the expensive sort operation

`std::future<DataFrame>`

`groupby_async (F &&func,`

`const char *gb_col_name = nullptr,`

`sort_state already_sorted = sort_state::not_sorted) const;`

Same as `groupby()` above, but executed asynchronously

---

`template<typename T>`

`StdDataFrame<T> value_counts (const char *col_name) const;`

It counts the unique values in the named column.

It returns a StdDataFrame of following specs:

- 1) The index is of type T and contains all unique values in the named column.
- 2) There is only one column named "counts" of type `size_type` that contains the count for each index row.

For this method to compile and work, 3 conditions must be met:

- 1) Type T must be hashable. If this is a user defined type, you must enable and specialize `std::hash`.
- 2) The equality operator (`==`) must be well defined for type T.
- 3) Type T must match the actual type of the named column.

Of course, if you never call this method in your application, you need not be worried about these conditions.

T: Type of the `col_name` column.

---

`template<typename F, typename ... types>`

`DataFrame bucketize (F &&func, const IndexType &bucket_interval) const;`

It bucketizes the data and index into `bucket_interval`'s, based on index values and calls the functor for each bucket. The result of each bucket will be stored in a new `DataFrame` with same shape and returned. Every data bucket is guaranteed to be as wide as `bucket_interval`. This mean some data items at the end may not be included in the new bucketized `DataFrame`. The index of each bucket will be the last index in the original `DataFrame` that is less than `bucket_interval` away from the previous bucket

NOTE: The `DataFrame` must already be sorted by index.

F: type functor to be applied to columns to bucketize

types: List of the types of all data columns.

A type should be specified in the list only once.

`bucket_interval`: Bucket interval is in the index's single value unit.

For example, if index is in minutes, `bucket_interval` will be in the unit of minutes and so on.

`already_sorted`: If the `DataFrame` is already sorted by index, this will save the expensive sort operation

```
template<typename F, typename ... types>
```

```
std::future<DataFrame>
```

```
bucketize_async(F &&func, const IndexType &bucket_interval) const;
```

Same as `bucketize()` above, but executed asynchronously

```
template<typename F, typename ... types>
```

```
void self_bucketize(F &&func, const IndexType &bucket_interval);
```

This is exactly the same as `bucketize()` above. The only difference is it stores the result in itself and returns void. So, after the return the original data is lost and replaced with bucketized data

---

```
template<typename T, typename V>
```

```
DataFrame transpose(IndexVecType &&indices,  
                    const V &current_col_order,  
                    const V &new_col_names) const;
```

It transposes the data in the `DataFrame`.

The `transpose()` is only defined for `DataFrame`'s that have a single data type.

NOTE: Since `DataFrame` columns have no ordering, the user must specify the order with `current_col_order`.

T: The single type for all data columns

V: The type of string vector specifying the new names for new columns after transpose

`indices`: A vector on indices for the new transposed `DataFrame`.

Its length must equal the number of rows in this `DataFrame`.

Otherwise an exception is thrown

`current_col_order`: A vector of strings specifying the order of columns

in the original DataFrame.  
new\_col\_names: A vector of strings, specifying the column names for the new transposed DataFrame.  
Its length must equal the number of rows in this DataFrame. Otherwise an exception is thrown

---

`template<typename RHS_T, typename ... types>  
StdDataFrame<I> join_by_index (const RHS_T &rhs, join_policy mp) const;`  
It joins the data between self (lhs) and rhs and returns the joined data in a StdDataFrame, based on specification in join\_policy.  
The following conditions must be met for this method to compile and work properly:  
1) I type must be the same between lhs and rhs.  
2) Ordering (< > != ==) must be well defined for type I  
3) Both lhs and rhs must be sorted by index  
4) In both lhs and rhs, columns with the same name must have the same  
5) Type

RHS\_T: Type of DataFrame rhs  
types: List all the types of all data columns.  
A type should be specified in the list only once.  
rhs: The rhs DataFrame  
join\_policy: Specifies how to join. For example inner join, or left join, etc. (See join\_policy definition)

---

`template<typename ... types>  
void self_shift (size_type periods, shift_policy sp);`  
It shifts all the columns in self up or down based on shift\_policy.  
Values that are shifted will be assigned to NaN. The index column remains unchanged.  
If user shifts with periods that is larger than the column length, all values in that column become NaN.

types: List all the types of all data columns.  
A type should be specified in the list only once.  
periods: Number of periods to shift  
shift\_policy: Specifies the direction (i.e. up/down) to shift

`template<typename ... types>  
StdDataFrame<I> shift (size_type periods, shift_policy sp) const;`  
It is exactly the same as self\_shift, but it leaves self unchanged and returns a new DataFrame with columns shifted.

---

`template<typename ... types>  
void self_rotate (size_type periods, shift_policy sp);`  
It rotates all the columns in self up or down based on shift\_policy.

The index column remains unchanged.  
If user rotates with periods that is larger than the column length, the behavior is undefined.

types: List all the types of all data columns.

A type should be specified in the list only once.

periods: Number of periods to rotate

shift\_policy: Specifies the direction (i.e. up/down) to rotate

`template<typename ... types>`

`StdDataFrame<I> rotate (size_type periods, shift_policy sp) const;`

It is exactly the same as self\_rotate, but it leaves self unchanged and returns a new DataFrame with columns rotated.

---

`template<typename S, typename ... types>`

`bool write (S &o, bool values_only = false, io_format iof = io_format::csv) const;`

It outputs the content of DataFrame into the stream o as text in the following format:

INDEX:<Comma delimited list of values>

<Column1 name>:<Column1 type>:<Comma delimited list of values>

<Column2 name>:<Column2 type>:<Comma delimited list of values>

S: Output stream type

types: List all the types of all data columns.

A type should be specified in the list only once.

o: Reference to a streamable object (e.g. cout)

values\_only: If true, the name and type of each column is not written

iof: Specifies the I/O format. The default is CSV

`template<typename S, typename ... Ts>`

`std::future<bool> write_async (S &o,`

`bool values_only = false,`

`io_format iof = io_format::csv) const;`

Same as write() above, but executed asynchronously

---

`bool read (const char *file_name, io_format iof = io_format::csv);`

It inputs the contents of a text file into itself (i.e. DataFrame). The format of the file must be:

INDEX:<Comma delimited list of values>

<Column1 name>:<Column1 type>:<Comma delimited list of values>

<Column2 name>:<Column2 type>:<Comma delimited list of values>

All empty lines or lines starting with # will be skipped.

file\_name: Complete path to the file

iof: Specifies the I/O format. The default is CSV

```
std::future<bool> read_async (const char *file_name,  
                             io_format iof = io_format::csv);
```

Same as read() above, but executed asynchronously

---

```
template<typename T>  
typename type_declare<H, T>::type &  
get_column (const char *name);
```

It returns a reference to the container of named data column

The return type depends on if we are in standard or view mode

T: Data type of the named column

```
template<typename T>  
const typename type_declare<H, T>::type &  
get_column (const char *name) const;
```

It returns a const reference to the container of named data column

The return type depends on if we are in standard or view mode

T: Data type of the named column

---

```
template<size_t N, typename ... types>
```

HeteroVector

```
get_row(size_type row_num, const std::array<const char *, N> col_names) const;
```

It returns the data in row row\_num for columns in col\_names. The order of data items in the returned vector is the same as order of columns on col\_names.

The first item in the returned vector is always the index value corresponding to the row\_num

It returns a HeteroVector which contains a different type for each column.

N: Size of col\_names and values array

types: List all the types of all data columns. A type should be specified in the list only once.

row\_num: The row number

col\_names: Names of columns to get data from. It also specifies the order of data in the returned vector

---

```
template<typename T>  
std::vector<T> get_col_unique_values (const char *name) const;
```

It returns a vector of unique values in the named column in the same order that exists in the column.

For this method to compile and work, 3 conditions must be met:

- 1) Type T must be hash-able. If this is a user defined type, you must enable and specialize std::hash.
- 2) The equality operator (==) must be well defined for type T.
- 3) Type T must match the actual type of the named column.

Of course, if you never call this method in your application,

you need not be worried about these conditions.

T: Data type of the named column

---

`template<typename ... types>`

`DataFrame get_data_by_idx (Index2D<I> range) const;`

It returns a DataFrame (including the index and data columns) containing the data from index begin to index end. This function assumes the DataFrame is consistent and sorted by index. The behavior is undefined otherwise.

types: List all the types of all data columns.

A type should be specified in the list only once.

range: The begin and end iterators for index specified with index values

---

`template<typename ... types>`

`DataFrameView<I> get_view_by_idx (Index2D<I> range) const;`

It behaves like `get_data_by_idx()`, but it returns a DataFrameView.

A view is a DataFrame that is a reference to the original DataFrame.

So if you modify anything in the view the original DataFrame will also be modified.

Note: There are certain operations that you cannot do with a view.

For example, you cannot add/delete columns, etc.

types: List all the types of all data columns.

A type should be specified in the list only once.

range: The begin and end iterators for index specified with index values

---

`template<typename ... types>`

`DataFrame get_data_by_loc (Index2D<int> range) const;`

It returns a DataFrame (including the index and data columns) containing the data from location begin to location end.

This function supports Python-like negative indexing. That is why the range type is int.

This function assumes the DataFrame is consistent and sorted by index. The behavior is undefined otherwise.

types: List all the types of all data columns.

A type should be specified in the list only once.

range: The begin and end iterators for data

---

`template<typename ... types>`

`DataFrameView<I> get_view_by_loc (Index2D<int> range) const;`

It behaves like `get_data_by_loc()`, but it returns a DataFrameView.

A view is a DataFrame that is a reference to the original DataFrame.

So if you modify anything in the view the original DataFrame will also be modified.

Note: There are certain operations that you cannot do with a view.  
For example, you cannot add/delete columns, etc.

types: List all the types of all data columns.

A type should be specified in the list only once.

range: The begin and end iterators for data

---

`const IndexVecType &get_index () const { return (indices_); }`

It returns a const reference to the index container

`IVec &get_index () { return (indices_); }`

It returns a reference to the index container

---

`template<typename ... Ts>`

`void multi_visit (Ts ... args);`

This is the most generalized visit function. It visits multiple columns with the corresponding function objects sequentially. Each function object is passed every single value of the given column along with its name and the corresponding index value. All functions objects must have this signature

`bool (const IndexType &i, const char *name, T &col_value)`

If the function object returns false, the DataFrame will stop iterating at that point on that column..

NOTE: This method could be used to implement a pivot table.

Ts: The list of types for columns in args

args: A variable list of arguments consisting of

`std::pair(<const char *name,`

`&std::function<bool (const IndexType &, const char *, T &)>).`

Each pair represents a column name and the functor to run on it.

NOTE: The second member of pair is a `_pointer_` to the function or functor object

---

`template<typename T, typename V>`

`V &visit (const char *name, V &visitor);`

It passes the values of each index and each named column to the functor visitor sequentially from beginning to end

NOTE: This method could be used to implement a pivot table.

T: Type of the named column

V: Type of the visitor functor

name: Name of the data column

`template<typename T1, typename T2, typename V>`

`V &visit (const char *name1, const char *name2, V &visitor);`

It passes the values of each index and the two named columns to the functor visitor sequentially from beginning to end

NOTE: This method could be used to implement a pivot table.

T1: Type of the first named column

T2: Type of the second named column

V: Type of the visitor functor

name1: Name of the first data column

name2: Name of the second data column

```
template<typename T1, typename T2, typename T3, typename V>
```

```
V &visit (const char *name1, const char *name2, const char *name3, V &visitor);
```

It passes the values of each index and the three named columns to the functor

visitor sequentially from beginning to end

NOTE: This method could be used to implement a pivot table.

T1: Type of the first named column

T2: Type of the second named column

T3: Type of the third named column

V: Type of the visitor functor

name1: Name of the first data column

name2: Name of the second data column

name3: Name of the third data column

```
template<typename T1, typename T2, typename T3, typename T4, typename V>
```

```
V &visit (const char *name1,  
          const char *name2,  
          const char *name3,  
          const char *name4,  
          V &visitor);
```

It passes the values of each index and the four named columns to the functor

visitor sequentially from beginning to end

NOTE: This method could be used to implement a pivot table.

T1: Type of the first named column

T2: Type of the second named column

T3: Type of the third named column

T4: Type of the forth named column

V: Type of the visitor functor

name1: Name of the first data column

name2: Name of the second data column

name3: Name of the third data column

name4: Name of the fourth data column

```
template<typename T1, typename T2, typename T3, typename T4, typename T5,  
        typename V>
```

```
V &visit (const char *name1,  
          const char *name2,
```



```
const char *name3,  
const char *name4,  
const char *name5,  
V &visitor);
```

It passes the values of each index and the five named columns to the functor visitor sequentially from beginning to end

NOTE: This method could be used to implement a pivot table.

T1: Type of the first named column

T2: Type of the second named column

T3: Type of the third named column

T4: Type of the forth named column

T5: Type of the fifth named column

V: Type of the visitor functor

name1: Name of the first data column

name2: Name of the second data column

name3: Name of the third data column

name4: Name of the forth data column

name5: Name of the fifth data column

---

```
template<typename T, typename V>
```

```
V & single_act_visit (const char *name, V &visitor);
```

This is similar to visit(), but it passes a const reference to the index vector and the named column vector at once the functor visitor. This is convenient for calculations that need the whole data vector, for example auto-correlation.

T: Type of the named column

V: Type of the visitor functor

name: Name of the data column

```
template<typename T1, typename T2, typename V>
```

```
V & single_act_visit (const char *name1, const char *name2, V &visitor);
```

This is similar to visit(), but it passes a const reference to the index vector and the two named column vectors at once the functor visitor. This is convenient for calculations that need the whole data vector.

NOTE: This method could be used to implement a pivot table.

T1: Type of the first named column

T2: Type of the second named column

V: Type of the visitor functor

name1: Name of the first data column

name2: Name of the second data column

---

```
template<typename ... types>
```

```
bool is_equal (const DataFrame &rhs) const;
```

It compares self with rhs. If both have the same indices, same number of columns, same names for each column, and all columns are equal, then it returns true. Otherwise it returns false

types: List all the types of all data columns.

A type should be specified in the list only once.

---

template<typename ... types>

DataFrame &modify\_by\_idx (DataFrame &rhs,  
sort\_state already\_sorted = sort\_state::not\_sorted);

It iterates over all indices in rhs and modifies all the data columns in self that correspond to the given index value. If not already\_sorted, both rhs and self will be sorted by index. It returns a reference to self

types: List all the types of all data columns.

A type should be specified in the list only once.

already\_sorted: If the self and rhs are already sorted by index,  
this will save the expensive sort operations

---

## GLOBAL OPERATORS

These are currently arithmetic operators declared in *include/DataFrame.h*. Because they all have to be templated, they cannot be defined as redefined built-in operators.

```
template<typename DF, typename ... types>  
inline DF df_plus (const DF &lhs, const DF &rhs);
```

```
template<typename DF, typename ... types>  
inline DF df_minus (const DF &lhs, const DF &rhs);
```

```
template<typename DF, typename ... types>  
inline DF df_multiplies (const DF &lhs, const DF &rhs);
```

```
template<typename DF, typename ... types>  
inline DF df_divides (const DF &lhs, const DF &rhs);
```

These arithmetic operations operate on the same-name and same-type columns on lhs and rhs. Each pair of entries is operated on, only if they have the same index value.

They return a new DataFrame

NOTE: Both lhs and rhs must be already sorted by index, otherwise the result is nonsensical.

---

## BUILT-IN VISITORS

These are all defined in file *include/DataFrameVisitors.h*.

```
template<typename T,  
        typename I = unsigned long,  
        typename = typename std::enable_if<std::is_arithmetic<T>::value, T>::type>  
struct MeanVisitor;
```

This functor class calculates the mean of a given column. See this document and `datasci_tester.cc` for examples.

*T*: Column/data type

*I*: Index type

*T* must be an arithmetic-enabled type

---

```
template<typename T,  
        typename I = unsigned long,  
        typename = typename std::enable_if<std::is_arithmetic<T>::value, T>::type>  
struct SumVisitor;
```

This functor class calculates the sum of a given column. See this document and `datasci_tester.cc` for examples.

*T*: Column/data type

*I*: Index type

*T* must be an arithmetic-enabled type

---

```
template<typename T,  
        typename I = unsigned long,  
        typename = typename std::enable_if<std::is_arithmetic<T>::value, T>::type>  
struct CumSumVisitor;
```

This is a “single action visitor”, meaning it is passed the whole data vector in one call and you must use the `single_action_visit()` interface.

This functor class calculates the cumulative sum of a given column. See this document and `datasci_tester.cc` for examples.

The result is a vector of running sums

*T*: Column/data type

*I*: Index type

*T* must be an arithmetic-enabled type

---

```
template<typename T,  
        typename I = unsigned long,  
        typename = typename std::enable_if<std::is_arithmetic<T>::value, T>::type>  
struct ProdVisitor;
```

This functor class calculates the product of a given column. See this document and `datasci_tester.cc` for examples.

*T*: Column/data type  
*I*: Index type  
*T must be an arithmetic-enabled type*

---

```
template<typename T,  
        typename I = unsigned long,  
        typename = typename std::enable_if<std::is_arithmetic<T>::value, T>::type>  
struct CumProdVisitor;
```

This is a “single action visitor”, meaning it is passed the whole data vector in one call and you must use the `single_action_visit()` interface.

This functor class calculates the cumulative product of a given column. See this document and `datasci_tester.cc` for examples.

The result is a vector of running products.

*T*: Column/data type  
*I*: Index type  
*T must be an arithmetic-enabled type*

---

```
template<typename T, typename I = unsigned long>  
struct MaxVisitor;
```

This functor class calculates the maximum of a given column. See this document and `datasci_tester.cc` for examples.

*T*: Column/data type  
*I*: Index type

---

```
template<typename T, typename I = unsigned long>  
struct CumMaxVisitor;
```

This is a “single action visitor”, meaning it is passed the whole data vector in one call and you must use the `single_action_visit()` interface.

This functor class calculates the cumulative maximum of a given column. See this document and `datasci_tester.cc` for examples.

The result is a vector of running maximums

*T*: Column/data type  
*I*: Index type

---

```
template<typename T, typename I = unsigned long>  
struct MinVisitor;
```

This functor class calculates the minimum of a given column. See this document and `datasci_tester.cc` for examples.

*T*: Column/data type  
*I*: Index type

---

```
template<typename T, typename I = unsigned long>
```

`struct CumMinVisitor;`

This is a “single action visitor”, meaning it is passed the whole data vector in one call and you must use the `single_action_visit()` interface.

This functor class calculates the cumulative minimum of a given column. See this document and `datasci_tester.cc` for examples.

The result is a vector of running minimum

*T*: Column/data type

*I*: Index type

---

`template<std::size_t N, typename T, typename I = unsigned long>`

`struct NlargestVisitor;`

This functor class calculates the *N* largest values of a column. It runs in  $O(N*M)$ , where *N* is the number of largest values and *M* is the total number of all values.

If *N* is relatively small this is better than  $O(M*\log M)$ .

See this document and `datasci_tester.cc` for examples.

*N*: Number of largest values

*T*: Column/data type

*I*: Index type

---

`template<std::size_t N, typename T, typename I = unsigned long>`

`struct NSmallestVisitor;`

This functor class calculates the *N* smallest values of a column. It runs in  $O(N*M)$ , where *N* is the number of largest values and *M* is the total number of all values.

If *N* is relatively small this is better than  $O(M*\log M)$ .

See this document and `datasci_tester.cc` for examples.

*N*: Number of largest values

*T*: Column/data type

*I*: Index type

---

`template<typename T,`

`typename I = unsigned long,`

`typename = typename std::enable_if<std::is_arithmetic<T>::value, T>::type>`

`struct CovVisitor;`

This functor class calculates the covariance of two given columns. In addition, it provides the variances of both columns

See this document and `datasci_tester.cc` for examples.

*T*: Column/data type

*I*: Index type

*T must be an arithmetic-enabled type*

---

`template<typename T,`

`typename I = unsigned long,`

```
typename = typename std::enable_if<std::is_arithmetic<T>::value, T>::type>
struct VarVisitor;
```

This functor class calculates the variance of a given column. See this document and `datasci_tester.cc` for examples.

*T*: Column/data type

*I*: Index type

*T must be an arithmetic-enabled type*

---

```
template<typename T,
        typename I = unsigned long,
        typename = typename std::enable_if<std::is_arithmetic<T>::value, T>::type>
struct BetaVisitor;
```

This functor class calculates the beta (i.e. exposure) of the given first column to the given second column (benchmark). See this document and `datasci_tester.cc` for examples.

*T*: Column/data type

*I*: Index type

*T must be an arithmetic-enabled type*

---

```
template<typename T,
        typename I = unsigned long,
        typename = typename std::enable_if<std::is_arithmetic<T>::value, T>::type>
struct StdVisitor;
```

This functor class calculates the standard deviation of a given column. See this document and `datasci_tester.cc` for examples.

*T*: Column/data type

*I*: Index type

*T must be an arithmetic-enabled type*

---

```
template<typename T,
        typename I = unsigned long,
        typename = typename std::enable_if<std::is_arithmetic<T>::value, T>::type>
struct TrackingErrorVisitor;
```

This functor class calculates the tracking error between two columns. Tracking error is the standard deviation of the difference vector.

See this document and `datasci_tester.cc` for examples.

*T*: Column/data type

*I*: Index type

*T must be an arithmetic-enabled type*

---

```
template<typename T,
        typename I = unsigned long,
```

```
typename = typename std::enable_if<std::is_arithmetic<T>::value, T>::type>
struct CorrVisitor;
```

This functor class calculates the correlation of two given columns. See this document and `datasci_tester.cc` for examples.

T: Column/data type

I: Index type

*T must be an arithmetic-enabled type*

---

```
template<typename T,
        typename I = unsigned long,
        typename = typename std::enable_if<std::is_arithmetic<T>::value, T>::type>
struct AutoCorrVisitor;
```

This is a “single action visitor”, meaning it is passed the whole data vector in one call and you must use the `single_action_visit()` interface.

This functor class calculates the auto correlation of given column. The result is a vector of auto correlations with lags of 0 up to length of column – 4.

See this document and `datasci_tester.cc` for examples.

T: Column/data type

I: Index type

*T must be an arithmetic-enabled type*

---

```
template<typename T,
        typename I = unsigned long,
        typename = typename std::enable_if<std::is_arithmetic<T>::value, T>::type>
struct ReturnVisitor;
```

This is a “single action visitor”, meaning it is passed the whole data vector in one call and you must use the `single_action_visit()` interface.

This functor class calculates the return of a given column, according to the return policy (monetary, percentage, or log). The result is a vector of returns.

See this document and `datasci_tester.cc` for examples.

T: Column/data type

I: Index type

*T must be an arithmetic-enabled type*

---

```
template<typename T,
        typename I = unsigned long,
        typename = typename std::enable_if<std::is_arithmetic<T>::value, T>::type>
struct KthValueVisitor;
```

This is a “single action visitor”, meaning it is passed the whole data vector in one call and you must use the `single_action_visit()` interface.

This functor class finds the Kth element in the given column in linear time.

T: Column/data type



I: Index type

*T must be an arithmetic-enabled type*

---

```
template<typename T,  
        typename I = unsigned long,  
        typename = typename std::enable_if<std::is_arithmetic<T>::value, T>::type>  
struct MedianVisitor;
```

This is a “single action visitor”, meaning it is passed the whole data vector in one call and you must use the `single_action_visit()` interface.

This functor class finds the median of the given column, using the above Kth element visitor. It computes in linear time.

T: Column/data type

I: Index type

*T must be an arithmetic-enabled type*

---

```
template<std::size_t N, typename T, typename I = unsigned long>  
struct ModeVisitor;
```

This is a “single action visitor”, meaning it is passed the whole data vector in one call and you must use the `single_action_visit()` interface.

This functor class finds the N highest mode (N most repeated values) of the given column.

The result is an array of N items each of this type:

```
struct DataItem {  
    // Value of the column item  
    value_type      value { };  
    // List of indices where value occurred  
    std::vector<index_type> indices { };  
    // Number of times value occurred  
    inline size_type repeat_count() const { return (indices.size()); }  
    // List of column indices where value occurred  
    std::vector<size_type> value_indices_in_col { };  
};
```

N: Number of modes to find

T: Column/data type

I: Index type

*T must be an arithmetic-enabled type*

---

```
template<typename T,  
        typename I = unsigned long,  
        typename = typename std::enable_if<std::is_arithmetic<T>::value, T>::type>  
struct DotProdVisitor;
```

This functor class calculates the dot-product of two given columns. See this document and `datasci_tester.cc` for examples.

T: Column/data type

I: Index type

*T must be an arithmetic-enabled type*

---

```
template<typename T,  
        typename I = unsigned long,  
        typename = typename std::enable_if<std::is_arithmetic<T>::value, T>::type>  
struct StatsVisitor;
```

This functor class calculates the following statistics of a given column; mean, variance, standard deviation, skew, and kurtosis. See this document and `datasci_tester.cc` for examples.

T: Column/data type

I: Index type

*T must be an arithmetic-enabled type*

---

```
template<typename T,  
        typename I = unsigned long,  
        typename = typename std::enable_if<std::is_arithmetic<T>::value, T>::type>  
struct SLRRegressionVisitor;
```

This functor class calculates simple linear regression, in one pass, of two given columns (x, y). See this document and `datasci_tester.cc` for examples.

T: Column/data type

I: Index type

*T must be an arithmetic-enabled type*

---