

DataFrame Methods	DataFrame Types	DataFrame built-in Functors	Random Generators
append_column( 2 )	enum class drop_policy { }	struct AffinityPropVisitor { }	gen_bernoulli_dist( )
append_index( 2 )	enum class fill_policy { }	struct AutoCorrVisitor { }	gen_binomial_dist( )
bucketize( )	enum class exponential_decay_spec { }	struct BetaVisitor { }	gen_cauchy_dist( )
bucketize_async( )	enum class int_dist_policy { }	struct BollingerBand { }	gen_chi_squared_dist( )
create_column( )	enum class io_format { }	struct CorrVisitor { }	gen_exponential_dist( )
drop_missing( )	enum class join_policy { }	struct CovVisitor { }	gen_extreme_value_dist( )
fill_missing( )	enum class nan_policy { }	struct CumMaxVisitor { }	gen_fisher_f_dist( )
gen_datetime_index( )	enum class random_policy { }	struct CumMinVisitor { }	gen_gamma_dist( )
gen_sequence_index( )	enum class return_policy { }	struct CumProdVisitor { }	gen_geometric_dist( )
get_col_unique_values( )	enum class shift_policy { }	struct CumSumVisitor { }	gen_lognormal_dist( )
get_column( 2 )	enum class sort_spec { }	struct DotProdVisitor { }	gen_negative_binomial_dist( )
get_data_by_idx( 2 )	enum class sort_state { }	struct DoubleCrossOver { }	gen_normal_dist( )
get_data_by_loc( 2 )	enum class time_frequency { }	struct ExponentialRollAdopter { }	gen_poisson_dist( )
get_data_by_rand( )	struct BadRange { }	struct GeometricMeanVisitor { }	gen_student_t_dist( )
get_data_by_sel( 3 )	struct ColNotFound { }	struct GroupbySum { }	gen_uniform_int_dist( )
get_index( 2 )	struct DataFrameError { }	struct HarmonicMeanVisitor { }	gen_uniform_real_dist( )
get_memory_usage( )	struct InconsistentData { }	struct KMeansVisitor { }	gen_weibull_dist( )
get_row( )	struct Index2D { }	struct KthValueVisitor { }	
get_view_by_idx( 2 )	struct MemUsage { }	struct MaxVisitor { }	
get_view_by_loc( 2 )	struct NotFeasible { }	struct MeanVisitor { }	
get_view_by_rand( )	struct NotImplemented { }	struct MedianVisitor { }	
get_view_by_sel( 3 )		struct MinVisitor { }	
groupby( )		struct ModeVisitor { }	
groupby_async( )		struct NLargestVisitor { }	
is_equal( )		struct NSmallestVisitor { }	
join_by_column( )		struct ProdVisitor { }	
join_by_index( )		struct ReturnVisitor { }	
load_column( 3 )		struct SampleZScoreVisitor { }	
load_data( )		struct SimpleRollAdopter { }	
load_index( 2 )		struct SLRegressionVisitor { }	
make_consistent( )		struct StatsVisitor { }	
modify_by_idx( )		struct StdVisitor { }	
multi_visit( )		struct SumVisitor { }	
read( )		struct TrackingErrorVisitor { }	
read_async( )		struct ZScoreVisitor { }	
remove_column( )			
remove_data_by_idx( )			
remove_data_by_loc( )			
remove_data_by_sel( 3 )			
rename_lock( )			
rename_column( )			
replace( 2 )			
replace_async( 2 )			
replace_index( )			
rotate( )			
self_bucketize( )			
self_rotate( )			
self_shift( )			
shape( )			
set_lock( )			
shift( )			
shrink_to_fit( )			
shuffle( )			
single_act_visit( 2 )			
sort( 5 )			
sort_async( 5 )			
transpose( )			
value_counts( )			
visit( 5 )			
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.

*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 my spare time is limited.*

**Views** were subsequently added. It is a very useful concept with practical use-cases. 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 corresponding data point(s) in 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 visitation 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 one data point at a time (e.g. correlation, variance).
- 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 (e.g. return, median).

See this document, *DataFrameStatsVisitors.h*, *DataFrameMLVisitors.h*, *DataFrameFinancialVisitors.h*, and *dataframe\_tester.cc* for more examples and documentation.

**Random Generators** were added as a series of convenient stand-alone functions to generate random numbers (it covers all C++ standard distributions). You can seamlessly use these routines to generate random DataFrame columns.

See this *document* and file *RandGen.h* and *dataframe\_tester.cc*.

## CODE STRUCTURE

The DataFrame library is “almost” a header-only library with a few boilerplate source file exceptions, *HeteroVector.cc* and *HeteroView.cc* and a few others. 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 (they are mostly located in the *Internals* subdirectory). The main header file is *DataFrame.h*. It contains the DataFrame class and its interface. There are comprehensive comments for each public interface call in that file. The rest of the files there will show you how the sausage is made.

Include directory also contains subdirectories that contain mostly internal DataFrame implementation.

One exception, the *DateTime.h* is located in the *Utils* subdirectory

*SPC* directory contains Linux-only make files and a few subdirectories that contain various source codes.

*test* directory contains all the test source files, mocked data files, and test output 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.

Since the containers are static, they are not thread safe by default. See below for how to make them thread safe.

```
using namespace hmdf;

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

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>
("INDEX", sort_spec::ascen);
// Sort the Frame by column "dbl_col_2"
df.sort<double, int, double, std::string>("dbl_col_2", sort_spec::desce);

// A functor to calculate mean, variance, skew, kurtosis, defined in
// DataFrameStatsVisitors.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>(Index2D<long> { 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;

```

### Multithreading safe Example:

```

const size_t    vec_size = 100000;
auto            do_work = [vec_size]() {
    MyDataFrame df;
    std::vector<size_t> vec;

    for (size_t i = 0; i < vec_size; ++i)
        vec.push_back(i);

    df.load_data(MyDataFrame::gen_sequence_index(0, vec_size, 1),
                 std::make_pair("col1", vec));

    // This is an extremely inefficient way of doing it, especially in
    // a multithreaded program. Each "get_column" is a hash table
    // look up and in multithreaded programs requires a lock.
    // It is much more efficient to call "get_column" outside the loop
    // and loop over the referenced vector.
    // Here I am doing it this way to make sure synchronization
    // between threads are bulletproof.
    //
    for (size_t i = 0; i < vec_size; ++i) {
        const size_t j = df.get_column<size_t>("col1")[i];

        assert(i == j);
    }
    df.shrink_to_fit();
};

SpinLock        lock;
std::vector<std::thread> thr_vec;

// Use this lock to protect internal DataFrame static members
MyDataFrame::set_lock(&lock);
for (size_t i = 0; i < 20; ++i)
    thr_vec.push_back(std::thread(do_work));
for (size_t i = 0; i < 20; ++i)
    thr_vec[i].join();
MyDataFrame::remove_lock();

```

For more code examples see file *dataframe\_teststr.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

---

```
enum class sort_spec : unsigned char {  
    ascen = 1,  
    desce = 2,  
};
```

Enumerated type to specify the direction of sort; ascending vs. descending.

---

```
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 exponential_decay_spec : unsigned char {  
    center_of_gravity = 1,    // decay = 1 / (1 + value), for value >= 0  
    span = 2,                // decay = 2 / (1 + value), for value >= 1  
    halflife = 3,            // decay = 1 - exp(log(0.5) / value), for value > 0  
    fixed = 4,               // decay = value, for 0 < value <= 1  
};
```

This spec determines how an exponentially moving stat decays. It is used as a parameter to the ExponentialRollAdopter adopter constructor. Based on this spec, the value parameter is converted to decay.

---

```
enum class io_format : unsigned char {  
    csv = 1,  
    json = 2,
```



```

    hdf5 = 3,    // Not Implemented
    binary = 4, // Not Implemented
};

```

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(present / past)
percentage: (present - past) / past
monetary: present - past

```

---

```

enum class random_policy : unsigned char {
    num_rows_with_seed = 1,    // Number of rows with specifying a seed
    num_rows_no_seed = 2,      // Number of rows with no seed specification
    frac_rows_with_seed = 3,   // Fraction of rows with specifying a seed
    frac_rows_no_seed = 4,     // Fraction of rows with no seed specification
};

```

Specification for calling get\_[data|view]\_by\_rand()

Number of rows means the n parameter is an positive integer specifying the number of rows to select

Fraction of rows means the n parameter is a positive real number [0:1]

specifying a fraction of rows to select

---

```
template<typename T>
struct RandGenParams {
    // Minimum value
    T min_value { std::numeric_limits<T>::min() };
    // Maximum value
    T max_value { std::numeric_limits<T>::max() };

    // Generator seed. By using the same seed, you can generate the
    // same random numbers each time. -1 means do not use a seed,
    // so each generation will be different.
    unsigned int seed { (unsigned int) -1 };

    // The p distribution parameter (probability of generating true)
    double prob_true { 0.5 };
    // The t or k distribution parameter (number of trials)
    std::size_t t_dist { 1 };
    // The  $\mu$  distribution parameter (the mean of the distribution)
    double mean { 1.0 };
    // the  $\sigma$  distribution parameter (standard deviation)
    double std { 0 };
    // The  $\lambda$  distribution parameter (the rate parameter)
    double lambda { 1.0 };
    // The  $\alpha$  distribution parameter (shape, location)
    double alpha { 1.0 };
    // The  $\beta$  distribution parameter (scale)
    double beta { 1.0 };
    // The m distribution parameter (log-scale)
    double m { 0 };
    // The s distribution parameter (shape)
    double s { 1.0 };
    // The n distribution parameter (degrees of freedom)
    double n { 1.0 };
    // degrees of freedom for fisher_f_distribution
    double n2 { 1.0 };
};
```

This structure is used as a list of parameters for the stand-alone random number generators. Note, not all parameters are applicable to all generators. Please refer to each generator documentation in this document for list of applicable parameters.

---

```
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

---

```
struct MemUsage {
```

```
    size_t column_used_memory { 0 };
    size_t column_capacity_memory { 0 };
    size_t column_type_size { 0 };
    size_t index_used_memory { 0 };
    size_t index_capacity_memory { 0 };
    size_t index_type_size { 0 };
```

```
    template<typename S>
```

```
    friend S &operator << (S &stream, const MemUsage &mu);
```

```
};
```

This struct holds the result of calling `get_memory_usage()` method on DataFrame.

---

```
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, “T” stands for the Index type and “H” stands for a Heterogenous vector type:

**static void set\_lock (SpinLock \*lock);**  
**static void remove\_lock ();**

DataFrame has unprotected static data. If you are using DataFrame in a multi-threaded program, you must provide a SpinLock. DataFrame will use your SpinLock to protect its static data.

This is done this way, so by default, there is no locking overhead.

*lock*: A pointer to *SpinLock* defined in *Utils/ThreadGranularity.h* file

---

**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.

---

**static std::vector<IndexType>**

```

gen_sequence_index(const IndexType &start_value,
                  const IndexType &end_value,
                  long increment = 1);

```

This static method generates a vector of sequential values of IndexType that could be fed directly to one of the load methods.

The values are incremented by "increment".

The index type must be incrementable.

If by incrementing "start\_value" by increment you would never reach "end\_value", the behavior will be undefined.

It returns a vector of IndexType values.

*start\_value, end\_value*: Starting and ending values of IndexType.

Start value is included. End value is excluded.

*increment*: Increment by value

---

```

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 ... Ts>  
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.

*Ts*: The list of types for all columns. A type should be specified only once.

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

---

**template<typename ... Ts>  
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.

*Ts*: The list of types for all columns. A type should be specified only once.

*range*: The begin and end iterators for data

---

**template<typename T, typename F, typename ... Ts>  
void**

**remove\_data\_by\_sel (const char \*name, F &sel\_func);**

It removes data rows by boolean filtering selection via the `sel_func` (e.g. a functor, function, or lambda). Each element of the named column along with its corresponding index is passed to the `sel_func`. If `sel_func` returns true, that row will be removed.

The signature of `sel_func`:

*bool ()(const IndexType &, const T &)*

NOTE: If the selection logic results in empty column(s), the empty column(s) will `_not_` be padded with NaN's. You can always call `make_consistent()` afterwards to make all columns into consistent length

*T*: Type of the named column

*F*: Type of the selecting functor

*Ts*: The list of types for all columns. A type should be specified only once



*name*: Name of the data column

*sel\_functor*: A reference to the selecting functor

```
template<typename T1, typename T2, typename F, typename ... Ts>  
void  
remove_data_by_sel (const char *name1,  
                    const char *name2,  
                    F &sel_functor);
```

This does the same function as above `remove_data_by_sel()` but operating on two columns.

The signature of `sel_functor`:

*bool ()(const IndexType &, const T1 &, const T2 &)*

*T1*: Type of the first named column

*T2*: Type of the second named column

*F*: Type of the selecting functor

*Ts*: The list of types for all columns. A type should be specified only once

*name1*: Name of the first data column

*name2*: Name of the second data column

*sel\_functor*: A reference to the selecting functor

```
template<typename T1, typename T2, typename T3, typename F,  
        typename ... Ts>  
void  
remove_data_by_sel (const char *name1,  
                    const char *name2,  
                    const char *name3,  
                    F &sel_functor);
```

This does the same function as above `remove_data_by_sel()` but operating on three columns.

The signature of `sel_functor`:

*bool ()(const IndexType &, const T1 &, const T2 &, const T3 &)*

*T1*: Type of the first named column

*T2*: Type of the second named column

*T3*: Type of the third named column

*F*: Type of the selecting functor

*Ts*: The list of types for all columns. A type should be specified only once

*name1*: Name of the first data column

*name2*: Name of the second data column

*name3*: Name of the third data column

*sel\_functor*: A reference to the selecting functor

---

```
template<size_t N, typename ... Ts>  
void  
shuffle(const std::array<const char *, N> col_names,
```

**bool also\_shuffle\_index);**

It randomly shuffles the named column(s) non-deterministically.

**also\_shuffle\_index**: If true, it shuffles the named column(s) and the index column. Otherwise, index is not shuffled.

*N*: Number of named columns

*Ts*: The list of types for all columns. A type should be specified only once.

---

**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 ... Ts>**

**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

*Ts*: The list of types for all columns. A type should be specified 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<size_t N>
size_type
replace_index(const std::array<IndexType, N> old_values,
              const std::array<IndexType, N> new_values,
              int limit = -1);
```

This does the same thing as `replace()` above for the index column

*N*: Size of `old_values` and `new_values` arrays

*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 ... Ts>
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

*Ts*: The list of types for all columns. A type should be specified only once.

---

```
template<typename ... Ts>
void
shrink_to_fit ();
```

This will reclaim unused/reserve memory from all columns including the index.

If your DataFrame has grown organically from different data sources, `shrink_to_fit()` could potentially reduce your memory footprint significantly.

After this call, any iterator or reference you hold to any data point in the DataFrame could be invalidated.

*Ts*: The list of types for all columns. A type should be specified only once.

---

```
template<typename T, typename ... Ts>
void
```

```
sort(const char *name, sort_spec dir);
```

Sort the DataFrame by the named column. If `name` equals "INDEX", it sorts by index. Otherwise it sorts by the named column. 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

*Ts*: The list of types for all columns. A type should be specified only once.

*name*: The name of a column or string "INDEX".

*dir*: Direction of sorting, ascending or descending

**template<typename T1, typename T2, typename ... Ts>**

**void**

**sort(const char \*name1, sort\_spec dir1, const char \*name2, sort\_spec dir2);**

This sort function sorts DataFrame based on two columns, also specified by the two directions. name[n] could be "INDEX" in which case the index column is taken as the sorting column.

NOTE: The order of T[n] type specifications must match the order of name[n] column names. In addition, all column types must be specified separately. Otherwise, the behavior is undefined.

NOTE: Sort first calls make\_consistent() that may add nan values to data columns.

nan values make sorting nondeterministic.

*T1*: Type of the first named column. You always must specify this type, even if it is being sorted by the index.

*T2*: Type of the second named column. You always must specify this type, even if it is being sorted by the index.

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

*name1*: Name of the first column or string "INDEX"

*name2*: Name of the second column or string "INDEX"

*dir1*: Direction of sorting for the first column

*dir2*: Direction of sorting for the second column

**template<typename T1, typename T2, typename T3, typename ... Ts>**

**void**

**sort(const char \*name1, sort\_spec dir1,  
      const char \*name2, sort\_spec dir2,  
      const char \*name3, sort\_spec dir3);**

This sort function is similar to above, but it uses 4 columns

**template<typename T1, typename T2, typename T3, typename T4,  
          typename ... Ts>**

**void**

**sort(const char \*name1, sort\_spec dir1,  
      const char \*name2, sort\_spec dir2,  
      const char \*name3, sort\_spec dir3,  
      const char \*name4, sort\_spec dir4);**

This sort function is similar to above, but it uses 4 columns

**template<typename T1, typename T2, typename T3, typename T4,**

```

        typename T5, typename ... Ts>
void
sort(const char *name1, sort_spec dir1,
      const char *name2, sort_spec dir2,
      const char *name3, sort_spec dir3,
      const char *name4, sort_spec dir4,
      const char *name5, sort_spec dir5);

```

This sort function is similar to above, but it uses 4 columns

```

template<typename T, typename ... Ts>
std::future<void>
sort_async(const char *name, sort_spec dir = sort_spec::ascen);

```

Same as sort() above, but executed asynchronously

```

template<typename T1, typename T2, typename ... Ts>
std::future<void>
sort_async(const char *name1, sort_spec dir1,
           const char *name2, sort_spec dir2);

```

```

template<typename T1, typename T2, typename T3, typename ... Ts>
std::future<void>
sort_async(const char *name1, sort_spec dir1,
           const char *name2, sort_spec dir2,
           const char *name3, sort_spec dir3);

```

```

template<typename T1, typename T2, typename T3, typename T4,
        typename ... Ts>
std::future<void>
sort_async(const char *name1, sort_spec dir1,
           const char *name2, sort_spec dir2,
           const char *name3, sort_spec dir3,
           const char *name4, sort_spec dir4);

```

```

template<typename T1, typename T2, typename T3, typename T4,
        typename T5, typename ... Ts>
std::future<void>
sort_async(const char *name1, sort_spec dir1,
           const char *name2, sort_spec dir2,
           const char *name3, sort_spec dir3,
           const char *name4, sort_spec dir4,
           const char *name5, sort_spec dir5);

```

---

```

template<typename F, typename T, typename ... Ts>
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

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

A type should be specified in the list only once.

*Ts*: The list of types for all columns. A type should be specified 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

**template<typename F, typename T, typename ... Ts>**

**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 ... Ts>**

**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 means 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

*Ts*: The list of types for all columns. A type should be specified 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 ... Ts>**

**std::future<DataFrame>**

**bucketize\_async (F &&func, const IndexType &bucket\_interval) const;**

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

**template<typename F, typename ... Ts>**

**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 ... Ts>**

**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) In both lhs and rhs, columns with the same name must have the same Type

*RHS\_T*: Type of DataFrame rhs

*Ts*: The list of types for all columns. A type should be specified 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 RHS\_T, typename T, typename ... Ts>**

**StdDataFrame<unsigned int>**

**join\_by\_column(const RHS\_T &rhs, const char \*name, join\_policy jp) 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 returned DataFrame is indexed by a sequence of unsigned integers from 0 to  
N. The returned DataFrame will at least have two columns names lhs.INDEX and  
rhs.INDEX containing the lhs and rhs indices based on join policy.

The following conditions must be met for this method  
to compile and work properly:

- 1) Ordering (< > != ==) must be well defined for the type of the named column.
- 2) Both lhs and rhs must contain the named column
- 3) In both lhs and rhs, columns with the same name must have the same type

*RHS\_T*: Type of DataFrame rhs

*T*: Type of the named column

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

*rhs*: The rhs DataFrame

*name*: Name of the column which the join will be based on

*join\_policy*: Specifies how to join. For example inner join, or left join, etc. (See  
join\_policy definition)

---

**template<typename ... Ts>**

**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.

*Ts*: The list of types for all columns. A type should be specified only once.

*periods*: Number of periods to shift

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

**template<typename ... Ts>**

**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 ... Ts>**

**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.

*Ts*: The list of types for all columns. A type should be specified only once.

*periods*: Number of periods to rotate

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

**template<typename ... Ts>**

**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 ... Ts>**

**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. Currently two formats (i.e. csv, json) are supported specified by the iof parameter.

The csv file format must be:

*INDEX:<Number of data points>:<Comma delimited list of values>*

*<Column1 name>:<Number of data points>:<Column1 type>:<Comma delimited list of values>*

*<Column2 name>:<Number of data points>:<Column2 type>:<Comma delimited list of values>*

*.*

*.*

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

The JSON file format looks like this:

```
{
    "INDEX": {"N": 3, "T": "ulong", "D": [123450, 123451, 123452]},
    "col_3": {"N": 3, "T": "double", "D": [15.2, 16.34, 17.764]},
    "col_4": {"N": 3, "T": "int", "D": [22, 23, 24]},
    "col_str": {"N": 3, "T": "string", "D": ["11", "22", "33"]},
    "col_2": {"N": 3, "T": "double", "D": [8, 9.001, 10]},
    "col_1": {"N": 3, "T": "double", "D": [1, 2, 3.456]}
}
```

Please note DataFrame json does not follow json spec 100%. In json, there is no particular order in dictionary fields. But in DataFrame json:

- 1) Column "INDEX" must be the first column
- 2) Fields in column dictionaries must be in N, T, D order

*S*: Output stream type

*Ts*: The list of types for all columns. A type should be specified 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). Currently two formats (i.e. csv, json) are supported specified by the iof parameter.

The csv file format must be:

```
INDEX:<Number of data points>:<Comma delimited list of values>
<Column1 name>:<Number of data points>:<Column1 type>:<Comma delimited list of values>
<Column2 name>:<Number of data points>:<Column2 type>:<Comma delimited list of values>
.
.
.
```

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

The JSON file format looks like this:

```
{
    "INDEX": {"N": 3, "T": "ulong", "D": [123450, 123451, 123452]},
    "col_3": {"N": 3, "T": "double", "D": [15.2, 16.34, 17.764]},
    "col_4": {"N": 3, "T": "int", "D": [22, 23, 24]},
    "col_str": {"N": 3, "T": "string", "D": ["11", "22", "33"]},
    "col_2": {"N": 3, "T": "double", "D": [8, 9.001, 10]},
    "col_1": {"N": 3, "T": "double", "D": [1, 2, 3.456]}
}
```

```

        "col_2": {"N": 3, "T": "double", "D": [8, 9.001, 10]},
        "col_1": {"N": 3, "T": "double", "D": [1, 2, 3.456]}
    }

```

Please note DataFrame json does not follow json spec 100%. In json, there is no particular order in dictionary fields. But in DataFrame json:

- 3) Column "INDEX" must be the first column
- 4) Fields in column dictionaries must be in N, T, D order

*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

---

**std::pair<size\_type, size\_type> shape();**

It returns a pair containing number of rows and columns.

Note: Number of rows is the number of index rows. Not every column has the same number of rows, necessarily. But each column has, at most, this number of rows.

---

**template<typename T>**

**MemUsage**

**get\_memory\_usage(const char \*col\_name) const;**

It returns the memory used by the given column and index column.

All numbers are in bytes.

MemUsage is a structure defined in DataFrameTypes.h file.

NOTE: The returned values are only estimates. The actual allocated memory by OS is unknown to any container object.

In other words, the actual memory used might be and probably is larger than numbers returned by this call.

Also if a type (T) allocates dynamic memory, it is not included in the result

*T*: Type of the col\_name column.

*col\_name*: Name of the column

---

**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 ... Ts>
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

*Ts*: The list of types for all columns. A type should be specified 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 ... Ts>
DataFrame
get_data_by_idx (Index2D<IndexType> 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.

*Ts*: The list of types for all columns. A type should be specified only once.  
*range*: The begin and end iterators for index specified with index values

**template<typename ... Ts>**

**DataFrame**

**get\_data\_by\_idx(const std::vector<IndexType> &values) const;**

It returns a DataFrame (including the index and data columns) containing the data corresponding to the indices specified in "values" vector.

This method runs in O(n), where n is the number of indices, by creating a hash table of values. IndexType must be hash able.

NOTE: The returned DataFrame is in the same order as original DataFrame

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

*values*: List of indices to copy data from

---

**template<typename ... Ts>**

**DataFrameView<I>**

**get\_view\_by\_idx (Index2D<IndexType> range) const;**

It behaves like get\_data\_by\_idx(range), 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.

*Ts*: The list of types for all columns. A type should be specified only once.

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

**template<typename ... Ts>**

**DataFramePtrView<I>**

**get\_view\_by\_idx(const std::vector<IndexType> &values);**

It behaves like get\_data\_by\_idx(values), but it returns a DataFramePtrView.

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.

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

*values*: List of indices to copy data from

---

**template<typename ... Ts>**

**DataFrame**

**get\_data\_by\_loc (Index2D<long> 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 long.

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

*Ts*: The list of types for all columns. A type should be specified only once.

*range*: The begin and end iterators for data

**template<typename ... Ts>**

**DataFrame**

**get\_data\_by\_loc(const std::vector<long> &locations) const;**

It returns a DataFrame (including the index and data columns) containing the data from locations, specified in locations vector. This function supports Python-like negative indexing. That is why the locations vector type is long.

NOTE: The negative indexing is relative to the "index" column, which may not be the size as all other column.

The returned DataFrame is in the same order as locations parameter

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

*locations*: List of indices into the index column to copy data

---

**template<typename ... Ts>**

**DataFrameView<I>**

**get\_view\_by\_loc (Index2D<long> 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.

*Ts*: The list of types for all columns. A type should be specified only once.

*range*: The begin and end iterators for data

**template<typename ... Ts>**

**DataFrameView<I>**

**get\_view\_by\_loc(const std::vector<long> &locations);**

It behaves like get\_data\_by\_loc(locations), but it returns a DataFramePtrView.

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.

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

*locations*: List of indices into the index column to copy data

---

**template<typename T, typename F, typename ... Ts>**

**DataFrame**

**get\_data\_by\_sel (const char \*name, F &sel\_func) const;**

This method does Boolean filtering selection via the sel\_func (e.g. a functor, function, or lambda). It returns a new DataFrame. Each element of the named column along with its corresponding index is passed to the sel\_func. If sel\_func returns true, that index is selected and all the elements of all column for that index will be included in the returned DataFrame.

The signature of sel\_func:

*bool ()(const IndexType &, const T &)*

NOTE: If the selection logic results in empty column(s), the result empty columns will not be padded with NaN's. You can always call make\_consistent() on the original or result DataFrame to make all columns into consistent length

*T*: Type of the named column

*F*: Type of the selecting functor

*Ts*: The list of types for all columns. A type should be specified only once

*name*: Name of the data column

*sel\_func*: A reference to the selecting functor

**template<typename T, typename F, typename ... Ts>**

**DataFramePtrView<IndexType>**

**get\_view\_by\_sel (const char \*name, F &sel\_func);**

This is identical with above get\_data\_by\_sel(), but:

- 1) The result is a view
- 2) Since the result is a view, you cannot call make\_consistent() on the result.

*T*: Type of the named column

*F*: Type of the selecting functor

*Ts*: The list of types for all columns. A type should be specified only once

*name*: Name of the data column

*sel\_func*: A reference to the selecting functor

**template<typename T1, typename T2, typename F, typename ... Ts>**

**DataFrame**

**get\_data\_by\_sel (const char \*name1,  
                  const char \*name2,  
                  F &sel\_func) const;**

This does the same function as above get\_data\_by\_sel() but operating on two columns.



The signature of `sel_fucntor`:

*bool ()(const IndexType &, const T1 &, const T2 &)*

*T1*: Type of the first named column

*T2*: Type of the second named column

*F*: Type of the selecting functor

*Ts*: The list of types for all columns. A type should be specified only once

*name1*: Name of the first data column

*name2*: Name of the second data column

*sel\_functor*: A reference to the selecting functor

**template<typename T1, typename T2, typename F, typename ... Ts>**

**DataFramePtrView<IndexType>**

**get\_view\_by\_sel (const char \*name1, const char \*name2, F &sel\_functor);**

This is identical with above `get_data_by_sel()`, but:

1) The result is a view

2) Since the result is a view, you cannot call `make_consistent()` on the result.

*T1*: Type of the first named column

*T2*: Type of the second named column

*F*: Type of the selecting functor

*Ts*: The list of types for all columns. A type should be specified only once

*name1*: Name of the first data column

*name2*: Name of the second data column

*sel\_functor*: A reference to the selecting functor

**template<typename T1, typename T2, typename T3, typename F,**

**typename ... Ts>**

**DataFrame**

**get\_data\_by\_sel (const char \*name1,  
const char \*name2,  
const char \*name3,  
F &sel\_functor) const;**

This does the same function as above `get_data_by_sel()` but operating on three columns.

The signature of `sel_fucntor`:

*bool ()(const IndexType &, const T1 &, const T2 &, const T3 &)*

*T1*: Type of the first named column

*T2*: Type of the second named column

*T3*: Type of the third named column

*F*: Type of the selecting functor

*Ts*: The list of types for all columns. A type should be specified only once

*name1*: Name of the first data column

*name2*: Name of the second data column

*name3*: Name of the third data column

*sel\_functor*: A reference to the selecting functor

```
template<typename T1, typename T2, typename T3, typename F,  
        typename ... Ts>  
DataFramePtrView<IndexType>  
get_view_by_sel (const char *name1,  
                 const char *name2,  
                 const char *name3,  
                 F &sel_functor);
```

This is identical with above `get_data_by_sel()`, but:

- 1) The result is a view
- 2) Since the result is a view, you cannot call `make_consistent()` on the result.

*T1*: Type of the first named column

*T2*: Type of the second named column

*T3*: Type of the third named column

*F*: Type of the selecting functor

*Ts*: The list of types for all columns. A type should be specified only once

*name1*: Name of the first data column

*name2*: Name of the second data column

*name3*: Name of the third data column

*sel\_functor*: A reference to the selecting functor

---

```
template<typename ... Ts>
```

```
DataFrame
```

```
get_data_by_rand (random_policy spec, double n, size_type seed = 0) const;
```

It returns a DataFrame (including the index and data columns) containing the data from uniform random selection. `random_policy` determines the behavior of method.

**Note:** The actual number of rows returned might be smaller than requested. That is because the random process might produce the same number more than once.

**Note:** The columns in the result are not padded with NaN.

*Ts*: The list of types for all columns. A type should be specified only once.

*random\_policy*: Please see `random_policy` in `DataFrameTypes.h`. It specifies how this function should proceed.

*n*: Depending on the random policy, it is either the number of rows to sample or a fraction of rows to sample. In case of fraction, for example 0.4 means 40% of rows.

*seed*: depending on the random policy, user could specify a seed. The same seed should always produce the same random selection.

```
template<typename ... Ts>
```

```
DataFramePtrView<IndexType>
```

```
get_view_by_rand (random_policy spec, double n, size_type seed = 0) const;
```

It behaves like `get_data_by_rand()`, 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.

**Note:** The columns in the result are not padded with NaN.

*Ts*: The list of types for all columns. A type should be specified only once.

*random\_policy*: Please see `random_policy` in `DataFrameTypes.h`. It specifies how this function should proceed.

*n*: Depending on the random policy, it is either the number of rows to sample or a fraction of rows to sample. In case of fraction, for example 0.4 means 40% of rows.

*seed*: depending on the random policy, user could specify a seed. The same seed should always produce the same random selection.

---

**const IndexVecType &  
get\_index () const { return (indices\_); }**

It returns a const reference to the index container

**IndexVecType &  
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 fourth 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 fourth 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 ... Ts>  
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

*Ts*: The list of types for all columns. A type should be specified only once.

---

**template<typename ... Ts>**

**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

*Ts*: The list of types for all columns. A type should be specified 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 ... Ts>  
inline DF df_plus (const DF &lhs, const DF &rhs);
```

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

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

```
template<typename DF, typename ... Ts>  
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/DataFrameStatsVisitors.h*, *include/DataFrameMLVisitors.h* and *include/DataFrameFinancialVisitors.h*. Also see *test/data\_frame\_tester.cc* for example usage.

There are some common interfaces in most of the visitors. For example the following interfaces are common between most (but not all) visitors:

*get\_result()* -- It returns the result of the visitor/algo.  
*pre()* -- It is called by DataFrame each time before starting to pass the data to the visitor. *pre()* is the place to initialize the process  
*post()* -- It is called by DataFrame each time it is done with passing data to the visitor.

```
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. The constructor takes a single optional Boolean argument to whether skip NaN values. The default is True.

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 GeometricMeanVisitor;
```

This functor class calculates the geometric mean of a given column. The constructor takes a single optional Boolean argument to whether skip NaN values. The default is True.

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 HarmonicMeanVisitor;
```



This functor class calculates the harmonic mean of a given column. The constructor takes a single optional Boolean argument to whether skip NaN values. The default is True.  
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. The constructor takes a single optional Boolean argument to whether skip NaN values. The default is True.

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. The constructor takes a single optional Boolean argument to whether skip NaN values. The default is True.

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. The constructor takes a single optional Boolean argument to whether skip NaN values. The default is True.

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. The constructor takes a single optional Boolean argument to whether skip NaN values. The default is True.  
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. The constructor takes a single optional Boolean argument to whether skip NaN values. The default is True.  
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. The constructor takes a single optional Boolean argument to whether skip NaN values. The default is True.  
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. The constructor takes a single optional Boolean argument to whether skip NaN values. The default is True.

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. The constructor takes a single optional Boolean argument to whether skip NaN values. The default is True.

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  $O(M*\log M)$ . The constructor takes a single optional Boolean argument to whether skip NaN values. The default is True.

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  $O(M*\log M)$ . The constructor takes a single optional Boolean argument to whether skip NaN values. The default is True.

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.

*explicit CovVisitor (bool bias = true, bool skipnan = true);*

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.

*explicit VarVisitor (bool bias = true);*

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).

*explicit BetaVisitor (bool bias = true);*

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.

*explicit StdVisitor (bool bias = true);*

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.

*explicit TrackingErrorVisitor (bool bias = true);*

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.

*explicit CorrVisitor (bool bias = true);*

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.

*explicit ReturnVisitor (return\_policy rp);*

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

*T*: Column data type

*I*: Index type

*T* must be an arithmetic-enabled type

---

### **template<size\_t K, typename T, typename I = unsigned long> struct KMeansVisitor;**

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 K means in the data. It could also cluster the data round the means.

The constructor takes two parameters

1. Number of iterations
2. A function to calculate distance between to data points of type T with a default value

```
KMeansVisitor(size_type num_of_iter,  
distance_func f =  
    [](const value_type &x, const value_type &y) -> double {  
        return ((x - y) * (x - y));  
    })
```

The result type is an array of K means of type T.

There is also a `get_clusters()` method that returns an array of K `VectorPtrView`'s which contain the data clustered around the K-Means. The first element in each `VectorPtrView` is the mean and the rest are the data points belonging to that cluster.

*K*: Number of means to find

*T*: Column data type

*I*: Index type

---

### **template<typename T, typename I = unsigned long> struct AffinityPropVisitor;**

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 clusters in data using Affinity Propagation algorithm.

The constructor takes three parameters

1. Number of iterations
2. A function to calculate distance between to data points of type T with a default value

```

KMeansVisitor(size_type num_of_iter,
               distance_func f =
               [](const value_type &x, const value_type &y) -> double {
                   return ((x - y) * (x - y));
               })

```

3. Damping factor used in the algorithm. The default is 0.9. (1 – damping factor) prevents numerical oscillations.

The result type is VectorPtrView of type T containing the centers of clusters.

There is also a get\_clusters() method that returns an vector of VectorPtrView's which contain the data clustered around the centers.

*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 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.

*explicit KthValueVisitor (size\_type ke, bool skipnan = true);*

*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 ZScoreVisitor;
```

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 z-score each value in a given column against the same column as the population. Its result is a vector of z-scores.

*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 SampleZScoreVisitor;
```

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 z-score of a sample against its population. It requires two columns. The first column is taken as population and the second column as sample. Its result is a single value.

*T*: Columns 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 F, typename T, typename I = unsigned long>
struct SimpleRollAdopter;
```

This functor applies functor *F* to the data in a rolling progression. The roll count is given to the constructor of *SimpleRollAdopter*.

The result is a vector of values with same number of items as the given column.

The first `roll_count` items, in the result, will be NAN.

```
inline SimpleRollAdopter(F &&functor, size_t roll_count)
```

*F*: Functor type

*T*: Column data type

*I*: Index type

---

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

This functor applies functor *F* to the data in an exponentially decaying rolling progression. The roll count is given to the constructor of *ExponentialRollAdopter*.

The other parameters to the constructor determine how the decay is calculated.

Please see the *exponential\_decay\_spec* type above.

The result is a vector of values with same number of items as the given column.

The first `roll_count` items, in the result, will be NAN.

The formula used is:  $(decay * X_t) + ((1 - decay) * Y_{t-1})$

Where

*X<sub>t</sub>* is current value

*Y<sub>t-1</sub>* is last calculated stats

```
inline ExponentialRollAdopter(F &&functor, size_t roll_count,
exponential_decay_spec eds, double value)
```

*F*: Functor type

*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 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*

---

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

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 crossover of a data vector with two of its moving averages. It could be used to generate signals within financial applications.

The constructor takes the two adopters:

*DoubleCrossOver(S\_RT &&short\_moving, L\_RT &&long\_moving)*

There are 3 methods that gives you the results:

- 1) *const result\_type &get\_raw\_to\_short\_term() const* – Returns a vector of data column minus short moving average
- 2) *const result\_type &get\_raw\_to\_long\_term() const* – Returns a vector of data column minus long moving average
- 3) *const result\_type &get\_short\_term\_to\_long\_term () const* – Returns a vector of short term moving average minus long moving average

*S\_RT*: A short term moving average adopter. For example, a simple moving adopter using a geometric mean

*L\_RT*: A longer term moving average adopter. For example, an exponential moving adopter using a simple mean

*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 BollingerBand;
```

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 Bollinger bands and generates the spread between the given column and lower/upper bands. It could be used to generate signals within financial applications.

The constructor takes:

- Upper band multiplier to be multiplied by standard-deviation and added to the moving average
- Lower band multiplier to be multiplied by standard-deviation and subtracted from the moving average
- Number of periods for a simple moving mean and std.
- Biased; whether the moving std is biased. The default is false meaning the denominator is “ $n - 1$ ”.

```
BollingerBand(double upper_band_multiplier,  
              double lower_band_multiplier,  
              size_type moving_mean_period,  
              bool biased = false)
```

There are 2 methods that gives you the results:

- 1) `const result_type &get_upper_band_to_raw() const` – Returns a vector of upper band minus data column.
- 2) `const result_type &get_raw_to_lower_band() const` – Returns a vector of data column minus lower band.

*T*: Column data type

*I*: Index type

*T must be an arithmetic-enabled type*

## RANDOM GENERATORS

A set of convenient routines to generate random number.

For the definition and defaults of *RandGenParams*, see this *document* and file *DataFrameTypes.h*

It generates n uniform integer distribution random numbers.

$$P(i|a,b) = \frac{1}{b - a + 1}$$

It returns the vector of results

Optional parameters to set:

max\_value, min\_value, seed

*T*: Type can only be [unsigned] char, [unsigned] short, [unsigned] int, [unsigned] long int, or [unsigned] long long int

*n*: Number of numeric to generate

*params*: List of all applicable parameters, see DataFrameTypes.h

**template<typename T>**

**std::vector<T>**

**gen\_uniform\_int\_dist(std::size\_t n,  
const RandGenParams<T> &params = { });**

---

It generates n uniform real distribution random numbers.

$$P(i|a,b) = \frac{1}{b - a}$$

It returns the vector of results

Optional parameters to set:

max\_value, min\_value, seed

*T*: Type can only be float, double, or long double

*n*: Number of numeric to generate

*params*: List of all applicable parameters, see DataFrameTypes.h

**template<typename T>**

**std::vector<T>**

**gen\_uniform\_real\_dist(std::size\_t n,  
const RandGenParams<T> &params = { });**

---

It generates random Boolean values, according to the discrete probability function. The probability of true is:

$$P(b|p) = \begin{cases} p & \text{if } b == \text{true} \\ 1 - p & \text{if } b == \text{false} \end{cases}$$

It returns the vector of results

Optional parameters to set:

prob\_true, seed

*n*: Number of numeric to generate

*params*: List of all applicable parameters, see DataFrameTypes.h

**std::vector<bool>**

**gen\_bernoulli\_dist(std::size\_t n,**

**const RandGenParams<bool> &params = {});**

---

Produces random non-negative integer values *i*, distributed according to discrete probability function:

$$P(i|t,p) = \binom{t}{i} \cdot p^i \cdot (1-p)^{t-i}$$

It returns the vector of results

Optional parameters to set:

t\_dist, prob\_true, seed

*T*: Type can only be [unsigned] char, [unsigned] short, [unsigned] int, [unsigned] long int, or [unsigned] long long int

*n*: Number of numerics to generate

*params*: List of all applicable parameters, see DataFrameTypes.h

**template<typename T>**

**std::vector<T>**

**gen\_binomial\_dist(std::size\_t n, const RandGenParams<T> &params = {});**

---

Produces random non-negative integer values *i*, distributed according to discrete probability function:

$$P(i|k,p) = \binom{k+i-1}{i} \cdot p^i \cdot (1-p)^k$$

It returns the vector of results

Optional parameters to set:

t\_dist, prob\_true, seed

*T*: Type can only be [unsigned] char, [unsigned] short, [unsigned] int, [unsigned] long int, or [unsigned] long long int

*n*: Number of numerics to generate

*params*: List of all applicable parameters, see DataFrameTypes.h

**template<typename T>**

**std::vector<T>**

**gen\_negative\_binomial\_dist(std::size\_t n,**

**const RandGenParams<T> &params = {});**

---

Produces random non-negative integer values *i*, distributed according to discrete probability function:

$$P(i|p)=p \cdot (1-p)^i$$

It returns the vector of results

Optional parameters to set:

prob\_true, seed

*T*: Type can only be [unsigned] char, [unsigned] short, [unsigned] int, [unsigned] long int, or [unsigned] long long int

*n*: Number of numerics to generate

*params*: List of all applicable parameters, see DataFrameTypes.h

**template<typename T>**

**std::vector<T>**

**gen\_geometric\_dist(std::size\_t n, const RandGenParams<T> &params = { });**

Produces random non-negative integer values *i*, distributed according to discrete probability function:

$$P(i|u) = \frac{e^{-u} u^i}{i!}$$

It returns the vector of results

Optional parameters to set:

mean, seed

*T*: Type can only be [unsigned] char, [unsigned] short, [unsigned] int, [unsigned] long int, or [unsigned] long long int

*n*: Number of numerics to generate

*params*: List of all applicable parameters, see DataFrameTypes.h

**template<typename T>**

**std::vector<T>**

**gen\_poisson\_dist(std::size\_t n, const RandGenParams<T> &params = { });**

duces random non-negative floating-point values *x*, distributed according to probability density function:

$$P(x|\lambda) = \lambda e^{-\lambda x}$$

It returns the vector of results

Optional parameters to set:

lambda, seed

*T*: Type can only be float, double, or long double

*n*: Number of numerics to generate

*params*: List of all applicable parameters, see DataFrameTypes.h

**template<typename T>**

**std::vector<T>**

**gen\_exponential\_dist(std::size\_t n,  
const RandGenParams<T> &params = { });**

---

Produces random non-negative floating-point values  $x$ ,  
distributed according to probability density function:

$$p(x|\alpha, \beta) = e^{-x/\beta} \beta \alpha \Gamma(\alpha) \cdot x^{\alpha-1}$$

It returns the vector of results

Optional parameters to set:

alpha, beta, seed

*T*: Type can only be float, double, or long double

*n*: Number of numerics to generate

*params*: List of all applicable parameters, see DataFrameTypes.h

**template<typename T>**

**std::vector<T>**

**gen\_gamma\_dist(std::size\_t n, const RandGenParams<T> &params = { });**

---

Produces random non-negative floating-point values  $x$ ,  
distributed according to probability density function:

see [https://en.cppreference.com/w/cpp/numeric/random/weibull\\_distribution](https://en.cppreference.com/w/cpp/numeric/random/weibull_distribution)

It returns the vector of results

Optional parameters to set:

alpha (shape), beta (scale), seed

*T*: Type can only be float, double, or long double

*n*: Number of numerics to generate

*params*: List of all applicable parameters, see DataFrameTypes.h

**template<typename T>**

**std::vector<T>**

**gen\_weibull\_dist(std::size\_t n, const RandGenParams<T> &params = { });**

---

Produces random numbers according to the extreme value distribution  
(it is also known as Gumbel Type I, log-Weibull, Fisher-Tippett Type I):

see [https://en.cppreference.com/w/cpp/numeric/random/extreme\\_value\\_distribution](https://en.cppreference.com/w/cpp/numeric/random/extreme_value_distribution)

It returns the vector of results

Optional parameters to set:

alpha (location), beta (scale), seed

*T*: Type can only be float, double, or long double

*n*: Number of numerics to generate

*params*: List of all applicable parameters, see DataFrameTypes.h

**template<typename T>**

**std::vector<T>**

**`gen_extreme_value_dist(std::size_t n, const RandGenParams<T> &params = {});`**

---

Generates random numbers according to the Normal (or Gaussian) random number distribution. It is defined as:

see [https://en.cppreference.com/w/cpp/numeric/random/normal\\_distribution](https://en.cppreference.com/w/cpp/numeric/random/normal_distribution)

It returns the vector of results

Optional parameters to set:

mean, std (scale), seed

*T*: Type can only be float, double, or long double

*n*: Number of numerics to generate

*params*: List of all applicable parameters, see DataFrameTypes.h

**`template<typename T>`**

**`std::vector<T>`**

**`gen_normal_dist(std::size_t n, const RandGenParams<T> &params = {});`**

---

The lognormal\_distribution random number distribution produces random numbers  $x > 0$  according to a log-normal distribution:

see [https://en.cppreference.com/w/cpp/numeric/random/lognormal\\_distribution](https://en.cppreference.com/w/cpp/numeric/random/lognormal_distribution)

It returns the vector of results

Optional parameters to set:

m (log-scale), s (shape), seed

*T*: Type can only be float, double, or long double

*n*: Number of numerics to generate

*params*: List of all applicable parameters, see DataFrameTypes.h

**`template<typename T>`**

**`std::vector<T>`**

**`gen_lognormal_dist(std::size_t n,  
const RandGenParams<T> &params = {});`**

---

The chi\_squared\_distribution produces random numbers  $x > 0$  according to the Chi-squared distribution:

see [https://en.cppreference.com/w/cpp/numeric/random/chi\\_squared\\_distribution](https://en.cppreference.com/w/cpp/numeric/random/chi_squared_distribution)

It returns the vector of results

Optional parameters to set:

n (degree of freedom), seed

*T*: Type can only be float, double, or long double

*n*: Number of numerics to generate

*params*: List of all applicable parameters, see DataFrameTypes.h

**`template<typename T>`**

**`std::vector<T>`**



**gen\_chi\_squared\_dist(std::size\_t n,  
const RandGenParams<T> &params = {});**

---

Produces random numbers according to a Cauchy distribution  
(also called Lorentz distribution):

see [https://en.cppreference.com/w/cpp/numeric/random/cauchy\\_distribution](https://en.cppreference.com/w/cpp/numeric/random/cauchy_distribution)

It returns the vector of results

Optional parameters to set:

alpha (location), beta (scale), seed

*T*: Type can only be float, double, or long double

*n*: Number of numerics to generate

*params*: List of all applicable parameters, see DataFrameTypes.h

**template<typename T>**

**std::vector<T>**

**gen\_cauchy\_dist(std::size\_t n, const RandGenParams<T> &params = {});**

---

Produces random numbers according to the f-distribution:

see [https://en.cppreference.com/w/cpp/numeric/random/fisher\\_f\\_distribution](https://en.cppreference.com/w/cpp/numeric/random/fisher_f_distribution)

It returns the vector of results

Optional parameters to set:

n (degree of freedom), n2 (degree of freedom), seed

*T*: Type can only be float, double, or long double

*n*: Number of numerics to generate

*params*: List of all applicable parameters, see DataFrameTypes.h

**template<typename T>**

**std::vector<T>**

**gen\_fisher\_f\_dist(std::size\_t n, const RandGenParams<T> &params = {});**

---

Produces random floating-point values x, distributed according  
to probability density function:

see [https://en.cppreference.com/w/cpp/numeric/random/student\\_t\\_distribution](https://en.cppreference.com/w/cpp/numeric/random/student_t_distribution)

It returns the vector of results

Optional parameters to set:

n (degree of freedom), seed

*T*: Type can only be float, double, or long double

*n*: Number of numerics to generate

*params*: List of all applicable parameters, see DataFrameTypes.h

**template<typename T>**

**std::vector<T>**

**gen\_student\_t\_dist(std::size\_t n, const RandGenParams<T> &params = {});**

---

