Table of Contents

Why Use the DataFrames Package?

We want to use Julia for statistical programming. While Julia is a very powerful tool for general mathematical programming, it is missing several basic features that are essential for statistical applications. This introductory section describes some of the features that are missing from Julia's core. The rest of the manual describes the ways in which the DataFrames package extends Julia to make up for those missing features.

Missing Data Points

Suppose that we want to calculate the mean of a list of five `Float64` numbers: `x1`, `x2`, `x3`, `x4` and `x5`. We would normally do this in Julia as follows:

- Represent these five numbers as a `Vector`: `v = [x1, x2, x3, x4, x5]`.
- Compute the mean of `v` using the `mean()` function.

But what if one of the five numbers were missing?

The concept of a missing data point cannot be directly expressed in Julia because there is no scalar value to denote missingness. While Java has a `NULL` value and R has an `NA` value, there is, *by design*, nothing equivalent in Julia. As such, the DataFrames package's first extension of Julia's core type system is to add a new `NA` value.

Data Structures for Storing Missing Data Points

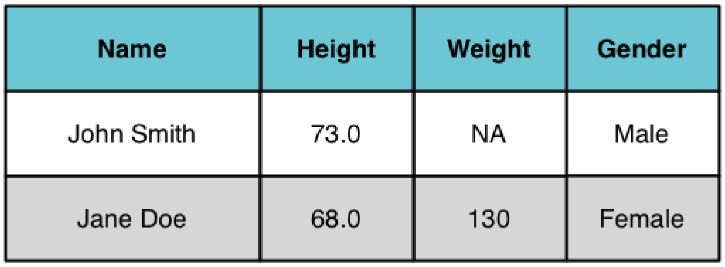
Even if we can express the notion that the value of the numeric variable `x2` is unknown by using a new `NA` value, there is little that we can do with this new value because it cannot be directly stored in a standard Julian `Vector` unless that `Vector` has no type constraints on its entries. While we could use a `Vector{Any}` to work around this, that approach would produce very inefficient code.

Instead of trying to use overly generic data structures, we have created extensions of the core Julia `Vector` and `Matrix` data structures that can store `NA` values. These augmented data structures are called `DataVector` and `DataMatrix`. Both `DataVector{T}` and `DataMatrix{T}` can contain either (a) values of any specific type `T` or (b) values of our new `NA` type.

For example, a standard `Vector{Float64}` can contain `Float64` and nothing else. Our new `DataVector{Float64}` can contain `Float64` or `NA` values, but nothing else. This makes the new data types much more efficient than using generic containers like `Vector{Any}` or `Matrix{Any}`.

Tabular Data Structures

`DataVector` and `DataMatrix` are very powerful data structures, but they are not sufficient for describing most real world data sets. Although most standard data sets are easily described using a simple table of data, these kinds of tables are generally not like matrices. The example table of data shown below highlights some of the ways in which a data set is not like a `DataMatrix`:



Name	Height	Weight	Gender
John Smith	73.0	NA	Male
Jane Doe	68.0	130	Female

Figure 1: Tabular Data

We highlight three major differences below:

- The columns of a tabular data set may have different types. A `DataMatrix` can only contain values of one type: these might all be `String` or `Int`, but we cannot have one column of `String` and another column of `Int`.
- The values of the entries within a column generally have a consistent type. This means that a single column could be represented using a `DataVector`. Unfortunately, the heterogeneity of types between columns means that we need some way of wrapping a group of columns together into a coherent whole. We could use a `Vector` to wrap up all of the columns of the table, but this will not enforce an important constraint imposed by our intuitions: *every column of a tabular data set has the same length as all of the other columns.* A tabular data set is not just a haphazard collection of vectors.

- The columns of a tabular data set are typically named using `String`. Most programs for working with data can access the columns of a data set using these names in addition to simple numeric indices. In other words, a tabular data structure can sometimes behave like an `Array` and can sometimes behave like a `Dict`. This dual indexing strategy makes it particular easy to work with tabular data.

We can summarize these concerns by noting that we face four problems when with working with tabular data sets that are not well solved by existing Julian data structures:

- Tabular data sets may have heterogeneous types of columns.
- Each column of a tabular data set has a consistent type.
- All columns of a tabular data set have a consistent length, although some entries within columns may be missing.
- The columns of a tabular data set should be addressable by both name and numeric index.

We solve all of these four problems by adding a `DataFrame` type to Julia. This type will be familiar to anyone who has worked with R's `data.frame` type or with Pandas' `DataFrame` type. Even if you have never used R or Python to work with data, this tabular data structure will be satisfy many of the intuitions that you've developed while working with spreadsheet programs like Excel.

A Language for Expressing Statistical Models

Statistical programming is generally focused on answering substantive questions about the properties of a data set. We are generally not interested in thinking about algorithms, but instead want to spend our time thinking about mathematical models.

Part of the power of the R programming language is that it provides a coherent mini-language for talking about various types of linear models ranging from ANOVA's to GLM's. For example, R describes a regression in which a variable `Z` is regressed against the variables `X` and `Y` using the notation:

```
Z ~ X + Y
```

Julia, by default, provides no similar sort of mini-language for describing the mathematical structure of statistical models. To remedy this, we have added a `Formula` type to Julia that provides a simple DSL for describing linear models in Julia.

Getting Started

Installation

The DataFrames package is available through the Julia package system. The first time you use it you will need to run

```
Pkg.add("DataFrames")
```

Loading the DataFrames Package

In all of the examples that follow, we're going to assume that you've already loaded the DataFrames package. You can do that by typing the following commands before trying out any of the examples in this manual:

```
using DataFrames
```

Some Basic Examples

As we described in the introduction, the first thing you'll want to do is to confirm that we have a new type that represents a missing value. Type the following into the REPL to see that this is working for you:

```
NA
```

One of the essential properties of `NA` is that it poisons other items. To see this, try to add something to `NA`:

```
1 + NA
```

As we described earlier, you'll get a lot more power out of `NA` values when they can occur in other data structures. Let's create our first `DataVector` now:

```
dv = DataArray([1, 3, 2, 5, 4])
dv[1] = NA
```

To see how `NA` poisons even complex calculations, let's try to take the mean of those five numbers:

```
mean(dv)
```

In many cases we're willing to just ignore `NA` values and remove them from our vector. We can do that using the `removeNA` function:

```
removeNA(dv)
mean(removeNA(dv))
```

Instead of removing `NA` values, you can try to ignore them using the `failNA` function. The `failNA` function attempt to convert a `DataVector{T}` to a `Vector{T}` and will throw an error if any `NA` values are encountered. If we were dealing with a vector like the following, `failNA` will work just right:

```
dv = DataArray([1, 3, 2, 5, 4])
mean(failNA(dv))
```

In addition to removing or ignoring `NA` values, it's possible to replace them using the `replaceNA` function:

```
dv = DataArray([1, 3, 2, 5, 4])
dv[1] = NA
mean(replaceNA(dv, 11))
```

Which strategy for dealing with `NA` values is most appropriate will typically depend on the details of your situation.

In modern data analysis `NA` values don't simply arise in vector-like data. The `DataMatrix` and `DataFrame` structures are also capable of handling `NA` values. You can confirm for yourself that the presence of `NA` values poisons matrix operations in the same way that it poisons vector operations by creating a simple `DataMatrix` and trying to perform matrix multiplication:

```
dm = DataArray([1.0 0.0; 0.0 1.0])
dm[1, 1] = NA
dm * dm
```

Working with Tabular Data Sets

As we said before, working with simple `DataVector` and `DataMatrix` gets boring after a while. To express interesting types of tabular data sets, we'll create a simple `DataFrame` piece-by-piece:

```
df = DataFrame()
df["A"] = 1:4
df["B"] = ["M", "F", "F", "M"]
df
```

Because we know all of the columns in advance, this specific `DataFrame` could have been created more concisely using keyword arguments:

```
df = DataFrame(A = 1:4, B = ["M", "F", "F", "M"])
```

In practice, we're more likely to use an existing data set than to construct one from scratch. To load a more interesting data set, we can use the `readtable()` function. To make use of it, we'll need a data set stored in a simple format like the comma separated values (CSV) standard. There are some simple examples of CSV files included with the DataFrames package. We can find them using basic file operations in Julia:

```
mydir = Pkg.dir("DataFrames", "test", "data", "scaling")
filenames = readdir(mydir)
df = readtable(joinpath(mydir, filenames[1]))
```

The resulting `DataFrame` has many similar rows. We can check its size using the `size` function:

```
nrows = size(df, 1)
ncols = size(df, 2)
```

We can also look at small subsets of the data in a couple of ways:

```
head(df)
tail(df)

df[1:3, :]
```

Having seen what some of the rows look like, we can try to summarize the entire data set using:

```
describe(df)
```

To focus our search, we start looking at just the means and medians of the columns:

```
mean(df, 1)
median(df, 1)
```

Or, alternatively, we can look at the columns one-by-one:

```
mean(df["E"])
range(df["E"])
```

If you'd like to get your hands on more data to play with, we strongly encourage you to try out the RDatasets package. This package supplements the DataFrames package by providing access to 570 classical data sets that will be familiar to R programmers. You can install and load the RDatasets package using the Julia package manager:

```
Pkg.add("RDatasets")
using RDatasets
```

Once that's done, you can use the `data()` function from RDatasets to gain access to data sets like Fisher's Iris data:

```
iris = data("datasets", "iris")
head(iris)
```

The Iris data set is a really interesting testbed for examining simple contrasts between groups. To get at those kind of group differences, we can split apart our data set based on the species of flower being studied and then analyze each group separately. To do that, we'll use the Split-Apply-Combine strategy made popular by R's plyr library. In Julia, we do this using the `by` function:

```
function g(df)
    res = DataFrame()
    res["nrows"] = nrow(df)
    res["MeanPetalLength"] = mean(df["Petal.Length"])
    res["MeanPetalWidth"] = mean(df["Petal.Width"])
    return res
end

by(iris, "Species", g)
```

Instead of passing in a function that constructs a `DataFrame` piece-by-piece to summarize each group, you can pass in a Julia expression that will construct columns one-by-one. The simplest example looks like:

```
by(iris, "Species", :(NewColumn = 1))
```

This example is admittedly a little silly. The reason we've started with something trivial is that it's quite difficult to work with our current version of the `iris` `DataFrame` because the current set of column names includes names like `"Petal.Length"`, which are not valid Julia variable names. As such, we can't use these names in Julia expressions. To work around that, the DataFrames package provides a function called `clean_colnames!()` which will replace non-alphanumeric characters with underscores in order to produce valid Julia identifiers:

```
clean_colnames!(iris)
colnames(iris)
```

Now that the column names are clean, we can put the expression-based

```
by(iris, "Species", :(MeanPetalLength = mean(Petal_Length)))
by(iris, "Species", :(MeanPetalWidth = mean(Petal_Width)))
```

This style of expression-based manipulation is quite handy once you get used to it. But sometimes you need to summarize groups based on properties of the entire group-level `DataFrame` rather than something describable using just the column names alone. In that case, you can exploit the fact that each group-level DataFrame is temporarily given the name `_DF`:

```
by(iris, "Species", :(N = nrow(_DF)))
```

If none of these ways of working with individual groups of data appeal to you, you can also use the `groupby` function to produce an iterable set of `DataFrame` that you can step though one-by-one:

```
for df in groupby(iris, "Species")
    println({unique(df["Species"]),
             mean(df["Petal_Length"]),
             mean(df["Petal_Width"])})
end
```

We hope this brief tutorial introduction has convinced you that you can do quite complex data manipulations using the DataFrames package. To really dig in, we're now going to describe the design of the DataFrames package in greater depth.

DataFrames I/O

Reading data in from tabular data files

To read data from a normal delimited values file, use the `readtable` function. Some examples are shown below:

```
using DataFrames

df = readtable("data.csv")

df = readtable("data.tsv")

df = readtable("data.wsv")

df = readtable("data.txt", separator = '\t')

df = readtable("data.txt", header = false)
```

`readtable` requires that you specify the path of the file that you would like to read.

In addition to this one required argument, `readtable` accepts the following optional keyword arguments:

- `header::Bool` – Use the information from the file's header line to determine column names. Defaults to `true`.
- `separator::Char` – Assume that fields are split by the `separator` character. If not specified, it will be guessed from the filename: `.csv` defaults to `','`, `.tsv` defaults to `'\t'`, `.wsv` defaults to `' '`.
- `quotemark::Char` – Assume that fields contained inside of two `quotemark` characters are quoted, which disables processing of separators and linebreaks. Defaults to `'"'`.
- `decimal::Char` – Assume that the decimal place in numbers is written using the `decimal` character. Defaults to `','`.
- `nastrings::Vector{ASCIIString}` – Translate any of the strings into this vector into an `NA`. Defaults to `["", "NA"]`.

- `truestrings::Vector{ASCIIString}` – Translate any of the strings into this vector into a Boolean `true`. Defaults to `["T", "t", "TRUE", "true"]`.

- `falsestrings::Vector{ASCIIString}` – Translate any of the strings into this vector into a Boolean `true`. Defaults to `["F", "f", "FALSE", "false"]`.

- `makefactors::Bool` – Convert string columns into `PooledDataVector`'s for use as factors. Defaults to `false`.

- `nrows::Int` – Read only `nrows` from the file. Defaults to `-1`, which indicates that the entire file should be read.

- `colnames::Vector{UTF8String}` – Use the values in this array as the names for all columns instead of or in lieu of the names in the file's header. Defaults to `[]`, which indicates that the header should be used if present or that numeric names should be invented if there is no header.

- `cleannames::Bool` – Call `cleancolnames!` on the resulting DataFrame to ensure that all column names are valid identifers in Julia.

- `coltypes::Vector{Any}` – Specify the types of all columns. Defaults to `{}`.

- `allowcomments::Bool` – Ignore all text inside comments. Defaults to `false`.

- `commentmark::Char` – Specify the character that starts comments. Defaults to `'#'`.

- `ignorepadding::Bool` – Ignore all whitespace on left and right sides of a field. Defaults to `true`.

- `skipstart::Int` – Specify the number of initial rows to skip. Defaults to `0`.

- `skiprows::Vector{Int}` – Specify the indices of lines in the input to ignore. Defaults to `[]`.

- `skipblanks::Bool` – Skip any blank lines in input. Defaults to `true`.

- `encoding::Symbol` – Specify the file's encoding as either `:utf8` or `:latin1`. Defaults to `:utf8`.

- `allowquotes::Bool` – Ignore the special meaning of quotes. Defaults to `false`.

Exporting data to a tabular data file

To export data into a standard format, use the `writetable` function. Some examples are shown below:

```
using DataFrames

df = DataFrame(A = 1:10)

writetable("output.csv", df)

writetable("output.dat", df, separator = ',', header = false)

writetable("output.dat", df, quotemark = '\'', separator = ',')

writetable("output.dat", df, header = false)
```

`writetable` requires the following arguments:

- `filename::String` – The path of the file that you wish to write to.
- `df::DataFrame` – The DataFrame you wish to write to disk.

In addition to the two required arguments, `writetable` accepts the following optional keyword arguments:

- `separator::Char` – The separator character that you would like to use. Defaults to the output of `getseparator(filename)`, which uses commas for files that end in `.csv`, tabs for files that end in `.tsv` and a single space for files that end in `.wsv`.
- `quotemark::Char` – The character used to delimit string fields. Defaults to `'"'`.
- `header::Bool` – Should the file contain a header that specifies the column names from `df`. Defaults to `true`.

The Design of DataFrames

The Type Hierarchy

Before we do anything else, let's go through the hierarchy of types introduced by the DataFrames package. This type hierarchy is depicted visually in the figures at the end of this section and can be summarized in a simple nested list:

- NAtype
- AbstractDataVector
  - DataVector
  - PooledDataVector
- AbstractMatrix
  - DataMatrix
- AbstractDataArray
  - DataArray
- AbstractDataFrame
  - DataFrame
- AbstractDataStream
  - FileDataStream
  - DataFrameDataStream
  - MatrixDataStream

We'll step through each element of this hierarchy in turn in the following sections.

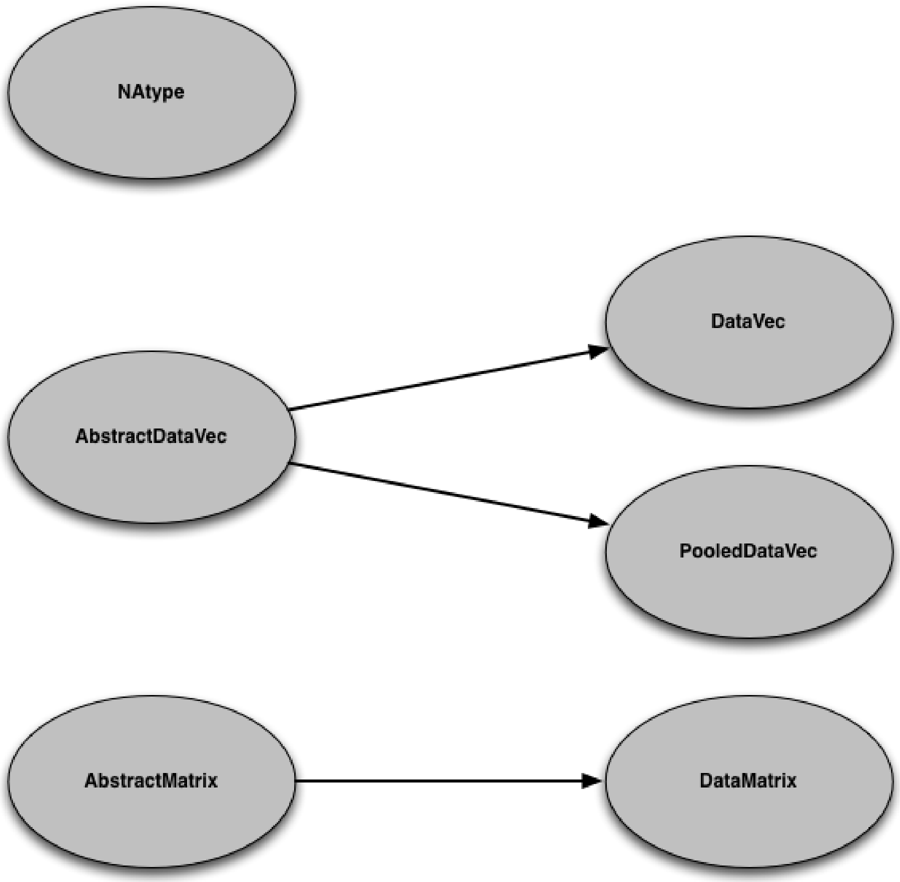


Figure 2: Scalar and Array Types

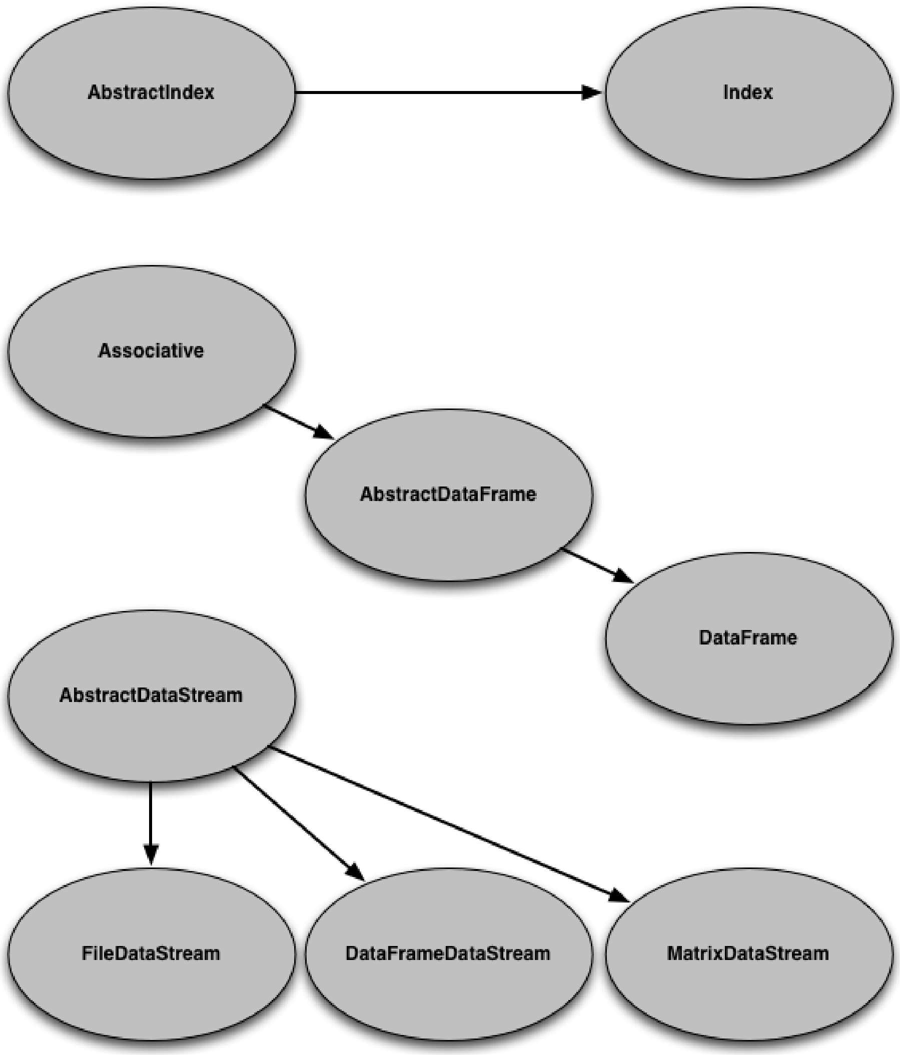


Figure 3: Tabular Data Types

Overview of Basic Types for Working with Data

There are four new types introduced by the current generation of the DataFrames package:

- NAType: A scalar value that represents a single missing piece of data. This value behaves much like `NA` in R.
- DataVector: A vector that can contain values of a specific type as well as `NA` values.
- PooledDataVector: An alternative to DataVector's that can be more memory-efficient if a small number of distinc values are present in the underlying vector of data.
- DataFrame: A tabular data structure that is similar to R's `data.frame` and Pandas' `DataFrame`.

In the future, we will also be introducing generic Arrays of arbitrary dimension. After this, we will provide two new types:

- DataMatrix: A matrix that can contain values of a specific type as well as `NA` values.
- DataFrame: An array that can contain values of a specific type as well as `NA` values.

The NA Type

The core problem with using the data structures built into Julia for data analysis is that there is no mechanism for expressing the absence of data. Traditional database systems express the absence of data using a `NULL` value, while data analysis packages typically follow the tradition set by S and use `NA` for this purpose when referring to data. (NB: *In S and R, `NULL` is present in addition to `NA`, but it refers to the absence of any specific value for a variable in code, rather than the absence of any specific value for something inside of a data set.*)

The DataFrames package expresses the absence of data by introducing a new type called `NAtype`. This value is used everywhere to indicate missingness in the underlying data set.

To see this value, you can type

```
NAtype
```

in the Julia REPL. You can learn more about the nature of this new type using standard Julia functions for navigating Julia's type system:

```
typeof(NAtype)

super(NAtype)

dump(NAtype)
```

While the `NAtype` provides the essential type needed to express missingness, the practical way that missing data is denoted uses a special constant `NA`, which is an instance of `NAtype`:

```
NA
NAtype()
```

You can explore this value to confirm that `NA` is just an instance of the `NAtype`:

```
typeof(NA)

dump(NA)
```

Simply being able to express the notion that a data point is missing is important, but we're ultimately not interested in just expressing data: we want to build tools for interacting with data that may be missing. In a later section, we'll describe the details of interacting with `NA`, but for now we'll state the defining property of `NA`: *because `NA` expresses ignorance about the value of something, every interaction with `NA` corrupts known values and transforms them into `NA` values.* Below we show how this works for addition:

```
1 + NA
```

We'll discuss the subtleties of `NA` values ability to corrupt known values in a later section. For now the essential point is this: `NA` values exist to represent missingness that occurs in scalar data.

The DataVector Type

To express the notion that a complex data structure like an `Array` contains missing entries, we need to construct a new data structure that can contain standard Julia values like `Float64` while also allowing the presence of `NA` values.

Of course, a Julian `Array{Any}` would allow us to do this:

```
{1, NA}
```

But consistently using `Any` arrays would make Julia much less efficient. Instead, we want to provide a new data structure that parallels a standard Julia `Array`, while allowing exactly one additional value: `NA`.

This new data structure is the `DataVector` type. You can construct your first `DataVector` using the following code:

```
DataVector[1, NA, 3]
```

As you'll see when entering this into the REPL, this snippet of code creates a `3-element DataVector{Int64}`. A `DataVector` of type `DataVector{Int64}` can store `Int64` values or `NA` values. In general, a `DataVector` of type `DataVector{T}` can store values of type `T` or `NA` values.

This is achieved by a very simple mechanism: a `DataVector{T}` is a new parametric composite type that we've added to Julia that wraps around a standard Julia `Vector` and complements this basic vector with a metadata store that indicates whether any entry of the wrapped vector is missing. In essence, a `DataVector` of type `T` is defined as:

```
type DataVector{T}
    data::Vector{T}
    na::BitVector
end
```

This allows us to assess whether any entry of the vector is `NA` at the cost of exactly one additional bit per item. We are able to save space by using `BitArray` instead of an `Array{Bool}`. At present, we store the non-missing data values in a vector called `data` and we store the metadata that indicates which values are missing in a vector called `na`. But end-users should not worry about these implementation details.

Instead, you can simply focus on the behavior of the `DataVector` type. Let's start off by exploring the basic properties of this new type:

```
DataVector

typeof(DataVector)
typeof(DataVector{Int64})

super(DataVector)
super(super(DataVector))

DataVector.names
```

If you want to drill down further, you can always run `dump()`:

```
dump(DataVector)
```

We're quite proud that the definition of `DataVector` is so simple: it makes it easier for end-users to start contributing code to the DataFrames package.

Constructing DataVector's

Let's focus on ways that you can create new `DataVector`. The simplest possible constructor requires the end-user to directly specify both the underlying data values and the missingness metadata as a `BitVector`:

```
dv = DataArray([1, 2, 3], falses(3))
```

This is rather ugly, so we've defined many additional constructors that make it easier to create a new `DataVector`. The first simplification is to ignore the distinction between a `BitVector` and an `Array{Bool, 1}` by allowing users to specify `Bool` values directly:

```
dv = DataArray([1, 2, 3], [false, false, false])
```

In practice, this is still a lot of useless typing when all of the values of the new `DataVector` are not missing. In that case, you can just pass a Julian vector:

```
dv = DataArray([1, 2, 3])
```

When the values you wish to store in a `DataVector` are sequential, you can cut down even further on typing by using a Julian `Range`:

```
dv = DataArray(1:3)
```

In contrast to these normal-looking constructors, when some of the values in the new `DataVector` are missing, there is a very special type of constructor you can use:

```
dv = DataVector[1, 2, NA, 4]
```

Technical Note: This special type of constructor is defined by overloading the `getindex()` function to apply to values of type `DataVector`.

DataVector's with Special Types

One of the virtues of using metadata to represent missingness instead of sentinel values like `NaN` is that we can easily define `DataVector` over arbitrary types. For example, we can create `DataVector` that store arbitrary Julia types like `ComplexPair` and `Bool`:

```
dv = DataArray([1 + 2im, 3 - 1im])

dv = DataArray([true, false])
```

In fact, we can add a new type of our own and then wrap it inside of a new sort of `DataVector`:

```
type MyNewType
    a::Int64
    b::Int64
    c::Int64
end

dv = DataArray([MyNewType(1, 2, 3), MyNewType(2, 3, 4)])
```

Of course, specializing the types of `DataVector` means that we sometimes need to convert between types. Just as Julia has several specialized conversion functions for doing this, the DataFrames package provides conversion functions as well. For now, we have three such functions:

- `dataint()`
- `datafloat()`
- `databool()`

Using these, we can naturally convert between types:

```
dv = DataArray([1.0, 2.0])

dataint(dv)
```

In the opposite direction, we sometimes want to create arbitrary length `DataVector` that have a specific type before we insert values:

```
dv = DataArray(Float64, 5)

dv[1] = 1
```

DataArray created in this way have NA in all entries. If you instead wish to initialize a DataArray with standard initial values, you can use one of several functions:

- datazeros()
- dataones()
- datafalses()
- datatrues()

Like the similar functions in Julia's Base, we can specify the length and type of these initialized vectors:

```
dv = datazeros(5)
dv = datazeros(Int64, 5)

dv = dataones(5)
dv = dataones(Int64, 5)

dv = datafalses(5)

dv = datatrues(5)
```

The PooledDataArray Type

On the surface, PooledDataArrays look like DataArrays, but their implementation allows the efficient storage and manipulation of DataVectors and DataArrays which only contain a small number of values. Internally, PooledDataArrays hold a pool of unique values, and the actual DataArray simply indexes into this pool, rather than storing each value individually.

A PooledDataArray can be constructed from an Array or DataArray, and as with regular DataArrays, it can hold NA values:

```
pda  = PooledDataArray([1, 1, 1, 1, 2, 3, 2, 2, 3, 3, 3])
    pda2 = PooledDataArray(DataArray["red", "green", "yellow", "yellow", "red", "orange", "r
```

PooledDataArrays can also be created empty or with a fixed size and a specific type:

```
pda3 = PooledDataArray(String, 2000)   # A pooled data array of 2000 strings, intially fille
pda4 = PooledDataArray(Float64)        # An empty pooled data array of floats
```

By default, the index into the pool of values is a Uint32, allowing 2^32 possible pool values. If you know that you will only have a much smaller number of unique values, you can specify a smaller reference index type, to save space:

```
pda5 = PooledDataArray(String, Uint8, 5000, 2)  # Create a 5000x2 array of String values,
                                                # initialized to NA,
                                                  # with at most 2^8=256 unique values
```

`PooledDataVectorss` can be used as columns in DataFrames.

The DataFrame Type

While `DataVector` are a very powerful tool for dealing with missing data, they only bring us part of the way towards representing real-world data in Julia. The final missing data structure is a tabular data structure of the sort used in relational databases and spreadsheet software.

To represent these kinds of tabular data sets, the DataFrames package provides the `DataFrame` type. The `DataFrame` type is a new Julian composite type with just two fields:

- `columns`: A Julia `Vector{Any}`, each element of which will be a single column of the tabular data. The typical column is of type `DataVector{T}`, but this is not strictly required.
- `colindex`: An `Index` object that allows one to access entries in the columns using both numeric indexing (like a standard Julian `Array`) or key-valued indexing (like a standard Julian `Dict`). The details of the `Index` type will be described later; for now, we just note that an `Index` can easily be constructed from any array of `ByteString`. This array is assumed to specify the names of the columns. For example, you might create an index as follows: `Index(["ColumnA", "ColumnB"])`.

In the future, we hope that there will be many different types of `DataFrame`-like constructs. But all objects that behave like a `DataFrame` will behave according to the following rules that are enforced by an `AbstractDataFrame` protocol:

- A DataFrame-like object is a table with `M` rows and `N` columns.
- Every column of a DataFrame-like object has its own type. This heterogeneity of types is the reason that a DataFrame cannot simply be represented using a matrix of `DataVector`.
- Each columns of a DataFrame-like object is guaranteed to have length `M`.

- Each columns of a DataFrame-like object is guaranteed to be capable of storing an `NA` value if one is ever inserted. NB: *There is ongoing debate about whether the columns of a DataFrame should always be `DataVector` or whether the columns should only be converted to `DataVector` if an `NA` is introduced by an assignment operation.*

Constructing DataFrame's

Now that you understand what a `DataFrame` is, let's build one:

```
df_columns = {datazeros(5), datafalses(5)}
df_colindex = Index(["A", "B"])

df = DataFrame(df_columns, df_colindex)
```

In practice, many other constructors are more convenient to use than this basic one. The simplest convenience constructors is to provide only the columns, which will produce default names for all the columns.

```
df = DataFrame(df_columns)
```

One often would like to construct `DataFrame` from columns which may not yet be `DataVector`. This is possible using the same type of constructor. All columns that are not yet `DataVector` will be converted to `DataVector`:

```
df = DataFrame({ones(5), falses(5)})
```

Often one wishes to convert an existing matrix into a `DataFrame`. This is also possible:

```
df = DataFrame(ones(5, 3))
```

Like `DataVector`, it is possible to create empty `DataFrame` in which all of the default values are `NA`. In the simplest version, we specify a type, the number of rows and the number of columns:

```
df = DataFrame(Int64, 10, 5)
```

Alternatively, one can specify a `Vector` of types. This implicitly defines the number of columns, but one must still explicitly specify the number of rows:

```
df = DataFrame({Int64, Float64}, 4)
```

When you know what the names of the columns will be, but not the values, it is possible to specify the column names at the time of construction.

SHOULD THIS BE `DataFrame(types, nrow, names)` INSTEAD?

```
DataFrame({Int64, Float64}, ["A", "B"], 10)
DataFrame({Int64, Float64}, Index(["A", "B"]), 10) # STILL NEED TO MAKE THIS WORK
```

A more uniquely Julian way of creating `DataFrame` exploits Julia's ability to quote `Expression` in order to produce behavior like R's delayed evaluation strategy.

```
df = DataFrame(quote
                 A = rand(5)
                 B = datatrues(5)
               end)
```

Accessing and Assigning Elements of DataVector's and DataFrame's

Because a `DataVector` is a 1-dimensional Array, indexing into it is trivial and behaves exactly like indexing into a standard Julia vector.

```
dv = dataones(5)
dv[1]
dv[5]
dv[end]
dv[1:3]
dv[[true, true, false, false, false]]

dv[1] = 3
dv[5] = 5.3
dv[end] = 2.1
dv[1:3] = [3.2, 3.2, 3.1]
dv[[true, true, false, false, false]] = dataones(2) # SHOULD WE MAKE THIS WORK?
```

In contrast, a DataFrame is a random-access data structure that can be indexed into and assigned to in many different ways. We walk through many of them below.

Simple Numeric Indexing

Index by numbers:

```
df = DataFrame(Int64, 5, 3)
df[1, 3]
df[1]
```

Range-Based Numeric Indexing

Index by ranges:

```
df = DataFrame(Int64, 5, 3)

df[1, :]
df[:, 3]
df[1:2, 3]
df[1, 1:3]
df[:, :]
```

Column Name Indexing

Index by column names:

```
df["x1"]
df[1, "x1"]
df[1:3, "x1"]

df[["x1", "x2"]]
df[1, ["x1", "x2"]]
df[1:3, ["x1", "x2"]]
```

Unary Operators for NA, DataVector's and DataFrame's

In practice, we want to compute with these new types. The first requirement is to define the basic unary operators:

- `+`
- `-`

- `!`
- *MISSING: The transpose unary operator*

You can see these operators in action below:

```
+NA
-NA
!NA

+dataones(5)
-dataones(5)
!datafalses(5)
```

Binary Operators

- Arithmetic Operators:
  - Scalar Arithmetic: `+`, `-`, `*`, `/`, `^`,
  - Array Arithmetic: `+`, `.+`, `-`, `.-`, `.*`, `./`, `.^`
- Bit Operators: `&`, `|`, `$`
- Comparison Operators:
  - Scalar Comparisons: `==`, `!=`, `<`, `<=`, `>`, `>=`
  - Array Comparisons: `.==`, `.!=`, `.<`, `.<=`, `.>`, `.>=`

The standard arithmetic operators work on DataVector's when they interact with Number's, NA's or other DataVector's.

```
dv = dataones(5)
dv[1] = NA
df = DataFrame(quote
                  a = 1:5
                end)
```

NA's with NA's

```
NA + NA
NA .+ NA
```

And so on for `-`, `.-`, `*`, `.*`, `/`, `./`, `^`, `.^`.

NA's with Scalars and Scalars with NA's

```
1 + NA
1 .+ NA
NA + 1
NA .+ 1
```

And so on for `-`, `.-`, `*`, `.*`, `/`, `./`, `^`, `.^`.

NA's with DataVector's

```
dv + NA
dv .+ NA
NA + dv
NA .+ dv
```

And so on for `-`, `.-`, `*`, `.*`, `/`, `./`, `^`, `.^`.

DataVector's with Scalars

```
dv + 1
dv .+ 1
```

And so on for `-`, `.-`, `.*`, `./`, `.^`.

Scalars with DataVector's

```
1 + dv
1 .+ dv
```

And so on for `-`, `.-`, `*`, `.*`, `/`, `./`, `^`, `.^`.

HOW MUCH SHOULD WE HAVE OPERATIONS W/ DATAFRAMES?

```
NA + df
df + NA
1 + df
df + 1
dv + df # SHOULD THIS EXIST?
df + dv # SHOULD THIS EXIST?
df + df
```

And so on for `-`, `.-`, `.*`, `./`, `.^`.

The standard bit operators work on `DataVector`:

TO BE FILLED IN

The standard comparison operators work on `DataVector`:

```
NA .< NA
NA .< "a"
NA .< 1
NA .== dv

dv .< NA
dv .< "a"
dv .< 1
dv .== dv

df .< NA
df .< "a"
df .< 1
df .== dv # SHOULD THIS EXIST?
df .== df
```

Elementwise Functions

- `abs`
- `sign`
- `acos`
- `acosh`
- `asin`
- `asinh`
- `atan`
- `atan2`
- `atanh`
- `sin`
- `sinh`
- `cos`
- `cosh`

- `tan`
- `tanh`
- `ceil`
- `floor`
- `round`
- `trunc`
- `signif`
- `exp`
- `log`
- `log10`
- `log1p`
- `log2`
- `exponent`
- `sqrt`

Standard functions that apply to scalar values of type `Number` return `NA` when applied to `NA`:

```
abs(NA)
```

Standard functions are broadcast to the elements of `DataVector` and `DataFrame` for elementwise application:

```
dv = dataones(5)
df = DataFrame({dv})

abs(dv)
abs(df)
```

Pairwise Functions

- `diff`

Functions that operate on pairs of entries of a `Vector` work on `DataVector` and insert `NA` where it would be produced by other operator rules:

```
diff(dv)
```

Cumulative Functions

- `cumprod`
- `cumsum`
- `cumsum_kbn`
- MISSING: `cummin`
- MISSING: `cummax`

Functions that operate cumulatively on the entries of a `Vector` work on `DataVector` and insert `NA` where it would be produced by other operator rules:

```
cumprod(dv)
cumsum(dv)
cumsum_kbn(dv)
```

Aggregative Functions

- `min`
- `max`
- `prod`
- `sum`
- `mean`
- `median`
- `std`
- `var`
- `fft`
- `norm`

You can see these in action:

```
min(dv)
```

To broadcast these to individual columns, use the `col*s` versions:

- `colmins`

- `colmaxs`
- `colprods`
- `colsums`
- `colmeans`
- `colmedians`
- `colstds`
- `colvars`
- `colffts`
- `colnorms`

You can see these in action:

```
colmins(df)
```

Loading Standard Data Sets

The DataFrames package is easiest to explore if you also install the RDatasets package, which provides access to 570 classic data sets:

```
require("RDatasets")

iris = RDatasets.data("datasets", "iris")
dia = RDatasets.data("ggplot2", "diamonds")
```

Split-Apply-Combine

The basic mechanism for spliting data is the `groupby()` function, which will produce a `GroupedDataFrame` object that is easiest to interact with by iterating over its entries:

```
for df in groupby(iris, "Species")
    println("A DataFrame with $(nrow(df)) rows")
end
```

The `|>` (pipe) operator for `GroupedDataFrame` allows you to run simple functions on the columns of the induced `DataFrame`. You pass a simple function by producing a symbol with its name:

```
groupby(iris, "Species") |> :mean
```

Another simple way to split-and-apply (without clear combining) is to use the `map()` function:

```
map(df -> mean(df[1]), groupby(iris, "Species"))
```

Reshaping

If you are looking for the equivalent of the R "Reshape" packages `melt()` and `cast()` functions, you can use `stack()` and `unstack()`. Note that these functions have exactly the oppposite syntax as `melt()` and `cast()`:

```
stack(iris, ["Petal.Length", "Petal.Width"])
```

Model Formulas

Design

Once support for missing data and tabular data structures are in place, we need to begin to develop a version of the model formulas "syntax" used by R. In reality, it is better to regard this "syntax" as a complete domain-specific language (DSL) for describing linear models. For those unfamilar with this DSL, we show some examples below and then elaborate upon them to demonstrate ways in which Julia might move beyond R's formula system.

Let's consider the simplest sort of linear regression model: how does the height of a child depend upon the height of the child's mother and father? If we let the variable `C` denote the height of the child, `M` the height of the mother and `F` the height of the father, the standard linear model approach in statistics would try to model their relationship using the following equation: `C = a + bM + cF + epsilon`, where `a`, `b` and `c` are fixed constants and `epsilon` is a normally distributed noise term that accounts for the imperfect match between any specific child's height and the predictions based solely on the heights of that child's mother and father.

In practice, we would fit such a model using a function that performs linear regression for us based on information about the model and the data source. For example, in R we would write `lm(C ~ M + F, data = heights.data)` to fit this model, assuming that `heights.data` refers to a tabular data structure containing the heights of the children, mothers and fathers for which we have data.

If we wanted to see how the child's height depends only on the mother's height, we would write `lm(C ~ M)`. If we were concerned only about dependence on the father's height, we would write `lm(C ~ H)`. As you can see, we can perform many different statistical analyses using a very consise language for describing those analyses.

What is that language? The R formula language allows one to specify linear models by specifying the terms that should be included. The language is defined by a very small number of constructs:

- The `~` operator: The `~` operator separates the pieces of a Formula. For linear models, this means that one specifies the outputs to be predicted on the left-hand side of the `~` and the inputs to be used to make predictions on the right-hand side.
- The `+` operator: If you wish to include multiple predictors in a linear model, you use the `+` operator. To include both the columns `A` and `B` while predicting `C`, you write: `C ~ A + B`.
- The `&` operator: The `&` operator is equivalent to `:` in R. It computes interaction terms, which are really an entirely new column created by combining two existing columns. For example, `C ~ A&B` describes a linear model with only one predictor. The values of this predictor at row `i` is exactly `A[i] * B[i]`, where `*` is the standard arithmetic multiplication operation. Because of the precedence rules for Julia, it was not possible to use a `:` operator without writing a custom parser.
- The `*` operator: The `*` operator is really shorthand because `C ~ A*B` expands to `C ~ A + B + A:B`. In other words, in a DSL with only three operators, the `*` is just syntactic sugar.

In addition to these operators, the model formulas DSL typically allows us to include simple functions of single columns such as in the example, `C ~ A + log(B)`.

For Julia, this DSL will be handled by constructing an object of type `Formula`. It will be possible to generate a `Formula` using explicitly quoted expression. For example, we might write the Julian equivalent of the models above as `lm(:(C ~ M + F), heights_data)`. A `Formula` object describes how one should convert the columns of a `DataFrame` into a `ModelMatrix`, which fully specifies a linear model. *MORE DETAILS NEEDED ABOUT HOW `ModelMatrix` WORKS.*

How can Julia move beyond R? The primary improvement Julia can offer over R's model formula approach involves the use of hierarchical indexing of columns to control the inclusion of groups of columns as predictors. For example, a text regression model that uses word counts for thousands of different words as columns in a `DataFrame` might involve writing `IsSpam ~ Pronouns + Prepositions + Verbs` to exclude most words from the analysis except for those included in the `Pronouns`, `Prepositions` and `Verbs` groups. In addition, we might try to improve upon some of the tricks R provides for writing hierarchical models in which each value of a categorical predictor gets its own coefficients. This occurs, for example, in hierarchical regression models of the sort implemented by R's `lmer` function. In addition, there are plans to support multiple LHS and RHS components of a `Formula` using a `|` operator.

Implementation

DETAILS NEEDED

Factors

Design

As noted above, statistical data often involves that are not quantitative, but qualitative. Such variables are typically called categorical variables and can take on only a finite number of different values. For example, a data set about people might contain demographic information such as gender or nationality for which we can know the entire set of possible values in advance. Both gender and nationality are categorical variables and should not be represented using quantitative codes unless required as this is confusing to the user and mathematically suspect since the numbering used is entirely artificial.

In general, we can require that a `Factor` type allow us to express variables that can take on a known, finite list of values. This finite list is called the levels of a `Factor`. In this sense, a `Factor` is like an enumeration type.

What makes a `Factor` more specialized than an enumeration type is that modeling tools can interpret factors using indicator variables. This is very important for specifying regression models. For example, if we run a regression in which the right-hand side includes a gender `Factor`, the regression function can replace this factor with two dummy variable columns that encode the levels of this factor. (In practice, there are additional complications because of issues of identifiability or collinearity, but we ignore those for the time being and address them in the Implementation section.)

In addition to the general `Factor` type, we might also introduce a subtype of the `Factor` type that encodes ordinal variables, which are categorical variables that encode a definite ordering such as the values, "very unhappy", "unhappy", "indifferent", "happy" and "very happy". By introducing an `OrdinalFactor` type in which the levels of this sort of ordinal factor are represented in their proper ordering, we can provide specialized functionality like ordinal logistic regression that go beyond what is possible with `Factor` types alone.

Implementation

We have a `Factor` type that handles `NA`s. This type is currently implemented using `PooledDataVector`.

DataStreams

Specification of DataStream as an Abstract Protocol

A `DataStream` object allows one to abstractly write code that processes streaming data, which can be used for many things:

- Analysis of massive data sets that cannot fit in memory
- Online analysis in which interim answers are required while an analysis is still underway

Before we begin to discuss the use of `DataStream` in Julia, we need to distinguish between streaming data and online analysis:

- Streaming data involves low memory usage access to a data source. Typically, one demands that a streaming data algorithm use much less memory than would be required to simply represent the full raw data source in main memory.
- Online analysis involves computations on data for which interim answers must be available. For example, given a list of a trillion numbers, one would like to have access to the estimated mean after seeing only the first N elements of this list. Online estimation is essential for building practical statistical systems that will be deployed in the wild. Online analysis is the *sine qua non* of active learning, in which a statistical system selects which data points it will observe next.

In Julia, a `DataStream` is really an abstract protocol implemented by all subtypes of the abstract type, `AbstractDataStream`. This protocol assumes the following:

- A `DataStream` provides a connection to an immutable source of data that implements the standard iterator protocol use throughout Julia:
    - `start(iter)`: Get initial iteration state.
    - `next(iter, state)`: For a given iterable object and iteration state, return the current item and the next iteration state.
    - `done(iter, state)`: Test whether we are done iterating.

- Each call to `next()` causes the `DataStream` object to read in a chunk of rows of tabular data from the streaming source and store these in a `DataFrame`. This chunk of data is called a minibatch and its maximum size is specified at the time the DataStream is created. It defaults to *1* if no size is explicitly specified.
- All rows from the data source must use the same tabular schema. Entries may be missing, but this missingness must be represented explicitly by the `DataStream` using `NA` values.

Ultimately, we hope to implement a variety of `DataStream` types that wrap access to many different data sources like CSV files and SQL databases. At present, have only implemented the `FileDataStream` type, which wraps access to a delimited file. In the future, we hope to implement:

- MatrixDataStream
- DataFrameDataStream
- SQLDataStream
- Other tabular data sources like Fixed Width Files

Thankfully the abstact `DataStream` protocol allows one to specify algorithms without regard for the specific type of `DataStream` being used. NB: *NoSQL databases are likely to be difficult to support because of their flexible schemas. We will need to think about how to interface with such systems in the future.*

Constructing DataStreams

The easiest way to construct a `DataStream` is to specify a filename:

```
ds = DataStream("my_data_set.csv")
```

You can then iterate over this `DataStream` to see how things work:

```
for df in ds
    print(ds)
end
```

Use Cases for DataStreams:

We can compute many useful quantities using `DataStream`:

- *Means*: `colmeans(ds)`
- *Variances*: `colvars(ds)`
- *Covariances*: `cov(ds)`
- *Correlations*: `cor(ds)`
- *Unique element lists and counts*: *MISSING*
- *Linear models*: *MISSING*
- *Entropy*: *MISSING*

Advice on Deploying DataStreams

- Many useful computations in statistics can be done online:
- Estimation of means, including implicit estimation of means in Reinforcement Learning
- Estimation of entropy
- Estimation of linear regression models
- But many other computations cannot be done online because they require completing a full pass through the data before quantities can be computed exactly.
- Before writing a DataStream algorith, ask yourself: "what is the performance of this algorithm if I only allow it to make one pass through the data?"

References

- McGregor: Crash Course on Data Stream Algorithms
- Muthukrishnan : Data Streams - Algorithms and Applications
- Chakrabarti: CS85 - Data Stream Algorithms
- Knuth: Art of Computer Programming

Ongoing Debates about NA's

- What are the proper rules for the propagation of missingness? It is clear that there is no simple absolute rule we can follow, but we need to formulate some general principles for how to set reasonable defaults. R's strategy seems to be:
  - For operations on vectors, `NA` values are absolutely poisonous by default.
  - For operations on `data.frames`, `NA` values are absolutely poisonous on a column-by-column basis by default. This stems from a more general which assumes that most operations on `data.frame` reduce to the aggregation of the same operation performed on each column independently.
  - Every function should provide an `na.rm` option that allows one to ignore `NA` values. Essentially this involves replacing `NA` by the identity element for that function: `sum(na.rm = TRUE)` replaces `NA` values with `0`, while `prod(na.rm = TRUE)` replaces `NA` values with `1`.
- Should there be multiple types of missingness?
  - For example, SAS distinguishes between:
    - Numeric missing values
    - Character missing values
    - Special numeric missing values
  - In statistical theory, while the *fact* of missingness is simple and does not involve multiple types of `NA` values, the *cause* of missingness can be different for different data sets, which leads to very different procedures that can appropriately be used. See, for example, the different suggestions in Little and Rubin (2002) about how to treat data that has entries missing completely at random (MCAR) vs. data that has entries missing at random (MAR). Should we be providing tools for handling this? External data sources will almost never provide this information, but multiple dispatch means that Julian statistical functions could insure that the appropriate computations are performed for properly typed data sets without the end-user ever understanding the process that goes on under the hood.
- How is missingness different from `NaN` for `Float`? Both share poisonous behavior and `NaN` propagation is very efficient in modern computers. This can provide a clever method for making `NA` fast for `Float`, but does not apply to other types and seems potentially problematic as two different concepts are now aliased. For example, we are not uncertain about the value of `0/0` and should not allow any method to impute a value for it – which any imputation method will do if we treat every `NaN` as equivalent to a `NA`.

- Should cleverness ever be allowed in propagation of `NA`? In section 3.3.4 of the R Language Definition, they note that in cases where the result of an operation would be the same for all possible values that an `NA` value could take on, the operation may return this constant value rather than return `NA`. For example, `FALSE & NA` returns `FALSE` while `TRUE | NA` returns `TRUE`. This sort of cleverness seems like a can-of-worms.

Ongoing Debates about DataFrame's

- How should RDBMS-like indices be implemented? What is most efficient? How can we avoid the inefficient vector searches that R uses?
- How should `DataFrame` be distributed for parallel processing?

Formal Specification of DataFrames Data Structures

- Type Definitions and Type Hierarchy
- Constructors
- Indexing (Refs / Assigns)
- Operators
  - Unary Operators:
    - `+`, `-`, `!`, `'`
  - Elementary Unary Functions
    - `abs`, ...
  - Binary Operators:
    - Arithmetic Operators:
      - Scalar Arithmetic: `+`, `-`, `*`, `/`, `^`,
      - Array Arithmetic: `+`, `.+`, `-`, `.-`, `.*`, `./`, `.^`
    - Bit Operators: `&`, `|`, `$`
    - Comparison Operators:
      - Scalar Comparisons: `==`, `!=`, `<`, `<=`, `>`, `>=`
      - Array Comparisons: `.==`, `.!=`, `.<`, `.<=`, `.>`, `.>=`
- Container Operations
- Broadcasting / Recycling
- Type Promotion and Conversion
- String Representations
- IO
- Copying
- Properties
  - size
  - length
  - ndims
  - eltype

- Predicates
- Handling NA's
- Iteration
- Miscellaneous

The NAtype

Behavior under Unary Operators

The unary operators

Behavior under Unary Operators

The unary operators

Behavior under Arithmetic Operators

Constructors

- NA's
  - Constructor: `NAtype()`
  - Const alias: `NA`
- DataVector's
  - From (Vector, BitVector): `DataArray([1, 2, 3], falses(3))`
  - From (Vector, Vector{Bool}): `DataArray([1, 2, 3], [false, false, false])`
  - From (Vector): `DataArray([1, 2, 3])`
  - From (BitVector, BitVector): `DataArray(trues(3), falses(3))`
  - From (BitVector): `DataArray(trues(3))`
  - From (Range1): `DataArray(1:3)`
  - From (DataVector): `DataArray(DataArray([1, 2, 3]))`
  - From (Type, Int): `DataArray(Int64, 3)`
  - From (Int): `DataArray(3)` (Type defaults to Float64)
  - From (): `DataArray()` (Type defaults to Float64, length defaults to 0)

  - Initialized with Float64 zeros: `datazeros(3)`
  - Initialized with typed zeros: `datazeros(Int64, 3)`
  - Initialized with Float64 ones: `dataones(3)`
  - Initialized with typed ones: `dataones(Int64, 3)`
  - Initialized with falses: `datafalses(3)`
  - Initialized with trues: `datatrues(3)`
  - Literal syntax: `DataVector[1, 2, NA]`

- PooledDataVector's
  - From (Vector, BitVector): `PooledDataArray([1, 2, 3], falses(3))`
  - From (Vector, Vector{Bool}): `PooledDataArray([1, 2, 3], [false, false, false])`
  - From (Vector): `PooledDataArray([1, 2, 3])`
  - From (BitVector, BitVector): `PooledDataArray(trues(3), falses(3))`
  - From (BitVector, Vector{Bool}): `PooledDataArray(trues(3), [false, false, false])`
  - From (BitVector): `PooledDataArray(trues(3))`
  - From (Range1): `PooledDataArray(1:3)`
  - From (DataVector): `PooledDataArray(DataArray([1, 2, 3]))`
  - From (Type, Int): `PooledDataArray(Int64, 3)`
  - From (Int): `PooledDataArray(3)` (Type defaults to Float64)
  - From (): `PooledDataArray()` (Type defaults to Float64, length defaults to 0)
  - Initialized with Float64 zeros: `pdatazeros(3)`
  - Initialized with typed zeros: `pdatazeros(Int64, 3)`
  - Initialized with Float64 ones: `pdataones(3)`
  - Initialized with typed ones: `pdataones(Int64, 3)`
  - Initialized with falses: `pdatafalses(3)`
  - Initialized with trues: `pdatatrues(3)`
  - Literal syntax: `PooledDataVector[1, 2, NA]`

- DataMatrix
  - From (Array, BitArray): `DataMatrix([1 2; 3 4], falses(2, 2))`
  - From (Array, Array{Bool}): `DataMatrix([1 2; 3 4], [false false; false false])`

  - From (Array): `DataMatrix([1 2; 3 4])`
  - From (BitArray, BitArray): `DataMatrix(trues(2, 2), falses(2, 2))`
  - From (BitArray): `DataMatrix(trues(2, 2))`
  - From (DataVector...): `DataMatrix(DataVector[1, NA], DataVector[NA, 2])`
  - From (Range1...): `DataMatrix(1:3, 1:3)`
  - From (DataMatrix): `DataMatrix(DataArray([1 2; 3 4]))`
  - From (Type, Int, Int): `DataMatrix(Int64, 2, 2)`
  - From (Int, Int): `DataMatrix(2, 2)` (Type defaults to Float64)
  - From (): `DataMatrix()` (Type defaults to Float64, length defaults to (0, 0))
  - Initialized with Float64 zeros: `dmzeros(2, 2)`
  - Initialized with typed zeros: `dmzeros(Int64, 2, 2)`
  - Initialized with Float64 ones: `dmones(2, 2)`
  - Initialized with typed ones: `dmones(Int64, 2, 2)`
  - Initialized with falses: `dmfalses(2, 2)`
  - Initialized with trues: `dmtrues(2, 2)`
  - Initialized identity matrix: `dmeye(2, 2)`
  - Initialized identity matrix: `dmeye(2)`
  - Initialized diagonal matrix: `dmdiagm([2, 1])`
  - Literal syntax: `DataMatrix[1 2; NA 2]`

- DataFrame
  - From (): `DataFrame()`
  - From (Vector{Any}, Index): `DataFrame({datazeros(3), dataones(3)}, Index(["A", "B"]))`
  - From (Vector{Any}): `DataFrame({datazeros(3), dataones(3)})`
  - From (Expr): `DataFrame(quote A = [1, 2, 3, 4] end)`
  - From (Matrix, Vector{String}): `DataFrame([1 2; 3 4], ["A", "B"])`
  - From (Matrix): `DataFrame([1 2; 3 4])`
  - From (Tuple): `DataFrame(dataones(2), datafalses(2))`
  - From (Associative): ???
  - From (Vector, Vector, Groupings): ???
  - From (Dict of Vectors): `DataFrame({"A" => [1, 3], "B" => [2, 4]})`

- From (Dict of Vectors, Vector{String}): `DataFrame({"A" => [1, 3], "B" => [2, 4]}, ["A"])`
- From (Type, Int, Int): `DataFrame(Int64, 2, 2)`
- From (Int, Int): `DataFrame(2, 2)`
- From (Vector{Types}, Vector{String}, Int): `DataFrame({Int64, Float64}, ["A", "B"], 2)`
- From (Vector{Types}, Int): `DataFrame({Int64, Float64}, 2)`

Indexing

Types on indices:

```
NA
```

```
dv = datazeros(10)
```

```
dv[1]
```

```
dv[1:2]
```

```
dv[:]
```

```
dv[[1, 2 3]]
```

```
dv[[false, false, true, false, false]]
```

```
dmzeros(10)
```

Indexers: Int, Range, Colon, Vector{Int}, Vector{Bool}, String, Vector{String}

DataVector's and PooledDataVector's implement:

- Int
- Range
- Colon
- Vector{Int}
- Vector{Bool}

DataMatrix's implement the Cartesian product:

- Int, Int
- Int, Range
- Int, Colon
- Int, Vector{Int}
- Int, Vector{Bool} ...
- Vector{Bool}, Int
- Vector{Bool}, Range
- Vector{Bool}, Colon
- Vector{Bool}, Vector{Int}
- Vector{Bool}, Vector{Bool}

Single Int access?

DataFrame's add two new indexer types:

- String
- Vector{String}

These can only occur as (a) the only indexer or (b) in the second slot of a paired indexer

Anything that can be getindex()'d can also be setindex!()'d

Where do we allow Expr indexing?

Function Reference Guide

DataFrames

`DataFrame(cols::Vector, colnames::Vector{ByteString})` Construct a DataFrame from the columns given by `cols` with the index generated by `colnames`. A DataFrame inherits from `Associative{Any,Any}`, so Associative operations should work. Columns are vector-like objects. Normally these are AbstractDataVector's (DataVector's or PooledDataVector's), but they can also (currently) include standard Julia Vectors.

`DataFrame(cols::Vector)` Construct a DataFrame from the columns given by `cols` with default column names.

`DataFrame()` An empty DataFrame.

`copy(df::DataFrame)` A shallow copy of `df`. Columns are referenced, not copied.

`deepcopy(df::DataFrame)` A deep copy of `df`. Copies of each column are made.

`similar(df::DataFrame, nrow)` A new DataFrame with `nrow` rows and the same column names and types as `df`.

Basics

`size(df)`, `ndims(df)` Same meanings as for Arrays.

`has(df, key)`, `get(df, key, default)`, `keys(df)`, and `values(df)` Same meanings as Associative operations. `keys` are column names; `values` are column contents.

`start(df)`, `done(df,i)`, and `next(df,i)` Methods to iterate over columns.

`ncol(df::AbstractDataFrame)` Number of columns in `df`.

nrow(df::AbstractDataFrame) Number of rows in `df`.

length(df::AbstractDataFrame) Number of columns in `df`.

isempty(df::AbstractDataFrame) Whether the number of columns equals zero.

head(df::AbstractDataFrame) and head(df::AbstractDataFrame, i::Int) First `i` rows of `df`. Defaults to 6.

tail(df::AbstractDataFrame) and tail(df::AbstractDataFrame, i::Int) Last `i` rows of `df`. Defaults to 6.

show(io, df::AbstractDataFrame) Standard pretty-printer of `df`. Called by `print()` and the REPL.

dump(df::AbstractDataFrame) Show the structure of `df`. Like R's `str`.

describe(df::AbstractDataFrame) Show a description of each column of `df`.

complete_cases(df::AbstractDataFrame) A Vector{Bool} of indexes of complete cases in `df` (rows with no NA's).

duplicated(df::AbstractDataFrame) A Vector{Bool} of indexes indicating rows that are duplicates of prior rows.

unique(df::AbstractDataFrame) DataFrame with unique rows in `df`.

Indexing, Assignment, and Concatenation

DataFrames are indexed like a Matrix and like an Associative. Columns may be indexed by column name. Rows do not have names. Referencing with one argument normally indexes by columns: `df["col"]`, `df[["col1","col3"]]` or `df[i]`. With two arguments, rows and columns are selected. Indexing along rows works like Matrix indexing. Indexing along columns works like Matrix indexing with the addition of column name access.

getindex(df::DataFrame, ind) or df[ind] Returns a subset of the columns of df as specified by ind, which may be an Int, a Range, a Vector{Int}, ByteString, or Vector{ByteString}. Columns are referenced, not copied. For a single-element ind, the column by itself is returned.

getindex(df::DataFrame, irow, icol) or df[irow,icol] Returns a subset of df as specified by irow and icol. irow may be an Int, a Range, or a Vector{Int}. icol may be an Int, a Range, or a Vector{Int}, ByteString, or, ByteString, or Vector{ByteString}. For a single-element ind, the column subset by itself is returned.

index(df::DataFrame) Returns the column Index for df.

set_group(df::DataFrame, newgroup, names::Vector{ByteString})

get_groups(df::DataFrame)

set_groups(df::DataFrame, gr::Dict) See the Indexing section for these operations on column indexes.

colnames(df::DataFrame) or names(df::DataFrame) The column names as an Array{ByteString}

setindex!(df::DataFrame, newcol, colname) or df[colname] = newcol Replace or add a new column with name colname and contents newcol. Arrays are converted to DataVector's. Values are recycled to match the number of rows in df.

insert!(df::DataFrame, index::Integer, item, name) Insert a column of name name and with contents item into df at position index.

insert!(df::DataFrame, df2::DataFrame) Insert columns of df2 into df1.

del!(df::DataFrame, cols) Delete columns in df at positions given by cols (noted with any means that columns can be referenced).

del(df::DataFrame, cols) Nondestructive version. Return a DataFrame based on the columns in df after deleting columns specified by cols.

cbind(df1, df2, ...) or hcat(df1, df2, ...) or [df1 df2 ...] Concatenate columns. Duplicated column names are adjusted.

rbind(df1, df2, ...) or vcat(df1, df2, ...) or [df1, df2, ...] Concatenate rows.

I/O

csvDataFrame(filename, o::Options) Return a DataFrame from file `filename`. Options `o` include `colnames` (`"true"`, `"false"`, or `"check"` (the default)) and `poolstrings` (`"check"` (default) or `"never"`).

Expression/Function Evaluation in a DataFrame

with(df::AbstractDataFrame, ex::Expr) Evaluate expression `ex` with the columns in `df`.

within(df::AbstractDataFrame, ex::Expr) Return a copy of `df` after evaluating expression `ex` with the columns in `df`.

within!(df::AbstractDataFrame, ex::Expr) Modify `df` by evaluating expression `ex` with the columns in `df`.

based_on(df::AbstractDataFrame, ex::Expr) Return a new DataFrame based on evaluating expression `ex` with the columns in `df`. Often used for summarizing operations.

colwise(f::Function, df::AbstractDataFrame)

colwise(f::Vector{Function}, df::AbstractDataFrame) Apply `f` to each column of `df`, and return the results as an Array{Any}.

colwise(df::AbstractDataFrame, s::Symbol)

colwise(df::AbstractDataFrame, s::Vector{Symbol}) Apply the function specified by Symbol `s` to each column of `df`, and return the results as a DataFrame.

SubDataFrames

`sub(df::DataFrame, r, c)`

`sub(df::DataFrame, r)` Return a SubDataFrame with references to rows and columns of `df`.

`sub(sd::SubDataFrame, r, c)`

`sub(sd::SubDataFrame, r)` Return a SubDataFrame with references to rows and columns of `df`.

`getindex(sd::SubDataFrame, r, c)` or `sd[r,c]`

`getindex(sd::SubDataFrame, c)` or `sd[c]` Referencing should work the same as DataFrames.

Grouping

`groupby(df::AbstractDataFrame, cols)` Return a GroupedDataFrame based on unique groupings indicated by the columns with one or more names given in `cols`.

`start(gd)`, `done(gd,i)`, and `next(gd,i)` Methods to iterate over GroupedDataFrame groupings.

`getindex(gd::GroupedDataFrame, idx)` or `gd[idx]` Reference a particular grouping. Referencing returns a SubDataFrame.

`with(gd::GroupedDataFrame, ex::Expr)` Evaluate expression `ex` with the columns in `gd` in each grouping.

`within(gd::GroupedDataFrame, ex::Expr)`

`within!(gd::GroupedDataFrame, ex::Expr)` Return a DataFrame with the results of evaluating expression `ex` with the columns in `gd` in each grouping.

`based_on(gd::GroupedDataFrame, ex::Expr)` Sweeps along groups and applies `based_on` to each group. Returns a DataFrame.

`map(f::Function, gd::GroupedDataFrame)` Apply `f` to each grouping of `gd` and return the results in an Array.

`colwise(f::Function, gd::GroupedDataFrame)`

`colwise(f::Vector{Function}, gd::GroupedDataFrame)` Apply `f` to each column in each grouping of `gd`, and return the results as an Array{Any}.

`colwise(gd::GroupedDataFrame, s::Symbol)`

`colwise(gd::GroupedDataFrame, s::Vector{Symbol})` Apply the function specified by Symbol `s` to each column of in each grouping of `gd`, and return the results as a DataFrame.

`by(df::AbstractDataFrame, cols, s::Symbol)` or `groupby(df, cols) |> s`

`by(df::AbstractDataFrame, cols, s::Vector{Symbol})` Return a DataFrame with the results of grouping on `cols` and `colwise` evaluation based on `s`. Equivalent to `colwise(groupby(df, cols), s)`.

`by(df::AbstractDataFrame, cols, e::Expr)` or `groupby(df, cols) |> e` Return a DataFrame with the results of grouping on `cols` and evaluation of `e` in each grouping. Equivalent to `based_on(groupby(df, cols), e)`.

Reshaping / Merge

`stack(df::DataFrame, cols)` For conversion from wide to long format. Returns a DataFrame with stacked columns indicated by `cols`. The result has column `"key"` with column names from `df` and column `"value"` with the values from `df`. Columns in `df` not included in `cols` are duplicated along the stack.

`unstack(df::DataFrame, ikey, ivalue, irefkey)` For conversion from long to wide format. Returns a DataFrame. `ikey` indicates the key column–unique values in column `ikey` will be column names in the result. `ivalue` indicates the value column. `irefkey` is the column with a unique identifier for that . Columns not given by `ikey`, `ivalue`, or `irefkey` are currently ignored.

`merge(df1::DataFrame, df2::DataFrame, bycol)`

`merge(df1::DataFrame, df2::DataFrame, bycol, jointype)` Return the database join of `df1` and `df2` based on the column `bycol`. Currently only a single merge key is supported. Supports `jointype` of "inner" (the default), "left", "right", or "outer".

Index

`Index()`

`Index(s::Vector{ByteString})` An Index with names `s`. An Index is like an Associative type. An Index is used for column indexing of DataFrames. An Index maps ByteStrings and Vector{ByteStrings} to Indices.

`length(x::Index)`, **`copy(x::Index)`**, **`has(x::Index, key)`**, **`keys(x::Index)`**, **`push!(x::Index, name)`** Normal meanings.

`del(x::Index, idx::Integer)`, **`del(x::Index, s::ByteString)`**, Delete the name `s` or name at position `idx` in `x`.

`names(x::Index)` A Vector{ByteString} with the names of `x`.

`names!(x::Index, nm::Vector{ByteString})` Set names `nm` in `x`.

`rename(x::Index, f::Function)`

`rename(x::Index, nd::Associative)`

`rename(x::Index, from::Vector, to::Vector)` Replace names in `x`, by applying function `f` to each name, by mapping old to new names with a dictionary (Associative), or using `from` and `to` vectors.

`getindex(x::Index, idx)` or `x[idx]` This does the mapping from name(s) to Indices (positions). `idx` may be ByteString, Vector{ByteString}, Int, Vector{Int}, Range{Int}, Vector{Bool}, AbstractDataVector{Bool}, or AbstractDataVector{Int}.

`set_group(idx::Index, newgroup, names::Vector{ByteString})` Add a group to `idx` with name `newgroup` that includes the names in the vector `names`.

`get_groups(idx::Index)` A Dict that maps the name of each group to the names in the group.

`set_groups(idx::Index, gr::Dict)` Set groups in `idx` based on the mapping given by `gr`.

Missing Values

Missing value behavior is implemented by instantiations of the `AbstractDataVector` abstract type.

`NA` A constant indicating a missing value.

`isna(x)` Return a `Bool` or `Array{Bool}` (if `x` is an `AbstractDataVector`) that is `true` for elements with missing values.

`nafilter(x)` Return a copy of `x` after removing missing values.

`nareplace(x, val)` Return a copy of `x` after replacing missing values with `val`.

`naFilter(x)` Return an object based on `x` such that future operations like `mean` will not include missing values. This can be an iterator or other object.

`naReplace(x, val)` Return an object based on `x` such that future operations like `mean` will replace NAs with `val`.

`na(x)` Return an `NA` value appropriate for the type of `x`.

`nas(x, dim)` Return an object like `x` filled with `NA` values with size `dim`.

DataVector's

`DataArray(x::Vector)`

`DataArray(x::Vector, m::Vector{Bool})` Create a DataVector from `x`, with `m` optionally indicating which values are NA. DataVector's are like Julia Vectors with support for NA's. `x` may be any type of Vector.

`PooledDataArray(x::Vector)`

`PooledDataArray(x::Vector, m::Vector{Bool})` Create a PooledDataVector from `x`, with `m` optionally indicating which values are NA. PooledDataVector's contain a pool of values with references to those values. This is useful in a similar manner to an R array of factors.

`size`, `length`, `ndims`, `ref`, `assign`, `start`, `next`, `done` All normal Vector operations including array referencing should work.

`isna(x), nafilter(x), nareplace(x, val), naFilter(x), naReplace(x, val)` All NA-related methods are supported.

Utilities

`cut(x::Vector, breaks::Vector)` Returns a PooledDataVector with length equal to `x` that divides values in `x` based on the divisions given by `breaks`.

Formulas and Models

`Formula(ex::Expr)` Return a Formula object based on `ex`. Formulas are two-sided expressions separated by `~`, like `:(y ~ w*x + z + i&v)`.

`model_frame(f::Formula, d::AbstractDataFrame)`

`model_frame(ex::Expr, d::AbstractDataFrame)` A ModelFrame.

`model_matrix(mf::ModelFrame)`

`model_matrix(f::Formula, d::AbstractDataFrame)`

`model_matrix(ex::Expr, d::AbstractDataFrame)` A ModelMatrix based on `mf`, `f` and `d`, or `ex` and `d`.

`lm(ex::Expr, df::AbstractDataFrame)` Linear model results (type OLSResults) based on formula `ex` and `df`.

Merging Data Sets Together

Often we have several related data sets that we need to merge together. For example, we might have data about the flowers from Fisher's iris data set:

```
require("DataFrames")
using DataFrames

require("RDatasets")
using RDatasets

iris = data("datasets", "iris")
```

This data set describes individual flowers from three species, but we might want to incorporate generic knowledge about the typical properties of those species into our analysis. Suppose that we have another data set called `flowers` like that defined below:

```
flowers = DataFrame()
flowers["Species"] = ["virginica", "versicolor", "setosa"]
flowers["PrimaryColor"] = ["purplish", "purple", "purple"]
```

How could we merge in the information about primary colors from the `flowers` data set into the `iris` data set?

In Julia, we use a function called `merge` that is inspired by techniques for joining together different database tables. The simplest example of merge is:

```
iris_with_colors = merge(iris, flowers)
```

When called on two data sets, `merge(A, B)` tries to identify a commonly named column that will guide the process of matching rows from `A` with rows from `B`. In this example, that column is the `Species` column. We can help `merge` out by naming this column explicitly:

```
iris_with_colors = merge(iris, flowers, "Species")
```

In this example, it is clear which rows from `iris` should be associated with which rows from `flowers`. But what if `flowers` mentioned a fourth species of flower not found in the `iris` data set? For example, imagine that we added information about daisies to `flowers`:

```
flowers = DataFrame()
flowers["Species"] = ["virginica", "versicolor", "setosa", "daisy"]
flowers["PrimaryColor"] = ["purplish", "purple", "purple", "yellow"]
```

What will happen now? We can see by calling `merge` again:

```
merge(iris, flowers, "Species")
```

If you inspect the results, you'll see that nothing has changed. This is because `merge` defaults to a style of merging called an "inner join" which looks at the values of "Species" in both `iris` and `flowers` and only uses the values found in both data sets. We can insure this behavior by explicitly specifying that we want an "inner" join using a fourth argument to `merge`:

```
merge(iris, flowers, "Species", "inner")
```

What other types of merging operations are there? In total, there are four:

- *Inner join*: Use the values of the Species column that are found in both the `iris` and `flowers` data sets.
- *Left join*: Use only the values of the Species column that are found in the `iris` data set.
- *Right join*: Use only the values of the Species column that are found in the `flowers` data set.
- *Outer join*: Use the values of the Species column that are found in either the `iris` or `flowers` data set.

In our current example, it isn't easy to tell these apart. To make it more clear, we'll use a different data set in which `flowers` is missing data about the "setosa" species, but also has unneeded data about the irrelevant "daisy" species:

```
flowers = DataFrame()
flowers["Species"] = ["virginica", "versicolor", "daisy"]
flowers["PrimaryColor"] = ["purplish", "purple", "yellow"]
```

In that case, we get quite different results from the four types of joins:

```
merge(iris, flowers, "Species", "inner")
merge(iris, flowers, "Species", "left")
merge(iris, flowers, "Species", "right")
merge(iris, flowers, "Species", "outer")
```

As you'll see, the inner join produces 100 rows and contains no information about the "setosa" Species because that species was not found in the `flowers` data set. The left join contains 150 rows, but is missing color information for "setosa" because it wasn't present in the `flowers` data set. The right join contains 101 rows, including an *almost* completely empty row describing the "daisy" species that doesn't appear in the `iris` data aset. Finally, the outer join contains 151 rows describing all four species, incluing both the "setosa" species from the `iris` data set and the "daisy" species from the `flowers` data set.

Indexing: Making Subsetting and Mergers Faster

One problem with merging large data sets is that the merging process can take a long time to complete. This is because the merging process has to determine which subset of rows from `A` should be combined with which subset of rows from `B`. Selecting subsets in this way is slow in general for most DataFrames because the entries of each column have to be exhaustively examined.

But it is possible to make subset selection much faster if we allow the `DataFrame` to store indexing information that tells the system where to expect certain subsets to be located inside of the `DataFrame`. If you are familar with database systems, this indexing involves either explicit index metadata that is added to a database or an implicit index defined by a "primary key" for the database.

For the `iris` data set, an indexing step would

MORE TO BE FILLED IN HERE

Reshaping and Pivoting Data

```
require("DataFrames")
using DataFrames

require("RDatasets")
using RDatasets

iris = data("datasets", "iris")

stack(iris, "Sepal.Length")
```

The Split-Apply-Combine Strategy

```
require("DataFrames")
using DataFrames

require("RDatasets")
using RDatasets

iris = data("datasets", "iris")

by(iris, "Species", nrow)
by(iris, "Species", df -> mean(df["Petal.Length"]))
by(iris, "Species", :(N = nrow(_DF)))
```

Processing Streaming Data

In modern data analysis settings, we often need to work with streaming data sources. This is particularly important when:

- Data sets are too large to store in RAM
- Data sets are being generated in real time

Julia is well-suited to both. The DataFrames package handles streaming data by construcing a `DataStream`, which is an iterable object that returns `DataFrame` one-by-one in small minibatches. By default, the minibatches are single rows of data, but this can be easily changed. To see how a `DataStream` works, it's easiest to convert an existing `DataFrame` into a `DataStream` using the `DataStream` function:

```
using DataFrames
using RDatasets

iris = data("datasets", "iris")

iris = DataStream(iris)
```

We can then iterate over this stream of data using a standard `for` loop:

```
for minibatch in iris
    print_table(minibatch)
end
```

Streaming Large Scale Data Sets

COMING SOON

Real Time Data Analysis

Another important case in which data must be dealt with using a streaming data type comes up in real-time data analysis, when new data is constantly being generated and an existing analysis needs to be updated as soon as possible.

In Julia, this can be addressed by piping new data into Julia using standard UNIX pipes. To see how to work with data that comes in from a UNIX pipe, copy the following code into a program called streaming.jl:

```
using DataFrames

ds = DataStream(STDIN, 2)

for df in ds
  println("==================  MINIBATCH  ==================")
  print_table(df)
  print("\n\n\n")
end
```

Now call this program from a UNIX terminal with a command like:

```
cat ~/.julia/DataFrames/test/data/bool.csv | julia streaming.jl
```

Once that's done, sit back and watch how minibatches of data come streaming in. Because the reader infers column names and types on the fly, you only need to tell the reader what size the minibatches of data that you want to process should be. You can then write simple for loops to process the incoming data stream. You even do this by typing data into `STDIN`: just type `julia streaming.jl`, then enter a data set line-by-line and hit `CTRL-D`.