Why Use the DataFrames Package?

We believe that Julia is the future of technical computing. Nevertheless, Base Julia is not sufficient for statistical computing. The DataFrames package (and its sibling, DataArrays) extends Base Julia by introducing three basic types needed for statistical computing:

- `NA`: An indicator that a data value is missing
- `DataArray`: An extension to the `Array` type that can contain missing values
- `DataFrame`: A data structure for representing tabular data sets

`NA`: An Indicator for 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. In contrast with languages like Java and R, which provide `NULL` and `NA` values that represent missingness, there is no missing data value in Base Julia.

The DataArrays package therefore provides `NA`, which serves as an indicator that a specific value is missing. In order to exploit Julia's multiple dispatch rules, `NA` is a singleton object of a new type called `NAtype`.

Like R's `NA` value and unlike Java's `NULL` value, Julia's `NA` value represents epistemic uncertainty. This means that operations involving `NA` return `NA` when the result of the operation cannot be determined, but operations whose value can be determined despite the presence of `NA` will return a value that is not `NA`.

For example, `false && NA` evaluates to `false` and `true || NA` evaluates to `true`. In contrast, `1 + NA` evaluates to `NA` because the outcome is uncertain in the absence of knowledge about the missing value represented by `NA`.

`DataArray`: Efficient Arrays with Missing Values

Although the `NA` value is sufficient for representing missing scalar values, it cannot be stored efficiently inside of Julia's standard `Array` type. To represent arrays with potentially missing entries, the DataArrays package introduces a `DataArray` type. For example, a `DataArray{Float64}` can contain `Float64`

values and `NA` values, but nothing else. In contrast, the most specific `Array` that can contain both `Float64` and `NA` values is an `Array{Any}`.

Except for the ability to store `NA` values, the `DataArray` type is meant to behave exactly like Julia's standard `Array` type. In particular, `DataArray` provides two typealiases called `DataVector` and `DataMatrix` that mimic the `Vector` and `Matrix` typealiases for 1D and 2D `Array` types.

`DataFrame`: Tabular Data Sets

`NA` and `DataArray` provide mechanisms for handling missing values for scalar types and arrays, but most real world data sets have a tabular structure that does not correspond to a simple `DataArray`.

For example, the data table shown below highlights some of the ways in which a typical data set is not like a `DataArray`:

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

Figure 1: Tabular Data

Note three important properties that this table possesses:

- 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` type and another column of `Int` type.
- The values of the entries within a column always 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 standard `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.*
- The columns of a tabular data set are typically named using some sort of `String`. Often, one wants to access the entries of a data set by using a combination of verbal names and numeric indices.

We can summarize these concerns by noting that we face four problems when with working with tabular data sets:

- Tabular data sets may have columns of heterogeneous type
- Each column of a tabular data set has a consistent type across all of its entries
- All of the columns of a tabular data set have the same length
- The columns of a tabular data set should be addressable using both verbal names and numeric indices

The DataFrames package solves these problems by adding a `DataFrame` type to Julia. This type will be familiar to anyone who has worked with R's `data.frame` type, Pandas' `DataFrame` type, an SQL-style database, or Excel spreadsheet.

Getting Started

Installation

The DataFrames package is available through the Julia package system. Throughout the rest of this tutorial, we will assume that you have installed the DataFrames package and have already typed `using DataArrays, DataFrames` to bring all of the relevant variables into your current namespace. In addition, we will make use of the `RDatasets` package, which provides access to hundreds of classical data sets.

The `NA` Value

To get started, let's examine the `NA` value. Type the following into the REPL:

```
NA
```

One of the essential properties of `NA` is that it poisons other items. To see this, try to add something like `1` to `NA`:

```
1 + NA
```

The `DataArray` Type

Now that we see that `NA` is working, let's insert one into a `DataArray`. We'll create one now using the `@data` macro:

```
dv = @data([NA, 3, 2, 5, 4])
```

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

```
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 `dropna` function:

```
dropna(dv)
mean(dropna(dv))
```

Instead of removing `NA` values, you can try to conver the `DataArray` into a normal Julia `Array` using `convert`:

```
convert(Array, dv)
```

This fails in the presence of `NA` values, but will succeed if there are no `NA` values:

```
dv[1] = 3
convert(Array, dv)
```

In addition to removing `NA` values and hoping they won't occur, you can also replace any `NA` values using the `array` function, which takes a replacement value as an argument:

```
dv = @data([NA, 3, 2, 5, 4])
mean(array(dv, 11))
```

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

Although the examples above employed only 1D `DataArray` objects, the `DataArray` type defines a completely generic N-dimensional array type. Operations on generic `DataArray` objects work in higher dimensions in the same way that they work on Julia's Base `Array` type:

```
dm = @data([NA 0.0; 0.0 1.0])
dm * dm
```

The `DataFrame` Type

The `DataFrame` type can be used to represent data tables, each column of which is a `DataArray`. You can specify the columns using keyword arguments:

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

It is also possible to construct a `DataFrame` in stages:

```
df = DataFrame()
df["A"] = 1:8
df["B"] = ["M", "F", "F", "M", "F", "M", "M", "F"]
df
```

The `DataFrame` we build in this way has 8 rows and 2 columns. You can check this using `size` function:

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

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

```
describe(df)
```

To focus our search, we start looking at just the means and medians of specific columns. In the example below, we use numeric indexing to access the columns of the `DataFrame`:

```
mean(df[1])
median(df[1])
```

We could also have used column names to access individual columns:

```
mean(df["A"])
median(df["A"])
```

We can also apply a function to each column of a `DataFrame` with the `colwise` function. For example:

```
df = DataFrame(A = 1:4, B = randn(4))
colwise(cumsum, df)
```

Accessing Classic Data Sets

To see more of the functionality for working with `DataFrame` objects, we need a more complex data set to work with. We'll use the `RDatasets` package, which provides access to many of the classical data sets that are available in R.

For example, we can access Fisher's iris data set using the following functions:

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

In the next section, we'll discuss generic I/O strategy for reading and writing `DataFrame` objects that you can use to import and export your own data files.

DataFrames I/O

Importing data from tabular data files

To read data from a CSV-like file, use the `readtable` function:

```
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 as a `String`. It supports many additional keyword arguments: these are documented in the section on advanced I/O operations.

Exporting data to a tabular data file

To write data to a CSV file, use the `writetable` function:

```
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.

Additional advanced options are documented below.

Advanced Options for Reading CSV Files

`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::Vector{Char}` – Assume that fields contained inside of two `quotemark` characters are quoted, which disables processing of separators and linebreaks. Set to `Char[]` to disable this feature and slightly improve performance. 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`.

Advanced Options for Writing CSV Files

`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`.

Accessing and Modifying Entries of DataArray and DataFrame Objects

The `DataArray` type is meant to behave like a standard Julia `Array` and tries to implement identical indexing rules:

```
dv = data([1, 2, 3])
dv[1]
dv[2] = NA
dv[2]

dm = data([1 2; 3 4])
dm[1, 1]
dm[2, 1] = NA
dm[2, 1]
```

In contrast, a `DataFrame` offers substantially more forms of indexing because columns can be referred to by name:

```
df = DataFrame(A = 1:10, B = 2:2:20)

df[1]
df["A"]

df[1, 1]
df[1, "A"]

df[1:3, ["A", "B"]]
df[1:3, ["B", "A"]]

df[df["A"] % 2 .== 0, :]
df[df["B"] % 2 .== 0, :]
```

To simplify the last example (in which we examined the properties of column of `df` to determine which rows to return), you can also index rows using quoted expressions that will be evaluated using the columns of the `DataFrame`:

```
df[:(A % 2 .== 0), :]
```

Expression indexing makes it easier to build up complex subsets:

```
df[:((A % 2 .== 0) & (B % 4 .== 0)), :]
```

This kind of indexing can also be accomplished using `select`:

```
select(:((A % 2 .== 0) & (B % 4 .== 0)), df)
```

Database-Style Joins and Indexing

Joining Data Sets Together

We often need to combine two or more data sets together to provide a complete picture of the topic we are studying. For example, suppose that we have the following two data sets:

```
names = DataFrame(ID = [1, 2], Name = ["John Doe", "Jane Doe"])
jobs = DataFrame(ID = [1, 2], Job = ["Lawyer", "Doctor"])
```

We might want to work with a larger data set that contains both the names and jobs for each ID. We can do this using the `join` function:

```
full = join(names, jobs, on = "ID")
```

In relational database theory, this operation is generally referred to as a join. The column used to determine which rows should be combined during a join is called the key. If you do not specify the key when calling `join`, the `join` function will attempt to identity a commonly named column in the DataFrame arguments for use as a key:

```
full = join(names, jobs)
```

There are four sorts of joins supported by the DataFrames package:

- Inner: The output contains rows for values of the key that exist in both the first (left) and second (right) arguments to `join`.
- Left: The output contains rows for values of the key that exist in the first (left) argument to `join`, whether or not that value exists in the second (right) argument.
- Right: The output contains rows for values of the key that exist in the second (right) argument to `join`, whether or not that value exists in the first (left) argument.
- Outer: The output contains rows for values of the key that exist in the first (left) or second (right) argument to `join`.

You can control the kind of join that `join` performs using the `kind` keyword argument:

```
a = DataFrame(ID = [1, 2], Name = ["A", "B"])
b = DataFrame(ID = [1, 3], Job = ["Doctor", "Lawyer"])
join(a, b, on = "ID", kind = :inner)
join(a, b, on = "ID", kind = :left)
join(a, b, on = "ID", kind = :right)
join(a, b, on = "ID", kind = :outer)
```

The Split-Apply-Combine Strategy

Many data analysis tasks involve splitting a data set into groups, applying some functions to each of the groups and then combining the results. A standardized framework for handling this sort of computation is described in the paper, The Split-Apply-Combine Strategy for Data Analysis, written by Hadley Wickham.

The DataFrames package supports the Split-Apply-Combine strategy through the `by` function, which takes in three arguments: (1) a DataFrame, (2) a column to split the DataFrame on, and (3) a function or expression to apply to each subset of the DataFrame.

We show several examples of the `by` function applied to the `iris` dataset below:

```
using DataFrames, RDatasets

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

by(iris, "Species", nrow)
by(iris, "Species", df -> mean(df["PetalLength"]))
by(iris, "Species", :(N = size(_DF, 1)))
```

If you only want to split the data set into subsets, use the `groupby` function:

```
for subdf in groupby(iris, "Species")
    println(size(subdf, 1))
end
```

Reshaping and Pivoting Data

Reshape data using the `stack` function:

```
using DataFrames, RDatasets

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

stack(iris, "SepalLength")
```

Sorting

Sorting is a fundamental component of data analysis. Basic sorting is trivial: just calling `sort!` will sort all columns, in place.

```
using DataFrames, RDatasets

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

In Sorting DataFrames, you may want to sort different columns with different options. Here are some examples showing most of the possible options.

```
sort!(iris, rev = true)

sort!(iris, cols = ["SepalLength", "SepalWidth"])

sort!(iris, cols = [order("Species", by = uppercase),
                    order("SepalLength", rev = true)])
```

Keywords used above include `cols` (to specify columns), `rev` (to sort a column or the whole DataFrame in reverse), and `by` (to apply a function to a column/DataFrame). Each keyword can either be a single value, or can be a tuple or array, with values corresponding to individual columns.

As an alternative to using array or tuple values, `order` to specify an ordering for a particular column within a set of columns

The following two examples show two ways to sort the `iris` dataset with the same result: `Species` will be ordered in reverse lexicographic order, and within species, rows will be sorted by increasing sepal length and width.

```
sort!(iris, cols = ("Species", "SepalLength", "SepalWidth"),
            rev = (true, false, false))

sort!(iris,
      cols = (order("Species", rev = true), "SepalLength", "SepalWidth"))
```

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 an `AbstractDataStream` object, which is an iterable object that allows programmers to work with a sequence of small `DataFrame` objects rather than one large `DataFrame` object.

Conventionally, these small `DataFrame` objects are called minibatches. By default, Julia generates minibatches that contain **exactly** one row of data, but this can be easily changed. To see how an `AbstractDataStream` works, we'll loop over the rows of a dataset we use for benchmarking DataFrames.

```
using DataFrames

path = Pkg.dir("DataFrames",
               "test",
               "data",
               "scaling",
               "10000rows.csv")

ds = readstream(path)

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

As the input makes clear, every `df` object generated by this for-loop is a single row of the input data set. To work with larger minibatches, we use the `nrows` keyword argument:

```
ds = readstream(path, nrows = 1_000)

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

Note that the `df` objects generated during this for-loop are not separate objects: the memory used by `df` is rewritten during each iteration of the loop to make it easier to work with large data sets. If you need to get a separate object for each minibatch, you need to call `copy(df)` on each `df` object generated during the loop.

The Formula, ModelFrame and ModelMatrix Types

In regression model, we often want to describe the relationship between a response variable and one or more input variables in terms of main effects and interactions. To facilitate the specification of a regression model in terms of the columns of a DataFrame, the DataFrames package provides a `Formula` type, which is created by the `~` binary operator in Julia:

```
fm = Z ~ X + Y
```

A `Formula` object can be used to transform a DataFrame into a ModelFrame object:

```
df = DataFrame(X = randn(10), Y = randn(10), Z = randn(10))
mf = ModelFrame(Z ~ X + Y, df)
```

A `ModelFrame` object is just a simple wrapper around a `DataFrame`. For modeling purposes, one generally wants to construct a `ModelMatrix`, which constructs a `Matrix{Float64}` that can be used directly to fit a statistical model:

```
mm = ModelMatrix(ModelFrame(Z ~ X + Y, df))
```

Note that `mm` contains an additional column consisting entirely of `1.0` values. This is used to fit an intercept term in a regression model.

In addition to specifying main effects, it is possible to specify interactions using the `&` operator inside a `Formula`:

```
mm = ModelMatrix(ModelFrame(Z ~ X + Y + X&Y, df))
```

If you would like to specify both main effects and an interaction term at once, use the `*` operator inside a `Formula`:

```
mm = ModelMatrix(ModelFrame(Z ~ X*Y, df))
```

The construction of model matrices makes it easy to formulate complex statistical models. These are used to good effect by the GLM package.

Representing Factors using the PooledDataArray Type

Often, we have to deal with factors that take on a small number of levels:

```
dv = @data(["Group A", "Group A", "Group A",
            "Group B", "Group B", "Group B"])
```

The naive encoding used in a `DataArray` represents every entry of this vector as a full string. In contrast, we can represent the data more efficiently by replacing the strings with indices into a small pool of levels. This is what the `PooledDataArray` does:

```
pdv = @pdata(["Group A", "Group A", "Group A",
              "Group B", "Group B", "Group B"])
```

In addition to representing repeated data efficiently, the `PooledDataArray` allows us to determine the levels of the factor at any time using the `levels` function:

```
levels(pdv)
```

By default, a `PooledDataArray` is able to represent `2^32` differents levels. You can use less memory by calling the `compact` function:

```
pdv = compact(pdv)
```

Often, you will have factors encoded inside a DataFrame with `DataArray` columns instead of `PooledDataArray` columns. You can do conversion of a single column using the `pool` function:

```
pdv = pool(dv)
```

Or you can edit the columns of a `DataFrame` in-place using the `pool!` function:

```
df = DataFrame(A = [1, 1, 1, 2, 2, 2],
               B = ["X", "X", "X", "Y", "Y", "Y"])
pool!(df, ["A", "B"])
```

Pooling columns is important for working with the GLM package. When fitting regression models, `PooledDataArray` columns in the input are translated into 0/1 indicator columns in the `ModelMatrix` – with one column for each of the levels of the `PooledDataArray`. This allows one to analyze categorical data efficiently.