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 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 DataFrames 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 that operations whose
value can be determined despite the presence of `NA` 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`.

`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 DataFrames package introduces a
`DataArray` type. For example, a `DataArray{Float64}` can contain `Float64`

1

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 `DataArray` 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 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 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 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:

```
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 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 `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 will attempt to convert a `DataArray{T}` to a `Array{T}`. The `failNA` function will throw an error if any `NA` values are encountered during the conversion process. If the input vector does not contain any `NA` values, the conversion will succeed and return a standard Julia `Array` object:

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

In addition to removing or ignoring `NA` values, you can also replace any `NA` values 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 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 = DataArray([1.0 0.0; 0.0 1.0])
dm[1, 1] = NA
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` column-by-column:

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

The `DataFrame` we build in this way has 4 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 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 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"])
range(df["A"])
```

Accessing Classic Data Sets

To see more of the functionality for working with `DataFrame` objects, we need a more complex data to work with. We'll use the `RDatasets` package, which provides 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.

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::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`.

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`.

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)
```

Indexing: Making Joins and Subsetting Faster

One problem with merging large data sets is that determining which subset of rows from the left argument should be combined with which subset of rows from the right argument takes a long time. Selecting subsets in this way is slow for most DataFrames because the entries of each column have to be exhaustively examined.

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.

We are currently developing a method for indexing DataFrames, but the prototype is not yet ready for widespread use.

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["Petal.Length"]))
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, "Sepal.Length")
```

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`.