Week 9 Module 1 – Data Frames

CSCI E-5a: Programming in R

Let's clear the global computing environment:

```
rm( list = ls() )
```

Module Overview and Learning Objectives

Hello! And welcome to Module 1: Data Frames.

In this module, we'll meet one of the fundamental data structures of R: the data frame.

- In Section 1, we'll discuss the concept of the rectangular data format.
- In Section 2, we'll define data frames.
- In Section 3, we'll examine some of the built-in data frames that come with R.
- In Section 4, we'll review some useful data frame functions.
- In Section 5, we'll see how to construct a data frame.

When you've completed this module, you'll be able to:

- Explain the concept of the rectangular data format.
- Define the data frame structure in R.
- Demonstrate some of the built-in data frames that come with R.
- Use standard data frame functions.
- Construct a data frame by hand.

There are four new built-in R functions in this module:

- `str()`
- `nrow()`
- `ncol()`
- `data.frame()`

All right! Let's get started by defining the concept of the rectangular data format.

Section 1: The Rectangular Data Format

Main Idea: *We can store data in a rectangular format*

In this section, we'll discuss the concept of the rectangular data format.

So far in this course we've focused entirely on one-dimensional structures such as vectors or factors.

We've also often worked with what I call "coordinated data": that is, 2 or more vectors where the corresponding elements have some sort of association.

For instance, we can store the cereal brand for each transaction in a vector, and store the number of boxes sold in another vector, and these two vectors will be "coordinated" because corresponding elements refer to the same transaction.

This was the simplest possible example of coordinated data, and there are many variations to this basic concept.

If we have a set of vectors that represent coordinated data, it's often convenient to bundle them together.

Since we are working with coordinated data, then the vectors all have to be the same length.

They *don't* have to be the same class, though – for instance, some can be numeric vectors, and some can be character vectors.

But they all have to have the same length.

Conventionally, we bundle these vectors together, we visualize them as columns that run vertically.

Also, each column should have a name, because the columns represent variables.

Then each row represents a set of coordinated or associated values.

In our course, this is called a *rectangular data* format.

In the rectangular data format, we have a two-dimensional grid consisting of rows (which go across the grid) and columns (which go up and down the grid).

Each column has a name, and all the data in that column represents the same kind of measurement.

Here are some examples of columns in a data set in rectangular data format:

- One column consists of numeric measurements of a patient's systolic blood pressure, and this would be a `numeric` vector.
- Another column consists of categorical data indicating the presence of heart disease, and this would be a factor.
- A variable could contain the patient's name, and this would be a `character` variable.

So we can think of the columns of the rectangular data format as really just being a bunch of vectors or factors that are bound together and that all have the same length.

The rows are a little more subtle.

Each row represents an "observation", although I think a more precise term might be an "observational event".

The important idea is that there will be some point of data collection, and that all the variables in that row represent the observed values for that particular observation.

For instance, we could have a dataset which consists of a set of measurements on different people, like this:

Subject	Systolic Blood Pressure	Heart Disease
Bob	142	Present
Taylor	117	Absent
Tina	138	Absent

In this case, the observational event is each individual patient, and all the data on the row for "Bob" is associated with the patient Bob.

Notice that the rows are **NOT** vectors, because a row can cut across different types of vectors or factors.

There can be other ways to structure a data set's observational events.

For instance, it might be more interesting to measure these variables at the start and at the finish of the study.

In this case, we might have something like this:

Subject	Time	Systolic Blood Pressure	Heart Disease
Bob	Start	142	Present
Bob	Finish	133	Present
Taylor	Start	117	Absent
Taylor	Finish	119	Absent
Tina	Start	138	Present
Tina	Finish	126	Absent

In this case, the observational event is a particular patient at a certain time in the study, and all the data on each row is associated with this person at that time.

Notice that in this approach there will be multiple rows associated with each study participant.

It can become even more complicated.

For instance, suppose we have a study that consists in performing a brain scan at the beginning and end of the trial, and two brain regions are scanned.

Then we could have a dataset where the "observational event" is a specific brain region for an individual patient at a particular study time:

Subject	Time	Region	Enzyme 1	Enyme 2
Bob	Start	Hippocampus	100	104
Bob	Start	Thalamus	52	57
Bob	Finish	Hippocampus	96	110
Bob	Finish	Thalamus	61	53
Tina	Start	Hippocampus	109	100
Tina	Start	Thalamus	61	52
Tina	Finish	Hippocampus	102	107
Tina	Finish	Thalamus	64	104
Taylor	Start	Hippocampus	98	113
Taylor	Start	Thalamus	55	57
Taylor	Finish	Hippocampus	103	110
Taylor	Finish	Thalamus	55	104

Thus, the structure of the rows can be complex.

So these are the crucial properties of this rectangular data format:

- The columns represent vectors or factors.
- The data in the columns is coordinated across rows, so corresponding elements in each row are somehow associated.
- The columns all have to be of the same length.
- The columns have to all have unique names.

This rectangular data format is so common and so familiar that you might be wondering what all the fuss is about – I mean, how else would you organize data?

In fact, there are many other ways to store data, and the rectangular data format is only one of them.

For instance, pure text files can contain a lot of information, and sometimes you have to parse these files to extract this data.

There can also be more sophisticated types of database systems that use a hierarchical or linked structure to store data.

One popular modern database format is called a "graph" database, in which data is structured as a network of nodes, rather than a two-dimensional rectangular array.

Nonetheless, while the rectangular data format is not the only way to store data, it is very common, and it's the format that most people typically encounter.

So that's the rectangular data format.

Now let's see how R implements this data format.

Section 2: Data Frames

> **Main Idea:** *We can store vectors and factors in a data frame structure*

In this section, we'll define data frames.

So far, this discussion of how data can be organized has been very abstract, and if you look back at the last section you'll notice that there was not even a single line of R code.

So how can we implement the concept of the rectangular data format in R?

The answer is the *data frame*, which is just a bundle of vectors and factors that all have the same length.

The data fame structure enables us to treat this set of coordinated data as a single combined unit, but we can also select the individual vectors and factors and perform standard operations on them.

Thus, all of the techniques that we developed in the first half of the course for working with vectors and factors will be applicable.

Data frames are one of the most important compound data structures that we'll study in this course, and we'll be working with data frames for the rest of the semester.

For our course, there are three ways to obtain data frames:

- We can use built-in data frames that come with the R installation.
- We can construct them ourselves.
- We can read them in from a file.

The built-in R data frames are not very exciting, but they are useful for demonstrating techniques because everyone already has them.

It can also sometimes be useful to construct a data frame.

But the most powerful and important way to obtain a data frame is to read data in from a file, and we'll explore that in Module 2.

So, that's an overview of data frames.

Now let's meet some of the built-in data frames in R.

Section 3: Built-In R Data Frames

> **Main Idea:** *R has a collection of built-in data frames*

In this section, we'll examine some of the built-in data frames that come with R.

R has a number of collections of data for you to experiment with.

These are not particularly exciting datasets, but they are useful because they are automatically available in every R session, so everyone can practice with them.

We'll consider three simple built-in data frames.

The first is the `cars` dataframe, which consists of two variables and 50 observations:

```
cars
```

The `speed` variable of the car is how fast the car was traveling when the brakes were applied.

The `dist` variable measures the distance that the car traveled before coming to a complete stop.

The second built-in data frame that we'll use is the `iris` data frame:

```
iris
```

This data frame consists of 5 variables with 150 observations.

The third built-in data frame that we'll use is the `mtcars` data frame:

```
mtcars
```

You can obtain a complete listing of all the built-in data in R with this:

```
library( help = "datasets" )
```

So those are some of the built-in data frames in R.

Now let's see some built-in R functions for working with data frames.

Section 4: Data Frame Functions

> **Main Idea:** *R provides built-in functions to work with data frames*

In this section, we'll review some useful data frame functions.

There are a number of simple but useful functions for working with data frames.

The `head()` function

The `head()` function is useful for displaying just the first rows of the data frame.

Here are the first 8 observations for the `cars` data frame:

```
head( cars, n = 8 )
```

```
##   speed dist
## 1     4    2
## 2     4   10
## 3     7    4
## 4     7   22
## 5     8   16
## 6     9   10
## 7    10   18
## 8    10   26
```

Here are the first few observations for the `iris` data frame:

```
head( iris )
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
## 6          5.4         3.9          1.7         0.4  setosa
```

Here are the first 4 observations for the `mtcars` data frame:

```
head( mtcars, n = 4 )
```

```
##                 mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4      21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag  21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710     22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive 21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
```

Notice here that the output of the `head()` function is a data frame with the same number of columns as the original data frame.

Also, the `n` argument determines the number of *rows*, **not** the number of columns.

The `head()` function is useful when you are starting to work with a new data frame, because it enables you to quickly get a sense for how the data is represented and organized.

It can also be useful to make sure that your code is working properly.

The `str()` function

The `str()` function provides a detailed report of the structure of the data frame.

Here's the structure for the built-in `cars` data frame:

```
str( cars )
```

```
## 'data.frame':    50 obs. of  2 variables:
##  $ speed: num  4 4 7 7 8 9 10 10 10 11 ...
##  $ dist : num  2 10 4 22 16 10 18 26 34 17 ...
```

Here's the structure for the built-in `iris` data frame:

```
str( iris )
```

```
## 'data.frame':    150 obs. of  5 variables:
##  $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
##  $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
##  $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
##  $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
##  $ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1 1 ...
```

Here's the structure for the built-in `mtcars` data frame:

```
str( mtcars )
```

```
## 'data.frame':    32 obs. of  11 variables:
##  $ mpg : num  21 21 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 ...
##  $ cyl : num  6 6 4 6 8 6 8 4 4 6 ...
##  $ disp: num  160 160 108 258 360 ...
##  $ hp  : num  110 110 93 110 175 105 245 62 95 123 ...
##  $ drat: num  3.9 3.9 3.85 3.08 3.15 2.76 3.21 3.69 3.92 3.92 ...
##  $ wt  : num  2.62 2.88 2.32 3.21 3.44 ...
##  $ qsec: num  16.5 17 18.6 19.4 17 ...
##  $ vs  : num  0 0 1 1 0 1 0 1 1 1 ...
##  $ am  : num  1 1 1 0 0 0 0 0 0 0 ...
##  $ gear: num  4 4 4 3 3 3 3 4 4 4 ...
##  $ carb: num  4 4 1 1 2 1 4 2 2 4 ...
```

The `str()` function isn't particularly exciting, but sometimes it can be very useful if you really need to understand a particular detail of an R object.

The `nrow()` function

We can determine the number of rows of a data frame by using the `nrow()` function.

We saw previously with the `str()` function that the `cars` data frame has exactly 50 rows, and this is the value we obtain when we call the `nrow()` function:

```
nrow( cars )
```

```
## [1] 50
```

Likewise, we know that the `iris` data frame has 150 rows, and this is the value that the `nrow()` function returns:

```
nrow( iris )
```

```
## [1] 150
```

Finally, we know that the `mtcars` data frame has 32 rows, and this is the value that the `nrow()` function returns:

```
nrow( mtcars )
```

```
## [1] 32
```

The `ncol()` function

Just as we can use the built-in R function `nrow()` to determine the number of rows in a data frame, we can also use the built-in R function `ncol()` to determine the number of columns in the data frame.

For instance, for the `cars` data frame, there are 2 columns:

```
ncol( cars )
```

```
## [1] 2
```

Likewise, for the `iris` data frame, there are 5 columns:

```
ncol( iris )
```

```
## [1] 5
```

Finally, for the `mtcars` data frame, there are 11 columns:

```
ncol( mtcars )
```

```
## [1] 11
```

Remember, the rows are indexed by the experimental units in the study, and so `nrow()` is really counting the number of these experimental units.

Likewise, the columns are indexed by the variables in the data frame, so `ncol()` is really counting the number of variables in the data frame.

The `length()` function

There is another way to determine the number of columns in a data frame.

When we call the familiar function `length()` using a data frame as its input argument then it will return the number of columns in that data frame.

For instance, the number of columns in the `cars` data frame is 2:

```
length( cars )
```

```
## [1] 2
```

Likewise, the number of columns in the `iris` data frame is 5:

```
length( iris )
```

```
## [1] 5
```

I'm not enthusiastic about using the `length()` function this way.

I don't have any intuition about what the "length" of a data frame is – should it be the number of rows, or the number of columns, or maybe even the total number of elements in the data frame?

Because of this, the code is a little unclear.

On the other hand, the `ncol()` function does exactly the same thing as the `length()` function, and despite the abbreviation it's still clear to me that this function is going to return the number of columns().

So I think that when you want to determine the number of columns of a data frame, you should use the `ncol()` function rather than the `length()` function.

But this is just my personal preference, and strictly speaking there's nothing wrong with using the `length()` function this way.

In Module 4, we'll learn about lists, another R data structure, and we'll see why the `length()` function returns the number of columns in a data frame.

The `names()` function

Recall that each column in a data frame has a *name*, that is, a unique character string associated with it.

The names of the variables in the `cars` data frame are `speed` and `distance`:

```
head( cars )
```

```
##   speed dist
## 1     4    2
## 2     4   10
## 3     7    4
## 4     7   22
## 5     8   16
## 6     9   10
```

You can obtain a character vector consisting of the names of the columns of a data frame using the `names()` function:

```
names( cars )
```

```
## [1] "speed" "dist"
```

Here are the names of the 5 columns of the `iris` data frame:

```
names( iris )
```

```
## [1] "Sepal.Length" "Sepal.Width"  "Petal.Length" "Petal.Width"  "Species"
```

Here are the names of the columns of the `mtcars` data frame:

```
names( mtcars )
```

```
##  [1] "mpg"  "cyl"  "disp" "hp"   "drat" "wt"   "qsec" "vs"   "am"   "gear"
## [11] "carb"
```

So those are some of the built-in functions in R for working with data frames.

Now let's see how to construct data frames.

Section 5: Constructing a Data Frame

> **Main Idea:** *We can construct data frames*

In this section, we'll see how to construct a data frame.

Sometimes it's useful to be able to build our own data frames inside R.

We use the function `data.frame()` to do this, and we assign each column a name by using the `name =` construct:

```
first.vector <- 1:5
second.vector <- 2 * 6:10
third.vector <- (8:12)^2

constructed.data.frame <-
    data.frame(
        first = first.vector,
        second = second.vector,
        third = third.vector
    )

constructed.data.frame
```

```
##   first second third
## 1     1     12    64
## 2     2     14    81
## 3     3     16   100
## 4     4     18   121
## 5     5     20   144
```

We can even construct the vectors inside the `data.frame` function call:

```
constructed.data.frame <-
    data.frame(
        first = 1:5,
        second = 2 * 6:10,
        third = (8:12)^2
    )

constructed.data.frame
```

```
##   first second third
## 1     1     12    64
## 2     2     14    81
## 3     3     16   100
## 4     4     18   121
## 5     5     20   144
```

Strictly speaking, you don't have to supply names for the columns, but if you don't then R will come up with its own set of machine-generated column names, and these are typically not very pleasant:

```
unnamed.data.frame <-
    data.frame(
        1:5,
        2 * 6:10,
        (8:12)^2
    )

unnamed.data.frame
```

```
##   X1.5 X2...6.10 X.8.12..2
## 1    1        12        64
## 2    2        14        81
## 3    3        16       100
## 4    4        18       121
## 5    5        20       144
```

On the other hand, you can always repair the names directly:

```
names( unnamed.data.frame ) <-
    c( "first", "second", "third" )

unnamed.data.frame
```

```
##   first second third
## 1     1     12    64
## 2     2     14    81
## 3     3     16   100
## 4     4     18   121
## 5     5     20   144
```

So that's how to construct data frames.

Now let's review what we've learned in this module.

Module Review

In this module, we met one of the fundamental data structures of R: the data frame.

- In Section 1, we discussed the concept of the rectangular data format.
- In Section 2, we defined data frames.
- In Section 3, we examined some of the built-in data frames that come with R.
- In Section 4, we reviewed some useful data frame functions.
- In Section 5, we saw how to construct a data frame.

Now that you've completed this module, you should be able to:

- Explain the concept of the rectangular data format.
- Define the data frame structure in R.
- Demonstrate some of the built-in data frames that come with R.
- Use standard data frame functions.
- Construct a data frame by hand.

There were four new built-in R functions in this module:

- `str()`
- `nrow()`
- `ncol()`
- `data.frame()`

Allright, that's it for Module 1: Data Frames.

Now let's move on to Module 2: Reading and Writing Data Frames.

See you there!