Data Structures

Questions:

- "How can I read data in R?"
- "How do I subset data to check it out?"
- "What are the basic data types in R?"
- "How do I represent categorical information in R?"

Objectives:

- To be aware of the different types of data.
- To be able to subset using indices
- To begin exploring data frames, and understand how they are related to vectors, factors and lists.
- To be able to ask questions from R about the type, class, and structure of an object.

Keypoints:

- Use `read.csv` to read tabular data in R.
- The basic data types in R are double, integer, complex, logical, and character.
- Use factors to represent categories in R.

One of R's most powerful features is its ability to deal with tabular data - such as you may already have in a spreadsheet or a CSV file. Let's start by importing a dataset from your `data/` directory, called `stream-temps.csv`.

We can load this into R via the following:

```
temps <- read.csv(file = "data/stream-temps.csv", stringsAsFactors=TRUE)
head(temps)
```

```
##   OBSPRED_ID NoRWeST_ID MonthYear MonthlyMax MonthlyMin MonthlyMean
## 1          1 MidSnake_1   07-2010      29.36      16.79    22.42294
## 2          1 MidSnake_1   08-2010      27.87      10.38    19.77653
## 3          1 MidSnake_1   09-2010      22.67       9.38    15.05344
## 4          1 MidSnake_1   10-2010      18.57      13.40    15.59854
## 5          2 MidSnake_2   06-2010      23.97      17.60    21.02750
## 6          2 MidSnake_2   07-2010      25.98      12.79    19.82013
##          SD Ndays MaxDailyRange SampleMonth SampleYear
## 1 0.7741483    17         12.57           7       2010
## 2 2.4337631    31          9.44           8       2010
## 3 1.4410775    30         10.44           9       2010
## 4 0.3197159     4          5.17          10       2010
## 5 0.7271415     2          6.03           6       2010
## 6 1.2707267    31          9.81           7       2010
```

The `read.table` function is used for reading in tabular data stored in a text file where the columns of data are separated by punctuation characters such as CSV files (csv = comma-separated values). Tabs and commas are the most common punctuation characters used to separate or delimit data points in csv files. For convenience R provides 2 other versions of `read.table`. These are: `read.csv` for files where the data are separated with commas and `read.delim` for files where the data are separated with tabs. Of these three functions `read.csv` is the most commonly used. If needed it is possible to override the default delimiting punctuation marks for both `read.csv` and `read.delim`.

Accessing elements using their indices

To extract elements of a vector we can give their corresponding index, starting from one. Here we are calling the first value from the vector `MonthlyMax` that is held within the `temps` dataframe.

```
temps$MonthlyMax[1]
```

```
## [1] 29.36
```

```
temps[1,4]
```

```
## [1] 29.36
```

There are many ways to subset data, but these two options are the most common.
We can select a column by using the `$` operator after the name of the dataframe. We then index into the column using `[]` to pull out a single value. Alternatively we can use `[]` with two values to select a value [*row*, *column*].

If we want to see multiple values we can show a *slice* of the vector using `:`

```
temps$NoRWeST_ID[1:5]
```

```
## [1] MidSnake_1 MidSnake_1 MidSnake_1 MidSnake_1 MidSnake_2
## 3369 Levels: MidSnake_1 MidSnake_10 MidSnake_100 ... MidSnake_999
```

```
temps$MonthlyMin[1:5]
```

```
## [1] 16.79 10.38  9.38 13.40 17.60
```

Perhaps we want the data in Celcius instead of Fahrenheit, we can create a new column with the converted data:

```
temps$MonthlyMin_C = (temps$MonthlyMin-32) * (5/9)
```

Or determine the departure from mean at a given location:

```
temps$MonthlyMean[1] - temps$MonthlyMin[1]
```

```
## [1] 5.632941
```

But if you try combining a date with temperature what happens?

```
temps$MonthlyMean[1] +  temps$MonthYear[1]
```

```
## Warning in Ops.factor(temps$MonthlyMean[1], temps$MonthYear[1]): '+' not
## meaningful for factors
```

```
## [1] NA
```

Understanding what happened here is key to successfully analyzing data in R.

Data Types

AS you'd imagine, you cannot preform addition with a date, this concept of *data types* is important for programming, particularly when parsing environmental datasets. We can ask what type of data something is by:

```r
typeof(temps$MonthlyMean)
```

```
## [1] "double"
```

There are 5 main types: `double`, `integer`, `complex`, `logical` and `character`.

```r
typeof(3.14)
```

```
## [1] "double"
```

```r
typeof(1L) # The L suffix forces the number to be an integer, since by default R uses float numbers
```

```
## [1] "integer"
```

```r
typeof(1+1i)
```

```
## [1] "complex"
```

```r
typeof(TRUE)
```

```
## [1] "logical"
```

```r
typeof('mayfly')
```

```
## [1] "character"
```

No matter how complicated our analyses become, all data in R is interpreted as one of these basic data types. This strictness has some really important consequences.

Let's say someone has just started QAQC-ing the raw version of the data in the file `stream-temps-raw.csv`:

We'll load the data like we did before:

```r
temps_raw <- read.csv(file="data/stream-temps-raw.csv")
typeof(temps_raw$MonthlyMin)
```

```
## [1] "integer"
```

Our temperatures aren't the double type anymore! If we try to do the same math we did on them before, we run into trouble:

```r
temps_raw$MonthlyMean[1] - temps_raw$MonthlyMin[1]
```

```
## Warning in Ops.factor(temps_raw$MonthlyMean[1], temps_raw$MonthlyMin[1]):
## '-' not meaningful for factors
```

```
## [1] NA
```

What happened? When R reads a csv file into one of these tables, it insists that everything in a column be the same basic type; if it can't understand *everything* in the column as a double, then *nothing* in the column can be a double. The table that R loaded our temperature data into is called a *data.frame*, and it is our first example of something called a data structure - that is, a structure which R knows how to build out of the basic data types.

We can see that it is a *data.frame* by calling the `class` function on it:

```r
class(temps)
```

```
## [1] "data.frame"
```

In order to successfully use our data in R, we need to understand what the basic data structures are, and how they behave. For now, we'll use the cleaned temperature data while we investigate this behavior further.

Vectors and Type Coercion

To better understand this behavior, let's meet another of the data structures: the *vector*.

```
my_vector <- vector(length = 3)
my_vector
```

```
## [1] FALSE FALSE FALSE
```

A vector in R is essentially an ordered list of things, with the special condition that *everything in the vector must be the same basic data type*. If you don't choose the datatype, it'll default to `logical`; or, you can declare an empty vector of whatever type you like.

```
another_vector <- vector(mode='character', length=3)
another_vector
```

```
## [1] "" "" ""
```

You can check if something is a vector:

```
str(another_vector)
```

```
##  chr [1:3] "" "" ""
```

The output from this command indicates a few things 1) the basic data type found in this vector - in this case `chr`, character; 2) an indication of the number of things in the vector - actually, the indexes of the vector, in this case `[1:3]`; and 3) a few examples of what's actually in the vector - in this case empty character strings. If we similarly do

```
str(temps$SD)
```

```
##  num [1:16003] 0.774 2.434 1.441 0.32 0.727 ...
```

we see that `temps$SD` is a vector, too - *the columns of data we load into R data.frames are all vectors*, and that's the root of why R forces everything in a column to be the same basic data type.

Discussion 1

Why is R so opinionated about what we put in our columns of data? How does this help us?

Discussion 1

By keeping everything in a column the same, we allow ourselves to make simple assumptions about our data; if you can interpret one entry in the column as a number, then you can interpret *all* of them as numbers, so we don't have to check every time. This consistency is what people mean when they talk about *clean data*; in the long run, strict consistency goes a long way to making our lives easier in R.

You can also make vectors with explicit contents with the combine function:

```
combine_vector <- c(2,6,3)
combine_vector
```

```
## [1] 2 6 3
```

Given what we've learned so far, what do you think the following will produce?

```
quiz_vector <- c(2,6,'3')
```

This is something called *type coercion*, and it is the source of many surprises and the reason why we need to be aware of the basic data types and how R will interpret them. When R encounters a mix of types (here numeric and character) to be combined into a single vector, it will force them all to be the same type. Consider:

```
coercion_vector <- c('a', TRUE)
coercion_vector
```

```
## [1] "a"    "TRUE"
```

```
another_coercion_vector <- c(0, TRUE)
another_coercion_vector
```

```
## [1] 0 1
```

The coercion rules go: `logical` -> `integer` -> `numeric` -> `complex` -> `character`, where -> can be read as *are transformed into*. You can try to force coercion against this flow using the `as.` functions:

```
character_vector_example <- c('0','2','4')
character_vector_example
```

```
## [1] "0" "2" "4"
```

```
character_coerced_to_numeric <- as.numeric(character_vector_example)
character_coerced_to_numeric
```

```
## [1] 0 2 4
```

```
numeric_coerced_to_logical <- as.logical(character_coerced_to_numeric)
numeric_coerced_to_logical
```

```
## [1] FALSE  TRUE  TRUE
```

As you can see, some surprising things can happen when R forces one basic data type into another! Nitty-gritty of type coercion aside, the point is: if your data doesn't look like what you thought it was going to look like, type coercion may well be to blame; make sure everything is the same type in your vectors and your columns of data.frames, or you will get nasty surprises!

But coercion can also be very useful! For example, in our `temps_raw` data `QAQC` is a numeric vector, but we know that the 1s and 0s actually represent `TRUE` and `FALSE` (a common way of representing them). We should use the `logical` datatype here, which has two states: `TRUE` or `FALSE`, which is exactly what our data represents. We can 'coerce' this column to be `logical` by using the `as.logical` function:

```
temps_raw$QAQC[1:5]
```

```
## [1] 0 1 1 1 1
```

```
temps_raw$QAQC <- as.logical(temps_raw$QAQC)
temps_raw$QAQC[1:10]
```

```
##  [1] FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
```

The combine function, `c()`, will also append things to an existing vector:

```
ab_vector <- c('a', 'b')
ab_vector
```

```
## [1] "a" "b"
```

```
combine_example <- c(ab_vector, 'SWC')
combine_example
```

```
## [1] "a"   "b"   "SWC"
```

You can also make series of numbers:

```
mySeries <- 1:10
mySeries
```

```
## [1]  1  2  3  4  5  6  7  8  9 10
```

```r
seq(10)
```

```
## [1]  1  2  3  4  5  6  7  8  9 10
```

```r
seq(1,10, by=0.1)
```

```
##  [1]  1.0  1.1  1.2  1.3  1.4  1.5  1.6  1.7  1.8  1.9  2.0  2.1  2.2  2.3
## [15]  2.4  2.5  2.6  2.7  2.8  2.9  3.0  3.1  3.2  3.3  3.4  3.5  3.6  3.7
## [29]  3.8  3.9  4.0  4.1  4.2  4.3  4.4  4.5  4.6  4.7  4.8  4.9  5.0  5.1
## [43]  5.2  5.3  5.4  5.5  5.6  5.7  5.8  5.9  6.0  6.1  6.2  6.3  6.4  6.5
## [57]  6.6  6.7  6.8  6.9  7.0  7.1  7.2  7.3  7.4  7.5  7.6  7.7  7.8  7.9
## [71]  8.0  8.1  8.2  8.3  8.4  8.5  8.6  8.7  8.8  8.9  9.0  9.1  9.2  9.3
## [85]  9.4  9.5  9.6  9.7  9.8  9.9 10.0
```

We can ask a few questions about vectors:

```r
sequence_ex <- seq(10)
head(sequence_ex, n=3)
```

```
## [1] 1 2 3
```

```r
tail(sequence_ex, n=4)
```

```
## [1]  7  8  9 10
```

```r
length(sequence_ex)
```

```
## [1] 10
```

```r
class(sequence_ex)
```

```
## [1] "integer"
```

```r
typeof(sequence_ex)
```

```
## [1] "integer"
```

Challenge 1

What is an alternative way to show the beginning or end of a vector?

Solution 1

beginning: sequence_ex[1:3] end: sequence_ex[7:10] or sequence_ex[7:length(sequence_ex)]

There are multiple ways to many things, this will come up often when you are looking > > at someone elses code, or examples online. It's a great way to find cool tricks and > > more efficient ways to do things!

Finally, you can give names to elements in your vector:

```r
my_example <- 5:8
names(my_example) <- c("a", "b", "c", "d")
my_example
```

```
## a b c d
## 5 6 7 8
```

```r
names(my_example)
```

```
## [1] "a" "b" "c" "d"
```

Data Frames

We said that columns in data.frames were vectors:

```r
str(temps$MonthlyMax)
```

```
##  num [1:16003] 29.4 27.9 22.7 18.6 24 ...
```

```r
str(temps$QAQC)
```

```
##  NULL
```

These make sense. But what about

```r
str(temps$NoRWeST_ID)
```

```
##  Factor w/ 3369 levels "MidSnake_1","MidSnake_10",..: 1 1 1 1 1102 1102 1102 1102 2213 2213 ...
```

Factors

Another important data structure is called a *factor*. Factors usually look like character data, but are typically used to represent categorical information. For example, let's make a vector of strings labelling a few reaches of the Middle Snake:

```r
reaches <- c('MidSnake_1', 'MidSnake_2', 'MidSnake_1', 'MidSnake_3', 'MidSnake_2')
reaches
```

```
## [1] "MidSnake_1" "MidSnake_2" "MidSnake_1" "MidSnake_3" "MidSnake_2"
```

```r
str(reaches)
```

```
##  chr [1:5] "MidSnake_1" "MidSnake_2" "MidSnake_1" "MidSnake_3" ...
```

We can turn a vector into a factor like so:

```r
reach_factors<- factor(reaches)
class(reach_factors)
```

```
## [1] "factor"
```

```r
str(reach_factors)
```

```
##  Factor w/ 3 levels "MidSnake_1","MidSnake_2",..: 1 2 1 3 2
```

Now R has noticed that there are three possible categories in our data - but it also did something surprising; instead of printing out the strings we gave it, we got a bunch of numbers instead. R has replaced our human-readable categories with numbered indices under the hood, this is necessary as many statistical calculations utilise such numerical representations for categorical data:

```r
typeof(reaches)
```

```
## [1] "character"
```

```r
typeof(reach_factors)
```

```
## [1] "integer"
```

Challenge 2

Is there a factor in our `temps` data.frame? what is its name? Try using `?read.csv` to figure out how to keep text columns as character vectors instead of factors; then write a command or two to show that the factor in `temps` is actually a character vector when loaded in this way.

Solution to Challenge 2

One solution is use the argument `stringAsFactors`:

```
temps <- read.csv(file="data/stream-temps.csv", stringsAsFactors=FALSE)
str(temps$MonthYear)
```

Another solution is use the argument `colClasses` that allow finer control.

```
temps <- read.csv(file="data/stream-temps.csv", colClasses=c(NA, NA, "character"))
str(temps$MonthYear)
```

Note: help files can be difficult to understand; take your best guess based on semantic meaning, even if you aren't sure, and remember google is a great resource.

Lists

Another data structure is the `list`. A list is simpler in some ways than the other types, because you can put anything you want in it:

```
list_example <- list(1, "a", TRUE, 1+4i)
list_example
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] "a"
##
## [[3]]
## [1] TRUE
##
## [[4]]
## [1] 1+4i
```

```
another_list <- list(title = "Numbers", numbers = 1:10, data = TRUE )
another_list
```

```
## $title
## [1] "Numbers"
##
## $numbers
##  [1]  1  2  3  4  5  6  7  8  9 10
##
## $data
## [1] TRUE
```

Matrices

Last but not least is the matrix. We can declare a matrix full of zeros:

```r
matrix_example <- matrix(0, ncol=6, nrow=3)
matrix_example
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]    0    0    0    0    0    0
## [2,]    0    0    0    0    0    0
## [3,]    0    0    0    0    0    0
```

And similar to other data structures, we can ask things about our matrix:

```r
class(matrix_example)
```

```
## [1] "matrix"
```

```r
typeof(matrix_example)
```

```
## [1] "double"
```

```r
str(matrix_example)
```

```
##  num [1:3, 1:6] 0 0 0 0 0 0 0 0 0 0 ...
```

```r
dim(matrix_example)
```

```
## [1] 3 6
```

```r
nrow(matrix_example)
```

```
## [1] 3
```

```r
ncol(matrix_example)
```

```
## [1] 6
```

Challenge 3

What do you think will be the result of `length(matrix_example)`?

Were you right? Why / why not?

Solution to Challenge 3

Because a matrix is a vector with added dimension attributes, `length` gives you the total number of elements in the matrix.

Challenge 4

Make another matrix, this time containing the numbers 1:50, with 5 columns and 10 rows.

Did the `matrix` function fill your matrix by column, or by row, as its default behaviour? See if you can figure out how to change this. (hint: read the documentation for `matrix`!)

Solution to Challenge 4

```r
x <- matrix(1:50, ncol=5, nrow=10)
x <- matrix(1:50, ncol=5, nrow=10, byrow = TRUE) # to fill by row
```