Data Management in R

Instructor: Eric Dunford

Email: edunford@gmail.com

Overview

Today, we are going to cover the basics of data manipulation in R. By focusing on:

- Operations and variables
- Merging and Manipulating Data
- Cleaning Text

Operators

Mathematical Operators

Broadly speaking, `R` functions as general calculator that can process a variety of data types.

As we can see, most operators in `R` are the usual suspects, but some forms are particular to `R`.

Operation	Calc	Out
Addition	3 + 4	7
Subtraction	3 - 4	-1
Multiplication	3 * 4	12
Division	3 / 4	.75
Exponentiation	3 ^ 4	81

In the example, we'll walk through a few more operators.

Mathematical Operators: Functions

There are a range of functions designed to ease mathematical calculations. Some of these functions are to calculate specific values, such as the **natural log** or **Euler's number** (e^a).

```r
log(4)
```

```
## [1] 1.386294
```

```r
exp(5)
```

```
## [1] 148.4132
```

There are a range of functions designed to ease mathematical calculations. Others can be used to find the **sum** for a numerical vector, the **mean**, or the **median**

```r
x <-  c(1,3,7,100)
sum(x)
```

```
## [1] 111
```

```
mean(x)
```

```
## [1] 27.75
```

```
median(x)
```

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

Logical Operators

Boolean statement (i.e. true/false statements) are central to any computer programming environment. Boolean statements allow us to make quick conditional evaluations, which are key to **subsetting** data.

The following outlines the various types of boolean statements available.

```
x == y       # equals to
x != y       # does not equal
x >= y       # greater than or equal to
x <= y       # less than or equal to
x > y        # greater than
x < y        # less than
```

Statements can be combined using **and** (`&`) **or** (`|`) statements to make more specific queries.

```
x==1 & y==5 # "and" conditional statements
x==1 | y==5 # "or" conditional statements
```

Boolean statements can be fed directly into data objects via the brackets method `[]`. This offers a powerful and simple way to subset data.

```
x <-  c(1,33,100,.6,5,77)
x
```

```
## [1]   1.0  33.0 100.0   0.6   5.0  77.0
```

```
x[x > 30]
```

```
## [1]  33 100  77
```

There are also a number of base functions that provide useful boolean evaluations. Here are just a few examples...

```
is.character("hello") # for class
```

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

```
all(c(T,F,F)) # are all entries True?
```

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

```
identical(1+1,2) # are these entries the same?
```

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

Finally, boolean statements have a nice property in `R`. If we convert a boolean statement to a **numeric class**, `TRUE` values convert to `1` and `FALSE` values convert to `0`.

This offers us a quick way of generating **dichotomous** values.

```
x <- 1:10
x >= 5
```

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

```r
as.numeric(x >= 5)
```

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

Variables

The classic understanding of a "*variable*" gets an expanded meaning in `R`. Technically, any vector of information that varies constitutes a variable.

We're interested in ***relational*** data: that is, how certain variables vary alongside other variables. Such data has a **unique identifier** (an individual's name, a country ID, a year) which we use to relate different values to each other.

It is easy to see how having seperate objects for each variable can cause problems down the line. There is always a chance that the ordering of two vectors might change. Thus, we store relational data as a `data.frame` in `R`.

For the following example, let's use the `iris` data, which is a demonstration dataset inherent to `R`.

```r
data <- iris # assign the data to an object
head(data) # the first 6 entries
```

```
##   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
```

Variable names

To get find all available variable names in a `data.frame`, we can use the function `colnames()`

```r
dim(data) # dimensions of the data
```

```
## [1] 150   5
```

```r
colnames(data) # names of the variables
```

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

We can also look at the **structure** of a `data.frame` to get a summary of variable names and data types, and the number of observations/variables.

```r
str(data)
```

```
## '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 ...
```

We're already familiar with accessing information from data objects in `R`.

Recall that we access variables in a `data.frame` by:

- **dimension**
- **variable name**
- **object + handled ($) + variable name**

```
data[,1]
```

```
##   [1] 5.1 4.9 4.7 4.6 5.0 5.4 4.6 5.0 4.4 4.9 5.4 4.8 4.8 4.3 5.8 5.7 5.4
##  [18] 5.1 5.7 5.1 5.4 5.1 4.6 5.1 4.8 5.0 5.0 5.2 5.2 4.7 4.8 5.4 5.2 5.5
##  [35] 4.9 5.0 5.5 4.9 4.4 5.1 5.0 4.5 4.4 5.0 5.1 4.8 5.1 4.6 5.3 5.0 7.0
##  [52] 6.4 6.9 5.5 6.5 5.7 6.3 4.9 6.6 5.2 5.0 5.9 6.0 6.1 5.6 6.7 5.6 5.8
##  [69] 6.2 5.6 5.9 6.1 6.3 6.1 6.4 6.6 6.8 6.7 6.0 5.7 5.5 5.5 5.8 6.0 5.4
##  [86] 6.0 6.7 6.3 5.6 5.5 5.5 6.1 5.8 5.0 5.6 5.7 5.7 6.2 5.1 5.7 6.3 5.8
## [103] 7.1 6.3 6.5 7.6 4.9 7.3 6.7 7.2 6.5 6.4 6.8 5.7 5.8 6.4 6.5 7.7 7.7
## [120] 6.0 6.9 5.6 7.7 6.3 6.7 7.2 6.2 6.1 6.4 7.2 7.4 7.9 6.4 6.3 6.1 7.7
## [137] 6.3 6.4 6.0 6.9 6.7 6.9 5.8 6.8 6.7 6.7 6.3 6.5 6.2 5.9
```

```
data[,'Sepal.Length']
```

```
##   [1] 5.1 4.9 4.7 4.6 5.0 5.4 4.6 5.0 4.4 4.9 5.4 4.8 4.8 4.3 5.8 5.7 5.4
##  [18] 5.1 5.7 5.1 5.4 5.1 4.6 5.1 4.8 5.0 5.0 5.2 5.2 4.7 4.8 5.4 5.2 5.5
##  [35] 4.9 5.0 5.5 4.9 4.4 5.1 5.0 4.5 4.4 5.0 5.1 4.8 5.1 4.6 5.3 5.0 7.0
##  [52] 6.4 6.9 5.5 6.5 5.7 6.3 4.9 6.6 5.2 5.0 5.9 6.0 6.1 5.6 6.7 5.6 5.8
##  [69] 6.2 5.6 5.9 6.1 6.3 6.1 6.4 6.6 6.8 6.7 6.0 5.7 5.5 5.5 5.8 6.0 5.4
##  [86] 6.0 6.7 6.3 5.6 5.5 5.5 6.1 5.8 5.0 5.6 5.7 5.7 6.2 5.1 5.7 6.3 5.8
## [103] 7.1 6.3 6.5 7.6 4.9 7.3 6.7 7.2 6.5 6.4 6.8 5.7 5.8 6.4 6.5 7.7 7.7
## [120] 6.0 6.9 5.6 7.7 6.3 6.7 7.2 6.2 6.1 6.4 7.2 7.4 7.9 6.4 6.3 6.1 7.7
## [137] 6.3 6.4 6.0 6.9 6.7 6.9 5.8 6.8 6.7 6.7 6.3 6.5 6.2 5.9
```

```
data$Sepal.Length
```

```
##   [1] 5.1 4.9 4.7 4.6 5.0 5.4 4.6 5.0 4.4 4.9 5.4 4.8 4.8 4.3 5.8 5.7 5.4
##  [18] 5.1 5.7 5.1 5.4 5.1 4.6 5.1 4.8 5.0 5.0 5.2 5.2 4.7 4.8 5.4 5.2 5.5
##  [35] 4.9 5.0 5.5 4.9 4.4 5.1 5.0 4.5 4.4 5.0 5.1 4.8 5.1 4.6 5.3 5.0 7.0
##  [52] 6.4 6.9 5.5 6.5 5.7 6.3 4.9 6.6 5.2 5.0 5.9 6.0 6.1 5.6 6.7 5.6 5.8
##  [69] 6.2 5.6 5.9 6.1 6.3 6.1 6.4 6.6 6.8 6.7 6.0 5.7 5.5 5.5 5.8 6.0 5.4
##  [86] 6.0 6.7 6.3 5.6 5.5 5.5 6.1 5.8 5.0 5.6 5.7 5.7 6.2 5.1 5.7 6.3 5.8
## [103] 7.1 6.3 6.5 7.6 4.9 7.3 6.7 7.2 6.5 6.4 6.8 5.7 5.8 6.4 6.5 7.7 7.7
## [120] 6.0 6.9 5.6 7.7 6.3 6.7 7.2 6.2 6.1 6.4 7.2 7.4 7.9 6.4 6.3 6.1 7.7
## [137] 6.3 6.4 6.0 6.9 6.7 6.9 5.8 6.8 6.7 6.7 6.3 6.5 6.2 5.9
```

Creating, Renaming, and Deleting Variables

We can use these same properties to **create** and **delete** variables.

To **create** a variable, we simply specify a new variable name and a vector of information of **equal length**

> ... that is: we cannot add a variable that only has *50* observations to a `data.frame` that has *150* observations.

For the following example, let's create a **dichotomous** variable that takes on the value of `1` if a plant's sepal lenght is greater than 5, `0` otherwise.

Again, we can do this by using a **boolean statement** and then converting that statement into a **numeric class** with the function `as.numeric()`.

```r
as.numeric(data$Sepal.Length >= 5)
```

```
##   [1] 1 0 0 0 1 1 0 1 0 0 1 0 0 0 1 1 1 1 1 1 1 1 0 1 0 1 1 1 1 0 0 1 1 1 0
##  [36] 1 1 0 0 1 1 0 0 1 1 0 1 0 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1
##  [71] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [106] 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [141] 1 1 1 1 1 1 1 1 1 1
```

Creating Variables

We **generate a new variable** by specifying a new variable name using the `$` handle and then assigning our variable to that name.

```r
x <- as.numeric(data$Sepal.Length >= 5)
data$new_var <- x
head(data)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species new_var
## 1          5.1         3.5          1.4         0.2  setosa       1
## 2          4.9         3.0          1.4         0.2  setosa       0
## 3          4.7         3.2          1.3         0.2  setosa       0
## 4          4.6         3.1          1.5         0.2  setosa       0
## 5          5.0         3.6          1.4         0.2  setosa       1
## 6          5.4         3.9          1.7         0.4  setosa       1
```

Likewise, we can specify the variable name **within** brackets `[]` to create a variable. Here let's set `1` to any length that is less than 5.

```r
y <- as.numeric(data$Sepal.Length < 5)
data[,'new_var2'] <- y
head(data)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species new_var
## 1          5.1         3.5          1.4         0.2  setosa       1
## 2          4.9         3.0          1.4         0.2  setosa       0
## 3          4.7         3.2          1.3         0.2  setosa       0
## 4          4.6         3.1          1.5         0.2  setosa       0
## 5          5.0         3.6          1.4         0.2  setosa       1
## 6          5.4         3.9          1.7         0.4  setosa       1
##   new_var2
## 1        0
## 2        1
## 3        1
## 4        1
## 5        0
## 6        0
```

Renaming Variables

After creating a variable, we often need to assign it a new name. We can do this by directly manipulating the `colnames()` function.

As we know `colnames()` or `names()` prints the names of each variable in a data frame.

```r
colnames(data)
```

```
## [1] "Sepal.Length" "Sepal.Width"  "Petal.Length" "Petal.Width"
## [5] "Species"      "new_var"      "new_var2"
```

`colnames()` presents us with a vector of character names. We can use positions in that vector to change a variables name.

```r
colnames(data)[7]
```

```
## [1] "new_var2"
```

```r
colnames(data)[7] <- "new_name"
colnames(data)[7]
```

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

Creating Categorical Variables

When creating variables, often we wish to **generate categories** rather than dummies.

We can do this easily using the base `ifelse()` function in `R`.

```r
ifelse(data$Sepal.Length >=5,"large","small")
```

```
##   [1] "large" "small" "small" "small" "large" "large" "small" "large"
##   [9] "small" "small" "large" "small" "small" "small" "large" "large"
##  [17] "large" "large" "large" "large" "large" "large" "small" "large"
##  [25] "small" "large" "large" "large" "large" "small" "small" "large"
##  [33] "large" "large" "small" "large" "large" "small" "small" "large"
##  [41] "large" "small" "small" "large" "large" "small" "large" "small"
##  [49] "large" "large" "large" "large" "large" "large" "large" "large"
##  [57] "large" "small" "large" "large" "large" "large" "large" "large"
##  [65] "large" "large" "large" "large" "large" "large" "large" "large"
##  [73] "large" "large" "large" "large" "large" "large" "large" "large"
##  [81] "large" "large" "large" "large" "large" "large" "large" "large"
##  [89] "large" "large" "large" "large" "large" "large" "large" "large"
##  [97] "large" "large" "large" "large" "large" "large" "large" "large"
## [105] "large" "large" "small" "large" "large" "large" "large" "large"
## [113] "large" "large" "large" "large" "large" "large" "large" "large"
## [121] "large" "large" "large" "large" "large" "large" "large" "large"
## [129] "large" "large" "large" "large" "large" "large" "large" "large"
## [137] "large" "large" "large" "large" "large" "large" "large" "large"
## [145] "large" "large" "large" "large" "large" "large"
```

Recall that `Factors` are special object class in `R` specifically designed for categorical varaibles.

Consider the following vector containing the categories: “small”, “medium”, and “large”.

```r
var <- c("small","medium","small","large","medium")
var
```

```
## [1] "small"  "medium" "small"  "large"  "medium"
```

By converting this `character` vector to a `factor`, we add a numerical layer (or “**Levels**”) to each ***unique*** category.

```r
var_new <- as.factor(var)
var_new
```

```
## [1] small  medium small  large  medium
## Levels: large medium small
```

Factors offer a quick way for `R` to processes character data when running analysis. As each unique character value is inherently assigned a numerical value.

```
var_new
```

```
## [1] small  medium small  large  medium
## Levels: large medium small
```

```
as.numeric(var_new)
```

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

Deleting Variables

To get rid of a variable…

- switch the variable value out with a `NULL`
- **drop** the variable using a **negative** value on the dimension (and writing over the existing object `data`)
- **subsetting** a variable out.

```
data$new_var <- NULL
colnames(data)
```

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

```
data <- data[,-6]
colnames(data)
```

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

```
var_names <- colnames(data)
var_names != "Species"
```

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

```
data <- data[ , var_names != "Species"]
```

Dealing with Missing Values

Missingness is a common issue for most data.

Detecting and **removing** missing values is often necessary to run analysis.

`R` contains a number of functions that ease the the identification and removal of missing values.

`NA` is `R`s internal identifier for a missing value.

Let's create a "fake" dataset, that contains missing values

```
data2 <- data.frame(
  var1 = c("a","b","c",NA),
  var2 = c(1,NA,3,4)
)
data2
```

```
##   var1 var2
## 1    a    1
## 2    b   NA
## 3    c    3
## 4 <NA>    4
```

`is.na()` is a base `R` function that provides a boolean statement for whether a value is missing or not.

```
is.na(data2)
```

```
##       var1  var2
## [1,] FALSE FALSE
## [2,] FALSE  TRUE
## [3,] FALSE FALSE
## [4,]  TRUE FALSE
```

By using `!` before a boolean, we can get the **inverse** logical statement.

```
!is.na(data2)
```

```
##       var1  var2
## [1,]  TRUE  TRUE
## [2,]  TRUE FALSE
## [3,]  TRUE  TRUE
## [4,] FALSE  TRUE
```

We can use `!is.na()` to **subset out missing values**. For this, we need to specify a specific variable with potentially missing values

```
data2[!is.na(data2$var1), ]
```

```
##   var1 var2
## 1    a    1
## 2    b   NA
## 3    c    3
```

```
data2[!is.na(data2$var2), ]
```

```
##   var1 var2
## 1    a    1
## 3    c    3
## 4 <NA>    4
```

The base function `complete.cases()` performs a similar task by only identifying observations that are "complete" (i.e. contain no missing information),

```
complete.cases(data2)
```

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

This provides us with a logical vector equal to the number of observations in our dataset. We can insert this vector into the *rows* dimension to subset out all incomplete observations.

```
data2[complete.cases(data2), ]
```

```
##   var1 var2
## 1    a    1
## 3    c    3
```

Finally, recall that all boolean statements can be easily converted into **numerical values**. We can use this in conjunction with the `sum()` function to get a count of the total number of incomplete observations in our data.

```r
sum(complete.cases(data2))
```

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

The following tells us that two observations contain incomplete information.

Data Management

Merging Data

Merging data in `R` requires that there is at least one **unique** column that is *shared between* two data frames.

We use the base function `merge()` to match two data sets.

Consider the following two example datasets…

```r
data_A
```

```
##    country Var1
## 1  Nigeria    4
## 2  England    3
## 3 Botswana    6
```

```r
data_B
```

```
##         country   Var2
## 1       Nigeria    Low
## 2 United States   High
## 3      Botswana Medium
```

We can merge these two datasets by their shared column, or "unique identifier".

```r
merge(data_A,data_B,by=c("country"))
```

```
##    country Var1   Var2
## 1 Botswana    6 Medium
## 2  Nigeria    4    Low
```

When we do this, we only get back the observations that match.

But maybe we don't want to drop observations…

We can get around this issue by specifying the argument `all=T`. This tells `merge()` that we want *all* observation back even if they didn't match.

```r
merge(data_A,data_B,by=c("country"),all=T)
```

```
##         country Var1   Var2
## 1      Botswana    6 Medium
## 2       England    3   <NA>
## 3       Nigeria    4    Low
## 4 United States   NA   High
```

We can be specific about which dataset we retain information by using the arguments `all.x=T` or `all.y=T`

```r
merge(data_A,data_B,by=c("country"),all.x=T)
```

```
##    country Var1   Var2
## 1 Botswana    6 Medium
## 2  England    3   <NA>
## 3  Nigeria    4    Low
```

```
merge(data_A,data_B,by=c("country"),all.y=T)
```

```
##         country Var1   Var2
## 1      Botswana    6 Medium
## 2       Nigeria    4    Low
## 3 United States   NA   High
```

Appending Data

Sometimes we aren't looking to match observations but rather we just want to **append** (or stack) one dataset onto another.

We can use the `rbind()` — "bind by row" — or the `cbind()` — "bind by column" — for this.

The only requirements is that the columns share the ***same column names***.

For example, consider let's bind the following data entries…

```
entry1
```

```
##    ID Value
## 1 e23   4.5
```

```
entry2
```

```
##    ID Value
## 1 e22   5.4
```

```
rbind(entry1,entry2) # by row
```

```
##    ID Value
## 1 e23   4.5
## 2 e22   5.4
```

```
cbind(entry1,entry2) # by column
```

```
##    ID Value  ID Value
## 1 e23   4.5 e22   5.4
```

Summarizing Data

Now that we know how to create, drop, and merge variables, we can begin to explore ways to summarize.

`R` has `summary()` and `table()` functions that ease the breakdown of variables so that they are easier to understand and analyze.

We'll explore some of these base functions and then go into depth exploring the powerful `dplyr` package, which makes data summary real easy!

summary()

The `summary()` function is inherent to almost any dat object and will offer a summary print that offers summary statistics on the distribution of a variable.

Let's look at `Sepal.Length` from the `iris` data we created earlier.

```
summary(data$Sepal.Length)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   4.300   5.100   5.800   5.843   6.400   7.900
```

table()

The `table()` function offers a count breakdown of how many unique observations fall within a specific category or interval. This function is most useful for categorical variables.

Here we have three plant species categories for which we have 50 observations each.

```
table(data$Species)
```

```
##
##     setosa versicolor  virginica
##         50         50         50
```

`table()` can also be used simultaneously with two variables to produce a simple crosstab.

```
large <- as.numeric(data$Sepal.Length>=6)

data$large <- large

table(data$large,data$Species)
```

```
##
##     setosa versicolor virginica
##   0     50         26         7
##   1      0         24        43
```

dplyr package

There are many different ways to perform the same task in `R`. Some are easier than others, and ultimately, what we are concerned with is producing results quickly, effectively, and reliably.

The `dplyr` package offers an intuitive **verb based** approach to data management.

To get started, let's install and load the `dplyr` package.

```
install.packages("dplyr")
require(dplyr)
```

```
## Loading required package: dplyr
```

```
##
## Attaching package: 'dplyr'
```

```
## The following objects are masked from 'package:stats':
##
##     filter, lag
```

```
## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

dplyr functions

- `select()` :: select particular variables
- `arrange()` :: order by a specific variable
- `rename()` :: quickly rename variables
- `filter()` :: subset data by some condition
- `group_by()` :: group data by a category
- `summarize()` :: summarize data by some criteria
- `tally()` :: generates counts
- `mutate()` :: create a new variable off some manipulation
- `transmute()` :: create a new variable and only retain that variable

The goal of the package is to provide easy, intuitive names that perform a clear task that is ubiquitous to data management.

select()

With `select()` we can quickly subset our data by each variable name.

```
x <- select(data,Sepal.Length,Species)
head(x)
```

```
##   Sepal.Length Species
## 1          5.1  setosa
## 2          4.9  setosa
## 3          4.7  setosa
## 4          4.6  setosa
## 5          5.0  setosa
## 6          5.4  setosa
```

Or variable ranges using `:` —- the following will provide all variables in-between Sepal.Length and Petal.Length.

```
x <- select(data,Sepal.Length:Petal.Length)
head(x)
```

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

Lastly, `select()` offers us a convenient way to drop variables by using the same logic that we employed with putting a **negative sign** in front of a dimension. The only difference here is that we can do the same but with a variable name.

Here we **drop** the variables `Sepal.Length`, `Petal.Length`, and `Petal.Width`

```
x <- select(data,-Sepal.Length,-Petal.Length,
       -Petal.Width)
head(x)
```

```
##   Sepal.Width Species large
## 1         3.5  setosa     0
## 2         3.0  setosa     0
## 3         3.2  setosa     0
## 4         3.1  setosa     0
## 5         3.6  setosa     0
## 6         3.9  setosa     0
```

arrange()

With `arrange()`, we can quickly order data by a variable.

```r
# Order by petal length
x <- arrange(data,Petal.Length)
# select Petal Length and Species Columns
x <- select(x,Petal.Length,Species)
head(x)
```

```
##   Petal.Length Species
## 1          1.0  setosa
## 2          1.1  setosa
## 3          1.2  setosa
## 4          1.2  setosa
## 5          1.3  setosa
## 6          1.3  setosa
```

filter()

`filter()` offers a quick way to **subset** data by some condition.

```r
x <- filter(data,Petal.Length >= 6.5)
x <- select(x,Petal.Length,Species)
head(x)
```

```
##   Petal.Length   Species
## 1          6.6 virginica
## 2          6.7 virginica
## 3          6.9 virginica
## 4          6.7 virginica
```

group_by() & summarize()

`group_by()` offers a way to cluster the data by a specific category. And `summarize()` offers usa quick way of summarizing some variable by a specific function.

```r
# Group the data by species
x <- group_by(data,Species)
x2 <- summarize(x,mean(Petal.Length))
head(x2)
```

```
## # A tibble: 3 × 2
##      Species `mean(Petal.Length)`
##       <fctr>                <dbl>
## 1     setosa                1.462
## 2 versicolor                4.260
## 3  virginica                5.552
```

Note that `dplyr` has its own naming feature for it's functions. We merely need to set the name up `name =` prior to any manipulation.

```
summarize(x,pl_mean = mean(Petal.Length))
```

```
## # A tibble: 3 × 2
##      Species pl_mean
##       <fctr>   <dbl>
## 1     setosa   1.462
## 2 versicolor   4.260
## 3  virginica   5.552
```

Finally, just as we can `group_by()` specific categories of data. we can also `ungroup()`. Remember that you'd need to `ungroup()` if you want to group by a different category.

rename()

In fact, `dplyr` has a function `rename()` specifically designed for renaming variables.

```
x <- rename(data,
           SL = Sepal.Length,
           SW = Sepal.Width,
           PL = Petal.Length,
           PW = Petal.Width,
           S = Species)
head(x)
```

```
##    SL  SW  PL  PW      S large
## 1 5.1 3.5 1.4 0.2 setosa     0
## 2 4.9 3.0 1.4 0.2 setosa     0
## 3 4.7 3.2 1.3 0.2 setosa     0
## 4 4.6 3.1 1.5 0.2 setosa     0
## 5 5.0 3.6 1.4 0.2 setosa     0
## 6 5.4 3.9 1.7 0.4 setosa     0
```

tally()

`tally()` offers quick counts of a variable which can be quite useful when used alongside some of the other functions.

Here we are seeing how many observations we have *by group*.

```
x <- group_by(data,Species)
tally(x)
```

```
## # A tibble: 3 × 2
##      Species     n
##       <fctr> <int>
## 1     setosa    50
## 2 versicolor    50
## 3  virginica    50
```

mutate()

`mutate()` provides us a way to quickly generate a new variable.

Here we are going to use a conditional statement to create a boolean variable.

```r
x <- select(data,Sepal.Length,Species)
x <- mutate(x,var = Sepal.Length<=5)
head(x)
```

```
##   Sepal.Length Species   var
## 1          5.1  setosa FALSE
## 2          4.9  setosa  TRUE
## 3          4.7  setosa  TRUE
## 4          4.6  setosa  TRUE
## 5          5.0  setosa  TRUE
## 6          5.4  setosa FALSE
```

`mutate()` also allows us to use variables we just created.

```r
x <- select(data,Sepal.Length,Species)
x <- mutate(x,var = Sepal.Length<=5,
       var2 = as.numeric(var))
head(x)
```

```
##   Sepal.Length Species   var var2
## 1          5.1  setosa FALSE    0
## 2          4.9  setosa  TRUE    1
## 3          4.7  setosa  TRUE    1
## 4          4.6  setosa  TRUE    1
## 5          5.0  setosa  TRUE    1
## 6          5.4  setosa FALSE    0
```

transmute()

Like `mutate()`, `transmute()` provides a method for generating a new variable, but unlike the former, it **returns only the newly created variable**.

```r
x <- transmute(data,var = Sepal.Length<=5)
head(x)
```

```
##     var
## 1 FALSE
## 2  TRUE
## 3  TRUE
## 4  TRUE
## 5  TRUE
## 6 FALSE
```

Combining Functions

When we need to do a series of manipulations, we can **perform each manipulation individually and save each entry as a new object** that we write over.

```r
x <- filter(data,Sepal.Length >= 6)
x <- mutate(x,var = Sepal.Length>5)
x <- group_by(x,Species)
x <- summarize(x,num_large = sum(var),
         slMean =  mean(Sepal.Length))
x
```

```
## # A tibble: 2 × 3
##      Species num_large  slMean
##       <fctr>     <int>   <dbl>
## 1 versicolor        24 6.37500
## 2  virginica        43 6.74186
```

Nesting Functions

Or we can **nest** functions within each other.

```
tally(group_by(filter(data,Sepal.Length >= 6),Species))
```

```
## # A tibble: 2 × 2
##      Species     n
##       <fctr> <int>
## 1 versicolor    24
## 2  virginica    43
```

The issue with **nesting functions** is that it is difficult to detect a mistake.

Piping Functions

The **pipe** is a useful tool that allows us to **pass** output from one function to the next.

To pipe we write `%>%` ***in-between*** each function.

`function1() %>% function2()`

Piping provides a clear way to perform sophistocated manipulations with ease.

Here we **pass** our `data` to `filter()` then to `group_by()` then to `tally()` which returns our output!

```
data %>%  filter(Sepal.Length >= 6) %>%
  group_by(Species) %>%
  tally()
```

```
## # A tibble: 2 × 2
##      Species     n
##       <fctr> <int>
## 1 versicolor    24
## 2  virginica    43
```

What this offers is a clean way of manipulating data that is intuitive and easy to read.

Cleaning Text

Dealing with text

We often must deal with problematic text data. Sometimes we need to format responses from a survey so that we can use them in some analysis; other times we are just trying to calculate the date.

Most data is often riddled with errors and issues that are costly to resolve. In a sense, this data is ***dirty***. We can't run analysis on dirty data.

Regular expressions are a special text string for describing a search pattern. We can extract, clean, and manipulate text using these expressions — which can save one **hours** from needing to manually clean data.

Regular Expressions

Consider the following string vector…

```
countries <- c("Canada","Russia","New Zealand","New Guinea")
```

Say, from this vector, we want to identify which entry contains the word "new". The `grep()` function can help us identify a specific pattern, which it will then return the position of the string.

```
grep(pattern = "New", countries)
```

```
## [1] 3 4
```

Here it return position **3** and position **4**, which correspond with the ***position in the vector***.

grep()

We can use that position to draw out specific content.

```
position <- grep(pattern = "New", countries)
countries[position]
```

```
## [1] "New Zealand" "New Guinea"
```

This feature can be useful to identify relevant content in variable or body of text.

gsub()

`gsub()` can help us actually manipulate the content in a string by identifying a pattern and then ***replacing it*** with something new.

```
countries
```

```
## [1] "Canada"      "Russia"      "New Zealand" "New Guinea"
```

```
gsub(pattern = "New",replacement = "Old",countries)
```

```
## [1] "Canada"      "Russia"      "Old Zealand" "Old Guinea"
```

Cases

We can also manipulate cases with the `tolower()` and `toupper()` functions.

```
string <- "This Is ReAlLY imPORtant."
tolower(string)
```

```
## [1] "this is really important."
```

```
toupper(string)
```

```
## [1] "THIS IS REALLY IMPORTANT."
```

Trimming White Space

We can also get rid of excessive **spaces** using the `trimws()` function.

```
sent <- "         This sentence has a ton of white space             "
sent
```

```
## [1] "         This sentence has a ton of white space             "
```

```
trimws(sent)
```

```
## [1] "This sentence has a ton of white space"
```

Generic Patterns

There are generic ways to draw out specific kinds of content from a string: such as **digits** or **punctuation**. There are many different types of regular expressions, and we don't have time to review all of them here, but here are a few useful ones.

- `"\\w"` → words
- `"\\d"` → digits
- `"\\s"` → space character
- `"*"` → fuzzy
- `"+"` → More than one
- `"[]"` → Match anything inside the brackets

Here let's remove problems from the following string using `gsub()`.

```
trouble <- "This ::String is a 2Problem; 56"
trouble <- gsub("[::]","",trouble)
trouble
```

```
## [1] "This String is a 2Problem; 56"
```

```
trouble <- gsub("\\d*","",trouble)
trouble
```

```
## [1] "This String is a Problem; "
```

```
trouble <- gsub("[;]",".",trouble)
trimws(trouble)
```

```
## [1] "This String is a Problem."
```

We can also target all **punctuation** with the `"[[:punct:]]"` regular expression.

```
dirt <- "C^lean%% this $%&*_@string((!"
dirt
```

```
## [1] "C^lean%% this $%&*_@string((!"
```

```
gsub("[[:punct:]]","",dirt)
```

```
## [1] "Clean this string"
```

Joining Text

We can also **join** or **paste** text using `R`. To do so, we'll use the `paste()` function, which takes two arguments: the strings and a specified seperator.

```
sent1 <- "It is nice outside."
sent2 <- "I'll go for a walk."
paste(sent1,sent2,sep = " ")
```

```
## [1] "It is nice outside. I'll go for a walk."
```

```
paste(sent1,sent2,sep = "::::")
```

```
## [1] "It is nice outside.::::I'll go for a walk."
```

Collapsing

We can also use `paste()` to **collapse** a string vectors down into a single line. We do this by specifying the `collapse=` argument, which is like seperate in that it wants to know how the vector should be collapsed.

```
countries
```

```
## [1] "Canada"      "Russia"      "New Zealand" "New Guinea"
```

```
paste(countries,collapse=", ")
```

```
## [1] "Canada, Russia, New Zealand, New Guinea"
```

`collapse=` can be used with `paste()` in useful ways.

```
sent1 <- "These are the countries that matter:"

countries_sent <- paste(countries,
                        collapse=", ")

paste(sent1, countries_sent,sep=" ")
```

```
## [1] "These are the countries that matter: Canada, Russia, New Zealand, New Guinea"
```

Dates and Time

`R` has a specific `Date` class. We will use the function `as.Date()` to coerce a relevant string into a date class.

```
str <- "2006-04-30"
class(str)
```

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

```
date_str <- as.Date(str)
class(date_str)
```

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

Objects of class date have some nice properties, that makes analysis and manipulation easy.

```
date_str
```

```
## [1] "2006-04-30"
```

```
date_str + 30 # date in 30 days
```

```
## [1] "2006-05-30"
```

```
date_str - 3000 # date 300 days ago.
```

```
## [1] "1998-02-11"
```

This also allows us to look at the distance between two dates.

```
date1
```

```
## [1] "2015-06-07"
```

```
date2
```

```
## [1] "2013-02-14"
```

```
date1-date2
```

```
## Time difference of 843 days
```

Formatting Dates

That said, dates come in many different formats. To let `R` know that a specific string is a date, we have to tell it the **date format**.

```
example <- "February 3, 1987"
as.Date(example)
```

```
  Error in charToDate(x) :
  character string is not in a standard unambiguous format
```

That said, dates come in many different formats. To let `R` know that a specific string is a date, we have to tell it the **date format**.

```
example <- "February 3, 1987"
as.Date(example, format = "%B %d, %Y")
```

```
## [1] "1987-02-03"
```

Formatting dates is similar to regular expressions in that it has a special syntax. In a string (i.e. using " "), we specify the exact pattern of the date with ***all appropriate punctuation and spacing***.

The following are the main expressions used in formatting.

- `%d` → day as a number
- `%a` → abbreviated weekday
- `%A` → unabbreviated weekday
- `%m` → month as number
- `%b` → abbreviated month
- `%B` → unabbreviated month
- `%y` → 2 digit year
- `%Y` → 4 digit year

```
as.Date("Friday March 13, 2009","%A %B %d, %Y")
```

```
## [1] "2009-03-13"
```

```
as.Date("11/13/14","%m/%d/%y")
```

```
## [1] "2014-11-13"
```

```
as.Date("7th of May 2000","%dth of %B %Y")
```

```
## [1] "2000-05-07"
```