Data Management

April 12, 2016

Contents

1 Basic Data Management **2**
1.1 Generating data . . . 3
1.1.1 Generating sequences . . . 3
1.1.2 Generating repeats . . . 4
1.1.3 Generating factor levels . . . 4
1.2 Recoding variables . . . 5
1.3 Renaming variables . . . 6
1.4 Missing values . . . 6
1.4.1 Check for missing values . . . 6
1.4.2 Excluding missing values . . . 7
1.4.3 Using NULL . . . 8
1.5 Date values . . . 9
1.6 Sorting data . . . 11
1.7 Subsetting datasets . . . 12
1.7.1 Selecting (keeping) variables . . . 12
1.7.2 Excluding (dropping) variables . . . 12
1.7.3 Selecting observations . . . 13
1.7.4 Random samples . . . 14

2 Aggregation and restructuring **15**
2.1 Aggregation . . . 15
2.2 Merging datasets . . . 17
2.2.1 Adding columns . . . 17
2.2.2 plyr join . . . 19
2.2.3 Adding rows . . . 19
2.2.4 denominate . . . 20
2.2.5 Transpose . . . 20
2.3 The reshape package . . . 20
2.3.1 Melting . . . 20
2.3.2 Casting . . . 21
2.3.3 Convert from wide to long (tall) format . . . 22
2.3.4 Convert from long (tall) to wide format . . . 22

3 Manipulating Strings **22**
3.1 Paste . . . 22
3.2 Sprintf . . . 23
3.3 Extracting Text . . . 23
3.4 Regular Expressions . . . 24
3.5 Example: Download data from website . . . 27

1 Basic Data Management

In a typical research project, you'll need to create new variables and transform existing ones. This is accomplished with statements of the form

```
variable <- expression
```

A wide array of operators and functions can be included in the expression portion of the statement. Let's say that you have a data frame named mydata, with variables x1 and x2, and you want to create a new variable sumx that adds these two variables and a new variable called meanx that averages the two variables. If you use the code

```
mydata <- data.frame(x1 = c(2, 2, 6, 4), x2 = c(3, 4, 2, 8))
sumx <- x1 + x2
meanx <- (x1 + x2)/2
```

you'll get an error, because R doesn't know that x1 and x2 are from data frame mydata. If you use this code instead

```
mydata <- data.frame(x1 = c(2, 2, 6, 4), x2 = c(3, 4, 2, 8))
sumx <- mydata$x1 + mydata$x2
meanx <- (mydata$x1 + mydata$x2)/2
```

the statements will succeed but you'll end up with a data frame (mydata) and two separate vectors (sumx and meanx). This is probably not what you want. Ultimately, you want to incorporate new variables into the original data frame. The following listing provides three separate ways to accomplish this goal.

```
mydata <- data.frame(x1 = c(2, 2, 6, 4), x2 = c(3, 4, 2, 8))
mydata$sumx <- mydata$x1 + mydata$x2
mydata$meanx <- (mydata$x1 + mydata$x2)/2
mydata

##   x1 x2 sumx meanx
## 1  2  3    5   2.5
## 2  2  4    6   3.0
## 3  6  2    8   4.0
## 4  4  8   12   6.0
```

```
attach(mydata)
mydata$sumx <- x1 + x2
mydata$meanx <- (x1 + x2)/2
detach(mydata)
mydata

##   x1 x2 sumx meanx
## 1  2  3    5   2.5
## 2  2  4    6   3.0
## 3  6  2    8   4.0
## 4  4  8   12   6.0
```

```r
mydata <- transform(mydata, sumx = x1 + x2, meanx = (x1 + x2)/2)
mydata

##   x1 x2 sumx meanx
## 1  2  3    5   2.5
## 2  2  4    6   3.0
## 3  6  2    8   4.0
## 4  4  8   12   6.0
```

1.1 Generating data

1.1.1 Generating sequences

An important way of creating vectors is to generate a sequence of numbers. The simplest sequences are in steps of 1, and the colon operator is the simplest way of generating such sequences.

```r
0:10

##  [1]  0  1  2  3  4  5  6  7  8  9 10

15:5

##  [1] 15 14 13 12 11 10  9  8  7  6  5

seq(0, 1.5, 0.1)

##  [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0 1.1 1.2 1.3 1.4 1.5

seq(6,4,-0.2)

##  [1] 6.0 5.8 5.6 5.4 5.2 5.0 4.8 4.6 4.4 4.2 4.0
```

In many cases, you want to generate a sequence to match an existing vector in length. Rather than having to figure out the increment that will get from the initial to the final value and produce a vector of exactly the appropriate length, R provides the along and length options.

```r
N <- c(55,76,92,103,84,88,121,91,65,77,99)
seq(from=0.04,by=0.01,length=11)

##  [1] 0.04 0.05 0.06 0.07 0.08 0.09 0.10 0.11 0.12 0.13 0.14

seq(0.04,by=0.01,along=N)

##  [1] 0.04 0.05 0.06 0.07 0.08 0.09 0.10 0.11 0.12 0.13 0.14

seq(from=0.04,to=0.14,along=N)

##  [1] 0.04 0.05 0.06 0.07 0.08 0.09 0.10 0.11 0.12 0.13 0.14
```

Notice that when the increment does not match the final value, then the generated sequence stops short of the last value (rather than overstepping it):

```
seq(1.4,2.1,0.3)

## [1] 1.4 1.7 2.0
```

If you want a vector made up of sequences of unequal lengths, then use the sequence function. Suppose that most of the five sequences you want to string together are from 1 to 4, but the second one is 1 to 3 and the last one is 1 to 5, then:

```
sequence(c(4,3,4,4,4,5))

##  [1] 1 2 3 4 1 2 3 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4 5
```

1.1.2 Generating repeats

You will often want to generate repeats of numbers or characters, for which the function is rep.

```
rep(9,5)

## [1] 9 9 9 9 9

rep(1:4, 2)

## [1] 1 2 3 4 1 2 3 4

rep(1:4, each = 2)

## [1] 1 1 2 2 3 3 4 4

rep(1:4, each = 2, times = 3)

##  [1] 1 1 2 2 3 3 4 4 1 1 2 2 3 3 4 4 1 1 2 2 3 3 4 4

rep(1:4,1:4)

##  [1] 1 2 2 3 3 3 4 4 4 4

rep(1:4,c(4,1,4,2))

##  [1] 1 1 1 1 2 3 3 3 3 4 4

rep(c("cat","dog","gerbil","goldfish","rat"),c(2,3,2,1,3))

##  [1] "cat"      "cat"      "dog"      "dog"      "dog"      "gerbil"
##  [7] "gerbil"   "goldfish" "rat"      "rat"      "rat"
```

1.1.3 Generating factor levels

The function gl ('generate levels') is useful when you want to encode long vectors of factor levels. The syntax for the three arguments is: 'up to', 'with repeats of', 'to total length'.

```
gl(4,3)

##  [1] 1 1 1 2 2 2 3 3 3 4 4 4
## Levels: 1 2 3 4
```

```r
gl(4,3,24)

##  [1] 1 1 1 2 2 2 3 3 3 4 4 4 1 1 1 2 2 2 3 3 3 4 4 4
## Levels: 1 2 3 4

Soft <- gl(3, 2, 6, labels = c("Hard","Medium","Soft"))
Temp <- gl(2, 3, 6, labels = c("Low", "High"))
data.frame(Temp,Soft)

##   Temp   Soft
## 1  Low   Hard
## 2  Low   Hard
## 3  Low Medium
## 4 High Medium
## 5 High   Soft
## 6 High   Soft
```

1.2 Recoding variables

Recoding involves creating new values of a variable conditional on the existing values of the same and/or other variables. For example, you may want to

- Change a continuous variable into a set of categories
- Replace miscoded values with correct values
- Create a pass/fail variable based on a set of cutoff scores

Assuming that you want to recode the ages of the managers in our leadership dataset from the continuous variable age to the categorical variable agecat (Young, Middle Aged, Elder, Missing Value).

```r
manager <- c(1, 2, 3, 4, 5)
date <- c("10/24/08", "10/28/08", "10/1/08", "10/12/08", "5/1/09")
country <- c("US", "US", "UK", "UK", "UK")
gender <- c("M", "F", "F", "M", "F")
age <- c(32, 56, 25, 89, 99)
leadership <- data.frame(manager, date, country, gender, age)
leadership

##   manager     date country gender age
## 1       1 10/24/08      US      M  32
## 2       2 10/28/08      US      F  56
## 3       3  10/1/08      UK      F  25
## 4       4 10/12/08      UK      M  89
## 5       5   5/1/09      UK      F  99
```

The statement variable[condition] <- expression will only make the assignment when condition is TRUE.

```r
leadership$age[leadership$age == 99] <- NA
leadership$agecat[leadership$age > 75] <- "Elder"
leadership$agecat[leadership$age>=55 & leadership$age <= 75] <- "Middle Aged"
leadership$agecat[leadership$age < 55] <- "Young"
leadership
```

```
##   manager     date country gender age       agecat
## 1       1 10/24/08      US      M  32        Young
## 2       2 10/28/08      US      F  56 Middle Aged
## 3       3  10/1/08      UK      F  25        Young
## 4       4 10/12/08      UK      M  89        Elder
## 5       5   5/1/09      UK      F  NA         <NA>
```

This code can be written more compactly as

```
leadership <- within(leadership,{
                     agecat <- NA agecat[age > 75] <- "Elder"
                     agecat[age >= 55 & age <= 75] <- "Middle Aged"
                     agecat[age < 55] <- "Young"})
```

The within() function is similar to the with() function, but allows you to modify the data frame. First, the variable agecat variable is created and set to missing for each row of the data frame. Then the remaining statements within the braces are executed in order.
Practice: Turn agecat into an ordered factor.

1.3 Renaming variables

If you're not happy with your variable names, you can change them interactively or programmatically.

```
fix(leadership)
```

you can rename variables via the names() function .

```
names(leadership)[2] <- "testDate"
leadership
```

In a similar fashion,

```
names(leadership)[1:2] <- c("ID", "Date")
leadership
```

1.4 Missing values

1.4.1 Check for missing values

In a project of any size, data is likely to be incomplete because of missed questions, faulty equipment, or improperly coded data. In R, missing values are represented by the symbol NA (not available) . Impossible values (for example, dividing by 0) are represented by the symbol NaN (not a number) . he function is.na() allows you to test for the presence of missing values.

```
y <- c(1, 2, 3, NA)
is.na(y)

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

Notice how the is.na() function works on an object. It returns an object of the same size, with the entries replaced by TRUE if the element is a missing value, and FALSE if the element is not a missing value.

NOTE: Missing values are considered noncomparable, even to themselves. This means that you can't use comparison operators to test for the presence of missing values. For example, the logical test myvar == NA is never TRUE. Instead, you have to use missing values functions, like those in this section, to identify the missing values in R data objects.

1.4.2 Excluding missing values

Once you've identified the missing values, you need to eliminate them in some way before analyzing your data further. The reason is that arithmetic expressions and functions that contain missing values yield missing values.

```
x <- c(1, 2, NA, 3)
y <- x[1] + x[2] + x[3] + x[4]
print(y)

## [1] NA

y <- sum(x, na.rm=TRUE)
print(y)

## [1] 6
```

A common task is to remove missing values

```
bad <- is.na(x)
x[!bad]

## [1] 1 2 3
```

What if there are multiple things and you want to take the subset with no missing values?

```
x <- c(1, 2, NA, 4, NA, 5)
y <- c("a", "b", "c", "d", NA, "f")
good <- complete.cases(x, y)
good

## [1]  TRUE  TRUE FALSE  TRUE FALSE  TRUE

x[good]

## [1] 1 2 4 5

y[good]

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

You can remove any observation with missing data by using the na.omit() function . na.omit() deletes any rows with missing data.

```
leadership

##   manager     date country gender age      agecat
## 1       1 10/24/08      US      M  32       Young
## 2       2 10/28/08      US      F  56 Middle Aged
```

```
## 3        3  10/1/08      UK      F  25        Young
## 4        4 10/12/08      UK      M  89        Elder
## 5        5   5/1/09      UK      F  NA         <NA>

good <- complete.cases(leadership)
good

## [1]  TRUE  TRUE  TRUE  TRUE FALSE

newdata <- leadership[good, ]
newdata

##   manager     date country gender age      agecat
## 1       1 10/24/08      US      M  32        Young
## 2       2 10/28/08      US      F  56 Middle Aged
## 3       3  10/1/08      UK      F  25        Young
## 4       4 10/12/08      UK      M  89        Elder

newdata <- na.omit(leadership)
newdata

##   manager     date country gender age      agecat
## 1       1 10/24/08      US      M  32        Young
## 2       2 10/28/08      US      F  56 Middle Aged
## 3       3  10/1/08      UK      F  25        Young
## 4       4 10/12/08      UK      M  89        Elder
```

Deleting all observations with missing data (called listwise deletion) is one of several methods of handling incomplete datasets. If there are only a few missing values or they're concentrated in a small number of observations, listwise deletion can provide a good solution to the missing values problem. But if missing values are spread throughout the data, or there's a great deal of missing data in a small number of variables, listwise deletion can exclude a substantial percentage of your data.

1.4.3 Using NULL

In statistical data sets, we often encounter missing data, which we represent in R with the value NA. NULL, on the other hand, represents that the value in question simply doesn't exist, rather than being existent but unknown.

One use of NULL is to build up vectors in loops, in which each iteration adds another element to the vector. In this simple example, we build up a vector of even numbers:

```
z <- NULL
for (i in 1:10) if (i %%2 == 0) z <- c(z,i)
z

## [1]  2  4  6  8 10
```

Recall that %% is the modulo operator, giving remainders upon division. For example, 13 %% 4 is 1, as the remainder of dividing 13 by 4 is 1.

But the point here is to demonstrate the difference between NA and NULL. If we were to use NA instead of NULL in the preceding example, we would pick up an unwanted NA:

```r
z <- NA
for (i in 1:10) if (i %%2 == 0) z <- c(z,i)
z

## [1] NA  2  4  6  8 10
```

NULL values really are counted as nonexistent, as you can see here:

```r
u <- NULL
length(u)

## [1] 0

v <- NA
length(v)

## [1] 1
```

NULL is a special R object with no mode.

1.5 Date values

Dates and Times in R

R has developed a special representation of dates and times

- Dates are represented by the Date class
- Times are represented by the POSIXct or the POSIXlt class
- Dates are stored internally as the number of days since 1970-01-01
- Times are stored internally as the number of seconds since 1970-01-01

Dates are represented by the Date class and can be coerced from a character string using the as.Date() function.

```r
x <- as.Date("1970-01-01")
x

## [1] "1970-01-01"

unclass(x)

## [1] 0

unclass(as.Date("1970-01-02"))

## [1] 1
```

Times are represented using the POSIXct or the POSIXlt class. "ct" can stand for calendar time, it stores the number of seconds since the origin. "lt", or local time, keeps the date as a list of time attributes (such as "hour" and "mon").

- POSIXct is just a very large integer under the hood; it use a useful class when you want to store times in something like a data frame

- POSIXlt is a list underneath and it stores a bunch of other useful information like the day of the week, day of the year, month, day of the month

```
x <- Sys.time()
x

## [1] "2016-04-12 16:44:27 CST"

p <- as.POSIXlt(x)
names(unclass(p))

##  [1] "sec"    "min"    "hour"   "mday"   "mon"    "year"   "wday"
##  [8] "yday"   "isdst"  "zone"   "gmtoff"

p$sec

## [1] 27.84637
```

```
x <- Sys.time()
x

## [1] "2016-04-12 16:44:27 CST"

unclass(x)

## [1] 1460450668

#x$sec
p <- as.POSIXlt(x)
p$sec

## [1] 27.88038
```

Two functions are especially useful for time-stamping data . Sys.Date() returns today's date and date() returns the current date and time.

```
date()

## [1] "Tue Apr 12 16:44:27 2016"

today <- Sys.Date()
format(today, format="%m %d %Y")

## [1] "04 12 2016"

format(today, format="%y")

## [1] "16"
```

When R stores dates internally, they're represented as the number of days since January 1, 1970, with negative values for earlier dates. That means you can perform arithmetic operations on them.

```
startdate <- as.Date("2004-02-13")
enddate <- as.Date("2011-01-22")
days <- enddate - startdate
days

## Time difference of 2535 days
```

You can also use the function difftime() to calculate a time interval and express it as seconds, minutes, hours, days, or weeks.

```
today <- Sys.Date()
dob <- as.Date("1984-7-12")
difftime(today, dob, units="weeks")

## Time difference of 1656.714 weeks
```

The default format for inputting dates is yyyy-mm-dd.

```
leadership$date

## [1] 10/24/08 10/28/08 10/1/08  10/12/08 5/1/09
## Levels: 10/1/08 10/12/08 10/24/08 10/28/08 5/1/09

myformat <- "%m/%d/%y"
leadership$date <- as.Date(leadership$date, myformat)
leadership$date

## [1] "2008-10-24" "2008-10-28" "2008-10-01" "2008-10-12" "2009-05-01"
```

Although less commonly used, you can also convert date variables to character variables. Date values can be converted to character values using the as.character() function. To learn more about converting character data to dates, take a look at help(as. Date) and help(strftime) . To learn more about formatting dates and times, see help(ISOdatetime) .

Practice: my 100 days birthday.which day of the week was I born???

1.6 Sorting data

Sometimes, viewing a dataset in a sorted order can tell you quite a bit about the data.

```
newdata <- leadership[order(leadership$age),]
newdata

##   manager       date country gender age      agecat
## 3       3 2008-10-01      UK      F  25       Young
## 1       1 2008-10-24      US      M  32       Young
## 2       2 2008-10-28      US      F  56 Middle Aged
## 4       4 2008-10-12      UK      M  89       Elder
## 5       5 2009-05-01      UK      F  NA        <NA>

attach(leadership)

## The following objects are masked _by_ .GlobalEnv:
##
##    age, country, date, gender, manager
```

```
newdata <-leadership[order(gender, -age),]
detach(leadership)
newdata

##   manager       date country gender age      agecat
## 5       5 2009-05-01      UK      F  NA        <NA>
## 2       2 2008-10-28      US      F  56 Middle Aged
## 3       3 2008-10-01      UK      F  25       Young
## 4       4 2008-10-12      UK      M  89       Elder
## 1       1 2008-10-24      US      M  32       Young
```

sorts the rows by gender, and then from oldest to youngest manager within each gender.

1.7 Subsetting datasets

1.7.1 Selecting (keeping) variables

It's a common practice to create a new dataset from a limited number of variables chosen from a larger dataset.

```
newdata <- leadership[, c(1,5)]
newdata

##   manager age
## 1       1  32
## 2       2  56
## 3       3  25
## 4       4  89
## 5       5  NA

myvars <- c("manager", "age")
newdata <-leadership[myvars]
newdata

##   manager age
## 1       1  32
## 2       2  56
## 3       3  25
## 4       4  89
## 5       5  NA
```

1.7.2 Excluding (dropping) variables

There are many reasons to exclude variables. For example, if a variable has several missing values, you may want to drop it prior to further analyses.

```
names(leadership)

## [1] "manager" "date"    "country" "gender"  "age"     "agecat"

myvars <- names(leadership) %in% c("manager", "age")
myvars
```

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

newdata <- leadership[!myvars]
newdata

##         date country gender      agecat
## 1 2008-10-24      US      M       Young
## 2 2008-10-28      US      F Middle Aged
## 3 2008-10-01      UK      F       Young
## 4 2008-10-12      UK      M       Elder
## 5 2009-05-01      UK      F        <NA>
```

1. names(leadership) produces a character vector containing the variable names.
2. names(leadership) %in% c("manager", "age") returns a logical vector with TRUE for each element in names(leadership)that matches manager or age and FALSE otherwise.
3. The not (!) operator reverses the logical values.
4. leadership[!myvars] selects columns with TRUE logical values, so manager and age are excluded.

Knowing that manager and age are the 1th and 5th variable, you could exclude them with the statement

```
newdata <- leadership[c(-1,-5)]
newdata

##         date country gender      agecat
## 1 2008-10-24      US      M       Young
## 2 2008-10-28      US      F Middle Aged
## 3 2008-10-01      UK      F       Young
## 4 2008-10-12      UK      M       Elder
## 5 2009-05-01      UK      F        <NA>
```

This works because prepending a column index with a minus sign (-) excludes that column.

1.7.3 Selecting observations

Selecting or excluding observations (rows) is typically a key aspect of successful data preparation and analysis.

```
leadership

##   manager       date country gender age      agecat
## 1       1 2008-10-24      US      M  32       Young
## 2       2 2008-10-28      US      F  56 Middle Aged
## 3       3 2008-10-01      UK      F  25       Young
## 4       4 2008-10-12      UK      M  89       Elder
## 5       5 2009-05-01      UK      F  NA        <NA>

#which(leadership$gender=="M" & leadership$age > 30)
newdata <- leadership[which(leadership$gender=="M" & leadership$age > 30),]
newdata

##   manager       date country gender age agecat
## 1       1 2008-10-24      US      M  32  Young
## 4       4 2008-10-12      UK      M  89  Elder
```

1. The logical comparison leadership$gender=="M".
2. The logical comparison leadership$age > 30.
3. The logical comparison (1) and (2), produces the vector of results.
4. The function which() gives the indices of a vector that are TRUE.
5. Leadership[c(.,.),] selects the observations from the data frame.

```
leadership$date <- as.Date(leadership$date)
leadership$date

## [1] "2008-10-24" "2008-10-28" "2008-10-01" "2008-10-12" "2009-05-01"

startdate <- as.Date("2009-01-01")
enddate <- as.Date("2009-10-31")
newdata <- leadership[which(leadership$date >= startdate & leadership$date <= enddate),]
newdata

##   manager       date country gender age agecat
## 5       5 2009-05-01      UK      F  NA   <NA>
```

The subset function is probably the easiest way to select variables and observations.

```
newdata <- subset(leadership, age >= 35 | age < 80, select=c(manager,age))
newdata

##   manager age
## 1       1  32
## 2       2  56
## 3       3  25
## 4       4  89

newdata <- subset(leadership, gender=="M" & age > 50, select=manager:age)
newdata

##   manager       date country gender age
## 4       4 2008-10-12      UK      M  89
```

1.7.4 Random samples

Sampling from larger datasets is a common practice in data mining and machine learning. For example, you may want to select two random samples, creating a predictive model from one and validating its effectiveness on the other. The sample() function enables you to take a random sample (with or without replacement) of size n from a dataset.

```
mysample <- leadership[sample(1:nrow(leadership), 3, replace=FALSE),]
mysample

##   manager       date country gender age      agecat
## 2       2 2008-10-28      US      F  56 Middle Aged
## 3       3 2008-10-01      UK      F  25       Young
## 4       4 2008-10-12      UK      M  89       Elder
```

The first argument to the sample() function is a vector of elements to choose from. Here, the vector is 1 to the number of observations in the data frame. The second argument is the number of elements to be selected, and the third argument indicates sampling without replacement. The sample() function returns the randomly sampled elements, which are then used to select rows from the data frame.

R has extensive facilities for sampling, including drawing and calibrating survey samples (see the sampling package) and analyzing complex survey data (see the survey package).

2 Aggregation and restructuring

2.1 Aggregation

To demonstrate aggregate we once again turn to the diamonds data in ggplot2.

```
require(ggplot2)

## Loading required package:  ggplot2

data(diamonds)
head(diamonds)

##   carat       cut color clarity depth table price    x    y    z
## 1  0.23     Ideal     E     SI2  61.5    55   326 3.95 3.98 2.43
## 2  0.21   Premium     E     SI1  59.8    61   326 3.89 3.84 2.31
## 3  0.23      Good     E     VS1  56.9    65   327 4.05 4.07 2.31
## 4  0.29   Premium     I     VS2  62.4    58   334 4.20 4.23 2.63
## 5  0.31      Good     J     SI2  63.3    58   335 4.34 4.35 2.75
## 6  0.24 Very Good     J    VVS2  62.8    57   336 3.94 3.96 2.48
```

We calculate the average price for each type of cut: Fair, Good, Very Good, Premium and Ideal.

```
aggregate(price ~ cut, diamonds, mean)

##         cut    price
## 1      Fair 4358.758
## 2      Good 3928.864
## 3 Very Good 3981.760
## 4   Premium 4584.258
## 5     Ideal 3457.542
```

To group the data by more than one variable, add the additional variable to the right side of the formula separating it with a plus sign (+).

```
aggregate(price ~ cut + color, diamonds, mean)

##          cut color    price
## 1       Fair     D 4291.061
## 2       Good     D 3405.382
## 3  Very Good     D 3470.467
## 4    Premium     D 3631.293
## 5      Ideal     D 2629.095
## 6       Fair     E 3682.312
## 7       Good     E 3423.644
## 8  Very Good     E 3214.652
```

```
## 9    Premium     E 3538.914
## 10     Ideal     E 2597.550
## 11      Fair     F 3827.003
## 12      Good     F 3495.750
## 13 Very Good     F 3778.820
## 14   Premium     F 4324.890
## 15     Ideal     F 3374.939
## 16      Fair     G 4239.255
## 17      Good     G 4123.482
## 18 Very Good     G 3872.754
## 19   Premium     G 4500.742
## 20     Ideal     G 3720.706
## 21      Fair     H 5135.683
## 22      Good     H 4276.255
## 23 Very Good     H 4535.390
## 24   Premium     H 5216.707
## 25     Ideal     H 3889.335
## 26      Fair     I 4685.446
## 27      Good     I 5078.533
## 28 Very Good     I 5255.880
## 29   Premium     I 5946.181
## 30     Ideal     I 4451.970
## 31      Fair     J 4975.655
## 32      Good     J 4574.173
## 33 Very Good     J 5103.513
## 34   Premium     J 6294.592
## 35     Ideal     J 4918.186
```

To aggregate two variables (for now we still just group by cut), they must be combined using cbind on the left side of the formula.

```
aggregate(cbind(price, carat) ~ cut, diamonds, mean)

##         cut    price     carat
## 1      Fair 4358.758 1.0461366
## 2      Good 3928.864 0.8491847
## 3 Very Good 3981.760 0.8063814
## 4   Premium 4584.258 0.8919549
## 5     Ideal 3457.542 0.7028370
```

This finds the mean of both price and carat for each value of cut. It is important to note that only one function can be supplied, and hence applied, to the variables. Of course, multiple variables can be supplied to both the left and right sides at the same time.

```
aggregate(cbind(price, carat) ~ cut + color, diamonds, mean)

##          cut color    price     carat
## 1       Fair     D 4291.061 0.9201227
## 2       Good     D 3405.382 0.7445166
## 3  Very Good     D 3470.467 0.6964243
## 4    Premium     D 3631.293 0.7215471
## 5      Ideal     D 2629.095 0.5657657
```

```
## 6       Fair     E 3682.312 0.8566071
## 7       Good     E 3423.644 0.7451340
## 8  Very Good     E 3214.652 0.6763167
## 9    Premium     E 3538.914 0.7177450
## 10     Ideal     E 2597.550 0.5784012
## 11      Fair     F 3827.003 0.9047115
## 12      Good     F 3495.750 0.7759296
## 13 Very Good     F 3778.820 0.7409612
## 14   Premium     F 4324.890 0.8270356
## 15     Ideal     F 3374.939 0.6558285
## 16      Fair     G 4239.255 1.0238217
## 17      Good     G 4123.482 0.8508955
## 18 Very Good     G 3872.754 0.7667986
## 19   Premium     G 4500.742 0.8414877
## 20     Ideal     G 3720.706 0.7007146
## 21      Fair     H 5135.683 1.2191749
## 22      Good     H 4276.255 0.9147293
## 23 Very Good     H 4535.390 0.9159485
## 24   Premium     H 5216.707 1.0164492
## 25     Ideal     H 3889.335 0.7995249
## 26      Fair     I 4685.446 1.1980571
## 27      Good     I 5078.533 1.0572222
## 28 Very Good     I 5255.880 1.0469518
## 29   Premium     I 5946.181 1.1449370
## 30     Ideal     I 4451.970 0.9130291
## 31      Fair     J 4975.655 1.3411765
## 32      Good     J 4574.173 1.0995440
## 33 Very Good     J 5103.513 1.1332153
## 34   Premium     J 6294.592 1.2930941
## 35     Ideal     J 4918.186 1.0635937
```

2.2 Merging datasets

If your data exist in multiple locations, you'll need to combine them before moving forward. This section shows you how to add columns (variables) and rows (observations) to a data frame.

2.2.1 Adding columns

To merge two data frames (datasets) horizontally, you use the merge() function. In most cases, two data frames are joined by one or more common key variables (that is an inner join).

```
total <- merge(dataframeA, dataframeB, by="ID")
```

to merges dataframeA and dataframeB by ID.

```
lifeforms <- read.table("C:/Users/XXXHHF/Documents/R/workfile/therbook/lifeforms.txt",header=T)
flowering <- read.table("C:/Users/XXXHHF/Documents/R/workfile/therbook/fltimes.txt",header=T)
lifeforms

##    Genus     species lifeform
## 1   Acer platanoides     tree
## 2   Acer    palmatum     tree
```

```
## 3  Ajuga     reptans     herb
## 4 Conyza sumatrensis   annual
## 5 Lamium       album     herb
```

```
flowering
```

```
##       Genus       species flowering
## 1      Acer   platanoides       May
## 2     Ajuga       reptans      June
## 3  Brassica         napus     April
## 4 Chamerion angustifolium      July
## 5    Conyza     bilbaoana    August
## 6    Lamium         album   January
```

```
merge(flowering,lifeforms)
```

```
##    Genus     species flowering lifeform
## 1   Acer platanoides       May     tree
## 2  Ajuga     reptans      June     herb
## 3 Lamium       album   January     herb
```

```
merge(flowering,lifeforms,all=T)
```

```
##       Genus       species flowering lifeform
## 1      Acer   platanoides       May     tree
## 2      Acer      palmatum      <NA>     tree
## 3     Ajuga       reptans      June     herb
## 4  Brassica         napus     April     <NA>
## 5 Chamerion angustifolium      July     <NA>
## 6    Conyza     bilbaoana    August     <NA>
## 7    Conyza   sumatrensis      <NA>   annual
## 8    Lamium         album   January     herb
```

If you're joining two matrices or data frames horizontally and don't need to specify a common key, you can use

```
total <- cbind(A, B)
```

This function will horizontally concatenate the objects A and B. For the function to work properly, each object has to have the same number of rows and be sorted in the same order.

```
m <- matrix(c(45, 23, 66, 77, 33, 44, 56, 12), 2, 4, byrow = T)
m
```

```
##      [,1] [,2] [,3] [,4]
## [1,]   45   23   66   77
## [2,]   33   44   56   12
```

```
cbind(c(4, 76), m[, 4])
```

```
##      [,1] [,2]
## [1,]    4   77
## [2,]   76   12
```

Another Example

I have prepared data originally made available as part of the USAID Open Government initiative. The data have been chopped into eight separate files so that they can be joined together.

```
Aid_90s<-read.delim("US_Foreign_Aid_90s.csv",sep=",")
Aid_00s<-read.delim("US_Foreign_Aid_00s.csv",sep=",")
head(Aid_90s)
head(Aid_00s)
Aid90s00s <- merge(x=Aid_90s, y=Aid_00s,
                   by.x=c("Country.Name", "Program.Name"),
                   by.y=c("Country.Name", "Program.Name"))
head(Aid90s00s)
```

2.2.2 plyr join

Returning to Hadley Wickham's plyr package, we see it includes a join function, which works similarly to merge but is much faster. The biggest drawback, though, is that the key column(s) in each table must have the same name. We use the same data used previously to illustrate.

```
require(plyr)
Aid90s00sJoin <- join(x = Aid_90s, y = Aid_00s, by = c("Country.Name",
                                                        "Program.Name"))
head(Aid90s00sJoin)
```

2.2.3 Adding rows

To join two data frames (datasets) vertically, use the rbind() function

`total <- rbind(dataframeA, dataframeB)`

The two data frames must have the same variables, but they don't have to be in the same order. If dataframeA has variables that dataframeB doesn't, then before joining them do one of the following:

- Delete the extra variables in dataframeA
- Create the additional variables in dataframeB and set them to NA (missing)

Vertical concatenation is typically used to add observations to a data frame.

```
m2 <- matrix(rep(10, 12), 3, 4)
m2

##      [,1] [,2] [,3] [,4]
## [1,]   10   10   10   10
## [2,]   10   10   10   10
## [3,]   10   10   10   10

m3 <- rbind(m[1, ], m2[3, ])
m3

##      [,1] [,2] [,3] [,4]
## [1,]   45   23   66   77
## [2,]   10   10   10   10
```

2.2.4 denominate

You can also give names to the columns and rows of matrices, using the functions colnames() and rownames(), respectively.

```
colnames(m3) <- c("1qrt", "2qrt", "3qrt", "4qrt")
rownames(m3) <- c("store1", "store2")
m3

##        1qrt 2qrt 3qrt 4qrt
## store1   45   23   66   77
## store2   10   10   10   10
```

2.2.5 Transpose

The transpose (reversing rows and columns) is perhaps the simplest method of reshaping a dataset. Use the t() function to transpose a matrix or a data frame. In the latter case, row names become variable (column) names.

```
cars <- mtcars[1:5,1:4]
cars

##                    mpg cyl disp  hp
## Mazda RX4         21.0   6  160 110
## Mazda RX4 Wag     21.0   6  160 110
## Datsun 710        22.8   4  108  93
## Hornet 4 Drive    21.4   6  258 110
## Hornet Sportabout 18.7   8  360 175

t(cars)

##      Mazda RX4 Mazda RX4 Wag Datsun 710 Hornet 4 Drive Hornet Sportabout
## mpg         21            21       22.8           21.4              18.7
## cyl          6             6        4.0            6.0               8.0
## disp       160           160      108.0          258.0             360.0
## hp         110           110       93.0          110.0             175.0
```

2.3 The reshape package

The reshape package is a tremendously versatile approach to both restructuring and aggregating datasets.

2.3.1 Melting

When you melt a dataset, you restructure it into a format where each measured variable is in its own row, along with the ID variables needed to uniquely identify it.

```
ID<-c(1,1,2,2) #ID<-c(1,1,1,2,2)
Time<-c(1,2,1,2) #Time<-c(1,1,2,1,2)
X1<-c(5,3,6,2) #X1<-c(5,5,3,6,2)
X2<-c(6,5,1,4) #X2<-c(6,6,5,1,4)
mydata<-data.frame(ID,Time,X1,X2)
```

```
#mydata<-cbind(ID,Time,X1,X2) different results
mydata

##   ID Time X1 X2
## 1  1    1  5  6
## 2  1    2  3  5
## 3  2    1  6  1
## 4  2    2  2  4

library(reshape)
md <- melt(mydata, id=(c("ID", "Time")))
md

##   ID Time variable value
## 1  1    1       X1     5
## 2  1    2       X1     3
## 3  2    1       X1     6
## 4  2    2       X1     2
## 5  1    1       X2     6
## 6  1    2       X2     5
## 7  2    1       X2     1
## 8  2    2       X2     4
```

Note that you must specify the variables needed to uniquely identify each measurement (ID and Time) and that the variable indicating the measurement variable names (X1 or X2) is created for you automatically.

2.3.2 Casting

The cast() function starts with melted data and reshapes it using a formula that you provide and an (optional) function used to aggregate the data. The format is newdata <- cast(md, formula, FUN).

```
cast(md, ID+variable~Time)

##   ID variable 1 2
## 1  1       X1 5 3
## 2  1       X2 6 5
## 3  2       X1 6 2
## 4  2       X2 1 4

cast(md, ID~variable+Time)

##   ID X1_1 X1_2 X2_1 X2_2
## 1  1    5    3    6    5
## 2  2    6    2    1    4

cast(md, Time~variable, mean)

##   Time  X1  X2
## 1    1 5.5 3.5
## 2    2 2.5 4.5
```

As you can see, the flexibility provided by the melt() and cast() functions is amazing. There are many times when you'll have to reshape or aggregate your data prior to analysis.

2.3.3 Convert from wide to long (tall) format

Sometimes data are available in a dierent shape than that required for analysis.

```
#library(reshape)
country<-c("China","USA","Japan")
GDP2000<-c(5000,6000,7000)
GDP2005<-c(5500,6500,7500)
GDP2010<-c(5010,6010,7001)
developed<-as.factor(c(0,1,1))
Data<-data.frame(country,developed,GDP2000,GDP2005,GDP2010)
Data
```

```
##   country developed GDP2000 GDP2005 GDP2010
## 1   China         0    5000    5500    5010
## 2     USA         1    6000    6500    6010
## 3   Japan         1    7000    7500    7001
```

```
long<-reshape(Data,idvar="country",varying=list(names(Data)[3:5]),
     v.names="GDP",timevar="year",times=c(2000,2005,2010),direction="long")
long
```

```
##            country developed year  GDP
## China.2000   China         0 2000 5000
## USA.2000       USA         1 2000 6000
## Japan.2000   Japan         1 2000 7000
## China.2005   China         0 2005 5500
## USA.2005       USA         1 2005 6500
## Japan.2005   Japan         1 2005 7500
## China.2010   China         0 2010 5010
## USA.2010       USA         1 2010 6010
## Japan.2010   Japan         1 2010 7001
```

2.3.4 Convert from long (tall) to wide format

```
wide <- reshape(long, v.names="GDP", idvar="country", timevar="year", direction="wide")
wide
```

```
##            country developed GDP.2000 GDP.2005 GDP.2010
## China.2000   China         0     5000     5500     5010
## USA.2000       USA         1     6000     6500     6010
## Japan.2000   Japan         1     7000     7500     7001
```

3 Manipulating Strings

3.1 Paste

The first function new R users reach for when putting together strings is paste. This function takes a series of strings, or expressions that evaluate to strings, and puts them together into one string.

```r
paste("Hello",c("Hello", "Hey", "Howdy"), c("Jared", "Bob", "David"))

## [1] "Hello Hello Jared" "Hello Hey Bob"     "Hello Howdy David"

paste("Hello", "Jared", "and others", sep = "/")

## [1] "Hello/Jared/and others"
```

3.2 Sprintf

While paste is convenient for putting together short bits of text, it can become unwieldy when piecing together long pieces of text, such as when inserting a number of variables into a long piece of text. For instance, we might have a lengthy sentence that has a few spots that require the insertion of special variables. An example is "Hello Jared, your party of eight will be seated in 25 minutes" where "Jared," "eight" and "25" could be replaced with other information. Reforming this with paste can make reading the line in code difficult. To start, we make some variables to hold the information.

```r
person <- "Jared"
partySize <- "eight"
waitTime <- 25
paste("Hello ", person, ", your party of ", partySize,
      " will be seated in ", waitTime, " minutes.", sep="")

## [1] "Hello Jared, your party of eight will be seated in 25 minutes."

sprintf("Hello %s, your party of %s will be seated in %s minutes",
      person, partySize, waitTime)

## [1] "Hello Jared, your party of eight will be seated in 25 minutes"

sprintf("Hello %s, your party of %s will be seated in %s minutes",
      c("Jared", "Bob"), c("eight", 16, "four", 10), waitTime)

## [1] "Hello Jared, your party of eight will be seated in 25 minutes"
## [2] "Hello Bob, your party of 16 will be seated in 25 minutes"
## [3] "Hello Jared, your party of four will be seated in 25 minutes"
## [4] "Hello Bob, your party of 10 will be seated in 25 minutes"
```

3.3 Extracting Text

Often text needs to be ripped apart to be made useful, and while R has a number of functions for doing so, the stringr package is much easier to use. We download a table of United States presidents from Wikipedia. Examining it more closely, we see that the last few rows contain information we do not want, so we keep only the first 64 rows.

```r
require(stringr)
load("presidents.rdata")
#theURL <- "http://www.loc.gov/rr/print/list/057_chron.html"
#presidents <- readHTMLTable(theURL, which=3, as.data.frame=TRUE,
#                             skip.rows=1, header=TRUE,
#                             stringsAsFactors=FALSE)
```

```
head(presidents)
tail(presidents$YEAR)
presidents <- presidents[1:64, ]
```

To start, we create two new columns, one for the beginning of the term and one for the end of the term. To do this we need to split the Year column on the hyphen (-). The stringr package has the str split function that splits a string based on some value. It returns a list with an element for each element of the input vector. Each of these elements has as many elements as necessary for the split, in this case either two (a start and stop year) or one (when the president served less than one year).

```
# split the string
yearList <- str_split(string = presidents$YEAR, pattern = "-")
head(yearList)
# combine them into one matrix
yearMatrix <- data.frame(Reduce(rbind, yearList))
#a=c(12,25,3,8)#48
head(yearMatrix)
# give the columns good names
names(yearMatrix) <- c("Start", "Stop")
# bind the new columns onto the data.frame
presidents <- cbind(presidents, yearMatrix)
# convert the start and stop columns into numeric
presidents$Start <- as.numeric(as.character(presidents$Start))
presidents$Stop <- as.numeric(as.character(presidents$Stop))
head(presidents)
tail(presidents)
str_sub(string = presidents$PRESIDENT, start = 1, end = 3)
presidents[str_sub(string = presidents$Start, start = 4,
                   end = 4) == 1, c("YEAR", "PRESIDENT", "Start", "Stop")]
```

It is possible to select specified characters from text using str_sub.

```
# get the first 3 characters
str_sub(string = presidents$PRESIDENT, start = 1, end = 3)
```

This is good for finding a president whose term started in a year ending in 1, which means he got elected in a year ending in 0, a preponderance of which ones died in office.

```
presidents[str_sub(string = presidents$Start, start = 4,
                   end = 4) == 1, c("YEAR", "PRESIDENT", "Start", "Stop")]
```

3.4 Regular Expressions

Sifting through text often requires searching for patterns, and usually these patterns have to be general and flexible. This is where regular expressions are very useful. We will not make an exhaustive lesson of regular expressions but will illustrate how to use them within R.

Let's say we want to find any president with "John" in his name, either first or last. Since we do not know where in the name "John" would occur, we cannot simply use str_sub. Instead we use str_detect.

```r
johnPos <- str_detect(string = presidents$PRESIDENT, pattern = "John")
presidents[johnPos, c("YEAR", "PRESIDENT", "Start", "Stop")]
```

This found John Adams, John Quincy Adams, John Tyler, Andrew Johnson, John F. Kennedy and Lyndon B. Johnson. Note that regular expressions are case sensitive, so to ignore case we have to put the pattern in ignore.case

```r
badSearch <- str_detect(presidents$PRESIDENT, "john")
goodSearch <- str_detect(presidents$PRESIDENT, ignore.case("John"))
sum(badSearch)
sum(goodSearch)
```

Another Example

To show off some more interesting regular expressions we will make use of yet another table from Wikipedia, the list of United States wars. This dataset holds the starting and stopping dates of the wars. Sometimes it has just years, sometimes it also includes months and possibly days. There are instances where it has only one year. Because of this, it is a good dataset to comb through with various text functions.

```r
load("warTimes.rdata")
#con <- url("http://www.jaredlander.com/data/warTimes.rdata")
#load(con)
#close(con)
head(warTimes, 10)
class(warTimes)
```

We want to create a new column that contains information for the start of the war. To get at this information we need to split the Time column.

```r
warTimes[str_detect(string = warTimes, pattern = "-")]
```

So when we are splitting our string, we need to search for either "ACAEA" or "-." In str split the pattern argument can take a regular expression. In this case it will be "(ACAEA)|-," which tells the engine to search for either "(ACAEA)" or (denoted by the vertical pipe) "-" in the string. To avoid the instance, seen before, where the hyphen is used in "mid-July" we set the argument n to 2 so it returns at most only two pieces for each element of the input vector.

```r
theTimes <- str_split(string = warTimes, pattern = "(ACAEA)|-", n = 2)
head(theTimes)
```

Seeing that this worked for the first few entries, we also check on the two instances where a hyphen was the separator.

```r
which(str_detect(string = warTimes, pattern = "-"))
theTimes[[147]]
theTimes[[150]]
class(theTimes)
```

For our purposes we only care about the start date of the wars, so we need to build a function that extracts the first (in some cases only) element of each vector in the list.

```r
theStart <- sapply(theTimes, FUN = function(x) x[1])
head(theStart)
```

The original text sometimes had spaces around the separators and sometimes did not, meaning that some of our text has trailing white spaces. The easiest way to get rid of them is with the str_trim function.

```r
theStart <- str_trim(theStart)
head(theStart)
```

To extract the word "January" wherever it might occur, use str extract. In places where it is not found will be NA.

```r
# pull out 'January' anywhere it's found, otherwise return NA
str_extract(string = theStart, pattern = "January")
# just return elements where 'January' was detected
theStart[str_detect(string = theStart, pattern = "January")]
```

To extract the year, we search for an occurrence of four numbers together. Because we do not know specific numbers, we have to use a pattern. In a regular expression search, "[0-9]" searches for any number. We use "[0-9][0-9][0-9][0-9]" to search for four consecutive numbers.

```r
# get incidents of 4 numeric digits in a row
head(str_extract(string = theStart, "[0-9][0-9][0-9][0-9]"), 20)
```

Writing "[0-9]" repeatedly is inefficient, especially when searching for many occurences of a number. Putting "4" in curly braces after "[0-9]" causes the engine to search for any set of four numbers.

```r
# a smarter way to search for four numbers
head(str_extract(string = theStart, "[0-9]{4}"), 20)
```

Even writing "[0-9]" can be inefficient, so there is a shortcut to denote any integer. In most other languages the shortcut is "\d" but in R there needs to be two backslashes ("\\d").

```r
# "\\d" is a shortcut for "[0-9]"
head(str_extract(string = theStart, "\\d{4}"), 20)
```

The curly braces offer even more functionality: for instance, searching for a number one to three times.

```r
# this looks for any digit that occurs either once, twice or thrice
str_extract(string = theStart, "\\d{1,3}")
```

Regular expressions can search for text with anchors indicating the beginning of a line ("^") and the end of a line ("$").

```r
# extract 4 digits at the beginning of the text
head(str_extract(string = theStart, pattern = "^\\d{4}"), 30)
# extract 4 digits at the end of the text
head(str_extract(string = theStart, pattern = "\\d{4}$"), 30)
# extract 4 digits at the beginning AND the end of the text
head(str_extract(string = theStart, pattern = "^\\d{4}$"), 30)
```

Replacing text selectively is another powerful feature of regular expressions. We start by simply replacing numbers with a fixed value.

```
# replace all digits seen with "x"
# this means "7" -> "x" and "382" -> "xxx"
head(str_replace_all(string=theStart, pattern="\\d", replacement="x"),30)
# replace any strings of digits from 1 to 4 in length with "x"
# this means "7" -> "x" and "382" -> "x"
head(str_replace_all(string=theStart, pattern="\\d{1,4}", replacement="x"), 30)
```

About Time

```
Sys.time()
str_split(Sys.time()," ")
class(str_split(Sys.time()," "))
unlist(str_split(Sys.time()," "))[2]
```

3.5 Example: Download data from website

Download one Excel file

```
require(RCurl)
require(stringr)
#http://www.stats.govt.nz/browse_for_stats/government_finance/local_government/GovernmentFinanceStati
url<-"http://www.stats.govt.nz/~/media/Statistics/Browse%20for%20stats/GovernmentFinanceStatisticsLoc
temp<-getBinaryURL(url)
note <-file("hellodata.xls",open = "wb")
writeBin(temp,note)
close(note)
```

Download multi-files

```
url<-"http://rfunction.com/code/1202/"
html<-getURL(url)
html
files<-str_split(html,"<li><a href=\"")
class(files)
files<-unlist(files)
files<-str_extract_all(string = files, "^\\d{6}.R")
head(files)
class(files)
files<-unlist(files)
files
class(files)
baseurl<-"http://rfunction.com/code/1202/"
for(i in 1:length(files)) {
        url<-paste(baseurl,files[i],sep="")
```

```
        temp<-getBinaryURL(url)
        note <-file(paste(files[i],"txt",sep="."), open = "wb")
        writeBin(temp,note)
        close(note)
        Sys.sleep(2)
}
```