UNIT-II

Topics

Group Manipulation

Data Reshaping

Manipulating Strings

Doing Math & Simulation in R

Group Manipulation

1. Apply Family
2. Aggregate
3. plyr
4. Data.table

Apply Family

I. apply
II. lapply & sapply
III. mapply
IV. Other apply functions

I. apply

- It must be used on a matrix, meaning all of the elements must be of the same type whether they are character, numeric or logical.
- If used on some other object, such as data.frame, it will be converted to a matrix first.
- The first argument to apply is the object we are working with. The second argument is the margin to apply the function over, with 1 meaning to operate over the rows and 2 meaning is to operate over the columns. The third argument is the function we want to apply.
- Any following arguments will be passed on to that function. Apply will iterate over each row or coloumns of the matrix treating them as individual inputs to the first argument of the specified function.

```
>theMatrix<- matrix(1:9,nrow=3)
>apply(theMatrix,1,sum)
[1]  12  15  18
>apply(theMatrix,2,sum)
[1]  6  15  24
```

Note:

rowSums(), colSums() can also be used to get the above results

```
>rowSums(theMatrix)
[1]  12  15  18
>colSums(theMatrix)
[1]  6  15  24
```

- Let’s set an element of theMatrix to NA to see how we handle missing data using the na.rm argument and the use of additional arguments.

```
>theMatrix[2,1]<- NA
>apply(theMatrix,1,sum)
[1]  12  NA  18
>apply(theMatrix,1,sum, na.rm=TRUE)
[1]  12  13  18
>rowSums(theMatrix)
[1]  12  NA  18
>rowSums(theMatrix, na.rm=TRUE)
[1]  12   13   18
```

lapply and sapply

- lapply works by applying a function to each element of a list and returning the results as a list.

```
>theList<- list(A= matrix(1:9, 3), B=1:5, C=matrix(1:4, 2), D=2)
>lapply(theList, sum)
$A
[1]  45
$B
[1]  15
$C
[1]  10
$D
[1]  2
```

- Dealing with lists can be cumbersome, so to return the result of lapply as a vector instead, use sapply.

```
>sapply(theList,sum)
A   B   C   D
45 15  10  2
```

- Because a vector is technically a form of a list, lapply and sapply can also take a vector as their input.

```
>theNames<-c(“Jared”, “Deb”, “Paul”)
>lapply(theNames, nchar)
[[1]]
[1]  5
[[2]]
[1]  3
[[3]]
[1]  4
```

mapply:

- Which applies a function to each element of multiple lists.

```
>## build two lists
>firstList<-list(A=matrix(1:16,4), B=matrix(1:16,2),C=1:5)
>secondList<-list(A=matrix(1:16,4),B=matrix(1:16,2),C=15:1)
># test element by element if they are identical
>mapply(identical, firstList,secondList)
[1]  TRUE  TRUE  FALSE
>##  build a simple function that adds the number of rows (or
>##  length) of each corresponding element
>simpleFunc<-function(x,y)
{
NROW(x)+NROW(y)
}
>#apply the function to the two lists
>mapply(simpleFunc,firstList,secondList)
A  B  C
8  10  20
```

Other apply functions

- There are many other members of the apply family that either do not get used much or have been superseded by functions, in the plyr package. These include

I. tapply
II. rapply
III. eapply
IV. vapply
V. by

Aggregate

>require(ggplot2)

>data(diamonds)

>head(diamonds)

- We want to calculate the average price for each type of cut: Fair, Good, Very Good, Premium and Ideal. The first argument to aggregate is the formula specifying that price should be broken up (or group by in SQL terms) by cut. The second argument is the data to use, in this case diamonds. The third argument is the function to apply to each subset of the data, i.e the mean.

>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

- For the 1st argument we specified that price should be aggregated by cut. Notice that we only specified the column name and did not have to identify the data because that is given in the second argument. After the third argument specifying the function, additional named arguments to that function can be passed, such as aggregate(price~cut+color,diamonds,mean,na.rm=TRUE).
- To group the data by more than one variable, add the additional variable to the right side of the formula seperating it with a plus sign (+).

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

- 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)
```

- To apply more than one function it is easier to use the plyr package.

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

plyr

- One of the best things to ever happen to R was the development of the plyr package by **Hadley Wickham.**
- It epitomizes the "split-apply-combine" method of data manipulation.
- The core of plyr consists of functions such as ddply, llply and ldply.
- All the manipulation functions consists of 5 letters, where the last three letters were in common and the first letter for each manipulation function describes the type of input and the 2nd letter for each manipulation function describes the type of output.
- For instance, ddply takes in a data.frame and outputs a data.frame.

plyr functions and their corresponding Inputs & Outputs

Function	Input Type	Output Type
ddply	data.frame	data.frame
llply	list	list
aaply	array/vector/matrix	array/vector/matrix
dlply	data.frame	list
daply	data.frame	array/vector/matrix
d_ply	data.frame	None(used for sideeffects)
ldply	list	data.frame
laply	list	array/vector/matrix
l_ply	list	None(used for sideeffects)
adply	array/vector/matrix	data.frame
alply	array/vector/matrix	list
a_ply	array/vector/matrix	None(used for sideeffects)

ddply

- Takes a data.frame, splits it according to some variables, perform a desired action on it and returns a data.frame.

>require(plyr)

>head(baseball)

- A common statistic in baseball is On Base Percentage(OBP), which is calculated as

  $$OBP=(H+BB+HBP)/(AB+BB+HBP+SF)$$

  where H=Hits, BB=Base on Balls (walks), HBP=Times Hit by Pitch, AB=At Bats, SF=Sacrifice Flies.
- Before 1954 sacrifice flies are counted as part of sacrifice hits, which includes bunts, so for players before 1954 sacrifice flies should be assumed to be 0. That will be the first change we make to the data. There are many instances of HBP that are NA, so we set those to 0 as well. We also exclude players with less than 50 at bats in a season.

```
>#subsetting with [is faster than using ifelse]
>baseball$sf[baseball$year<1954] <- 0
>#check that it worked
>any(is.na(baseball$sf))
[1]  FALSE
>#set NA hbp's to 0
>baseball$hbp[is.na(baseball$hbp)]<- 0
>#check that it worked
>any(is.na(baseball$hbp))
[1]  FALSE
>#only keep players with at least 50 at bats in a season
>baseball<- baseball[baseball$ab >=50,  ]
```

Calculating the OBP for a given player in a given year is easy enough with just vector operations.

```
>#calculate OBP
>baseball$OBP<- with(baseball, (h + bb + hbp)/(ab + bb + hbp + sf))
>tail(baseball)
```

- Here we used a new function **with.** This allows us to specify the columns of a data.frame without having to specify the data.frame name each time.
- To calculate the OBP for a player's entire career we cannot just average his individual season OBP's, we need to calculate and sum the numerator, and then divide by the sum of the denominator. This requires the use of ddply.
- First we make a function to do the calculation, then we use ddply to run that calculation for each player.

```
>#this function assumes that the column names for the data are as
>#below
>obp<-function(data)
>{
>c(OBP=with(data, sum(h+bb+hbp)/sum(ab+bb+hbp+sf)))
>}
>#use ddply to calculate career OBP for each player
>careerOBP<- ddply(baseball, .variable= "id", .fun= obp)
>#sort the results
>careerOBP<- careerOBP[order(careerOBP$OBP, decreasing = TRUE), ]
>#see the results
>head(careerOBP, 10)
```

This returns the top ten players by career on base percentage.

llply

- We use lapply to sum each element of a list.

```
>theList<- list(A= matrix(1:9, 3), B= 1:5, C= matrix(1:4, 2), D=2)
>lapply(theList, sum)
$A
[1]  45
$B
[1]  15
$C
[1]  10
$D
[1]  2
```

This can be done with llply, yielding identical results.

```
>llply(theList, sum)
$A
[1]  45
$B
[1]  15
$C
[1]  10
$D
[1]  2
```

>identical(lapply(theList, sum), llply(theList, sum))

[1] TRUE

To get the result as a vector, lapply can be used similarly to sapply.

>sapply(theList, sum)

A B C D

45 15 10 2

>laply(theList, sum)

[1] 45 15 10 2

- However, that while the results are same, laply did not include names for the vector.

plyr Helper Functions

- plyr has a great deal of useful helper functions such as each, which lets us supply multiple functions to a function like aggregate.

>aggregate(price ~ cut, diamonds, each(mean, median))

- Another great function is idata.frame, which creates a reference to a data.frame so that subsetting is much faster and more memory efficient.

>system.time(dlply(baseball, “id”, nrow))

user system elapsed

0.29 0.00 0.33

>iBaseball<- idata.frame(baseball)

>system.time(dlply(iBaseball, “id”, nrow))

user system elapsed

0.42 0.00 0.47

- While saving less than a second in run time might seen trivial the savings can really add up with more complex operations, bigger data, more groups to split by and repeated operation.

Speed vs Convenience

- A criticism often leveled at plyr is that it can run slowly.
- Typical response to this is that using plyr is a question of speed vs convenience.
- Most of the functionality in plyr can be accomplished using base functions or other packages, but few of those offer the ease of use of plyr.
- In recent years Hadley Wickham has taken great steps to speed up plyr, including optimized R code, C++ code and parallelization.

data.table

- For speed junkies there is a package called data.table that extends and enhances the functionality of data.frames.
- The syntax is little different from regular data.frames, so it will take getting used to, which probably the primary reason it has not seen near-universal adoption.
- The secret to the speed is that data.tables have an **index** like databases. This allows faster value accessing, group by operations and joins.

- Creating data.tables is just like creating data.frames, and the two are very similar.

```
>require(data.table)
>#create a regular data.frame
>theDF<-data.frame(A=1:10,
    B=letters[1:10],C=LETTERS[11:20],D=rep(c("One","Two","Three"),
    length.out=10))
>#create a data.table
>theDT<data.table(A=1:10,
    B=letters[1:10],C=LETTERS[11:20],D=rep(c("One","Two","Three"),
    length.out=10))
>#print them and compare
>theDF
```

	A	B	C	D
1	1	a	K	One
2	2	b	L	Two
3	3	c	M	Three
4	4	d	N	One

5	5	e	O	Two
6	6	f	P	Three
7	7	g	Q	One
8	8	h	R	Two
9	9	i	S	Three
10	10	j	T	One

>theDT

	A	B	C	D
1:	1	a	K	One
2:	2	b	L	Two
3:	3	c	M	Three
4:	4	d	N	One
5:	5	e	O	Two
6:	6	f	P	Three
7:	7	g	Q	One
8:	8	h	R	Two
9:	9	i	S	Three
10:	10	j	T	One

>#Notice by default data.frame turns character data into factors.

>#while data.table does not

>class(theDF$B)

[1] “factor”

>class(theDT$B)

[1] “character”

- The data are identical-except that data.frame turned B into a factor while data.table did not-and only the way it was printed looks different.
- It is also possible to create a data.table out of an existing data.frame.

>diamondsDT<-data.table(diamonds)

>diamondsDT

Note:

- Notice that printing the diamonds data would try to print out all the data but data.table intelligently just prints the first five and last five rows.
- Accessing rows can be done similarly to accessing rows in a data.frame.

>theDT[1:2,]

	A	B	C	D
1:	1	a	K	One
2:	2	b	L	Two

```
>theDT[theDT$A>=7,  ]
      A  B      C      D
1:    7  g      Q      One
2:    8  h      R      Two
3:    9  i      S      Three
4:    10 j      T      One
```

- Accessing individual columns must be done a little differently than accessing columns in data.frames.

```
>theDT[  , list(A,C)]
      A  C
1:    1  K
2:    2  L
3:    3  M
4:    4  N
5:    5  O
6:    6  P
7:    7  Q
8:    8  R
9:    9  S
10:   10 T
```

```
>#just one column
>theDT[  , B]
[1]    “a”  “b”  “c”  “d”  “e”  “f”  “g”  “h”  “i”  “j”
>#  one column while maintaining data.table structure
>theDT[  , list(B)]
        B
1:      a
2:      b
3:      c
4:      d
5:      e
6:      f
7:      g
8:      h
9:      i
10:     j
```

- If we must specify the column names as characters, the with argument should be set to FALSE.

```
>theDT[  , “B”, with= FALSE]
```

```
        B
1:      a
2:      b
3:      c
4:      d
5:      e
6:      f
7:      g
8:      h
9:      i
10:     j
>theDT[  , c(“A”, “C”), with=  FALSE]
        A   C
1:      1   K
2:      2   L
3:      3   M
4:      4   N
5:      5   O
6:      6   P
7:      7   Q
8:      8   R
9:      9   S
10:     10  T
```

Keys

- Now that we have a few data.tables in memory, we might be interested in seeing some information about them.

```
>#show tables
>tables()
      NAME        NROW    MB
[1,]  diamondDT   53,940  4
[2,]  theDT       10      1
      COLS                                                    KEY
[1,]  carat,cut,color,clarity,depth,table,price,x,y,z
[2,]  A,B,C,D
Total: 5MB
```

- This shows, for each data.table in memory, the name, the number of rows, the size in megabytes, the column names and the key. We have not assigned keys for any of the tables so that column is blank. The key is used to index the data.table and will provide the extra speed.
- We start by adding a key to theDT. We will use the D column to index the data.table. This is done using setkey, which takes the name of the data.table as its first argument and the name of the desired column as the second argument.

```
>#set the key
>setkey(theDT, D)
>#show the data.table again
>theDT
```

	A	B	C	D
1:	1	a	K	One
2:	4	d	N	One
3:	7	g	Q	One
4:	10	j	T	One
5:	3	c	M	Three
6:	6	f	P	Three
7:	9	i	S	Three
8:	2	b	L	Two
9:	5	e	O	Two
10:	8	h	R	Two

- The data have been reordered according to column D, which is sorted alphabetically, We can confirm the key was set with key.

```
>key(theDT)
[1]  “D”
```

Or tables.

```
>tables()
      NAME                NROW    MB
[1,]  diamondsDT          53,940  4
[2,]  theDT               10      1
      COLS                                                KEY
[1,]  carat,cut,color,clarity,depth,table,price,x,y,z
[2,]  A,B,C,D                                             D
Total: 5MB
```

- This adds some new functionality to selecting rows from data.tables. In addition to selecting rows by the row number or by some expression that evaluates to TRUE or FALSE, a value of the key column can be specified.

```
>theDT[“One”,  ]
     D      A     B     C
1:   One    1     a     K
2:   One    4     d     N
3:   One    7     g     Q
4:   One    10    j     T
```

>theDT[c(“One”, “Two”),]

	D	A	B	C
1:	One	1	a	K
2:	One	4	d	N
3:	One	7	g	Q
4:	One	10	j	T
5:	Two	2	b	L
6:	Two	5	e	O
7:	Two	8	h	R

More than one column can be set as the key.

>#set the key

>setkey(diamondsDT, cut, color)

- To access the rows according to both keys, there is a special function named J. It takes multiple arguments, each of which is a vector of values to select.

>#access some rows

>diamondsDT(J(“Ideal”, “E”),]

>diamondsDT[J(“Ideal”,c(“E”, “D”)),]

data.table Aggregation

- The primary benefit of indexing is faster aggregation. While aggregate and the various d*ply functions will work because data.tables are just enhances data.frames, they will be slower than using the built-in aggregation functionality of data.table.

Calculating the mean price of diamonds for each type of cut:

```
>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 get the same result sing data.table, we do this:

```
>diamondsDT[  , mean(price), by=cut]
        cut         V1
1:      Fair        4358.758
2:      Good        3928.864
3:      Very Good   3981.760
4:      Premium     4584.258
5.      Ideal       3457.542
```

- The only difference between this and the previous result is that the columns have different names. To specify the name of the resulting column, pass the aggregation function as a named list.

```
>diamondsDT[  , list(price=mean(price)), by=cut]
        cut        price
1:      Fair       4358.758
2:      Good       3928.864
3:      Very Good  3981.760
4:      Premium    4584.258
5.      Ideal      3457.542
```

To aggregate multiple columns, specify them as a list().

```
>diamondsDT[  ,  mean(price), by=list(cut,color)]
        cut        color      V1
1:      Fair       D          4291.061
2:      Fair       E          3682.312
3:      Fair       F          3827.003
4:      Fair       G          4239.255
5:      Fair       H          5135.683
6:      Fair       I          4685.446
7:      Fair       J          4975.655
8:      Good       D          3405.382
‘
‘
```

- To aggregate multiple arguments, pass them as a list. Unlike with aggregate, a different metric can be measured for each column.

```
>diamondDT[  , list(price=mean(price), carat= mean(carat)), by = cut]
```

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

```
>diamondsDT[  , list(price=mean(price), carat=mean(carat), caratSum=sum(carat)),
  by=cut]
```

	cut	price	carat	caratSum
1:	Ideal	3457.542	0.7028370	15146.84
2:	Premium	4584.258	0.8919549	12300.95
3:	Good	3928.864	0.8491847	4166.10
4:	Very Good	3981.760	0.8063814	9742.70
5:	Fair	4358.758	1.0461366	1684.28

- Finally, both multiple metrices can be calculated and multiple grouping variables can be specified at the same time.

```
>diamondsDT[  ,  list(price=mean(price), carat=mean(carat)), by=list(cut,color)]
      cut       color    price          carat
1:    Ideal     E        2597.550       0.5784012
2:    Premium   E        3538.914       0.7177450
3:    Good      E        3423.644       0.7451340
4:
‘
‘
‘
‘
‘
‘
‘
‘
‘
‘
‘
```

Data Reshaping

- Focus is on when the data needs to be rearranged from column oriented to row oriented and when the data are in multiple, separate sets and need to be combined into one.
- There are base functions to accomplish these tasks but will focus on those in plyr, reshape2 and data.table.

cbind and rbind:

- The simplest case is when we have two datasets with either identical columns(both the number of and names) or the same number of rows. In this case, either rbind or cbind work great.

Eg: We create two simple data.frames by combining a few vectors with cbind, and then stack them using rbind.

```
>#make two vectors and combine them as a columns in a data.frame
>sport<- c("Hockey", "Baseball", "Football")
>league<- c("NHL", "MLB", "NFL")
>trophy<- c("Stanley Cup", "Commissioner's Trophy", "Vince Lombardi Trophy")
>trophies1<- cbind(sport, league, trophy)
>#make another data.frame using data.frame()
>trophies2<- data.frame(sport=c("Basketball", "Golf"), league=c("NBA", "PGA"), trophy=c("Larry O'Brien Championship Trophy", "Wanamaker Trophy"), stringsAsFactors=FALSE)
```

>#combine them into one data.frame with rbind

>trophies<-rbind(trophies1,trophies2)

- Both cbind and rbind can take multiple arguments to combine an arbitary number of objects. Note that it is possible to assign new column names to vectors in cbind.

>cbind(Sport=sport, Association=league, Prize= trophy)

	Sport	Association	Prize
[1,]	“Hockey”	“NHL”	“Stanley Cup”
[2,]	“Baseball”	“MLB”	“Commissioner’s Trophy”
[3,]	“Football”	“NFL”	“Vince Lombardi Trophy”

Joins

- Data do not always come so nicely aligned for combining using cbind, so they need to be joined together using a common key. This concept should be familiar to SQL users.
- Joins in R are not flexible as SQL Joins, but are still an essential operation in the data analysis process.
- The three most commonly used functions for joins are merge in base R, join in plyr and the merging functionality in data.table. Each has pros and cons with some pros outweighing their respective cons.

- To illustrate these functions I have prepared data originally made available as part of USAID Open Govt initiative. The data have been chopped into eight separate files so that they can be joined together. They are all available in a zip file at http://jaredlander.com/data/US_Foreign_Aid.zip.
- These should be downloaded and unzipped to a folder on our computer. This can be done a number of ways but we show how to download and unzip using R.

```
>download.file(url=http://jaredlander.com/data/US_Foreign_Aid.zip,destfile="data/ForeignAid.zip")
>unzip("data/ForeignAid.zip",exdir="data")
```

- To load all of these files programmatically, we use a for loop. We get a list of files using dir, and then loop through that list assigning each dataset to a name specified using assign.

```
>require(stringr)
>#first get a list of the files
>theFiles<-dir("data/",pattern="\\.csv")
>##loop through those files
>for(a in theFiles)
{
    # build a good name to assign to the data
    nameToUse<- str_sub(string=a, start=12, end=18)
```

```
    #read in the csv using read.table
    #file.path is a convenient way to specify a folder and file name
    temp<-read.table(file=file.path("data", a), header=TRUE, sep=",",
    stringsAsFactors=FALSE)
    #assign them into the workspace
    assign(x=nameToUse, value=temp)
}
```

1. merge
2. plyr join
3. data.table merge

merge:

- R comes with a built-in function, called merge, to merge two data.frames.

```
>Aid90s00s<-merge(x=Aid_90s, y=Aid_00s, by.x=c("Country.Name",
    "Program.Name"), by.y=c("Country.Name", "Program.Name"))
>head(Aid90s00s)
```

- The by.x specifies the key column(s) in the left data.frame and the by.y does the same for the right data.frame. The ability to specify different column names for each data.frame is the most useful feature of merge. The biggest drawback, is that **merge can be much slower than the alternatives**.

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, through, is that the key column(s) in each table must have the same name.

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

- Join has an argument for specifying a left, right, inner or full (outer) join.
- We have eight data.frames containing foreign assistance data that we would like to combine into one data.frames without hand coding each join.
- The best way to do this is to put all the data.frames into a list, and then successively join them together using Reduce.

```
>#first figure out the names of the data.frames
>frameNames<-str_sub(string=theFiles, start=12, end=18)
#build an empty list
>frameList<- vector("list", length(frameNames))
>names(frameList)<-frameNames
># add each data.frame into the List
```

```
>for(a in fameNames)
{
    frameList[[a]] <- eval(parse(text=a))
}
```

- Fist we reconstructed the names of the data.frames using str_sub from Hadley Wickham's stringr package. Then we built an empty list with as many elements as there are data.frames, in this case eight, using vector and assigning its mode to "list". We than set appropriate names to the List.
- Now that the list is built and named, we loop through it, assigning to each element the appropriate data.frame. The problem is that we have the names of the data.frames as characters but the <- operator requires a variable, not a character. So we parse and evaluate the character, which realizes the actual variable. Inspecting, we see that the list does indeed contain the appropriate data.frames.

```
>head(frameList[[1]])
>head(frameList[["Aid_00s"]])
>head(frameList[[5]])
>head(frameList[["Aid_60s"]])
```

- Having all the data.frames in a list allows us to iterate through the list, joining all the elements together. Rather than using a loop, we use the Reduce function to speed up the operation.

```
>allAid<-Reduce(function(. . . )
{
        join(. . . , by=c(“Country.Name”, “Program.Name”))
}, frameList)
>dim(allAid)
[1]   2453  67
>require(useful)
>corner(allAid, c=15)
>bottomleft(allAid, c=15)
```

data.table merge

- Like many other operations in data.table, joining data requires a different syntax, and possibly a different way of thinking. To start, we convert two of our foreign aid datasets’ data.frames into data.tables.

```
>require(data.table)
>dt90<-data.table(Aids_90s, key=c(“Country.Name”, “Program.Name”))
>dt00<-data.table(Aid_00s, key=c(“Country.Name”, “Program.Name”))
```

- Then, doing the join is a simple operation. Note that join requires specifying the keys for the data.tables, which we did during their creation.

```
>dt0090<-dt90[dt00]
```

- In this case dt90 is the left side, dt00 is the right side and a left join was performed.

reshape2

- The next most common munging need is either melting data(going from column orientation to row orientation) or casting data (going from row orientation to column orientation).
- As with most other procedures in R, there are multiple functions available to accomplish these tasks but we will focus on Hadley Wickham's reshape2 package.

1.melt

2.dcast

melt

- Looking at the Aid_00s data.frame, we see that each year is stored in its own column. That is, the dollar amount for a given country and program is found in a different column for each year. This is called a cross table, which , while nice for human consumption, is not ideal for graphing with ggplot2 or for some analysis algorithms.

```
>head(Aid_00s)


>require(reshape2)
>melt00<-melt(Aid_00s, id.vars=c("Country.Name", "Program.Name"),
    variable.name="Year", value.name="Dollars")
```

>tail(melt00,10)

- The id.vars argument specifies which columns uniquely identify a row.

>require(scales)

>#strip the “FY” out of the year column and convert it to numeric

>melt00$Year<-as.numeric(str_sub(melt00$Year, start=3, 6)

>#aggregate the data so we have yearly numbers by program

>meltAgg<-aggregate(Dollars~Program.Name+Year, data=melt00, sum, na.rm=TRUE)

>#just keep the first 10 characters of program name

>#then it will fit in the plot

>meltAgg$Program.Name<-str_sub(meltAgg$Program.Name, start=1, end=10)

>ggplot(meltAgg, aes(x=Year, y=Dollars)) + geom_line(aes(group=Program.Name)) + facet_wrap(~Program.Name) + scale_x_continuous(breaks=seq(from=2000, to=2009, by=2)) + theme(axis.text.x=element_text(angle=90, vjust=1, hjust=0)) + scale_y_continuous(labels=multiple_format(extra=dollar, multiple=“B”))

dcast

- Now that we have the foreign aid data melted, we cast it back into the wide format for illustration purposes. The function for this is dcast, and it has trickier arguments than melt. The first is the data to be used, in our case melt00. The second argument is a formula where the left side specifies the columns that should remain columns and the right side specifies the columns that should become row names.

- The third argument is the column (as a character) that holds the values to be populated into the new columns representing the unique values of the right side of the formula argument.

>cast00<- dcast(melt00, Country.Name + Program.Name ~ Year, value.var="Dollars")

>head(cast00)

Manipulating Strings:

- Strings (character data) often need to be constructed or deconstructed to identify observations, preprocess text, combine information or satisfy any number of other needs.
- R offers functions for building strings, like paste and sprintf. It also provides a number of functions for using regular expressions and examining text data, although for those purposes it is better to use Hadley Wickham's stringr package.

1.paste

2.sprintf

3.Extracting Text

4.Regular expressions

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. We start by putting together three simple strings.

>paste(“Hello”, “Jared”, “and others”)

[1] “Hello Jared and others”

Note:

- Spaces were put between the strings. This is because paste has a third argument, sep, that determines what to put in between entries. This can be any valid text, including empty text(“ “).

>paste(“Hello”, “Jared”, “and others”, sep= “/”)

[1] “Hello/Jared/and others”

- Like many functions in R, paste is vectorized. This means each element can be a vector of data to be put together.

>paste(c(“Hello”, “Hey”, “Howdy”), c(“Jared”, “Bob”, “David”))

[1] “Hello Jared” “Hey Bob” “Howdy David”

- In this case each vector had the same number of entries so they paired one-to-one. When the vectors do not have the same length they are recycled.

>paste(“Hello”, c(“Jared”, “Bob”, “David”))

[1] “Hello Jared” “Hello Bob” “Hello David”

>paste(“Hello”, c(“Jared”, “Bob”, “David”), c(“Goodbye”, “Seeya”))

[1] “Hello Jared Goodbye” “Hello Bob Seeya” “Hello David Goodbye”

- paste also has the ability to collapse a vector of text into one vector containing all the elements with any arbitrary separator, using the collapse argument.

>vrctorOfText<- c(“Hello”, “Everyone”, “out there”, “.”)

>paste(vectorOfText, collapse=“ “)

[1] “Hello Everyone out there .”

>paste(vectorOfText, collapse=“*”)

[1] “Hello*Everyone*out there*.”

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 25minutes” 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.

>person<- “Jared”

>partySize<- “eight”

>WaitTime<- 25

Now we build the paste expression.

>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.”

- Making even a small change to this sentence would require putting the commas in just the right places.
- A good alternative is the sprintf function. With this function we build one long string with special markers indicating where to insert values.

>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”

- Here, each %s was replaced with its corresponding variable. While the long sentence is easier to read in code, we must maintain the order of %s’s and variables.
- sprintf is also vectorized.
- Note: The vector lengths must be multiples of each other.

>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”

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.
- First we need some data, so we use the XML package to download a table of United States presidents from Wikipedia.

>require(XML)

Then we use readHTMLTable to parse the table.

>load(“data/presidents.rdata”)

>theURL<- “http://www.loc.gov/rr/print/list/057_chron.html” skip.rows=1, header=TRUE, stringsAsFactor=FALSE)

Now we take a look at the data

>head(presidents)

>tail(presidents)

```
>presidents<-presidents[1:64,  ]
>require(stringr)
>#split the string
>yearList<- str_split(string=presidents$YEAR, pattern= “-”)
>head(yearList)
[[1]]
[1]  “1789”  “1797”
[[2]]
[1]  “1797”  “1801”
[[3]]
[1]  “1801”  “1805”
[[4]]
[1]  “1805”  “1809”
[[5]]
[1]  “1809”  “1812”
[[6]]
[1]  “1812”  “1813”
```

```
>#combine them into one matrix
>yearMatrix<-data.frame(Reduce(rbind,yearList))
>head(yearMatrix)
      X1        X2
1   1789      1797
2   1797      1801
3   1801      1805
4   1805      1809
5   1809      1812
6   1812      1813
>#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$Start<- as.numeric(as.charatcter(presidents$Start))
>president$Stop<- as.numeric(as.character(presidents$Stop))
>#view the changes
>head(presidents)
```

>tail(presidents)

>#get the first 3 characters

>str_sub(string= presidents$PRESIDENT, start=1, end=3)

># get the 4th through 8th characters

>str_sub(string=presidents$PRESIDENT, start=4, end=8)

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

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.

>#returns TRUE/FALSE if John was found in the name

>johnPos<- str_detect(string= presidents$PRESIDENT, pattern= "John")

>presidents[johnPos, c("YEAR", "PRESIDENT", "Start", "Stop")]

Note:

- The regular expressions are case sensitive, so to ignore case we have to put the pattern in ignore.case.

```
>badSearch<- str_detect(presidents$PRESIDENT, “john”)
>goodSearch<- str_detect(presidents$PRSEIDENT, ignore.case(“John”))
>sum(badSearch)
[1]  0
>sum(goodSearch)
[1]  7

>con<- url(“http://www.jaredlander.com/data/warTimes.rdata”)
>load(con)
>close(con)

>head(warTimes, 10)
[1]  “September 1, 1774 ACAEA September 3, 1783”
[2]
```

>warTimes[str_detect(string=warTimes, pattern= “-”)]

>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.

>which(str_detect(string=warTimes, pattern= “-”))

>theStart<- sapply(theTimes, FUN= function(x) x[1])

>head(theStart)

>theStart<-str_trim(theStart)

>head(theStart)

>#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”)]

>#get incidents of 4 numeric digits in a row

>head(str_extract(string= theStart, “[0-9] [0-9] [0-9]”), 20)

># a smarter way to search for four numbers

>head(str_extract(string= theStart, “[0-9] {4}”), 20)

```
># “\\d”   is a shortcut for “[0-9]”
>head(str_extract(string= theStart, “\\d{4}”), 20)
>#this looks for any digit that occurs either once, twice or thrice
>str_extract(string= theStart, “\\d{1,3}”)
># 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)
># replace the first digit seen with “x”
>head(str_replace(string=theStart, pattern=“\\d” , replacement=“x”), 30)
>#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)
```

```
>#create a vector of HTML commands
>commands<-c("<a href=index.html>The Link is here</a>", "<b>This is bold text</b>")
>#get the text between the HTML tags
>#the content in (.+?) is substituted using 1
>str_replace(string=commands, pattern="<.+?>(.+>", replacement="\\1")
[1]  "The Link is here" "This is bold text"
```

- Since R has its own regular expression peculiarities, there is a handy help file that can be accessed with ?regex.

Math Functions

R includes an extensive set of built-in math functions. Hare is a partial list:

- exp(): Exponential function, base e
- log(): Natural logarithm
- log10(): Logarithm base 10
- sqrt(): Square root
- abs(): Absolute value
- sin(), cos(), and so on: Trig functions
- min() and max(): minimum value and maximum value within a vector.

- which.min() and which.max(): Index of the minimal element and maximal element of a vector
- pmin() and pmax(): Element-wise minima and maxima of several vectors.
- sum() and prod(): Sum and product of the elements of a vector
- cumsum() and cumprod(): Cumulative sum and product of the elements of a vector.
- round(), floor(), and ceiling(): Round to the closet integer, to the closet integer below, and to the closet integer above.
- factorial():Factorial function.

Calculating Probability

Cumulative sums and products

Minima and maxima

Calculus

Sorting

Set operations

Simulation Programming in R:

Built-in-Random variable generators

Obtaining the same random stream in repeated runs

An example to a combinatorial simulation