Module 7: Data reshaping

BIOSTAT 213: Introduction to Computing in the R Software Environment

Monday, November 4, 2019

Date objects

Date objects

```
> pp<-read.csv('presidencies.csv')
> head(pp$start)
[1] 30/04/1789 4/03/1797  4/03/1801  4/03/1809  4/03/1817
[6] 4/03/1825
44 Levels: 12/4/1945 14/9/1901 15/04/1865 ... 9/8/1974
> class(pp$start)
[1] "factor"
```

Date objects

```
> pp$end-pp$start
Warning in Ops.factor(pp$end, pp$start): '-' not meaningful for
factors
 [1] NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA
[21] NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA
[41] NA NA NA NA
```

Date objects

- one method for converting strings to dates is `as.Date()`

Date objects

```
> head(pp$start)
[1] 30/04/1789 4/03/1797  4/03/1801  4/03/1809  4/03/1817
[6] 4/03/1825
44 Levels: 12/4/1945 14/9/1901 15/04/1865 ... 9/8/1974
> pp$start<-as.Date(pp$start,format='%d/%m/%Y')
```

- the `format` should replicate the string strings
- `%d`: day of the month
- the `/`: the / see in `head()`
- `%m`: month, as a number
- `%Y`: year, stored with 4 digits (if the year is stored with 2 digits, use %y)
- a table of the % codes is available here

Date objects

- checking work:

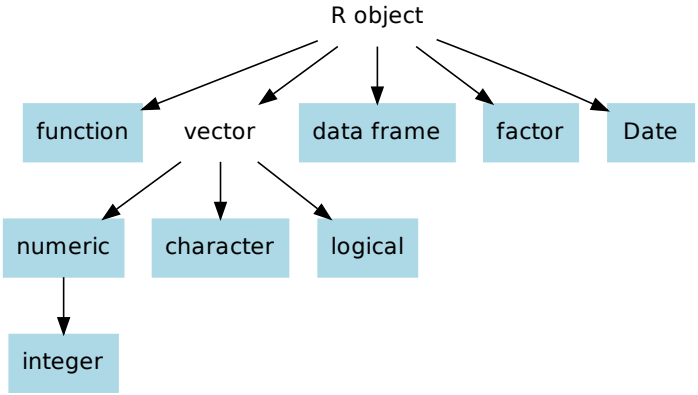
```
> head(pp$start)
[1] "1789-04-30" "1797-03-04" "1801-03-04" "1809-03-04"
[5] "1817-03-04" "1825-03-04"
```

Date objects

```
> class(pp$start)
[1] "Date"
> is.vector(pp$start)
[1] FALSE
```

- `start` is now a Date object
- Date objects look like vectors but are not vectors

Date objects



Date objects

```
> head(pp$end)
[1] 4/03/1797 4/03/1801 4/03/1809 4/03/1817 4/03/1825 4/03/1829
44 Levels: 12/4/1945 14/9/1901 15/04/1865 ... 9/8/1974
> pp$end<-as.Date(pp$end,format='%d/%m/%Y')
> head(pp$end)
[1] "1797-03-04" "1801-03-04" "1809-03-04" "1817-03-04"
[5] "1825-03-04" "1829-03-04"
> class(pp$end)
[1] "Date"
```

Date objects

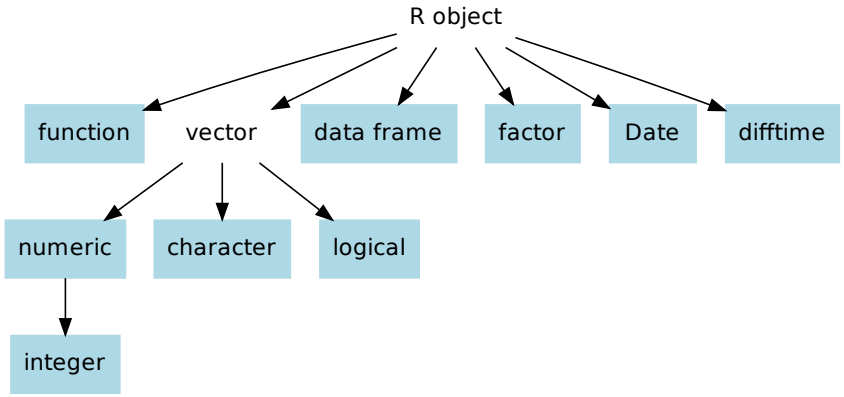
```
> pp$duration<-pp$end-pp$start
> min(pp$duration)
Time difference of 31 days
> max(pp$duration)
Time difference of 4422 days
```

difftime objects

```
> head(pp$duration)
Time differences in days
[1] 2865 1460 2922 2922 2922 1461
> class(pp$duration)
[1] "difftime"
> is.vector(pp$duration)
[1] FALSE
```

- `duration` is a difftime object
- difftime objects look like vectors but are not vectors

difftime objects



difftime objects

- to convert a difftime object to a numeric object, use `as.numeric()`:

```
> head(pp$duration)
Time differences in days
[1] 2865 1460 2922 2922 2922 1461
> pp$duration<-as.numeric(pp$duration)
> head(pp$duration)
[1] 2865 1460 2922 2922 2922 1461
```

Date objects

```
> head(pp$dob)
[1] February 22, 1732 October 30, 1735  April 13, 1743
[4] March 16, 1751    April 28, 1758    July 11, 1767
43 Levels: April 13, 1743 April 23, 1791 ... September 15, 1857
> pp$dob<-as.Date(pp$dob,format='%B %d, %Y')
```

- %B for month, as non-abbreviated names
- the spaces and comma replicate the format seen in head()

Date objects

- checking work:

```
> head(pp$dob)
[1] "1732-02-22" "1735-10-30" "1743-04-13" "1751-03-16"
[5] "1758-04-28" "1767-07-11"
> class(pp$dob)
[1] "Date"
```

idu5.csv

```
> idu5<-read.csv('idu5.csv')
```

- a stripped-down data set of individuals from San Francisco's 5th implementation of CDC's National HIV Behavioral Surveillance for injection drug users (IDU)

Date objects

```
> head(idu5$idate)
[1] 21394 21394 21394 21394 21394 21394
> class(idu5$idate)
[1] "integer"
> idu5$idate<-as.Date(idu5$idate,origin='1960-01-01')
```

- to convert SAS-like dates, use the `origin` argument
- specify the `origin` using a `year-month-day` format

Date objects

- checking work:

```
> head(idu5$idate)
[1] "2018-07-29" "2018-07-29" "2018-07-29" "2018-07-29"
[5] "2018-07-29" "2018-07-29"
> class(idu5$idate)
[1] "Date"
> summary(idu5$idate)
        Min.      1st Qu.       Median         Mean      3rd Qu.
"2018-07-29" "2018-09-02" "2018-10-07" "2018-10-07" "2018-11-08"
        Max.
"2018-12-20"
```

Date objects

- Date objects (capital D) do not store time
- there are other types of date objects in R however, that do
- to convert to date objects that store time as well, try `strptime()`

rbind() and cbind()

rbind()

```
> ii<-read.csv('nhanes.csv')
> ii.123<-ii[1:3,c('bmxht','bmxwt','bmxbmi')]
> ii.45<-ii[4:5,c('bmxht','bmxwt','bmxbmi')]
> rbind(ii.123,ii.45)
  bmxht bmxwt bmxbmi
1 184.5  94.8   27.8
2 171.4  90.4   30.8
3 160.9 109.8   42.4
4 164.9  55.2   20.3
5 165.4  76.6   28.0
```

- `rbind()` stands for row bind and, as the name implies, combines (binds) data vertically

rbind()

```
> class.i<-subset(ii,bmicat=='Class I obesity',select=1:3)
> class.ii<-subset(ii,bmicat=='Class II obesity',select=1:3)
> dim(class.i)
[1] 876   3
> dim(class.ii)
[1] 469   3
> class.i.or.ii<-rbind(class.i,class.ii)
> dim(class.i.or.ii)
[1] 1345    3
```

rbind()

```
> ii.1<-ii[1,]
> ii.2<-ii[2,]
> ii.3<-ii[3,]
> ii.123<-rbind(ii.1,ii.2,ii.3)
> nrow(ii.123)
[1] 3
```

rbind()

- the number of columns must match
- the column names must also match, if column names exist
- interestingly, the columns do not need to be in the same order:

```
> ii.1<-ii[1,c('bmxwt','bmxht')]
> ii.2<-ii[2,c('bmxht','bmxwt')]
> rbind(ii.1,ii.2)
  bmxwt bmxht
1  94.8 184.5
2  90.4 171.4
```

cbind()

```
> ii.bmx<-ii[1:3,c('bmxht','bmxwt')]
> ii.dmd<-ii[1:3,c('dmdborn4','dmdcitzn')]
> cbind(ii.bmx,ii.dmd)
  bmxht bmxwt dmdborn4 dmdcitzn
1 184.5  94.8        1        1
2 171.4  90.4        2        2
3 160.9 109.8        1        1
```

- `cbind()` stands for column bind and, as the name implies, combines (binds) data horizontally

cbind()

- the number of rows must match
- the row names do not need to match, if row names exist
- in other words, it is possible to misalign rows:

```
> ii.bmx<-ii[1:2,c('bmxht','bmxwt')]
> ii.dmd<-ii[101:102,c('dmdborn4','dmdcitzn')]
> row.names(ii.bmx)
[1] "1" "2"
> row.names(ii.dmd)
[1] "101" "102"
> cbind(ii.bmx,ii.dmd)
  bmxht bmxwt dmdborn4 dmdcitzn
1 184.5  94.8        1        1
2 171.4  90.4        1        1
```

Importing other data files

kevin_love.txt

- `kevin_love.txt` contains data on Kevin Love's NBA career
- it is a tab-delimited text file

read.delim()

- to import tab-delimited text files, use `read.delim()`:

```
> ll<-read.delim('kevin_love.txt')
> ll[1:3,1:10]
   season age  tm  lg pos  g gs   mp  fg  fga
1 2008-09  20 MIN NBA   C 81 37 25.3 3.9  8.5
2 2009-10  21 MIN NBA  PF 60 22 28.6 4.9 10.8
3 2010-11  22 MIN NBA  PF 73 73 35.8 6.6 14.1
```

read.table()

- `read.delim()` and `read.csv()` are special cases of `read.table()`
- some useful arguments to `read.table()`:
  - `header`: `TRUE` if the first row of the data contains the variable names
  - `sep`: the value used to separate cells (for example, `'\t'` for tab-delimited data)
  - `na.strings`: the character used to indicate missing values (for example, `''` for empty cells)

Importing other data files

- useful packages for reading other data files:
  - foreign for reading Stata, SPSS, or SAS XPORT files
  - haven for reading Stata, SPSS, or SAS XPORT files
  - readstata13 for reading Stata 13+ files
  - sas7bdat for reading SAS sas7bdat files
  - readxl for reading Excel files

DIQ_I.XPT

- DIQ_I.XPT contains data from the diabetes component of NHANES

Importing SAS XPORT files

- just once ever:

```
> install.packages('foreign')
```

- just once per session:

```
> library(foreign)
```

Importing SAS XPORT files

```
> diabetes<-read.xport('DIQ_I.XPT')
```

Importing SAS XPORT files

- checking work:

```
> dim(diabetes)
[1] 9575   54
> diabetes[1:3,1:8]
   SEQN DIQ010 DID040 DIQ160 DIQ170 DIQ172 DIQ175A DIQ175B
1 83732      1     46     NA     NA     NA      NA      NA
2 83733      2     NA      2      2      2      NA      NA
3 83734      1     52     NA     NA     NA      NA      NA
```

Importing SAS XPORT files

- to optionally convert variable names to lower case, use `tolower()`:

```
> names(diabetes)<-tolower(names(diabetes))
```

Merging data

Merging data

- prior to merging, it's useful to check for duplicated IDs:

```
> sum(duplicated(ii$seqn))
[1] 0
> sum(duplicated(diabetes$seqn))
[1] 0
```

Merging data

- to merge data frames, use `merge()`:

```
> ii.new<-merge(ii,diabetes,by='seqn',all.x=TRUE,all.y=FALSE)
```

- `by` specifies the identifying variable(s) (usually some ID number)
- if the variable names for the ID variable(s) differ, use `by.x` and `by.y` for the first and second data, respectively
- `all.x` and `all.y` indicate whether to keep all rows in the first and second data, respectively

Merging data

- thinking about a Venn diagram:
  - `all=TRUE` means *X OR Y*
  - `all=FALSE` means *X AND Y*
  - `all.x=TRUE, all.y=FALSE` means *X*
  - `all.x=FALSE, all.y=TRUE` means *Y*

Merging data

- checking work:

```
> nrow(ii)
[1] 4209
> nrow(diabetes)
[1] 9575
> nrow(ii.new)
[1] 4209
```

Merging data

- replacing the original data:

```
> ii<-ii.new
> rm(ii.new)
```

Reshaping data

Data structures

Wide data

- wide data refer to data structures in which each row is an individual (or animal, or village, or whatever the unit of analysis is)
- a key feature of wide data is that if there are repeated measures, the measurements appear as multiple columns
- for example, in the NHANES data, there are 4 variables for diastolic blood pressure: `bpxdi1`, `bpxdi2`, `bpxdi3`, and `bpxdi4`

Wide data

- diastolic blood pressure in NHANES:

```
> ii[1:4,c('seqn','bpxdi1','bpxdi2','bpxdi3','bpxdi4')]
   seqn bpxdi1 bpxdi2 bpxdi3 bpxdi4
1 83732     70     64     62     NA
2 83733     88     88     82     NA
3 83735     72     68     70     NA
4 83736     70     54     56     NA
```

Long data (tall data)

- long data refer to data structures in which each individual (or unit of analysis) may appear in more than one row
- I prefer thinking of long data as tall data

Tall data

- data from UNPS (*not* available on the CLE):

```
> uu[uu$pid=='10830027090305',c('pid','wave','year')]
                pid wave year
1753 10830027090305 2009 2010
3355 10830027090305 2010 2011
4778 10830027090305 2011 2012
```

- each row represents a person-wave

Tall data

- a hypothetical longitudinal study of 2 individuals, with 3 visits per individual:

```
> fake
  individual visit systolic diastolic
1          1     1      116        70
2          1     2      156        58
3          1     3      124        88
4          2     1      124        44
5          2     2      112        60
6          2     3      124        78
```

- each row represents a person-visit

Tall data

- a hypothetical cross-sectional study of 2 individuals, with information collected on up to 5 sexual partners per individual:

```
> fake
  individual partner partner.age condomless
1          1       1          25          1
2          1       2          32          0
3          1       3          27          0
4          1       4          29          1
5          2       1          42          0
6          2       2          46          0
```

- each row represents a sexual partnership

Wide or tall?

- what we consider wide or tall depends on what we consider to be the unit of analysis

Wide to tall manually

Reshaping wide to tall manually

- suppose we want to reshape the NHANES data from wide to tall, using only the first 2 measurements of diastolic blood pressure
- we could do this manually using `data.frame()` and `rbind()`:

```
> dia1<-data.frame(seqn=ii$seqn,order=1,diastolic=ii$bpxdi1)
> dia2<-data.frame(seqn=ii$seqn,order=2,diastolic=ii$bpxdi2)
> mm<-rbind(dia1,dia2)
```

Reshaping wide to tall manually

- checking work:

```
> head(mm[order(mm$seqn),])
      seqn order diastolic
1    83732     1        70
4210 83732     2        64
2    83733     1        88
4211 83733     2        88
3    83735     1        72
4212 83735     2        68
```

Reshaping wide to tall manually

- checking work:

```
> nrow(ii)*2
[1] 8418
> nrow(mm)
[1] 8418
> length(unique(mm$seqn))
[1] 4209
> nrow(ii)
[1] 4209
```

Reshaping wide to tall manually

- we have 4 measurements of diastolic blood pressure, not just 2
- it would be slightly tedious to create a tall data frame of all 4 diastolic measurements manually
- with other data sets, and more complex requirements, this would be even more tedious
- so, it is useful to use existing functions rather than reshape the data manually

Wide to tall using `reshape()`

Reshaping wide to tall using `reshape()`

- optionally, first store the names of the repeating variables:

```
> vv<-c('bpxdi1','bpxdi2','bpxdi3','bpxdi4')
```

- for simplicity of presentation on these slides, let's define a subset of `ii`:

```
> ss<-ii[,c('seqn','ridageyr',vv)]
```

Reshaping wide to tall using `reshape()`

```
mm<-reshape(data=ss,varying=vv,sep='',direction='long',
            idvar='seqn',timevar='order')
```

- `data`: the data to reshape
- `varying`: the variable names of the repeated measurements
- `sep`: if `varying` use a naming convention of *stub name* plus *some number*, `sep` is the character separating the stub and the number
- `direction`: the direction of reshaping
- `idvar` (*optional*): an ID variable, possibly using a variable name in the original data (if you don't include this, R will create a new variable called `id`)
- `timevar` (*optional*): the name for the new variable that will distinguish rows (the default name is `time`)

Reshaping wide to tall using `reshape()`

- checking work:

```
> head(mm[order(mm$seqn),])
         seqn ridageyr order bpxdi
83732.1 83732       62     1    70
83732.2 83732       62     2    64
83732.3 83732       62     3    62
83732.4 83732       62     4    NA
83733.1 83733       53     1    88
83733.2 83733       53     2    88
```

Reshaping wide to tall using `reshape()`

- `reshape()` adds row names
- to optionally rename the rows, use `row.names()`:

```
> head(row.names(mm))
[1] "83732.1" "83733.1" "83735.1" "83736.1" "83741.1" "83742.1"
> row.names(mm)<-NULL
> head(row.names(mm))
[1] "1" "2" "3" "4" "5" "6"
```

Reshaping wide to tall using `reshape()`

- checking work:

```
> nrow(ss)*4
[1] 16836
> nrow(mm)
[1] 16836
> length(unique(mm$seqn))
[1] 4209
> length(unique(ss$seqn))
[1] 4209
```

Reshaping wide to tall using `reshape()`

- what if we omit `idvar` and `timevar`?

```
> mm.simple<-reshape(data=ss,varying=vv,sep='',direction='long')
```

Reshaping wide to tall using `reshape()`

```
> mm[mm$seqn==86032,]
       seqn ridageyr order bpxdi
996   86032       50     1    84
5205  86032       50     2    90
9414  86032       50     3    94
13623 86032       50     4    NA
> mm.simple[mm.simple$seqn==86032,]
       seqn ridageyr time bpxdi  id
996.1 86032       50    1    84 996
996.2 86032       50    2    90 996
996.3 86032       50    3    94 996
996.4 86032       50    4    NA 996
```

Reshaping wide to tall using `reshape()`

- what if we want to include systolic blood pressure as well?
- optionally, first store the names of the repeating variables:

```
> vv<-c(paste('bpxdi',1:4,sep=''),paste('bpxsy',1:4,sep=''))
```

- again, for simplicity of presentation on these slides, let's define a subset of `ii`:

```
> ss<-ii[,c('seqn','ridageyr',vv)]
```

Reshaping wide to tall using `reshape()`

```
mm<-reshape(data=ss,varying=vv,sep='',direction='long',
            idvar='seqn',timevar='order')
```

Reshaping wide to tall using `reshape()`

- checking work:

```
> nrow(ss)*4
[1] 16836
> nrow(mm)
[1] 16836
> length(unique(mm$seqn))
[1] 4209
> length(unique(ss$seqn))
[1] 4209
```

Reshaping wide to tall using `reshape()`

- checking work:

```
> head(mm[order(mm$seqn),])
         seqn ridageyr order bpxdi bpxsy
83732.1 83732       62     1    70   128
83732.2 83732       62     2    64   124
83732.3 83732       62     3    62   116
83732.4 83732       62     4    NA    NA
83733.1 83733       53     1    88   146
83733.2 83733       53     2    88   140
```

Exercise

Exercise

You may work in groups of 2–3.

Use `presidencies.csv` for the next few problems.

1. If you have not already, convert `start`, `end`, and `dob` to Date objects.
2. Find the president who was youngest at the start of his term. How old was he? Find the president who was oldest at the start of his term. How old was he?

More on the next slide...

Exercise

3. Find the president who was youngest at the end of his term. How old was he? Find the president who was oldest at the end of his term. How old was he?
4. Find the president with the longest term. How long was that term? Find the president with the shortest term. How long was that term?

More on the next slide...

Exercise

Use `DEMO_I.XPT` and `DR1TOT_I` for the next few problems.

5. Subset the demographic data set to individuals 50 years of age (`RIDAGEYR`) or younger. Use this reduced demographic data set for the subsequent problems.
6. Check for duplicated ID (`SEQN`) in each data set.

More on the next slide...

Exercise

7. Merge the dietary data into the demographic data by ID (`SEQN`), using `all=TRUE`, and giving the new data frame a new arbitrary name. Check the number of rows. This should match the following, since `all=TRUE` means *X OR Y*:

```
> length(unique(c(tt1$seqn,tt2$seqn)))
[1] 9847
```

More on the next slide...

Exercise

8. Merge the dietary data into the demographic data by ID (`SEQN`), using `all=FALSE`, and giving the new data frame a new arbitrary name. Check the number of rows. This should match the following, since `all=FALSE` means *X AND Y*:

```
> sum(is.element(tt1$seqn,tt2$seqn))
[1] 6930
> sum(is.element(tt2$seqn,tt1$seqn))
[1] 6930
```

More on the next slide...

Exercise

9. Merge the dietary data into the demographic data by ID (`SEQN`), using `all.x=TRUE, all.y=FALSE`, and giving the new data frame a new arbitrary name. Check the number of rows. This should match the following, since `all.x=TRUE, all.y=FALSE` means simply X:

```
> length(unique(tt1$seqn))
[1] 7233
```

More on the next slide...

Exercise

10. Merge the dietary data into the demographic data by ID (`SEQN`), using `all.x=FALSE, all.y=TRUE`, and giving the new data frame a new arbitrary name. Check the number of rows. This should match the following, since `all.x=FALSE, all.y=TRUE` means simply Y:

```
> length(unique(tt2$seqn))
[1] 9544
```