

2657 R FUNCTIONS

Maintained by
Ananda Mahto

Last updated
December 26, 2012

2657 PRODUCTIONS

SANTA BARBARA, CALIFORNIA, USA

CHENNAI, TAMIL NADU, INDIA

2657 R FUNCTIONS

2657 PRODUCTIONS
SANTA BARBARA, CALIFORNIA, USA
CHENNAI, TAMIL NADU, INDIA

The scripts and documentation within this collection © 2012–2012 by Ananda Mahto under a “Creative Commons *Attribution-ShareAlike* license”. See <http://creativecommons.org/licenses/by-sa/3.0/>.

- Partial script contributions by:
 - Ben Bolker <http://www.math.mcmaster.ca/~bolker>, <http://stackoverflow.com/users/190277/ben-bolker>: `stringseed.sampling`
 - cbeleites <http://stackoverflow.com/users/755257/cbeleites>: `which.quantile` funtion in `row.extractor`
 - David Winsemius <http://stackoverflow.com/users/1855677/dwin>: `concat.split`
 - Justin <http://stackoverflow.com/users/906490/justin>: `multi.freq.table`
- Relevant questions or answers on Stack Overflow:
 - `concat.split`: <http://stackoverflow.com/q/10100887/1270695>; <http://stackoverflow.com/a/13912721/1270695>
 - `multi.freq.table`: <http://stackoverflow.com/q/11348391/1270695>; <http://stackoverflow.com/a/11623623/1270695>
 - `row.extractor`: <http://stackoverflow.com/q/10256503/1270695>
 - `stringseed.sampling`: <http://stackoverflow.com/q/10910698/1270695>
- “Borrowed” functions:
 - `LinearizeNestedList` function (loaded automatically when the `CBIND` function is run) by Akhil S Bhel: https://sites.google.com/site/akhilsbehl/geekspace/articles/r/linearize_nested_lists_in_r
 - `mv` function by Rolf Turner: (<https://stat.ethz.ch/pipermail/r-help/2008-March/156035.html>)
 - `round2` function by an anonymous commenter at the *Statistically Significant* blog (see: <http://www.webcitation.org/68djeLBtJ>). See also: <http://stackoverflow.com/q/12688717/1270695>

ANANDA MAHTO

<http://news.mrdwab.com>

<http://stackoverflow.com/users/1270695/ananda-mahto>

<https://github.com/mrdwab>

E-mail: ananda@mahto.info

Contents

I	Function Descriptions and Examples	1
1	concat.split	3
	Arguments	3
	Examples	3
	Advanced Usage	6
	References	9
2	df.sorter	11
	Arguments	11
	Examples	11
	To Do	14
3	multi.freq.table	15
	Arguments	15
	Examples	16
	Boolean Data	16
	Non-Boolean Data	18
	Extended Examples	19
	References	21
4	RandomNames	23
	Arguments	23
	Dataset Details	23
	Examples	24
	Using Your Own Data	26
	References	26
5	row.extractor	27
	Arguments	27
	Examples	27
	To Do	28
	References	28

6	sample.size	29
	Arguments	29
	Examples	29
	Advanced Usage	30
	References	31
7	stratified	33
	Arguments	33
	Examples	33
	Additional Information	36
	seed	36
	strata.sampling	38
	To Do	38
	References	39
8	stringseed.sampling	41
	Arguments	41
	Examples	41
	References	42
II	The Functions	43
9	Where to Get the Functions	45
10	concat.split	47
11	df.sorter	51
12	multi.freq.table	53
13	RandomNames	57
14	row.extractor	59
15	sample.size	61
16	stratified	63
17	stringseed.sampling	65

III Snippets and Tips 67

18 Snippets 69

Load All Scripts and Data Files From Multiple Directories	69
Convert a List of Data Frames Into Individual Data Frames	69
Example	69
Convert a Data Frame Into a List With Each Column Becoming a List Item	70
Examples	71
Rename an Object in the Workplace	71
Basic Usage	72
Scrape Data From a Poorly Formatted HTML Page	72
Example	72
“Rounding in Commerce”	72
Example	72
References	73
<code>cbind</code> <code>data.frames</code> When the Number of Rows are Not Equal	73
Examples	73
Generate Random Names With an Online Service	75
Arguments	75
Examples	75
Use strings to set seed when generating a random sample	76
Draw a stratified random sample from a <code>data.frame</code>	77
Examples	77

19 Tips 81

Batch Convert Factor Variables to Character Variables	81
Using Reduce to Merge Multiple Data Frames at Once	81
How Much Memory Are the Objects in Your Workspace Using?	82
Convert a Table to a Data Frame	82

IV Appendices 85

A Sample Generator for Students at the Tata-Dhan Academy 87

The <code>TDASample()</code> Function	88
Function Arguments	89
Examples	89
Advanced Example	91
How the Function Works	93

Part I

Function Descriptions and Examples

Chapter 1

concat.split

The `concat.split` function takes a column with multiple values, splits the values into a list or into separate columns, and returns a new `data.frame`.

Arguments

- **data**: the source `data.frame`.
- **split.col**: the variable that needs to be split; can be specified either by the column number or the variable name.
- **sep**: the character separating each value (defaults to ",").

Note: If using `structure = "compact"`, the value for `sep` can only be a single character. See the “Advanced Usage” example of how to specify multiple characters for batch conversion of columns.

- **structure**: Can be either "compact", "expanded", or "list". Defaults to "compact".
 - "compact" creates as many columns as the maximum *length* of the resulting split. This is the most useful general-case application of this function.
 - When the input is numeric, "expanded" creates as many columns as the maximum *value* of the input data. This is most useful when converting to `mode = "binary"`.
 - "list" creates a single new column that is structurally a `list` within a `data.frame`.
- **mode**: can be either `binary` or `value` (where `binary` is default and it recodes values to 1 or NA, like Boolean data, but without assuming 0 when data is not available). This setting only applies when `structure = "expanded"`; an warning message will be issued if used with other structures.
- **drop.col**: logical (whether to remove the original variable from the output or not; defaults to `TRUE`).
- **fixed**: Is the input for the `sep` value *fixed*, or a *regular expression*? When `structure = "expanded"` or `structure = "list"`, it is possible to supply a regular expression containing the characters to split on. For example, to split on ",", ";", or "|", you can set `sep = ",|;|\\|"` or `sep = "[,;|]"`, and `fixed = FALSE` to split on *any* of those characters.

Examples

First load some data from a CSV stored at [github](#). The URL is an HTTPS, so we need to use `getURL` from `Rcurl`.

```
require(RCurl)
baseURL = c("https://raw.githubusercontent.com/mrdwab/2657-R-Functions/master/")
temp = getURL(paste0(baseURL, "data/concatenated-cells.csv"))
concat.test = read.csv(textConnection(temp))
rm(temp)
```

How big is the dataset?

```
dim(concat.test)
```

```
## [1] 48 4
```

Just show me the first few rows

```
head(concat.test)
```

```
##      Name      Likes      Siblings      Hates
## 1   Boyd 1,2,4,5,6 Reynolds , Albert , Ortega 2;4;
## 2  Rufus 1,2,4,5,6 Cohen , Bert , Montgomery 1;2;3;4;
## 3   Dana 1,2,4,5,6      Pierce      2;
## 4 Carole 1,2,4,5,6 Colon , Michelle , Ballard 1;4;
## 5 Ramona 1,2,5,6      Snyder , Joann , 1;2;3;
## 6 Kelley 1,2,5,6      James , Roxanne , 1;4;
```

Notice that the data have been entered in a very silly manner. Let's split it up!

Load the function!

```
# require(RCurl)
```

```
# baseURL = c("https://raw.githubusercontent.com/mrdwab/2657-R-Functions/master/")
```

```
source(textConnection(getURL(paste0(baseURL, "scripts/concat.split.R"))))
```

Split up the second column, selecting by column number

```
head(concat.split(concat.test, 2))
```

```
##      Name      Likes      Siblings      Hates Likes_1 Likes_2 Likes_3
## 1   Boyd 1,2,4,5,6 Reynolds , Albert , Ortega 2;4;      1      2      4
## 2  Rufus 1,2,4,5,6 Cohen , Bert , Montgomery 1;2;3;4;      1      2      4
## 3   Dana 1,2,4,5,6      Pierce      2;      1      2      4
## 4 Carole 1,2,4,5,6 Colon , Michelle , Ballard 1;4;      1      2      4
## 5 Ramona 1,2,5,6      Snyder , Joann , 1;2;3;      1      2      5
## 6 Kelley 1,2,5,6      James , Roxanne , 1;4;      1      2      5
## Likes_4 Likes_5
## 1      5      6
## 2      5      6
## 3      5      6
## 4      5      6
## 5      6     NA
## 6      6     NA
```

... or by name, and drop the offensive first column

```
head(concat.split(concat.test, "Likes", drop.col = TRUE))
```

```
##      Name      Siblings      Hates Likes_1 Likes_2 Likes_3 Likes_4
## 1   Boyd Reynolds , Albert , Ortega 2;4;      1      2      4      5
## 2  Rufus Cohen , Bert , Montgomery 1;2;3;4;      1      2      4      5
## 3   Dana      Pierce      2;      1      2      4      5
## 4 Carole Colon , Michelle , Ballard 1;4;      1      2      4      5
## 5 Ramona      Snyder , Joann , 1;2;3;      1      2      5      6
```

```
## 6 Kelley          James , Roxanne ,      1;4;      1      2      5      6
## Likes_5
## 1      6
## 2      6
## 3      6
## 4      6
## 5      NA
## 6      NA
```

The "Hates" column uses a different separator:

```
head(concat.split(concat.test, "Hates", sep = ";", drop.col = TRUE))
```

```
##      Name      Likes      Siblings Hates_1 Hates_2 Hates_3 Hates_4
## 1   Boyd 1,2,4,5,6 Reynolds , Albert , Ortega      2      4      NA      NA
## 2   Rufus 1,2,4,5,6 Cohen , Bert , Montgomery      1      2      3      4
## 3    Dana 1,2,4,5,6      Pierce      2      NA      NA      NA
## 4 Carole 1,2,4,5,6 Colon , Michelle , Ballard      1      4      NA      NA
## 5 Ramona 1,2,5,6      Snyder , Joann ,      1      2      3      NA
## 6 Kelley 1,2,5,6      James , Roxanne ,      1      4      NA      NA
## Hates_5
## 1      NA
## 2      NA
## 3      NA
## 4      NA
## 5      NA
## 6      NA
```

You'll get a warning here, when trying to retain the original values

```
head(concat.split(concat.test, 2, mode = "value", drop.col = TRUE))
```

```
## Warning: 'mode' supplied but ignored. 'mode' setting only applicable when
## structure='expanded'.
```

```
##      Name      Siblings      Hates Likes_1 Likes_2 Likes_3 Likes_4
## 1   Boyd Reynolds , Albert , Ortega      2;4;      1      2      4      5
## 2   Rufus Cohen , Bert , Montgomery 1;2;3;4;      1      2      4      5
## 3    Dana      Pierce      2;      1      2      4      5
## 4 Carole Colon , Michelle , Ballard      1;4;      1      2      4      5
## 5 Ramona      Snyder , Joann ,      1;2;3;      1      2      5      6
## 6 Kelley James , Roxanne ,      1;4;      1      2      5      6
## Likes_5
## 1      6
## 2      6
## 3      6
## 4      6
## 5      NA
## 6      NA
```

Try again. Notice the differing number of resulting columns

```
head(concat.split(concat.test, 2, structure = "expanded",
mode = "value", drop.col = TRUE))
```

```
##      Name      Siblings      Hates Likes_1 Likes_2 Likes_3 Likes_4
## 1   Boyd Reynolds , Albert , Ortega      2;4;      1      2      NA      4
## 2   Rufus Cohen , Bert , Montgomery 1;2;3;4;      1      2      NA      4
## 3    Dana      Pierce      2;      1      2      NA      4
```

```
## 4 Carole Colon , Michelle , Ballard      1;4;      1      2      NA      4
## 5 Ramona      Snyder , Joann ,      1;2;3;      1      2      NA      NA
## 6 Kelley      James , Roxanne ,      1;4;      1      2      NA      NA
## Likes_5 Likes_6
## 1      5      6
## 2      5      6
## 3      5      6
## 4      5      6
## 5      5      6
## 6      5      6
```

```
# Let's try splitting some strings... Same syntax
head(concat.split(concat.test, 3, drop.col = TRUE))
```

```
##      Name      Likes      Hates Siblings_1 Siblings_2 Siblings_3
## 1   Boyd 1,2,4,5,6      2;4; Reynolds      Albert      Ortega
## 2   Rufus 1,2,4,5,6 1;2;3;4;      Cohen      Bert      Montgomery
## 3    Dana 1,2,4,5,6      2;      Pierce
## 4 Carole 1,2,4,5,6      1;4;      Colon      Michelle      Ballard
## 5 Ramona      1,2,5,6      1;2;3;      Snyder      Joann
## 6 Kelley      1,2,5,6      1;4;      James      Roxanne
```

```
# Split up the "Likes column" into a list variable; retain original column
head(concat.split(concat.test, 2, structure = "list", drop.col=FALSE))
```

```
##      Name      Likes      Siblings      Hates      Likes_list
## 1   Boyd 1,2,4,5,6 Reynolds , Albert , Ortega      2;4; 1, 2, 4, 5, 6
## 2   Rufus 1,2,4,5,6 Cohen , Bert , Montgomery 1;2;3;4; 1, 2, 4, 5, 6
## 3    Dana 1,2,4,5,6      Pierce      2; 1, 2, 4, 5, 6
## 4 Carole 1,2,4,5,6 Colon , Michelle , Ballard      1;4; 1, 2, 4, 5, 6
## 5 Ramona      1,2,5,6      Snyder , Joann ,      1;2;3;      1, 2, 5, 6
## 6 Kelley      1,2,5,6      James , Roxanne ,      1;4;      1, 2, 5, 6
```

```
# View the structure of the output for the first 10 rows to verify
# that the new column is a list; note the difference between "Likes"
# and "Likes_list".
```

```
str(concat.split(concat.test, 2, structure = "list",
                  drop.col=FALSE)[1:10, c(2, 5)])
```

```
## 'data.frame':      10 obs. of  2 variables:
## $ Likes      : Factor w/ 5 levels "1,2,3,4,5","1,2,4,5",...: 3 3 3 3 5 5 3 3 3 4
## $ Likes_list:List of 10
## ..$ : num  1 2 4 5 6
## ..$ : num  1 2 4 5 6
## ..$ : num  1 2 4 5 6
## ..$ : num  1 2 4 5 6
## ..$ : num  1 2 5 6
## ..$ : num  1 2 5 6
## ..$ : num  1 2 4 5 6
## ..$ : num  1 2 4 5 6
## ..$ : num  1 2 4 5 6
## ..$ : num  1 2 5
```

Advanced Usage

It is also possible to use `concat.split` to split multiple columns at once. This can be done in stages, or it can be all wrapped in nested statements, as follows:

```
do.call(cbind,
  c(concat.test[1],
    lapply(1:(ncol(concat.test)-1),
      function(x) {
        splitchars = c(",", " ", ";")
        concat.split(concat.test[-1][x], 1,
          splitchars[x],
          drop.col=TRUE)
      })
  )))
```

In the example above:

- The `lapply()` function is applied to each column in the `data.frame` except the first one.
- Before applying the `concat.split` function, we enter a vector of the characters on which we should split, in the same order as the columns. Here, the first two columns are separated by commas, and the third is separated by a semicolon.
- The `concat.split` function arguments can then be included as you would if splitting a single column.
- We use `do.call(cbind, ...)` to “bind” the data together by columns. Since we had dropped the first column for the `lapply` step, we add that back in at this stage.

Show just the first few lines, compact structure
Note that the split characters must be specified
in the same order that lapply will encounter them

```
head(do.call(cbind,
  c(concat.test[1],
    lapply(1:(ncol(concat.test)-1),
      function(x) {
        splitchars = c(",", " ", ";")
        concat.split(concat.test[-1][x], 1,
          splitchars[x],
          drop.col=TRUE)
      })
  )))
```

##	Name	Likes_1	Likes_2	Likes_3	Likes_4	Likes_5	Siblings_1	Siblings_2
## 1	Boyd	1	2	4	5	6	Reynolds	Albert
## 2	Rufus	1	2	4	5	6	Cohen	Bert
## 3	Dana	1	2	4	5	6	Pierce	
## 4	Carole	1	2	4	5	6	Colon	Michelle
## 5	Ramona	1	2	5	6	NA	Snyder	Joann
## 6	Kelley	1	2	5	6	NA	James	Roxanne

##	Siblings_3	Hates_1	Hates_2	Hates_3	Hates_4	Hates_5
## 1	Ortega	2	4	NA	NA	NA
## 2	Montgomery	1	2	3	4	NA
## 3		2	NA	NA	NA	NA
## 4	Ballard	1	4	NA	NA	NA
## 5		1	2	3	NA	NA
## 6		1	4	NA	NA	NA

Show just the first few lines, Boolean mode
Note the use of a regular expression for sep
and the setting of fixed to FALSE

```
head(do.call(cbind,
  c(concat.test[1],
```

```

lapply(1:(ncol(concat.test)-1),
  function(x) {
    concat.split(concat.test[-1][x], 1,
      sep = "[,;]",
      structure = "expanded",
      fixed = FALSE,
      drop.col=TRUE)
  })))

```

```

##      Name Likes_1 Likes_2 Likes_3 Likes_4 Likes_5 Likes_6 Siblings_1 Siblings_2
## 1   Boyd      1      1      NA      1      1      1   Reynolds   Albert
## 2  Rufus      1      1      NA      1      1      1     Cohen    Bert
## 3   Dana      1      1      NA      1      1      1     Pierce   <NA>
## 4 Carole      1      1      NA      1      1      1     Colon   Michelle
## 5 Ramona      1      1      NA      NA      1      1     Snyder   Joann
## 6 Kelley      1      1      NA      NA      1      1      James   Roxanne
##  Siblings_3 Hates_1 Hates_2 Hates_3 Hates_4
## 1   Ortega      NA      1      NA      1
## 2 Montgomery      1      1      1      1
## 3    <NA>      NA      1      NA      NA
## 4   Ballard      1      NA      NA      1
## 5    <NA>      1      1      1      NA
## 6    <NA>      1      NA      NA      1

```

```

# Show just the first few lines, value mode
head(do.call(cbind,
  c(concat.test[1],
    lapply(1:(ncol(concat.test)-1),
      function(x) {
        concat.split(concat.test[-1][x], 1,
          sep = "[,;]",
          structure = "expanded",
          mode = "value",
          fixed = FALSE,
          drop.col=TRUE)
      }))))

```

```

##      Name Likes_1 Likes_2 Likes_3 Likes_4 Likes_5 Likes_6 Siblings_1 Siblings_2
## 1   Boyd      1      2      NA      4      5      6   Reynolds   Albert
## 2  Rufus      1      2      NA      4      5      6     Cohen    Bert
## 3   Dana      1      2      NA      4      5      6     Pierce   <NA>
## 4 Carole      1      2      NA      4      5      6     Colon   Michelle
## 5 Ramona      1      2      NA      NA      5      6     Snyder   Joann
## 6 Kelley      1      2      NA      NA      5      6      James   Roxanne
##  Siblings_3 Hates_1 Hates_2 Hates_3 Hates_4
## 1   Ortega      NA      2      NA      4
## 2 Montgomery      1      2      3      4
## 3    <NA>      NA      2      NA      NA
## 4   Ballard      1      NA      NA      4
## 5    <NA>      1      2      3      NA
## 6    <NA>      1      NA      NA      4

```

```

# Show just the first few lines, list output mode
head(do.call(cbind,

```



```

c(concat.test[1],
  lapply(1:(ncol(concat.test)-1),
    function(x) {
      concat.split(concat.test[-1][x], 1,
        sep = "[,;]",
        structure = "list",
        fixed = FALSE,
        drop.col=TRUE)
    })
  )
)

```

```

##      Name      Likes_list      Siblings_list Hates_list
## 1   Boyd 1, 2, 4, 5, 6 Reynolds, Albert, Ortega      2, 4
## 2  Rufus 1, 2, 4, 5, 6 Cohen, Bert, Montgomery 1, 2, 3, 4
## 3   Dana 1, 2, 4, 5, 6                Pierce      2
## 4 Carole 1, 2, 4, 5, 6 Colon, Michelle, Ballard      1, 4
## 5 Ramona      1, 2, 5, 6          Snyder, Joann      1, 2, 3
## 6 Kelley      1, 2, 5, 6          James, Roxanne      1, 4

```

References

See: <http://stackoverflow.com/q/10100887/1270695>

The "condensed" setting was inspired by an answer from David Winsemius (@DWin) to a question at Stack Overflow. See: <http://stackoverflow.com/a/13924245/1270695>

Chapter 2

df.sorter

The `df.sorter` function allows you to sort a `data.frame` by columns or rows or both. You can also quickly subset data columns by using the `var.order` argument.

Arguments

- `data`: the source `data.frame`.
- `var.order`: the new order in which you want the variables to appear.
 - Defaults to `names(data)`, which keeps the variables in the original order.
 - Variables can be referred to either by a vector of their index numbers or by a vector of the variable name; partial name matching also works, but requires that the partial match identifies similar columns uniquely (see examples).
 - Basic subsetting can also be done using `var.order` simply by omitting the variables you want to drop.
- `col.sort`: the columns *within* which there is data that need to be sorted.
 - Defaults to `NULL`, which means no sorting takes place.
 - Variables can be referred to either by a vector of their index numbers or by a vector of the variable names; full names must be provided.
- `at.start`: Should the pattern matching be from the start of the variable name? Defaults to “TRUE”.

NOTE: If you are sorting both by variables and within the columns, the `col.sort` order should be based on the location of the columns in the *new data.frame*, not the original `data.frame`.

Examples

```
# Load the function!
require(RCurl)
baseURL = c("https://raw.githubusercontent.com/mrdwab/2657-R-Functions/master/")
source(textConnection(getURL(paste0(baseURL, "scripts/df.sorter.R"))))

# Make up some data
set.seed(1)
dat = data.frame(id = rep(1:5, each=3), times = rep(1:3, 5),
```

```

    measure1 = rnorm(15), score1 = sample(300, 15),
    code1 = replicate(15, paste(sample(LETTERS[1:5], 3),
                                sep="", collapse="")),
    measure2 = rnorm(15), score2 = sample(150:300, 15),
    code2 = replicate(15, paste(sample(LETTERS[1:5], 3),
                                sep="", collapse="")))

# Preview your data
dat

##      id times measure1 score1 code1 measure2 score2 code2
## 1     1     1  -0.6265   145   DAB  -0.7075   299   CEB
## 2     1     2   0.1836   180   DCB   0.3646   224   ECD
## 3     1     3  -0.8356   148   EBA   0.7685   222   DAE
## 4     2     1   1.5953    56   AED  -0.1123   175   DBA
## 5     2     2   0.3295   245   CEB   0.8811   260   DAC
## 6     2     3  -0.8205   198   EBD   0.3981   216   DCA
## 7     3     1   0.4874   234   BCA  -0.6120   300   CEA
## 8     3     2   0.7383    32   CDA   0.3411   179   CAD
## 9     3     3   0.5758   212   EBC  -1.1294   182   BEC
## 10    4     1  -0.3054   120   BED   1.4330   234   CDE
## 11    4     2   1.5118   239   EDB   1.9804   231   CAB
## 12    4     3   0.3898   188   DEB  -0.3672   160   DBE
## 13    5     1  -0.6212   226   DBA  -1.0441   154   EDB
## 14    5     2  -2.2147   159   DAC   0.5697   238   BDE
## 15    5     3   1.1249   152   AED  -0.1351   277   DCE

# Change the variable order, grouping related columns
# Note that you do not need to specify full variable names,
# just enough that the variables can be uniquely identified
head(df.sorter(dat, var.order = c("id", "ti", "cod", "mea", "sco")))

##      id times code1 code2 measure1 measure2 score1 score2
## 1     1     1   DAB   CEB  -0.6265  -0.7075   145   299
## 2     1     2   DCB   ECD   0.1836   0.3646   180   224
## 3     1     3   EBA   DAE  -0.8356   0.7685   148   222
## 4     2     1   AED   DBA   1.5953  -0.1123    56   175
## 5     2     2   CEB   DAC   0.3295   0.8811   245   260
## 6     2     3   EBD   DCA  -0.8205   0.3981   198   216

# Same output, but with a more awkward syntax
head(df.sorter(dat, var.order = c(1, 2, 5, 8, 3, 6, 4, 7)))

##      id times code1 code2 measure1 measure2 score1 score2
## 1     1     1   DAB   CEB  -0.6265  -0.7075   145   299
## 2     1     2   DCB   ECD   0.1836   0.3646   180   224
## 3     1     3   EBA   DAE  -0.8356   0.7685   148   222
## 4     2     1   AED   DBA   1.5953  -0.1123    56   175
## 5     2     2   CEB   DAC   0.3295   0.8811   245   260
## 6     2     3   EBD   DCA  -0.8205   0.3981   198   216

# As above, but sorted by ,times, and then ,id,
head(df.sorter(dat, var.order = c("id", "tim", "cod", "mea", "sco"),
                    col.sort = c(2, 1)))

##      id times code1 code2 measure1 measure2 score1 score2
## 1     1     1   DAB   CEB  -0.6265  -0.7075   145   299

```

```
## 4 2 1 AED DBA 1.5953 -0.1123 56 175
## 7 3 1 BCA CEA 0.4874 -0.6120 234 300
## 10 4 1 BED CDE -0.3054 1.4330 120 234
## 13 5 1 DBA EDB -0.6212 -1.0441 226 154
## 2 1 2 DCB ECD 0.1836 0.3646 180 224
```

```
# Drop ,measure1, and ,measure2,, sort by ,times,, and ,score1,
head(df.sorter(dat, var.order = c("id", "tim", "sco", "cod"),
  col.sort = c(2, 3)))
```

```
## id times score1 score2 code1 code2
## 4 2 1 56 175 AED DBA
## 10 4 1 120 234 BED CDE
## 1 1 1 145 299 DAB CEB
## 13 5 1 226 154 DBA EDB
## 7 3 1 234 300 BCA CEA
## 8 3 2 32 179 CDA CAD
```

```
# As above, but using names
head(df.sorter(dat, var.order = c("id", "tim", "sco", "cod"),
  col.sort = c("times", "score1")))
```

```
## id times score1 score2 code1 code2
## 4 2 1 56 175 AED DBA
## 10 4 1 120 234 BED CDE
## 1 1 1 145 299 DAB CEB
## 13 5 1 226 154 DBA EDB
## 7 3 1 234 300 BCA CEA
## 8 3 2 32 179 CDA CAD
```

```
# Just sort by columns, first by ,times, then by ,id,
head(df.sorter(dat, col.sort = c("times", "id")))
```

```
## id times measure1 score1 code1 measure2 score2 code2
## 1 1 1 -0.6265 145 DAB -0.7075 299 CEB
## 4 2 1 1.5953 56 AED -0.1123 175 DBA
## 7 3 1 0.4874 234 BCA -0.6120 300 CEA
## 10 4 1 -0.3054 120 BED 1.4330 234 CDE
## 13 5 1 -0.6212 226 DBA -1.0441 154 EDB
## 2 1 2 0.1836 180 DCB 0.3646 224 ECD
```

```
head(df.sorter(dat, col.sort = c("code1"))) # Sorting by character values
```

```
## id times measure1 score1 code1 measure2 score2 code2
## 4 2 1 1.5953 56 AED -0.1123 175 DBA
## 15 5 3 1.1249 152 AED -0.1351 277 DCE
## 7 3 1 0.4874 234 BCA -0.6120 300 CEA
## 10 4 1 -0.3054 120 BED 1.4330 234 CDE
## 8 3 2 0.7383 32 CDA 0.3411 179 CAD
## 5 2 2 0.3295 245 CEB 0.8811 260 DAC
```

```
# Pattern matching anywhere in the variable name
head(df.sorter(dat, var.order= "co", at.start=FALSE))
```

##	code1	code2	score1	score2
## 1	DAB	CEB	145	299
## 2	DCB	ECD	180	224
## 3	EBA	DAE	148	222
## 4	AED	DBA	56	175
## 5	CEB	DAC	245	260
## 6	EBD	DCA	198	216

To Do

- Add an option to sort ascending or descending—at the moment, not supported.

Chapter 3

multi.freq.table

The `multi.freq.table` function takes a data frame containing Boolean responses to multiple response questions and tabulates the number of responses by the possible combinations of answers. In addition to tabulating the frequency (**Freq**), there are two other columns in the output: *Percent of Responses* (**Pct.of.Resp**) and *Percent of Cases* (**Pct.of.Cases**). *Percent of Responses* is the frequency divided by the total number of answers provided; this column should sum to 100%. In some cases, for instance when a combination table is generated and there are cases where a respondent did not select any option, the *Percent of Responses* value would be more than 100%. *Percent of Cases* is the frequency divided by the total number of valid cases; this column would most likely sum to more than 100% when a basic table is produced since each respondent (case) can select multiple answers, but should sum to 100% with other tables.

Arguments

- **data**: The multiple responses that need to be tabulated.
- **sep**: The desired separator for collapsing the combinations of options; defaults to "" (collapsing with no space between each option name).
- **boolean**: Are you tabulating boolean data (see **dat** examples)? Defaults to **TRUE**.
- **factors**: If you are trying to tabulate non-boolean data, and the data are not factors, you can specify the factors here (see **dat2** examples).
 - Defaults to **NULL** and is not used when **boolean = TRUE**.
- **NAtO0**: Should NA values be converted to 0.
 - Defaults to **TRUE**, in which case, the number of valid cases should be the same as the number of cases overall.
 - If set to **FALSE**, any rows with NA values will be dropped as invalid cases.
 - Only applies when **boolean = TRUE**.
- **basic**: Should a basic table of each item, rather than combinations of items, be created? Defaults to **FALSE**.
- **dropzero**: Should combinations with a frequency of zero be dropped from the final table?
 - Defaults to **TRUE**.
 - Does not apply when **boolean = TRUE**.
- **clean**: Should the original tabulated data be retained or dropped from the final table?
 - Defaults to **TRUE**.
 - Does not apply when **boolean = TRUE**.

Examples

Boolean Data

```
# Load the function!
require(RCurl)
baseURL = c("https://raw.githubusercontent.com/mrdwab/2657-R-Functions/master/")
source(textConnection(getURL(paste0(baseURL, "scripts/multi.freq.table.R"))))
```

```
# Make up some data
set.seed(1)
dat = data.frame(A = sample(c(0, 1), 20, replace=TRUE),
                 B = sample(c(0, 1, NA), 20,
                           prob=c(.3, .6, .1), replace=TRUE),
                 C = sample(c(0, 1, NA), 20,
                           prob=c(.7, .2, .1), replace=TRUE),
                 D = sample(c(0, 1, NA), 20,
                           prob=c(.3, .6, .1), replace=TRUE),
                 E = sample(c(0, 1, NA), 20,
                           prob=c(.4, .4, .2), replace=TRUE))
```

```
# View your data
dat
```

```
##      A  B C  D  E
## 1    0 NA 1 NA  0
## 2    0  1 0  1  0
## 3    1  0 1  1  1
## 4    1  1 0  1  1
## 5    0  1 0  0  0
## 6    1  1 1  1  1
## 7    1  1 0  1  0
## 8    1  1 0  0  1
## 9    1  0 1  1  1
## 10   0  1 0  0  1
## 11   0  1 0  1  1
## 12   0  1 1  0  1
## 13   1  1 0  1  0
## 14   0  1 0  1 NA
## 15   1  0 0  1  0
## 16   0  0 0  0  0
## 17   1  0 0  0  0
## 18   1  1 0  1  0
## 19   0  0 0  0 NA
## 20   1  1 0 NA  0
```

```
# How many cases have "NA" values?
table(is.na(rowSums(dat)))
```

```
##
## FALSE  TRUE
##      16      4
```

```
# Apply the function with all defaults accepted
multi.freq.table(dat)
```

```
##      Combn Freq Weighted.Freq Pct.of.Resp Pct.of.Cases
```



```
## 1      2      2      4.167      10
## 2      A      1      1      2.083      5
## 3      B      1      1      2.083      5
## 4      AB     1      2      4.167      5
## 5      C      1      1      2.083      5
## 6      AD     1      2      4.167      5
## 7      BD     2      4      8.333      10
## 8      ABD    3      9     18.750      15
## 9      BE     1      2      4.167      5
## 10     ABE     1      3      6.250      5
## 11     BCE     1      3      6.250      5
## 12     BDE     1      3      6.250      5
## 13     ABDE    1      4      8.333      5
## 14     ACDE    2      8     16.667      10
## 15     ABCDE   1      5     10.417      5
```

```
# Tabulate only on variables "A", "B", and "D", with a different
# separator, keep any zero frequency values, and keeping the
# original tabulations. There are no solitary "D" responses.
multi.freq.table(dat[c(1, 2, 4)], sep="-", dropzero=FALSE, clean=FALSE)
```

```
##   A B D Freq Combn Weighted.Freq Pct.of.Resp Pct.of.Cases
## 1 0 0 0    3      3      8.571      15
## 2 1 0 0    1      A      2.857      5
## 3 0 1 0    3      B      8.571      15
## 4 1 1 0    2     A-B     11.429      10
## 5 0 0 1    0      D      0.000      0
## 6 1 0 1    3     A-D     17.143      15
## 7 0 1 1    3     B-D     17.143      15
## 8 1 1 1    5  A-B-D     42.857      25
```

```
# As above, but without converting "NA" to "0".
# Note the difference in the number of valid cases.
multi.freq.table(dat[c(1, 2, 4)], NAto0=FALSE,
                  sep="-", dropzero=FALSE, clean=FALSE)
```

```
##   A B D Freq Combn Weighted.Freq Pct.of.Resp Pct.of.Cases
## 1 0 0 0    2      2      6.061     11.111
## 2 1 0 0    1      A      3.030      5.556
## 3 0 1 0    3      B      9.091     16.667
## 4 1 1 0    1     A-B      6.061      5.556
## 5 0 0 1    0      D      0.000      0.000
## 6 1 0 1    3     A-D     18.182     16.667
## 7 0 1 1    3     B-D     18.182     16.667
## 8 1 1 1    5  A-B-D     45.455     27.778
```

```
# View a basic table.
multi.freq.table(dat, basic=TRUE)
```

```
##   Freq Pct.of.Resp Pct.of.Cases
## A    11      22.92      55
## B    13      27.08      65
## C     5      10.42      25
## D    11      22.92      55
## E     8      16.67      40
```

Non-Boolean Data

```
# Make up some data
dat2 = structure(list(Reason.1 = c("one", "one", "two", "one", "two",
    "three", "one", "one", NA, "two"),
    Reason.2 = c("two", "three", "three", NA, NA,
    "two", "three", "two", NA, NA),
    Reason.3 = c("three", NA, NA, NA, NA,
    NA, NA, "three", NA, NA)),
    .Names = c("Reason.1", "Reason.2", "Reason.3"),
    class = "data.frame",
    row.names = c(NA, -10L))

# View your data
dat2
```

```
##      Reason.1 Reason.2 Reason.3
## 1         one      two    three
## 2         one    three    <NA>
## 3         two    three    <NA>
## 4         one    <NA>    <NA>
## 5         two    <NA>    <NA>
## 6      three      two    <NA>
## 7         one    three    <NA>
## 8         one      two    three
## 9        <NA>    <NA>    <NA>
## 10        two    <NA>    <NA>
```

```
# The following will not work.
# The data are not factored.
multi.freq.table(dat2, boolean=FALSE)
```

```
## Error: Input variables must be factors. Please provide factors using the
## 'factors' argument or convert your data to factor before using function.
```

```
# Factor create the factors.
multi.freq.table(dat2, boolean=FALSE,
    factors = c("one", "two", "three"))
```

```
##      Combos Freq Weighted.Freq Pct.of.Resp Pct.of.Cases
## 1          1      1           1      5.882         10
## 8         one      1           1      5.882         10
## 12        two      2           2     11.765         20
## 15    onethree      2           4     23.529         20
## 17   threetwo      2           4     23.529         20
## 22 onethreetwo      2           6     35.294         20
```

```
# And, a basic table.
multi.freq.table(dat2, boolean=FALSE,
    factors = c("one", "two", "three"),
    basic=TRUE)
```

```
##      Item Freq Pct.of.Resp Pct.of.Cases
## 1  one     5      29.41         50
## 2  two     6      35.29         60
## 3 three     6      35.29         60
```

Extended Examples

The following example is based on some data available from the University of Auckland's Student Learning Resources¹.

When the data are read into R, the factor labels are very long, which makes it difficult to see on the screen. Thus, in the first example that follows, the factor levels are first recoded before the multiple frequency tables are created. Additionally, the data for the binary information in the second example was coded in a common 1 = Yes and 2 = No format, but we need 0 = No instead, so we need to do some recoding there too before using the function.

```
# Get the data
library(foreign)
temp = "http://cad.auckland.ac.nz/file.php/content/files/slc/"
computer = read.spss(paste0(temp,
                             "computer_multiple_response.sav"),
                    to.data.frame=TRUE)

rm(temp)
# Preview
dim(computer)

## [1] 100 20

names(computer)

## [1] "id"          "ms_word"    "ms_excel"   "ms_ppt"     "ms_outlk"   "ms_pub"
## [7] "ms_proj"    "ms_acc"     "netscape"  "int_expl"   "adobe_rd"   "endnote"
## [13] "spss"       "quality1"   "quality2"   "quality3"   "quality4"   "quality5"
## [19] "quality6"   "gender"

# First, let's just tabulate the instructor qualities.
# Extract the relevant columns, and relevel the factors.
instructor.quality =
  computer[, grep("quali", names(computer))]
# View the existing levels.
lapply(instructor.quality, levels)[[1]]

## [1] "Ability to provide practical examples"
## [2] "Ability to answer questions positively"
## [3] "Ability to clearly explain concepts"
## [4] "Ability to instruct at a suitable pace"
## [5] "Knowledge of software"
## [6] "Humour"
## [7] "Other"

instructor.quality = lapply(instructor.quality,
                           function(x) { levels(x) =
list(Q1 = "Ability to provide practical examples",
      Q2 = "Ability to answer questions positively",
      Q3 = "Ability to clearly explain concepts",
      Q4 = "Ability to instruct at a suitable pace",
      Q5 = "Knowledge of Software",
      Q6 = "Humour", Q7 = "Other"); x })
# Now, apply multi.freq.table to the data.
multi.freq.table(data.frame(instructor.quality),
                 boolean=FALSE, basic=TRUE)
```

¹See: <http://www.cad.auckland.ac.nz/index.php?p=spss>

```
##      Item Freq Pct.of.Resp Pct.of.Cases
## 1    Q1   47      18.077         47
## 2    Q2   59      22.692         59
## 3    Q3   55      21.154         55
## 4    Q4   43      16.538         43
## 5    Q5    0       0.000          0
## 6    Q6   47      18.077         47
## 7    Q7    9       3.462          9
```

```
list(head(multi.freq.table(data.frame(instructor.quality),
                                     boolean=FALSE, sep="-")),
      tail(multi.freq.table(data.frame(instructor.quality),
                                     boolean=FALSE, sep="-")))
```

```
## [[1]]
##      Combos Freq Weighted.Freq Pct.of.Resp Pct.of.Cases
## 1      Q1    1           1      0.3846         1
## 21     Q2    3           3      1.1538         3
## 31     Q3    2           2      0.7692         2
## 37     Q4    2           2      0.7692         2
## 39     Q6    3           3      1.1538         3
## 41    Q1-Q2   8          16      6.1538         8
##
## [[2]]
##      Combos Freq Weighted.Freq Pct.of.Resp Pct.of.Cases
## 133    Q1-Q3-Q6-Q7    1           4      1.538         1
## 141    Q2-Q3-Q4-Q6    4          16      6.154         4
## 151    Q3-Q4-Q6-Q7    1           4      1.538         1
## 161    Q1-Q2-Q3-Q4-Q6    1           5      1.923         1
## 164    Q1-Q2-Q3-Q6-Q7    1           5      1.923         1
## 201    Q1-Q2-Q3-Q4-Q6-Q7    1           6      2.308         1
```

```
# Now, let's look at the software.
instructors.sw = computer[2:13]
# These columns are coded as 1 = Yes and 2 = No,
# so, convert to integers, and subtract two, and
# take the absolute value to convert to binary.
instructors.sw = lapply(instructors.sw,
                        function(x) abs(as.integer(x)-2))
# Apply multi.freq.table
multi.freq.table(data.frame(instructors.sw), basic=TRUE)
```

```
##      Freq Pct.of.Resp Pct.of.Cases
## ms_word   77      13.975         77
## ms_excel  48       8.711         48
## ms_ppt    55       9.982         55
## ms_outlk  52       9.437         52
## ms_pub    19       3.448         19
## ms_proj   21       3.811         21
## ms_acc    57      10.345         57
## netscape  10       1.815         10
## int_expl  84      15.245         84
## adobe_rd  48       8.711         48
## endnote   55       9.982         55
## spss      25       4.537         25
```

```
# The output here is not pretty. To get prettier (or more meaningful)
# output, provide shorter names for the variables or use just a
```

```
# meaningful subset of the variables.
list(head(multi.freq.table(data.frame(instructors.sw), sep="-")),
      tail(multi.freq.table(data.frame(instructors.sw), sep="-")))

## [[1]]
##
##                               Combn Freq Weighted.Freq Pct.of.Resp
## 1                ms_word-ms_excel-ms_ppt-ms_acc      1           4      0.7260
## 2 ms_word-ms_excel-ms_ppt-ms_outlk-ms_pub-ms_acc      1           6      1.0889
## 3                                int_expl      2           2      0.3630
## 4                        ms_word-int_expl      1           2      0.3630
## 5                ms_word-ms_ppt-int_expl      1           3      0.5445
## 6                ms_word-ms_outlk-int_expl      1           3      0.5445
##   Pct.of.Cases
## 1             1
## 2             1
## 3             2
## 4             1
## 5             1
## 6             1
##
## [[2]]
##
##                               Combn Freq
## 91 ms_word-ms_excel-ms_outlk-ms_pub-ms_proj-int_expl-adobe_rd-endnote-spss      1
## 92                ms_word-ms_excel-ms_ppt-ms_acc-int_expl-adobe_rd-endnote-spss      1
## 93                        ms_word-ms_outlk-ms_acc-int_expl-adobe_rd-endnote-spss      1
## 94                ms_word-ms_ppt-ms_outlk-ms_acc-int_expl-adobe_rd-endnote-spss      1
## 95                        ms_word-ms_pub-ms_acc-int_expl-adobe_rd-endnote-spss      1
## 96                ms_outlk-ms_proj-ms_acc-int_expl-adobe_rd-endnote-spss      1
##   Weighted.Freq Pct.of.Resp Pct.of.Cases
## 91             9      1.633           1
## 92             8      1.452           1
## 93             7      1.270           1
## 94             8      1.452           1
## 95             7      1.270           1
## 96             7      1.270           1
```

References

apply shortcut for creating the Combn column in the output by [Justin](#)
 See: <http://stackoverflow.com/q/11348391/1270695> and <http://stackoverflow.com/q/11622660/1270695>

Chapter 4

RandomNames

The `RandomNames()` function uses data from the *Genealogy Data: Frequently Occurring Surnames from Census 1990–Names Files* web page¹ to generate a `data.frame` with random names.

Arguments

- **N**: The number of random names you want. Defaults to 100.
- **cat**: Do you want "common" names, "rare" names, names with an "average" frequency, or some combination of these? Should be specified as a character vector (for example, `c("rare", "common")`). Defaults to `NULL`, in which case all names are used as the sample frame.
- **gender**: Do you want first names from the "male" dataset, the "female" dataset, or from all available names? Should be specified as a quoted string (for example, "male"). Defaults to `NULL`, in which case all available first names are used as the sample frame.
- **MFprob**: What proportion of the sample should be male names and what proportion should be female? Specify as a numeric vector that sums to 1 (for example, `c(.6, .4)`). The first number represents the probability of sampling a "male" first name, and the second number represents the probability of sampling a "female" name. This argument is not used if only one **gender** has been specified in the previous argument. Defaults to `NULL`, in which case, the probability used is `c(.5, .5)`.
- **dataset**: What do you want to use as the dataset of names from which to sample? A default dataset is provided that can generate over 400 million unique names. See the “*Dataset Details*” section for more information.

Dataset Details

This function samples from a provided dataset of names. By default, it uses the data from the *Genealogy Data: Frequently Occurring Surnames from Census 1990–Names Files* web page. Those data have been converted to `list` named “`CensusNames1990`” containing three `data.frames` (named “`surnames`”, “`malenames`”, and “`femalenames`”) and saved as an `.RData` file named `CensusNames.RData`. The data file (approximately 615 kb) can be manually downloaded from [Github](https://github.com/mrdwab/2657-R-Functions/blob/master/data/CensusNames.RData)² and loaded to your workspace. The function will perform some basic checking to see if either the `CensusNames.RData` file or the `CensusNames1990` objects are available in your workspace or working directory. If neither is found and an internet connection is active during your R session, the function will offer you the option to automatically download the dataset and add it to your *current* session.

Alternatively, you may provide your own data in a `list` formatted according to the following specifications (see the “*myCustomNames*” data in the “*Examples*” section). *Please remember that R is case sensitive!*

¹See http://www.census.gov/genealogy/www/data/1990surnames/names_files.html

²See: <https://github.com/mrdwab/2657-R-Functions/blob/master/data/CensusNames.RData>

- This must be a named list with three items: "surnames", "malenames", and "femalenames".
- The contents of each list item is a `data.frame` with at least the following named columns: "Name" and "Category".
- Acceptable values for "Category" are "common", "rare", and "average".

Examples

```
# Load the function!
require(RCurl)
baseURL = c("https://raw.githubusercontent.com/mrdwab/2657-R-Functions/master/")
source(textConnection(getURL(paste0(baseURL, "scripts/random.names.R"))))
```

```
# Generate 20 random names
RandomNames(N = 20)
```

```
##      Gender FirstName  Surnames
## 1      F    Dalila    Cordrey
## 2      M    Raymon    Selic
## 3      M    Wilber    Rife
## 4      M  Federico    Helena
## 5      M      Rey Vanderroest
## 6      M   Maynard   Madhavan
## 7      M   Agustin    Queja
## 8      M   Gregory   Woollard
## 9      F   Kazuko    Feasel
## 10     M    Gavin    Musolf
## 11     M    Huey    Dominique
## 12     M   Tristan   Anzualda
## 13     M    Neil    Gasbarro
## 14     F  Lashawn    Deland
## 15     M   Jamison    Brucki
## 16     F   Sharyl    Martinz
## 17     F   Eugenie    Sifers
## 18     M    Galen    Fabozzi
## 19     F   Suzette   Camareno
## 20     M   Harlan   Suellentrop
```

```
# Generate a reproducible list of 100 random names with approximately 80% of
# the names being female names, and 20% being male names.
set.seed(1)
temp <- RandomNames(cat = "common", MFprob = c(.2, .8))
list(head(temp), tail(temp))
```

```
## [[1]]
##      Gender FirstName  Surnames
## 1      F   Mildred    Moring
## 2      F  Gertrude    Duron
## 3      F    Marta    Croom
## 4      F  Angelita   Neuberger
## 5      M   Morris    Gallucci
## 6      F    Enid Barrientos
##
## [[2]]
##      Gender FirstName  Surnames
```



```
## 95      F      Jeanie Toussaint
## 96      F Rosalinda Beauvais
## 97      F      Blanche Schaeffer
## 98      F      Lena      Hepp
## 99      F      Louisa      Struck
## 100     F      Dorothy      Divito
```

```
table(temp$Gender)
```

```
##
##  F  M
## 84 16
```

```
# Cleanup
```

```
rm(.Random.seed, envir=globalenv()) # Resets your seed
rm(temp)
```

```
# Generate 10 names from the common and rare categories of names
```

```
RandomNames(N = 10, cat = c("common", "rare"))
```

```
##      Gender FirstName      Surnames
## 1      F      Flora      Todt
## 2      F      Willie      Dehl
## 3      F      Ingrid      Fetter
## 4      F      Emilie      Gnagey
## 5      F      Elli      Fahner
## 6      F      Gregory      Linsley
## 7      F      Marisa      Dewees
## 8      F      Jeanice Bloomstrand
## 9      F      Kyoko      Watral
## 10     M      Rafael      Farria
```

```
# Error messages
```

```
RandomNames(cat = c("common", "rare", "avg"))
```

```
## Error: cat must be either "all", NULL, or a combination of "common", "average",
## or "rare"
```

```
# Generate 10 female names
```

```
RandomNames(N = 10, gender = "female")
```

```
##      Gender FirstName Surnames
## 1      F      Julie  Lenberg
## 2      F      Trinidad Killings
## 3      F      Terri   Alier
## 4      F      Donnetta Golanski
## 5      F      Cindie  Helder
## 6      F      Shayna  Stepien
## 7      F      Geri    Gostlin
## 8      F      James   Missey
## 9      F      Rosenda Scroggin
## 10     F      Rosella Lantrip
```

Using Your Own Data

As mentioned, it is possible to use your own list of names as the basis for generating the random names (though this is perhaps unnecessary, given the number of random names possible with the provided dataset). The following is an example of how your dataset must be structured. Note that the dataset name in the `dataset` argument is *not* quoted.

```
myCustomNames <- list(
  surnames = data.frame(
    Name = LETTERS[1:26],
    Category = c(rep("rare", 10), rep("average", 10), rep("common", 6))),
  malenames = data.frame(
    Name = letters[1:10],
    Category = c(rep("rare", 4), rep("average", 4), rep("common", 2))),
  femalenames = data.frame(
    Name = letters[11:26],
    Category = c(rep("rare", 8), rep("average", 4), rep("common", 4))))
str(myCustomNames)

## List of 3
## $ surnames      : 'data.frame': 26 obs. of  2 variables:
##   ..$ Name       : Factor w/ 26 levels "A","B","C","D",...: 1 2 3 4 5 6 7 8 9 10 ...
##   ..$ Category: Factor w/ 3 levels "average","common",...: 3 3 3 3 3 3 3 3 3 3 ...
## $ malenames     : 'data.frame': 10 obs. of  2 variables:
##   ..$ Name       : Factor w/ 10 levels "a","b","c","d",...: 1 2 3 4 5 6 7 8 9 10
##   ..$ Category: Factor w/ 3 levels "average","common",...: 3 3 3 3 1 1 1 1 2 2
## $ femalenames   : 'data.frame': 16 obs. of  2 variables:
##   ..$ Name       : Factor w/ 16 levels "k","l","m","n",...: 1 2 3 4 5 6 7 8 9 10 ...
##   ..$ Category: Factor w/ 3 levels "average","common",...: 3 3 3 3 3 3 3 3 1 1 ...

RandomNames(N = 15, dataset = myCustomNames)

##      Gender FirstName Surnames
## 1      M      f      J
## 2      M      d      U
## 3      F      s      K
## 4      F      w      L
## 5      M      h      Y
## 6      F      x      C
## 7      M      i      B
## 8      M      b      L
## 9      M      c      J
## 10     M      a      J
## 11     M      c      E
## 12     F      m      J
## 13     M      h      N
## 14     F      r      H
## 15     M      c      M
```

References

- Inspired by the online Random Name Generator (<http://random-name-generator.info/>).
- Uses data from the 1990 US Census (http://www.census.gov/genealogy/www/data/1990surnames/names_files.html)

Chapter 5

row.extractor

The `row.extractor` function takes a `data.frame` and extracts rows with the `min`, `median`, or `max` values of a given variable, or extracts rows with specific quantiles of a given variable.

Arguments

- `data`: the source `data.frame`.
- `extract.by`: the column which will be used as the reference for extraction; can be specified either by the column number or the variable name.
- `what`: options are `min` (for all rows matching the minimum value), `median` (for the median row or rows), `max` (for all rows matching the maximum value), or `all` (for `min`, `median`, and `max`); alternatively, a numeric vector can be specified with the desired quantiles, for instance `c(0, .25, .5, .75, 1)`

Examples

```
# Load the function!
require(RCurl)
baseURL = c("https://raw.githubusercontent.com/mrdwab/2657-R-Functions/master/")
source(textConnection(getURL(paste0(baseURL, "scripts/row.extractor.R"))))

# Make up some data
set.seed(1)
dat = data.frame(V1 = 1:50, V2 = rnorm(50),
                 V3 = round(abs(rnorm(50)), digits=2),
                 V4 = sample(1:30, 50, replace=TRUE))

# Get a summary of the data
summary(dat)
```

##	V1	V2	V3	V4
## Min.	: 1.0	Min. :-2.215	Min. :0.000	Min. : 2.00
## 1st Qu.:	:13.2	1st Qu.: -0.372	1st Qu.:0.347	1st Qu.: 8.25
## Median :	:25.5	Median : 0.129	Median :0.590	Median :13.00
## Mean :	:25.5	Mean : 0.100	Mean :0.774	Mean :14.80
## 3rd Qu.:	:37.8	3rd Qu.: 0.728	3rd Qu.:1.175	3rd Qu.:20.75
## Max.	:50.0	Max. : 1.595	Max. :2.400	Max. :29.00

```
# Get the rows corresponding to the ,min,, ,median,, and ,max, of ,V4,
row.extractor(dat, 4)
```

```
##      V1      V2   V3 V4
## 28 28 -1.4708 0.00  2
## 47 47  0.3646 1.28 13
## 29 29 -0.4782 0.07 13
## 11 11  1.5118 2.40 29
## 14 14 -2.2147 0.03 29
## 18 18  0.9438 1.47 29
## 19 19  0.8212 0.15 29
## 50 50  0.8811 0.47 29

# Get the ,min, rows only, referenced by the variable name
row.extractor(dat, "V4", "min")

##      V1      V2 V3 V4
## 28 28 -1.471  0  2

# Get the ,median, rows only. Notice that there are two rows
#   since we have an even number of cases and true median
#   is the mean of the two central sorted values
row.extractor(dat, "V4", "median")

##      V1      V2   V3 V4
## 47 47  0.3646 1.28 13
## 29 29 -0.4782 0.07 13

# Get the rows corresponding to the deciles of ,V3,
row.extractor(dat, "V3", seq(0.1, 1, 0.1))

##      V1      V2   V3 V4
## 10 10 -0.30539 0.14 22
## 26 26 -0.05613 0.29 16
## 39 39  1.10003 0.37 13
## 41 41 -0.16452 0.54 10
## 30 30  0.41794 0.59 26
## 44 44  0.55666 0.70  5
## 37 37 -0.39429 1.06 21
## 49 49 -0.11235 1.22 14
## 34 34 -0.05381 1.52 19
## 11 11  1.51178 2.40 29
```

To Do

- Add some error checking to make sure a valid `what` is provided.

References

which.quantile function by [cbeleites](#)
 See: <http://stackoverflow.com/q/10256503/1270695>

Chapter 6

sample.size

The `sample.size` function either calculates the optimum survey sample size when provided with a population size, or the confidence interval of using a certain sample size with a given population. It can be used to generate tables (`data.frames`) of different combinations of inputs of the following arguments, which can be useful for showing the effect of each of these in sample size calculation.

Arguments

- `population`: The population size for which a sample size needs to be calculated.
- `samp.size`: The sample size.
 - This argument is only used when calculating the confidence interval, and defaults to `NULL`.
- `c.lev`: The desired confidence level. Defaults to a reasonable 95%.
- `c.int`: The confidence interval.
 - This argument is only used when calculating the sample size.
 - If not specified when calculating the sample size, defaults to 5% and a message is provided indicating this; this is also the default action if `c.int = NULL`.
- `what`: Should the function calculate the desired sample size or the confidence interval?
 - Accepted values are "sample" and "confidence" (quoted), and defaults to "sample".
- `distribution`: Response distribution. Defaults to 50%, which will give you the largest sample size.

Examples

```
# Load the function!
require(RCurl)
baseURL = c("https://raw.githubusercontent.com/mrdwab/2657-R-Functions/master/")
source(textConnection(getURL(paste0(baseURL, "scripts/sample.size.R"))))
# What should our sample size be for a population of 300?
# All defaults accepted.
sample.size(population = 300)
```

```
##   population conf.level conf.int distribution sample.size
## 1          300         95         5          50         169
```

```

# What sample should we take for a population of 300
#   at a confidence level of 97%?
sample.size(population = 300, c.lev = 97)

##   population conf.level conf.int distribution sample.size
## 1         300         97         5          50         183

# What about if we change our confidence interval?
sample.size(population = 300, c.int = 2.5, what = "sample")

##   population conf.level conf.int distribution sample.size
## 1         300         95         2.5          50         251

# What about if we want to determine the confidence interval
#   of a sample of 140 from a population of 300? A confidence
#   level of 95% is assumed.
sample.size(population = 300, samp.size = 140, what = "confidence")

##   population conf.level conf.int distribution sample.size
## 1         300         95         6.06          50         140

```

Advanced Usage

As the function is vectorized, it is possible to easily make tables with multiple scenarios.

```

# What should the sample be for populations of 300 to 500 by 50?
sample.size(population=c(300, 350, 400, 450, 500))

##   population conf.level conf.int distribution sample.size
## 1         300         95         5          50         169
## 2         350         95         5          50         183
## 3         400         95         5          50         196
## 4         450         95         5          50         207
## 5         500         95         5          50         217

# How does varying confidence levels or confidence intervals
#   affect the sample size?
sample.size(population=300,
            c.lev=rep(c(95, 96, 97, 98, 99), times = 3),
            c.int=rep(c(2.5, 5, 10), each=5))

##   population conf.level conf.int distribution sample.size
## 1         300         95         2.5          50         251
## 2         300         96         2.5          50         255
## 3         300         97         2.5          50         259
## 4         300         98         2.5          50         264
## 5         300         99         2.5          50         270
## 6         300         95         5.0          50         169
## 7         300         96         5.0          50         176
## 8         300         97         5.0          50         183
## 9         300         98         5.0          50         193
## 10        300         99         5.0          50         207
## 11        300         95        10.0          50          73
## 12        300         96        10.0          50          78
## 13        300         97        10.0          50          85
## 14        300         98        10.0          50          93
## 15        300         99        10.0          50         107

```

```
# What is are the confidence intervals for a sample of
# 150, 160, and 170 from a population of 300?
sample.size(population=300,
            samp.size = c(150, 160, 170),
            what="confidence")
```

```
## population conf.level conf.int distribution sample.size
## 1      300          95    5.67             50         150
## 2      300          95    5.30             50         160
## 3      300          95    4.96             50         170
```

Note that the use of `rep()` is required in constructing the arguments for the advanced usage examples where more than one argument takes on multiple values.

References

See the *2657 Productions News* site for how this function progressively developed¹. The `sample.size` function is based on the following formulas²:

$$ss = \frac{-Z^2 \times p \times (1-p)}{c^2}$$

$$pss = \frac{\frac{ss}{ss-1}}{1 + \frac{ss-1}{pop}}$$

¹<http://news.mrdwab.com/2010/09/10/a-sample-size-calculator-function-for-r/>

²See: Creative Research Systems. (n.d.). *Sample size formulas for our sample size calculator*. Retrieved from: <http://www.surveysystem.com/sample-size-formula.htm>. Archived on 07 August 2012 at <http://www.webcitation.org/69kNjMuKe>.

Chapter 7

stratified

The `stratified` function samples from a `data.frame` in which one of the columns can be used as a “stratification” or “grouping” variable. The result is a new `data.frame` with the specified number of samples from each group.

Arguments

- `df`: The source `data.frame`.
- `group`: Your grouping variable.
- `size`: The desired sample size.
 - If `size` is a value between 0 and 1 expressed as a decimal, size is set to be proportional to the number of observations per group.
 - If `size` is a single positive integer, it will be assumed that you want the same number of samples from each group.
 - If `size` is a vector, the function will check to see whether the length of the vector matches the number of groups and use those specified values as the desired sample sizes. The values in the vector should be in the same order as you would get if you tabulated the grouping variable (usually alphabetic order); alternatively, you can name each value to ensure it is properly matched.

Note: Because of how computers deal with floating-point arithmetic, and because R uses a “round to even” approach, the `size` per strata that results when specifying a proportionate sample may be slightly higher or lower per strata than you might have expected.

- `seed`: The seed that you want to use (using `set.seed()`), if any. Defaults to `NULL`.

Note: This is different from using `set.seed()` before using the function. Setting a seed using this argument is equivalent to using `set.seed(seed)` each time that you go to take a sample from a different group (in other words, the same seed is used for each group). See “*Additional Information*” for further information.

- `...` : Further arguments to be passed to the `sample()` function.

Examples

First, let’s make up some data. In the dataset below, we can treat variables “A” and “D” as potential grouping variables.

```
# Generate a couple of sample data.frames to play with
set.seed(1)
dat1 <- data.frame(ID = 1:100,
  A = sample(c("AA", "BB", "CC", "DD", "EE"), 100, replace=T),
  B = rnorm(100), C = abs(round(rnorm(100), digits=1)),
  D = sample(c("CA", "NY", "TX"), 100, replace=T))

dat2 <- data.frame(ID = 1:20,
  A = c(rep("AA", 5), rep("BB", 10),
    rep("CC", 3), rep("DD", 2)))
```

```
# What do the data look like in general?
```

```
summary(dat1)
```

```
##           ID           A           B           C           D
## Min.      : 1.0    AA:13    Min.      :-1.9144    Min.      :0.000    CA:23
## 1st Qu.: 25.8    BB:25    1st Qu.: -0.6141    1st Qu.: 0.300    NY:42
## Median : 50.5    CC:19    Median : -0.1176    Median : 0.650    TX:35
## Mean     : 50.5    DD:26    Mean     : -0.0176    Mean     : 0.825
## 3rd Qu.: 75.2    EE:17    3rd Qu.:  0.5382    3rd Qu.: 1.200
## Max.     :100.0                Max.      : 2.4016    Max.      :2.900
```

```
summary(dat2)
```

```
##           ID           A
## Min.      : 1.00    AA: 5
## 1st Qu.:  5.75    BB:10
## Median :10.50    CC: 3
## Mean     :10.50    DD: 2
## 3rd Qu.:15.25
## Max.     :20.00
```

Now, let's try different settings applying the stratified function.

```
# Load the function!
```

```
require(RCurl)
```

```
## Loading required package: RCurl
```

```
## Loading required package: bitops
```

```
baseURL = c("https://raw.githubusercontent.com/mrdwab/2657-R-Functions/master/")
source(textConnection(getURL(paste0(baseURL, "scripts/stratified.R"))))
```

```
# Let's take a 10% sample from all -A- groups in dat1, seed = 1
```

```
stratified(dat1, "A", .1, seed = 1)
```

```
##      ID  A      B      C  D
## 27 27 AA -0.44329 0.8 TX
## 22 22 BB -0.70995 0.1 TX
## 26 26 BB  0.29145 0.0 TX
## 40 40 CC  0.26710 0.9 NY
## 44 44 CC  0.70021 0.8 CA
## 23 23 DD  0.61073 0.5 NY
## 39 39 DD  0.37002 0.4 CA
## 49 49 DD -1.22461 0.4 NY
## 21 21 EE  0.47551 2.3 TX
## 29 29 EE  0.07434 1.0 TX
```

```
# Let,s take 5 samples from all -D- groups in dat1,
# seed = 1, specified by column number
stratified(dat1, group = 5, size = 5, seed = 1)
```

```
##      ID  A      B      C  D
## 36 36 DD  0.33295 0.2 CA
## 44 44 CC  0.70021 0.8 CA
## 57 57 BB  0.71671 0.7 CA
## 73 73 BB -0.21458 0.6 CA
## 32 32 CC -0.13518 1.0 CA
## 23 23 DD  0.61073 0.5 NY
## 37 37 DD  1.06310 1.5 NY
## 52 52 EE  0.04212 1.7 NY
## 91 91 BB -1.91436 0.7 NY
## 17 17 DD -1.80496 0.3 NY
## 26 26 BB  0.29145 0.0 TX
## 29 29 EE  0.07434 1.0 TX
## 54 54 BB  0.15803 0.3 TX
## 80 80 EE -0.32427 0.3 TX
## 15 15 DD -0.74327 0.6 TX
```

```
# Let,s try to take a sample from all -A- groups in dat1, seed = 1,
# where we specify the number wanted from each group,
# but we did not specify the correct number of groups
stratified(dat1, "A", size = c(3, 5, 7), seed = 1)
```

```
## Error: Number of groups is 5 but number of sizes supplied is 3
```

```
# Try again
```

```
stratified(dat1, "A", size = c(3, 5, 4, 5, 2), seed = 1)
```

```
## 'size' vector entered as:
```

```
##
```

```
## size = structure(c(3, 5, 4, 5, 2), .Names = c('AA', 'BB', 'CC', 'DD', 'EE'))
```

```
##      ID  A      B      C  D
## 27 27 AA -0.44329 0.8 TX
## 34 34 AA -1.52357 1.5 CA
## 47 47 AA -1.27659 1.4 TX
## 22 22 BB -0.70995 0.1 TX
## 26 26 BB  0.29145 0.0 TX
## 62 62 BB -0.46164 0.4 CA
## 78 78 BB -0.03763 1.6 NY
## 14 14 BB  0.02800 0.9 TX
## 40 40 CC  0.26710 0.9 NY
## 44 44 CC  0.70021 0.8 CA
## 51 51 CC -0.62037 0.4 TX
## 67 67 CC -0.31999 0.3 CA
## 23 23 DD  0.61073 0.5 NY
## 39 39 DD  0.37002 0.4 CA
## 49 49 DD -1.22461 0.4 NY
## 85 85 DD  0.30656 0.1 NY
## 17 17 DD -1.80496 0.3 NY
## 21 21 EE  0.47551 2.3 TX
## 29 29 EE  0.07434 1.0 TX
```

```

# Try a 10% sample from all -A- groups in dat2, seed = 1
stratified(dat2, "A", size = .1, seed = 1)

##      ID  A
## 8    8 BB

# How does that compare to -table(dat2$A) * .1)-?
table(dat2$A) * .1

##
## AA  BB  CC  DD
## 0.5 1.0 0.3 0.2

# Instead of -round()- you can use -floor()- or -ceiling()- to
#   round down or up to an integer
stratified(dat2, "A", size = ceiling(table(dat2$A) * .1), seed = 1)

##      ID  A
## 2     2 AA
## 8     8 BB
## 16    16 CC
## 19    19 DD

```

Additional Information

seed

The inclusion of a **seed** argument is mostly a matter of convenience, to be able to have a single seed with which the samples can be verified later. However, by using the **seed** argument, *the same seed is used to sample from each group*. This may be a problem if there are many groups that have the same number of observations, since it means that the same observation number will be selected from each of those groups. For instance, if group “AA” and “DD” both had the same number of observations (say, 5) and you were using a **seed** of 1, the second, fifth, and fourth observation would be taken from each of those groups. To avoid this, you can set the seed using **set.seed()** before you run the function.

The following examples should demonstrate the difference between the two approaches.

```

# Let,s manually split the dataset and sample 2 from each group, seed = 1
(seedy.demonstration <- split(dat2$ID, dat2$A))

## $AA
## [1] 1 2 3 4 5
##
## $BB
## [1] 6 7 8 9 10 11 12 13 14 15
##
## $CC
## [1] 16 17 18
##
## $DD
## [1] 19 20

```

```

set.seed(1); sample(seedy.demonstration$AA, 2)

## [1] 2 5

set.seed(1); sample(seedy.demonstration$BB, 2)

## [1] 8 9

set.seed(1); sample(seedy.demonstration$CC, 2)

## [1] 16 18

set.seed(1); sample(seedy.demonstration$DD, 2)

## [1] 19 20

# Now do the same with the stratified function.
# Note that the IDs are the same as we got manually.
stratified(dat2, "A", 2, 1)

##      ID  A
## 2      2 AA
## 5      5 AA
## 8      8 BB
## 9      9 BB
## 16     16 CC
## 18     18 CC
## 19     19 DD
## 20     20 DD

# Now, use -set.seed()- before running the function.
set.seed(1); stratified(dat2, "A", 2)

##      ID  A
## 2      2 AA
## 5      5 AA
## 11     11 BB
## 14     14 BB
## 16     16 CC
## 17     17 CC
## 20     20 DD
## 19     19 DD

# And the same manually...
set.seed(1)
sample(seedy.demonstration$AA, 2)

## [1] 2 5

sample(seedy.demonstration$BB, 2)

```

```
## [1] 11 14

sample(seedy.demonstration$CC, 2)

## [1] 16 17

sample(seedy.demonstration$DD, 2)

## [1] 20 19

# OK. So far so good. But what about if we do something else involving
# random number generation during our interactive session?
set.seed(1)
sample(seedy.demonstration$AA, 2) # This matches....

## [1] 2 5

rnorm(1) # This involves random number generation....

## [1] 0.1836

sample(seedy.demonstration$BB, 2) # Things go out of order now....

## [1] 8 14

# Or, let's try the same, but sampling in a different order.
set.seed(1)
sample(seedy.demonstration$CC, 2) # Already, no match....

## [1] 16 18
```

As a user, you need to weigh the benefits and drawbacks of setting the seed *before* running the function as opposed to setting the seed *with* the function. Setting the seed *before* would be useful if there are several groups with the same number of observations; however, in the slim chance that you need to verify the samples manually, you *may* run into problems.

strata.sampling

A related function, `strata.sampling()`, is a basic wrapper around the `strata()` function from the “sampling” package that allows the user to draw either a fixed size sample or proportionate sample. The `strata.sampling()` function has the advantage that it also includes a column indicating the probability for each observation being selected.

The `strata.sampling()` function is included as one of the “snippets” found under “Snippets and Tips.”

To Do

Can the function be modified to allow grouping by more than one column?

References

The evolution of this function can be found at the following URLs:

1. <http://news.mrdwab.com/2011/05/15/stratified-random-sampling-in-r-beta/>
2. <http://news.mrdwab.com/2011/05/20/stratified-random-sampling-in-r-from-a-data-frame/>
3. <http://stackoverflow.com/a/9714207/1270695>

The version here is entirely reworked and does not require an additional package to be loaded.

Chapter 8

stringseed.sampling

The `stringseed.sampling` function is designed as a batch sampling function that allows the user to specify any alphanumeric input as the seed *per sample in the batch*.

Arguments

- `seedbase`: A vector of seeds to be used for sampling.
- `N`: The “population” from which to draw the sample.
- `n`: The desired number of samples.
- `write.output`: Logical. Should the output be written to a file? Defaults to `FALSE`. If `TRUE`, a csv file is written with the sample “metadata”, and a plain text file is written with the details of the resulting sample. The names of the files written are “Sample frame generated on {date the script was run} .csv” and “Samples generated on {date the script was run} .txt” and will be found in your current working directory.

Examples

```
# Load the function!
require(RCurl)
baseURL = c("https://raw.githubusercontent.com/mrdwab/2657-R-Functions/master/")
source(textConnection(getURL(paste0(baseURL, "scripts/stringseed.sampling.R"))))
# We'll use a data.frame with a list of village names, the population,
# and the desired samples as our columns. The function will use the
# village names to generate a unique seed for each village before
# drawing the sample.
myListOfPlaces <- data.frame(
  villageName = c("Melakkal", "Sholavandan", "T. Malaipatti"),
  population = c(120, 130, 140),
  requiredSample = c(30, 25, 12))
myListOfPlaces

##      villageName population requiredSample
## 1      Melakkal        120              30
## 2    Sholavandan        130              25
## 3 T. Malaipatti        140              12

stringseed.sampling(seedbase = myListOfPlaces$villageName,
                     N = myListOfPlaces$population,
                     n = myListOfPlaces$requiredSample)
```

```
## $input
##      seedbase populations samplesizes      seeds
## 1      Melakkal          120           30 1331891848
## 2      Sholavandan        130           25 438637044
## 3 T. Malaipatti          140           12 1614276325
##
## $samples
## $samples$Melakkal
## [1] 108 13 54 96 56 111 110 27 112 84 60 62 22 12 23 117 93 67 79
## [20] 74 65 90 71 113 53 85 40 19 31 18
##
## $samples$Sholavandan
## [1] 94 14 27 96 102 11 47 18 118 91 120 57 40 89 5 105 116 70 109
## [20] 35 16 90 4 98 30
##
## $samples$`T. Malaipatti`
## [1] 130 102 20 123 85 104 5 105 7 115 96 120

# Manual verification of the samples generated for Melakkal village
# (for which the automatically generated seed was 1331891848)
set.seed(1331891848)
sample(120, 30)

## [1] 108 13 54 96 56 111 110 27 112 84 60 62 22 12 23 117 93 67 79
## [20] 74 65 90 71 113 53 85 40 19 31 18
```

```
# What about using the function on a single input?
stringseed.sampling("Santa Barbara", 1920, 100)
```

```
## $input
##      seedbase populations samplesizes      seeds
## 1 Santa Barbara          1920          100 323728098
##
## $samples
## [1] 129 1869 1170 192 344 18 694 1628 601 874 188 631 1910 605 367
## [16] 1411 755 1741 489 658 821 1160 1783 150 1556 423 753 416 1510 707
## [31] 1353 1744 520 1720 1608 990 1235 402 1669 1800 502 1516 1531 1860 1369
## [46] 1431 1570 1290 1731 1679 1070 931 68 1466 1836 316 815 24 1877 1689
## [61] 1141 981 279 1605 842 1773 1186 1081 17 661 1104 1668 1180 54 1233
## [76] 1879 1666 449 838 1167 1157 773 1707 916 1243 492 525 1308 1460 232
## [91] 1695 1644 1312 1051 1325 545 397 1551 477 1205
```

References

Ben Bolker¹ recommended the use of the “digest” package to convert a string to a numeric value.
See: <http://stackoverflow.com/q/10910698/1270695>.

¹Website: <http://www.math.mcmaster.ca/~bolker>; Stack Overflow profile: <http://stackoverflow.com/users/190277/ben-bolker>.

Part II

The Functions

Chapter 9

Where to Get the Functions

The most current source code for the functions described in this document follow. It is recommended that you *do not* copy-and-paste the functions from this document since there may be errors resulting from poorly parsed quotation marks and so on; instead, load the functions directly from the 2657 R Functions page at github.

To load the functions, you can directly source them from the 2657 R Functions page at github: <https://github.com/mrdwab/2657-R-Functions>

You should be able to load the functions using the following (replace ----- with the function name¹):

```
require(RCurl)
baseURL = c("https://raw.githubusercontent.com/mrdwab/2657-R-Functions/master/")
source(textConnection(getURL(paste0(baseURL, "scripts/-----.R"))))
```

¹The “snippets” in Part III of this document can all be loaded from a single script, `snippets.R`.

Chapter 10

concat.split

```
concat.split = function(data, split.col, sep = ",", structure = "compact",
                        mode = NULL, drop.col = FALSE, fixed = FALSE) {
  # Takes a column with multiple values, splits the values into
  # separate columns, and returns a new data.frame.
  # --data-- is the source data.frame; --split.col-- is the variable that
  # needs to be split; --structure-- the type of output that should be
  # returned, either a -compact- or -expanded- form, or a -list-
  # (defaults to -compact-). --mode-- can be either -binary- or -value-
  # (where -binary- is default and it recodes values to 1 or NA); --sep--
  # is the character separating each value (defaults to -,). --drop.col--
  # is logical (whether to remove the original variable from the output).
  #
  # === EXAMPLES ===
  #
  #   dat = data.frame(
  #     V1 = c("1, 2, 4", "3, 4, 5", "1, 2, 5", "4", "1, 2, 3, 5"),
  #     V2 = c("1;2;3;4", "1", "2;5", "3;2", "2;3;4")
  #   )
  #   dat2 = data.frame(
  #     V1 = c("Fred, John, Sue", "Jerry, Jill",
  #            "Sally, Ryan", "Susan, Amos, Ben")
  #   )
  #
  #   concat.split(dat, 1)
  #   concat.split(dat, 1, structure="expanded")
  #   concat.split(dat, 1, structure="expanded", mode = "value")
  #   concat.split(dat, 2, sep=";")
  #   concat.split(dat, "V2", sep=";", mode="value")
  #   concat.split(dat2, 1)
  #   concat.split(dat2, "V1", drop.col=TRUE)
  #   concat.split(dat2, "V1", structure="expanded", drop.col=TRUE)
  #
  # See: http://stackoverflow.com/q/10100887/1270695
  # See also: http://stackoverflow.com/a/13912721/1270695
  #
  # Check to see if split.col is specified by name or position
  if (is.numeric(split.col)) split.col = split.col
  else split.col = which(colnames(data) %in% split.col)
  #
  # Split the data
  a = as.character(data[, split.col])
  b = strsplit(a, sep, fixed = fixed)
  #
  temp <- switch(
```

```

structure,
compact = {
  t1 <- read.table(text = a, sep = sep, fill = TRUE,
                  row.names = NULL, header = FALSE,
                  blank.lines.skip = FALSE)
  names(t1) <- paste(names(data[split.col]),
                    seq(ncol(t1)), sep="_")
  if (!is.null(mode))
    warning("
      ,mode, supplied but ignored.
      ,mode, setting only applicable
      when structure=,expanded,.")
  if (isTRUE(drop.col)) cbind(data[-split.col], t1)
  else cbind(data, t1)
},
list = {
  varname = paste(names(data[split.col]), "list", sep="_")
  if (suppressWarnings(is.na(try(max(as.numeric(unlist(b))))))) {
    data[varname] = list(
      lapply(lapply(b, as.character),
              function(x) gsub("^\\s+|\\s+$", "", x)))
  } else if (!is.na(try(max(as.numeric(unlist(b)))))) {
    data[varname] = list(lapply(b, as.numeric))
  }
  if (!is.null(mode))
    warning("
      ,mode, supplied but ignored.
      ,mode, setting only applicable
      when structure=,expanded,.")
  if (isTRUE(drop.col)) data[-split.col]
  else data
},
expanded = {
  if (suppressWarnings(is.na(try(max(as.numeric(unlist(b))))))) {
    what = "string"
    ncol = max(unlist(lapply(b, function(i) length(i))))
  } else if (!is.na(try(max(as.numeric(unlist(b)))))) {
    what = "numeric"
    ncol = max(as.numeric(unlist(b)))
  }
  temp1 <- switch(
    what,
    string = {
      temp = as.data.frame(t(sapply(b, ,[, 1:ncol])))
      names(temp) = paste(names(data[split.col]),
                          1:ncol, sep="_")
      temp = apply(
        temp, 2, function(x) gsub("^\\s+|\\s+$", "", x))
      temp1 = cbind(data, temp)
    },
    numeric = {
      temp = lapply(b, as.numeric)
      m = matrix(nrow = nrow(data), ncol = ncol)
      for (i in 1:nrow(data)) {
        m[i, temp[[i]]] = temp[[i]]
      }
      m = setNames(data.frame(m),

```



```

        paste(names(data[split.col]), 1:ncol, sep="_"))

    if (is.null(mode)) mode = "binary"
    temp1 <- switch(
      mode,
      binary = {cbind(data, replace(m, m != "NA", 1))},
      value = {cbind(data, m)},
      stop(",mode, must be ,binary, or ,value,")
    )
    if (isTRUE(drop.col)) temp1[-split.col]
    else temp1
  },
  stop(",structure, must be either ,compact,, ,expanded,, or ,list,")
temp
}

```


Chapter 11

df.sorter

```
df.sorter <- function(data, var.order=names(data),
                      col.sort=NULL, at.start=TRUE ) {
  # Sorts a data.frame by columns or rows or both. Can also subset the
  # data columns by using --var.order--. Can refer to variables either
  # by names or number. If referring to variable by number, and sorting
  # both the order of variables and the sorting within variables,
  # refer to the variable numbers of the final data.frame.
  #
  # === EXAMPLES ===
  #
  # library(foreign)
  # temp = "http://www.ats.ucla.edu/stat/stata/modules/kidshtwt.dta"
  # kidshtwt = read.dta(temp); rm(temp)
  # df.sorter(kidshtwt, var.order = c("fam", "bir", "wt", "ht"))
  # df.sorter(kidshtwt, var.order = c("fam", "bir", "wt", "ht"),
  #           col.sort = c("birth", "famid")) # USE FULL NAMES HERE
  # df.sorter(kidshtwt, var.order = c(1:4), # DROP THE WT COLUMNS
  #           col.sort = 3) # SORT BY HT1

  if (is.numeric(var.order))
    var.order = colnames(data)[var.order]
  else var.order = var.order

  if (isTRUE(at.start)) {
    x = unlist(lapply(var.order, function(x)
      sort(grep(paste("^", x, sep="", collapse=""),
        names(data), value = TRUE))))
  } else if (!isTRUE(at.start)) {
    x = unlist(lapply(var.order, function(x)
      sort(grep(x, names(data), value = TRUE))))
  }

  y = data[, x]

  if (is.null(col.sort)) {
    y
  } else if (is.numeric(col.sort)) {
    y[do.call(order, y[colnames(y)[col.sort]]), ]
  } else if (!is.numeric(col.sort)) {
    y[do.call(order, y[col.sort]), ]
  }
}
```


Chapter 12

multi.freq.table

```
multi.freq.table <- function(data, sep = "", boolean = TRUE, factors = NULL,
                             NAto0 = TRUE, basic = FALSE, dropzero=TRUE,
                             clean=TRUE) {
  # Takes multiple-response data and tabulates it according
  # to the possible combinations of each variable.
  #
  # === EXAMPLES ===
  #
  #   set.seed(1)
  #   dat = data.frame(A = sample(c(0, 1), 20, replace=TRUE),
  #                     B = sample(c(0, 1), 20, replace=TRUE),
  #                     C = sample(c(0, 1), 20, replace=TRUE),
  #                     D = sample(c(0, 1), 20, replace=TRUE),
  #                     E = sample(c(0, 1), 20, replace=TRUE))
  #   multi.freq.table(dat)
  #   multi.freq.table(dat[1:3], sep="-", dropzero=TRUE)
  #
  # See: http://stackoverflow.com/q/11348391/1270695
  #      http://stackoverflow.com/q/11622660/1270695

  if (!is.data.frame(data)) {
    stop("Input must be a data frame.")
  }

  if (isTRUE(boolean)) {
    CASES = nrow(data)
    RESPS = sum(data, na.rm=TRUE)

    if(isTRUE(NAto0)) {
      data[is.na(data)] = 0
      VALID = CASES
      VRESP = RESPS
    } else if(!isTRUE(NAto0)) {
      data = data[complete.cases(data), ]
      VALID = CASES - (CASES - nrow(data))
      VRESP = sum(data)
    }

    if(isTRUE(basic)) {
      counts = data.frame(Freq = colSums(data),
                           Pct.of.Resp = (colSums(data)/sum(data))*100,
                           Pct.of.Cases = (colSums(data)/nrow(data))*100)
```

```

} else if (!isTRUE(basic)) {
  counts = data.frame(table(data))
  Z = counts[, c(intersect(names(data), names(counts)))]
  Z = rowSums(sapply(Z, as.numeric)-1)
  if(Z[1] == 0) { Z[1] = 1 }
  N = ncol(counts)
  counts$Combn = apply(counts[-N] == 1, 1,
                        function(x) paste(names(counts[-N])[x],
                                           collapse=sep))

  counts$Weighted.Freq = Z*counts$Freq
  counts$Pct.of.Resp = (counts$Weighted.Freq/sum(data))*100
  counts$Pct.of.Cases = (counts$Freq/nrow(data))*100
  if (isTRUE(dropzero)) {
    counts = counts[counts$Freq != 0, ]
  } else if (!isTRUE(dropzero)) {
    counts = counts
  }
  if (isTRUE(clean)) {
    counts = data.frame(Combn = counts$Combn, Freq = counts$Freq,
                       Weighted.Freq = counts$Weighted.Freq,
                       Pct.of.Resp = counts$Pct.of.Resp,
                       Pct.of.Cases = counts$Pct.of.Cases)
  }
}
message("Total cases:      ", CASES, "\n",
        "Valid cases:      ", VALID, "\n",
        "Total responses: ", RESPS, "\n",
        "Valid responses: ", VRESP, "\n")
counts
} else if (!isTRUE(boolean)) {
  CASES = nrow(data)
  RESPS = length(data[!is.na(data)])
  if (!isTRUE(any(sapply(data, is.factor)))) {
    if (is.null(factors)) {
      stop("Input variables must be factors.
Please provide factors using the ,factors, argument or
convert your data to factor before using function.")
    } else {
      data[sapply(data, is.character)] =
        lapply(data[sapply(data, is.character)],
               function(x) factor(x, levels=factors))
    }
  }
}
if (isTRUE(basic)) {
  ROWS = levels(unlist(data))
  OUT = table(unlist(data))
  PCT = (OUT/sum(OUT)) * 100
  OUT = data.frame(ROWS, OUT, PCT, row.names=NULL)
  OUT = data.frame(Item = OUT[, 1], Freq = OUT[, 3],
                   Pct.of.Resp = OUT[, 5],
                   Pct.of.Cases = (OUT[, 3]/CASES)*100)
  message("Total cases:      ", CASES, "\n",
          "Total responses: ", RESPS, "\n")
  OUT
} else if (!isTRUE(basic)) {
  Combos = apply(data, 1, function(x) paste0(sort(x), collapse = sep))
  Weight = as.numeric(rowSums(!is.na(data)))
  OUT = data.frame(table(Combos, Weight))

```

```

OUT = OUT[OUT$Freq > 0, ]
OUT$Weight = as.numeric(as.character(OUT$Weight))
if(OUT$Weight[1] == 0) { OUT$Weight[1] = 1 }
OUT$Weighted.Freq = OUT$Weight*OUT$Freq
OUT$Pct.of.Resp = (OUT$Weighted.Freq/RESPS)*100
OUT$Pct.of.Cases = (OUT[, 3]/CASES)*100
message("Total cases:      ", CASES, "\n",
        "Total responses: ", RESPS, "\n")
OUT[-2]
}
}
}

```


Chapter 13

RandomNames

```
RandomNames <- function(N = 100, cat = NULL, gender = NULL,
                        MFprob = NULL, dataset = NULL) {
  # Generates a "data.frame" of random names with the following columns:
  #   "Gender", "FirstName", and "Surname". All arguments have preset
  #   defaults, so the function can be run simply by typing RandomNames(),
  #   which will generate 100 random male and female names.
  #
  # === EXAMPLES ===
  #
  #   RandomNames()
  #   RandomNames(N = 20)
  #   RandomNames(cat = "common", MFprob = c(.2, .8))
  #
  # See:
  #   - http://www.census.gov/genealogy/www/data/1990surnames/names\_files.html
  #   - http://random-name-generator.info/

  if (is.null(dataset)) {
    if (!exists("CensusNames1990", where = 1)) {
      if (isTRUE(list.files(
        pattern = "~CensusNames.RData$" == "CensusNames.RData")) {
        load("CensusNames.RData")
      } else {
        ans = readline("
CensusNames.RData dataset not found in working directory.
CensusNames1990 object not found in workspace. \n
Download and load the dataset now? (y/n) ")
        if (ans != "y")
          return(invisible())
        require(RCurl)
        baseURL = c("https://raw.githubusercontent.com/mrdwab/2657-R-Functions/master/")
        temp = getBinaryURL(paste0(baseURL, "data/CensusNames.RData"))
        load(rawConnection(temp), envir=.GlobalEnv)
        message("CensusNames1990 data downloaded from \n",
                paste0(baseURL, "data/CensusNames.RData \n"),
                "and added to your workspace\n\n")
        rm(temp, baseURL)
      }
    }
    dataset <- CensusNames1990
  }
  TEMP <- dataset
```

```

possiblecats <- c("common", "rare", "average")
if(all(cat %in% possiblecats) == FALSE)
  stop(cat must be either "all", NULL,
       or a combination of "common", "average", or "rare",)
possiblelegenders <- c("male", "female", "both")
if (all(gender %in% possiblelegenders) == FALSE) {
  stop(gender must be either "both", NULL, "male", or "female",)
}
if (isTRUE(identical(gender, c("male", "female")))) ||
  isTRUE(identical(gender, c("female", "male")))) {
  gender <- "both"
}
if (is.null(cat) || cat == "all") {
  surnames <- TEMP[["surnames"]][["Name"]]
  malenames <- paste("M-", TEMP[["malenames"]][["Name"]], sep="")
  femalenames <- paste("F-", TEMP[["femalenames"]][["Name"]], sep="")
} else {
  surnames <- suppressWarnings(
    with(TEMP[["surnames"]],
         TEMP[["surnames"]][Category == cat, "Name"]))
  malenames <- paste("M-", suppressWarnings(
    with(TEMP[["malenames"]],
         TEMP[["malenames"]][Category == cat, "Name"])), sep="")
  femalenames <- paste("F-", suppressWarnings(
    with(TEMP[["femalenames"]],
         TEMP[["femalenames"]][Category == cat, "Name"])), sep="")
}

if (is.null(gender) || gender == "both") {
  if (is.null(MFprob)) MFprob <- c(.5, .5)
  firstnames <- sample(c(malenames, femalenames), N, replace = TRUE,
                      prob = c(rep(MFprob[1]/length(malenames),
                                   length(malenames)),
                              rep(MFprob[2]/length(femalenames),
                                   length(femalenames))))
} else if (gender == "female") {
  firstnames <- sample(femalenames, N, replace = TRUE)
} else if (gender == "male") {
  firstnames <- sample(malenames, N, replace = TRUE)
}

Surnames <- sample(surnames, N, replace = TRUE)
temp <- setNames(data.frame(do.call(rbind, strsplit(firstnames, "-")),
                          c("Gender", "FirstName")))
cbind(temp, Surnames)
}

```

Chapter 14

row.extractor

```
row.extractor = function(data, extract.by, what="all") {
  # Extracts rows with min, median, and max values, or by quantiles. Values
  # for --what-- can be "min", "median", "max", "all", or a vector
  # specifying the desired quantiles. Values for --extract.by-- can be
  # the variable name or number.
  #
  # === EXAMPLES ===
  #
  #   set.seed(1)
  #   dat = data.frame(V1 = 1:10, V2 = rnorm(10), V3 = rnorm(10),
  #                     V4 = sample(1:20, 10, replace=T))
  #   dat2 = dat[-10,]
  #   row.extractor(dat, 4, "all")
  #   row.extractor(dat1, 4, "min")
  #   row.extractor(dat, "V4", "median")
  #   row.extractor(dat, 4, c(0, .5, 1))
  #   row.extractor(dat, "V4", c(0, .25, .5, .75, 1))
  #
  # "which.quantile" function by cbeleites:
  # http://stackoverflow.com/users/755257/cbeleites
  # See: http://stackoverflow.com/q/10256503/1270695

  if (is.numeric(extract.by)) {
    extract.by = extract.by
  } else if (is.numeric(extract.by) != 0) {
    extract.by = which(colnames(data) %in% "extract.by")
  }

  if (is.character(what)) {
    which.median = function(data, extract.by) {
      a = data[, extract.by]
      if (length(a) %% 2 != 0) {
        which(a == median(a))
      } else if (length(a) %% 2 == 0) {
        b = sort(a)[c(length(a)/2, length(a)/2+1)]
        c(max(which(a == b[1])), min(which(a == b[2])))
      }
    }
  }

  X1 = data[which(data[extract.by] == min(data[extract.by])), ] # min
  X2 = data[which(data[extract.by] == max(data[extract.by])), ] # max
  X3 = data[which.median(data, extract.by), ] # median
}
```

```

if (identical(what, "min")) {
  X1
} else if (identical(what, "max")) {
  X2
} else if (identical(what, "median")) {
  X3
} else if (identical(what, "all")) {
  rbind(X1, X3, X2)
}
} else if (is.numeric(what)) {
  which.quantile <- function (data, extract.by, what, na.rm = FALSE) {

    x = data[ , extract.by]

    if (! na.rm & any (is.na (x)))
      return (rep (NA_integer_, length (what)))

    o <- order (x)
    n <- sum (! is.na (x))
    o <- o [seq_len (n)]

    nppm <- n * what - 0.5
    j <- floor(nppm)
    h <- ifelse((nppm == j) & ((j%2L) == 0L), 0, 1)
    j <- j + h

    j [j == 0] <- 1
    o[j]
  }
  data[which.quantile(data, extract.by, what), ] # quantile
}
}

```

Chapter 15

sample.size

```
sample.size <- function(population, samp.size = NULL, c.lev = 95,
                        c.int = NULL, what = "sample", distribution=50) {
  # Returns a data.frame of sample sizes or confidence intervals for
  # different conditions provided by the following arguments.
  #
  # --> populaton      Population size
  # --> samp.size      Sample size
  # --> c.lev          Confidence level
  # --> c.int          Confidence interval (+/-)
  # --> what           Whether sample size or confidence interval
  #                   is being calculated.
  # --> distribution   Response distribution
  #
  # === EXAMPLES ===
  #
  # sample.size(300)
  # sample.size(300, 150, what="confidence")
  # sample.size(c(300, 400, 500), c.lev=97)

  z = qnorm(.5+c.lev/200)

  if (identical(what, "sample")) {
    if (is.null(c.int)) {
      c.int = 5

      message("NOTE! Confidence interval set to 5.
              To override, set >> c.int << to desired value.\n")

    } else if (!is.null(c.int) == 1) {
      c.int = c.int
    }

    if (!is.null(samp.size)) {
      message("NOTE! >> samp.size << value provided but ignored.
              See output for actual sample size(s).\n")
    }

    ss = (z^2 * (distribution/100) *
          (1-(distribution/100)))/((c.int/100)^2)
    samp.size = ss/(1 + ((ss-1)/population))

  } else if (identical(what, "confidence")) {
```

```

if (is.null(samp.size)) {
  stop("Missing >> samp.size << with no default value.")
}
if (!is.null(c.int)) {
  message("NOTE! >> c.int << value provided but ignored.
  See output for actual confidence interval value(s).\n")
}

ss = ((population*samp.size-samp.size)/(population-samp.size))
c.int = round(sqrt((z^2 * (distribution/100) *
  (1-(distribution/100)))/ss)*100, digits = 2)

} else if (what %in% c("sample", "confidence") == 0) {
  stop(">> what << must be either -sample- or -confidence-")
}

RES = data.frame(population = population,
  conf.level = c.lev,
  conf.int = c.int,
  distribution = distribution,
  sample.size = round(samp.size, digits = 0))
RES
}

```

Chapter 16

stratified

```
stratified <- function(df, group, size, seed = NULL, ...) {
  # Returns a stratified random subset of a data.frame.
  #
  # --> df      The source data.frame
  # --> group   Your grouping variable
  # --> size    The desired sample size. If -size- is a decimal,
  #             a proportionate sample is drawn. If it is >= 1,
  #             a sample will be taken of that specified size
  # --> seed    The seed that you want to use, if any
  # --> ...     Further arguments to the sample function
  #
  # === EXAMPLES ===
  #
  #   set.seed(1)
  #   dat = data.frame(A = 1:100,
  #                     B = sample(c("AA", "BB", "CC", "DD", "EE"),
  #                               100, replace=T),
  #                     C = rnorm(100), D = abs(round(rnorm(100), digits=1)),
  #                     E = sample(c("CA", "NY", "TX"), 100, replace=T))
  #
  #   stratified(dat, 5, .1, 1)
  #   stratified(dat, group = "E", size = .1, seed = 1)
  #   stratified(dat, "B", 5)

  temp0 <- table(df[[group]])
  temp1 <- split(df, df[[group]])

  if (length(size) > 1) {
    if (length(size) != length(temp1))
      stop("Number of groups is ", length(temp1),
           " but number of sizes supplied is ", length(size))
    if (is.null(names(size))) {
      n <- setNames(size, names(temp1))
      message(sQuote("size"), " vector entered as:\n\nsize = structure(c(",
                    paste(n, collapse = ", "), ")\n\nNames = c(",
                    paste(shQuote(names(n)), collapse = ", "), ")\n\n")
    } else {
      ifelse(all(names(size) %in% names(temp1)),
              n <- size[names(temp1)],
              stop("Named vector supplied with names ",
                    paste(names(size), collapse = ", "),
                    "\n but the names for the group levels are ",

```

```

        paste(names(temp1), collapse = ", "))
    }
  } else {
    ifelse(size < 1,
      n <- setNames(
        round(temp0 * size, digits = 0), names(temp1)),
      ifelse(
        all(sapply(temp1, length) >= size),
        n <- setNames(rep(size, length.out = length(temp1)),
          names(temp1)),
        {
          temp <- sapply(temp1, length)
          message(
            "Some groups---",
            paste(names(temp[temp < size]), collapse = ", "),
            "---\ncontain fewer observations than desired number of samples.\n",
            "All observations have been returned from those groups.")
          n <- c(sapply(temp[temp >= size], function(x) x = size),
            temp[temp < size])[names(temp1)]
        })
  })
}

seedme <- ifelse(is.null(seed), "No", "Yes")

temp <- switch(
  seedme,
  No = { temp <- lapply(
    names(temp1),
    function(x) temp1[[x]][sample(temp0[x], n[x], ...), ]) },
  Yes = { temp <- lapply(
    names(temp1),
    function(x) { set.seed(seed)
      temp1[[x]][sample(temp0[x], n[x], ...), ] })})

rm(.Random.seed, envir=.GlobalEnv) # "resets" the seed
do.call("rbind", temp)
}

```


Chapter 17

stringseed.sampling

```
stringseed.sampling <- function(seedbase, N, n, write.output = FALSE) {
  # Designed for batch sampling scenarios using alpha-numeric strings as a
  # --seedbase--. --N-- represents the "population", and --n-- the sample
  # size needed. A vector is supplied for each argument (or, alternatively,
  # a data.frame with the required information). Optionally, the function
  # can write the output of the function to a file.
  #
  # === EXAMPLE ===
  #
  # stringseed.sampling(seedbase = c("Village 1", "Village 2", "Village 3"),
  #                     N = c(150, 309, 297), n = c(15, 31, 30))
  #
  # See: http://stackoverflow.com/q/10910698/1270695

  require(digest)
  hexval = paste0("0x", sapply(seedbase, digest, "crc32"))
  seeds = type.convert(hexval) %% .Machine$integer.max
  seedbase = as.character(seedbase)

  temp <- data.frame(seedbase, N, n, seeds)
  if (length(seedbase) == 1) {
    set.seed(temp$seeds); sample.list <- sample(temp$N, temp$n)
  } else {
    sample.list <- setNames(
      apply(temp[-1], 1, function(x)
        {set.seed(x[3]); sample(x[1], x[2])} ), temp[, 1])
  }

  temp <- list(
    input = data.frame(seedbase = seedbase, populations = N,
                      samplesizes = n, seeds = seeds),
    samples = sample.list)
  if(isTRUE(write.output)) {
    write.csv(temp[[1]], file=paste("Sample frame generated on",
                                   Sys.Date(), ".csv", collapse=""))
    capture.output(temp[[2]], file=paste("Samples generated on",
                                         Sys.Date(), ".txt", collapse=""))
  }
  rm(.Random.seed, envir=globalenv()) # "resets" the seed
  temp
}
```


Part III

Snippets and Tips

Chapter 18

Snippets

Load All Scripts and Data Files From Multiple Directories

```
load.scripts.and.data <- function(path, pattern = list(scripts = "*.R$",
                                                       data = "*.rda$|*.Rdata$"), ignore.case=TRUE) {
  # Reads all the data files and scripts from specified directories. In
  # general, should only need to specify the directories. Specify
  # directories without trailing slashes.
  #
  # === EXAMPLE ===
  #
  #   load.scripts.and.data(c("~/Dropbox/Public",
  #                           "~/Dropbox/Public/R Functions"))

  file.sources = list.files(path, pattern=pattern$scripts,
                             full.names=TRUE, ignore.case=ignore.case)
  data.sources = list.files(path, pattern=pattern$data,
                             full.names=TRUE, ignore.case=ignore.case)
  sapply(data.sources, load, .GlobalEnv)
  sapply(file.sources, source, .GlobalEnv)
}
```

Convert a List of Data Frames Into Individual Data Frames

```
unlist.dfs <- function(data) {
  # Specify the quoted name of the source list.
  q = get(data)
  prefix = paste0(data, "_", 1:length(q))
  for (i in 1:length(q)) assign(prefix[i], q[[i]], envir=.GlobalEnv)
}
```

Example

Note that the list name must be quoted.

```
# Sample data
temp = list(A = data.frame(A = 1:2, B = 3:4),
            B = data.frame(C = 5:6, D = 7:8))
temp
```

```
## $A
##   A B
## 1 1 3
## 2 2 4
##
## $B
##   C D
## 1 5 7
## 2 6 8

# Remove any files with similar names to output
rm(list=ls(pattern="temp_"))
# The following should not work
temp_1
```

```
## Error: object 'temp_1' not found
```

```
# Split it up!
unlist.dfs("temp")
# List files with the desired pattern
ls(pattern="temp_")
```

```
## [1] "temp_1" "temp_2"
```

```
# View the new files
temp_1
```

```
##   A B
## 1 1 3
## 2 2 4
```

```
temp_2
```

```
##   C D
## 1 5 7
## 2 6 8
```

Convert a Data Frame Into a List With Each Column Becoming a List Item

```
dfcols.list <- function(data, vectorize = FALSE) {
  # Specify the unquoted name of the data.frame to convert
  if (isTRUE(vectorize)) {
    dat.list = sapply(1:ncol(data), function(x) data[x])
  } else if (!isTRUE(vectorize)) {
    dat.list = lapply(names(data), function(x) data[x])
  }
  dat.list
}
```

Examples

```
# Sample data
dat = data.frame(A = c(1:2), B = c(3:4), C = c(5:6))
dat

##   A B C
## 1 1 3 5
## 2 2 4 6

# Split into a list, retaining data.frame structure
dfcols.list(dat)

## [[1]]
##   A
## 1 1
## 2 2
##
## [[2]]
##   B
## 1 3
## 2 4
##
## [[3]]
##   C
## 1 5
## 2 6

# Split into a list, converting to vector
dfcols.list(dat, vectorize=TRUE)

## $A
## [1] 1 2
##
## $B
## [1] 3 4
##
## $C
## [1] 5 6
```

Rename an Object in the Workplace

```
mv <- function (a, b) {
  # Source: https://stat.ethz.ch/pipermail/r-help/2008-March/156035.html
  anm = deparse(substitute(a))
  bnm = deparse(substitute(b))
  if (!exists(anm, where=1, inherits=FALSE))
    stop(paste(anm, "does not exist.\n"))
  if (exists(bnm, where=1, inherits=FALSE)) {
    ans = readline(paste("Overwrite ", bnm, "? (y/n) ", sep = " "))
    if (ans != "y")
      return(invisible())
  }
  assign(bnm, a, pos = 1)
  rm(list = anm, pos = 1)
  invisible()
}
```

Basic Usage

If there is already an object with the same name in the workplace, the function will ask you if you want to replace the object or not. Otherwise, the basic usage is:

```
# Rename "object_1" to "object_2"
mv(object_1, object_2)
```

Scrape Data From a Poorly Formatted HTML Page

Reformats a web page using HTML Tidy and uses the XML package to parse the resulting file. Can optionally save the reformatted page.

```
tidyHTML <- function(URL, saveTidy = TRUE) {
  require(XML)
  URL1 = gsub("/", "%2F", URL)
  URL1 = gsub(":", "%3A", URL1)
  URL1 = paste("http://services.w3.org/tidy/tidy?docAddr=",
              URL1, "&indent=on", sep = "")
  Parsed = htmlParse(URL1)
  if (isTRUE(saveTidy)) saveXML(Parsed, file = basename(URL))
  Parsed
}
```

Example

```
# Set „saveTidy„ to „TRUE„ to save the resulting tidied file
URL <- "http://www.bcn.gob.ni/estadisticas/trimestrales_y_mensuales/siec/datos/4.IMAE.htm"
temp <- tidyHTML(URL, saveTidy = FALSE)
```

“Rounding in Commerce”

R rounds to even—something that some people might not be accustomed to or comfortable with. For the more commonly known rounding rule, use this `round2` function.

```
round2 <- function(x, n = 0) {
  posneg = sign(x)
  z = abs(x)*10^n
  z = z + 0.5
  z = trunc(z)
  z = z/10^n
  z*posneg
}
```

Example

```
x = c(1.85, 1.54, 1.65, 1.85, 1.84)
round(x, 1)

## [1] 1.8 1.5 1.6 1.8 1.8

round2(x, 1)
```



```
## [1] 1.9 1.5 1.7 1.9 1.8

round(seq(0.5, 9.5, by=1))

## [1] 0 2 2 4 4 6 6 8 8 10

round2(seq(0.5, 9.5, by=1))

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

References

Original function: <http://www.webcitation.org/68dJeLBtJ> – see the comments section.
 See also: <http://stackoverflow.com/questions/12688717/round-up-from-5-in-r/>.

`cbind` `data.frames` When the Number of Rows are Not Equal

`cbind()` does not work when trying to combine `data.frames` with differing numbers of rows. This function takes a list of `data.frames`, identifies how many extra rows are required to make `cbind` work correctly, and does the combining for you.

The function also works with nested lists by first “flattening” them using the `LinearizeNestedList` by [Akhil S Bhel](#). The first time you run the `CBIND()` function, it check your current environment to identify whether `LinearizeNestedList` is already available; if it is not, it will download and load the function from its [Gist page](#). Subsequent calls to the function in the same session will not re-download the function.

```
CBIND <- function(datalist) {
  if ("LinearizeNestedList" %in% ls(envir=.GlobalEnv) == FALSE) {
    require(devtools)
    suppressMessages(source_gist(4205477))
    message("LinearizeNestedList loaded from https://gist.github.com/4205477")
  }
  datalist <- LinearizeNestedList(datalist)
  nrows <- max(sapply(datalist, nrow))
  expandmyrows <- function(mydata, rowsneeded) {
    temp1 = names(mydata)
    rowsneeded = rowsneeded - nrow(mydata)
    temp2 = setNames(data.frame(
      matrix(rep(NA, length(temp1) * rowsneeded),
            ncol = length(temp1))), temp1)
    rbind(mydata, temp2)
  }
  do.call(cbind, lapply(datalist, expandmyrows, rowsneeded = nrows))
}
```

Examples

```
# Example data
df1 <- data.frame(A = 1:5, B = letters[1:5])
df2 <- data.frame(C = 1:3, D = letters[1:3])
df3 <- data.frame(E = 1:8, F = letters[1:8], G = LETTERS[1:8])
# Try to use cbind directly
cbind(df1, df2, df3)
```

```
## Error: arguments imply differing number of rows: 5, 3, 8
```

```
# Use our new function
```

```
CBIND(list(df1, df2, df3))
```

```
##   1.A 1.B 2.C 2.D 3.E 3.F 3.G
## 1   1   a   1   a   1   a   A
## 2   2   b   2   b   2   b   B
## 3   3   c   3   c   3   c   C
## 4   4   d  NA <NA> 4   d   D
## 5   5   e  NA <NA> 5   e   E
## 6  NA <NA> NA <NA> 6   f   F
## 7  NA <NA> NA <NA> 7   g   G
## 8  NA <NA> NA <NA> 8   h   H
```

```
test1 <- list(df1, df2, df3)
```

```
str(test1)
```

```
## List of 3
```

```
## $ : 'data.frame': 5 obs. of 2 variables:
```

```
## ..$ A: int [1:5] 1 2 3 4 5
```

```
## ..$ B: Factor w/ 5 levels "a","b","c","d",...: 1 2 3 4 5
```

```
## $ : 'data.frame': 3 obs. of 2 variables:
```

```
## ..$ C: int [1:3] 1 2 3
```

```
## ..$ D: Factor w/ 3 levels "a","b","c": 1 2 3
```

```
## $ : 'data.frame': 8 obs. of 3 variables:
```

```
## ..$ E: int [1:8] 1 2 3 4 5 6 7 8
```

```
## ..$ F: Factor w/ 8 levels "a","b","c","d",...: 1 2 3 4 5 6 7 8
```

```
## ..$ G: Factor w/ 8 levels "A","B","C","D",...: 1 2 3 4 5 6 7 8
```

```
CBIND(test1)
```

```
##   1.A 1.B 2.C 2.D 3.E 3.F 3.G
## 1   1   a   1   a   1   a   A
## 2   2   b   2   b   2   b   B
## 3   3   c   3   c   3   c   C
## 4   4   d  NA <NA> 4   d   D
## 5   5   e  NA <NA> 5   e   E
## 6  NA <NA> NA <NA> 6   f   F
## 7  NA <NA> NA <NA> 7   g   G
## 8  NA <NA> NA <NA> 8   h   H
```

```
test2 <- list(test1, df1)
```

```
str(test2)
```

```
## List of 2
```

```
## $ :List of 3
```

```
## ..$ : 'data.frame': 5 obs. of 2 variables:
```

```
## .. ..$ A: int [1:5] 1 2 3 4 5
```

```
## .. ..$ B: Factor w/ 5 levels "a","b","c","d",...: 1 2 3 4 5
```

```
## ..$ : 'data.frame': 3 obs. of 2 variables:
```

```
## .. ..$ C: int [1:3] 1 2 3
```

```
## .. ..$ D: Factor w/ 3 levels "a","b","c": 1 2 3
```

```
## ..$ : 'data.frame': 8 obs. of 3 variables:
```

```
## .. ..$ E: int [1:8] 1 2 3 4 5 6 7 8
```

```
## .. ..$ F: Factor w/ 8 levels "a","b","c","d",...: 1 2 3 4 5 6 7 8
```

```
## ..$ G: Factor w/ 8 levels "A","B","C","D",...: 1 2 3 4 5 6 7 8
## $ : 'data.frame':   5 obs. of  2 variables:
## ..$ A: int [1:5] 1 2 3 4 5
## ..$ B: Factor w/ 5 levels "a","b","c","d",...: 1 2 3 4 5
```

```
CBIND(test2)
```

```
## 1/1.A 1/1.B 1/2.C 1/2.D 1/3.E 1/3.F 1/3.G 2.A 2.B
## 1      1      a      1      a      1      a      A      1      a
## 2      2      b      2      b      2      b      B      2      b
## 3      3      c      3      c      3      c      C      3      c
## 4      4      d     NA <NA>    4      d      D      4      d
## 5      5      e     NA <NA>    5      e      E      5      e
## 6     NA <NA>    NA <NA>    6      f      F     NA <NA>
## 7     NA <NA>    NA <NA>    7      g      G     NA <NA>
## 8     NA <NA>    NA <NA>    8      h      H     NA <NA>
```

Generate Random Names With an Online Service

This function uses the random name generator from the [Random Name Generator](http://random-name-generator.info/) website¹. This is included here mostly for “fun”, and to show how we can use form input parameters from some websites in our R scripts.

Note: Since there is no concept of a *seed* at the website from which these names are drawn, you should expect to get different names each time the function is run. If you want more control, but similar functionality, use the `RandomNames()` function instead.

Arguments

- `number`: How many names do you want?
- `gender`: Specify whether you want "male" names, "female" names, or "both".
- `type`: Specify whether you want "common", "average", or "rare" names.

```
randomNamesOnline <- function(number = 100, gender = "both", type = "rare") {
  gender <- tolower(gender); type <- tolower(type)
  gender <- switch(gender, both = "&g=1", male = "&g=2", female = "&g=3",
    stop("gender" must be either "male", "female", or, "both",))
  type <- switch(type, rare = "&st=3", average = "&st=2", common = "&st=1",
    stop("type" must be either "rare", "average", or "common",))
  tempURL <- paste("http://random-name-generator.info/random/?n=",
    number, gender, type, sep = "", collapse = "")
  temp <- suppressWarnings(readLines(tempURL))
  temp <- gsub("\t|<li>|</ol>", "", temp[102:(102 + number - 1)])
  temp
}
```

Examples

```
randomNamesOnline(10)
```

¹See: <http://random-name-generator.info/>

```
## [1] "Rod Casper"      "Elina Escobar"      "Rodrick Vickers"
## [4] "Alejandra Townes" "Shad Barela"        "Jackson Thurston"
## [7] "Darline Greenwood" "Tera Griswold"      "Alonso Deleon"
## [10] "Junko Ferraro"

randomNamesOnline(10, gender = "both", type = "common")

## [1] "Nancy Anderson" "Jesse Hall"          "Kathleen Flores" "Larry Reed"
## [5] "Thomas Rodriguez" "Roger Garcia"        "Linda Henderson" "Nicole Ward"
## [9] "Martin Walker"   "Brandon Diaz"

randomNamesOnline(10, "male", "average")

## [1] "Alejandro Becker" "Charles Black"      "Eddie Lewis"      "Alfred Soto"
## [5] "Joseph Rodriguez" "Eugene Walton"      "Ignacio Jordan"   "Ray Payne"
## [9] "Orville Wong"     "Edgar Jimenez"
```

Use strings to set seed when generating a random sample

The `stringseed.basic()` function is a more simplified (but less robust) seed generator and random sampling function, similar to the `stringseed.sampling()` function. Unlike `stringseed.sampling()`, this function does not require loading any extra packages for generating the seed, but uses basic methods such as letter substitutions and basic numeric transformations to create some “noise” before assigning a seed.

The function was originally written for students at the Tata-Dhan Academy (and named `TDASample()`) to help them draw samples during their fieldwork. See the Appendix for a more detailed concept note on how the function works and how it was expected to be used.

```
stringseed.basic <- function(inString, N, n, toFile = FALSE) {
  if (is.factor(inString)) inString <- as.character(inString)
  if (nchar(inString) <= 3) stop("inString must be > 3 characters")
  string1 <- "jnt3g127rbfeqixkos 586d90pyal4chzmvwu"
  string2 <- "2dyn0uxq ovalrpk sieb3fhjw584cm9t7z16g"
  instring <- chartr(string1, string2, tolower(inString))
  t1 <- sd(c(suppressWarnings(sapply(strsplit(instring, ""),
                                         as.numeric))), na.rm = TRUE)
  t2 <- c(sapply(strsplit(instring, " "), nchar))
  t3 <- c(na.omit(sapply(strsplit(instring, ""), match, letters)))
  seed <- floor(sum(t1, sd(t2), mean(t2), prod(fivenum(t3)),
                  mean(t3), sd(t3), na.rm=TRUE))

  set.seed(seed)
  temp0 <- sample(N, n)

  temp1 <- list(
    Metadata =
      noquote(c(sprintf("                The sample was drawn on: %s.",
                        Sys.time()),
                sprintf("                The seed input was: %s.",
                        inString),
                sprintf("The total number of households was: %d.", N),
                sprintf(" The desired number of samples was: %d.", n))),
    SeedUsed = seed,
    FinalSample = temp0,
    FinalSample_sorted = sort(temp0))
}
```

```

rm(.Random.seed, envir=globalenv())

if (isTRUE(toFile)) {
  capture.output(temp1,
    file = paste("Sample from",
      Sys.Date(), ".txt",
      collapse=""),
    append = TRUE)
}
temp1
}

```

Draw a stratified random sample from a data.frame

The `strata.sampling()` function is a wrapper around the `strata()` and `getdata()` functions from the “sampling” package that allows you to quickly draw a stratified random sample from a `data.frame` after specifying a single grouping variable. The sample sizes can be proportionate to the number of observations in each strata, a uniform size from all groups, or a vector specifying the number of samples to take from each group.

A very similar function (but one which does not depend on other packages) is the `stratified()` function.

```

strata.sampling <- function(data, group, size, method = NULL) {
  # USE:
  # * Specify a data.frame and grouping variable.
  # * Decide on your sample size. For a sample proportional to the
  #   population, enter "size" as a decimal. For an equal number of
  #   samples from each group, enter "size" as a whole number. For
  #   a specific number of samples from each group, enter the numbers
  #   required as a vector.

  require(sampling)
  if (is.null(method)) method <- "srswor"
  if (!method %in% c("srswor", "srswr"))
    stop("method must be 'srswor' or 'srswr'.")
  temp <- data[order(data[[group]]), ]
  ifelse(length(size) > 1,
    size <- size,
    ifelse(size < 1,
      size <- round(table(temp[group]) * size),
      size <- rep(size, times=length(table(temp[group])))))
  strat = strata(temp, stratanames = names(temp[group]),
    size = size, method = method)
  getdata(temp, strat)
}

```

Examples

```

# Generate a couple of sample data.frames to play with
set.seed(1)
dat1 <- data.frame(ID = 1:100,
  A = sample(c("AA", "BB", "CC", "DD", "EE"), 100, replace=T),
  B = rnorm(100), C = abs(round(rnorm(100), digits=1)),
  D = sample(c("CA", "NY", "TX"), 100, replace=T))

```

```
# Let,s take a 10% sample from all -A- groups in dat1
strata.sampling(dat1, "A", .1)
```

##	ID	B	C	D	A	ID_unit	Prob	Stratum
## 88	88	-0.5283	1.2	NY	AA	11	0.07692	1
## 11	11	2.4016	0.4	TX	BB	17	0.08000	2
## 89	89	-0.6521	1.1	TX	BB	37	0.08000	2
## 67	67	-0.3200	0.3	CA	CC	53	0.10526	3
## 97	97	2.0872	0.5	NY	CC	56	0.10526	3
## 37	37	1.0631	1.5	NY	DD	66	0.11538	4
## 39	39	0.3700	0.4	CA	DD	67	0.11538	4
## 100	100	-1.6406	1.0	NY	DD	83	0.11538	4
## 6	6	1.9804	1.1	TX	EE	85	0.11765	5
## 99	99	-1.2863	0.2	NY	EE	100	0.11765	5

```
# Let,s take 5 samples from all -D- groups in dat1,
# specified by column number
strata.sampling(dat1, group = 5, size = 5)
```

##	ID	A	B	C	D	ID_unit	Prob	Stratum
## 39	39	DD	0.37002	0.4	CA	8	0.2174	1
## 46	46	DD	0.55849	1.0	CA	11	0.2174	1
## 53	53	CC	-0.91092	1.6	CA	12	0.2174	1
## 58	58	CC	0.91017	0.5	CA	14	0.2174	1
## 87	87	DD	-0.30098	0.6	CA	22	0.2174	1
## 4	4	EE	-1.12936	0.9	NY	26	0.1190	2
## 12	12	AA	-0.03924	0.2	NY	30	0.1190	2
## 31	31	CC	-0.56867	1.2	NY	37	0.1190	2
## 40	40	CC	0.26710	0.9	NY	41	0.1190	2
## 61	61	EE	-0.63574	0.2	NY	48	0.1190	2
## 11	11	BB	2.40162	0.4	TX	70	0.1429	3
## 21	21	EE	0.47551	2.3	TX	73	0.1429	3
## 27	27	AA	-0.44329	0.8	TX	76	0.1429	3
## 51	51	CC	-0.62037	0.4	TX	83	0.1429	3
## 80	80	EE	-0.32427	0.3	TX	95	0.1429	3

```
# Let,s take a sample from all -A- groups in dat1
# where we specify the number wanted from each group
strata.sampling(dat1, "A", size = c(3, 5, 4, 5, 2))
```

##	ID	B	C	D	A	ID_unit	Prob	Stratum
## 27	27	-0.44329	0.8	TX	AA	4	0.2308	1
## 47	47	-1.27659	1.4	TX	AA	7	0.2308	1
## 56	56	1.76729	2.5	NY	AA	9	0.2308	1
## 22	22	-0.70995	0.1	TX	BB	20	0.2000	2
## 26	26	0.29145	0.0	TX	BB	22	0.2000	2
## 57	57	0.71671	0.7	CA	BB	26	0.2000	2
## 78	78	-0.03763	1.6	NY	BB	33	0.2000	2
## 86	86	-1.53645	0.1	NY	BB	36	0.2000	2
## 16	16	0.18879	2.2	CA	CC	40	0.2105	3
## 40	40	0.26710	0.9	NY	CC	44	0.2105	3
## 48	48	-0.57327	1.0	NY	CC	47	0.2105	3
## 75	75	-0.10019	1.2	TX	CC	54	0.2105	3
## 9	9	0.56972	1.4	TX	DD	59	0.1923	4
## 23	23	0.61073	0.5	NY	DD	64	0.1923	4

##	65	65	-0.20738	1.0	TX	DD	74	0.1923	4
##	68	68	-0.27911	1.3	NY	DD	75	0.1923	4
##	85	85	0.30656	0.1	NY	DD	78	0.1923	4
##	18	18	1.46555	1.4	NY	EE	87	0.1176	5
##	94	94	-0.46353	0.9	NY	EE	99	0.1176	5

Chapter 19

Tips

Many of the following tips are useful for reducing repetitious tasks. They might seem silly or unnecessary with the small examples provided, but they can be *huge* time-savers when dealing with larger objects or larger sets of data.

Batch Convert Factor Variables to Character Variables

In the example data below, `author` and `title` are automatically converted to factor (unless you add the argument `stringsAsFactor = FALSE` when you are creating the data). What if you forgot and actually needed the variables to be in mode `as.character` instead?

Use `sapply` to identify which variables are currently factors and convert them to `as.character`.

```
dat = data.frame(title = c("title1", "title2", "title3"),
                 author = c("author1", "author2", "author3"),
                 customerID = c(1, 2, 1))

str(dat)

## 'data.frame':   3 obs. of  3 variables:
## $ title      : Factor w/ 3 levels "title1","title2",...: 1 2 3
## $ author     : Factor w/ 3 levels "author1","author2",...: 1 2 3
## $ customerID: num  1 2 1

# Left of the equal sign identifies and extracts the factor variables;
# right converts them from factor to character
dat[sapply(dat, is.factor)] = lapply(dat[sapply(dat, is.factor)],
                                     as.character)

str(dat)

## 'data.frame':   3 obs. of  3 variables:
## $ title      : chr  "title1" "title2" "title3"
## $ author     : chr  "author1" "author2" "author3"
## $ customerID: num  1 2 1
```

Using Reduce to Merge Multiple Data Frames at Once

The `merge` function in R only merges two objects at a time. This is usually fine, but what if you had several `data.frames` that needed to be merged?

Consider the following data, where we want to take monthly tables and merge them into an annual table:

```
set.seed(1)
JAN = data.frame(ID = sample(5, 3), JAN = sample(LETTERS, 3))
FEB = data.frame(ID = sample(5, 3), FEB = sample(LETTERS, 3))
MAR = data.frame(ID = sample(5, 3), MAR = sample(LETTERS, 3))
APR = data.frame(ID = sample(5, 3), APR = sample(LETTERS, 3))
```

If we wanted to merge these into a single `data.frame` using `merge`, we might end up creating several temporary objects and merging those, like this:

```
temp_1 = merge(JAN, FEB, all=TRUE)
temp_2 = merge(temp_1, MAR, all=TRUE)
temp_3 = merge(temp_2, APR, all=TRUE)
```

Or, we might nest a whole bunch of `merge` commands together, something like this:

```
merge(merge(merge(JAN, FEB, all=TRUE),
              MAR, all=TRUE),
      APR, all=TRUE)
```

However, that first option requires a lot of unnecessary typing and produces unnecessary objects that we then need to remember to remove, and the second option is not very reader-friendly—try doing a merge like that with, say, 12 `data.frames` if we had an entire year of data!

Use `Reduce` instead, simply specifying all the objects to be merged in a `list`:

```
Reduce(function(x, y) merge(x, y, all=TRUE),
       list(JAN, FEB, MAR, APR))
```

```
##   ID  JAN  FEB  MAR  APR
## 1  2    X    E    R    F
## 2  3 <NA>    F    X    D
## 3  4    V <NA>    M    Q
## 4  5    F    B <NA> <NA>
```

How Much Memory Are the Objects in Your Workspace Using?

Sometimes you need to just check and see how much memory the objects in your workspace occupy.

```
sort(sapply(ls(), function(x) {object.size(get(x))}))
```

Convert a Table to a Data Frame

Creating tables are easy and fast, but sometimes, it is more convenient to have the output as a `data.frame`. Get the `data.frame` by nesting the command in `as.data.frame.matrix`.

```
# A basic table
x <- with(airquality, table(cut(Temp, quantile(Temp)), Month))
str(x)

##  'table' int [1:4, 1:5] 24 5 1 0 3 15 7 5 0 2 ...
##  - attr(*, "dimnames")=List of 2
##    ..$      : chr [1:4] "(56,72]" "(72,79]" "(79,85]" "(85,97]"
##    ..$ Month: chr [1:5] "5" "6" "7" "8" ...
```

x

```
##           Month
##           5  6  7  8  9
## (56,72] 24  3  0  1 10
## (72,79]  5 15  2  9 10
## (79,85]  1  7 19  7  5
## (85,97]  0  5 10 14  5
```

The same table as a data.frame

```
y <- as.data.frame.matrix(x)
str(y)
```

```
## 'data.frame':  4 obs. of  5 variables:
## $ 5: int  24 5 1 0
## $ 6: int  3 15 7 5
## $ 7: int  0 2 19 10
## $ 8: int  1 9 7 14
## $ 9: int 10 10 5 5
```

y

```
##           5  6  7  8  9
## (56,72] 24  3  0  1 10
## (72,79]  5 15  2  9 10
## (79,85]  1  7 19  7  5
## (85,97]  0  5 10 14  5
```


Part IV

Appendices

Appendix A

Sample Generator for Students at the Tata-Dhan Academy

Abstract: This note¹ describes a function written to assist students at the Tata-Dhan Academy to generate random samples in a systematic and reproducible (and, thus, verifiable) manner. A common method for reproducible random samples is to use the *seed* function available in major statistics and data analysis software packages. To minimize researcher bias, even the choice of seed must be justified. The function described in this note obfuscates the seed setting process but still results in output that is reproducible. Furthermore, the seed used for generating the sample is included in the output to allow others to independently validate the results.

Many times, students need to do a pretty straightforward task of taking a random sample of households from a given village to complete their study. There are a lot of random number generators available. For instance, most scientific calculators have a feature to generate random numbers, spreadsheets often have a `RAND()` function, and student statistics or research textbooks may have a random table in their appendix. However, none of these methods are verifiable or reproducible.²

Common statistics and data analysis software (for example SPSS, Stata, and R) use the concept of a **seed** with their random number generator. These packages have their own methods for automatically setting a seed so that there are different numbers each time a function that uses a random number generator is run; this number is not readily visible to the end user. Most professional packages will also allow the user to specify the seed, in case they want to make their result reproducible, for instance if they want to share their scripts with another user to verify the output.

Following is a simple example of where using a seed is useful. Using R, we are going to draw a sample of 10 from a population of 50 twice. You'll note that the resulting samples are different. After that, we will set the seed (arbitrarily) to 1 and repeat the exercise.

```
sample(50, 10)
```

```
## [1] 30 35 22 46 5 37 49 31 47 15
```

```
sample(50, 10)
```

```
## [1] 41 35 33 47 44 8 43 37 1 34
```

¹This concept note was written on 24 December 2012 by Ananda Mahto and relates to V1.1 of the `TDASample()` function. Please consider using the `stringseed.basic()` function (which may be more up-to-date) or the `stringseed.sampling()` function (which uses a more robust method for generating seeds). Both can be found at the [2657-R-Functions Github page: https://github.com/mrdwab/2657-R-Functions](https://github.com/mrdwab/2657-R-Functions).

²It may seem counter-intuitive to want to reproduce a *random* sequence, but this is sometimes important in research settings. It is not uncommon to hear, for example, a question like “How did you ‘randomly’ select your sample?”

```

set.seed(1)
sample(50, 10)

## [1] 14 19 28 43 10 41 42 29 27 3

set.seed(1)
sample(50, 10)

## [1] 14 19 28 43 10 41 42 29 27 3

```

As can be seen, by using the `set.seed()` function in R, you are able to generate a verifiable sample.

Since there is a choice (in other words, user decision) in selecting a seed, you do still run the risk of introducing bias. An investigator who wants household 46 to be a part of their sample might, for instance, try different seeds until they get a sample that includes 46 in its selection (in this case, `set.seed(3); sample(50, 10)` would include household 46). Also, some people are simply confused by the concept of a seed and do not really want to think about what it is or why it is necessary.

Most of the students at the Tata-Dhan Academy conduct several participatory rural appraisals before getting into more traditional research methods. Of these varied methods, it is expected that the “*social mapping*” exercise would assist them in any subsequent sampling exercises they may need to do for their study. After completing a social mapping exercise, students are generally able to provide a list something like the following:

Household_ID	Head_of_Household
-----	-----
1	A Umarani
2	LB Rajkumar
3	Damodar Jena
...	...
...	...
118	Madhan Kumar
119	Ananda Mahto
120	JAN Vijayabharathi

The sampling function presented in this note makes use of this information to help students generate a reproducible random (non-stratified) sample (without replacement) of all the households without having to think about what an appropriate seed would be. The envisaged usage is that the student would enter a string, the population size they are sampling from, and the desired number of samples. The string can be anything, but for the case of reproducible analysis using available data, it is suggested that the string should be the name of the first person and the last person in the village “census” (as illustrated in the example household listing) or the village name. The social mapping exercise would also be the basis for N (the total number of households).

The TDASample() Function

The following code can be copied and pasted into an R session to make the function available to the user. For convenience, you should copy and paste the function into a plain text file, save that file to your system as “TDASample.R”, and load it by typing `source("--path/to/file--")`. For instance, if you saved the file to a folder called “Scripts” in your “C” drive, you can load it using `source("C:/Scripts/TDASample.R")`. Alternatively, if an internet connection is available, you can load the function by typing: `source("http://ideone.com/plain/BR066P")`

```

TDASample <- function(inString, N, n, toFile = FALSE) {
  if (is.factor(inString)) inString <- as.character(inString)
  if (nchar(inString) <= 3) stop("inString must be > 3 characters")

```



```

string1 <- "jnt3g127rbfeqixkos 586d90pyal4chzmvwu"
string2 <- "2dyn0uxq ovalrpksieb3fhjw584cm9t7z16g"
instring <- chartr(string1, string2, tolower(inString))
t1 <- sd(c(suppressWarnings(sapply(strsplit(instring, ""),
                                   as.numeric))), na.rm = TRUE)
t2 <- c(sapply(strsplit(instring, " "), nchar))
t3 <- c(na.omit(sapply(strsplit(instring, ""), match, letters)))
seed <- floor(sum(t1, sd(t2), mean(t2), prod(fivenum(t3)),
                 mean(t3), sd(t3), na.rm=TRUE))

set.seed(seed)
temp0 <- sample(N, n)

temp1 <- list(
  Metadata =
    noquote(c(sprintf("          The sample was drawn on: %s.",
                      Sys.time()),
              sprintf("          The seed input was: ,%s,",
                      inString),
              sprintf("The total number of households was: %d.", N),
              sprintf(" The desired number of samples was: %d.", n))),
  SeedUsed = seed,
  FinalSample = temp0,
  FinalSample_sorted = sort(temp0))

rm(.Random.seed, envir=globalenv())

if (isTRUE(toFile)) {
  capture.output(temp1,
    file = paste("Sample from",
                 Sys.Date(), ".txt",
                 collapse=""),
    append = TRUE)
}
temp1
}

```

Function Arguments

- **inString**: A quoted string. The name of the first person and last person from your social mapping result is recommended. For instance, using the example data provided earlier, the **inString** value for this dataset would be "A Umarani, JAN Vijayabharathi".
- **N**: The number of households. From the example above, **N** = 120.
- **n**: The desired number of samples.
- **toFile**: Logical. Should the output of your sample be written to a file? If **toFile** = **TRUE**, a file named "*Sample from -Date-.txt*" (where date is the current date) will be written to your working directory. The contents of this file will be appended to if further samples are run using the **TDASample()** function.

Examples

```
TDASample("A Umarani, JAN Vijayabharathi", 120, 30)
```

```
## $Metadata
```

```

## [1]          The sample was drawn on: 2012-12-24 11:39:15.
## [2]          The seed input was: 'A Umarani, JAN Vijayabharathi'
## [3] The total number of households was: 120.
## [4] The desired number of samples was: 30.
##
## $SeedUsed
## [1] 171640
##
## $FinalSample
## [1] 1 75 49 53 119 105 20 71 55 12 95 62 113 18 2 50 99 22 110
## [20] 46 26 85 15 54 70 118 5 83 78 60
##
## $FinalSample_sorted
## [1] 1 2 5 12 15 18 20 22 26 46 49 50 53 54 55 60 62 70 71
## [20] 75 78 83 85 95 99 105 110 113 118 119

# Manual verification. Compare results below with "FinalSample" above
set.seed(187241); sample(120, 30)

## [1] 27 23 45 32 14 54 17 56 90 101 70 105 119 118 22 72 107 78 113
## [20] 117 40 69 68 89 61 94 85 62 109 44

# Was a file written with our output?
list.files(pattern="Sample from")

## character(0)

# Nope. Nothing was written. Let,s write the output to a file.
TDASample("A Umarani, JAN Vijayabharathi", 120, 30, toFile=TRUE)

## $Metadata
## [1]          The sample was drawn on: 2012-12-24 11:39:15.
## [2]          The seed input was: 'A Umarani, JAN Vijayabharathi'
## [3] The total number of households was: 120.
## [4] The desired number of samples was: 30.
##
## $SeedUsed
## [1] 171640
##
## $FinalSample
## [1] 1 75 49 53 119 105 20 71 55 12 95 62 113 18 2 50 99 22 110
## [20] 46 26 85 15 54 70 118 5 83 78 60
##
## $FinalSample_sorted
## [1] 1 2 5 12 15 18 20 22 26 46 49 50 53 54 55 60 62 70 71
## [20] 75 78 83 85 95 99 105 110 113 118 119

# Check again
list.files(pattern="Sample from")

## [1] "Sample from 2012-12-24 .txt"

cat(noquote(readLines(list.files(pattern="Sample from")[1])), sep="\n")

```

```
## $Metadata
## [1]          The sample was drawn on: 2012-12-24 11:39:15.
## [2]          The seed input was: 'A Umarani, JAN Vijayabharathi'
## [3] The total number of households was: 120.
## [4] The desired number of samples was: 30.
##
## $SeedUsed
## [1] 171640
##
## $FinalSample
## [1] 1 75 49 53 119 105 20 71 55 12 95 62 113 18 2 50 99 22 110
## [20] 46 26 85 15 54 70 118 5 83 78 60
##
## $FinalSample_sorted
## [1] 1 2 5 12 15 18 20 22 26 46 49 50 53 54 55 60 62 70 71
## [20] 75 78 83 85 95 99 105 110 113 118 119

# Try a different string,
# for example a seed based on a village name
TDASample("Melakkal", 120, 30)
```

```
## $Metadata
## [1]          The sample was drawn on: 2012-12-24 11:39:15.
## [2]          The seed input was: 'Melakkal'
## [3] The total number of households was: 120.
## [4] The desired number of samples was: 30.
##
## $SeedUsed
## [1] 6032
##
## $FinalSample
## [1] 89 35 43 26 60 83 25 75 58 101 40 104 5 30 47 53 28 103 98
## [20] 94 67 7 62 27 42 108 69 29 31 78
##
## $FinalSample_sorted
## [1] 5 7 25 26 27 28 29 30 31 35 40 42 43 47 53 58 60 62 67
## [20] 69 75 78 83 89 94 98 101 103 104 108
```

Advanced Example

It is possible to use this in a more sophisticated way, for instance to perform batch sampling provided a `data.frame` with at least the following information:

1. A column containing the information to be used as your `inString`.
2. A column containing the “population” from which to draw a sample.
3. A column containing the desired sample size.

Here is one such dataset:

```
myListOfPlaces <- data.frame(
  villageName = c("Melakkal", "Sholavandan", "T. Malaipatti"),
  population = c(120, 130, 140),
  requiredSample = c(30, 25, 12))
myListOfPlaces
```

```
##      villageName population requiredSample
## 1      Melakkal      120          30
## 2    Sholavandan      130          25
## 3 T. Malaipatti      140          12
```

To batch generate the samples, you can use `apply()`, specifying the column numbers to be used for each argument. For instance, `inString` is represented by the first column (`x[1]`), `N` by the second (`x[2]`), and `n` by the third (`x[3]`).

```
setNames(apply(myListOfPlaces, 1, function(x)
  TDASample(x[1], as.numeric(x[2]), as.numeric(x[3])),
  myListOfPlaces[[1]]))

## $Melakkal
## $Melakkal$Metadata
## [1]          The sample was drawn on: 2012-12-24 11:39:15.
## [2]          The seed input was: 'Melakkal'
## [3] The total number of households was: 120.
## [4] The desired number of samples was: 30.
##
## $Melakkal$SeedUsed
## [1] 6032
##
## $Melakkal$FinalSample
## [1] 89 35 43 26 60 83 25 75 58 101 40 104 5 30 47 53 28 103 98
## [20] 94 67 7 62 27 42 108 69 29 31 78
##
## $Melakkal$FinalSample_sorted
## [1] 5 7 25 26 27 28 29 30 31 35 40 42 43 47 53 58 60 62 67
## [20] 69 75 78 83 89 94 98 101 103 104 108
##
##
## $Sholavandan
## $Sholavandan$Metadata
## [1]          The sample was drawn on: 2012-12-24 11:39:15.
## [2]          The seed input was: 'Sholavandan'
## [3] The total number of households was: 130.
## [4] The desired number of samples was: 25.
##
## $Sholavandan$SeedUsed
## [1] 26909
##
## $Sholavandan$FinalSample
## [1] 31 39 119 71 56 59 63 49 44 104 24 16 79 107 85 54 125 34 105
## [20] 4 76 47 55 62 48
##
## $Sholavandan$FinalSample_sorted
## [1] 4 16 24 31 34 39 44 47 48 49 54 55 56 59 62 63 71 76 79
## [20] 85 104 105 107 119 125
##
##
## $T. Malaipatti`
## $T. Malaipatti`$Metadata
## [1]          The sample was drawn on: 2012-12-24 11:39:15.
## [2]          The seed input was: 'T. Malaipatti'
## [3] The total number of households was: 140.
## [4] The desired number of samples was: 12.
##
```

```
## $`T. Malaipatti`$SeedUsed
## [1] 482178
##
## $`T. Malaipatti`$FinalSample
## [1] 75 17 4 107 96 16 68 79 27 99 120 93
##
## $`T. Malaipatti`$FinalSample_sorted
## [1] 4 16 17 27 68 75 79 93 96 99 107 120
```

How the Function Works

The function works by using various methods to generate “noise” in your `inString` and ultimately converting your `inString` to a numeric value (though in a somewhat obfuscated manner) that can be used as the seed in R. For instance, at one level, the actual string you input is changed using basic character replacement techniques. Any numeric values in the resulting string are extracted and basic functions (like taking their product, or means, or standard deviation) are applied to add further “noise”. Characters are converted to numeric values based on their position in the alphabet, and similar basic functions are applied to them as part of the “formula” for generating the seed. Once the seed is generated, the sample is drawn and displayed in a convenient format that can be used for reporting purposes.

It should be noted that the method of adding “noise” might actually not be noisy enough. At the base is simple character replacement (for example, replacing all instances of “a” with, say, “x”). Thus, using “ananda” as your `inString` will result in the same seed as using “adnana”; ideally, this would not be the case. There are methods around this, for instance using the `digest` package to convert a string to a `crc32` value and then converting it to a number to use as a seed³. However, the function shared here should be fine for most student uses, and has the advantage that no extra R packages are required for the function to run.

³A function that uses this approach is “stringseed.sampling” available at <https://github.com/mrdwab/2657-R-Functions/blob/master/scripts/stringseed.sampling.R>