# 1 Overview and History of R

**What is R?** A dialect of S

**What is S?**

- Developed by John Chambers at Bell Labs
- Initiated in 1976 as internal statistical analysis environment, initially as Fortran libraries.
- Early versions didn't contain functions for statistical modeling
- In 1988, it was rewritten in C to increase portability, and Version 3 was released, starting to resemble the curren version.
  - *Statistical Models in S* (the white book) by Chambers and Hastie documents statistical analysis functionality.
- Version 4 was released in 1998, and is the current version
  - *Programming with Data* by Chambers (green book) documents this version.
  - Fundamentals have not significantly changed.

**Historical Notes on S**

- In 1993 Bell Labs gave StatSci (now Insightful Corp.) an exclusive license to develop and sell S.
- In 2004, Insightful purchased S from Lucent (Bell Labs) for \$2m, and is current owner.
- In 2006, Alcatel bought Lucent; now Alcatel-Lucent.
- In 2008, TIBCO acquired Insightful for \$25m
- Insightful sells its implementation as S-PLUS (plus because it has GUIs, etc.)
- In 1998, S won the Association for Computing Machinery's prestigious Software System Award.

**S Philosophy** In *"Stages in the Evolution of S"* Chambers writes:
"We wanted users to be able to begin in an interactive environment where they did not consciously think of themselves as programming. Then as their needs became clearer and their sophistication increased, they should be able to slide gradually into programming, when the language and systm aspects would become more important."

## Background on R

### History of R

- 1991: Created in New Zealand by Ross Ihaka and Robert Gentleman. Development documented in 1996 JCGS paper.
- 1993: First announcement of R to the public.
- 1995: Martin Mächler convinces them to use GNU General Public License.
- 1996: Public mailing lists created (R-help and R-devel)
- 1997: R Core Group formed. Controls source. Contains some people associated with S-PLUS.
- 2000: R version 1.0.0 released
- 2013: R version 3.0.2 released in December

### Features of R

- Syntax very similar to S, so easy for S-PLUS users to switch over (though nowadays R is far more common).
- Semantics are superficially similar to S, though there are deep differences.
- Runs on almost any standard software platform / operating system (even the PS3).
- Frequent releases (annual and bugfix); active development.
- Quite lean for software; functionality divided into modular packages.
- Sophisticated graphics capabilities, better than most statistics packages.
- Useful for interactive work, but contains powerful programming language for developing new tools (reflecting S' philosophy of user becoming programmer).
- Very active and vibrant user community (R-help, R-devel, and Stackoverflow)
- Free (in sense of beer and speech); the FSF defines some applicable freedoms:
  - Freedom 0: freedom to run for any purpose
  - Freedom 1: freedom to study how it works, and adapt it to own needs. Access to source is a precondition of this freedom.
  - Freedom 2: freedom to redistribute copies.
  - Freedom 3: freedom to improve program, and release improvements to public, benefiting the whole community. Access to source is a precondition of this freedom.

**Drawbacks**

- Based on 40 year old technology
- Little built in support for dynamic and 3D graphics, though this is improving.
- Functionality is based on consumer demand and user contributions, so might need to implement a desired method single-handedly (or pay somebody to do it).
- Objects generally must be stored in physical memory, therefore there are size restrictions on what can study by default; there are advancements being made, such as computers with obscene amounts of memory.
- Not ideal for all possible situations (though this is true of all languages).

**Design of R System**

- Divided into two conceptual parts:
  1. "base" R system downloaded from CRAN
  2. everything else
- R functionality is divided into a number of packages
  - The base system contains **base** package, which is required to run R and contains most fundamental functions; other packages in the base system include: **utils, stats, datasets, graphics, grDevices, grid, methods, tools, parallel, compiler, splines, TclTk,** and **stats4.**
  - Also includes "Recommend" packages which are commonly used but not critical: **boot, class, cluster, codetools, foreign, KernSmooth, lattice, mgcv, nlme, rpart, survival, MASS, spatial, mmet,** and **Matrix.**
  - There are many other packages available via CRAN and BioConductor. In part because CRAN imposes restrictions to ensure the quality of packages it publishes, many people host packages on their personal websites, and there's no telling how many of these there are.

**Some Resources for Learning R**

- CRAN offers a number of resources:
  - An Introduction to R
  - Writing R Extensions
  - R Data Import / Export

  - R Installation and Administration (mostly for building R from sources)
  - R Internals (highly technical; not for faint of heart)

- Standard texts include
  - Chambers (2008). *Software for Data Analysis,* Springer.
  - Chambers (1998). *Programming with Data,* Springer.
  - Venables & Ripley (2002). *Modern Applied Statistics with S*, Springer.
  - Venables & Ripley (2000). *S Programming*, Springer.
  - Pinheiro & Bates (2000). *Mixed-Effects Models in S and S-PLUS*, Springer.
  - Murrell (2005). *R Graphics*, Chapman & Hall / CRC Press.
- Other resources:
  - "*Use R!*" book series by Springer
  - bibliography at http://www.r-project.org/doc/bib/R-books.html

**Getting Help** Dos:

- Describe goal, not just the step, because people may have a better solution for what trying to do.
- Be explicit about what trying to do.
- Provide the minimum needed amount of information (volume is not precision).
- Follow up

Do not:

- Claim that found bug; 99% of time you're wrong and look foolish
- Grovel about homework, or post homework
- Email multiple lists at once
- Ask others to debug code without giving some hint as to what the problem might be.

## 2 Data Types

### Objects

- Everything that you manipulate in R is an object
- R has five basic or *atomic* classes of objects:
  - character
  - numeric (real numbers)
  - integer
  - complex
  - logical (TRUE/FALSE or T/F)
- Most basic object in R is a vector
  - contains multiple objects, but only objects of the same class; the exception is lists, which are represented as a vector, but can contain objects of different classes
  - empty vectors can be created with the *vector()* function, which takes class of objects as the first arg, and number of objects as second.

### Numbers

- Numbers are generally treated as numeric objects (double precision real numbers).
- If explicitly want an integer, must specify the *L* suffix.
  - ex: 1 gives a numeric object, but *1L* explicitly gives integer object.
- Special number Inf represents infinity, e.g., 1/0=Inf
  - *Inf* can be used in ordinary calculations, e.g., 1/Inf=0
- Value NaN represents an undefined value, e.g., 0/0
  - can also be thought of as a missing value

### Attributes

- R objects can (but don't need to) have attributes.
- Common attributes include names, dimnames, dimensions (e.g matrices, arrays), class, and length. The user can also define attributes/metadata.
- Attributes of an object can be accessed and modified with *attributes()* function.

**Entering Input**

- The things we type at the R prompt are called "expressions."
  - ex: <- is the assignment operator, assigning values to symbol, as in
  - > msg<-"hello" #creates a 5-element character vector called msg
- Grammar of the language determines whether an expression is complete or not
- # indicates a comment, meaning everything to the right of the # will be ignored by the R engine.

**Evaluation**

- When a complete expression is entered at the prompt, it is "evaluated", and the result returned. May auto-print result.
  - > *print*(x)
  - [1] 5 #indicates x is a vector, and 5 is first element

**Sequences**

- The : operator is used to create integer sequences.
  - > x<-1:20
  - >x
  - [1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
  - [16] 16 17 18 19 20

**Creating Vectors**

- The *c()* function can be used to create vectors; think of c as standing for concatenate
  - > x <- c(0.5,0.6) #numerical vector
  - > x <- c(TRUE, FALSE) #logical vector; could also do c(T, F)
  - > x <- c("a", "b", "c") #character vector
  - > x <- c(1+0i, 2+4i) #complex vector
- Can use *vector()* function to create vector of specified length, initialized with default values
  - > x <- vector("numeric", length=10)
  - > x
  - [1] 0 0 0 0 0 0 0 0 0 0

**Mixing Objects**

- When different objects are mixed in a vector, no error is produced; "coercion" occurs so that all elements are of the same class, a common class sort of like the least common denominator.
- What about the following code:
  - > y <- c(1.7, "a") ##coerces 1.7 into a character
  - > y <- c(T, 2) ##coerces T into a numerical value, i.e., 1
  - > y <- c("a", T) ##coerces T into a character

**Explicit Conversion**

- Objects can be explicitly coerced from one class to another using *as.** functions, if available
  - ex: > x <- 0:6
  - >class (x)
  - [1] "integer"
  - >as.numeric(x)
  - [1] 0 1 2 3 4 5 6
  - >as.logical(x)
  - [1] F T T T T T T #0 becomes false, everything else true
  - >as.character(x)
  - [1] "0" "1" "2" "3" "4" "5" "6"
- Nonsensical coercion results in NA values and a warning
  - >x <- c("a", "b", "c")
  - >as.numeric(x)
  - [1] NA NA NA
  - Warning message:
  - NAs introduced by coercion

**Matrices**

- Matrices are vectors with a dimension attribute, an integer vector (nrow, ncol). Can be created with *matrix()* function, and the dimensions can be obtained with *dim()*:
  - >m <- matrix(nrow = 2, ncol = 3)
  - >m

  - $\begin{array}{cccc} & [,1] & [,2] & [,3] \\ [1,] & NA & NA & NA \\ [2,] & NA & NA & NA \end{array}$
  - >dim(m)
  - [1] 2 3 ##2 rows, 3 columns

- Matrices are constructed column-wise, so entries can be thought of as starting in "upper left" corner, and running down columns. Ex:
  - >m <- matrix(1:6, nrow=2, ncol=3)
  - >m
  - $\begin{array}{cccc} & [,1] & [,2] & [,3] \\ [1,] & 1 & 3 & 5 \\ [2,] & 2 & 4 & 6 \end{array}$

- Matrices can also be created directly from vectors by adding a dimension attribute. Ex:
  - >m <- 1:10
  - >dim(m) <- c(2,5) ##will create a matrix with 2 rows and 5 columns

- Matrices can be created by column-binding (*cbind()*) or row-binding (*rbind()*) vectors:
  - > x <- 1:3
  - > y <- 10:12
  - >rbind(x,y)
  - $\begin{array}{cccc} & [,1] & [,2] & [,3] \\ x & 1 & 2 & 3 \\ y & 10 & 11 & 12 \end{array}$

**Lists**

- Lists are a special type of vector that can contain elements of different classes. Created with *list()*.
  - >x <- list(1, "a", TRUE, 1+4i)
  - >x
  - [[1]]
  - [1] 1
  - [[2]]
  - [1] "a"
  - [[3]]

  - [1] TRUE
  - [[4]]
  - [1] 1+4i

- NB: double brackets around the element numbers differentiate the output from vector output, and each element is a vector.

**Factors**

- Factor are used to represent categorical data. Factors can be ordered (ranked, e.g., professor, assistant professor, etc.) or unordered. One can think of a factor as an integer vector where each integer has a label.
  - Factors are treated specially by modelling functions like *lm()* an *glm()*.
- Using factors with labels is better than using integers because factors are self-describing; having a variable that has values "Male" and "Female" is better than a variable that has values 1 and 2.
- Can be created with *factor()* function.

  -
    ```
    >x<-factor(c("yes", "yes", "no", "yes", "no"))
    > x
    [1] yes yes no yes no
    Levels: no yes
    > table(x)
    x
    no  yes
    2    3
    > unclass(x)
    [1] 2 2 1 2 1
    attr(,"levels")
    [1] "no" "yes"
    ```
  - *table()* returns frequency count of each of "levels"
  - *unclass*() removes levels attribute, returning equivalent integer vector
- The order of the levels can be set using the *levels* argument to *factor()*. This can be important in linear modelling because the first level is used as the baseline level. Unless overridden, baseline level is determined by alphabetical order.

  -
    ```
    > x<- factor(c("yes", "yes", "no", "yes", "no"), levels=c("yes", "no"
    > x
    [1] yes yes no yes no
    Levels: yes no #baseline is yes, because overrode
    ```

**Missing Values**

- Missing values are denoted by NA, or NaN for undefined mathematical operations.
  - *is.na(x)* is used to test if objects in vector x are NA; returns TRUE if object in that position is NA
  - *is.nan(x)* is used to test for NaN
  - NA values have a class also, so there are integer NA, character NA, etc.
  - A NaN value is also NA, but the converse is not true, i.e., NA value is not necessarily also NaN

**Data Frames**

- Data frames are used to store tabular data, and are very important
  - They are represented as a special type of list where every element of the list has to have the same length.
  - Each element of the list can be thought of as a column, and the length of each element of the list is the number of rows. However, each column doesn't have to be the same type.
  - Unlike matrices, data frames can store different classes of objects in each column (just like lists); matrices must have every element be the same class
  - Data frames also have a special attribute called *row.names*, which is useful for annotating data. For example, each row may be a subject in a study, and each row name their subject ID.
  - Data frames are usually created by calling *read.table()* or *read.csv()*
  - Can be converted to a matrix by calling *data.matrix()*–if convert a heterogeneous data frame to a matrix, coercion will take place.
  - Can also be created with *data.frame()*

  -
    ```
    > x <- data.frame(foo=1:4, bar=c(T, T, F, F))
    > x
          foo    bar
    1     1      TRUE  #note that row names default
    2     2      TRUE  #to integers because
    3     3      FALSE #row.names not specified
    4     4      FALSE
    >nrow(x)
    [1] 4
    >ncol(x)
    [1] 2
    ```

**Names**

- R objects can also have names, which is very useful writing readable code and self-describing objects. Access names with *names(x)*

  ```
  – > x <- 1:3
    > names(x)
    NULL
    > names(x) <- c("foo", "bar", "norf")
    > x
    foo    bar    norf
           1      2      3
    > names(x)
    [1] "foo" "bar" "norf"
  ```

- Lists can also have names

  ```
  – > x <- list(a=1, b=2, c=3)
    > x
    $a
    [1] 1

    $b
    [1] 2

    $c
    [1] 3
  ```

- Matrices can have names (*dimnames(m)*).

  ```
  – > m <-matrix(1:4, nrow=2, ncol=2)
    > dimnames(m) <- list(c("a", "b"), c("c", "d")) #first vector is row
    > m
           c      d
    a      1      3
    b      2      4
  ```

**Summary of Data Types**

- atomic classes: numeric, logical, character, integer, complex
- missing values
- vectors, lists
- factors
- data frames, matrices
- names

## 3 Subsetting

**Operators** There are a number of operators that can be used to extract subsets of R objects.

- [ always returns an object of the same class as the original, i.e., if subset a vector, returns a vector, if subset a list, returns a list; can be used to select more than one element (with one exception).
- [[ is used to extract elements of a list or a data frame; it can only be used to extract a single element, and the class of the returned object will not necessarily be a list or data frame
- $ is used to extract elements of a list or data frame by name; semantics are similar to that of [[

**Example** This example shows subsetting with a numeric index, a numeric sequence index, and a logical index

-
  ```
  >x <- c("a", "b", "c", "c", "d", "a")
  >x[1]
  [1] "a"
  >x[2]
  [1] "b"
  >x[1:4]  #extract subset with index sequence
  [1] "a" "b" "c" "c"
  >x[x > "a"] #extract with logical index, according to lexicographical ord
  [1] "b" "c" "c" "d"
  >u <- x > "a" #create logical vector
  >u
  [1] FALSE TRUE TRUE TRUE TRUE FALSE
  >x[u] #subset x with logical vector
  [1] "b" "c" "c" "d"
  ```

**Subsetting a Matrix** Matrices can be subsetted in the usual way with (i,j) type indices (i=row, j=col)

-
  ```
  >x <- matrix(1:6, 2, 3)
  >x[1,2]
  [1] 3
  >x[2,1]
  [1] 2
  ```

Indices can also be missing

-
  ```
  >x[1, ]  #subset row 1
  [1] 1 3 5
  >x[, 2] #subset col 2
  [1] 3 4
  ```

By default, when a single element of a matrix is retrieved, it is returned as a vector of length 1, not a 1x1 matrix. This behavior can be turned off by setting subsetting argument drop=FALSE.

-
  ```
  >x <- matrix(1:6, 2, 3)
  >x[1,2] #by default returns a vector
  [1] 3
  >x[1, 2, drop=FALSE] #drop arg returns a matrix
                      [,1]
  [1,]      3
  ```

- Similarly, subsetting a single column or a single row will give you a vector, not a matrix (by default). Can be turned off with drop=FALSE

  -
    ```
    >x <- matrix(1:6, 2, 3)
    >x[1, ]  #returns a row vector
    [1] 1 3 5
    >x[1, , drop=FALSE] #drop returns 1x3 matrix
              [,1]     [,2]     [,3]
    [1,]      1        3                 5
    ```

**Subsetting Lists**

-
  ```
  >x <- list(foo=1:4, bar=0.6)
  >x[1]  #single bracket, so returns list
  $foo
  [1] 1 2 3 4

  >x[[1]] #double bracket, so returns element, a sequence
  [1] 1 2 3 4

  >x$bar #returns elemented assoc with name bar
  [1] 0.6
  >x[["bar"]] #equiv to $bar
  [1] 0.6
  >x["bar"] #returns list with element bar
  $bar
  [1] 0.6
  ```

- Subsetting a list by name means don't need to remember position

- Extracting multiple elements of a list requires using [

  -
    ```
    >x <- list(foo=1:4, bar=0.6, baz="hello")
    >x[c(1,3)] #returns two element list
    $foo
    [1] 1 2 3 4

    $baz
    [1] "hello"
    ```

- The [[ operator can be used with *computed* indices; $ can only be used with literal names. Useful when name of element is result of some computation.

  -
    ```
    >x <- list(foo=1:4, bar=0.6, baz="hello")
    >name <- "foo"
    >x[[name]] ##computed index for 'foo'
    [1] 1 2 3 4
    >x$name  ##element 'name' doesn't exist
    NULL
    >x$foo
    [1] 1 2 3 4 ##element 'foo' does exist
    ```

**Subsetting Nested Elements of a List** The [[ can take an integer sequence; think of as recursing into lists

-
  ```
  >x <- list(a=list(10,12,14), b=c(3.14, 2.81))
  >x[[c(1,3)]]
  [1] 14   ##third element of first list
  >x[[1]][[3]] #equivalent syntax
  [1] 14   ##third element of first list

  >x[[c(2, 1)]]
  [1] 3.14  ##returns first element of second list
  ```

**Partial Matching** Partial matching of names is allowed with [[ and $. Useful in command line, programs not so much.

-
  ```
  >x <- list(aardvark = 1:5)
  >x$a ## $ looks for name matching 'a'
  [1] 1 2 3 4 5
  >x[["a"]] ##expects exact name
  NULL
  >x[["a"]], exact=FALSE]] ##FALSE exact arg overrides exact matching
  [1] 1 2 3 4 5
  ```

**Removing NA Values** A common task is to remove missing values (NAs), because most realistic data has a lot of missing values. Works for vector, matrix, or data frame. Create logical vector of missing vectors, then subset the inverse of this vector to get the good values.

-
  ```
  >x <- c(1, 2, NA, 4, NA, 5)
  >bad <- is.na(x)  ##true if element is missing
  >x[!bad] ##false if element is missing
  [1] 1 2 4 5
  ```

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

- *complete.cases(x, y, ...)* returns a logical vector specifying which cases are complete, i.e., which observations/rows have no missing values across the entire sequence of vectors x, y, ...
- Can be used on data frames as well

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

## 4 Reading and Writing Data

**Principal Functions for Reading and Writing Data in R**

- *read.table, read.csv* for reading tabular data stored in text files (inverse of *write.table)*
- *readLines* for reading lines of a text file, returning character vector (inverse of *writeLines)*
- *source* for reading in R code files (inverse of *dump*)
- *dget* for reading in R code files (inverse of *dput*) [deparsed files]
- *load* for reading in saved workspaces (inverse of *save)*
- *unserialize* for reading single R objects in binary form (inverse of *serialize)*

## 4.1 Reading Tabular Data Files with read.table

The read.table function is one of the most commonly used functions for reading data.

**Important Arguments**

- `file`: string with the name of a file, or a connection
- `header`: logical indicating if the file has a header line (which obv isn't data)
- `sep`: a string indicating how the columns are separated, e.g., commas
- `colClasses`: a character vector of length ncols indicating the class of each column in the dataset (not required)
- `nrows`: the number of rows in the dataset (not required)
- `comment.char`: a character string indicating the comment character (default is #) (not required)
- `skip`: the number of lines to skip from the beginning (often want to skip header) (not required)
- `stringsAsFactors`: should character variables be encoded as factors? default is TRUE

For small to moderately sized datasets, you can usually call read.table only specifying `file`, e.g., data <- read.table("foo.txt"), and it will automatically:

- skip lines that begin with a #
- figure out how many rows there are, and how much memory to allocate
- figure what type of variable is in each column of the table
- NB: explicitly telling R these things will make it run faster and more efficiently

read.csv is identical to read.table, except that the default separator is a comma instead of a space, and always specifies header to be true

**Reading in Larger Datasets with read.table** With much larger datasets, doing the following make life easier and prevent R from choking:

- Read the help page for read.table, which has many optimization hints
- Make a rough calculation of memory required to store dataset. If dataset is larger than the amount of RAM on computer, can probably stop right there.

- Set comment.char="" if there are no commented lines in your file
- Use the colClasses argument. Specifying this instead of leaving the default can make it run up to twice as fast, because R doesn't have to go through and try to figure out what data type each column is. Have to know the class of each column in data frame. If all columns are numeric for example, can set colClasses="numeric".
  - Quick and dirty way to figure out the classes of each column is to read in, say, 100 rows, then use sapply to loop over each column and use class function

  -
    ```
    >initial <- read.table("table.txt", nrows=100)
    >classes <- sapply(initial, class)
    >tabAll <- read.table("table.txt", colClasses = classes)
    ```
- Set nrows. This doesn't make R run faster, but does help with memory usage, because can calculate memory ahead of time. A mild overestimate is OK. Can use Unix tool wc to calculate number of lines in file.

**Things to know when using R with large datasets**

- How much memory is available?
  - What other applications are in use?
  - Are there any other users logged in?
  - What operating system?
  - Is the OS 32 or 64 bit? 64 bit allows accessing more memory

**Calculating memory requirements**

- Suppose have a data frame with 1,500,000 rows and 120 columns, all of which are numeric. How much memory is required to store this data frame?
  - (1,500,00 x 120) numeric elements x (8 bytes / numeric element) = 1,440,000,000 bytes
    - =1,440,000,000 bytes / ($2^{20}$ bytes/MB)
    - =1,373.29 MB
    - =1.34 GB
  - Slight bit of overhead required, so rule of thumb is to double the amount of memory the object itself requires

## 4.2 Textual Formats

- Two main functions for writing textual data, are `dump` and `dput`, which produce a format different than tabular data, and include metadata like class information. `dumping` and `dputing` are useful because the resulting textual format is editable (though you shouldn't, because then no longer reproducible), and in the case of corruption, potentially recoverable

**Advantages of Textual Formats**

- Unlike writing out a table or csv file, `dump` and `dput` preserve the metadata (sacrificing readability), so that another user doesn't have to specify it all over again
- Textual formats can work much better with version control programs, which can only meaningfully track changes in text files.
- Textual formats can be longer-lived: it is easier to fix corruption in the file
- Textual formats adhere to "Unix philosophy" (store all kinds of data in text)

**Drawbacks of Textual Formats**

- The format is not very space-efficient, so often must be compressed

**dput-ting R Objects** Another way to pass data around is by deparsing the R object with `dput` and reading it back in with `dget.` dput takes an arbitrary R object (most kinds except some exotic ones) and creates some R code that will reconstruct the object in R.

```
>y <- data.frame(a = 1, b = "a")
>dput(y)
structure(list(a=1,
        b = structure(1L, .Label="a", class = "factor")),
        .Names = c("a", "b"), row.names = C(NA, -1L),
        class = "data.frame")
>dput(y, file = "y.R") ##file arg saves output to R script
>new.y <- dget("y.R") #reconstruct object
>new.y
        a       b
1       1       a
```

**Dumping R Objects** `dump` is similar to `dput`, but `dump` allows deparsing *multiple* objects, and reading them back in with `source`

```
>x <- "foo"
>y <- data.frame(a = 1, b = "a")
>dump(c("x", "y"), file = "data.R")
>rm(x, y) ##remove x,y
>source("data.R") ##reconstruct x and y
>y
        a       b
1       1       a
>x
[1] "foo"
```

**Interfaces to the Outside World** Data are read in using *connection* interfaces. Abstracts out the method of connecting. Connections can be made to files (most common) or other more exotic things:

- `file` opens a connection to a file
- `gzfile` opens a connection to a file compressed with gzip
- `bzfile` opens a connection to a file compressed with bzip2
- `url` opens a connection to a webpage
- This is usually done behind-the-scenes.

```
>str(file)
function (description = "", open="", blocking = TRUE, encoding = getOption("e
```

- `description` is the name of the file
- `open` is a code indicating
  - "r" read only
  - "w" writing (and initializing a new file)
  - "a" appending
  - "rb", "wb", "ab" doing the above in binary mode (Windows)

**Connections**

- In general, connections are powerful tools that let you navigate files or other external objects. In practice, don't often need to deal with the connection interface directly. For example,

  ```
  con <- file("foo.txt", "r")
  data <- read.csv(con)
  close(con)
  ```

  is equivalent to much shorter

  ```
  data <- read.csv("foo.txt")
  ```

- Connections can be useful to read parts of a file with `readLines`:

  ```
  >con <- gzfile("words.gz")
  >x <- readLines(con, 10)
  >x
  [1] "1080"      "10-point"     "10th"   "11-point"
  [5] "12-point" "16-point"     "18-point"      "1st"
  [9]     "2"     "20-point"
  ```

  - Inversely, `writeLines` takes a character vector and writes each element one line at a time to a text file.

- Connections and `readLines` can be useful for reading in lines of webpages

  ```
  >con <- url("http://jhsph.edu", "r")
  >x <- readLines(con)
  >head(x)
  [1] "<!DOCTYPE HTML PUBLIC ..... >"
  [2] ""
  [3] "<html>"
  [4] "<head>"
  [5] "\t<meta http..."
  ```

## 5 SWIRL

http://swirlstats.com is a resource that offers interactive lessons
Completing them offers extra credit