R Language Fundamentals
Data Types and Basic Manipulation

Steven Buechler

Department of Mathematics
276B Hurley Hall; 1-6233

Fall, 2007
Outline

Where did R come from?

Primitive Data Types in R
  Overview
  Atomic Vectors
  Subsetting Vectors
  Higher-order data types (slightly)
Where did R come from?

Programming with Data Began with S

- The S language has been developed since the late 1970s by John Chambers and colleagues at Bell Labs as a language for programming with data.
- The language combines ideas from a variety sources (awk, lisp, APL, e.g.) and provides an environment for quantitative computations and visualization.
- Provides an explicit and consistent structure for manipulating, analyzing statistically, and visualizing data.
Programming with Data Began with S

- The S language has been developed since the late 1970s by John Chambers and colleagues at Bell Labs as a language for programming with data.
- The language combines ideas from a variety sources (awk, lisp, APL, e.g.) and provides an environment for quantitative computations and visualization.
- Provides an explicit and consistent structure for manipulating, analyzing statistically, and visualizing data.
S Becomes Software

- S-Plus is a commercialization of the Bell Labs framework. It is “S” plus “graphics”.
- R is an independent open source implementation originally developed by Ross Ihaka and Robert Gentleman at the University of Auckland in the mid-1990s. R comes before S.
- R is currently distributed under the GNU open software license and developed by the user community.
S Becomes Software

- S-Plus is a commercialization of the Bell Labs framework. It is “S” plus “graphics”.
- R is an independent open source implementation originally developed by Ross Ihaka and Robert Gentleman at the University of Auckland in the mid-1990s. R comes before S.
- R is currently distributed under the GNU open software license and developed by the user community.
Applications of R normally use a **package**; i.e., a library of special functions designed for a specific problem.

Hundreds of packages are available, mostly written by users.

A user normally only loads a handful of packages for a particular analysis (e.g., `library(affy)`).

Standards determine how a package is structured, works well with other packages and creates new data types in an easily used manner.

Standardization makes it easy for users to learn new packages.
Bioconductor is a Set of Packages

- Bioconductor is a set R packages particularly designed for biological data analysis.
- Bioconductor Project sets standards used across packages, identifies needed packages and organizes development cycles and responsible parties.
- Bioconductor project is headed by Robert Gentleman and located at Fred Hutchinson in Seattle.
Where did R come from?

**Bioconductor** is a Set of Packages

- **Bioconductor** is a set of R packages particularly designed for biological data analysis.

- **Bioconductor Project** sets standards used across packages, identifies needed packages and organizes development cycles and responsible parties.

- Bioconductor project is headed by Robert Gentleman and located at Fred Hutchinson in Seattle.
Bioconductor is a Set of Packages

- **Bioconductor** is a set R packages particularly designed for biological data analysis.
- **Bioconductor Project** sets standards used across packages, identifies needed packages and organizes development cycles and responsible parties.
- Bioconductor project is headed by Robert Gentleman and located at Fred Hutchinson in Seattle.
Outline

Where did R come from?

Primitive Data Types in R

Overview
Atomic Vectors
Subsetting Vectors
Higher-order data types (slightly)
Fundamental Data Objects

- **vector** - a sequence of numbers or characters, or higher-dimensional arrays like matrices
- **list** - a collection of objects that may themselves be complicated
- **factor** - a sequence assigning a category to each index
- **data.frame** - a table-like structure (experimental results often collected in this form)
- **environment** - hash table. A collection of key-value pairs

Classes of objects, like expression data, are built from these. Most commands in R involve applying a function to an object.
A Variable is “Typed” by What it Contains

- Unlike C variables do not need to be declared and typed. Assigning a sequence of numbers to `x` forces `x` to be a numeric vector.
- Given `x`, executing `class(x)` reports the class. This indicates which functions can be used on `x`. 
Outline

Where did R come from?

Primitive Data Types in R

Overview
Atomic Vectors
Subsetting Vectors
Higher-order data types (slightly)
Atomic Data Elements

- In R the “base” type is a vector, not a scalar.
- A vector is an indexed set of values that are all of the same type. The type of the entries determines the class of the vector. The possible vectors are:
  - integer
  - numeric
  - character
  - complex
  - logical
- integer is a subclass of numeric
- Cannot combine vectors of different modes
Creating Vectors
and Learning R Syntax

Assignment of a value to a variable is done with <- (two symbols, no space).

> v <- 1
> v
[1] 1

> v <- c(1, 2, 3)
> v
[1] 1 2 3

> s <- "a string"
> class(s)
[1] "character"
Creating Vectors
and accessing attributes

Vectors can only contain entries of the same type: numeric or character; you can’t mix them. Note that characters should be surrounded by “ “. The most basic way to create a vector is with $c(x_1, \ldots, x_n)$, and it works for characters and numbers alike.

```r
> x <- c("a", "b", "c")
> length(x)
[1] 3
```
Vector Names

Entries in a vector can and normally should be “named”. It is a way of associating a numeric reading with a sample id, for example.

```r
> v <- c(s1 = 0.3, s2 = 0.1, s3 = 1.1)
> v
  s1  s2  s3
0.3 0.1 1.1
> sort(v)
  s2  s1  s3
0.1 0.3 1.1
> names(v)
[1] "s1" "s2" "s3"
> v2 <- c(0.3, 0.1, 1.1)
> names(v2) <- c("s1", "s2", "s3")
```
Vector Arithmetic

Basic operations on numeric vectors

Numeric vectors can be used in arithmetic expressions, in which case the operations are performed element by element to produce another vector.

```r
> x <- rnorm(4)
> y <- rnorm(4)
> x
[1] -0.6632  0.3255  0.7577 -1.0309
> x + 1
[1]  0.3368  1.3255  1.7577 -0.0309
> v <- 2 * x + y + 1
> v
[1] -0.1536  1.1608  2.4672 -2.0510
```

The elementary arithmetic operations are the usual `+`, `-`, `*`, `/`, `^`. See also `log`, `exp`, `log2`, `sqrt` etc., again applied to each entry.
More Vector Arithmetic

Statistical operations on numeric vectors

In studying data you will make frequent use of `sum`, which gives the sum of the entries, `max`, `min`, `mean`, and `var(x)`

```
> var(x)
[1] 0.6965
```

which is the same thing as

```
> sum((x - mean(x))^2)/(length(x) - 1)
[1] 0.6965
```

A useful function for quickly getting properties of a vector:

```
> summary(y)

       Min.    1st Qu.     Median       Mean    3rd Qu.       Max.  
-0.98900 -0.61500  -0.26900    -0.33900  0.00708     0.17300
```
Generating regular sequences

- `c` - concatenate
- `seq`, `:`, and `rep`
- `vector`, `numeric`, `character`, etc.

`seq(1,30)` is the same thing as `c(1, 2, 3, ..., 29, 30)`; and this is the same as `1:30`. Functions in R may have mutliple parameters that are set as arguments to the function. `seq` is an example.

```r
> x1 <- seq(-1, 0, by = 0.1)
> x1
[1] -1.0 -0.9 -0.8 -0.7 -0.6 -0.5 -0.4 -0.3 -0.2
[10] -0.1 0.0
> rep(x1, times = 2)
[1] -1.0 -0.9 -0.8 -0.7 -0.6 -0.5 -0.4 -0.3 -0.2
[10] -0.1 0.0 -1.0 -0.9 -0.8 -0.7 -0.6 -0.5 -0.4
[19] -0.3 -0.2 -0.1 0.0
```
Generating regular sequences

vector, etc.

a <- character(n) creates a character vector a of length n, with each entry "". integer(n) and numeric(n) create integer and numeric vectors of length n with entries 0. The first command is shorthand for

> a <- vector(mode = "character", length = 10)

vector has greater applicability than just creating these common vectors.
Logical Vectors

Working with data often involves comparing numbers. Comparisons in R output logical values TRUE, FALSE or NA, for “not available”. Just as with arithmetic operations, logical comparisons with a vector are applied to each entry and output as a vector of truth values; i.e. a logical vector.

```r
> v <- seq(-3, 3)
> tf <- v > 0
> tf

[1] FALSE FALSE FALSE FALSE TRUE TRUE TRUE
```
Logical Comparisons

Logical vectors are largely used to extract entries from a dataset satisfying certain conditions. Illustrated soon. The logical operators are: `<`, `<=`, `>`, `>=`, `==`, for exact equality and `!=` for inequality.

If \( c_1 \) and \( c_2 \) are logical expressions, then \( c_1 \& c_2 \) is their intersection ("and"), \( c_1 \mid c_2 \) is their union ("or"), and \( \neg c_1 \) is the negation of \( c_1 \).
Missing Values

One smart feature of R is that it allows for missing values in vectors and datasets; it denotes them as NA. You don’t have to coerce missing values to, say 0, in order to have a meaningful vector. Many functions, like cor(), have options for handling missing values without just exiting. How to find NA values in a vector?

- `> x <- c(1:3, NA)`
  - `> x == NA`
    - `[1] NA NA NA NA`
  - `> is.na(x)`
    - `[1] FALSE FALSE FALSE FALSE TRUE`
Missing Values

One smart feature of R is that it allows for missing values in vectors and datasets; it denotes them as NA. You don’t have to coerce missing values to, say 0, in order to have a meaningful vector. Many functions, like `cor()`, have options for handling missing values without just exiting. How to find NA values in a vector?

- \( > x \leftarrow c(1:3, \text{NA}) \)
  \( > x == \text{NA} \)
  
  \[1\] NA NA NA NA

- \( > \text{is.na}(x) \)
  
  \[1\] FALSE FALSE FALSE FALSE TRUE
Outline

Where did R come from?

**Primitive Data Types in R**
- Overview
- Atomic Vectors
- Subsetting Vectors
- Higher-order data types (slightly)
Extracting Subsequences of a Vector

Getting elements of a vector with desired properties is extremely common, so there are robust tools for doing it. An element of a vector \( v \) is assigned an index by its position in the sequence, starting with 1. The basic function for subsetting is \([ \]\). \( v[1] \) is the first element, \( v[length(v)] \) is the last. The subsetting function takes input in many forms.
Subsetting with Positive Integral Sequences

> v <- c("a", "b", "c", "d", "e")

> J <- c(1, 3, 5)
> v[J]
[1] "a" "c" "e"

> v[1:3]
[1] "a" "b" "c"

> v[2:length(v)]
[1] "b" "c" "d" "e"
Subsetting with Positive Integral Sequences

```r
> v <- c("a", "b", "c", "d", "e")

• > J <- c(1, 3, 5)
  > v[J]
  [1] "a" "c" "e"

• > v[1:3]
  [1] "a" "b" "c"

• > v[2:length(v)]
  [1] "b" "c" "d" "e"
```
Subsetting with Positive Integral Sequences

> v <- c("a", "b", "c", "d", "e")

- > J <- c(1, 3, 5)
  > v[J]
  [1] "a" "c" "e"
- > v[1:3]
  [1] "a" "b" "c"
- > v[2:length(v)]
  [1] "b" "c" "d" "e"
Subsetting with Negated Integral Sequences

This is a tool for removing elements or subsequences from a vector.

- \( v \)
  
  \[
  [1] \text{"a" "b" "c" "d" "e"}
  \]

- \( J \)
  
  \[
  [1] 1 3 5
  \]

- \( v[-J] \)
  
  \[
  [1] \text{"b" "d"}
  \]

- \( v[-1] \)
  
  \[
  [1] \text{"b" "c" "d" "e"}
  \]

- \( v[-\text{length}(v)] \)
  
  \[
  [1] \text{"a" "b" "c" "d"}
  \]
Subsetting with Negated Integral Sequences

This is a tool for removing elements or subsequences from a vector.

- \( v \)
  
  \[
  \text{[1] "a" "b" "c" "d" "e"}
  \]

- \( J \)
  
  \[
  \text{[1] 1 3 5}
  \]

- \( v[-J] \)
  
  \[
  \text{[1] "b" "d"}
  \]

- \( v[-1] \)
  
  \[
  \text{[1] "b" "c" "d" "e"}
  \]

- \( v[-\text{length}(v)] \)
  
  \[
  \text{[1] "a" "b" "c" "d"}
  \]
Subsetting with Logical Vector

Important!

Given a vector \( x \) and a logical vector \( L \) of the same length as \( x \), \( x[L] \) is the vector of entries in \( x \) matching a TRUE in \( L \).

- \( L \) <- \( c(\text{TRUE}, \text{FALSE}, \text{TRUE}, \text{FALSE}, \text{TRUE}) \)
  
  > \( v[L] \)
  
  [1] "a" "c" "e"

- \( x \) <- \( \text{seq}(-3, 3) \)
  
  > \( x \geq 0 \)
  
  [1] FALSE FALSE FALSE TRUE TRUE TRUE TRUE

  > \( x[x \geq 0] \)
  
  [1] 0 1 2 3
Subsetting with Logical Vector

Important!

Given a vector \( x \) and a logical vector \( L \) of the same length as \( x \), \( x[L] \) is the vector of entries in \( x \) matching a TRUE in \( L \).

- \( > \ L \leftarrow c(\text{TRUE}, \text{FALSE}, \text{TRUE}, \text{FALSE}, \text{TRUE}) \)
  \( > \ v[L] \)
  
  \[ 1\] "a" "c" "e"

- \( > \ x \leftarrow \text{seq}(-3, 3) \)
  \( > \ x \geq 0 \)

  \[ 1\] FALSE FALSE FALSE FALSE FALSE TRUE TRUE TRUE TRUE TRUE TRUE

  \( > \ x[x \geq 0] \)

  \[ 1\] 0 1 2 3
Subsetting with Logical Vector

More examples

```r
> names(x) <- paste("N", 1:length(x), sep = "")
> names(x)[x < 0]
[1] "N1" "N2" "N3"
> y <- c(x, NA, NA)
> z <- y[!is.na(y)]
> z

N1  N2  N3  N4  N5  N6  N7
-3  -2  -1   0   1   2   3
```
Subsetting by Names

If \( x \) is a vector with names and \( A \) is a subsequence of \( \text{names}(x) \), then \( x[A] \) is the corresponding subsequence of \( x \).

\[
> x \\
N1 N2 N3 N4 N5 N6 N7 \\
-3 -2 -1  0  1  2  3 \\
> x[c("N1", "N3")]
N1 N3 \\
-3 -1
\]
Assignment to a Subset

A subset expression can be on the receiving end of an assignment, in which case the assignment only applies the subset and leaves the rest of the vector alone.

> z <- 1:4
> z[1] <- 0
> z
[1] 0 2 3 4
> z[z <= 2] <- -1
> z
[1] -1 -1 3 4
> w <- c(1:3, NA, NA)
> w[is.na(w)] <- 0
> w
[1] 1 2 3 0 0
Outline

Where did R come from?

**Primitive Data Types in R**
- Overview
- Atomic Vectors
- Subsetting Vectors
- Higher-order data types (slightly)
Factors Represent Categorical Data

Just the basics

Typically in an experiment samples are classified into one of a set group of categories. In R such results are stored in a factor. A factor is a character vector augmented with information about the possible categories, called the levels of the factor.

```r
> d1 <- c("M", "F", "M", "F", "F", "F")
> d2 <- factor(d1)
> d2

[1] M F M F F F
Levels: F M
```
Factors (Continued)
Still just the basics

> table(d2)

d2
F M
4 2

> ht <- c(73, 68, 70, 69, 62, 64)
> htmeans <- tapply(ht, d2, mean)

The data contained in a factor can be coded in a character vector, but there are many additional functions that can apply to a factor. Factors are used in ANOVA.
Lists

An ordered collection of objects

Remember that a vector can only contain numbers, characters or logical values. Frequently, though, we want to create collections of vectors or other data objects of mixed type. In R this is done with a list. The objects in a list are known as its components. Lists are often created quite explicitly:

```
> Lst <- list(name = "Joe", height = "182",
+            no.children = 3, child.ages = c(5,
+            7, 10))
```

Components are always numbered and can be referenced as such. `Lst[[1]]` is the first component (namely "Joe"); etc. to

```
> Lst[[4]]
```

```
[1] 5 7 10
```

Since the last component is a vector you can extract the first entry of it as `Lst[[4]][1]`. 
List Length and Components

For Lst a list, `length(Lst)` is the number of components; `names(Lst)` is the character vector of component names.

Often the ordering of components is artificial. We want simple ways of getting the value of a component using the name. There are two ways:

```r
> Lst[["height"]]
[1] "182"
```

More commonly:

```r
> Lst$height
[1] "182"
```

```r
> Lst$name
[1] "Joe"
```
List Subsetting

Versus component extraction

For a list \( LL \) with \( n \) components and \( s \) a subsequence of \( 1:n \), \( LL[s] \) denotes the sublist with components corresponding to the indices in \( s \).

```r
> s <- 1:2
> L1 <- Lst[s]
> L1

$name
[1] "Joe"

 height
[1] "182"
```

**NOTE:** \( LL[[1]] \) is different from \( LL[1] \). \( LL[[1]] \) is the value of the first component; \( LL[1] \) is the list with one component whose value is \( LL[[1]] \).
List Subsetting Example
Versus component extraction

> Lst[[1]]
[1] "Joe"
> class(Lst[[1]])
[1] "character"
> Lst[1]
$name
[1] "Joe"
> class(Lst[1])
[1] "list"

Further note: Lst[[m]] only makes sense when m is a single integer. Lst[[1:2]] produces an error.
Lists can be concatenated like vectors.