Data Structures and Data Types

Lecture 2

Nicholas Christian

BIOST 2094 Spring 2011

Outline

1. Data Structures
   - Vectors
   - Matrices
   - Arrays
   - Lists
   - Dataframes
2. Data Types
   - Numeric
   - Logical
   - Character
   - Factor
   - Dates
   - Missing Data

Vectors

- A vector is an ordered collection of objects of the same type
- The function `c(...)` concatenates its arguments to form a vector
- To create a patterned vector
  - `:` Sequence of integers
  - `seq()` General sequence
  - `rep()` Vector of replicated elements

```
> v1 <- c(2.5, 4, 7.3, 0.1)
> v1
[1] 2.5 4.0 7.3 0.1

> v2 <- c("A", "B", "C", "D")
> v2
[1] "A" "B" "C" "D"

> v3 <- -3:3
> v3
[1] -3 -2 -1  0  1  2  3
```

```
> seq(0, 2, by=0.5)
[1] 0.0 0.5 1.0 1.5 2.0

> seq(0, 2, len=6)
[1] 0.0 0.4 0.8 1.2 1.6 2.0

> rep(1:5, each=2)
[1] 1 1 2 2 3 3 4 4 5 5

> rep(1:5, times=2)
[1] 1 2 3 4 5 1 2 3 4 5
```

Reference Elements of a Vector

- Use [ ] with a vector/scalar of positions to reference elements of a vector
- Include a minus sign before the vector/scalar to remove elements

```
> x <- c(4, 7, 2, 10, 1, 0)
> x[4]
[1] 10
> x[1:3]
[1] 4 7 2
> x[c(2,5,6)]
[1] 7 1 0
> x[-3]
[1]  4  7 10  1  0
> x[-c(4,5)]
[1] 4 7 2 0
> x[x>4]
[1]  7 10
> x[3] <- 99
> x
[1]  4  7 99 10  1  0
```

which() and match()

- Additional functions that will return the indices of a vector
  - `which()` Indices of a logical vector where the condition is `TRUE`
  - `which.max()` Location of the (first) maximum element of a numeric vector
  - `which.min()` Location of the (first) minimum element of a numeric vector
  - `match()` First position of an element in a vector

```
> x <- c(4, 7, 2, 10, 1, 0)
> x>=4
[1] TRUE TRUE FALSE TRUE FALSE FALSE
> which(x>=4)
[1] 1 2 4
> which.max(x)
[1] 4
> x[which.max(x)]
[1] 10
> max(x)
[1] 10
```

```
> y <- rep(1:5, times=5:1)
> y
[1] 1 1 1 1 1 2 2 2 2 3 3 3 4 4 5
> match(1:5, y)
[1]  1  6 10 13 15
> match(unique(y), y)
[1]  1  6 10 13 15
```

Vector Operations

- When vectors are used in math expressions the operations are performed element by element

```
> x <- c(4,7,2,10,1,0)
> y <- x^2 + 1
> y
[1]  17  50   5 101   2   1
> x*y
[1]   68  350   10 1010    2    0
```

Useful Vector Functions

`sum(x)`	`prod(x)`	Sum/product of the elements of `x`
`cumsum(x)`	`cumprod(x)`	Cumulative sum/product of the elements of `x`
`min(x)`	`max(x)`	Minimum/Maximum element of `x`
`mean(x)`	`median(x)`	Mean/median of `x`
`var(x)`	`sd(x)`	Variance/standard deviation of `x`
`cov(x,y)`	`cor(x,y)`	Covariance/correlation of `x` and `y`
`range(x)`		Range of `x`
`quantile(x)`		Quantiles of `x` for the given probabilities
`fivenum(x)`		Five number summary of `x`
`length(x)`		Number of elements in `x`
`unique(x)`		Unique elements of `x`
`rev(x)`		Reverse the elements of `x`
`sort(x)`		Sort the elements of `x`
`which()`		Indices of `TRUE`s in a logical vector
`which.max(x)`	`which.min(x)`	Index of the max/min element of `x`
`match()`		First position of an element in a vector
`union(x, y)`		Union of `x` and `y`
`intersect(x, y)`		Intersection of `x` and `y`
`setdiff(x, y)`		Elements of `x` that are not in `y`
`setequal(x, y)`		Do `x` and `y` contain the same elements?

Matrices

- A matrix is just a two-dimensional generalization of a vector
- To create a matrix,

```
matrix(data=NA, nrow=1, ncol=1, byrow = FALSE, dimnames = NULL)
```

`data` a vector that gives data to fill the matrix; if `data` does not have enough elements to fill the matrix, then the elements are recycled.

`nrow` desired number of rows

`ncol` desired number of columns

`byrow` if `FALSE` (default) matrix is filled by columns, otherwise by rows

`dimnames` (optional) list of length 2 giving the row and column names respectively, list names will be used as names for the dimensions

```
> x <- matrix(c(5,0,6,1,3,5,9,5,7,1,5,3), nrow=3, ncol=4, byrow=TRUE,
+             dimnames=list(rows=c("r.1", "r.2", "r.3"),
+                           cols=c("c.1", "c.2", "c.3", "c.4")))
> x
     cols
rows  c.1 c.2 c.3 c.4
  r.1   5   0   6   1
  r.2   3   5   9   5
  r.3   7   1   5   3
```

Reference Elements of a Matrix

- Reference matrix elements using the [ ] just like with vectors, but now with 2-dimensions

```
> x <- matrix(c(5,0,6,1,3,5,9,5,7,1,5,3), nrow=3, ncol=4, byrow=TRUE)
> x
     [,1] [,2] [,3] [,4]
[1,]    5    0    6    1
[2,]    3    5    9    5
[3,]    7    1    5    3
> x[2,3]                          # Row 2, Column 3
[1] 9
> x[1,]                           # Row 1
[1] 5 0 6 1
> x[,2]                           # Column 2
[1] 0 5 1
> x[c(1,3),]                      # Rows 1 and 3, all Columns
     [,1] [,2] [,3] [,4]
[1,]    5    0    6    1
[2,]    7    1    5    3
```

Reference Elements of a Matrix

- We can also reference parts of a matrix by using the row or column names
- Sometimes it is better to reference a row/column by its name rather than by the numeric index. For example, if a program adds or permutes the columns of a matrix then the numeric index of the columns may change. As a result you might reference the wrong column of the new matrix if you use the old index number. However the name of each column will not change.
- Reference matrix elements using the [ ] but now use the column or row name, with quotations, inplace of the index number
- You don't have to specify the names when you create a matrix. To get or set the column, row, or both dimension names of `A`:

```
colnames(A)
rownames(A)
dimnames(A)
```

- Can also name the elements of a vector, `c("name.1"=1, "name.2"=2)`. Use the function `names()` to get or set the names of vector elements.

Reference Elements of a Matrix

```
> N <- matrix(c(5,8,3,0,4,1), nrow=2, ncol=3, byrow=TRUE)
> colnames(N) <- c("c.1", "c.2", "c.3")
> N
     c.1 c.2 c.3
[1,]   5   8   3
[2,]   0   4   1
> N[,"c.2"]                    # Column named "c.2"
[1] 8 4
> colnames(N)
[1] "c.1" "c.2" "c.3"
> M  <- diag(2)
> (MN <- cbind(M, N))          # Placing the expression in parentheses
         c.1 c.2 c.3           # will print the result
[1,] 1 0   5   8   3
[2,] 0 1   0   4   1
> MN[,2]                       # Column 2
[1] 0 1
> MN[,"c.2"]                   # Column named "c.2"
[1] 8 4
```

Matrix Operations

- When matrices are used in math expressions the operations are performed element by element.
- For matrix multiplication use the %*% operator
- If a vector is used in matrix multiplication, it will be coerced to either a row or column matrix to make the arguments conformable. Using %*% on two vectors will return the inner product (%o% for outer product) *as a matrix* and not a scalar. Use either `c()` or `as.vector()` to convert to a scalar.

```
> A <- matrix(1:4, nrow=2)
> B <- matrix(1, nrow=2, ncol=2)
> A*B
     [,1] [,2]
[1,]    1    3
[2,]    2    4
> A%*%B
     [,1] [,2]
[1,]    4    4
[2,]    6    6
```

```
> y <- 1:3
> y%*%y
     [,1]
[1,]   14
> A/(y%*%y)
Error in A/(y%*%y):non-conformable arrays
> A/c(y%*%y)
           [,1]      [,2]
[1,] 0.07142857 0.2142857
[2,] 0.14285714 0.2857143
```

Useful Matrix Functions

`t(A)`	Transpose of `A`
`det(A)`	Determinate of `A`
`solve(A, b)`	Solves the equation `Ax=b` for `x`
`solve(A)`	Matrix inverse of `A`
`MASS::ginv(A)`	Generalized inverse of `A` (MASS package)
`eigen(A)`	Eigenvalues and eigenvectors of `A`
`chol(A)`	Choleski factorization of `A`
`diag(n)`	Create a `n`×`n` identity matrix
`diag(A)`	Returns the diagonal elements of a matrix `A`
`diag(x)`	Create a diagonal matrix from a vector `x`
`lower.tri(A),upper.tri(A)`	Matrix of logicals indicating lower/upper triangular matrix
`apply()`	Apply a function to the margins of a matrix
`rbind(...)`	Combines arguments by rows
`cbind(...)`	Combines arguments by columns and
`dim(A)`	Dimensions of `A`
`nrow(A), ncol(A)`	Number of rows/columns of `A`
`colnames(A), rownames(A)`	Get or set the column/row names of `A`
`dimnames(A)`	Get or set the dimension names of `A`

apply()

- The `apply()` function is used for applying functions to the margins of a matrix, array, or dataframes.

`apply(X, MARGIN, FUN, ...)`

`X`	A matrix, array or dataframe
`MARGIN`	Vector of subscripts indicating which margins to apply the function to 1=rows, 2=columns, c(1,2)=rows and columns
`FUN`	Function to be applied
`...`	Optional arguments for FUN

- You can also use your own function (more on this later)

```
> (x <- matrix(1:12, nrow=3, ncol=4))
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
> apply(x, 1, sum)      # Row totals
[1] 22 26 30
> apply(x, 2, mean)     # Column means
[1]  2  5  8 11
```

Example - Simulating Survival Data

- The `apply()` function is great for simulating survival data. Suppose we want to simulate $n = 10$ observations where, the event times T follow an exponential distribution with mean $\lambda = 0.25$ and the censoring times C are distributed uniformly from 0 to 1. Then the observed data is, $X = \min(T, C)$ and $\delta = I(T < C)$.

```
# Sample size
n = 10

# Generate event and censor times (look at the documentation to see
# how R parameterizes the exponential distribution)
event  <- rexp(n, 4)
censor <- runif(n)

# Select the minimum time and create an indicator variable
time  <- apply(cbind(censor, event), 1, min)
index <- apply(cbind(censor, event), 1, which.min)-1

cbind(event, censor, time, index)          # Verify
data <- cbind(time, index)                 # Simulated dataset
```

Arrays

- An array is a multi-dimensional generalization of a vector
- To create an array,

```
array(data = NA, dim = length(data), dimnames = NULL)
```

`data` A vector that gives data to fill the array; if `data` does not have enough elements to fill the matrix, then the elements are recycled.

`dim` Dimension of the array, a vector of length one or more giving the maximum indices in each dimension

`dimnames` Name of the dimensions, list with one component for each dimension, either NULL or a character vector of the length given by dim for that dimension. The list can be named, and the list names will be used as names for the dimensions.

- Values are entered by columns
- Like with vectors and matrices, when arrays are used in math expressions the operations are performed element by element.
- Also like vectors and matrices, the elements of an array must all be of the same type (numeric, character, logical, etc.)

Arrays

- Sample $2 \times 3 \times 2$ array,

```
> w <- array(1:12, dim=c(2,3,2),
             dimnames=list(c("A","B"), c("X","Y","Z"), c("N","M")))
> w
, , N

  X Y Z
A 1 3 5
B 2 4 6

, , M

  X  Y  Z
A 7  9 11
B 8 10 12
```

Reference Elements of an Array

- Reference array elements using the [ ] just like with vectors and matrices, but now with more dimensions

```
> w <- array(1:12, dim=c(2,3,2),
             dimnames=list(c("A","B"), c("X","Y","Z"), c("N","M")))
> w[2,3,1]      # Row 2, Column 3, Matrix 1
[1] 6
> w[,"Y",]      # Column named "Y"
  N  M
A 3  9
B 4 10
> w[1,,]        # Row 1
  N  M
X 1  7
Y 3  9
Z 5 11
> w[1:2,,"M"]  # Rows 1 and 2, Matrix "M"
  X  Y  Z
A 7  9 11
B 8 10 12
```

Useful Array Functions

`apply()`	Apply a function to the margins of an array
`aperm()`	Transpose an array by permuting its dimensions
`dim(x)`	Dimensions of `x`
`dimnames(x)`	Get or set the dimension names of `x`

apply()

- We can use the `apply()` function for more then one dimension
- For a 3-dimensional array there are now three margins to apply the function to: 1=rows, 2=columns, and 3=matrices.

```
 # Column sums
> apply(w, 2, sum)
 X  Y  Z
18 26 34

# Row and matrix sums
> apply(w, c(1,3), sum)
   N  M
A  9 27
B 12 30
```

Lists

- A list is a general form of a vector, where the elements don't need to be of the same type or dimension.
- The function `list(...)` creates a list of the arguments
- Arguments have the form *name=value*. Arguments can be specified with and without names.

```
> x <- list(num=c(1,2,3), "Nick", identity=diag(2))
> x
$num
[1] 1 2 3

[[2]]
[1] "Nick"

$identity
     [,1] [,2]
[1,]    1    0
[2,]    0    1
```

Reference Elements of a List

- Elements of a list can be referenced using [ ] as well as [[ ]] or $.

```
> x <- list(num=c(1,2,3), "Nick", identity=diag(2))

> x[[2]]                    # Second element of x
[1] "Nick"
> x[["num"]]                # Element named "num"
[1] 1 2 3
> x$identity                # Element named "identity"
     [,1] [,2]
[1,]    1    0
[2,]    0    1
> x[[3]][1,]                # First row of the third element
[1] 1 0
> x[1:2]                    # Create a sublist of the first two elements
$num
[1] 1 2 3

[[2]]
[1] "Nick"
```

Useful List Functions

`lapply()`	Apply a function to each element of a list, returns a list
`sapply()`	Same as `lapply()`, but returns a vector or matrix by default
`vapply()`	Similiar to `sapply()`, but has a pre-specified type of return value
`replicate()`	Repeated evaluation of an expression, useful for replicating lists
`unlist(x)`	Produce a vector of all the components that occur in `x`
`length(x)`	Number of objects in `x`
`names(x)`	Names of the objects in `x`

Example - `lapply()`, `sapply()`, `vapply()`

- First generate a list `L` of seven different vectors. Then calculate the five number summary of each vector in `L` using `lapply()`, `sapply()` and `vapply()`.

```
# List of seven different vectors
L <- lapply(3:9, seq)

# Calculate the five number summary for each vector in L
lapply(L, fivenum)
sapply(L, fivenum)
vapply(L, fivenum, c(Min.=0, "1st Quart"=0, Median=0, "3rd Qu"=0, Max.=0))
```

- Since 3:9 is not a list, `R` calls `as.list(3:9)` which coerces the vector 3:9 to a list of length 7 where each number is an element of `L`. Also note that `seq(n)` is the same as `1:n`.

Dataframes

- R refers to datasets as dataframes
- A dataframe is a matrix-like structure, where the columns can be of different types. You can also think of a dataframe as a list. Each column is an element of the list and each element has the same length.
- A dataframe is the fundamental data structure used by R 's statistical modeling functions
- Lecture 8 will be completely devoted to the managment and use of dataframes

	Population	Income	Illiteracy	Life.Exp	state.region
Alabama	3615	3624	2.1	69.05	South
Alaska	365	6315	1.5	69.31	West
Arizona	2212	4530	1.8	70.55	West
Arkansas	2110	3378	1.9	70.66	South
California	21198	5114	1.1	71.71	West
Colorado	2541	4884	0.7	72.06	West
Connecticut	3100	5348	1.1	72.48	Northeast

Numeric

- Technically, numeric data in `R` can be either `double` or `integer`, but in practice numeric data is almost always `double` (type `double` refers to real numbers). See `?integer` and `?double`.
- `.Machine` outputs numeric characteristics of the machine running `R`, such as the largest integer or the machine's precision
- `format()` formats an object for pretty printing. `format()` is a generic function that is used with other types of objects. See `?format()` for additional arguments.

```
> # trim - If FALSE right justified with common width
> format(c(1,10,100,1000), trim = FALSE)
[1] "   1" "  10" " 100" "1000"
> format(c(1,10,100,1000), trim = TRUE)
[1] "1"    "10"   "100"  "1000"
> # nsmall - Minimum number of digits to the right of the decimal point
> format(13.7, nsmall = 3)
[1] "13.700"
> # scientific - Use scientific notation
> format(2^16, scientific = TRUE)
[1] "6.5536e+04"
```

Logical

- Logical values are represented by the reserved words `TRUE` and `FALSE` in all caps or simply `T` and `F`.

`!x`	NOT `x`
`x & y`	`x` AND `y` elementwise, returns a vector
`x && y`	`x` AND `y`, returns a single value
`x | y`	`x` OR `y` elementwise, returns a vector
`x || y`	`x` OR `y`, returns a single value
`xor(x,y)`	Exclusive OR of `x` and `y`, elementwise
`x %in% y`	`x` IN `y`
`x < y`	$x < y$
`x > y`	$x > y$
`x <= y`	$x \leq y$
`x >= y`	$x \geq y$
`x == y`	$x = y$
`x != y`	$x \neq y$
`isTRUE(x)`	`TRUE` if `x` is `TRUE`
`all(...)`	`TRUE` if all arguments are `TRUE`
`any(...)`	`TRUE` if at least one argument is `TRUE`
`identical(x,y)`	Safe and reliable way to test two objects for being *exactly* equal
`all.equal(x,y)`	Test if two objects are *nearly* equal

Example - Logical Operations

```
> x <- 1:10

> (x%%2==0) | (x > 5)    # What elements of x are even or greater than 5?
 [1] FALSE  TRUE FALSE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE
> x[(x%%2==0) | (x > 5)]
[1]  2  4  6  7  8  9 10

> y <- 5:15              # What elements of x are in y?
> x %in% y
 [1] FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
> x[x %in% y]
[1]  5  6  7  8  9 10

> any(x>5)               # Are any elements of x greater then 5?
[1] TRUE
> all(x>5)               # Are all the elements of x greater then 5?
[1] FALSE
```

Isn't that equal?

- In general, logical operators may not produce a single value and may return an `NA` if an element is `NA` or `NaN`.
- If you must get a single `TRUE` or `FALSE`, such as with `if` expressions, you should *NOT* use == or !=. Unless you are absolutely sure that nothing unusual can happen, you should use the `identical()` function instead.
- `identical()` only returns a single logical value, `TRUE` or `FALSE`, never `NA`

```
> name <- "Nick"
> if(name=="Nick") TRUE else FALSE
[1] TRUE

> # But what if name is never set to "Nick"?
> name <- NA
> if(name=="Nick") TRUE else FALSE
Error in if (name == "Nick") TRUE else FALSE :
  missing value where TRUE/FALSE needed
> if(identical(name, "Nick")) TRUE else FALSE
[1] FALSE
```

Isn't that equal?

- With `all.equal()` objects are treated as equal if the only difference is probably the result of inexact floating-point calculations. Returns `TRUE` if the mean relative difference is less then the specified tolerance.
- `all.equal()` either returns `TRUE` or a character string that describes the difference. Therefore, do not use `all.equal()` directly in `if` expressions, instead use with `isTRUE()` or `identical()`.

```
> (x <- sqrt(2))
[1] 1.414214
> x^2
[1] 2
> x^2==2
[1] FALSE
> all.equal(x^2, 2)
[1] TRUE
> all.equal(x^2, 1)
[1] "Mean relative difference: 0.5"
> isTRUE(all.equal(x^2, 1))
[1] FALSE
```

Character

- Character strings are defined by quotation marks, single ' ' or double " "

`cat()`	Concatenate objects and print to console (\n for newline)
`paste()`	Concatenate objects and return a string
`print()`	Print an object
`substr()`	Extract or replace substrings in a character vector
`strtrim()`	Trim character vectors to specified display widths
`strsplit()`	Split elements of a character vector according to a substring
`grep()`	Search for matches to a pattern within a character vector, returns a vector of the indices that matched
`grepl()`	Like `grep()`, but returns a logical vector
`agrep()`	Similar to `grep()`, but searches for approximate matches
`regexpr()`	Similar to `grep()`, but returns the position of the first instance of a pattern *within* a string
`gsub()`	Replace all occurrences of a pattern with a character vector
`sub()`	Like `gsub()`, but only replaces the first occurrence
`tolower(), toupper()`	Convert to all lower/upper case
`noquote()`	Print a character vector without quotations
`nchar()`	Number of characters
`letters, LETTERS`	Built-in vector of lower and upper case letters

Example - Character Functions

```
animals <- c("bird", "horse", "fish")
home    <- c("tree", "barn", "lake")

length(animals)   # Number of strings
nchar(animals)    # Number of characters in each string

cat("Animals:", animals)          # Need \n to move cursor to a newline
cat(animals, home, "\n")          # Joins one vector after the other

paste(animals, collapse=" ")      # Create one long string of animals
a.h=paste(animals, home, sep=".") # Pairwise joining of animals and home

# Split strings at ".", fixed=TRUE since "." is used for pattern matching
unlist(strsplit(a.h, ".", fixed=TRUE))

substr(animals, 2, 4) # Get characters 2-4 of each animal
strtrim(animals, 3)   # Print the first three characters

toupper(animals)      # Print animals in all upper case
```

Example - Pattern Matching

- A regular expression is a pattern that describes a set of strings.
  - ^ Start of character string
  - $ End of character string
  - . Any character
  - .{n} Any n characters
  - [· − ·] Range of letters, i.e. [a-c] is a, b, c
- See ?regexp for more options

```
# colors() is a character vector of all the built-in color names
colors()[grep("red", colors())]    # All colors that contain "red"
colors()[grep("^red", colors())]   # Colors that start with "red"
colors()[grep("red$", colors())]   # Colors that end with "red"
colors()[grep("red.", colors())]   # Colors with one character after "red"
colors()[grep("^[r-t]", colors())] # Colors that begin with r, s, or t

places <- c("home", "zoo", "school", "work", "park")
gsub("o", "0", places)     # Replace all "o" with "0"
sub("o", "0", places)      # Replace the first "o" with "0"

# Capitalize the first letter, uses Perl-like regular expressions
gsub("(\\w)(\\w*)", "\\U\\1\\L\\2", places, perl=TRUE)
```

Factor

- A factor is a categorical variable with a defined number of ordered or unordered levels. Use the function `factor` to create a factor variable.

```
> factor(rep(1:2, 4), labels=c("trt.1", "trt.2"))
[1] trt.1 trt.2 trt.1 trt.2 trt.1 trt.2 trt.1 trt.2
Levels: trt.1 trt.2

> factor(rep(1:3, 4), labels=c("low", "med", "high"), ordered=TRUE)
 [1] low  med  high low  med  high low  med  high low  med  high
Levels: low < med < high
```

`levels(x)`	Retrieve or set the levels of `x`
`nlevels(x)`	Return the number of levels of `x`
`relevel(x, ref)`	Levels of `x` are reordered so that the level specified by `ref` is first
`reorder()`	Reorders levels based on the values of a second variable
`gl()`	Generate factors by specifying the pattern of their levels
`cut(x, breaks)`	Divides the range of `x` into intervals (factors) determined by `breaks`

Example - Factor Functions

```
# Generate factor levels for 3 treatments and 2 cases per treatment
f <- gl(3, 2, labels=paste("trt",1:3, sep="."))
levels(f)
nlevels(f)
relevel(f, "trt.2")


x <- runif(10)
cut(x, 3)                       # Cut x into three intervals
cut(x, c(0,.25,.5,.75,1))       # Cut x at the given cut points
```

Dates and Times

- `R` has objects that are dates only and objects that are dates and times. We will just focus on dates. Look at `?DateTimeClasses` for information about how to handles dates and times.
- An `R` date object has the format: *Year-Month-Day*
- Operations with dates,
  - Days can be added or subtracted to a date
  - Dates can be subtracted
  - Dates can be compared using logical operators

`Sys.Date()`	Current date
`as.Date()`	Convert a character string to a date object
`format.Date()`	Change the format of a date object
`seq.Date()`	Generate sequence of dates
`cut.Date()`	Cut dates into intervals
`weekdays, months, quarters`	Extract parts of a date object
`julian`	Number of days since a given origin

`.Date` suffix is optional for calling `format.Date()`, `seq.Date()` and `cut.Date()`, but is necessary for viewing the appropriate documentation

Convert Strings to Date Objects

- Converting a string to a date object requires specifying a format string that defines the date format
- Any character in the format string other then the % symbol is interpreted literally.
- Common conversion specifications (see `?strptime` for a complete list),

%a	Abbreviated weekday name
%A	Full weekday name
%d	Day of the month
%B	Full month name
%b	Abbreviated month name
%m	Numeric month (01-12)
%y	Year without century (be very careful)
%Y	Year with century

```
> dates.1 <- c("5jan2008", "19aug2008", "2feb2009", "29sep2009")
> as.Date(dates.1, format="%d%b%Y")
[1] "2008-01-05" "2008-08-19" "2009-02-02" "2009-09-29"
> dates.2 <- c("5-1-2008", "19-8-2008", "2-2-2009", "29-9-2009")
> as.Date(dates.2, format="%d-%m-%Y")
[1] "2008-01-05" "2008-08-19" "2009-02-02" "2009-09-29"
```

Sequence of Dates

- To create a sequence of dates,

```
seq.Date(from, to, by, length.out = NULL)
```

`from, to`	Start and ending date objects
`by`	A character string, containing one of "day", "week", "month" or "year". Can optionally be preceded by a (positive or negative) integer and a space, or followed by a "s".
`length.out`	Integer, desired length of the sequence

```
> seq.Date(as.Date("2011/1/1"), as.Date("2011/1/31"), by="week")
[1] "2011-01-01" "2011-01-08" "2011-01-15" "2011-01-22" "2011-01-29"
> seq.Date(as.Date("2011/1/1"), as.Date("2011/1/31"), by="3 days")
 [1] "2011-01-01" "2011-01-04" "2011-01-07" "2011-01-10" "2011-01-13"
 [6] "2011-01-16" "2011-01-19" "2011-01-22" "2011-01-25" "2011-01-28"
[11] "2011-01-31"
> seq.Date(as.Date("2011/1/1"), by="week", length.out=10)
 [1] "2011-01-01" "2011-01-08" "2011-01-15" "2011-01-22" "2011-01-29"
 [6] "2011-02-05" "2011-02-12" "2011-02-19" "2011-02-26" "2011-03-05"
```

Cutting Dates

- To divide a sequence of dates in to levels,

```
cut.Date(x, breaks, start.on.monday = TRUE)
```

```
> jan <- seq.Date(as.Date("2011/1/1"), as.Date("2011/1/31"), by="days")
> cut(jan, breaks="weeks")
 [1] 2010-12-27 2010-12-27 2011-01-03 2011-01-03 2011-01-03 2011-01-03
 [7] 2011-01-03 2011-01-03 2011-01-03 2011-01-10 2011-01-10 2011-01-10
[13] 2011-01-10 2011-01-10 2011-01-10 2011-01-10 2011-01-17 2011-01-17
[19] 2011-01-17 2011-01-17 2011-01-17 2011-01-17 2011-01-17 2011-01-24
[25] 2011-01-24 2011-01-24 2011-01-24 2011-01-24 2011-01-24 2011-01-24
[31] 2011-01-31
6 Levels: 2010-12-27 2011-01-03 2011-01-10 2011-01-17 ... 2011-01-31
```

January 2011

Sun	Mon	Tue	Wed	Thr	Fri	Sat
26	27	28	29	30	31	1
2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	29
30	31	1	2	3	4	5

Operations with Dates

- Operations with dates,
  - Days can be added or subtracted to a date
  - Dates can be subtracted
  - Dates can be compared using logical operators

```
> jan1 <- as.Date("2011/1/1")
> (jan8 <- jan1 + 7)        # Add 7 days to 2011/1/1
[1] "2011-01-08"
> jan1 - 14                 # Subtract 2 weeks from 2011/1/8
[1] "2010-12-18"
> jan8 - jan1               # Number of days between 2011/1/1 and 2011/1/8
Time difference of 7 days
> jan8 > jan1               # Compare dates
[1] TRUE

> # Use format to extract parts of a date object or change the appearance
> format.Date(jan8, "%Y")
[1] "2011"
> format.Date(jan8, "%b-%d")
[1] "Jan-01"
```

Missing Data

- R denotes data that is not available by NA
- How a function handles missing data depends on the function. For example mean only ignores NAs if the argument na.rm=TRUE, whereas which always ignores missing data.

```
> x <- c(4, 7, 2, 0, 1, NA)
> mean(x)
[1] NA
> mean(x, na.rm=TRUE)
[1] 2.8
> which(x>4)
[1] 2
```

- See the documentation for how a particular function handles missing data.
- Quantities that are not a number, such as 0/0, are denoted by NaN. In R NaN implies NA (NaN refers to unavailable numeric data and NA refers to any type of unavailable data)
- Undefined or null objects are denoted in R by NULL. This is useful when we do not want to add row labels to a matrix. For example,

```
x <- matrix(1:4, ncol=2, dimnames=list(NULL, c("c.1", "c.2")))
```

Detecting Missing Data

- To test for missing data avoid using `identical()` and never use ==. Using `identical()` relies on unreliable internal computations and == will always evaluate to `NA` or `NaN`.
- Functions used for detecting missing data,

`is.na(x)`	Tests for `NA` or `NaN` data in `x`
`is.nan(x)`	Tests for `NaN` data in `x`
`is.null(x)`	Tests if `x` is `NULL`

```
> x <- c(4, 7, 2, 0, 1, NA)
> x==NA
[1] NA NA NA NA NA NA
> is.na(x)
[1] FALSE FALSE FALSE FALSE FALSE  TRUE
> any(is.na(x))
[1] TRUE
> (y <- x/0)
[1] Inf Inf Inf NaN Inf  NA
> is.nan(y)
[1] FALSE FALSE FALSE  TRUE FALSE FALSE
> is.na(y)
[1] FALSE FALSE FALSE  TRUE FALSE  TRUE
```

Testing and Coercing Objects

- All objects in `R` have a type. We can test the type of an object using a `is.`*type*() function.
- We can also attempt to coerce objects of one type to another using a `as.`*type*() function.
- Automatic conversions,
  - Logical values are converted to numbers by setting `FALSE` as 0 and `TRUE` as 1
  - Logical, numeric, factor and date types are converted to characters by converting each element/level individually
- Some general rules for coercion,
  - Numeric values are coerced to logical by treating all non-zero values as `TRUE`
  - Numeric characters can be coerced to numbers, but non-numeric characters cannot
  - Factor levels can be coerced to numeric and numbers can be coerced to factors with a level for each unique number
  - Vectors, matrices and arrays are coerced to lists by making each element a vector of length 1
  - Vectors, matrices, arrays can also be coerced from one form to another

Testing and Coercing Functions

Type	Testing	Coercing
Array	`is.array()`	`as.array()`
Character	`is.character()`	`as.character()`
Dataframe	`is.data.frame()`	`as.data.frame()`
Factor	`is.factor()`	`as.factor()`
List	`is.list()`	`as.list()`
Logical	`is.logical()`	`as.logical()`
Matrix	`is.matrix()`	`as.matrix()`
Numeric	`is.numeric()`	`as.numeric()`
Vector	`is.vector()`	`as.vector()`

Example - Testing and Coercing Objects

```
> x <- 1:10
> x>5
 [1] FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE
> sum(x>5)    # Automatic conversion to numeric vector
[1] 5
> is.vector(x)
[1] TRUE
> is.numeric(x)
[1] TRUE
> as.list(x)
[[1]]
[1] 1

[[2]]
[1] 2
. . .
[[10]]
[1] 10
> as.numeric("123")
 [1] 123
```