同濟大學
TONGJI UNIVERSITY

生物信息学系
DEPARTMENT OF BIOINFORMATICS

Section 11

R: Data Structures

张勇

yzhang@tongji.edu.cn

同济大学

2021年5月24日

Vectors

- Vectors are homogeneous.
- Vectors can be indexed by position.
- Vectors can be indexed by multiple positions, returning a sub-vector.
- Vector elements can have names.

```
> v <- c(10, 20, 30)
> names(v) <- c("Moe", "Larry", "Curly")
```

- If vector elements have names then you can select them by name.

```
> v["Larry"]
```

Lists

- Lists are heterogeneous.
- Lists can be indexed by position.
  - So lst[[2]] refers to the second element of lst. Note the double square brackets.
- Lists let you extract sub-lists.
  - So lst[c(2,3)] is a sublist of lst that consists of the second and third elements. Note the single square brackets
- List elements can have names.
  - Both lst[["Moe"]] and lst$Moe refer to the element named “Moe”.

Scalars

- A scalar is simply a vector that contains exactly one element.

```
> pi[1]
[1] 3.141593
> pi[2]
[1] NA
```

Matrices

- A matrix is just a vector that has dimensions.

```
> A <- 1:6
> dim(A)
NULL
> dim(A) <- c(2,3)
> print(A)
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

Arrays

- 3-dimensional or even *n*-dimensional structures.

```
> D <- 1:12
> dim(D) <- c(2,3,2)
> print(D)
, , 1

     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6

, , 2

     [,1] [,2] [,3]
[1,]    7    9   11
[2,]    8   10   12
```

Factors

- R keeps track of the unique values in a vector, and each unique value is called a *level* of the associated factor.

- Factors are usually used for categorical variables or grouping.

Data Frames

- A data frame is powerful and flexible structure, with tabular (rectangular) data structure.
  - The elements of the list are vectors and/or factors.
  - Those vectors and factors are the columns of the data frame.
  - The vectors and factors must all have the same length; in other words, all columns must have the same height.
  - The equal-height columns give a rectangular shape to the data frame.
  - The columns must have names.

Mode: Physical Type

- In R, every object has a *mode*, which indicates how it is stored in memory:

```
> mode(c(2.7182, 3.1415))
[1] "numeric"
```

Object	Example	Mode
Number	`3.1415`	numeric
Vector of numbers	`c(2.7.182, 3.1415)`	numeric
Character string	`"Moe"`	character
Vector of character strings	`c("Moe", "Larry", "Curly")`	character
Factor	`factor(c("NY", "CA", "IL"))`	numeric
List	`list("Moe", "Larry", "Curly")`	list
Data frame	`data.frame(x=1:3, y=c("NY", "CA", "IL"))`	list
Function	`print`	function

Mode: Abstract Type

- In R, every object also has a *class*, which defines its abstract type.

```
> d <- as.Date("2021-05-24")
> mode(d)
[1] "numeric"
> class(d)
[1] "Date"
```

- When print an object, R calls the appropriate print function according to the object's class.

Appending Data to a Vector

- **Problem**
  - You want to append additional data items to a vector.

- **Solution**
  - Use **c** to construct a vector with the additional data item:

    ```
    > v <- c(v, newItems)
    ```

  - Assign the new item to the next vector element:

    ```
    > v[length(v)+1] <- newItem
    ```

  - Use **append** function, but runs more slowly than both the vector constructor and the element assignment.

Inserting Data into a Vector

- **Problem**
  - You want to insert one or more data items into a vector.

- **Solution**
  - The **append** function inserts data into a vector by using the **after** parameter:

```
> append(1:10, 99, after=5)
 [1]  1  2  3  4  5 99  6  7  8  9 10
> append(1:10, 99, after=0)
 [1] 99  1  2  3  4  5  6  7  8  9 10
```

Understanding the Recycling Rule

- **Problem**
  - How R handles vectors of unequal length?
- **Solution**
  - R will recycle the shorter-vector elements as often as necessary until the operation is complete:

```
> (1:6) + (1:3)
[1] 2 4 6 5 7 9
```

  - The **cbind** function can create column vectors:

```
> cbind(1:3, 1:2)
     [,1] [,2]
[1,]    1    1
[2,]    2    2
[3,]    3    1
Warning message:
In cbind(1:3, 1:2) :
  number of rows of result is not a multiple of vector length (arg 2)
```

Creating a Factor

- **Problem**
  - How to treat a vector of character strings or integers as a factor?

- **Solution**
  - The **factor** function encodes the vector of discrete values into a factor:

```
> f <- factor(c("Win","Win","Lose","Tie","Win","Lose"))
> f
[1] Win  Win  Lose Tie  Win  Lose
Levels: Lose Tie Win
```

Creating a Factor

– If a vector contains only a subset of possible values, then include a second argument of **factor** function will give the possible levels of the factor with the specified order:

```
> wday <-
c("Wed","Thu","Mon","Wed","Thu","Thu","Thu","Tue","Thu","Tue")
> f <- factor(wday, c("Mon","Tue","Wed","Thu","Fri"))
> f
 [1] Wed Thu Mon Wed Thu Thu Thu Tue Thu Tue
Levels: Mon Tue Wed Thu Fri
```

Combining Multiple Vectors into One Vector and a Factor

- **Problem**
  - You have several groups of data, with one vector for each group. You want to combine the vectors into one large vector and simultaneously create a parallel factor that identifies each value's original group.
- **Solution**
  - Use the **stack** function to combine the list into a two-column data frame. The data frame's columns are called **values** and **ind**. The first column contains the data, and the second column contains the parallel factor.

```
> freshmen <- c(0.60, 0.35, 0.44, 0.62, 0.60)
> sophomores <- c(0.70, 0.61, 0.63, 0.87, 0.85, 0.70, 0.64)
> juniors <- c(0.76, 0.71, 0.92, 0.87)
> comb <- stack(list(fresh=freshmen, soph=sophomores,
+ jrs=juniors))
```

Creating a List

- **Problem**
  - You want to create and populate a list.

- **Solution**
  - Use **list** function to create a list from individual data items:

```
> lst <- list(0.5, 0.841, 0.977)
> lst <- list(mid=0.5, right=0.841, far.right=0.977)
```

Building a Name/Value Association List

- **Problem**
  - You want to create a list that associates names and values.
- **Solution**
  - Use **list** function:

```
> values <- pnorm(-2:2)
> names <- c("far.left", "left", "mid", "right", "far.right")
> lst <- list()
> lst[names] <- values
> for (nm in names(lst)) cat("The", nm, "limit is", lst[[nm]], "\n")
The far.left limit is 0.02275013
The left limit is 0.1586553
The mid limit is 0.5
The right limit is 0.8413447
The far.right limit is 0.9772499
```

Removing an Element from a List

- **Problem**
  - You want to remove an element from a list.

- **Solution**
  - Assign **NULL** to the element. R will remove it from the list:

```
> years <- list(Kennedy=1960, Johnson=1964, Carter=1976,
+           Clinton=1994)
> years[["Johnson"]] <- NULL
> years[years > 1980] <- NULL
```

  - Your list contains **NULL** values. You want to remove them:

```
> lst <- list("Moe", NULL, "Curly")
> lst[sapply(lst, is.null)] <- NULL
```

Initializing a Matrix

- **Problem**
  - You want to create a matrix and initialize it from given values.

- **Solution**
  - Use the **matrix** function to shape the vector into a matrix:

```
> theData <- c(1.1, 1.2, 2.1, 2.2, 3.1, 3.2)
> mat <- matrix(theData, 2, 3)
> mat <- matrix(theData, 2, 3, byrow=TRUE)
```

Performing Matrix Operations

- **Problem**
  - You want to perform matrix operations such as transpose, matrix inversion, matrix multiplication, or constructing an identity matrix.

- **Solution**
  - t(A): matrix transposition of A;
  - solve(A): matrix inverse of A;
  - A %*% B: matrix multiplication of A and B;
  - A * B: element-wise multiplication;
  - diag(n): an *n*-by-*n* diagonal (identity) matrix.

Selecting One Row or Column from a Matrix

- **Problem**
  - You want to select a single row or column from a matrix.

- **Solution**
  - Use normal indexing or names of rows / columns:

```
> mat <- matrix(c(1.1, 1.2, 2.1, 2.2, 3.1, 3.2), 2, 3)
> rownames(mat) <- c("R1", "R2")
> colnames(mat) <- c("C1", "C2", "C3")
> mat["R1",]
> mat[1,]
> mat[,"C1"]
> mat[,1]
```

Initializing a Data Frame from Column Data

- **Problem**
  - Your data is organized by columns, and you want to assemble it into a data frame.

- **Solution**
  - Use the **data.frame** function to assemble them into a data frame:

```
> hour <- sample(1:12, 6, replace=TRUE)
> min <- sample(0:59, 6, replace=TRUE)
> sec <- sample(0:59, 6, replace=TRUE)
> ampm <- sample(c("AM", "PM"), 6, replace=TRUE)
> dfrm <- data.frame(hour, min, sec, ampm)
> dfrm <- data.frame(H=hour, M=min, S=sec, AP=ampm)
```

Appending Rows to a Data Frame

- **Problem**
  - You want to append one or more new rows to a data frame.
- **Solution**
  - Create a second, temporary data frame containing the new rows. Then use the **rbind** function to append the temporary data frame to the original data frame:

```
> suburbs <- data.frame(city="Chicago", county="Cook",
+         state="IL", pop=2853114)
> newRow <- data.frame(city="West Dundee", county="Kane",
+         state="IL", pop=5428)
> suburbs <- rbind(suburbs, newRow)
> suburbs
         city   county state    pop
1     Chicago     Cook    IL 2853114
2 West Dundee     Kane    IL    5428
```

Selecting Data Frame Columns

- **Problem**
  - You want to select columns from a data frame.

- **Solution**
  - To select a single column by position or name:

```
> suburbs[1]      # Returns a data frame containing one column.
> suburbs["city"]
> suburbs[[1]]    # Returns one column.
> suburbs[["city"]]
> suburbs$city
```

Selecting Rows and Columns More Easily

- **Problem**
  - You want an easier way to select rows and columns from a data frame or matrix.
- **Solution**
  - Use the **subset** function. The **select** argument is a column name, or a vector of column names, to be selected. The **subset** function is most useful when you combine the **select** and **subset** arguments.

```
> library(MASS)
> subset(Cars93, select=Model, subset=(MPG.city > 30))
> subset(Cars93, select=c(Model,Min.Price,Max.Price),
+          subset=(Cylinders == 4 & Origin == "USA"))
> subset(Cars93, select=c(Manufacturer,Model),
+          subset=c(MPG.highway > median(MPG.highway)))
```

Selecting Rows and Columns More Easily

– Use the **subset** function with a negated argument for the **select** parameter to exclude a column from a data frame using its name:

```
> subset(Cars93, select=c(-Model,-Min.Price,-Max.Price))
```

Removing NAs from a Data Frame

- **Problem**
  - Your data frame contains NA values, which is creating problems for you.

- **Solution**
  - Use **na.omit** to remove rows that contain any NA values:

```
> x <- c(1, NA, 3, 4, 5)
> y <- c(6, 7, NA, 9, 10)
> dfrm <- data.frame(x, y)
> cumsum(dfrm)          # cannot get what you want
> dfrm2 <- na.omit(dfrm)
> cumsum(dfrm2)         # can complete the summations
```

Combining Two Data Frames

- **Problem**
  - How to combine the contents of two data frames into one?
- **Solution**
  - Use **cbind** to combine the columns of two data frames side by side; use **rbind** to "stack" the rows of two data frames:

```
> stooges <- data.frame(name=c("Moe","Larry","Curly"),
+ n.marry=c(1,1,4),n.child=c(2,2,2))
> birth <- data.frame(birth.year=c(1887,1902,1903),
+ birth.place=c("Bensonhurst","Philadelphia","Brooklyn"))
> cbind(stooges,birth)
   name  n.marry    n.child   birth.year        birth.place
1 Moe      1        2         1887Bensonhurst
2 Larry    1        2         1902Philadelphia
3 Curly    4        2         1903Brooklyn
```

Combining Two Data Frames

```
> guys <- data.frame(name=c("Tom","Dick","Harry"),
+           n.marry=c(4,1,1),n.child=c(2,4,1))
> rbind(stooges,guys)
   name n.marry n.child
1   Moe       1       2
2 Larry       1       2
3 Curly       4       2
4   Tom       4       2
5  Dick       1       4
6 Harry       1       1
```

Merging Data Frames by Common Column

- **Problem**
  - You have two data frames that share a common column. You want to merge their rows into one data frame by matching on the common column.
- **Solution**
  - Use the **merge** function to join the data frames into one new data frame based on the common column:

```
> born <- data.frame(name=c("Moe","Larry","Curly", "Harry"),
+           year.born=c(1887,1902,1903, 1964))
> died <- data.frame(name=c("Curly","Moe","Larry"),
+           year.died=c(1952,1975,1975))
> merge(born, died, by="name")
   name year.born year.died
1 Curly      1903   1952
2 Larry      1902   1975
3   Moe      1887   1975
```

Accessing Data Frame Contents More Easily

- **Problem**
  - Your data is stored in a data frame. You are getting tired of repeatedly typing the data frame name and want to access the columns more easily.

- **Solution**
  - Use the **with** function to expose the column names:

```
> z <- (suburbs$pop - mean(suburbs$pop)) / sd(suburbs$pop)
> z <- with(suburbs, (pop - mean(pop)) / sd(pop))
```

Accessing Data Frame Contents More Easily

– Use the **attach** function to insert the data frame into your search list:

```
> attach(suburbs)
> z <- (pop - mean(pop)) / sd(pop)
> pop
[1] 2853114    5428
> pop <- pop / 1000
> pop
[1] 2853.114    5.428
> suburbs$pop              # Original data is unchanged
[1] 2853114    5428
```

Converting One Atomic Value into Another

- **Problem**
  - You have a data value which has an atomic data type: character, complex, double, integer, or logical. You want to convert this value into one of the other atomic data types.

- **Solution**
  - For each atomic data type, there is a function for converting values to that type:

    as.character(x)

    as.complex(x)

    as.numeric(x) or as.double(x)

    as.integer(x)

    as.logical(x)

Converting One Atomic Value into Another

```
> as.numeric(" 3.14 ")
[1] 3.14
> as.integer(3.14)
[1] 3
> as.numeric("foo")
[1] NA
Warning message:
NAs introduced by coercion
> as.numeric(c("1","2.718","7.389","20.086"))
[1]  1.000  2.718  7.389 20.086
> as.character(101:105)
[1] "101" "102" "103" "104" "105"
> as.numeric(FALSE)
[1] 0
> as.numeric(TRUE)
[1] 1
> as.logical(1)
[1] TRUE
```

Converting One Structured Data Type into Another

- **Problem**
  - You want to convert a variable from one structured data type to another.

- **Solution**
  - These functions convert their argument into the corresponding structured data type:

    as.data.frame(x)

    as.list(x)

    as.matrix(x)

    as.vector(x):

Converting One Structured Data Type into Another

Conversion	How	Notes
Vector→List	`as.list(vec)`	Don't use `list(vec)`; that creates a 1-element list whose only element is a copy of `vec`.
Vector→Matrix	To create a 1-column matrix: `cbind(vec)` *or* `as.matrix(vec)` To create a 1-row matrix: `rbind(vec)` To create an $n \times m$ matrix: `matrix(vec,n,m)`	See Recipe 5.14.
Vector→Data frame	To create a 1-column data frame: `as.data.frame(vec)` To create a 1-row data frame: `as.data.frame(rbind(vec))`	
List→Vector	`unlist(lst)`	Use `unlist` rather than `as.vector`; see Note 1 and Recipe 5.11.
List→Matrix	To create a 1-column matrix: `as.matrix(lst)` To create a 1-row matrix: `as.matrix(rbind(lst))` To create an $n \times m$ matrix: `matrix(lst,n,m)`	

Converting One Structured Data Type into Another

Conversion	How	Notes
List→Data frame	If the list elements are columns of data: `as.data.frame(lst)` If the list elements are rows of data: Recipe 5.19	
Matrix→Vector	`as.vector(mat)`	Returns all matrix elements in a vector.
Matrix→List	`as.list(mat)`	Returns all matrix elements in a list.
Matrix→Data frame	`as.data.frame(mat)`	
Data frame→Vector	To convert a 1-row data frame: `dfrm[1,]` To convert a 1-column data frame: `dfrm[,1]` or `dfrm[[1]]`	See Note 2.
Data frame→List	`as.list(dfrm)`	See Note 3.
Data frame→Matrix	`as.matrix(dfrm)`	See Note 4.

Splitting a Vector into Groups

- **Problem**
  - You have a vector. Each element belongs to a different group, and the groups are identified by a grouping factor. You want to split the elements into the groups.

- **Solution**
  - Suppose the vector is x and the factor is f. You can use the **split** function:

    ```
    > groups <- split(x, f)
    ```

  - You can also use the **unstack** function:

    ```
    > groups <- unstack(data.frame(x,f))
    ```

  - Both functions return a list of vectors, where each vector contains the elements for one group.

Splitting a Vector into Groups

```
>  library(MASS)
> g <- split(Cars93$MPG.city, Cars93$Origin)
> g
$USA
 [1] 22 19 16 19 16 16 25 25 19 21 18 15 17 17 20 23 20 29 23 22 17 21 18 29
20
[26] 31 23 22 22 24 15 21 18 17 18 23 19 24 23 18 19 23 31 23 19 19 19 28

$`non-USA`
 [1] 25 18 20 19 22 46 30 24 42 24 29 22 26 20 17 18 18 29 28 26 18 17 20 19
29
[26] 18 29 24 17 21 20 33 25 23 39 32 25 22 18 25 17 21 18 21 20

> median(g[[1]])
[1] 20
> median(g[[2]])
[1] 22
```

Applying a Function to Each List Element

- **Problem**
  - You have a list, and you want to apply a function to each element of the list.

- **Solution**
  - Use either the **lapply** function or the **sapply** function, depending upon the desired form of the result. **lapply** always returns the results in list, whereas **sapply** returns the results in a vector if that is possible:

```
> lst <- lapply(lst, fun)
> vec <- sapply(lst, fun)
```

Applying a Function to Each List Element

```
> scores <- list(S1, S2, S3, S4)
> lapply(scores, length)
[[1]]
[1] 36

[[2]]
[1] 39

[[3]]
[1] 38

[[4]]
[1] 36
```

Applying a Function to Each List Element

```
> sapply(scores, length)
[1] 36 39 38 36
> sapply(scores, mean)
[1] 88.77778 89.79487 89.23684 88.86111
> sapply(scores, sd)
[1]  7.720515 10.543592  7.178926  8.208542
> sapply(scores, range)
     [,1] [,2] [,3] [,4]
[1,]   68   60   75   63
[2,]   98  100   99   99
```

Applying a Function to Every Row

- **Problem**
  - You have a matrix. You want to apply a function to every row, calculating the function result for each row.

- **Solution**
  - Use the **apply** function. Set the second argument to 1 to indicate row-by-row application of a function:

```
> results <- apply(mat, 1, fun)
```

  - The **apply** function will call fun once for each row, assemble the returned values into a vector, and then return that vector.

Applying a Function to Every Row

```
> mat <- matrix(sample(1:100, 20, replace=TRUE), 4,5)
> apply(mat, 1, mean)
[1] 35.4 64.8 39.4 75.4
> apply(mat, 1, range)
     [,1] [,2] [,3] [,4]
[1,]    5   38   15   37
[2,]   71   86   78   99
```

Applying a Function to Every Column

- **Problem**
  - You have a matrix or data frame, and you want to apply a function to every column.
- **Solution**
  - For a matrix, use the **apply** function. Set the second argument to 2, which indicates column-by-column application of the function:

    ```
    > results <- apply(mat, 2, fun)
    ```
  - For a data frame, use the **lapply** or **sapply** functions:

    ```
    > lst <- lapply(dfrm, fun)
    > vec <- sapply(dfrm, fun)
    ```
  - You can use apply on data frames, too, but only if the data frame is homogeneous.

Applying a Function to Every Column

```
> Hirsch <- read.table("Hirsch_Cancer_Cell.tab", row.names=1,
+            header=TRUE)
> cors <- sapply(Hirsch[,-1], cor, y=Hirsch[,1])
> mask <- (rank(-abs(cors)) <= 4)
> best.sub <- Hirsch[,-1][,mask]
> head(best.sub)
           hr0_D3   hr1_D1   hr1_D2   hr2_D2
10       3.958808 4.161498 3.994645 3.881746
100      6.424001 6.428909 6.502970 6.273681
1000     5.803091 5.860483 5.748568 5.594535
10000 3.617919 3.624202 3.435009 3.297049
10001 5.560836 5.000575 5.408475 5.325964
10002 4.044407 4.255866 3.970381 4.103564
```

Applying a Function to Groups of Data

- **Problem**
  - How to process the data by groups?

- **Solution**
  - Create a grouping factor that identifies the group of each corresponding datum. Then use the **tapply** function, which will apply a function to each group of data:

    ```
    > tapply(x, f, fun)
    ```

Applying a Function to Groups of Data

```
> library(MASS)
> tapply(Cars93$MPG.city, Cars93$Origin, mean)
     USA  non-USA
20.95833 23.86667
> tapply(Cars93$MPG.city, Cars93$Origin, length)
  USA non-USA
 48      45
> tapply(Cars93$MPG.city, Cars93$Origin, sum)
 USA non-USA
 1006    1074
```

Applying a Function to Groups of Rows

- **Problem**
  - You want to apply a function to groups of rows within a data frame.

- **Solution**
  - Define a grouping factor for every row in your data frame. For each such group of rows, the **by** function puts the rows into a temporary data frame and calls your function with that argument. The **by** function collects the returned values into a list and returns the list:

```
> by(dfrm, fact, fun)
```

Applying a Function to Groups of Rows

```
> dfrm <- data.frame(f=c(rep("A",50), rep("B",30), rep("C", 20)),
+           v=c(rnorm(50)-0.2,rnorm(30), rnorm(20)+0.2))
> by(dfrm, dfrm$f, summary)
dfrm$f: A
 f           v
 A:50     Min.      :-2.2870
 B: 0     1st Qu.  :-0.8492
 C: 0     Median :-0.2448
          Mean    :-0.2351
          3rd Qu. : 0.4240
          Max.     : 1.4526
------------------------------------------------------------
dfrm$f: B
f           v
A: 0      Min.      :-1.6488
B:30      1st Qu.  :-0.1468
… …
```

Applying a Function to Parallel Vectors or Lists

- **Problem**
  - You have a function, say **f**, that takes multiple arguments. You want to apply the function element-wise to vectors and obtain a vector result. Unfortunately, the function is not vectorized; that is, it works on scalars but not on vectors.

- **Solution**
  - Use the **mapply** function. It will apply the function f to your arguments element-wise:

    > mapply(f, vec_1, vec_2, ..., vec_N)

Applying a Function to Parallel Vectors or Lists

```
> gcd <- function(a,b) {
+         if (b == 0) return(a)
+         else return(gcd(b, a %% b)) }
> gcd(c(25,9,12), c(9,6,3))
[1]   1 NaN NaN
Warning messages:
1: In if (b == 0) return(a) else return(gcd(b, a%%b)) :
  the condition has length > 1 and only the first element will be used
… …
> mapply(gcd, c(25,9,12), c(9,6,3))
[1] 1 3 3
```

Getting the Length of a String

- **Problem**
  - You want to know the length of a string.

- **Solution**
  - Use the **nchar** function, not the **length** function:

```
> nchar("Moe")
[1] 3
> length("Moe")
[1] 1
> nchar(c("Moe","Larry","Curly"))
[1] 3 5 5
```

Concatenating Strings

- **Problem**
  - You want to join together two or more strings into one string.

- **Solution**
  - Use the **paste** function:

```
> paste("Everybody", "loves", "stats.")
[1] "Everybody loves stats."
> paste("Everybody", "loves", "stats.", sep="-")
[1] "Everybody-loves-stats."
> paste("Everybody", "loves", "stats.", sep="")
[1] "Everybodylovesstats."
> paste("The square root of twice pi is approximately", sqrt(2*pi))
[1] "The square root of twice pi is approximately 2.506628274631"
```

Concatenating Strings

```
> stooges <- c("Moe", "Larry", "Curly")
> paste(stooges, "loves", "stats.")
[1] "Moe loves stats."   "Larry loves stats." "Curly loves stats."
> paste(stooges, "loves", "stats", collapse=", and ")
[1] "Moe loves stats, and Larry loves stats, and Curly loves stats”
```

Extracting Substrings

- **Problem**
  - You want to extract a portion of a string according to position.

- **Solution**
  - Use **substr**(string,start,end) to extract the substring that begins at *start* and ends at *end*:

```
> substr("Statistics", 1, 4)
[1] "Stat"
> ss <- c("Moe", "Larry", "Curly")
> substr(ss, 1, 3)
[1] "Moe" "Lar" "Cur"
> cities <- c("New York, NY", "Los Angeles, CA", "Peoria, IL")
> substr(cities, nchar(cities)-1, nchar(cities))
[1] "NY" "CA" "IL"
```

Splitting a String According to a Delimiter

- **Problem**
  - You want to split a string into substrings. The substrings are separated by a delimiter.

- **Solution**
  - Use **strsplit**, which takes two arguments: the string and the delimiter of the substrings:

    > strsplit(*string*, *delimiter*)

Splitting a String According to a Delimiter

```
> path <- "/home/mike/data/trials.csv"
> strsplit(path, "/")
[[1]]
[1] ""          "home"      "mike"      "data"      "trials.csv"

> paths <- c("/home/mike/data/trials.csv",
+            "/home/mike/data/errors.csv",
+            "/home/mike/corr/reject.doc")
> strsplit(paths, "/")
[[1]]
[1] ""          "home"      "mike"      "data"      "trials.csv"

[[2]]
[1] ""          "home"      "mike"      "data"      "errors.csv"

[[3]]
[1] ""          "home"      "mike"      "corr"      "reject.doc”
```

Replacing Substrings

- **Problem**
  - Within a string, you want to replace one substring with another.

- **Solution**
  - Use **sub** to replace the first instance of a substring:

    > sub(*old, new, string*)
  - Use **gsub** to replace all instances of a substring:

    > gsub(*old, new, string*)

Replacing Substrings

```
> s <- "Curly is the smart one. Curly is funny, too."
> sub("Curly", "Moe", s)
[1] "Moe is the smart one. Curly is funny, too."
> gsub("Curly", "Moe", s)
[1] "Moe is the smart one. Moe is funny, too.”
> sub(" and SAS", "",
+           "For really tough problems, you need R and SAS.")
[1] "For really tough problems, you need R.”
```

Generating All Pairwise Combinations of Strings

- **Problem**
  - You have two sets of strings, and you want to generate all combinations from those two sets.
- **Solution**
  - Use the **outer** and **paste** functions together to generate the matrix of all possible combinations:

```
> m <- outer(strings1, strings2, paste, sep="-")
> locations <- c("NY", "LA", "CHI", "HOU")
> treatments <- c("T1", "T2", "T3")
> outer(locations, treatments, paste, sep="-")
     [,1]     [,2]     [,3]
[1,] "NY-T1"  "NY-T2"  "NY-T3"
[2,] "LA-T1"  "LA-T2"  "LA-T3"
[3,] "CHI-T1" "CHI-T2" "CHI-T3"
[4,] "HOU-T1" "HOU-T2" "HOU-T3"
```

Getting the Current Date

- **Problem**
  - You need to know today's date.

- **Solution**
  - The **Sys.Date** function returns the current date:

```
> Sys.Date()
[1] "2020-05-25”
> class(Sys.Date())
[1] "Date"
```

Converting a String into a Date

- **Problem**
  - You have the string representation of a date, and you want to convert that into a **Date** object.
- **Solution**
  - You can use **as.Date**. By default, **as.Date** assumes the string looks like yyyy-mm-dd. To handle other formats, you must specify the **format** parameter of **as.Date**. Use format="%m/%d/%Y" if the date is in American style.

```
> as.Date("2010-12-31")
[1] "2010-12-31"
> as.Date("12/31/2010", format="%m/%d/%Y")
[1] "2010-12-31"
```

Converting a Date into a String

- **Problem**
  - You want to convert a Date object into a character string, usually because you want to print the date.
- **Solution**
  - Use either **format** or **as.character**:

```
> format(Sys.Date())
[1] "2021-05-24"
> as.character(Sys.Date())
[1] "2021-05-24”
```

  - Both functions allow a **format** argument that controls the formatting:

```
> format(Sys.Date(), format="%m/%d/%Y")
[1] "05/24/2021"
```

Converting a Date into a String

- The **format** argument:

%b	Abbreviated month name (“Jan”)
%B	Full month name (“January”)
%d	Day as a two-digit number
%m	Month as a two-digit number
%y	Year without century (00–99)
%Y	Year with century

Converting Year, Month, and Day into a Date

- **Problem**
  - You have a date represented by its year, month, and day. You want to merge these elements into a single Date object representation.

- **Solution**
  - Use the **ISOdate** function:

    > ISOdate(*year, month, day*)
  - The result is a **POSIXct** object that you can convert into a **Date** object:

    > as.Date(ISOdate(*year, month, day*))

Converting Year, Month, and Day into a Date

```
> years <- 2013:2015
> months <- 5:7
> days <- 13:15
> ISOdate(years, months, days)
[1] "2013-05-13 12:00:00 GMT" "2014-06-14 12:00:00 GMT"
[3] "2015-07-15 12:00:00 GMT"
> as.Date(ISOdate(years, months, days))
[1] "2013-05-13" "2014-06-14" "2015-07-15”
```

Creating a Sequence of Dates

- **Problem**
  - You want to create a sequence of dates, such as a sequence of daily, monthly, or annual dates.

- **Solution**
  - The **seq** function is a generic function that has a version for **Date** objects.

Creating a Sequence of Dates

```
> seq(from=s, to=e, by=1)
[1] "2012-01-01" "2012-01-02" "2012-01-03" "2012-01-04" "2012-01-05"
> seq(from=s, by=1, length.out=5)
[1] "2012-01-01" "2012-01-02" "2012-01-03" "2012-01-04" "2012-01-05"
> seq(from=s, by="month", length.out=5)
[1] "2012-01-01" "2012-02-01" "2012-03-01" "2012-04-01" "2012-05-01"
> seq(from=s, by="3 months", length.out=5)
[1] "2012-01-01" "2012-04-01" "2012-07-01" "2012-10-01" "2013-01-01"
> seq(from=s, by="year", length.out=5)
[1] "2012-01-01" "2013-01-01" "2014-01-01" "2015-01-01" "2016-01-01"
> seq(as.Date("2010-01-29"), by="month", length.out=5)
[1] "2010-01-29" "2010-03-01" "2010-03-29" "2010-04-29" "2010-05-29"
```

Summary

- R Cookbook
  - Chapter 5. Data Structures
  - Chapter 6. Data Transformations
  - Chapter 7. Strings and Dates