# R Language - Basics

# Creating new variables

## Recoding variables

## Renaming variables

Operators

Arithmetic Operators

Logical Operators

Built-in Functions

Numeric Functions

Character Functions

Statistical Probability Functions

Other Statistical Functions

# Unique, levels, distinct Functions

# Control Structures

## If condition

## Ifelse condition

## Ifelse If condition

## Switch

## While loop

## For loop

## Break

## Continue

# User-written Functions

# Sorting Data

# Merging Data

## Adding Columns

## Adding Rows

# Reshaping Data

## Transpose

## The Reshape Package

# Subsetting Data

## Selecting (Keeping) Variables

## Excluding (DROPPING) Variables

## Selecting Observations

## Selection using the Subset Function

## Random Samples

Data Type Conversion

# Creating new variables

Use the assignment operator **<-** to create new variables. A wide array of operators and functions are available here.

# Three examples for doing the same computations  
  
mydata$sum <- mydata$x1 + mydata$x2  
mydata$mean <- (mydata$x1 + mydata$x2)/2  
  
attach(mydata)  
mydata$sum <- x1 + x2  
mydata$mean <- (x1 + x2)/2  
detach(mydata)  
  
mydata <- transform( mydata,  
sum = x1 + x2,  
mean = (x1 + x2)/2   
)

## Recoding variables

In order to recode data, you will probably use one or more of R's control structures.

# create 2 age categories   
mydata$agecat <- ifelse(mydata$age > 70,   
c("older"), c("younger"))   
  
# another example: create 3 age categories   
attach(mydata)  
mydata$agecat[age > 75] <- "Elder"  
mydata$agecat[age > 45 & age <= 75] <- "Middle Aged"  
mydata$agecat[age <= 45] <- "Young"  
detach(mydata)

## Renaming variables

You can rename variables programmatically or interactively.

# rename interactively   
fix(mydata) # results are saved on close   
  
# rename programmatically   
library(reshape)  
mydata <- rename(mydata, c(oldname="newname"))  
  
# you can re-enter all the variable names in order  
# changing the ones you need to change.the limitation  
# is that you need to enter all of them!  
names(mydata) <- c("x1","age","y", "ses")

Operators

R's binary and logical operators will look very familiar to programmers. Note that binary operators work on vectors and matrices as well as scalars.

Arithmetic Operators

|  |  |
| --- | --- |
| **Operator** | **Description** |
| **+** | Addition |
| **-** | Subtraction |
| **\*** | Multiplication |
| **/** | Division |
| **^ or \*\*** | Exponentiation |
| **x %% y** | modulus (x mod y) 5%%2 is 1 |
| **x %/% y** | integer division 5%/%2 is 2 |

Logical Operators

|  |  |
| --- | --- |
| **Operator** | **Description** |
| **<** | less than |
| **<=** | less than or equal to |
| **>** | greater than |
| **>=** | greater than or equal to |
| **==** | exactly equal to |
| **!=** | not equal to |
| **!x** | Not x |
| **x | y** | x OR y |
| **x & y** | x AND y |
| **isTRUE(x)** | test if X is TRUE |

# An example   
x <- c(1:10)  
x[(x>8) | (x<5)]  
# yields 1 2 3 4 9 10  
  
# How it works   
x <- c(1:10)  
x  
1 2 3 4 5 6 7 8 9 10  
x > 8  
F F F F F F F F T T  
x < 5  
T T T T F F F F F F  
x > 8 | x < 5  
T T T T F F F F T T  
x[c(T,T,T,T,F,F,F,F,T,T)]  
1 2 3 4 9 10

Built-in Functions

Almost everything in R is done through functions. Here I'm only refering to numeric and character functions that are commonly used in creating or recoding variables.

Numeric Functions

|  |  |
| --- | --- |
| **Function** | **Description** |
| **abs(***x***)** | absolute value |
| **sqrt(***x***)** | square root |
| **ceiling(***x***)** | ceiling(3.475) is 4 |
| **floor(***x***)** | floor(3.475) is 3 |
| **trunc(***x***)** | trunc(5.99) is 5 |
| **round(***x***, digits=***n***)** | round(3.475, digits=2) is 3.48 |
| **signif(***x***, digits=***n***)** | signif(3.475, digits=2) is 3.5 |
| **cos(***x***), sin(***x***), tan(***x***)** | also acos(*x*), cosh(*x*), acosh(*x*), etc. |
| **log(***x***)** | natural logarithm |
| **log10(***x***)** | common logarithm |
| **exp(***x***)** | e^*x* |

Character Functions

|  |  |
| --- | --- |
| **Function** | **Description** |
| **substr(***x***, start=***n1***, stop=***n2***)** | Extract or replace substrings in a character vector. x <- "abcdef"  substr(x, 2, 4) is "bcd"  substr(x, 2, 4) <- "22222" is "a222ef" |
| **grep(***pattern***,***x***, ignore.case=**FALSE**, fixed=**FALSE**)** | Search for *pattern* in *x*. If fixed =FALSE then *pattern* is a regular expression. If fixed=TRUE then *pattern* is a text string. Returns matching indices. grep("A", c("b","A","c"), fixed=TRUE) returns 2 |
| **sub(***pattern***,***replacement***,***x***, ignore.case =**FALSE**, fixed=**FALSE**)** | Find *pattern* in *x* and replace with *replacement* text. If fixed=FALSE then *pattern* is a regular expression. If fixed = T then *pattern* is a text string.  sub("\\s",".","Hello There") returns "Hello.There" |
| **strsplit(***x***,***split***)** | Split the elements of character vector *x* at *split*.  strsplit("abc", "") returns 3 element vector "a","b","c" |
| **paste(..., sep="")** | Concatenate strings after using *sep* string to seperate them. paste("x",1:3,sep="") returns c("x1","x2" "x3") paste("x",1:3,sep="M") returns c("xM1","xM2" "xM3") paste("Today is", date()) |
| **toupper(***x***)** | Uppercase |
| **tolower(***x***)** | Lowercase |

Statistical Probability Functions

The following table describes functions related to probaility distributions. For random number generators below, you can use set.seed(1234) or some other integer to create reproducible pseudo-random numbers.

|  |  |
| --- | --- |
| **Function** | **Description** |
| **dnorm(***x***)** | normal density function (by default m=0 sd=1) # plot standard normal curve x <- pretty(c(-3,3), 30) y <- dnorm(x) plot(x, y, type='l', xlab="Normal Deviate", ylab="Density", yaxs="i") |
| **pnorm(***q***)** | cumulative normal probability for q  (area under the normal curve to the left of q) pnorm(1.96) is 0.975 |
| **qnorm(***p***)** | normal quantile.  value at the p percentile of normal distribution  qnorm(.9) is 1.28 # 90th percentile |
| **rnorm(***n***, m=**0**,sd=**1**)** | n random normal deviates with mean m  and standard deviation sd.  #50 random normal variates with mean=50, sd=10 x <- rnorm(50, m=50, sd=10) |
| **dbinom(***x***,***size***,***prob***) pbinom(***q***,***size***,***prob***) qbinom(***p***,***size***,***prob***) rbinom(***n***,***size***,***prob***)** | binomial distribution where size is the sample size  and prob is the probability of a heads (pi)  # prob of 0 to 5 heads of fair coin out of 10 flips dbinom(0:5, 10, .5)  # prob of 5 or less heads of fair coin out of 10 flips pbinom(5, 10, .5) |
| **dpois(***x***,***lamda***) ppois(***q***,***lamda***) qpois(***p***,***lamda***) rpois(***n***,***lamda***)** | poisson distribution with m=std=lamda #probability of 0,1, or 2 events with lamda=4 dpois(0:2, 4) # probability of at least 3 events with lamda=4  1- ppois(2,4) |
| **dunif(***x***, min=**0**, max=**1**) punif(***q***, min=0, max=**1**) qunif(***p***, min=**0**, max=**1**) runif(***n***, min=**0**, max=**1**)** | uniform distribution, follows the same pattern  as the normal distribution above.  #10 uniform random variates x <- runif(10) |

Other Statistical Functions

Other useful statistical functions are provided in the following table. Each has the option na.rm to strip missing values before calculations. Otherwise the presence of missing values will lead to a missing result. Object can be a numeric vector or data frame.

|  |  |
| --- | --- |
| **Function** | **Description** |
| **mean(***x***, trim=**0**, na.rm=**FALSE**)** | mean of object x # trimmed mean, removing any missing values and  # 5 percent of highest and lowest scores  mx <- mean(x,trim=.05,na.rm=TRUE) |
| **sd(***x***)** | standard deviation of object(x). also look at var(x) for variance and mad(x) for median absolute deviation. |
| **median(***x***)** | median |
| **quantile(***x***,***probs***)** | quantiles where x is the numeric vector whose quantiles are desired and probs is a numeric vector with probabilities in [0,1]. # 30th and 84th percentiles of x y <- quantile(x, c(.3,.84)) |
| **range(***x***)** | range |
| **sum(***x***)** | sum |
| **diff(***x***, lag=***1***)** | lagged differences, with lag indicating which lag to use |
| **min(***x***)** | minimum |
| **max(***x***)** | maximum |
| **scale(***x***, center=**TRUE**, scale=**TRUE**)** | column center or standardize a matrix. |

Other Useful Functions

|  |  |
| --- | --- |
| **Function** | **Description** |
| **seq(***from***,***to***,***by***)** | generate a sequence indices <- seq(1,10,2) #indices is c(1, 3, 5, 7, 9) |
| **rep(***x***,***ntimes***)** | repeat *x* *n* times y <- rep(1:3, 2) # y is c(1, 2, 3, 1, 2, 3) |
| **cut(***x***,***n***)** | divide continuous variable in factor with *n* levels  y <- cut(x, 5) |

# Control Structures

R has the standard control structures you would expect. **expr** can be multiple (compound) statements by enclosing them in braces { }. It is more efficient to use built-in functions rather than control structures whenever possible.

## if-else

if (cond) expr  
if (cond) expr1 else expr2

## for

for (var in seq) expr

## while

while (cond) expr

## switch

switch(expr, ...)

## ifelse

ifelse(test,yes,no)

## Example

# transpose of a matrix  
# a poor alternative to built-in t() function  
  
mytrans <- function(x) {   
  if (!is.matrix(x)) {  
    warning("argument is not a matrix: returning NA")  
    return(NA\_real\_)  
  }  
  y <- matrix(1, nrow=ncol(x), ncol=nrow(x))   
  for (i in 1:nrow(x)) {  
    for (j in 1:ncol(x)) {  
      y[j,i] <- x[i,j]   
    }  
  }  
return(y)  
}  
  
# try it  
z <- matrix(1:10, nrow=5, ncol=2)  
tz <- mytrans(z)

# User-written Functions

One of the great strengths of R is the user's ability to add functions. In fact, many of the functions in R are actually functions of functions. The structure of a function is given below.

myfunction <- function(arg1, arg2, ... ){  
statements  
return(object)  
}

Objects in the function are local to the function. The object returned can be any data type. Here is an example.

# function example - get measures of central tendency  
# and spread for a numeric vector x. The user has a  
# choice of measures and whether the results are printed.  
  
mysummary <- function(x,npar=TRUE,print=TRUE) {  
  if (!npar) {  
    center <- mean(x); spread <- sd(x)   
  } else {  
    center <- median(x); spread <- mad(x)   
  }  
  if (print & !npar) {  
    cat("Mean=", center, "\n", "SD=", spread, "\n")  
  } else if (print & npar) {  
    cat("Median=", center, "\n", "MAD=", spread, "\n")  
  }  
  result <- list(center=center,spread=spread)  
  return(result)  
}  
  
# invoking the function   
set.seed(1234)  
x <- rpois(500, 4)   
y <- mysummary(x)  
Median= 4  
MAD= 1.4826   
# y$center is the median (4)   
# y$spread is the median absolute deviation (1.4826)  
  
y <- mysummary(x, npar=FALSE, print=FALSE)  
# no output   
# y$center is the mean (4.052)  
# y$spread is the standard deviation (2.01927)

It can be instructive to look at the code of a function. In R, you can view a function's code by typing the function name without the ( ).

# Sorting Data

To sort a data frame in R, use the **order( )**function. By default, sorting is ASCENDING. Prepend the sorting variable by a minus sign to indicate DESCENDING order. Here are some examples.

# sorting examples using the mtcars dataset  
attach(mtcars)  
  
# sort by mpg  
newdata <- mtcars[order(mpg),]   
  
# sort by mpg and cyl  
newdata <- mtcars[order(mpg, cyl),]  
  
#sort by mpg (ascending) and cyl (descending)  
newdata <- mtcars[order(mpg, -cyl),]   
  
detach(mtcars)

# Merging Data

## Adding Columns

To merge two data frames (datasets) horizontally, use the **merge** function. In most cases, you join two data frames by one or more common key variables (i.e., an inner join).

# merge two data frames by ID  
total <- merge(data frameA,data frameB,by="ID")

# merge two data frames by ID and Country  
total <- merge(data frameA,data frameB,by=c("ID","Country"))

## Adding Rows

To join two data frames (datasets) vertically, use the**rbind** function. The two data frames **must** have the same variables, but they do not have to be in the same order.

total <- rbind(data frameA, data frameB)

If data frameA has variables that data frameB does not, then either:

1. Delete the extra variables in data frameA or
2. Create the additional variables in data frameB and set them to NA (missing)

before joining them with **rbind( )**.

# Aggregating Data

It is relatively easy to collapse data in R using one or more BY variables and a defined function.

# aggregate data frame mtcars by cyl and vs, returning means  
# for numeric variables  
attach(mtcars)  
aggdata <-aggregate(mtcars, by=list(cyl,vs),   
  FUN=mean, na.rm=TRUE)  
print(aggdata)  
detach(mtcars)

When using the **aggregate()** function, the by variables must be in a list (even if there is only one). The function can be built-in or user provided.

# Reshaping Data

R provides a variety of methods for reshaping data prior to analysis.

## Transpose

Use the t() function to transpose a matrix or a data frame. In the later case, rownames become variable (column) names.

# example using built-in dataset   
mtcars  
t(mtcars)

## The Reshape Package

Hadley Wickham has created a comprehensive package called reshape to massage data. Both an introduction and article are available. There is even a video!

Basically, you "melt" data so that each row is a unique id-variable combination. Then you "cast" the melted data into any shape you would like. Here is a very simple example.

**mydata**

|  |  |  |  |
| --- | --- | --- | --- |
| **id** | **time** | **x1** | **x2** |
| 1 | 1 | 5 | 6 |
| 1 | 2 | 3 | 5 |
| 2 | 1 | 6 | 1 |
| 2 | 2 | 2 | 4 |

# example of melt function   
library(reshape)  
mdata <- melt(mydata, id=c("id","time"))

**newdata**

|  |  |  |  |
| --- | --- | --- | --- |
| **id** | **time** | **variable** | **value** |
| 1 | 1 | x1 | 5 |
| 1 | 2 | x1 | 3 |
| 2 | 1 | x1 | 6 |
| 2 | 2 | x1 | 2 |
| 1 | 1 | x2 | 6 |
| 1 | 2 | x2 | 5 |
| 2 | 1 | x2 | 1 |
| 2 | 2 | x2 | 4 |

# cast the melted data  
# cast(data, formula, function)   
subjmeans <- cast(mdata, id~variable, mean)  
timemeans <- cast(mdata, time~variable, mean)

**subjmeans**

|  |  |  |
| --- | --- | --- |
| **id** | **x1** | **x2** |
| 1 | 4 | 5.5 |
| 2 | 4 | 2.5 |

**timemeans**

|  |  |  |
| --- | --- | --- |
| **time** | **x1** | **x2** |
| 1 | 5.5 | 3.5 |
| 2 | 2.5 | 4.5 |

There is much more that you can do with the **melt( )** and **cast( )** functions. See the documentation for more details.

# Subsetting Data

R has powerful indexing features for accessing object elements. These features can be used to select and exclude variables and observations. The following code snippets demonstrate ways to keep or delete variables and observations and to take random samples from a dataset.

## Selecting (Keeping) Variables

# select variables v1, v2, v3  
myvars <- c("v1", "v2", "v3")  
newdata <- mydata[myvars]  
  
# another method  
myvars <- paste("v", 1:3, sep="")  
newdata <- mydata[myvars]  
  
# select 1st and 5th thru 10th variables  
newdata <- mydata[c(1,5:10)]

To practice this interactively, try the selection of data frame elements exercises in the Data frames chapter of this introduction to R course.

## Excluding (DROPPING) Variables

# exclude variables v1, v2, v3  
myvars <- names(mydata) %in% c("v1", "v2", "v3")   
newdata <- mydata[!myvars]  
  
# exclude 3rd and 5th variable   
newdata <- mydata[c(-3,-5)]  
  
# delete variables v3 and v5  
mydata$v3 <- mydata$v5 <- NULL

## Selecting Observations

# first 5 observations  
newdata <- mydata[1:5,]  
  
# based on variable values  
newdata <- mydata[ which(mydata$gender=='F'   
& mydata$age > 65), ]  
  
# or  
attach(mydata)  
newdata <- mydata[ which(gender=='F' & age > 65),]  
detach(mydata)

## Selection using the Subset Function

The **subset( )**function is the easiest way to select variables and observations. In the following example, we select all rows that have a value of age greater than or equal to 20 or age less then 10. We keep the ID and Weight columns.

# using subset function   
newdata <- subset(mydata, age >= 20 | age < 10,   
select=c(ID, Weight))

In the next example, we select all men over the age of 25 and we keep variables weight through income (weight, income and all columns between them).

# using subset function (part 2)  
newdata <- subset(mydata, sex=="m" & age > 25,  
select=weight:income)

To practice the **subset()** function, try this this interactive exercise. on subsetting data.tables.

## Random Samples

Use the **sample( )** function to take a **random sample of size n** from a dataset.

# take a random sample of size 50 from a dataset mydata   
# sample without replacement  
mysample <- mydata[sample(1:nrow(mydata), 50,  
   replace=FALSE),]

Data Type Conversion

Type conversions in R work as you would expect. For example, adding a character string to a numeric vector converts all the elements in the vector to character.

Use **is.***foo* to test for data type*foo*. Returns TRUE or FALSE  
Use **as.***foo* to explicitly convert it.

**is.numeric(), is.character(), is.vector(), is.matrix(), is.data.frame()  
as.numeric(), as.character(), as.vector(), as.matrix(), as.data.frame)**

Examples

|  |  |  |  |
| --- | --- | --- | --- |
|  | **to one long  vector** | **to matrix** | **to data frame** |
| **from vector** | c(x,y) | cbind(x,y) rbind(x,y) | data.frame(x,y) |
| **from matrix** | as.vector(mymatrix) |  | as.data.frame(mymatrix) |
| **from data frame** |  | as.matrix(myframe) |  |