Introduction To Programming In R

Last updated November 20, 2013

The Institute *for* Quantitative Social Science at Harvard University

Outline

1. Workshop overview and materials
2. Data types
3. Extracting and replacing object elements
4. Applying functions to list elements
5. Writing functions
6. Control flow
7. The S3 object class system
8. Things that may surprise you
9. Additional resources
10. Loops (supplimental)

Topic

1. Workshop overview and materials
2. Data types
3. Extracting and replacing object elements
4. Applying functions to list elements
5. Writing functions
6. Control flow
7. The S3 object class system
8. Things that may surprise you
9. Additional resources
10. Loops (supplimental)

Workshop description

- This is an intermediate/advanced R course
- Appropriate for those with basic knowledge of R
- Learning objectives:
    - Index data objects by position, name or logical condition
    - Understand looping and branching
    - Write your own simple functions
    - Debug functions
    - Understand and use the S3 object system

Running example

Throughout this workshop we will return to a running example that involves calculating descriptive statistics for every column of a data.frame. We will often use the built-in *iris* data set. You can load the iris data by evaluating `data(iris)` at the R prompt.

Our main example today consists of writing a statistical summary function that calculates the min, mean, median, max, sd, and n for all numeric columns in a data.frame, the correlations among these variables, and the counts and proportions for all categorical columns. Typically I will describe a topic and give some generic examples, then ask you to use the technique to start building the summary.

Topic

1. Workshop overview and materials
2. Data types
3. Extracting and replacing object elements
4. Applying functions to list elements
5. Writing functions
6. Control flow
7. The S3 object class system
8. Things that may surprise you
9. Additional resources
10. Loops (supplimental)

Vectors and data classes

Values can be combined into vectors using the c() function

```
> num.var <- c(1, 2, 3, 4) # numeric vector
> char.var <- c("1", "2", "3", "4") # character vector
> log.var <- c(TRUE, TRUE, FALSE, TRUE) # logical vector
> char.var2 <- c(num.var, char.var) # numbers coverted to character
>
```

Vectors have a *class* which determines how functions treat them

```
> class(num.var)
[1] "numeric"
> mean(num.var) # take the mean of a numeric vector
[1] 2.5
> class(char.var)
[1] "character"
> mean(char.var) # cannot average characters
[1] NA
> class(char.var2)
[1] "character"
```

Vector conversion and info

Vectors can be converted from one class to another

```
> class(char.var2)
[1] "character"
> num.var2 <- as.numeric(char.var2) # convert to numeric
> class(num.var2)
[1] "numeric"
> mean(as.numeric(char.var2)) # now we can calculate the mean
[1] 2.5
> as.numeric(c("a", "b", "c")) # cannot convert letters to numeric
[1] NA NA NA
```

In addition to class, you can examine the `length()` and `str()` ucture of vectors

```
> ls() # list objects in our workspace
[1] "char.var"  "char.var2" "log.var"   "num.var"   "num.var2"
[6] "tmp"
> length(char.var) # how many elements in char.var?
[1] 4
> str(num.var2) # what is the structure of num.var2?
 num [1:8] 1 2 3 4 1 2 3 4
```

Factor vectors

Factors are stored as numbers, but have character labels. Factors are useful for

- Modeling (automatically contrast coded)
- Sorting/presenting values in arbitrary order

Most of the time we can treat factors as though they were character vectors

---

Lists and data.frames

- A *data.frame* is a list of vectors, each of the same length
- A *list* is a collection of objects each of which can be almost anything

```
> DF <- data.frame(x=1:5, y=letters[1:5])
> DF # data.frame with two columns and 5 rows
  x y
1 1 a
2 2 b
3 3 c
4 4 d
5 5 e
>
> # DF <- data.frame(x=1:10, y=1:7) # illegal becase lengths differ
> L <- list(x=1:5, y=1:3, z = DF)
> L # lists are much more flexible!
$x
[1] 1 2 3 4 5

$y
[1] 1 2 3

$z
  x y
1 1 a
```

Data types summary

Key points:

- vector classes include numeric, logical, character, and factors
- vectors can be combined into lists or data.frames
- a data.frame can almost always be thought of as a list of vectors of equal length
- a list is a collection of objects, each of which can by of almost any type

Functions introduced in this section:

c	combine elements
as.numeric	convert an object (e.g., a character verctor) to numeric
data.frame	combine oject into a data.frame
ls	list the objects in the workspace
class	get the class of an object
str	get the structure of an object
length	get the number of elements in an object
mean	calculate the mean of a vector

Exercise 0

1. Create a new vector called "test" containing five numbers of your choice [`c()`, `<-`]
2. Create a second vector called "students" containing five common names of your choice [`c()`, `<-`]
3. Determine the class of "students" and "test" [`class()` or `str()`]
4. Create a data frame containing two columns, "students" and "tests" as defined above [`data.frame`]
5. Convert "test" to character class, and confirm that you were successful [`as.numeric()`, `<-`, `str()`]

Topic

1. Workshop overview and materials
2. Data types
3. Extracting and replacing object elements
4. Applying functions to list elements
5. Writing functions
6. Control flow
7. The S3 object class system
8. Things that may surprise you
9. Additional resources
10. Loops (supplimental)

Indexing by position or name

Parts of vectors, matricies, data.frames, and lists can be extracted or replaced based on position or name

```
> ## indexing vectors by position
> x <- 101:110 # Creat a vector of integers from 101 to 110
> x[c(4, 5)] # extract the fourth and fifth values of x
[1] 104 105
> x[4] <- 1 # change the 4th value to 1
> x # print x
 [1] 101 102 103   1 105 106 107 108 109 110
>
> ## indexing vectors by name
> names(x) <- letters[1:10] # give x names
> print(x) #print x
  a   b   c   d   e   f   g   h   i   j
101 102 103   1 105 106 107 108 109 110
> x[c("a", "f")] # extract the values of a  and f from x
  a   f
101 106
```

Logical indexing

Elements can also be selected or replaced based on logical (TRUE/FALSE) vectors.

```
> x > 106 # shows which elements of x are > 106
    a     b     c     d     e     f     g     h     i     j
FALSE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE
> x[x > 106] # selects elements of x where x > 106
  g   h   i   j
107 108 109 110
```

Additional operators useful for logical indexing:

==	equal to
!=	not equal to
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to
%in%	is included in
&	and
|	or

```
> x[x > 106 & x <= 108]
```

Indexing matrices

Extraction on matrices operate in two dimensions: first dimension refers to rows, second dimension refers to columns

```
> ## indexing matricies
>  # create a matrix
> (M <- cbind(x = 1:5, y = -1:-5, z = c(6, 3, 4, 2, 8)))
     x  y z
[1,] 1 -1 6
[2,] 2 -2 3
[3,] 3 -3 4
[4,] 4 -4 2
[5,] 5 -5 8
> M[1:3, ] #extract rows 1 through 3, all columns
     x  y z
[1,] 1 -1 6
[2,] 2 -2 3
[3,] 3 -3 4
> M[c(5, 3, 1), 2:3] # rows 5, 3 and 1, columns 2 and 3
      y z
[1,] -5 8
[2,] -3 4
[3,] -1 6
> M[M[, 1] %in% 4:2, 2] # second column where first column <=4 & >= 2
[1] -2 -3 -4
```

Indexing lists

Lists can be indexed in the same way as vectors, with the following extension:

```
> # Lists can be indexed with single brackets, similar to vector indexing
> L[c(1, 2)] # the first two elements of L
$x
[1] 1 2 3 4 5

$y
[1] 1 2 3

> L[1] # a list with one element
$x
[1] 1 2 3 4 5

> ## double brackets select the content of a single selected element
> ## effectively taking it out of the list.
> L[[1]] # a vector
[1] 1 2 3 4 5
```

Indexing data.frames

A data.frame can be indexed in the same ways as a matrix, and also the same ways as a list:

```
> DF[c(3, 1, 2), c(1, 2)] # rows 3, 1, and 2, columns 1 and 2
  x y
3 3 c
1 1 a
2 2 b
> DF[[1]] # column 1 as a vector
[1] 1 2 3 4 5
```

There is a subtle but important difference between `[ , n]` and `[n]` when indexing data.frames: the first form returns a vector, the second returns a data.frame with one column.

```
> str(DF[1])# a data.frame with one column
'data.frame': 5 obs. of  1 variable:
 $ x: int  1 2 3 4 5
> str(DF[ ,1])# a vector
 int [1:5] 1 2 3 4 5
```

Extraction/replacement summary

Key points:

- elements of objects can be extracted or replaced using the `[` operator
- objects can be indexed by position, name, or logical (TRUE/FALSE) vectors
- vectors and lists have only one dimension, and hence only one index is used
- matricies and data.frames have two dimensions, and extraction methods for these objects use two indices

Functions introduced in this section:

[	extraction operator, used to extract/replace object elements
names	get the names of an object, usually a vector, list, or data.frame
print	print an object

Exercise 1

1. Select just the Sepal.Length and Species columns from the *iris* data set (built-in, will be available in your workspace automatically) and save the result to a new data.frame named *iris.2*
2. Calculate the mean of the Sepal.Length column in *iris.2*
3. BONUS (optional): Calculate the mean of Sepal.Length, but only for the setosa species
4. BONUS (optional): Calculate the number of sepal lengths that are more than one standard deviation below the average sepal length

Topic

1. Workshop overview and materials
2. Data types
3. Extracting and replacing object elements
4. Applying functions to list elements
5. Writing functions
6. Control flow
7. The S3 object class system
8. Things that may surprise you
9. Additional resources
10. Loops (supplimental)

The apply function

The apply function is used to apply a function to the rows or columns of a matrix

```
> M <- matrix(1:20, ncol=4)
> apply(M, 2, mean) ## average across the rows
[1]  3  8 13 18
> apply(M, 2, sum) ## sum the columns
[1] 15 40 65 90
```

The sapply function

It is often useful to apply a function to each element of a vector, list, or data.frame; use the sapply function for this

```
> sapply(DF, class) # get the class of each column in the DF data.frame
        x          y
"integer"  "factor"
> sapply(L, length) # get the length of each element in the L list
x y z
5 3 2
> sapply(DF, is.numeric) # check each column of DF to see if it is numeric
    x     y
 TRUE FALSE
```

Combining sapply and indexing

The `sapply` function can be used in combination with indexing to extract elements that meet certain criteria

- Recall that we can index using logical vectors:

```
> DF[, c(TRUE, FALSE)] # select the first column of DF, but not the second
[1] 1 2 3 4 5
```

```
> ## recall that we can index using logical vectors:
> DF[, c(TRUE, FALSE)] # select the first column of DF, but not the second
[1] 1 2 3 4 5
```

- `sapply()` can be used to generate the logical vector

```
> (DF.which.num <- sapply(DF, is.numeric))# check which columns of DF are numer
    x     y
 TRUE FALSE
> DF[DF.which.num] # select the numeric columns
  x
1 1
2 2
3 3
4 4
5 5
```

Applying functions summary

Key points:

- R has convenient methods for applying functions to matricies, lists, and data.frames
- other apply-style functions exist, e.g., lapply, tapply, and mapply (see documentation of these functions for details

Functions introduced in this section:

matrix	create a matrix (vector with two dimensions)
apply	apply a function to the rows or columns of a matrix
sapply	apply a function to the elements of a list
is.numeric	returns TRUE or FALSE, depending on the type of object

Topic

1. Workshop overview and materials
2. Data types
3. Extracting and replacing object elements
4. Applying functions to list elements
5. Writing functions
6. Control flow
7. The S3 object class system
8. Things that may surprise you
9. Additional resources
10. Loops (supplimental)

Functions

- A function is a collection of commands that takes input(s) and returns output.
- If you have a specific analysis or transformation you want to do on different data, use a function
- Functions are defined using the `function()` function
- Functions can be defined with any number of named arguments
- Arguments can be of any type (e.g., vectors, data.frames, lists . . .)

Function return value

- The return value of a function can be:
  - The last object stored in the body of the function
  - Objects explicitly returned with the `return()` function
- Other function output can come from:
  - Calls to `print()`, `message()` or `cat()` in function body
  - Error messages
- Assignment inside the body of a function takes place in a local environment
- Example:

```
> f <- function() { # define function f
+ print("setting x to 1") # print a text string
+ x <- 1} # set x to 1
>
> y <- f() # assign y the value returned by f
[1] "setting x to 1"
>
> y # print y
[1] 1
> x # x in the global is not 1!
  a   b   c   d   e   f   g   h   i   j
101 102 103   1 105 106 107 108 109 110
```

Writing functions example

Goal: write a function that returns the square of it's argument

```
> square <- function (x) { # define function named "square" with argument x
+   return(x*x) # multiple the x argument by itself
+ } # end the function definition
>
> # check to see that the function works
> square(x = 2) # square the value 2
[1] 4
> square(10) # square the value 10
[1] 100
> square(1:5) # square integers 1 through 5
[1]  1  4  9 16 25
```

Debugging basics

Stepping througth functions and setting breakpoints

```
> ## Debugging
> # write my.mean function
> my.mean <- function (x, ...) {
+   S <- sum(x, ...)
+   L <- length(na.omit(x))
+   return(S / L)}
> debug(my.mean) # turn debugger on for my.mean function
> # mymean() # step through the function
> undebug(my.mean) # to turn the debugger off
> # insert breakpoints
> my.mean <- function (x, ...) {
+   S <- sum(x, ...)
+   L <- length(na.omit(x))
+   browser() # function will stop here so you can inspect S and L
+   return(S / L)}
>
```

Use `traceback()` to see what went wrong after the fact

```
myModel <- lm(NA~NA)
traceback()
```

Writing functions summary

Key points:

- writing new functions is easy!
- most functions will have a return value, but functions can also print things, write things to file etc.
- functions can be stepped through to facilitate debugging

Functions introduced in this section

function	defines a new function
return	used inside a function definition to set the return value
browser	sets a break point
debug	turns on the debugging flag of a function so you can step through it
undebug	turns off the debugging flag
traceback	shows the error stack (call after an error to see what went wrong)

Exercise 2

1. Write a function that takes a data.frame as an argument and returns the mean of each numeric column in the data frame. Test your function using the iris data.
2. Modify your function so that it returns a list, the first element if which is the means of the numeric variables, the second of which is the counts of the levels of each categorical variable.

Topic

1. Workshop overview and materials
2. Data types
3. Extracting and replacing object elements
4. Applying functions to list elements
5. Writing functions
6. Control flow
7. The S3 object class system
8. Things that may surprise you
9. Additional resources
10. Loops (supplimental)

Control flow

- Basic idea: if some condition is true, do one thing. If false, do something else
- Carried out in R using `if()` and `else()` statements, which can be nested if necessary
- Especially useful for checking function arguments and performing different operations depending on function input

---

Control flow examples

Goal: write a function that tells us if a number is positive or negative

```
> ## use branching to return different result depending on the sign of the inpu
> isPositive <- function(x) { # define function "isPositive"
+   if (x > 0) { # if x is greater than zero, then
+     cat(x, "is positive \n") } # say so!
+   else { # otherwise
+       cat(x, "is negative \n")} # say x is negative
+ } # end function definition
>
> ## test the isPositive() function
> isPositive(10)
10 is positive
> isPositive(-1)
-1 is negative
> isPositive(0)
0 is negative
```

Need to do something different if x equals zero!

Control flow examples

Add a condition to handle `x = 0`

```
> ## add condition to handle the case that x is zero
>   isPositive <- function(x) { # define function "isPositive"
+     if (x > 0) { # if x is greater than zero, then
+       cat(x, "is positive \n") } # say so!
+     else if (x == 0) { # otherwise if x is zero
+       cat(x, "is zero \n")} # say so!
+     else { #otherwise
+         cat(x, "is negative \n")} # say x is negative
+   } # end function definition
>
```

Test the new function

```
> isPositive(0) # test the isPositive() function
0 is zero
> isPositive("a") #oops, that will not work!
a is positive
```

We fixed the problem when `x = 0`, but now we need to make sure x is numeric of length one (unless we agree with R that a is positive!)

Control flow examples

Do something reasonable if x is not numeric

```
> ## add condition to handle the case that x is zero
>   isPositive <- function(x) { # define function "isPositive"
+     if(!is.numeric(x) | length(x) > 1) {
+       cat("x must be a numeric vector of length one! \n")}
+     else if (x > 0) { # if x is greater than zero, then
+       cat(x," is positive \n") } # say so!
+     else if (x == 0) { # otherwise if x is zero
+       cat(x," is zero \n")} # say so!
+     else { #otherwise
+       cat(x," is negative \n")} # say x is negative
+   } # end function definition
>
> isPositive("a") # test the isPositive() function on character
x must be a numeric vector of length one!
```

Control flow summary

Key points:

- code can be conditionally executed
- conditions can be nested
- conditional execution is often used for argument checking, among other things

Functions introduced in this section

cat	Concatenates and prints R objects
if	execute code only if condition is met
else	used with if; code to execute if condition is not met

Exercise 3

1. Add argument checking code to return an error if the argument to your function is not a data.frame
2. Insert a break point with `browser()` and step through your function

Topic

1. Workshop overview and materials
2. Data types
3. Extracting and replacing object elements
4. Applying functions to list elements
5. Writing functions
6. Control flow
7. The S3 object class system
8. Things that may surprise you
9. Additional resources
10. Loops (supplimental)

The S3 object class system

R has two major object systems:

- Relatively informal "S3" classes
- Stricter, more formal "S4" classes
- We will cover only the S3 system, not the S4 system
- Basic idea: functions have different methods for different types of objects

Object class

The class of an object can be retrieved and modified using the `class()` function:

```
> x <- 1:10
> class(x)
[1] "integer"
> class(x) <- "foo"
> class(x)
[1] "foo"
```

Objects are not limited to a single class, and can have many classes:

```
> class(x) <- c("A", "B")
> class(x)
[1] "A" "B"
```

Function methods

- Functions can have many methods, allowing us to have (e.g.) one plot() function that does different things depending on what is being plotted()
- Methods can only be defined for generic functions: plot, print, summary, mean, and several others are already generic

```
> # see what methods have been defined for the mean function
> methods(mean)
[1] mean.Date     mean.default  mean.difftime mean.POSIXct
[5] mean.POSIXlt
> # which functions have methods for data.frames?
> methods(class="data.frame")[1:9]
[1] "[.data.frame"              "[[.data.frame"
[3] "[[<-.data.frame"           "[<-.data.frame"
[5] "$<-.data.frame"            "aggregate.data.frame"
[7] "anyDuplicated.data.frame" "as.data.frame.data.frame"
[9] "as.list.data.frame"
```

Creating new function methods

To create a new method for a function that is already generic all you have to do is name your function `function.class`

```
> # create a mean() method for objects of class "foo":
> mean.foo <- function(x) { # mean method for "foo" class
+   if(is.numeric(x)) {
+     cat("The average is", mean.default(x))
+     return(invisible(mean.default(x))) #use mean.default for numeric
+   } else
+     cat("x is not numeric \n")} # otherwise say x not numeric
>
> x <- 1:10
> mean(x)
[1] 5.5
> class(x) <- "foo"
> mean(x)
The average is 5.5>
> x <- as.character(x)
> class(x) <- "foo"
> mean(x)
x is not numeric
```

Creating generic functions

S3 generics are most often used for print, summary, and plot methods, but sometimes you may want to create a new generic function

```
> # create a generic disp() function
> disp <- function(x, ...) {
+   UseMethod("disp")
+ }
>
> # create a disp method for class "matrix"
> disp.matrix <- function(x) {
+   print(round(x, digits=2))
+ }
>
> # test it out
> disp(matrix(runif(10), ncol=2))
     [,1] [,2]
[1,] 0.45 0.79
[2,] 0.96 0.36
[3,] 0.14 0.06
[4,] 0.78 0.57
[5,] 0.80 0.66
```

S3 classes summary

Key points:

- there are several class systems in R, of which S3 is the oldest and simplest
- objects have *class* and functions have corresponding *methods*
- the class of an object can be set by simple assignment
- S3 generic functions all contain `UseMethod("x")` in the body, where `x` is the name of the function
- new methods for existing generic functions can be written simply by defining a new function with a special naming scheme: the name of the function followed by dot followed by the name of the class

Functions introduced in this section

plot	creates a graphical display, the type of which depends on the class of the object being plotted
methods	lists the methods defined for a function or class
UseMethod	the body of a generic function
invisible	returns an object but does not print it

Exercise 4

1. Modify your function so that it also returns the standard deviations of the numeric variables
2. Modify your function so that it returns a list of class "statsum"
3. Write a print method for the statsum class

Topic

1. Workshop overview and materials
2. Data types
3. Extracting and replacing object elements
4. Applying functions to list elements
5. Writing functions
6. Control flow
7. The S3 object class system
8. **Things that may surprise you**
9. Additional resources
10. Loops (supplimental)

Gotcha's

- There are an unfortunately large number of surprises in R programming
- Some of these "gotcha's" are common problems in other languages, many are unique to R
- We will only cover a few – for a more comprehensive discussion please see `http://www.burns-stat.com/pages/Tutor/R_inferno.pdf`

Floating point comparison

Floating point arithmetic is not exact:

```
> .1 == .3/3
[1] FALSE
```

Solution: use `all.equal()`:

```
> all.equal(.1, .3/3)
[1] TRUE
```

Missing values

R does not exclude missing values by default – a single missing value in a vector means that many thing are unknown:

```
> x <- c(1:10, NA, 12:20)
> c(mean(x), sd(x), median(x), min(x), sd(x))
[1] NA NA NA NA NA
```

NA is not equal to anything, not even NA

```
> NA == NA
[1] NA
```

Solutions: use `na.rm = TRUE` option when calculating, and is.na to test for missing

---

Automatic type conversion

Automatic type conversion happens a lot which is often useful, but makes it easy to miss mistakes

```
> # combining values coereces them to the most general type
> (x <- c(TRUE, FALSE, 1, 2, "a", "b"))
[1] "TRUE"  "FALSE" "1"     "2"     "a"     "b"
> str(x)
 chr [1:6] "TRUE" "FALSE" "1" "2" "a" "b"
>
> # comparisons convert arguments to most general type
> 1 > "a"
[1] FALSE
```

Maybe this is what you expect... I would like to at least get a warning!

Optional argument inconsistencies

Functions you might expect to work similarly don't always:

```
> mean(1, 2, 3, 4, 5)*5
[1] 5
> sum(1, 2, 3, 4, 5)
[1] 15
```

Why are these different?!?

```
> args(mean)
function (x, ...)
NULL
> args(sum)
function (..., na.rm = FALSE)
NULL
```

Ouch. That is not nice at all!

Trouble with Factors

Factors sometimes behave as numbers, and sometimes as characters, which can be confusing!

```
> (x <- factor(c(5, 5, 6, 6), levels = c(6, 5)))
[1] 5 5 6 6
Levels: 6 5
>
> str(x)
 Factor w/ 2 levels "6","5": 2 2 1 1
>
> as.character(x)
[1] "5" "5" "6" "6"
> # here is where people sometimes get lost...
> as.numeric(x)
[1] 2 2 1 1
> # you probably want
> as.numeric(as.character(x))
[1] 5 5 6 6
```

Topic

1. Workshop overview and materials
2. Data types
3. Extracting and replacing object elements
4. Applying functions to list elements
5. Writing functions
6. Control flow
7. The S3 object class system
8. Things that may surprise you
9. **Additional resources**
10. Loops (supplimental)

Additional reading and resources

- S3 system overview:
  https://github.com/hadley/devtools/wiki/S3
- S4 system overview:
  https://github.com/hadley/devtools/wiki/S4
- R documentation: http://cran.r-project.org/manuals.html
- Collection of R tutorials:
  http://cran.r-project.org/other-docs.html
- R for Programmers (by Norman Matloff, UC–Davis)

http://heather.cs.ucdavis.edu/~matloff/R/RProg.pdf

- Calling C and Fortran from R (by Charles Geyer, UMinn)

http://www.stat.umn.edu/~charlie/rc/

- State of the Art in Parallel Computing with R (Schmidberger et al.)

http://www.jstatso|.org/v31/i01/paper

- Institute for Quantitative Social Science: http://iq.harvard.edu
- Research technology consulting:
  http://projects.iq.harvard.edu/rtc

Feedback

- Help Us Make This Workshop Better!
- Please take a moment to fill out a very short feedback form
- These workshops exist for you – tell us what you need!
- `http://tinyurl.com/RprogrammingFeedback`

Topic

1. Workshop overview and materials
2. Data types
3. Extracting and replacing object elements
4. Applying functions to list elements
5. Writing functions
6. Control flow
7. The S3 object class system
8. Things that may surprise you
9. Additional resources
10. Loops (supplimental)

Looping

- A loop is a collection of commands that are run over and over again.
- A for loop runs the code a fixed number of times, or on a fixed set of objects.
- A while loop runs the code until a condition is met.
- If you're typing the same commands over and over again, you might want to use a loop!

Looping: for-loop examples

For each value in a vector, print the number and its square

```
> # For-loop example
> for (num in seq(-5,5)) {# for each number in [-5, 5]
+   cat(num, "squared is", num^2, "\n") # print the number
+ }
-5 squared is 25
-4 squared is 16
-3 squared is 9
-2 squared is 4
-1 squared is 1
0 squared is 0
1 squared is 1
2 squared is 4
3 squared is 9
4 squared is 16
5 squared is 25
```

Looping: while-loop example

Goal: simulate rolling two dice until we roll two sixes

```
> ## While-loop example: rolling dice
> set.seed(15) # allows repoducible sample() results
> dice <- seq(1,6) # set dice = [1 2 3 4 5 6]
> roll <- 0 # set roll = 0
> while (roll < 12) {
+   roll <- sample(dice,1) + sample(dice,1) # calculate sum of two rolls
+   cat("We rolled a ", roll, "\n") # print the result
+ } # end the loop
We rolled a  6
We rolled a  10
We rolled a  9
We rolled a  7
We rolled a  10
We rolled a  5
We rolled a  9
We rolled a  12
```

Using loops to fill in lists

Often you will want to store the results from a loop. You can create an object to hold the results generated in the loop and fill in the values using indexing

```
> ## save calculations done in a loop
> Result <- list() # create an object to store the results
>   for (i in 1:5) {# for each i in [1, 5]
+     Result[[i]] <- 1:i ## assign the sequence 1 to i to Result
+ }
> Result # print Result
[[1]]
[1] 1

[[2]]
[1] 1 2

[[3]]
[1] 1 2 3

[[4]]
[1] 1 2 3 4

[[5]]
[1] 1 2 3 4 5
```

Word of caution: don't overuse loops!

Most operations in R are vectorized – This makes loops unnecessary in many cases

- Use vector arithmatic instead of loops:

```
> x <- c() # create vector x
> for(i in 1:5) x[i] <- i+i # double a vector using a loop
> print(x) # print the result
[1]  2  4  6  8 10
>
> 1:5 + 1:5 #double a vector without a loop
[1]  2  4  6  8 10
> 1:5 + 5 # shorter vectors are recycled
[1]  6  7  8  9 10
```

- Use paste instead of loops:

```
> ## Earlier we said
>   ## for (num in seq(-5,5)) {# for each number in [-5, 5]
>   ##   cat(num, "squared is", num^2, "\n") # print the number
>   ## }
> ## a better way:
> paste(1:5, "squared = ", (1:5)^2)
[1] "1 squared =  1"  "2 squared =  4"  "3 squared =  9"
[4] "4 squared =  16" "5 squared =  25"
```

Exercise 5

1. use a loop to get the `class()` of each column in the iris data set
2. use the results from step 1 to select the numeric columns
3. use a loop to calculate the mean of each numeric column in the iris data