Statistics with R

Introduction and Examples

Scott Hetzel
University of Wisconsin – Madison
Summer Institute for Training in Biostatistics (2009)

Derived from: "Introductory Statistics with R" by: Peter Dalgaard

and from previous notes by Deepayan Sarkar, Ph.D

What is R?

- *From the text*: ***R*** *provides an environment in which you can perform statistical analysis and produce graphics. It is actually a complete programming language, although that is only marginally described in this book.*

- We will mostly use **R** as a toolbox for standard statistical techniques.

- Some knowledge of **R** programming is essential to use it well.

More information available at the R Project homepage:
http://www.r-project.org

More Background

- **R** is often referred to as a dialet of the S language
- **R** was used as a name because it is considered an offspring of S
- S was developed at the AT & T Bell Laboratories by John Chambers and his colleagues doing research in statistical computing, beginning in the late 1970's
- The original S implementation is used in the commercially available software S-PLUS
- **R** is an open source implementation developed independently, starting in the early 1990's
- Mostly similar, but there are differences as well

Our Goal for the First Two Weeks

. . . is basically to get comfortable using **R**. We will learn

- to do some elementary statistics

- to use the documentation / help system

- about the language basics

- about data manipulation

We will basically go through most of the first chapter of “Introductory Statistics with R” by Peter Dalgaard. We will learn about other specialized statistical tools a little later.

Interacting with R

R usually works interactively, using a question-and-answer model:

- Start **R**
- Type a command at the "`>`" prompt and press Enter
- **R** executes this command (**R** often prints the result)
- **R** then waits for more input
- Type `q()` to exit

Simple Examples

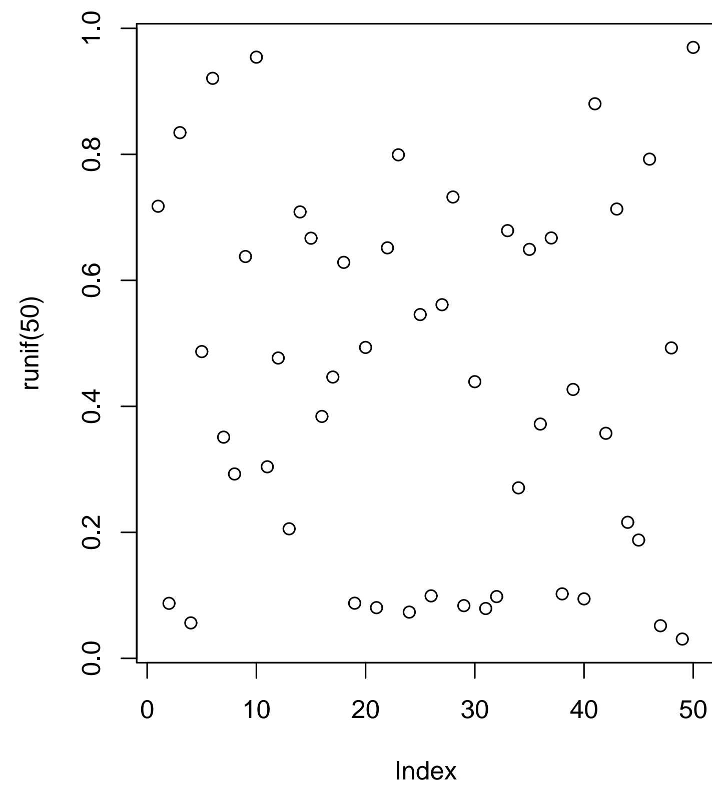
```
> 2 + 2
[1] 4
> exp(-2)
[1] 0.1353353
> log(100, base=10)
[1] 2
> runif(10)
[1] 0.59155 0.45356 0.53916 0.70938 0.79729
[6] 0.04349 0.47939 0.58595 0.37206 0.79621
```

The last command generates 10 random numbers between 0 and 1. The printed result is a vector of 10 numbers. The bracketed numbers at the beginning of each line indicate the index of the first number on that line.

`exp`, `log`, and `runif` are functions, indicated by the prescence of parentheses. Most useful things in **R** are done by functions.

Simple Examples: plotting

```
> plot(runif(50))
```



Variables

Like most programming languages, **R** has symbolic variables which can be assigned values. The traditional way to do this in **R** is the '`<-`' operator, '`=`' also works.

```
> x <- 2
> yVar2 = x + 3
> s <- ``this is a character string''
> x
[1] 2
> yVar2
[1] 5
> s
[1] "this is a character string"
> x + x
[1] 4
> x^yVar2
[1] 32
```

Variables (Cont.)

Possible variable names are very flexible. However, note that:

- Variable names cannot start with a number
- Names **are** case-sensitive
- Some common names are already used by **R**, e.g., `c, q, t, C, D, F, I`, and `T`, should be avoided
- It is recommended that a variable name be short and meaningful. One letter names are generally not useful because the meaning can be lost easily.

Vectorized Arithmetic

- The elementary data types in **R** are all vectors

- The `c(...)` construct can be used to create vectors:

```
> weight <- c(60,72,57,90,95,72)
> weight
[1] 60 72 57 90 95 72
```

- To generate a vector of regularly spaced numbers, use:

```
> seq(0,1,length=11)
[1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
> 1:10
[1] 1 2 3 4 5 6 7 8 9 10
```

- Common arithmetic operations (including +, -, *, /, ^) and mathematical functions (e.g. `sin`, `cos`, `log`) work elementwise on vectors, and produce another vector

Vectorized Arithmetic (Cont.)

```
> height <- c(1.75, 1.8, 1.65, 1.9, 1.74, 1.91)
> height^2
[1] 3.0625 3.2400 2.7225 3.6100 3.0276 3.6481
> bmi <- weight/height^2
> bmi
[1] 19.59184 22.22222 20.93664 24.93075 31.37799 19.73630
> log(bmi)
[1] 2.975113 3.101093 3.041501 3.216102 3.446107 2.982460
```

When two vectors are not of equal length, the shorter one is recycled. The following adds 0 to all the odd elements and 2 to all the even elements of 1:10.

```
> 1:10 + c(0, 2)
[1] 1 4 3 6 5 8 7 10 9 12
```

Scalars from Vectors

Many functions summarize a data vector by producing a scalar. For example:

```
> sum(weight)
[1] 446
> length(weight)
[1] 6
```

Say you want to save these values to refer to them later:

```
> TotSum <- sum(weight)
> NumObs <- length(weight)
```

Then find the average using the variable names:

```
> TotSum/NumObs
[1] 74.33333
```

But **R** is a statistical program so finding the average and other descriptive statistics all have their own functions.

Descriptive Statistics

Simple summary statistics: mean, median, standard deviation, and variance can be found using the functions:
`mean(), median(), sd(), var()`

```
> x <- rnorm(100) # Creates 100 random numbers from Standard Normal
> mean(x)
[1] 0.2001412
> median(x)
[1] 0.2922312
> sd(x)
[1] 1.042742
> var(x)
[1] 1.087311
```

Descriptive Statistics (Cont.)

Simple summary statistics: quantiles and inter-quartile range can be found using the functions: `quantile()`, and `IQR()`

```
> xquants <- quantile(x)
> xquants
      0%       25%       50%       75%      100%
-2.76089  -0.46857   0.29223   0.92658   3.04009
> xquants[4] - xquants[2] # Uses indexing of vector xquants
     75%
 1.39515
> IQR(x)
[1] 1.395158
> quantile(x, probs = seq(0, 1, length=11))
      0%       10%       20%       30%       40%       50%
-2.76089  -1.12359  -0.68802  -0.21753   0.09099   0.29223
     60%       70%       80%       90%      100%
 0.51554   0.74467   0.98356   1.42227   3.04009
```

The `summary` Function

When applied to a numeric vector, `summary` produces a nice summary display of the descriptive statistics we just discussed.

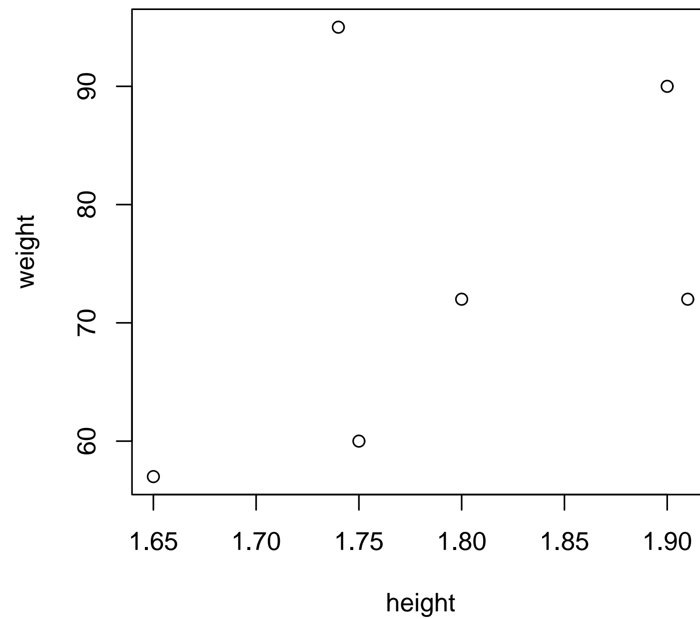
```
> summary(x)
   Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
-2.7610  -0.4686  0.2922  0.2001   0.9266  3.0400
```

The output of `summary` can be different when applied to other objects. We will discuss this when it arises later.

Graphics

The simplest way to produce **R** graphics output is to use the `plot()` function:

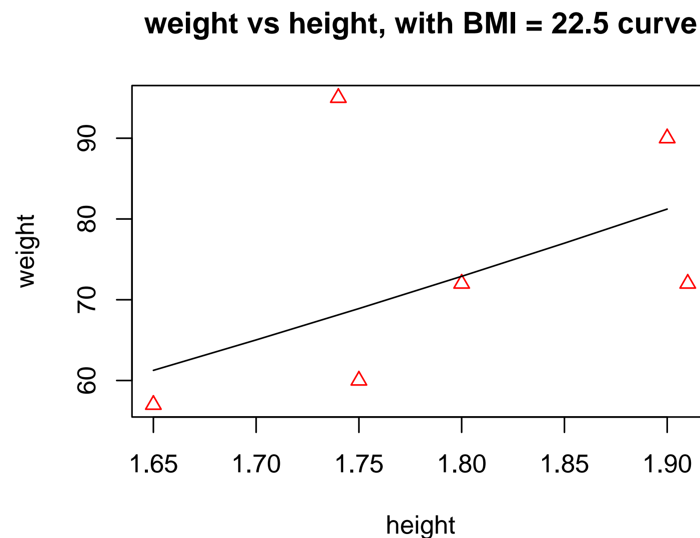
```
> plot(x = height, y=weight)
```



Graphics (Cont.)

There are many options that can control the details of what the plot looks like. Once created, plots can also be enhanced:

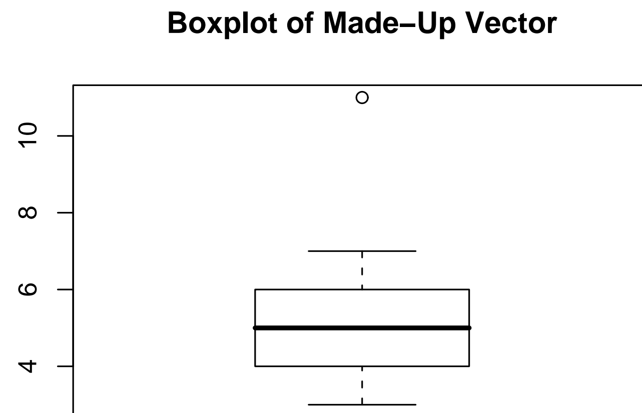
```
> plot(x = height, y = weight, pch = 2, col = ''red'')
> hh <- c(1.65, 1.7, 1.75, 1.8, 1.85, 1.9)
> lines(x = hh, y = 22.5*hh^2)
> title(main = ''weight vs height, with BMI = 22.5 curve'')
```



Graphics (Cont.)

Box plots can be used to summarize continuous variables. The line in the middle shows the median of the values. The edges of the box roughly show the quartiles and the ”whiskers” show the largest/smallest observation within 1.5 times the width of the box away from each edge of the box. Any dots outside the whiskers could be considered outliers. The function in **R** is simply `boxplot()`

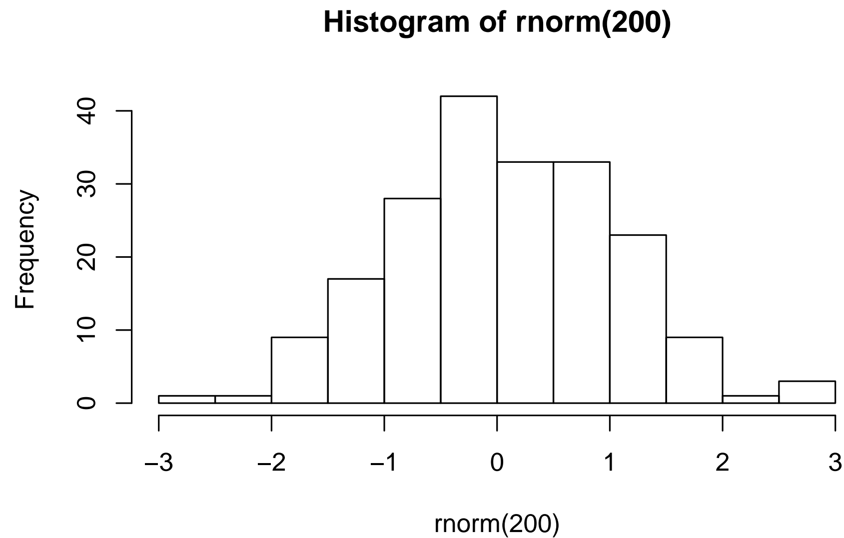
```
> boxplot(c(3,4,5,4,3,5,6,7,11), main="Boxplot of Made-Up Vector")
```



Histograms

Basic distributional shape of a variable (vector of data) is useful to get general information about the variable. The most popular graphical summary for numeric data is the histogram:

```
> hist(rnorm(200))
```

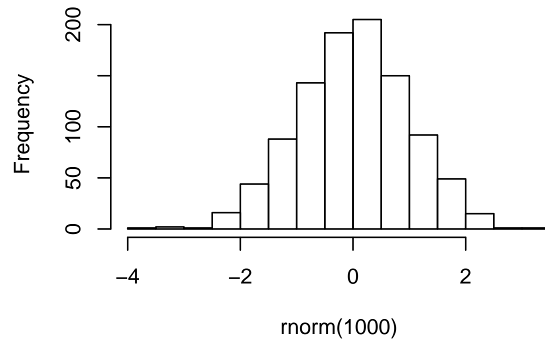


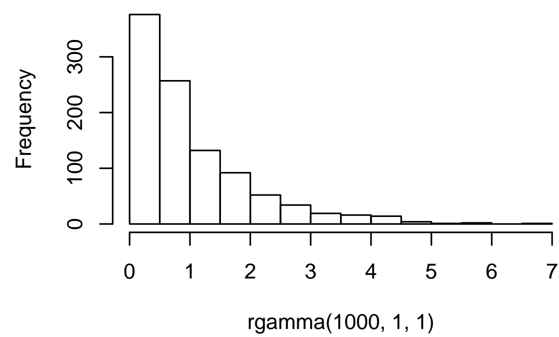
Example of Histograms

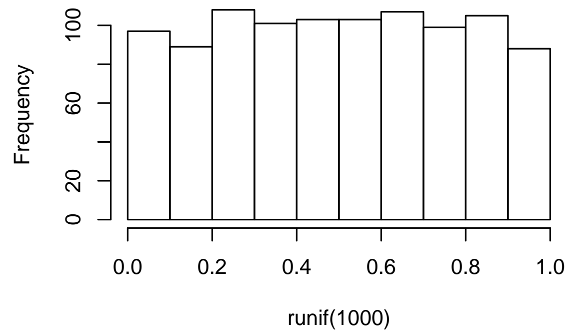
Some well known distributions have distinctive shapes, which can be seen in **R**.

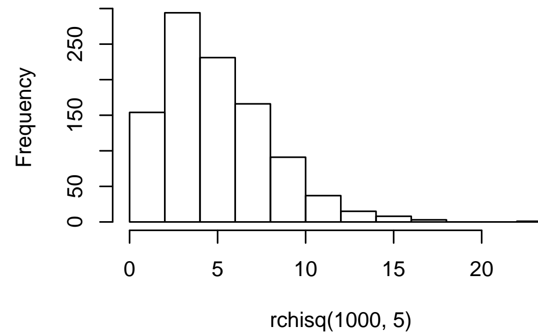
```
> layout(matrix(c(1,2,3,4), nrow=2, byrow=T))
> hist(rnorm(1000))
> hist(rgamma(1000, 1, 1))
> hist(runif(1000))
> hist(rchisq(1000, 5))
```

Example of Histograms





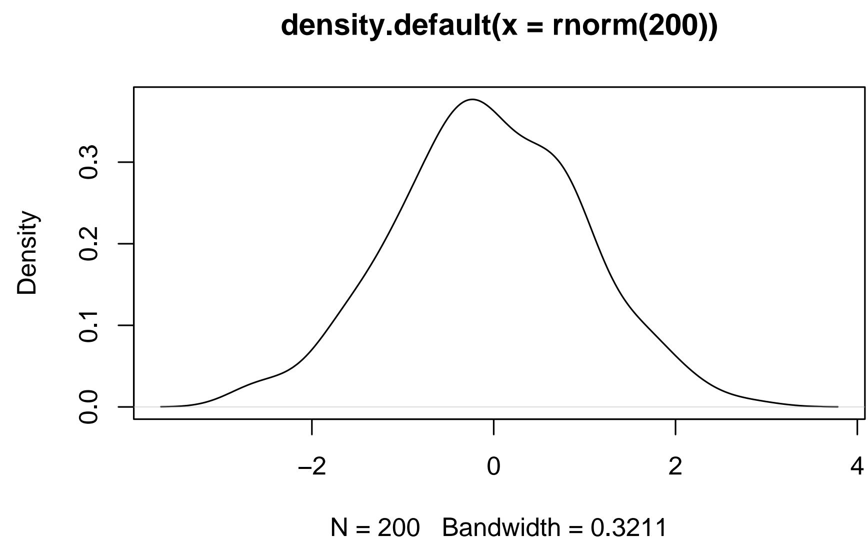




Density Plots

Density plots are generalized histograms

```
> plot(density(rnorm(200)))
```



The Iris Data Set

Let's look at a real example: The `iris` data set is one of many already available in **R** (type `data()` for a full list).

This data set contains measurements on 150 flowers, 50 each from 3 species: *Iris setosa, versicolor*, and *virginica*.

`iris` is typically used to illustrate the problem of classification – given the four measurements for a new flower, can we predict its Species?

When looking at a data set for the first time, it is a good idea to get a general feel for the type of variables in the data set. The `str()` function allows us to look at the structure of the data set and it's variables.

The output of `str(iris)` lets us see that there are 5 variables in `iris`: Sepal.Length, Sepal.Width, Petal.Length, Petal.Width, and Species. The first 4 are numeric variables and Species is a factor.

`iris` is a data frame. We will talk more about what this exactly means, how we can work with individual variables, and add variables to the data frame a little later.

The `summary` function revisited

```
> summary(iris)

  Sepal.Length       Sepal.Width        Petal.Length  
 Min.   :4.300   Min.        :2.000   Min.   :1.000  
 1st Qu.:5.100   1st Qu.     :2.800   1st Qu.:1.600  
 Median :5.800   Median      :3.000   Median :4.350  
 Mean   :5.843   Mean        :3.057   Mean   :3.758  
 3rd Qu.:6.400   3rd Qu.     :3.300   3rd Qu.:5.100  
 Max.   :7.900   Max.        :4.400   Max.   :6.900  
  Petal.Width          Species  
 Min.   :0.100   setosa      :50  
 1st Qu.:0.300   veriscolor  :50  
 Median :1.300   virginica   :50  
 Mean   :1.199                    
 3rd Qu.:1.800                    
 Max.   :2.500                    
```

Note the different format of the output based on type of variable. `Species` is summarized differently because it is a categorical variable (more commonly called a factor in **R**)

Grouped Graphic Displays for Iris Data

We have looked at some simple plotting functions. A more suitable plotting function is available in an add-on package called `lattice`, but that needs the data to be in a slightly different structure:

```
> iris2 <- reshape(iris, varying = list(names(iris)[1:4]),
+ v.names=''measure'', timevar = ''type'', times = names(iris)[1:4],
+ direction = ''long'')
> str(iris2, give.attr = FALSE)
```

```
'data.frame':  600 obs.  of 4 variables:
$ Species:  Factor w/ 3 levels "setosa","versicolor",..:  1 1 1 1 1 ...
$ type :  chr "Sepal.Length" "Sepal.Length" "Sepal.Length" ...
$ measure :  num 5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
$ id :  int 1 2 3 4 5 6 7 8 9 10 ...
```

R Packages

R makes use of a system of packages (section 1.5.3)

- Each package is a collection of routines with a common theme
- The core of R itself is a package called `base`
- A collection of packages is called a `library`
- Some packages are already loaded when R starts up. Other packages need to be loaded using the `library` function

New packages can be downloaded and installed using the `install.packages` function. For example, to install the `ISwR` package (if it's not already installed), one can use:

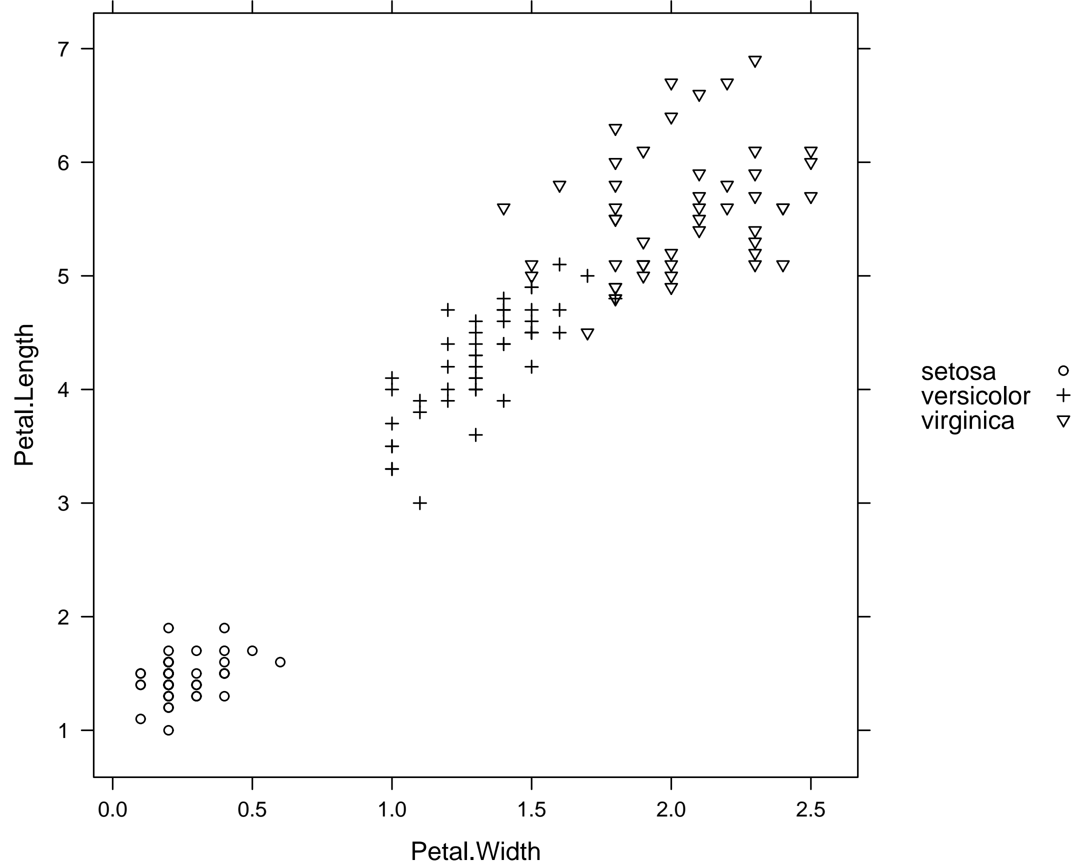
```
> install.packages("ISwR")
```

To load use:

```
> library("ISwR")
```

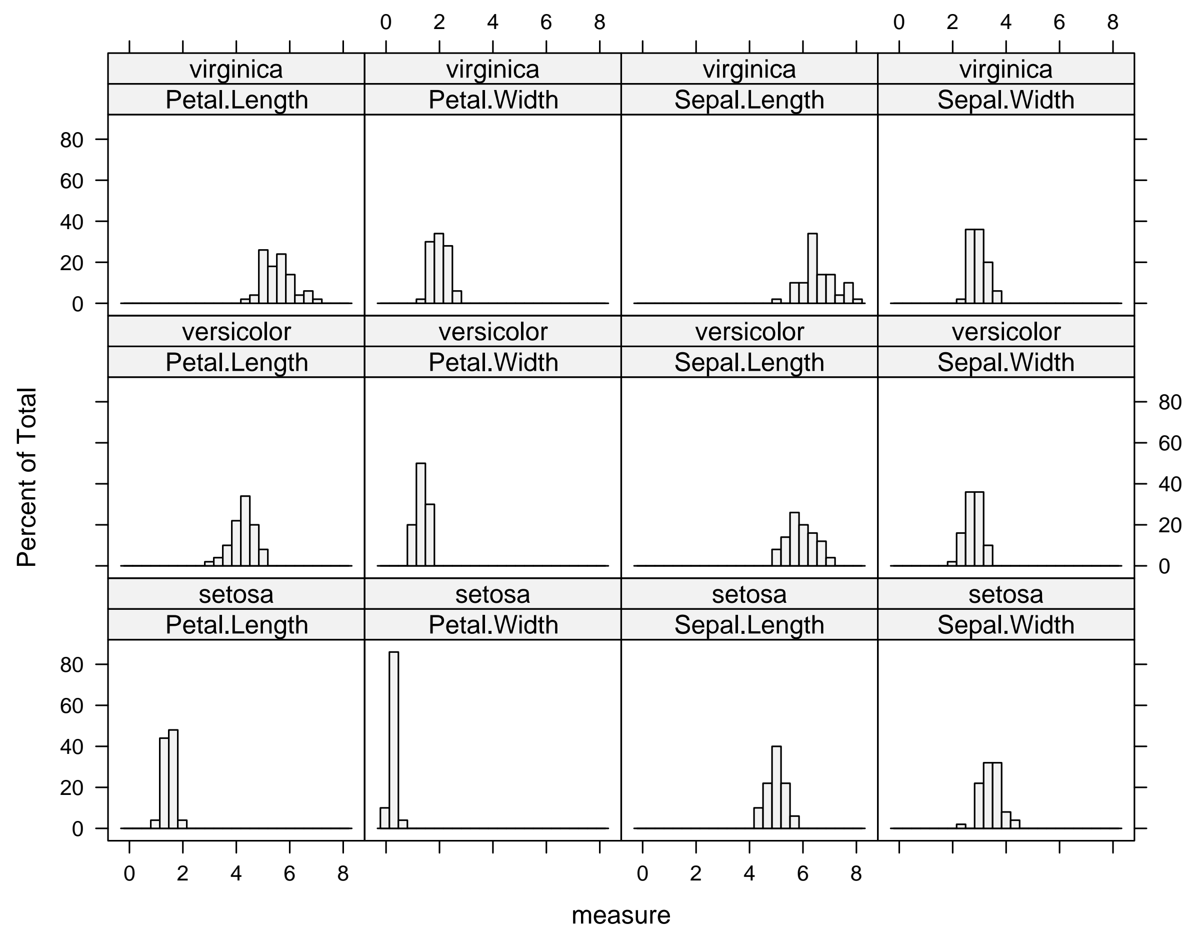
Grouped Scatter Plot

```
> library(lattice) # Loads lattice package for use
> xyplot(Petal.Length ~ Petal.Width, iris, groups = Species,
+ aspect = 1, auto.key = list(space = ''right''))
```



Grouped Display (Historgram)

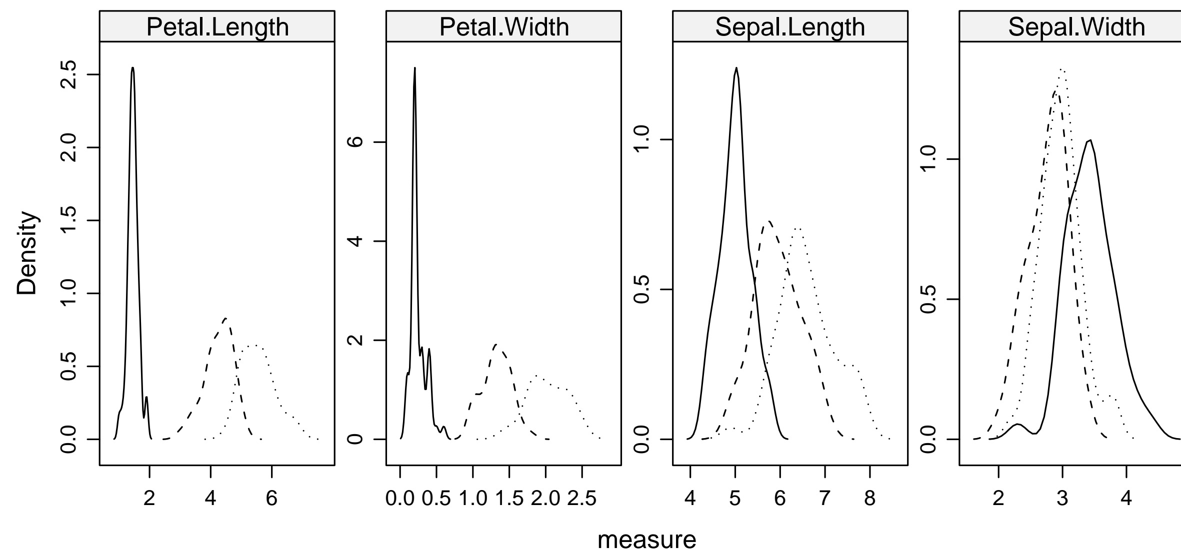
```
> histogram(~measure | type*Species, iris2, nint = 25)
```



Grouped Display (Density Plot)

Again, for grouped data, the analogous `lattice` functions are more suitable.

```
> densityplot(~measure | type, data = iris2, groups = Species,
+ scales = ''free'', plot.points = FALSE)
```



Exercises in Using R

1. Compute: The logarithm of 94 using base 4 and raise it to the 3rd power, then divide by 5, all in the same line of code.

2. Compute: The same logarithm as above. However, this time do it in a three step process using variable names.

3. Compute: Sample 100, 1000, and 10,000 random numbers, separately, from a standard normal distribution. How far do the means deviate from 0? To find the absolute value of a number is the `abs()` function.

Exercises in Using R Answers

1.
```
> log(94, base=4)^3/5
[1] 7.04006
```

2.
```
> bailey <- log(94, base=4)
> chocolate <- bailey^3
> chocolate/5
[1] 7.04006
```

3.
```
> abs(mean(rnorm(100)))
[1] 0.1033906
> abs(mean(rnorm(1000)))
[1] 0.05469583
> abs(mean(rnorm(10000)))
[1] 0.01153124
```

Notice the more random numbers drawn the more accurate the vector of numbers is to the distribution.

Exercises in using R

1. Create a vector of the age of a parent or friend of each of us.

2. Find the average, standard deviation, variance, and median of the ages.

3. Find the quantiles for the vector of ages

4. Find the interquantile range (IQR)

5. Find the number that cuts off the top 10% of the data.

Exercises in using R Answers

This is what it would look like using a different vector.

1. ```
   > ages <- c(45,21,20,35,39,19,24,25,31,24,20,21,19,51,23,22,38,40,21,
   + 26,20)
   ```

2. ```
   > mean(ages)
   [1] 28.2
   > sd(ages)
   [1] 9.687757
   > var(ages) # Same as sd(ages)^2
   [1] 93.85263
   > median(ages)
   [1] 24
   ```

3. ```
   > quantile(ages)
       0%    25%    50%    75%   100%
    19.00  21.00  24.00  35.75  51.00
   ```

4. ```
   > IQR(ages) # or quantile(ages)[4]-quantile(ages)[2]
   75%
   14.75
   ```

5. ```
   > quantile(ages, probs=0.9)
   90%
   40.5
   ```
   ```

Exercises in Using R

- Take the Height and Age of everyone in the class and create two vectors with these numbers.
- Create a new vector with values equal to the square root of the person's height times half their age
- Find the averages of the three vectors.
- Create a plot of Height vs. Age with the points marked as something other than a circle or triangle with a color other than red and with a meaningful title.
- Add a line to the graph that you think very roughly approximates the general slope of the points.
- Create a box plot of the Age vector and a histogram of the Height vector.

What We've Discussed

- **R** can be used as a basic calculator with operators, +, -, *, /, ^ for basic arithmetic.
- To give expressions/objects variable names, we use "`<-`".
- Variable names cannot start with numbers and are case sensitive.
- Functions are used all the time in **R**. Basic functions we have discussed so far include: `rnorm()` `plot()` `sum()` `length()` `mean()` `hist()` `boxplot()` ...
- To plot simple graphs use `plot()`, to add a line to the plot use `lines()`, and to add a title use `title()`. Most graphic functions have an argument that will put the title on the graph without having to use the `title()` function. We will talk about arguments today.

Expressions

As we have seen, **R** works by evaluating expressions typed at the command prompt.

- Expressions involve variable references, operators, function calls, etc.

- Most expressions, when evaluated, produce a value, which can be either assigned to a variable (e.g. `x <- 2 + 2`, or is printed in the **R** session

- Some expressions are useful for their side-effects (e.g. `plot` produces graphical output)

Since evaluated expression values can be quite large, and ofter need to be re-used, it is good practice to assign them to variables rather than print them directly.

Objects

Expressions work on *objects*. Objects are anything that can be assigned to a variable. They could be:

- Constants: 2, 13.005, "January"
- Special symbols: `NA`, `TRUE`, `FALSE`, `NULL`, `NaN`
- Things already in **R**: `seq`, `c` (functions), `month.name`, `letters` (character), `pi` (numeric)
- We can create new objects using existing objects by evaluating expressions
  - `> 1 / sin(seq(0, pi, length = 50))`
  - `> sum(c(1, 2, 3, 4, 5))`
  - `> x <- c(''One'', ''Two'', ''Buckle'', ''My'', ''Shoe'')`

R has several important types of objects that we will learn about, such as functions, vectors (numeric, character, logical), matrices, lists, and data frames.

Functions

Functions in **R** are simply objects of mode “function”. Like other objects, they can be assigned variable names.

Some functions we have already seen:

- `seq()`
- `mean()`
- `plot()`
- `summary()`
- `rnorm()`

Functions (Cont.)

Most useful things in **R** are done by function calls. Function calls look like a name followed by some **arguments** in parentheses.

- Apart from a special argument called `...`, all arguments have a formal name. When a function is evaluated, it needs to know what value has been assigned to each of its arguments

- There are several ways to specify arguments:
  - by position: The first two arguments of the `plot` function are `X` and `Y`. So, `plot(height, weight)` is equivalent to `plot(x=height, y=weight)`
  - by name: This is the safest way to match arguments, by specifying the argument names explicitly. This overrides positional matching, so we can write `plot(y=weight, x=height)` and get the same result.
  - with default values: Arguments will often have default values. If they are not specified in the call, these default values will be used.

But how can we remember what all of the function names are, and furthermore the arguments that go with each function?

Getting Help

R has too many tools for anyone to remember them all, so it is very important to know how to find relevant information using the help system.

- `help.start()`
  Starts a browser window with an HTML help interface. One of the best ways to get started. It has links to a very detailed manual for beginners called "An Introduction to R", as well as topic-wise listings

- `help(topic)`
  Displays the help page for a particular topic or function. Every **R** function has a help page.

- `help.search(''search string'')`
  Subject/keyword search. Helpful when you know generally what you want to do, but are unsure what the function name is to let you do what you want to do. Output will be a list of items that are somehow involved with your search string. Try
  `> help.search(''binomial'')`

The `help` function provides topic-wise help. When you know which function you are interested in, this is usually the best way to learn how to use it. There's also a short-cut for this; use a question mark (?) followed by the topic. The following are equivalent:

```
> help(plot)
> ?plot
```

Getting Help (Cont.)

The help pages can be opened in a browser as well:

```
> help(plot, htmlhelp = TRUE)
```

The help pages are usually very detailed. Among other things, they often contain:

- A 'See Also' section that lists related help pages
- A description of what the function returns
- An 'Examples' section, with actual code illustrating how to use the documented functions. These examples can actually be run directly using the `example` function. Try:

  ```
  > example(plot)
  ```

Functions (Cont.)

You can create your own function, with it's own arguments using the `function` ...um function. Say you have to do a sequence of arithmatic over and over, you might want to write a function to reduce the work.

```
> myfun <- function(vec1, vec2, multiplier=4){
+ temp1 <- vec1*multiplier
+ temp2 <- vec2/multiplier
+ final <- mean(temp1+temp2)
+ return(list(x=temp1, y=temp2, z=final))}
```

Try:

- `> myfun(c(2,4,6,8), c(1,2,3,4), 2)`
- `> myfun(vec1 = weight, vec2 = height)`
- `> myfun(vec1 = weight, multiplier = 3)`

Notice using arguments positionally works fine in the first line. The default value for multiplier works fine in the second line, and an error occurs because vec2 has no default and is not defined in line three.

Function Arguments

The arguments that a particular function accepts (along with their default values) can be listed by the `args` function:

```
> args(myfunc)
function (vec1, vec2, multiplier = 4)
NULL
> args(plot.default)
function (x, y = NULL, type = "p", xlim = NULL, ylim = NULL,
log = "", main = NULL, sub = NULL, xlab = NULL, ylab = NULL,
ann = par("ann"), axes = TRUE, frame.plot = axes,
panel.first = NULL, panel.last = NULL, asp = NA, ...)
NULL
```

The triple-dot (`...`) argument indicates that the function can accept any number of further named arguments. What happens to those arguments is determined by the function.

Vectors

The basic data types in **R** are all vectors. The simplest varieties are numeric, character and logical (`TRUE` or `FALSE`):

```
> c(1, 2, 3, 4, 5)
[1] 1 2 3 4 5
> c(``Huey'', ``Dewey'', ``Louie'')
[1] "Huey" "Dewey" "Louie"
> c(T, T, F, T)
[1]  TRUE TRUE FALSE TRUE
> c(1, 2, 3, 4, 5) > 3
[1]  FALSE FALSE FALSE TRUE TRUE
```

T and F are convenient and equivalent substitutes for TRUE and FALSE.

The length of any vector can be determined by the `length` function:

```
> gt.3 <- c(1, 2, 3, 4, 5) > 3
> gt.3
[1]  FALSE FALSE FALSE TRUE TRUE
> length(gt.3)
[1] 5
```

Special Values

- NA Denotes a “missing value”
- NaN “Not a Number”, e.g. 0/0
- -Inf, Inf negative and positive infinities, e.g. 1/0
- NULL Null object, mostly for programming convenience

Functions That Create Vectors

`seq` (sequence) creates a series of equidistant numbers

See `?seq` for details

```
> args(seq.details)
function (from = 1, to = 1, by = ((to - from)/(length.out - 1)),
length.out = NULL, along.with = NULL, ...)
NULL
```

```
> seq(4, 9)
[1] 4 5 6 7 8 9
> seq(4, 10, 0.5)
[1] 4.0 4.5 5.0 5.5 6.0 6.5 7.0 7.5 8.0 8.5 9.0 9.5 10.0
> seq(length=10)
[1] 1 2 3 4 5 6 7 8 9 10
> 1:10
[1] 1 2 3 4 5 6 7 8 9 10
```

Functions That Create Vectors (Cont.)

`c()` concatenates one of more vectors: Creating one long vector out of shorter vectors

```
> c(1:5, seq(10, 20, length=6))
[1] 1 2 3 4 5 10 12 14 16 18 20
```

Partial matching: Note that the named argument `length` *of the call to* `seq` *actually is the argument* `length.out`

`rep` replicates a vector

The arguments for `rep` are `rep(x, ...)` where `...` can take on “times”, “length.out”, or “each”. The default is that the second argument is “times”.

See `?rep` for details

```
> rep(1:5, 2) # Repeat 1:5 twice
[1] 1 2 3 4 5 1 2 3 4 5
> rep(1:5, length = 12) # Repeat 1:5 for 12 spots
[1] 1 2 3 4 5 1 2 3 4 5 1 2
> rep(c("one", "two"), c(6, 3)) # Repeat ”one” 6 times then ”two” three times
[1] "one" "one" "one" "one" "one" "one" "two" "two" "two"
```

Matrices

Matrices are stored in **R** as a vector with dimensions:

```
> x <- 1:12
> dim(x) <- c(3,4)
> x
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12

> nrow(x) # Function asking for number of rows in matrix
[1] 3
> ncol(x) # Similarly number of columns in matrix
[1] 4
```

The fact that the left hand side of an assignment can look like a function applied to an object (rather than a variable) is a very interesting and useful feature. These are called replacement functions.

Matrices (Cont.)

Matrices can also be created conveniently by the `matrix` function. Their row and column names can be set.

```
> x <- matrix(1:12, nrow=3, byrow=TRUE)
> rownames(x) <- LETTERS[1:3]
> colnames(x) <- c(''Peter'', ''Lois'', ''Stewie'', ''Meg'')
> x
   Peter  Lois  Stewie  Meg
 A     1     2       3    4
 B     5     6       7    8
 C     9    10      11   12
```

Notice what the `byrow` argument does to the numbers when `TRUE`

Matrices can be transposed by the `t()` function.

```
> t(x)
         A  B   C
 Peter   1  5   9
 Lois    2  6  10
 Stewie  3  7  11
 Meg     4  8  12
```

Matrices (Cont.)

Matrices do not need to be numeric. They can be character or logical, however they can't mix.:

```
> matrix(month.name, ncol = 6, byrow=TRUE)
     [,1]      [,2]       [,3]        [,4]      [,5]       [,6]
[1,] "January" "February" "March"     "April"   "May"      "June"
[2,] "July"    "August"   "September" "October" "November" "December"
```

```
> m < - matrix(c(rep(c(TRUE,FALSE,3),rep(LETTERS[1:3],2),1:6), ncol=3))
> colnames(m) <- c(``Higher'', ``TRT'', ``ptID'')
     Higher  TRT ptID
[1,] "TRUE"  "A" "1"
[2,] "FALSE" "B" "2"
[3,] "TRUE"  "C" "3"
[4,] "FALSE" "A" "4"
[5,] "TRUE"  "B" "5"
[6,] "FALSE" "C" "6"
```

Notice how since there was a mix of types of data, the matrix function coerces everything to a character type of data. There is a way to have multiple types of data in a matrix format, these are called data frames. We will discuss these shortly.

Matrix Multiplication

The multiplication operator (*) works element-wise, as with vectors. The matrix multiplication operator is `%*%`:

```
> x
    Peter  Lois  Stewie  Meg
 A      1     2       3    4
 B      5     6       7    8
 C      9    10      11   12

> x * x
    Peter  Lois  Stewie  Meg
 A      1     4       9   14
 B     25    36      49   64
 C     81   100     121  144

> x %*% t(x) # Remember dimensions need to match
      A    B    C
 A   30   70  110
 B   70  174  278
 C  110  278  446
```

Creating Matrices from Vectors

The `cbind` (column bind) and `rbind` (row bind) functions can create matrices or vectors:

```
> y <- cbind(A = 1:4, B = 5:8, C = 9:12) # Not arguments, naming columns
> y
     A B  C
[1,] 1 5  9
[2,] 2 6 10
[3,] 3 7 11
[4,] 4 8 12

> rbind(y,0)
     A B  C
[1,] 1 5  9
[2,] 2 6 10
[3,] 3 7 11
[4,] 4 8 12
[5,] 0 0  0
```

Note that the short vector (0) is replicated to the number of columns

Factors

Factors are how **R** handles categorical data (e.g. eye color, gender, pain level). Such data are often available as numeric codes, but should be converted to factors for proper analysis.

```
> pain <- c(0, 3, 2, 2, 1) # check summary(pain)
> ftpain <- factor(pain, levels=c(0,1,2,3)) # check summary(ftpain)
> ftpain # Notice pain and ftpain print the same but different summaries
[1] 0 3 2 2 1
Levels:  0 1 2 3
> levels(ftpain) <- c("none", "mild", "medium", "severe")
> ftpain
[1] none severe medium medium mild
Levels:  none mild medium severe
> as.numeric(ftpain) # Convert back to numeric, but not the same
[1] 1 4 3 3 2
```

The last function extracts the internal representation of factors, as integer codes starting from 1.

Factors (Cont.)

Factors can also be created from character vectors:

```
> text.pain <- c("none", "severe", "medium", "medium", "mild")
> factor(text.pain)
[1] none severe medium medium mild
Levels:  medium mild none severe
```

Note that the levels are sorted alphabetically by default, which may not be what you really want. It is usually a good idea to specify the levels explicitly when creating a factor, or make the factor an ordered factor.

```
> factor(text.pain, levels=c("none", "mild", "medium", "severe"))
[1] none severe medium medium mild
Levels:  none mild medium severe
> ordered(text.pain, levels=c("none", "mild", "medium", "severe"))
[1] none severe medium medium mild
Levels:  none < mild < medium < severe
```

The only real difference between the two is the class type, one is a factor and one is ordered. However, specifying the levels in `factor` does give an order to the levels. Personally, I feel if there is an order to the levels, make it an ordered factor.

Lists

It is sometimes useful to combine a collection of objects into a larger composite object. This can be done using `list()` to create a list.

- Lists are very flexible data structures that are used extensively in **R**
- A list is a vector, but the elements of a list do not need to be of the same type. Each element of a list can be any **R** object, including another list.
- List elements are usually extracted by name (using the `$` operator)

Lists (Cont.)

```
> x <- list(fun = seq, len=10)
> x$fun # Calls for just the fun element of x
function (...)
UseMethod("seq")
<environment:  namespace:base>
> x$len
[1] 10
> x$fun(length = x$len) # Runs the seq function using the list elements
[1] 1 2 3 4 5 6 7 8 9 10
```

Functions are **R** objects as well. In this case, the `fun` element of `x` is the already familiar `seq` function, and can be called like the function itself.

Lists give us the ability to create composite objects that contain several related, simpler objects. Many useful **R** functions return a list rather than a simple vector.

Lists (Cont.)

A more natural example, using a data set on energy intake in a group of women (pre– and post–menstrual) (section 1.2.8 of the book)

```
> intake.pre <- c(5260, 5470, 5640, 6180, 6390, 6515, 6805,
+ 7515, 7515, 8230, 8770)
> intake.post <- c(3910, 4220, 3885, 5160, 5645, 4680, 5265,
+ 5975, 6790, 6900, 7335)
> mylist <- list(before = intake.pre, after = intake.post)
> mylist
$before
[1] 5260 5470 5640 6180 6390 6515 6805 7515 7515 8230 8770


$after
[1] 3910 4220 3885 5160 5645 4680 5265 5975 6790 6900 7335


> mylist$before
[1] 5260 5470 5640 6180 6390 6515 6805 7515 7515 8230 8770
> mylist[[2]]
[1] 3910 4220 3885 5160 5645 4680 5265 5975 6790 6900 7335
```

List elements can be extracted by name as well as position. This is a form of indexing, which will we talk about after data frames.

Data Frames

We have talked about a lot of different types of objects that can contain data. We have progressed from the simplest vector, to matrcies, and now we finish with data frames. Lists are kind of on their own level.

Data frames are **R** objects that represent data sets (and is probably the object we will deal with most frequently). They are essentially lists with some additional structure.

- Each element of a data frame has to be either a factor, numeric, character, or logical vector.
- Each of these elements must have the same length
- Remember when we talked about matrices we could not have different data types with a matrix. Well, data frames are similar to matrices because they have the same retangular array structure; the only difference is that different columns of a data frame can be of a different data type.
- When we start loading our own external data sets they will be put into **R** as a data frame.

Data Frames (Cont.)

Data frames are created by the `data.frame` function:

```
> d <- data.frame(intake.pre, intake.post)
> d
     intake.pre  intake.post
 1         5260         3910
 2         5470         4220
 3         5640         3885
 4         6180         5160
 5         6390         5645
 6         6515         4680
 7         6805         5265
 8         7515         5975
 9         7515         6790
 10        8230         6900
 11        8770         7335

> d$intake.post
[1] 3910 4220 3885 5160 5645 4680 5265 5975 6790 6900 7335
```

Since data frames are lists, the `$` operator can be used to extract columns.

Exercises in Using R

1. Create a vector of 24 numbers evenly spaced as possible from 1 to 90, rounding to 2 decimal places.

2. With this vector, create an 8 x 3 matrix entering the numbers by row. Name the columns A, B, C.

3. Matrix multiply the created matrix with its transpose.

4. What is the 3rd quartile of the matrix from the matrix multiplication?

5. Turn the matrix into a data frame and add a new column of repeating trues and falses. Call this column ”TRT”.

6. What is the sum of column A?

7. What is the mean of column B?

8. What is the standard deviation of column C?

Exercises in Using R Answers

1. ```
   > vec <- round(seq(from=1, to=90, length=24),2)
   ```

2. ```
   > mat <- matrix(vec, nrow=8, ncol=3, byrow=T)
   > colnames(mat) <- c("A", "B", "C")
   ```

3. ```
   > newMat <- mat%*%t(mat)
   ```

4. ```
   > quantile(newMat, probs=.75)
   75%
   9711.803
   ```

5. ```
   > dfMat <- data.frame(mat)
   > TRT <- rep(c(TRUE, FALSE), times=4)
   > dfMat <- cbind(dfMat, TRT)
   ```

6. ```
   > sum(dfMat$A)
   [1] 333.04
   ```

7. ```
   > mean(dfMat$B)
   [1] 45.5
   ```

8. ```
   > sd(dfMat$C)
   [1] 28.43391
   ```

Exercises in Using R

Create a function that will calculate and return the roots of the following formulas.

1. $x^2 - 2x = -1$
2. $2x^2 - 3x - 3 = 0$
3. $10x^2 + 12x = 3$
4. $-4x^2 + 6x - 4 = 0$

Exercises in Using R Answers

```
> quadratic <- function(a, b, c)
+ {
+ posX <- (-b + sqrt(b^2 - 4*a*c))/(2*a)
+ negX <- (-b - sqrt(b^2 - 4*a*c))/(2*a)
+ return(c(posX, negX))
+ }
```

1. ```
   > quadratic(1, -2, 1)
   [1] 1 1
   ```

2. ```
   > quadratic(2, -3, -3)
   [1] 2.1861407 -0.6861407
   ```

3. ```
   > quadratic(10, 12, -3)
   [1] 0.2124038 -1.4124038
   ```

4. ```
   > quadratic(-4, 6, -4)
   [1] NaN NaN
   ```

See `?polyroot` to see this function is already in **R**.

Exercises in Using R

There is a data set in **R** called `mtcars` take a look at the structure of this data set. All of the variables are listed as numeric. However if you think about some of the variables they really should be treated as factors.

1. Change the variable `am` into a factor with levels 0 = Automatic and 1 = Manual. How many of each type of transmissions are there?

2. Change the variable `cyl` into an ordered factor with levels 4 = Four, 6 = Six, and 8 = Eight. How many of each are there?

3. What is the median miles per gallon?

4. What is the number that cuts the bottom 15% of the sample distribution of horsepower?

Exercises in Using R Answers

1.
```
> carData <- mtcars
> am.F <- factor(carData$am, labels=c("Automatic", "Manual"))
> carData <- cbind(carData, am.F)
> summary(carData$am.F)
 Automatic   Manual
        19       13
```

2.
```
> cyl.F <- ordered(carData$cyl, levels = c(4,6,8),
+ labels=c("Four", "Six", "Eight"))
> carData <- cbind(carData, cyl.F)
> summary(carData$cyl.F)
 Four   Six   Eight
   11     7      14
```

3.
```
> median(carData$mpg)
[1] 19.2
```

4.
```
> quantile(carData$hp, probs=.15)
   15%
 82.25
```

What we have Discussed

- Objects, Vectors, and Factors
- Functions and Arguments with functions
- Getting Help
- Special Values `NA`, `NaN`, `Inf`, `NULL`
- Matrices, Lists, Data Frames

Indexing

Okay so we now have the basics on the types of data and how we can manipulate some things as whole vectors. Well what if we want to look at a specific reading or a subset set of readings. That is where indexing and conditional selection comes into use.

There are several kinds of indexing possible in **R**, among them:

- Indexing by a vector of positive integers
- Indexing by a vector of negative integers
- Indexing by a logical vector
- Indexing by a vector of names

In each case, the extraction is done by following the vector by a pair of brackets [. . .]. The type of indexing depends on the object inside the brackets.

Indexing by Positive Integers

```
> intake.pre
[1] 5260 5470 5640 6180 6390 6515 6805 7515 7515 8230 8770
> intake.pre[5]
[1] 6390
> intake.pre[c(3, 5, 7)]
[1] 5640 6390 6805
> ind <- c(3,5,7)
> intake.pre[ind]
[1] 5640 6390 6805
> intake.pre[8:13]
[1] 7515 7515 8230 8770 NA NA
> intake.pre[rep(c(1,2),2)]
[1] 5260 5470 5260 5470
```

Works more or less as expected. Interesting features:

- using an index bigger than the length of the vector produces `NA`'s
- indices can be repeated, resulting in the same element being chosen more than once.

Indexing by Negative Integers

Using negative indices leaves out the specified elements.

```
> intake.pre
[1] 5260 5470 5640 6180 6390 6515 6805 7515 7515 8230 8770
> intake.pre[-5]
[1] 5260 5470 5640 6180 6515 6805 7515 7515 8230 8770
> ind <- -c(3, 5, 7)
> ind
[1] -3 -5 -7
> intake.pre[ind]
[1] 5260 5470 6180 6515 7515 7515 8230 8770
```

Negative indices cannot be mixed with positive indices.

Indexing by a Logical Vector

For this, the logical vector being used as the index should be exactly as long as the vector being indexed. If it is shorter, it is replicated to be as long as necessary.

```
> intake.pre
[1] 5260 5470 5640 6180 6390 6515 6805 7515 7515 8230 8770
> ind <- rep(c(TRUE, FALSE), length = length(intake.pre))
> ind
[1]  TRUE FALSE TRUE FALSE TRUE FALSE TRUE FALSE TRUE FALSE TRUE
> intake.pre[ind]
[1] 5260 5640 6390 6805 7515 8770
> intake.pre[c(T, F)]
[1] 5260 5640 6390 6805 7515 8770
```

Only the elements that correspond to `TRUE` are retained.

Indexing by Names

This works only for vectors that have names.

```
> names(intake.pre) <- LETTERS[1:11]
> intake.pre
   A    B    C    D    E    F    G    H    I    J    K
5260 5470 5640 6180 6390 6515 6805 7515 7515 8230 8770
```

```
> intake.pre[c("A","B","C","K")]
   A    B    C    K
5260 5470 5640 8770
```

```
> names(intake.pre) <- NULL # Removes the names
[1] 5260 5470 5640 6180 6390 6515 6805 7515 7515 8230 8770
```

All these types of indexing work for matrices and data frames as well, which will see a little later.

Logical Comparisons

All the usual logical comparisons are possible in **R**:

less than	$<$	less than or equal to	$<=$
greater than	$>$	greater than or equal to	$>=$
equals	$==$	does not equal	$!=$

Each of these operate on two vectors element–wise (the shorter one is replicated till it matches the larger vector's lenght).

```
> intake.pre
[1] 5260 5470 5640 6180 6390 6515 6805 7515 7515 8230 8770
> intake.pre > 7000
[1]  FALSE FALSE FALSE FALSE FALSE FALSE FALSE TRUE TRUE TRUE TRUE
> intake.pre > intake.post
[1]  TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
```

Logical Operations

Element-wise boolean operations on logical vectors are also possible.

AND	&
OR	|
NOT	!

```
> intake.pre
[1] 5260 5470 5640 6180 6390 6515 6805 7515 7515 8230 8770
> intake.pre > 7000
[1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE TRUE TRUE TRUE TRUE
> intake.pre < 8000
[1]  TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE FALSE FALSE
> intake.pre > 7000 & intake.pre < 8000
[1]  FALSE FALSE FALSE FALSE FALSE FALSE FALSE TRUE TRUE
[10] FALSE FALSE
> intake.pre < 7000 | intake.pre > 8000
[1]  TRUE TRUE TRUE TRUE TRUE TRUE TRUE FALSE FALSE TRUE TRUE
```

Conditional Selection

Logical comparisons and indexing by logical vectors allow subsetting a vector based on the properties of other (or perhaps the same) vectors.

```
> intake.post
[1] 3910 4220 3885 5160 5645 4680 5265 5975 6790 6900 7335

> intake.post[intake.pre > 7000]
[1] 5975 6790 6900 7335

> intake.post[intake.pre > 7000 & intake.pre < 8000]
[1] 5975 6790

> month.name[month.name > "n"]
[1] "September" "October" "November"
```

For character vectors, sorting is determined by alphabetical order.

Matrix and Data Frame Indexing

Indexing for matrices and data frames are very similar. They use brackets too, but need two indices. If one (or both) of the indices are unspecified all corresponding rows and/or columns are selected.

```
> x <- matrix(1:12, 3, 4, dimnames=list(LETTERS[1:3], LETTERS[4:7]))
> x
    D   E   F    G
 A  1   4   7   10
 B  2   5   8   11
 C  3   6   9   12
> x[1:2, 1:2]
    D   E
 A  1   4
 B  2   5
> x[1:2,]
    D   E   F    G
 A  1   4   7   10
 B  2   5   8   11
```

If only one row or column is selected, the result is converted to a vector. This can be suppressed by adding a `drop = FALSE`.

```
> x[1,]
D E F G
1 4 7 10
> x[1, , drop=FALSE]
    D   E   F    G
 A  1   4   7   10
```

Matrix and Data Frame Indexing (Cont.)

Data frames behave similarly

```
> d[1:3, ]
    intake.pre  intake.post
 1        5260         3910
 2        5470         4220
 3        5640         3885

> d[1:3, "intake.pre"]
[1] 5260 5470 5640


> d[d$intake.post < 5000, 1, drop = FALSE]
    intake.pre
 1        5260
 2        5470
 3        5640
 6        6515
```

Modifying Objects

It is usually possible to modify **R** objects by assigning a value to a subset or function of that object. For the most part, anything that makes sense works. This will become clearer with more experience.

```
> x <- runif(10, min = -1, max = 1)
> x
[1] 0.98444235 -0.67524557 0.69691926 -0.55093054 -0.96809742
[6] 0.85253382 0.93439271 0.99339256 -0.05905403 0.08357995
> x < 0
[1] FALSE TRUE FALSE TRUE TRUE FALSE FALSE FALSE TRUE FALSE
> x[x < 0] <- 0
> x
[1] 0.98444235 0.00000000 0.69691926 0.00000000 0.00000000
[6] 0.85253382 0.93439271 0.99339256 0.00000000 0.08357995
```

Adding Columns to a Data Frame

New columns can be added to data frame, by assigning to a currently non-existent column name (this works for lists too):

```
> d$decrease # Check if name is available
NULL
```

```
> d$decrease <- d$intake.pre - d$intake.post
> d
    intake.pre  intake.post  decrease
 1        5260         3910      1350
 2        5470         4220      1250
 3        5640         3885      1755
 4        6180         5160      1020
 5        6390         5645       745
 6        6515         4680      1835
 7        6805         5265      1540
 8        7515         5975      1540
 9        7515         6790       725
 10       8230         6900      1330
 11       8770         7335      1435
```

The subset function

Working with data frames can become a bit cumbersome because we always need to prefix the name of the data frame to every column. There are some functions that make this easier. `subset()` can be used to select rows of a data frame. Look at `?subset()` for details.

There is a data set in the `survival` package called lung. Say the investigator is only interested in outcomes for people over the age of 65. You can narrow the data set down using `subset()`. Looking at the output from `str(lung)` we see there are 228 obs. on 10 variables.

```
> library(survival)
> SeniorData <- subset(x=lung, subset=lung$age > 65)
```

Now look at `str(SeniorData)`. The data set has been narrowed to 92 obs.
Now say you want only the variables inst, time, age, and only males (which is sex=1).

```
> MaleData <- subset(x=lung, subset=lung$sex==1, select=c(1,2,4,5))
```

Again look at the structure, `str(MaleData)`. Notice only those variables are present and the observations have dropped to 138.

The transform function

Similarly, the `transform` function can be used to add new variables to a data frame using the old ones.

```
> lung2 <- transform(lung, log.meal = log(meal.cal))
> summary(lung2$log.meal)
  Min.  1st Qu.  Median   Mean  3rd Qu.   Max.    NA's
 4.564    6.454   6.882  6.727    7.048  7.863  47.000
```

Another similar function that can do the same this is `with`, which can be used to evaluate arbitary expressions using variables in a data frame:

```
> log.meal <- with(lung, log(meal.cal))
> summary(log.meal)
  Min.  1st Qu.  Median   Mean  3rd Qu.   Max.    NA's
 4.564    6.454   6.882  6.727    7.048  7.863  47.000
```

Sorting

Vectors can be sorted by `sort`:

```
> sort(intake.post)
[1] 3885 3910 4220 4680 5160 5265 5645 5975 6790 6900 7335
```

But it's usually more useful to work with the sort order, using the `order` function, which returns an integer indexing vector that be used to get the sorted vectors. This can be useful to re-order the rows of a data frame by one or more columns.

```
> ord <- order(intake.post)
[1] 3 1 2 6 4 7 5 8 9 10 11
> intake.post[ord]
[1] 3885 3910 4220 4680 5160 5265 5645 5975 6790 6900 7335
> intake.pre[ord]
[1] 5640 5260 5470 6515 6180 6805 6390 7515 7515 8230 8770
> d[ord, ]
     intake.pre  intake.post  decrease
 3         5640         3885      1755
 1         5260         3910      1350
 2         5470         4220      1250
 6         6515         4680      1835
 4         6180         5160      1020
 7         6805         5265      1540
 5         6390         5645       745
 8         7515         5975      1540
 9         7515         6790       725
 10        8230         6900      1330
 11        8770         7335      1435
```

Graphics

Its graphics capabilites are one of **R**'s strongest features. It can also be fairly complicated, with many features that are rarely used. Instead of going into details here, we will learn about **R** graphics by looking at some examples later. Meanwhile,

- Read section 1.3 of the text
- Look at `help(plot.default)`
- Look at `help(par)`

These two help pages cover most of the options and features common to standard graphics functions. They contain a lot of information, and are mostly useful as references to look up when you need to do something special.

Programming constructs

Since in **R** you can create your own functions, **R** has standard programming constructs some are: `if`, `else`, `for`, `while`

Since most **R** functions work on vectors, the `for` construct is rarely needed for simple use. The `for` keyword is always followed by an expression of the form (*variable* in *vector*). The block of statements that follow this is executed once for every value in *vector*, that value is stored in *variable*.

```
> for(i in 1:5)
+ {
+ print(i^2)
+ }
[1] 1
[1] 4
[1] 9
[1] 16
[1] 25
```

While and If Statements

```
> fibonacci <- function(length.out) {
+ if(length.out < 0) {
+ warning("length.out cannot be negative")
+ return(NULL)
+ }
+ else if(length.out < 2)
+ x <- seq(length = length.out) - 1
+ else {
+ x <- c(0,1)
+ while(length(x) < length.out) {
+ x <- c(x, sum(rev(x)[1:2]))
+ }
+ }
+ x
+ }
> fibonacci(-1)
NULL
> fibonacci(1)
[1] 0
> fibonacci(10)
[1] 0 1 1 2 3 5 8 13 21 34
```

Attaching and Detaching data frames

The notation for accessing variables in data frames gets rather heavy if you repeatedly have to write longish commands like:

```
> plot(lung$wt.loss, lung$meal.col)
```

As you have probably noticed, just using the variable like, wt.loss, produces an error:

```
> sum(wt.loss)
Error:  object "wt.loss" not found
```

Fortunately, you can make **R** look for objects among the variables in a given data frame. This is done through the `attach` function.

```
> attach(lung)
> sum(wt.loss, na.rm=TRUE)
[1] 2104
```

What happens is that the data frame `lung` is placed in the system's *search path*. You can view the search path with `search`:

```
> search()
```

Lung becomes the second place that **R** will search for the variable name that you write. Caution should be taken when attaching data frames and especially multiple data frames. I recommend not attaching the data frame when using more than one data frame in the same session. After attached, the $ operator still works. To remove a data frame from the search path use `detach`

```
> detach(lung)
```

Type Checking

It is often useful to know whether an object is of a certain type. There are several functions of the form `is.type` which do this. Note that `is` is not a generic function, even though the naming convention is similar.

```
> is.data.frame(lung)
[1] TRUE
```

```
> is.list(lung)
[1] TRUE
```

```
> is.numeric(lung)
[1] FALSE
```

```
> is.factor(lung)
[1] FALSE
```

Detecting Special Values

Some other functions are used for element-wise checking. The most important of these are `is.na` which is needed to identify which elements of a vector are missing.

```
> yLung <- subset(lung, subset=age < 45)
> yLung$meal.cal
[1] NA 1425 588 1025 588 1175 2450 338 1225 NA 2350
```

```
> is.na(yLung$meal.cal)
[1] TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE TRUE FALSE
```

```
> is.na(c(Inf, NaN, NA, 1))
[1] FALSE TRUE TRUE FALSE
```

```
> is.nan(c(Inf, NaN, NA, 1))
[1] FALSE TRUE FALSE FALSE
```

```
> is.finite(c(Inf, NaN, NA, 1))
[1] FALSE FALSE FALSE TRUE
```

`is.null` is used often in function writing. An argument can by given the default value of `NULL` and in the code the writer can check if that argument has been specified by using `is.null`. Normally used in an 'if then' statement, `if(is.null(var))` (meaning it has not been modified) then do something. If var is not `NULL` do something else.

Coercion Methods

There are a whole bunch of functions of the form `as.type` that are used to convert objects from one type to another.

```
> as.numeric(c("1", "2", "2a", "b", "3"))
[1] 1 2 NA NA 3
```

Notice characters cannot be coerced to a numeric, however logicals and factors can.

```
> as.numeric(as.factor(c("1", "2", "2a", "b", "3")))
[1] 1 2 3 5 4
```

```
> as.numeric(c(TRUE, FALSE, NA))
[1] 1 0 NA
```

```
> as.character(c(TRUE, FALSE, NA))
[1] "TRUE" "FALSE" NA
```

There are some automatic coercion rules that often simplify things:

```
> yLung$status == 1
[1] FALSE TRUE FALSE TRUE TRUE FALSE FALSE TRUE TRUE TRUE TRUE
> sum(yLung$status == 1)
[1] 7
```

Session Management

R has the ability to save objects that you have created/named in the time you have been running the program to be loaded again later. Whenever exiting, R tries to save all the objects currently in the workspace, and when starting up the next time (in the same directory), it will load the objects again.

For more information on session management, read section 1.5.1 in the text.

Exercises in Using R

There is a data set in **R** called `trees` that has three numeric variables: Girth, Height, and Volume. There are a few questions the tree researcher would like to have answered.

1. Trees with a Girth greater than or equal to 11 inches are considered wide trees. How many wide trees are there?

2. What are the sample means for height of wide trees and skinny trees?

3. What are the Heights of the trees that both have Girth larger than 16 inches and Volume greater than 52 cubic ft?

4. Trees that have a Volume less than 15 cubic ft or Volume greater than or equal to 55 are considered extreme. Add a column to the data set telling whether a tree is extreme for volume.

Exercises in Using R Answers

1.
```
> sum(trees$Girth > 11)
[1] 23
```

2.
```
> mean(trees$Height[trees$Girth > 11])
[1] 77.43478
> mean(trees$Height[trees$Girth <= 11])
[1] 71.875
```

3.
```
> trees$Height[trees$Girth > 16 & trees$Volume > 52]
[1] 81 82 80 87
```

4.
```
> trees$ExtVol <- trees$Volume < 15 | trees$Volume >= 55
> trees$ExtVol
[1] TRUE TRUE TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[11] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE

[20] FALSE FALSE FALSE FALSE FALSE FALSE TRUE TRUE TRUE FALSE
[30] FALSE TRUE
```

Exercises in Using R

Using the data set called `airquality`, which has measurements of New York Air for the months of May, June, July, August, and September in 1973.

1. Attach airquality to **R**'s search path.
2. Using `is.na()` how many missing values are there in the variable Ozone?
3. Find the median of the variable Solar.R. Notice `NA` is returned because of the missing values in Solar.R. Use a `help` function to figure out how to get around this.
4. Use a for loop to get a vector of means of Wind based on the month. Hint: your for loop could start like this: `for(i in 5:9)` and you should initialize your vector first with `meanVec <- c()` before the for loop

Exercises in Using R Answers

1.
```
> attach(airquality)
```

2.
```
> sum(is.na(Ozone))
[1] 37
```

3.
```
> median(Solar.R)
[1] NA
> median(Solar.R, na.rm=TRUE)
[1] 205
```

4.
```
> meanVec <- c()
> for(i in 5:9)
+ {
+ tempData <- subset(airquality, subset=Month==i)
+ meanVec[i-4] <- mean(tempData$Wind)
+ }
> meanVec
[1] 11.622581 10.266667 8.941935 8.793548 10.180000
```

There is a function `tapply` that will do the for loop for us.

```
> tapply(Wind, as.factor(Month), mean)
        5         6         7         8         9
11.622581 10.266667  8.941935  8.793548 10.180000
```

Exercises in Using R

A researcher comes to you and asks for you to use graphical representations to answer the following question: Is there a difference in chick weight at the **end** of the study based on different diets? The data set can be found in **R** as ChickWeight.

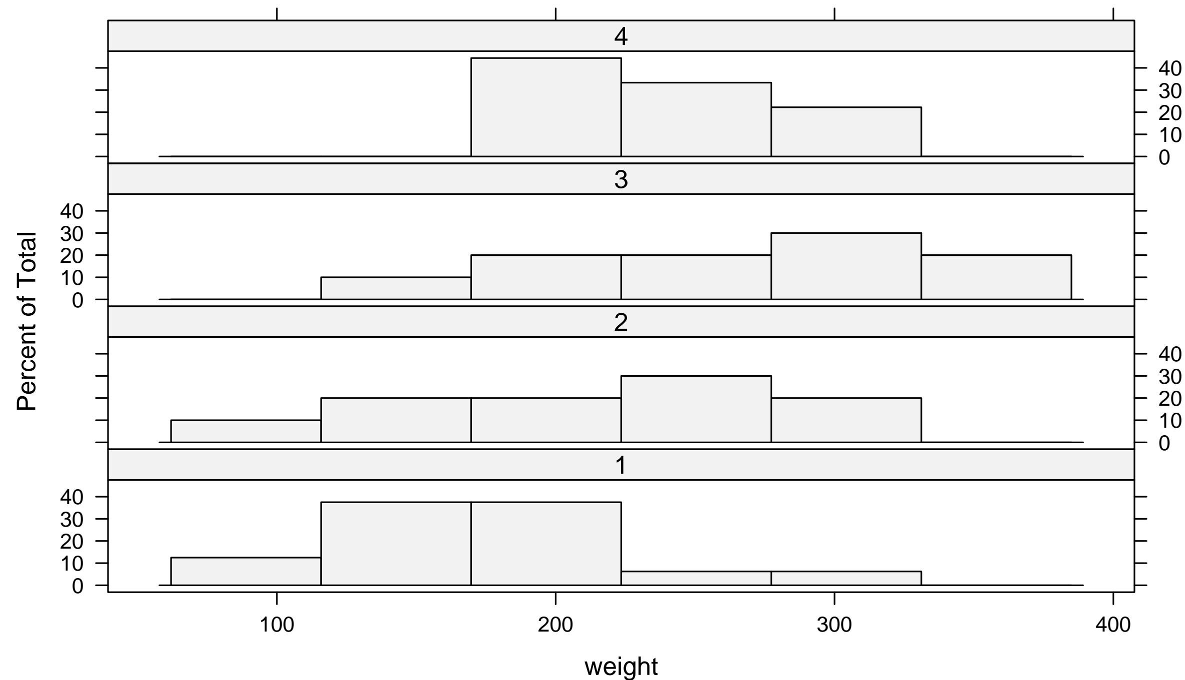
What kind of figure(s) would you use? Is there a difference?

To load the data set for your use you can just use ChickWeight which is already a data frame in **R** or rename ChickWeight to a name you like better.

Hint: Can look at iris data example from earlier for ideas.

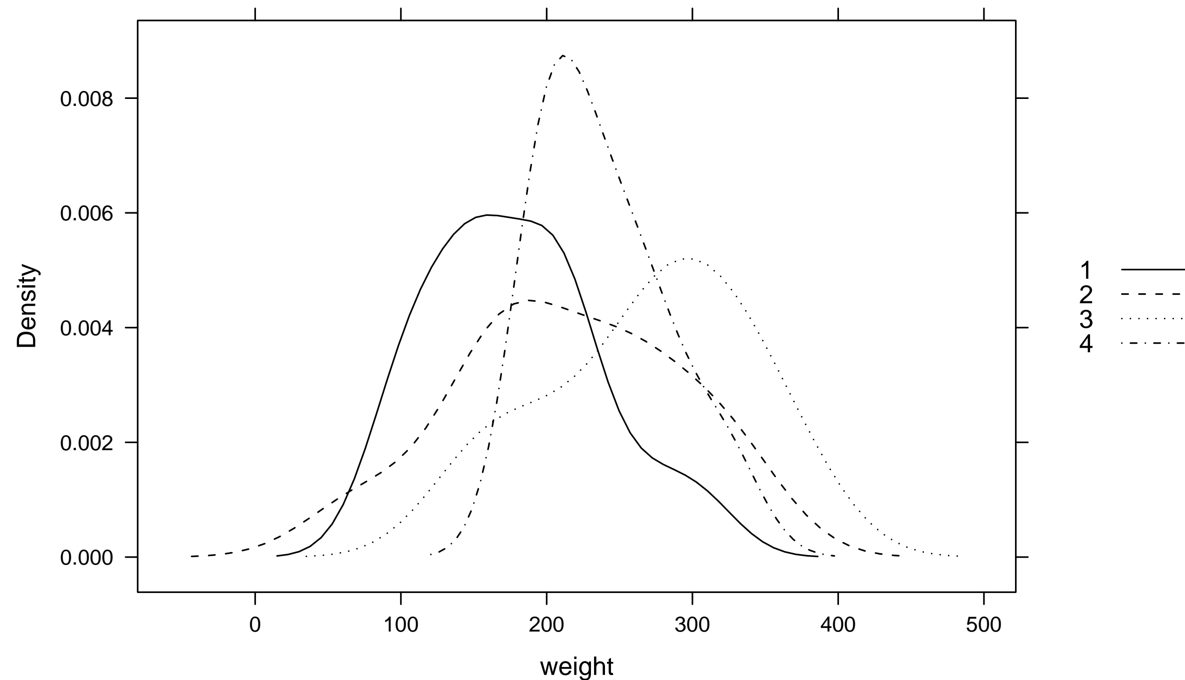
Exercises in Using R Answers

```
> library("lattice")
> histogram( weight | Diet, data=ChickWeight, subset=Time==21,
+ layout=c(1,4))
```



Exercises in Using R Answers

```
> layout(1)
> densityplot( weight, groups=Diet, data=ChickWeight, subset=Time==21,
+ plot,points=FALSE, auto.key = list(space = "right"))
```



Exercises in Using R Answers

```
> Day21CW <- subset(ChickWeight, subset=Time==21)
> uwBoxPlot(allData=Day21CW, trxName="Diet", metricName="weight",
+ LatexFileName=NULL, yLab="Weight in grams", xlab="Diet Type",
+ pTitle="Boxplots of Chick Weight at Day 21 by Diet",pWilcox=FALSE,
+ plotMean=TRUE)
```