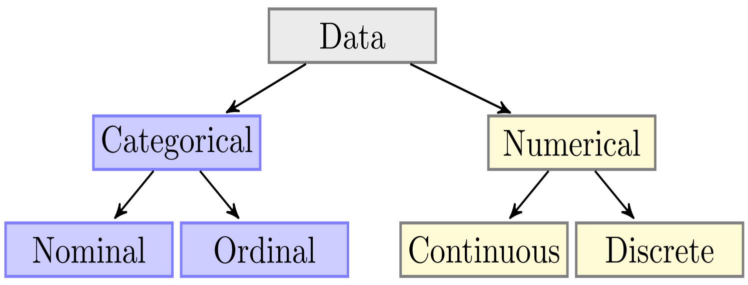
MTH 3270 Notes 1

1 Introduction to Data Science (1.1)

1.1 Variables and Data

- ***Data*** are observed values of ***variables*** (characteristics that vary from one individual to the next).
- Variables can be ***categorical*** (or ***qualitative***) (taking values in a set of categories) or ***numerical*** (or ***quantitative***) (taking numerical values).
- Categorical variables can be:
  - ***Ordinal*** (the categories have an inherent ordering, e.g. low, medium, high)
  - ***Nominal*** (the categories have no inherent ordering, e.g. red, green, and blue).
- Numerical variables can be:
  - ***Discrete*** (can only take integer values).
  - ***Continuous*** (can take any value on a continuum, or interval).



- Two additional types of variables are ***time*** variables (e.g. date) and ***spatial location*** variables (e.g. latitude and longitude).
- **Example**: The South Florida Ecosystem Assessment is a long-term monitoring project in the Florida Everglades initiated by the EPA.

  Environmental and ecological variables were recorded at each of a sample of 757 sites in the Everglades.

Florida Everglades Data

STATID	DECLAT	DECLONG	DATE	WEATHER	SOILTKNSFT	SOILTYPE	CO2SDF
M496	26.6	-80.4	36292	CLEAR	13.0	Peat	4.6
M497	26.6	-80.4	36292	CLEAR	12.6	Peat	4.5
M498	26.6	-80.4	36291	OVERCAST	8.5	Peat	2.2
M499	26.5	-80.4	36291	OVERCAST	8.0	Peat	3.3
M500	26.5	-80.2	36291	OVERCAST	1.3	Peat	1.8
M501	26.5	-80.3	36291	OVERCAST	14.0	Peat	3.4
M502	26.5	-80.3	36291	OVERCAST	9.6	Peat	2.7
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
M746	25.3	-80.6	36425	CLEAR	0.3	Marl	4.5
M747	25.3	-80.8	36425	CLEAR	0.5	Marl	4.3

Below is a description of the variables in the data set:

Variable	Description
STATID	Sampling station name
DECLAT	Latitude (decimal degrees)
DECLONG	Longitude (decimal degrees)
DATE	Sample collection date
WEATHER	Weather conditions
SOILTKNSFT	Soil thickness (ft)
SOILTYPE	Description of soil type
CO2SDF	Carbon Dioxide in soil (log of μmole/g dry weight)

- **DECLAT** and **DECLONG** are **spatial location** variables.
- **DATE** is a **time** variable.
- **WEATHER** and **SOILTYPE** are **categorical** variables.
- **SOILTKNSFT** and **CO2SDF** are **continuous numerical** variables.

1.2 What is data science?

- Combines **computer programming** and **statistics** to glean information from data.
- Incorporates knowledge about the **domain** of the data (e.g. business, biology, health care, climatology, etc.).
- Focused on **large data sets**.
- Data **visualization** and **"wrangling"** are key components.
- Focused on data-based **prediction** (rather than traditional statistical inference, i.e. estimation and hypothesis testing).

1.3 Acknowledgment

- These notes borrow heavily from the book:

  Modern Data Science with R, by Baumer, B. S., Kaplan, D. T., and Horton, N. J., Chapman Hall/CRC Press, 2017.

- The book's companion website is `https://mdsr-book.github.io`.

2 Introduction to R and RStudio (B.Intro-B.1)

2.1 About R

- R is a programming language for statistical and scientific computing, data analysis, graphics, and simulation.
- R is free, open-source software available for download from `http://www.r-project.org`.
- R was created by Ross Ihaka and Robert Gentleman at the University of Aukland, New Zealand in 1995. It's based on the (proprietary) S and S-Plus languages.
- R is distributed and maintained by the **R-Project**, whose financial stability is provided by the nonprofit **R Foundation**.
- The **R Core Team**, a group of about 20 programmers, are the only ones allowed write access to R source code.
- R incorporates features of object-oriented and functional programming languages.
- **R packages** (add-ons) are available for free download and installation. Anyone (e.g. you or me) can contribute an R package.

2.2 About R Studio

- R Studio is an ***integrated development environment*** (***IDE***) for R.
- R Studio is free software, and is available for download from `https://www.rstudio.com/`

3 Getting Started with R (B.2-B.3)

3.1 Arithmetic Operators

- R can be used as a calculator. Arithmetic expressions are typed on the command line in the R Console window, and are evaluated upon hitting 'Enter'.
- The syntax for several mathematical ***operators*** is shown below, ordered from highest to lowest precedence:

```
^                   # Exponentiation (right to left in the case of
                    # "stacked" exponents)
-                   # Unary minus sign
%%                  # Modulo (i.e. remainder)
%/%                 # Integer divide
*  /                # Multiplication, Division
+  -                # Addition, Subtraction
```

- Within an expression, higher precedence operations are carried out first. If two or more operators have equal precedence, they're evaluated from left to right.
- Parentheses, `( )`, can be used to change the order of operations. Operations in parentheses are carried out first.
- For more information on these and other operators, type:

```
help(Syntax)
```

or

```
? Syntax
```

Section 3.1 Exercises

Exercise 1 Guess what the result of each of the following will be, then check your answers:

a) `4 + 2 * 8`

b) `4 + 2 * 8 + 3`

c) `-2^2`

d) `1 + 2^2 * 4`

e) `(2 + 4) / 3 / 2`

3.2 Special Characters, Special Values, Etc.

3.2.1 White Spaces

- Extra spaces are ignored by R. For example, the following produce the same result:

```
2+2
2 + 2
2          +           2
```

3.2.2 Continuing a Command on the Next Line

- If a command isn't complete at the end of a line (and you hit 'Enter' anyway), R will give a different prompt, the `'+'` sign, on subsequent lines and continue to read input until the command is complete:

```
3 + 2 * 8 -
6

## [1] 13
```

If you don't want to complete command, hit the 'Escape' key.

3.2.3 Special Character: `#`

- The `#` symbol is used for comments. Anything after `#` (and in the same line) is not evaluated in R.

3.2.4 Special Values: `Inf` and `NaN`

- Occasionally a computation will result in one of the following special values:

```
Inf                   # Infinity
NaN                   # "Not a number"
```

- Any positive number divided by 0 will result in `Inf`, whereas 0 divided by 0 results in `NaN`.
- `Inf` can be used in calculations and it behaves just like ∞.

Section 3.2 Exercises

Exercise 2 Guess what the result of each of the following will be, then check your answers:

a) `5 / 0`

b) `1 / Inf`

c) `0 / 0`

d) `Inf + 1`

3.3 Variables and the Assignment Operator

3.3.1 Introduction

- In R, ***variables*** (or ***scalars***) are used to store numerical values. We assign values to variables using the ***assignment operator***:

```
<-                    # Assigns a value to a variable
```

For example, below, the value `10` is assigned to the variable `x`:

```
x <- 10
```

- To view the contents of a variable, type its name on the command line and hit 'Enter':

```
x

## [1] 10
```

3.3.2 Variable Naming Conventions

- Variable names can be any length and can contain letters, numbers, and '`.`' and '`_`' characters, but they must begin with a letter or a '`.`'.
- R is *case sensitive*, so `x` and `X` are different symbols and would refer to different variables.

3.3.3 Using Variables in Computations

- Once a value has been assigned to a variable, we can perform computations involving that variable. For example, using the variable `x` from above:

```
(x^2 + 50)/75

## [1] 2
```

3.3.4 Overwriting the Value of a Variable

- We can use the assignment operator `<-` to overwrite the value of a variable:

```
x <- 11
x

## [1] 11
```

Above, the value 10 previously stored in `x` was overwritten by the new value 11.

- The same variable can appear on both sides of an assignment operator. The right side is always evaluated first:

```
x <- x + 1
x

## [1] 12
```

3.3.5 Other Types of Variables

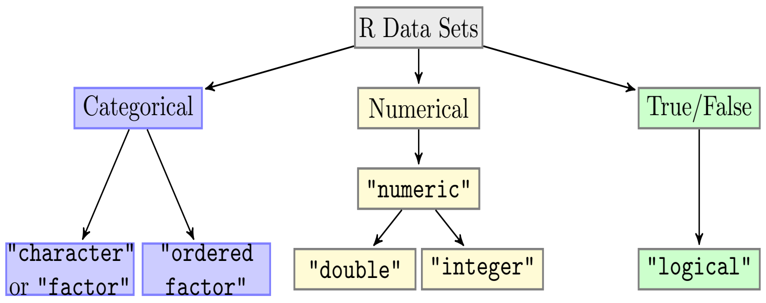
- Variables can store not just numerical values, but any of the so-called ***atomic*** types of values:

"`double`"	"`integer`"
"`character`"	"`logical`"
"`complex`"	"`raw`"

or they can be `NULL`, in which case the variable is interpreted as being empty:

```
NULL                 # Represents an "empty" variable
```

(We'll use quotes, as above, when referring to the *type* of a variable.)



- We can check the type of a variable using the `typeof()` function:

```
typeof()     # Check the type of a variable.  Returns either "double",
             # "integer", "character", "logical", "complex", "raw", or
             # "NULL".
```

- Here are a couple of examples:

```
num.var <- 3.14159
typeof(num.var)

## [1] "double"
```

```
char.var <- "a"
typeof(char.var)

## [1] "character"
```

```
logic.var <- TRUE
typeof(logic.var)

## [1] "logical"
```

- Most numeric variables are "`double`", which stands for *double-precision floating-point*. These variables can store both integer values and non-integer decimal values.

  Occasionally, a numeric variable is "`integer`". These can only store integer values.

  In either case, we can check that a variable is numeric using the `is.numeric()` function:

```
is.numeric()     # Checks to see if a variable is numeric.  Returns
                 # TRUE if the variable is either "double" or "integer"
                 # and FALSE otherwise.
```

- Similarly, we can check that a variable is `"character"`, `"logical"`, or `NULL` using the functions:

```
is.character()   # Checks to see if a variable is "character".
is.logical()     # Checks to see if a variable is "logical".
is.null()        # Checks to see if a variable is NULL.
```

- The last two variable types, `"complex"` and `"raw"`, are rarely encountered in practice.

Section 3.3 Exercises

Exercise 3 What type of variable is created in each of the following commands? Check your answers by typing `typeof(x)`:

a) `x <- 45`

b) `x <- "foo"`

c) `x <- FALSE`

d) `x <- NULL`

Exercise 4 Guess the final value of `x` in the following sequence of commands. Then check your answer.

```
x <- 2
x <- x * 2 + 1
x <- x * 3
x
```

Exercise 5 Write commands that do the following (in order):

1. Create a variable `y` containing the value 5.
2. Overwrite the value of `y` by the value `3 * y`.
3. Copy the value of `y` into a new variable `z`.

3.4 Introduction to Functions

3.4.1 Using Built-In Functions

- R has an extensive set of built-in ***functions***, a few of which are listed below:

```
sqrt()           # Square root
abs()            # Absolute value
round()          # Round a value to a specified number of digits
signif()         # Express a value to a specified number of
                 # significant digits
floor()          # Largest integer not greater than a value
ceiling()        # Smallest integer not less than a value
trunc()          # Truncate a value toward 0
log(); log10()   # Natural logarithm, base 10 logarithm
exp()            # Exponential function (exp(1) is the exponential
                 # constant e, exp(2) is the square of e, etc.)
```

```
factorial()             # Factorial
sin(); cos(); tan()     # Sine, cosine, tangent
```

- Each function accepts one or more values passed to it as ***arguments***, performs computations or operations on those values, and returns a result.
- To perform a ***function call***, type the name of the function with the values of its argument(s) in parentheses, then hit 'Enter':

```
sqrt(2)

## [1] 1.414214
```

Values passed as arguments can be in the form of variables, such as `x` below:

```
x <- 2
sqrt(x)

## [1] 1.414214
```

or they can be entire expressions, such as `x^2 + 5` below:

```
sqrt(x^2 + 5)

## [1] 3
```

- Function calls can be ***nested***, which is useful for combining operations. When they're nested, the inner one is evaluated first and the result then passed to the outer one. For example, below, the call to `abs()` is nested within the call to `sqrt()`. `abs(-4)` is evaluated, and the result passed along to `sqrt()`:

```
sqrt(abs(-4))           # The value returned by abs() is passed to sqrt()

## [1] 2
```

This is equivalent to the following:

```
x <- abs(-4)                # abs() returns (positive) 4
sqrt(x)                     # x is (positive) 4

## [1] 2
```

3.4.2 Getting Help

- Here are some ways to get help for an R function or operator:

```
help()          # Open the built-in html help page for a function (or
                # an operator in quotes, e.g. help("*"))
?               # Open the built-in html help page for a function (or
                # an operator in quotes, e.g. ? "*")
```

- Typing `?` followed by a function name opens the html help page for that function. For example, typing:

```
? sqrt                # We could also type help(sqrt).
```

opens the help page for the function `sqrt()`. Typing `help(sqrt)` would do the same thing.

- Use quotations for help on operators represented by symbols, for example:

```
? "%/%"
```

- If you're not sure how to do something in R, Google usually returns suggestions.

Section 3.4 Exercises

Exercise 6 Look at the help page for `sqrt()` by typing:

```
? sqrt
```

Besides `sqrt()`, what other R function is described on the help page?

3.4.3 Viewing a Function's Arguments

- Most functions take multiple arguments, each of which has a name, and some of which are optional.
- We can see what arguments a function takes and which ones are optional in its help page.
  For example, to see what arguments `round()` takes, we'd type:

  ```
  ? round
  ```

  We see from the help page that `round()` has two arguments, `x`, the numeric value to be rounded, and `digits`, an integer specifying the number of decimal places to round to. Thus to round 4.679 to 2 decimal places, we type:

  ```
  round(x = 4.679, digits = 2)

  ## [1] 4.68
  ```

Section 3.4 Exercises

Exercise 7 Look at the help page for `signif()` by typing:

```
? signif
```

From the help page, how many arguments does `signif()` have?

3.4.4 Optional Arguments and Default Values

- The specification `digits = 0` in the help page for `round()` tells us that `digits` has a ***default value*** of 0. This means that it's an ***optional argument*** and if no value is passed for that argument, rounding is done to 0 decimal places (i.e. to the nearest integer).

Section 3.4 Exercises

Exercise 8 Look at the arguments for `signif()` by typing:

```
? signif
```

The function `signif()` prints the value passed for `x` to the number of significant digits specified by `digits`.

a) From the help page for `signif()`, what is the default value for the `digits` argument?

b) To how many significant digits will the value 342.88937 be printed by the following command?

```
signif(x = 342.88937)
```

3.4.5 Named Argument Matching and Positional Matching

- When we type:

  ```
  round(x = 4.679, digits = 2)

  ## [1] 4.68
  ```

  we specify values for the arguments by their names (`x` and `digits`).

- When ***named argument matching*** is used, as above, the order of the arguments is irrelevant. For example, we get the same result by typing:

  ```
  round(digits = 2, x = 4.679)

  ## [1] 4.68
  ```

- Another way to specify values for the arguments by their positions, for example:

  ```
  round(4.679, 2)
  ```

  R knows, by ***positional matching***, that the first value, 4.679, is the value to be rounded and the second one, 2, is the number of decimal places to round to.

  For example, R would do something completely different if we typed:

  ```
  round(2, 4.679)
  ```

- The two types of argument specification (positional and named argument matching) can be be mixed in the same function call.

Section 3.4 Exercises

Exercise 9 Look again at the arguments for the function `signif()` by typing:

```
? signif
```

The function `signif()` prints the value passed for `x` to the number of significant digits specified by `digits`.

a) Write a command using *named argument matching* that prints the value 342.88937 to 5 significant digits.

b) Write a command using *positional matching* that prints the value 342.88937 to 5 significant digits.

3.5 The R Workspace

3.5.1 Viewing and Removing Objects from the Workspace

- R calls the directory (folder) in which it stores user-created ***objects*** such as variables and data sets the ***Workspace***. To view or remove objects from the Workspace, we use:

```
ls()                # List the objects in the Workspace
objects()           # Same as ls() (lists the objects
                    # in the Workspace)
rm()                # Remove objects from the Workspace
```

- For example, type `ls()` to see what's currently stored in the Workspace:

```
ls()

## [1] "char.var"  "logic.var" "num.var"   "x"
```

  Right now, the only objects in the Workspace are the four variables we created earlier.

  (Typing `objects()` would also list the objects in the Workspace.)

- To remove `x` from the Workspace, use `rm()`:

```
rm(x)
```

  Now we get:

```
ls()

## [1] "char.var"  "logic.var" "num.var"
```

  indicating that `x` no longer exists.

- To remove *all* objects from the Workspace, use:

```
rm(list = ls())
```

3.5.2 "Save the Workspace Image?"

- When you end an R session (for example by typing `q()`) you'll be asked if you want to "Save the Workspace Image?". If you choose to do so, the objects you created in the current R session will be available for re-use in future sessions. Otherwise, they won't.

Section 3.5 Exercises

Exercise 10 Create a few variables named `x`, `y`, and `z` (using any values). Then type the following sequence of commands, paying attention to the output from `ls()` each time:

```
ls()
rm(x)
ls()
rm(list = ls())
ls()
```

What are the outputs from the three calls to `ls()`?

3.6 A Preview of R Data Structures

- There are five ways to store data sets in R. These differ according to their dimensionality (1D, 2D, or nD) and whether they're *homogeneous* (all contents must be of the same type) or *heterogeneous* (contents can be of different types):
  - **Vectors** (1D, homog.)
  - **Lists** (1D, heterog.)
  - **Matrices** (2D, homog.)
  - **Data Frames** (2D, heterog.)
  - **Arrays** (nD, homog.)

3.6.1 A Preview of Vectors

- ***Vectors*** (sometimes called ***atomic vectors***) are created using the "combine" function:

```
c()                      # Combine values to form a vector
```

- Here's an example:

```
num.vec <- c(7, 4, 5)
num.vec

## [1] 7 4 5
```

- Vectors can store any of the *atomic* types. Here's one that stores `"character"` values:

```
char.vec <- c("a", "b", "c")
char.vec

## [1] "a" "b" "c"
```

  and here's one that stores `"logical"`:

```
logic.vec <- c(TRUE, TRUE, FALSE)
logic.vec

## [1]  TRUE  TRUE FALSE
```

- The functions `typeof()`, `is.numeric()`, `is.character()`, and `is.logical()` work on vectors too:

```
typeof(logic.vec)

## [1] "logical"
```

3.6.2 A Preview of Matrices

- ***Matrices*** are like two-dimensional vectors (i.e. they have rows and columns). One way to create a matrix is using:

```
matrix()                 # Create a matrix, from a vector, with a speci-
                         # fied number of rows and columns
```

- Here's an example:

```
my.mat <- matrix(c(1, 2, 3, 4, 5, 6, 7, 8, 9), nrow = 3, ncol = 3)
my.mat

##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
```

Note that by default, R fills the matrix by columns, left to right.

3.6.3 A Preview of Lists

- The elements of an *atomic* vector all have to be the same type. ***Lists*** are vectors whose elements can be different types. One way to create a list is using:

```
list()               # Create a list from a set of R objects
```

- Here's an example:

```
my.list <- list("d", 12, TRUE)
my.list

## [[1]]
## [1] "d"
##
## [[2]]
## [1] 12
##
## [[3]]
## [1] TRUE
```

- In fact, elements of a list can be *any* R objects, for example vectors:

```
my.list <- list(c("a", "b", "c"), c(7, 4, 5))
my.list

## [[1]]
## [1] "a" "b" "c"
##
## [[2]]
## [1] 7 4 5
```

- The elements of a list can even be other lists:

```
my.list <- list(list("a", 7), list("b", 4), list("c", 5))
```

3.6.4 A Preview of Data Frames

- ***Data frames*** are like matrices, but they can have a mix of `"character"` (or `"factor"`) (i.e. *categorical*) columns and `"numeric"` ones.

  Each *column* of a data frame is a *vector*.

- One way to create a data frame is with the function:

```
data.frame()          # Create a data frame from a set of vectors
                      # (which will form the columns of the data frame)
```

- We can choose names for the columns of a data frame. In the example below, we use `Var1` and `Var2` for the column names:

```
my.df <- data.frame(Var1 = c("a", "b", "c"), Var2 = c(7, 4, 5))
my.df

##   Var1 Var2
## 1    a    7
## 2    b    4
## 3    c    5
```

3.6.5 A Preview of Arrays

- An ***array*** is like a matrix, but it can have more than two dimensions (e.g. rows, columns, and layers). We can create an array using:

```
array()            # Create an array, from a vector, with a specified
                   # number of dimensions
```

- Array's aren't used very much in R.

Section 3.6 Exercises

Exercise 11 Write a command using `c()` that creates a vector containing the values:

$$3, 7, 2, 8$$

Exercise 12 Write a command using `matrix()` that creates the following matrix:

$$\begin{pmatrix} 2 & 6 \\ 4 & 8 \end{pmatrix}$$

Exercise 13 Write a command using `list()` that creates a list containing the following elements:

"e", 9, TRUE

Exercise 14 Write a command using `data.frame()` that creates a data frame containing the following data set:

Category	Value
A	5
A	4
B	6
B	6
C	9
C	8

4 Vectors (B.4)

4.1 Creating and Examining Vectors

- The following functions will be used to create and examine vectors:

```
c()             # Create a vector of values
length()        # Returns the number of elements in a vector
is.vector()     # Indicates whether or not an object is a vector
```

- Here's an example:

```
x <- c(7, 4, 5)
length(x)

## [1] 3

is.vector(x)

## [1] TRUE
```

- We can also use `c()` to combine two (or more) existing vectors end-to-end to create a new vector:

```
x <- c(7, 4, 5)
y <- c(1, 2, 3)
```

```
my.new.vec <- c(x, y)
```

```
my.new.vec

## [1] 7 4 5 1 2 3
```

- A (scalar) variable is actually a one-element vector:

```
x <- 7
is.vector(x)

## [1] TRUE

length(x)

## [1] 1
```

4.2 Vector Arithmetic and Recycling

- When we perform arithmetic operations ('+', '-', '*', '/', and '^') on two vectors, their elements are matched and the operation is performed one pair of elements at a time:

```
x <- c(7, 4, 5)
y <- c(1, 2, 3)
```

```
x + y

## [1] 8 6 8
```

- If the vectors have different lengths, the shorter one is repeated as necessary, i.e. its values are ***recycled***, and R prints a warning message:

```
z <- c(1, 2, 3, 4, 5)
y <- c(1, 2, 3)
```

```
z - y

## Warning in z - y:  longer object length is not a multiple of shorter object length

## [1] 0 0 0 3 3
```

Above, because `y` is shorter than `z`, the elements of `y` are recycled until the two vectors are of equal length. This is equivalent to subtracting `c(1, 2, 3, 1, 2)` from `z`, i.e.:

```
c(1, 2, 3, 4, 5) - c(1, 2, 3, 1, 2)

## [1] 0 0 0 3 3
```

Section 4.2 Exercises

Exercise 15 Guess what the result of each of the following will be, then check your answers:

a)
```
x <- c(2, 3, 4, 5)
y <- c(6, 7, 8, 9)
c(x, y)
```

b)
```
x <- c(2, 3, 4, 5)
y <- c(6, 7, 8, 9)
x + y
```

Exercise 16 Guess what the result of each of the following will be, then check your answers:

a)
```
x <- c(2, 3, 4, 5)
x + 1
```

b)
```
x <- c(2, 3, 4, 5)
x * 2
```

Exercise 17 Guess what the result of the following will be, then check your answer:

```
y <- c(6, 7, 8)
z <- c(2, 3)
y + z
```

Exercise 18 In R, single-valued variables and constants are vectors of length one. Guess what the result of each of the following will be, then check your answers:

a)
```
x <- 2
is.vector(x)
```

b)
```
is.vector(2)
```

4.3 Vector Coercion

- All elements of a vector must be of the same type, so if you try to combine vectors of different types, they'll be ***coerced*** to the *most flexible* type. Types from least to most flexible are:

Least Flexible	`"logical"`
↓	`"integer"`
	`"double"`
Most Flexible	`"character"`

- In particular, if we combine a numeric vector (`"double"`) with a `"character"` vector, the numerical values are coerced to `"character"`:

```
num.vec <- c(4, 7, 5)
char.vec <- c("a", "b")
c(num.vec, char.vec)

## [1] "4" "7" "5" "a" "b"
```

- If we combine a `"logical"` vector with a numeric vector (`"double"`), `TRUE` is coerced to 1 and `FALSE` to 0:

```
c(TRUE, FALSE, 7, 8)

## [1] 1 0 7 8
```

Section 4.3 Exercises

Exercise 19 Guess what the result of each of the following will be, then check your answers:

a)
```
x <- c(2, 3, "a")
x
```

b)
```
x <- c(2, 3, TRUE)
x
```

c)
```
x <- c("a", "b")
y <- c(FALSE, TRUE)
c(x, y)
```

4.4 Common Vector Operations

4.4.1 Vector Indexing Using `[ ]`

Accessing Vector Elements

- We access one or more elements of a vector using their indices in square brackets:

```
[ ]                    # Access vector elements via their indices
```

- For example, typing `x[3]` returns the 3rd element of a vector `x`:

```
x <- c(5, 7, 9, 8, 1)
x[3]

## [1] 9
```

and typing `x[c(3, 4)]` returns the 3rd and 4th elements:

```
x[c(3, 4)]

## [1] 9 8
```

Replacing Vector Elements

- We can also use the brackets `[ ]` to *replace* specific values in `x`:

```
x[3] <- 13
```

```
x

## [1]  5  7 13  8  1
```

Deleting Vector Elements

- A negative index returns all but that element from the vector. For example to obtain all but the 5th element of `x`, type:

```
x[-5]

## [1]  5  7 13  8
```

If we want to permanently delete the 5th element, we need to overwrite `x` by `x[-5]`:

```
x <- x[-5]
```

Rearranging Vector Elements

- One way to rearrange (permute) the elements of a vector is to specify the desired permutation in square brackets. For example, consider the vector `y`:

```
y

## [1] 11 18 15
```

If we want its elements in the order 18, 15, 11, we type:

```
y[c(2, 3, 1)]

## [1] 18 15 11
```

Above, the vector `c(2, 3, 1)` indicates that we want the 2nd element of `y` moved to the first position, the 3rd element to the second position, and the 1st element to the third position.

Other Ways of Rearranging the Elements of a Vector

- Here are some other functions that can be used to rearrange the elements of a vector:

```
sort()              # Returns the elements of a vector in sorted order
rev()               # Returns the elements of a vector in reverse order
order()             # Returns a vector of indices such that x[order(x)]
                    # returns the vector x in sorted order
```

4.4.2 Introduction to Filtering

- We can use a `"logical"` vector inside square brackets to ***filter*** out certain elements of a vector `x`:

```
x <- c(5, 7, 9, 8, 1)
x[c(TRUE, FALSE, TRUE, TRUE, FALSE)]

## [1] 5 9 8
```

Above, only the elements of `x` corresponding to `TRUE` in the `"logical"` vector are returned.

4.4.3 Creating More Specialized Vectors with `seq()`, `:`, and `rep()`

- The functions and operator below are useful for creating sequences and repeating patterns of values:

```
seq()           # Create a sequence of values
:               # Create a sequence of integers
rep()           # Create a repeating pattern of values
```

Creating Sequences of Values Using `seq()` and ':'

- `seq()` creates a sequence of values starting at `from` and ending at `to`, with increment `by`:

```
seq(from = 1, to = 5, by = 0.5)

## [1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0
```

- The colon operator ':' can be used to create a sequence of consecutive *integers*. For example:

```
1:10

##  [1]  1  2  3  4  5  6  7  8  9 10
```

produces the same result as:

```
seq(from = 1, to = 10, by = 1)
```

Creating Repeating Patterns of Values Using `rep()`

- `rep()` takes a value (via its first argument `x`) and repeats it a specified number of times (via `times`):

```
rep(1, times = 10)

##  [1] 1 1 1 1 1 1 1 1 1 1
```

- We can use `rep()` with `"character"` values too:

```
rep("a", times = 10)

##  [1] "a" "a" "a" "a" "a" "a" "a" "a" "a" "a"
```

- When the first argument is a vector, `rep()` repeats that vector end-to-end the specified number of `times`:

```r
rep(1:3, times = 2)

## [1] 1 2 3 1 2 3
```

Section 4.4 Exercises

Exercise 20 Consider the vector

```r
x <- c(7, 6, 4, 2, 3, 5)
```

Guess what the result of each of the following will be, then check your answers:

a)
```r
x[2]
```

b)
```r
x[-2]
```

c)
```r
x[c(1, 2)]
```

d)
```r
x[c(2, 1)]
```

e)
```r
x[1] <- 5
x
```

Exercise 21 Consider the vector

```r
x <- c(7, 6, 4, 2, 3, 5)
```

a) Write a command that returns the 4th element of `x`.

b) Write a command that replaces the 4th element of `x` with the value 1.

c) Write a command that returns all but the 6th element of `x`.

Exercise 22 Consider the vector

```r
x <- c(7, 6, 4, 2)
```

Recall, one way to rearrange (permute) the elements of a vector is to specify the desired permutation in square brackets Guess what the result of the following will be, then check your answer:

```r
x[c(2, 1, 3, 4)]
```

Exercise 23 Consider the vector

```r
x <- c(7, 6, 4, 2, 3, 5)
```

a) Write a command using `sort()` that returns the elements of `x` sorted in ascending order.

b) Write a command using `rev()` that returns the elements of `x` in reverse order.

c) Look at the help page for `sort()` by typing

```
? sort
```

Notice the `sort()` has an optional argument, `decreasing`, whose default value is `FALSE`. Write a command using `sort()` that returns the elements of `x` in *descending* order by setting `decreasing = TRUE`.

Exercise 24 Consider the vector

```
x <- c(7, 6, 4, 2, 3, 5)
```

Guess what the result of the following will be, then check your answer:

```
x[c(FALSE, FALSE, TRUE, FALSE, FALSE, TRUE)]
```

Exercise 25 Guess what the result of each of the following will be, then check your answers:

a)
```
1:5
```

b)
```
6:10
```

c)
```
5:1
```

Exercise 26 Guess what the result of the following will be, then check your answer:

```
is.vector(1:5)
```

Exercise 27 Guess what the result of each of the following will be, then check your answers:

a)
```
seq(from = 1, to = 2.5, by = 0.5)
```

b)
```
seq(from = 2.5, to = 1, by = -0.5)      # Note that by is negative
```

Exercise 28 Guess what the result of each of the following will be, then check your answers:

a)
```
rep(2, times = 3)
```

b)
```
rep(1:2, times = 3)
```

4.5 Comparison Operators

- R has several ***comparison operators*** that can be used on (scalar) variables as well as on vectors:

```
<                   # Less than
>                   # Greater than
==                  # Equal to
!=                  # Not equal to
<=                  # Less than or equal to
>=                  # Greater than or equal to
```

- Each one returns a `"logical"` value, `TRUE` or `FALSE`, depending on whether or not the relationship holds. Here are some examples:

```
5 < 6

## [1] TRUE

5 == 6

## [1] FALSE

5 != 6

## [1] TRUE
```

- They can be used on `"character"` values too:

```
"a" == "b"

## [1] FALSE
```

- When applied to vectors, they operate elementwise and return a `"logical"` vector:

```
x <- c(11, 3, 4, 12, 2)
y <- c(11, 9, 4, 12, 2)
```

```
x == y

## [1]  TRUE FALSE  TRUE  TRUE  TRUE
```

- They can also be used to compare each element of a vector to a single value. For example (using `x` from above):

```
x > 10

## [1]  TRUE FALSE FALSE  TRUE FALSE
```

- Recall that R coerces `TRUE` and `FALSE` to 1 and 0, respectively, when it needs to. This is *very* useful, in combination with the function `sum()`, when we want to count *how many* elements of a vector satisfy a given condition (using `x` from above):

```
sum(x > 10)           # Counts how many elements of x are > 10.

## [1] 2
```

Section 4.5 Exercises

Exercise 29 Consider the following vector:

```
x <- c(3, 4, 10)
```

Guess what the result of each of the following will be, then check your answers:

a) `x == 4`

b) `x > 4`

c) `x >= 4`

d) `x != 4`

Exercise 30 Consider the following two vectors:

```
x <- c(3, 4, 10)
y <- c(3, 4, 5)
```

Guess what the result of each of the following will be, then check your answers:

a) `x == y`

b) `x != y`

Exercise 31 Recall that R coerces `TRUE` and `FALSE` to 1 and 0, respectively, when it needs to. Guess what the result of each of the following will be, then check your answers:

a) `TRUE + TRUE + FALSE + FALSE + FALSE`

b) `sum(c(TRUE, TRUE, FALSE, FALSE, FALSE))`

Exercise 32 Consider the following vector:

```
x <- c(10, 8, -2, -6, -5)
```

Guess what will be returned by of each of the following commands, then check your answers:

a) `x > 0`

b) `sum(x > 0)`

4.6 Using `any()`, `all()`, and `which()`, `which.min()`, and `which.max()`

- The following are useful for searching vectors for values that satisfy a given condition:

```
any()        # Do any elements of a vector satisfy a given condition?
             # Returns TRUE or FALSE.
all()        # Do all of the elements of a vector satisfy a given
             # condition?  Returns TRUE or FALSE.
which()      # Returns the indices of the elements of a vector that
             # satisfy a given condition.
which.min()  # Returns the index of the minimum value in a vector.
which.max()  # Like which.min(), but returns the index of the maximum.
%in%         # Is a given value contained in a vector?
```

- `any()` tells us whether *any* values in a vector satisfy a certain condition:

```
x <- c(11, 3, 4, 12, 2)
any(x > 10)

## [1] TRUE
```

- `which()` determines *which* elements satisfy the condition, and returns their *indices*:

```
which(x > 10)

## [1] 1 4
```

  Thus we see that the 1st and 4th elements of `x` are greater than 10.

- `all()` tells us whether or not *all* of the values in a vector satisfy a condition:

```
all(x > 10)

## [1] FALSE
```

- We can use `any()`, `all()`, and `which()` for comparing *two* vectors. For example, consider the vectors:

```
x <- c(11, 3, 4, 12, 2)
y <- c(11, 9, 4, 12, 2)
```

  If we want to know which of the values in `x` are different from their corresponding value in `y`, we type:

```
which(x != y)

## [1] 2
```

  We see that only the 2nd values of `x` and `y` differ.

- `which.min()` and `which.max()` return the *indices* of the smallest and largest values in a vector. For example

```
which.max(x)

## [1] 4
```

  tells us that the largest value in `x` is the 4th value (12).

- If multiple elements are tied for the smallest (or largest) value, `which.min()` (or `which.max()`) only returns the index of the first one.

- The operator `%in%` returns `TRUE` or `FALSE` depending on whether a given value is in a vector. For example:

```
3 %in% x

## [1] TRUE
```

Section 4.6 Exercises

Exercise 33 Consider the vector:

```
x <- c(2, 8, 6, 7, 1, 4, 9)
```

Guess what each of the following commands will return, then check your answers:

a) `any(x == 4)`

b) `all(x == 4)`

c) `which(x == 4)`

d) `which(x != 4)`

Exercise 34 Consider the vectors:

```
x <- c(53, 42, 64, 71, 84, 62, 95)
y <- c(53, 41, 68, 71, 81, 66, 65)
```

a) Write a command involving `any()` and `==` to determine if *any* of the values in `x` are equal to their corresponding value in `y`.

b) Write a command involving `all()` and `==` to determine if *all* of the values in `x` are equal to their corresponding value in `y`.

c) Write a command involving `which()` and `==` to determine *which* of the values in `x` are equal to their corresponding value in `y`.

Exercise 35 Consider the vector:

```
x <- c(53, 42, 64, 71, 84, 62, 95)
```

Guess what the result of each of the following commands will be, then check your answers:

a) `which.min(x)`

b) `which.max(x)`

4.7 Computing Summary Statistics

- Several functions take vector arguments and compute summary statistics. Here are a few of them:

```
length()          # Number of elements in a vector
min(); max()      # Smallest and largest values in a vector
mean()            # The sample mean
median()          # Sample median
range()           # Range (smallest and largest values) of a vector
sd(); var()       # Sample standard deviation and variance
sum()             # The sum of the values in a vector
prod()            # The product of the values in a vector
cumsum()          # Cumulative sums of the values in a vector
cumprod()         # Cumulative products of the values in a vector
mad()             # Median absolute deviation
quantile()        # Sample quantile (percentile)
IQR()             # Interquartile range
summary()         # Five number summary (and sample mean)
```

- For example:

```r
x <- c(11, 3, 4, 12, 2)
mean(x)

## [1] 6.4

sd(x)

## [1] 4.722288

min(x)

## [1] 2
```

Section 4.7 Exercises

Exercise 36 Consider the following data set:

```r
x <- c(10, 147, 7, 6, 7, 12, 9, 12, 11, 8)
```

a) Use `mean()` to compute the mean.

b) Use `median()` to compute the median.

c) Use `sd()` to compute the standard deviation of `x`

Exercise 37 The standard deviation measures variation in a set of data.

a) What do you think the standard deviation of the following data set will be? Check your answer using `sd()`.

```r
u <- c(5, 5, 5, 5, 5)
```

b) Which of the following two data sets do you think will have a larger standard deviation? Check your answer using `sd()`.

```r
u <- c(5, 6, 7)
v <- c(1, 6, 11)
```

4.8 Vectorized Computations

- We've seen that the operators '+', '-', '*', '/', and '^' operate *elementwise* on vectors.
- Many of R's built-in functions operate one element at a time too. For example, watch what happens when we pass a vector to `sqrt()`:

```
x <- c(4, 9, 16, 25)
sqrt(x)

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

- Functions that perform calculations on vectors one element at a time are said to be ***vectorized***.

Section 4.8 Exercises

Exercise 38 The function `abs()` takes the absolute value of a number. It's a *vectorized* function. Guess what the result of the following command will be, then check your answer:

```
x <- c(-1, 3, -4, -2)
abs(x)
```

Exercise 39 Consider the following temperature measurements, in degrees Celsius:

```
degreesC <- c(23, 19, 21, 22, 18, 20, 24, 25)
```

The relation between Celsius (°C) and Fahrenheit (°F) is:

$$^\circ\mathrm{F} = \frac{9}{5} \times {}^\circ\mathrm{C} + 32$$

Recall that arithmetic operators such as `*` and `+` are *vectorized*. Describe in words what the following command will do to the Celsius temperatures? Try it.

```
degreesF <- (9/5) * degreesC + 32
```

4.9 Filtering

- ***Filtering*** refers to extracting from a vector those elements for which some condition is met.

4.9.1 Extracting and Replacing Elements that Satisfy Some Condition

- To extract (filter) from a vector the elements that satisfy some condition, we just state the condition inside square brackets `[ ]`. For example:

```
x <- c(1, 3, 12, 5, 13)
x[x > 10]                          # Note that x > 10 is a "logical" vector
```

```
## [1] 12 13
```

  Above, we extracted the elements of `x` that are greater than 10. Note that the expression `x > 10` is actually a `"logical"` vector.

- We can also use square brackets to *replace* elements that satisfy some condition. For example, to replace all of values in `x` that are greater than 10 by, say, 11, we can type:

```
x[x > 10] <- 11
x

## [1]  1  3 11  5 11
```

4.9.2 Using Values in One Vector to Extract Elements from Another

- Sometimes we need to extract (filter) from one vector the elements for which the values in another vector satisfy some condition. For example, suppose we have heights (inches) and weights (lbs) of 12 people:

```
ht <- c(69, 71, 67, 66, 72, 71, 61, 65, 73, 70, 68, 74)
wt <- c(175, 170, 210, 190, 195, 165, 163, 172, 158, 191, 213, 215)
```

  To find the weights of the people who are 72 inches tall or taller, we type:

```
wt[ht >= 72]                    # Note that ht >= 72 is a "logical" vector

## [1] 195 158 215
```

- This method is extremely useful with `"character"` vectors indicating group membership. For example, consider this data set:

Gender	Age
f	33
m	35
f	29
m	34
m	37
f	36
f	35
f	40
m	43
f	38
f	40
m	44

  After creating the vectors:

```
Gender <- c("f", "m", "f", "m", "m", "f", "f", "f", "m", "f", "f", "m")
Age <- c(33, 35, 29, 34, 37, 36, 35, 40, 43, 38, 40, 44)
```

  we can extract the ages of just the females by typing:

```
Age[Gender == "f"]

## [1] 33 29 36 35 40 38 40
```

4.9.3 Extracting Elements that Satisfy Some Condition Using `subset()`

- Another way to extract (filter) from a vector the elements that satisfy some condition is to use:

```
subset()          # Extract a subset of vector elements that satisfy a
                  # given condition
```

- `subset()` takes arguments `x`, a vector, and `subset`, a `"logical"` vector specifying the condition to be met by the elements extracted from `x`. For example, to (again) extract the elements from `x` that are greater than 10, we can type:

```
x <- c(1, 3, 12, 5, 13)
subset(x, subset = x > 10)

## [1] 12 13
```

Section 4.9 Exercises

Exercise 40 Consider again the vector:

```
x <- c(538, 432, 684, 716, 814, 624, 956)
```

a) Guess values will be returned by the following command, then check your answer:

```
x[x > 700]
```

b) Guess values will be returned by the following command, then check your answer:

```
subset(x, subset = x > 700)
```

c) Write a command involving square brackets `[ ]` that extracts from `x` all the values that are *not equal to* 814. **Hint**: Use the the comparison operator `!=`.

Exercise 41 Consider this data set, showing the genders, ages, and systolic blood pressures for 12 people:

Gender	Age	Blood Pressure
f	33	118
m	35	115
f	29	110
m	34	117
m	37	112
f	36	119
f	35	114
f	40	121
m	43	123
f	38	117
f	40	120
m	44	121

Here are the same data:

```
Gender <- c("f", "m", "f", "m", "m", "f", "f", "f", "m", "f", "f", "m")
Age <- c(33, 35, 29, 34, 37, 36, 35, 40, 43, 38, 40, 44)
BP <- c(118, 115, 110, 117, 112, 119, 114, 121, 123, 117, 120, 121)
```

One of the commands below extracts the blood pressures of just the males. The other one extracts the ages of people whose blood pressures exceed 117. Which does which?

```
Age[BP > 117]
BP[Gender == "m"]
```

4.10 `NA` Values

4.10.1 Introduction

- Data sets often contain missing values, for example when a survey question was left unanswered or a laboratory measurement failed due to equipment malfunction.

- In R, missing values are represented by `NA`, which stands for ”not available”:

```
NA                      # Indicates a missing value ("not available")
```

- We can test for a missing value using `is.na()`:

```
is.na()                 # Returns TRUE or FALSE depending on whether or
                        # not a value is NA
```

- When applied to a vector, `is.na()` returns a `"logical"` vector whose elements are `TRUE` or `FALSE` depending on whether the corresponding value in the original vector is `NA`:

```
x <- c(2.1, 4.1, NA, 4.4, 3.7)
is.na(x)

## [1] FALSE FALSE  TRUE FALSE FALSE
```

- To decide *which* values in a vector are missing, we can type:

```
which(is.na(x))

## [1] 3
```

4.10.2 Computing Summary Statistics from Data with `NA`s

- It helps to think of `NA`s as ”unknown” values rather than ”missing” values. This explains R’s behavior involving `NA`s.

  For example, most of the R functions for computing summary statistics return `NA` when passed a data set that contains `NA`s:

```
x <- c(2, 4, NA, 5, 7)
mean(x)

## [1] NA
```

- Some functions have an optional argument `na.rm` that can be used to remove the `NA`s before carrying out the calculations:

```
mean(x, na.rm = TRUE)

## [1] 4.5
```

4.10.3 Extracting and Replacing `NA`s

- R treats `NA`s as *unknown* values. For example, below, `2 == NA` isn’t `FALSE`, but `NA`, because R doesn’t have enough information to determine what the value on the right side is:

```
2 == NA

## [1] NA
```

  Likewise `NA == NA` isn’t `TRUE` but `NA`:

```
NA == NA

## [1] NA
```

So, when a vector has one or more NAs, we can't use '==' to check which elements are NAs:

```
x <- c(2, 4, NA, 5, 7)
x == NA

## [1] NA NA NA NA NA
```

Therefore we can't replace NAs by, say, 0 by typing `x[x == NA] <- 0`.

- But typing:

```
is.na(x)

## [1] FALSE FALSE  TRUE FALSE FALSE
```

tells us which elements are NAs, and therefore, to replace the NAs by 0, we can type:

```
x[is.na(x)] <- 0          # Note that is.na(x) is a "logical" vector.
x

## [1] 2 4 0 5 7
```

(Be aware that depending on the context, replacing NAs by 0 might not be appropriate.)

Section 4.10 Exercises

Exercise 42 Guess what the result of each of the following will be, then check your answers.

a)
```
3 == NA
```

b)
```
NA == NA
```

Exercise 43 Consider the vector:

```
x <- c(1, 2, NA)
```

Guess what the result of each of the following will be, then check your answers.

a)
```
is.na(x)
```

b)
```
x[is.na(x)] <- 0          # Note that is.na(x) is a "logical" vector.
x
```

Exercise 44 Consider the following vector:

```
x <- c(1, 2, NA)
```

a) Guess what the result of the following command will be, then check your answer:

```
sum(x)
```

b) The `sum()` function has an optional argument `na.rm` that, when set to `TRUE`, removes `NA`s before computing the sum. Guess what the result of the following command will be, then check your answer:

```
sum(x, na.rm = TRUE)
```

5 Matrices (B.4)

5.1 Creating and Examining Matrices

- ***Matrices*** are one way of storing data in a two-dimensional layout (i.e. in rows and columns). They're easily created and examined in R using the functions:

```
matrix()            # Create a matrix, from a vector, with nrow rows
                    # and ncol columns
dim()               # Returns the dimensions (number of rows and
                    # columns) of a matrix
nrow(); ncol()      # Number of rows, number of columns of a matrix
is.matrix()         # Indicates whether or not an object is a matrix
```

- Here's an example showing how to create a matrix from a vector (`1:9`) using `matrix()`:

```
x <- matrix(1:9, nrow = 3, ncol = 3)
x

##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9

is.matrix(x)

## [1] TRUE

dim(x)

## [1] 3 3
```

Note that `dim()` returns a *vector* with two element giving the number of rows (first element) and number of columns (second element).

- By default, the matrix is created by filling in its *columns*, left to right. By setting the optional argument `byrow = TRUE`, we can create it by filling its *rows*:

```
x <- matrix(1:9, nrow = 3, ncol = 3, byrow = TRUE)
x

##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6
## [3,]    7    8    9
```

- If there are too few elements in the vector passed to `matrix()` to fill the matrix, the vector elements are recycled until the matrix has been filled.

If the length of the vector isn't a multiple (or sub-multiple) of the number of rows (or columns if `byrow = TRUE`), a warning message is printed:

```
x <- matrix(c(1, 2, 3, 4), nrow = 3, ncol = 3)

## Warning in matrix(c(1, 2, 3, 4), nrow = 3, ncol = 3):  data length [4] is not a sub-multiple
or multiple of the number of rows [3]
```

```
x

##      [,1] [,2] [,3]
## [1,]    1    4    3
## [2,]    2    1    4
## [3,]    3    2    1
```

- If only one of `nrow` or `ncol` is specified, R infers the dimensions of the matrix based on the length of the vector.
- We can also create matrices using:

```
cbind()            # Create a matrix by "binding" vectors together
                   # in columns.
rbind()            # Like cbind(), but "binds" vectors together in
                   # rows.
```

- `cbind()` "binds" two or more vectors of the same length into columns of a matrix, and `rbind()` "binds" them into rows. For example:

```
x <- cbind(c(1,2,3), c(4,5,6), c(7,8,9))
x

##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
```

Section 5.1 Exercises

Exercise 45 Here's a matrix `x`:

```
x <- matrix(c(8, 8, 8, 4, 4, 4), nrow = 2, byrow = TRUE)
x

##      [,1] [,2] [,3]
## [1,]    8    8    8
## [2,]    4    4    4
```

Guess what the result of each of the following will be, then check your answers.

a) `dim(x)`

b) `nrow(x)`

c) `ncol(x)`

Exercise 46 If there are too few elements in the vector passed to `matrix()` to fill the matrix, the vector elements are recycled until the matrix has been filled.

If the length of the vector isn't a multiple (or sub-multiple) of the number of rows (or columns if `byrow = TRUE`), a warning message is printed.

What happens when you run the following command?

```
matrix(c(1, 2, 3), nrow = 4, ncol = 2)
```

Exercise 47 Here's a matrix `x`:

```
##      [,1] [,2]
## [1,]    5    5
## [2,]    4    4
```

Do you think the following commands will all produce the matrix above? Check your answer.

```
x <- matrix(c(5, 4, 5, 4), nrow = 2, ncol = 2)
x <- cbind(c(5, 4), c(5, 4))
x <- rbind(c(5, 5), c(4, 4))
```

5.2 General Matrix Operations

5.2.1 Matrix Arithmetic and Recycling

Matrix Arithmetic

- The operators '+', '-', '*', '/', and '^' operate on matrices elementwise. For example, using the matrices `x` and `y`,

```
x <- matrix(c(5, 5, 5, 4, 4, 4, 3, 3, 3), nrow = 3, byrow = TRUE)
y <- cbind(c(1, 1, 1), c(2, 2, 2), c(3, 3, 3))
```

```
x

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

y

##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    1    2    3
## [3,]    1    2    3
```

  we get

```
x + y

##      [,1] [,2] [,3]
## [1,]    6    7    8
## [2,]    5    6    7
## [3,]    4    5    6
```

```
x + 2

##      [,1] [,2] [,3]
## [1,]    7    7    7
## [2,]    6    6    6
## [3,]    5    5    5
```

- The built-in R functions that are vectorized, such as `log()`, `sqrt()`, `cos()`, etc. also operate elementwise on matrices.

Recycling

- We can also carry out arithmetic operations ('+', '-', '*', '/', and '^') using a *vector* and a matrix. For example:

```
x.mat

##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9

y.vec

## [1] 1 2 3 4 5 6 7 8 9

x.mat - y.vec

##      [,1] [,2] [,3]
## [1,]    0    0    0
## [2,]    0    0    0
## [3,]    0    0    0
```

Notice that elements of the vector are matched with the columns of the matrix, from left to right.

- If the vector is too short, its elements are recycled. If the length of the vector isn't a sub-multiple of the total number of elements in the matrix, a warning message is printed.

5.2.2 Matrix Indexing Using `[ ]`

Accessing Matrix Elements, Rows, or Columns

- We access matrix elements, rows, or columns using square brackets:

```
[ , ]         # Access matrix elements via their row and column indices
              # (separated by a comma)
```

- To extract a specific element, specify its row and column indices in square brackets `[ ]`, separated by a comma. For example, using the matrix `x`:

```
x <- matrix(1:9, nrow = 3, ncol = 3)
x

##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
```

we extract the element in the 2nd row and 3rd column by typing:

```
x[2, 3]                        # Extract the element in the 2nd row, 3rd column of x

## [1] 8
```

- An entire row is accessed by leaving the column index blank. For example the 2nd row of `x` is obtained via:

```
x[2, ]                         # Extract the 2nd row of x

## [1] 2 5 8
```

Likewise, an entire column is accessed by leaving the row index blank:

```
x[, 3]                         # Extract the 3rd column of x

## [1] 7 8 9
```

Replacing Matrix Elements, Rows, or Columns

- The assignment operator `<-` can be used to assign a new value to a matrix element:

```
x[2, 3] <- 0                        # Replace the element in the 2nd row, 3rd column by 0
x

##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    0
## [3,]    3    6    9
```

- Here's how to replace an entire column:

```
x[, 3] <- c(0, 0, 0)                # Replace the 3rd column by 0's
x

##      [,1] [,2] [,3]
## [1,]    1    4    0
## [2,]    2    5    0
## [3,]    3    6    0
```

Replacing an entire row is done in a similar manner, but the row index is specified *before* the comma in the square brackets.

Adding and Deleting Matrix Rows or Columns

- Negative indices in square brackets (e.g. `x[-1, ]`) return all but that row or column of a matrix.
- We can add a row or column to an existing matrix using `rbind()` and `cbind()`:

```
x <- matrix(1:9, nrow = 3, ncol = 3)
x

##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
```

```
cbind(x, c(10, 10, 10))

##      [,1] [,2] [,3] [,4]
## [1,]    1    4    7   10
## [2,]    2    5    8   10
## [3,]    3    6    9   10
```

Rearranging Matrix Rows or Columns

- Square brackets can also be used to rearrange (permute) the rows or columns of a matrix. To do so, we specify the desired permutation before or after the comma, depending on whether we're rearranging rows or columns. For example:

```
x

##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9

# This rearranges columns:  3rd column goes first, 1st column goes second, 2nd column
# goes third
x[, c(3, 1, 2)]

##      [,1] [,2] [,3]
## [1,]    7    1    4
## [2,]    8    2    5
## [3,]    9    3    6
```

Section 5.2 Exercises

Exercise 48 Consider the following matrix:

```
x <- matrix(1:9, nrow = 3, ncol = 3)
x

##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
```

Guess what the result of each of the following will be, then check your answers.

a) `x[1, 3]`

b) `x[1, ]`

c) `x[, 3]`

d) `x[, -3]`

Exercise 49 Consider the following matrix:

```
x <- matrix(1:6, nrow = 2, ncol = 3)
x

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

What does the following command do? Check your answer.

```
x[, c(3, 1, 2)]
```

5.3 The `apply()` Function

- Sometimes we need to apply a function separately to each row (or each column) of a matrix. To do so, we use:

```
apply()          # Apply a function separately to each row (or each
                 # column) of a matrix
```

- `apply()` has three main arguments, `X`, a matrix, `FUN`, the function to be applied to the rows (or columns) of `X`, and `MARGIN` (`MARGIN = 1` to apply a function to rows and `MARGIN = 2` to apply it to columns).
- For example, below, `apply()` is used to compute the sum of each row:

```
x <- matrix(1:9, nrow = 3, ncol = 3)
x

##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9

apply(X = x, MARGIN = 1, FUN = sum)             # Sum each row of x

## [1] 12 15 18
```

  Note that `apply()` returns a vector, each element of which is the sum of the corresponding row in `x`.

- If the function specified by `FUN` requires additional arguments, they're passed through `apply()` after `FUN`.

Section 5.3 Exercises

Exercise 50 Consider the following matrix `x`:

```
x <- matrix(c(8, 6, 3, 6, 5, 7),
            nrow = 3, ncol = 2)
x

##      [,1] [,2]
## [1,]    8    6
## [2,]    6    5
## [3,]    3    7
```

Guess what the result of each of the following will be, then check your answers.

a) `apply(x, MARGIN = 1, FUN = sum)`

b) `apply(x, MARGIN = 2, FUN = min)`

Exercise 51 R has several built-in datasets, one of which is `USPersonalExpenditure`, a matrix with five rows and five columns consisting of U.S. personal expenditures (in billions of dollars) in the categories: food and tobacco, household operation, medical and health, personal care, and private education for the years 1940, 1945, 1950, 1955 and 1960.

Type:

```
USPersonalExpenditure
```

to view the matrix and

```
? USPersonalExpenditure
```

for more information about the data set.

a) Which of the following commands finds the mean expenditure for each of the five expenditure categories? **Hint**: Should `MARGIN` be `1` or `2`?

```
apply(X = USPersonalExpenditure, MARGIN = 1, FUN = mean)
apply(X = USPersonalExpenditure, MARGIN = 2, FUN = mean)
```

b) Which of the following commands finds the total expenditure for each of the five years? **Hint**: Should `MARGIN` be `1` or `2`?

```
apply(X = USPersonalExpenditure, MARGIN = 1, FUN = sum)
apply(X = USPersonalExpenditure, MARGIN = 2, FUN = sum)
```

6 Lists (B.4)

6.1 Creating and Examining Lists

- Recall that elements of so-called *atomic* vectors must be *homogeneous* (e.g. all `"numeric"`, all `"character"`, or all `"logical"`).
- ***Lists*** are vectors that can be *heterogeneous* (elements of mixed types). They're sometimes called ***recursive*** vectors.

  In fact, their elements can be *any* R *objects*.

  Lists are used to bundle objects together under one name.
- Lists are created and examined using:

```
list()          # Create a list
length()        # Returns the number of elements in a list
is.list()       # Indicates whether or not an object is a list
str()           # Describes the structure of a list
```

- Here's a simple example:

```
y <- list(3, "a", TRUE)
y

## [[1]]
## [1] 3
##
## [[2]]
## [1] "a"
##
## [[3]]
## [1] TRUE

is.list(y)

## [1] TRUE
```

Notice that the indices of list elements are in double square brackets `[[ ]]`.

- Here's a list with three elements, two of which are vectors and the third a matrix:

```
x <- list(x1 = c(1, 2, 3),
          x2 = c("a", "b", "c"),
          x3 = matrix(1:4, nrow = 2, ncol = 2))
x

## $x1
## [1] 1 2 3
##
## $x2
## [1] "a" "b" "c"
##
## $x3
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
```

Notice that when list elements have names (like `x1`, `x2`, and `x3` above), R prints the names preceded by a dollar sign `$`.

- `length()` tells how many elements are contained in a list:

```
length(x)

## [1] 3
```

- `str()` tells us the "structure" of the list:

```
str(x)

## List of 3
##  $ x1: num [1:3] 1 2 3
##  $ x2: chr [1:3] "a" "b" "c"
##  $ x3: int [1:2, 1:2] 1 2 3 4
```

The output above says that `x` is a list with three elements: a `"logical"` vector `x1`, a `"character"` vector `x2`, and a 2×2 `"integer"` matrix `x3`.

Section 6.1 Exercises

Exercise 52 The following commands create a list named `Employees` containing three elements:

- `Name`, a `"character"` vector containing the names of four employees.
- `Salary`, a numeric vector containing their salaries.
- `Union`, a `"logical"` vector indicating whether or not they belong to a union.

```
Employees <- list(Name = c("Joe", "Kim", "Ann", "Bob"),
                  Salary = c(56000, 67000, 60000, 55000),
                  Union = c (TRUE, TRUE, FALSE, FALSE))
```

a) Use `str()` to look at the structure of the list. Report the results.

b) Use `length()` to find the number of elements in the list. Report the result.

6.2 General List Operations

6.2.1 Accessing List Elements

- We access elements of a list using double square brackets `[[ ]]`, the dollar sign `$`, or single square brackets `[ ]`:

```
[[ ]]          # Access a list element via its index or name
$              # Access a list element via its name
[ ]            # Access a list element via its index or name, re-
               # turning a list (rarely used)
```

- We *almost always* use either `[[ ]]` or `$`. The main distinctions are:
  - `[[ ]]` and `$` return the actual *object* from the list, e.g. if the list element being extracted is a *vector*, they return a *vector*.
  - `[ ]` returns a *list* containing the object, e.g. if the list element being extracted is a *vector*, it returns *list* containing that vector.
  - `[[ ]]` can be used with numerical indices or names of list elements (in quotation marks).
  - `$` can only be used with names of list elements (without quotation marks).
  - `[ ]` can extract more than one list element, but `[[ ]]` and `$` extract only a single element.
- Here's an example using `[[ ]]` to extract the second element (`x2`) of the list `x` (from above):

```
x[[2]]                          # We could also use x[["x2"]]

## [1] "a" "b" "c"
```

  Note that a `"character"` vector is returned. We could also have used `x[["x2"]]`.

- Here's an example using `$`:

```
x$x2

## [1] "a" "b" "c"
```

  A `"character"` vector is returned here too.

- Now watch what happens when we use single brackets `[ ]`:

```
x[2]

## $x2
## [1] "a" "b" "c"
```

  This time a *list* with one element (the vector `x2`) is returned (as indicated by the `$x2` label). This is usually *not what we want.*

6.2.2 Adding and Deleting List Elements

- The operators `[[ ]]` and `$` can also be used to add or delete a list element.

  Below, we add the value 16 as the fourth element of `x` and give it the name `x4`:

```
x$x4 <- 16                     # We could also use x[["x4"]] <- 16
x
```

```
## $x1
## [1] 1 2 3
##
## $x2
## [1] "a" "b" "c"
##
## $x3
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
##
## $x4
## [1] 16
```

Above, we could also have used `x[["x4"]] <- 16`.

- To delete a list element, we set its value to `NULL`:

```
x$x4 <- NULL
x
```

```
## $x1
## [1] 1 2 3
##
## $x2
## [1] "a" "b" "c"
##
## $x3
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
```

Section 6.2 Exercises

Exercise 53 Recreate the `Employees` list from Exercise 52.

a) Guess what the result of the following command will be, then check your answer:

```
Employees[[2]]
```

b) Guess what the result of the following command will be, then check your answer:

```
Employees$Salary
```

c) Write a command involving `[[ ]]` that returns the `"logical"` vector `Union` from the `Employees` list.

d) Now write a command involving `$` that returns the `"logical"` vector `Union` from the `Employees` list.

6.3 Named List Elements

- We can get or assign list element names using:

```
names()        # Examine or change the names of list elements
```

- As an example, consider again the list `x`:

```
x <- list(x1 = c(1, 2, 3),
          x2 = c("a", "b", "c"),
          x3 = matrix(1:4, nrow = 2, ncol = 2))
```

```
x

## $x1
## [1] 1 2 3
##
## $x2
## [1] "a" "b" "c"
##
## $x3
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
```

To get the list element names, we type:

```
names(x)

## [1] "x1" "x2" "x3"
```

and to change them to, say, `y1`, `y2`, and `y3`, we type:

```
names(x) <- c("y1", "y2", "y3")
names(x)

## [1] "y1" "y2" "y3"
```

- To remove the names, we'd type:

```
names(x) <- NULL
```

Section 6.3 Exercises

Exercise 54 Here's a simple list `x`:

```
x <- list(x1 = c(1, 2), x2 = c("a", "b"), x3 = 12)
```

What will the following command return? Check your answer.

```
names(x)
```

6.4 Applying a Function to a List Using `lapply()` and `sapply()`

- We can apply a function separately to each element of a *list* using the list versions of the `apply()` function:

```
lapply()      # Apply a function separately to each element of a list,
              # returning a list
sapply()      # Apply a function separately to each element of a list,
              # returning a vector
```

- Both `lapply()` and `sapply()` take two main arguments, `X`, a list, and `FUN`, a function to be applied to each element of the list.

  The difference between the two is that `lapply()` returns a *list* whereas `sapply()` returns a *vector*.

  (The name `lapply()` means ”list apply”, and `sapply()` is short for ”simplified apply”.)

- As an example, consider the list `HtWtAge` containing heights, weights, and ages of five people:

```
HtWtAge <- list(Height = c(65, 68, 70, 60, 61),
                Weight = c(160, 171, 158, 148, 215),
                Age = c(23, 20, 37, 40, 44))
```

```
HtWtAge

## $Height
## [1] 65 68 70 60 61
##
## $Weight
## [1] 160 171 158 148 215
##
## $Age
## [1] 23 20 37 40 44
```

- To compute the average of each variable (`Height`, `Weight`, and `Age`) using `lapply()`, we type:

```
lapply(X = HtWtAge, FUN = mean)

## $Height
## [1] 64.8
##
## $Weight
## [1] 170.4
##
## $Age
## [1] 32.8
```

  Note that `lapply()` returns a *list* (as indicated by the labels `$Height`, `$Weight`, and `$Age`).

- Usually, we'd prefer the results to be in a *vector*. To compute the averages and get the results in a vector we use `sapply()`:

```
sapply(X = HtWtAge, FUN = mean)

## Height Weight    Age
##   64.8  170.4   32.8
```

  Notice that `sapply()` returns a *vector* (with named elements).

- Additional arguments to `FUN` are specified after `FUN` in the call to `lapply()` or `sapply()`.

Section 6.4 Exercises

Exercise 55 Here's the `HtWtAge` list again:

```
HtWtAge <- list(Height = c(65, 68, 70, 60, 61),
                Weight = c(160, 171, 158, 148, 215),
                Age = c(23, 20, 37, 40, 44))
```

a) Write a command using `lapply()`, with `FUN = max`, that returns a *list* containing the maximum value of each variable (`Height`, `Weight`, and `Age`).

b) Now write a command using `sapply()` that does the same thing, but returns a *vector*.

7 Data Frames (B.4)

7.1 Creating and Viewing Data Frames

- ***Data frames*** are two-dimensional, like matrices, but they can be *heterogeneous* (i.e. they can have a mix of *categorical* (`"character"` or `"factor"`) columns and *numerical* (`"numeric"`) ones).
- Each each *row* corresponds to an individual (person, place, company, etc.), and each *column* contains the value of a ***variable*** for that individual.

  An entire *row* is called a ***multivariate observation*** (also sometimes called a ***case***.)
- For example, the following data set (from a study of eight mice) could be stored as a *data frame* in R:

Mice Data Set

Color	Weight	Length
white	23	3.8
grey	21	3.7
black	18	3.0
brown	26	3.4
black	25	3.4
white	22	3.1
black	26	3.5
white	19	3.2

Each row corresponds to a mouse. Note that the data set has a mix of *categorical* (`"character"` or *factor*) and *numerical* (`"numeric"`) columns, and therefore *couldn't* be stored as a matrix.

7.1.1 Creating Data Frames Using `data.frame()`

- We can create a data frame on the R command line using:

```
data.frame()       # Create a data frame from a set of vectors of the
                   # same length
```

(We'll see later how to create a data frame by reading the data from a file.)

- In addition, several functions let us look at various aspects of a data frame:

```
head(); tail()    # Prints the first (or last) six rows of a data frame
nrow(); ncol()    # Indicates the number of rows (or columns)
dim()             # Gives the dimensions (number of rows and columns)
str()             # Gives the structure of a data frame
View()            # Invoke a spreadsheet-style viewer for a data frame
is.data.frame()   # Indicates whether or not an object is a data frame
```

- Consider again the **mice data set** from above. After creating vectors containing the data:

```r
col <- c("white", "grey", "black", "brown", "black", "white", "black", "white")
wt <- c(23, 21, 12, 26, 25, 22, 26, 19)
len <- c(3.8, 3.7, 3.0, 3.4, 3.4, 3.1, 3.5, 3.2)
```

we create a data data frame named `mice.data` by typing:

```
mice.data <- data.frame(Color = col, Weight = wt, Length = len,
                        stringsAsFactors = FALSE)
```

We can check that the data frame was created properly by typing its name:

```
mice.data

##   Color Weight Length
## 1 white     23    3.8
## 2  grey     21    3.7
## 3 black     12    3.0
## 4 brown     26    3.4
## 5 black     25    3.4
## 6 white     22    3.1
## 7 black     26    3.5
## 8 white     19    3.2
```

and we can make sure its a data frame by typing:

```
is.data.frame(mice.data)

## [1] TRUE
```

Specifying `stringsAsFactors = FALSE` in `data.frame()` indicates that we *don't* want the `"character"` vector `col` to be converted to a so-called *factor* when the data frame is created. (The default for the argument `stringsAsFactors` is `TRUE`.) We'll discuss *factors* later.

- The functions `head()` and `tail()` print just the first and last six rows, respectively:

```
head(mice.data)

##   Color Weight Length
## 1 white     23    3.8
## 2  grey     21    3.7
## 3 black     12    3.0
## 4 brown     26    3.4
## 5 black     25    3.4
## 6 white     22    3.1
```

They're useful with large data sets (when we just want to look at a few rows of data).

- To find out how many rows a data frame has, use `nrow()`:

```
nrow(mice.data)

## [1] 8
```

To find out how many columns it has, use `ncol()`. We could also use `dim()`.

- To look at the "structure" of `mice.data`, type:

```
str(mice.data)
```

(Output not shown.)

7.1.2 Reading Data From .csv and .txt Files Using `read.csv()` and `read.table()`

- We can create a data frame by reading the data from a file using:

```
read.csv()          # Read data from a comma separated value (.csv)
                    # or text (.txt) file into a data frame
read.table()        # Read data from a text (.txt) file into a data
                    # frame
```

- Suppose we have a text (.txt) file, **C:\Users\Grevstad\Documents\mice.txt**, that contains the **mice data set**. The contents of the text file would look like this:

Color	Weight	Length
white	23	3.8
grey	21	3.7
black	18	3.0
brown	26	3.4
black	25	3.4
white	22	3.1
black	26	3.5
white	19	3.2

- We read it into a data frame named `mice` by typing:

```
# We could also use read.table():
mice.data <- read.csv("C:/Users/Grevstad/Documents/mice.txt",
                      sep = "",
                      header = TRUE,
                      stringsAsFactors = FALSE)
```

It's always a good idea to check that the data were read in correctly:

```
mice.data

##   Color Weight Length
## 1 white     23    3.8
## 2  grey     21    3.7
## 3 black     18    3.0
## 4 brown     26    3.4
## 5 black     25    3.4
## 6 white     22    3.1
## 7 black     26    3.5
## 8 white     19    3.2
```

Specifying `sep = ""` indicates that any 'white space' (i.e. spaces or tabs) counts as a column the separator.

Specifying `header = TRUE` indicates that the first row of the file containing the data consists of column names.

As before, `stringsAsFactors = FALSE` indicates that we *don't* want the `"character"` vector `Color` to be automatically converted to a so-called *factor*.

Above, we could also have used `read.table()` instead of `read.csv()`.

- `read.csv()` and `read.table()` are the *same function*, but have different default settings for some of the arguments.

- Here are some comments about using `read.csv()` and `read.table()`:
  - The usual back slashes (\) are written as forward slashes (/) in `read.csv()` and `read.table()`. They could also be written as double back slashes (\\).
  - Specifying `header = TRUE` is used to indicate that the first row of the text file contains the variable names.

- You can use either single (') or double (") quotations when specifying the location and name of the file.
- By default, R recognizes one or more white spaces, tabs, or newline characters as separators of data values in the input file. Other separators can be specified via the `sep` argument to `read.csv()` and `read.table()`.
- As before, specifying `stringsAsFactors = FALSE` indicates that we *don't* want the character column (Color) to be converted to a *factor*.
- To read data from an **Excel** file, you can save the Excel file as a 'comma separated value' (.csv) file, and then read it into R using `read.csv()` or `read.table()`.
  (There are also more specialized functions for reading in Excel files, but we won't cover them.)

Section 7.1 Exercises

Exercise 56

a) Here's the **mice data set** as three vectors:

```
col <- c("white", "grey", "black", "brown", "black", "white", "black",
         "white")
wt <- c(23, 21, 12, 26, 25, 22, 26, 19)
len <- c(3.8, 3.7, 3.0, 3.4, 3.4, 3.1, 3.5, 3.2)
```

After creating the vectors, write a command involving `data.frame()` that creates a data frame containing the data. Make sure the column names are `Color`, `Weight`, and `Length`.

Check that you created the data frame correctly by typing its name on the command line, for example:

```
mice.data
```

b) Before proceeding, remove the data frame you just created from your Workspace, for example by typing:

```
rm(mice.data)
```

and double check that it's no longer there by typing

```
ls()                    # Or you could use objects().
```

The file **mice.txt** contains the **mice data set**. After saving the file onto your computer, type the following:

```
my.file <- file.choose()
```

then in the dialog box choose the **mice.txt** file that you saved. Now type:

```
my.file
```

and report the results.

c) Now try the following:

```
mice.data <- read.csv(my.file, sep = "", header = TRUE)
```

Make sure to check that the data were read in correctly, for example by typing:

```
mice.data
```

d) Now type the following commands and report the results:

```
nrow(mice.data)      # Number of rows.
ncol(mice.data)      # Number of columns.
head(mice.data)      # First six rows.
names(mice.data)     # Column names.
str(mice.data)       # "Structure".
```

7.2 Accessing and Replacing Elements, Rows, or Columns of a Data Frame

- Data frames have features of both *matrices and lists.* To extract a specific value, row, or column, we use:

```
[ , ]      # Access data frame elements, rows, or columns via their
           # row and column indices (separated by a comma)
$          # Access a data frame variable (column) by specifying its
           # name
[[ ]]      # Access a data frame variable (column) by specifying its
           # index or name
```

7.2.1 Accessing Rows and Columns Using `[ ]`

- As with matrices, we can access a specific element, row, or column of a data frame using single square brackets `[ ]`. For example, the value in the 3rd row and 2nd column of `mice` is:

```
mice.data[3, 2]          # The value in the 3rd row, 2nd column

## [1] 18
```

The entire 3rd row is accessed via:

```
mice.data[3, ]           # The entire 3rd row

##   Color Weight Length
## 3 black     18      3
```

and the entire 2nd column would be accessed via `mice.data[, 2]`.

- We can also use square brackets and the assignment operator to *replace* a value in a data frame, for example:

```
mice.data[3, 2] <- 12
```

7.2.2 Accessing and Replacing Columns Using `$` and `[[ ]]`

- The *list* operator `$` can also be used to access columns of a data frame:

```
mice.data$Color      # Access the Color variable by name

## [1] "white" "grey"  "black" "brown" "black" "white" "black" "white"
```

- The list operator `[[ ]]` can be used to access columns of a data frame too:

```
mice.data[[1]]       # Access a variable by list index (i.e. column number).
                     # We could also use mice[["Color"]].
```

```
## [1] "white" "grey"  "black" "brown" "black" "white" "black" "white"
```

Above, we could also access the `Color` column using `mice.data[["Color"]]`.

- In fact, **data frames *are* lists**:

```
is.data.frame(mice.data)

## [1] TRUE

is.list(mice.data)

## [1] TRUE
```

The **list elements** are the **columns** of the data frame.

7.2.3 Removing and Adding Columns Using $ or [[ ]]

- We can *remove* a column from a data frame by setting it to `NULL`:

```
mice.data$Color <- NULL
```

- We can *add* a column (e.g. named `BodyFat` containing values from the vector `fat`) to a data frame as follows:

```
fat <- c(2.5, 2.1, 3.1, 3.0, 2.7, 2.6, 1.8, 2.0)
```

```
mice.data$Bodyfat <- fat
```

```
mice.data

##   Weight Length Bodyfat
## 1     23    3.8     2.5
## 2     21    3.7     2.1
## 3     12    3.0     3.1
## 4     26    3.4     3.0
## 5     25    3.4     2.7
## 6     22    3.1     2.6
## 7     26    3.5     1.8
## 8     19    3.2     2.0
```

- Double square brackets `[[ ]]` can be used in a similar manner to remove and add columns.

  The `cbind()` function can be used to add a column too (but `cbind()` will convert a `"character"` vector to a so-called *factor* unless you specify `stringsAsFactors = FALSE` in `cbind()`).

Section 7.2 Exercises

Exercise 57 Consider the following data on nine people:

Status	Age	Education
Married	36	HS Diploma
Single	33	Bachelor of Arts
Single	21	Bachelor of Science
Married	29	Bachelor of Science
Single	19	HS Diploma
Married	35	Bachelor of Arts
Married	39	Master of Science
Single	28	HS Diploma
Single	21	HS Diploma

The following commands will create a data frame containing the data:

```r
status <- c("Married", "Single", "Single", "Married", "Single",
            "Married", "Married", "Single", "Single")
age <- c(36, 33, 21, 29, 19, 35, 39, 28, 21)
educ <- c("HS Diploma", "Bachelor of Arts", "Bachelor of Science",
          "Bachelor of Science", "HS Diploma", "Bachelor of Arts",
          "Master of Science", "HS Diploma", "HS Diploma")
```

```r
my.data <- data.frame(Status = status, Age = age, Education = educ)
```

a) Guess what each of the following commands will return, then check your answers:

```r
my.data[6, 2]
```

```r
my.data[6, ]
```

```r
my.data[, 2]
```

```r
my.data$Age
```

```r
my.data[[2]]
```

b) Write three different commands that return the entire 3rd column (`Education`) of the data frame:

- Using single square brackets `[ ]`.
- Using the dollar sign operator `$`.
- Using double square brackets `[[ ]]`.

c) The nine people have each aged one year since the data were collected. Here's a vector containing their current ages:

```r
age2 <- c(37, 34, 22, 30, 20, 36, 40, 29, 22)
```

Describe in words what the following commands do.

```r
my.data$Age <- NULL
my.data$Age2 <- age2
```

d) Here's another vector:

```r
ageofspouse <- c(39, NA, NA, 34, NA, 27, 30, NA, NA)
```

Describe in words what the following commands both do.

```r
my.data$AgeOfSpouse <- ageofspouse
my.data[["AgeOfSpouse"]] <- ageofspouse
```

7.3 Viewing and Changing Variable Names in a Data Frame

- We can get or change the names of the columns of a data frame using:

```
names()           # Get or assign the names of the variables in a
                  # data frame
```

- For example (using the `mice.data` data frame from above):

```
names(mice.data)

## [1] "Weight"  "Length"  "Bodyfat"

names(mice.data) <- c("Col", "Wt", "Len")
```

```
names(mice.data)

## [1] "Col" "Wt"  "Len"
```

- To change just one of the column names, such as the 3rd one, we could use:

```
names(mice.data)[3] <- "Len"
```

Section 7.3 Exercises

Exercise 58 Create the following data frame:

```
x <- data.frame(A = 1:5, B = 6:10, C = c("a", "b", "c", "d", "e"))
```

a) Guess what each of the following commands will return, then check your answers:

```
names(x)
```

```
is.vector(names(x))
```

```
typeof(names(x))
```

b) Guess what the following will do, then check your answer:

```
names(x) <- c("AA", "BB", "CC")
```

7.4 Rearranging the Rows or Columns of a Data Frame

7.4.1 Rearranging the Rows or Columns

- We rearrange (permute) the rows (or columns) of a data frame just as we did for a matrix, using a vector of indices with the desired permutation before (or after) a comma in square brackets `[ ]`.

  For example here are the **mice data** again:

```
col <- c("white", "grey", "black", "brown", "black", "white", "black", "white")
wt <- c(23, 21, 12, 26, 25, 22, 26, 19)
len <- c(3.8, 3.7, 3.0, 3.4, 3.4, 3.1, 3.5, 3.2)

mice.data <- data.frame(Color = col, Weight = wt, Length = len,
                        stringsAsFactors = FALSE)
```

```
mice.data

##   Color Weight Length
## 1 white     23    3.8
## 2  grey     21    3.7
## 3 black     12    3.0
## 4 brown     26    3.4
## 5 black     25    3.4
## 6 white     22    3.1
## 7 black     26    3.5
## 8 white     19    3.2
```

This next command returns `Weight` (2nd column) in the 1st position, `Length` (3rd column) in the 2nd position, and `Color` (1st column) in the 3rd position:

```
mice.data[ , c(2, 3, 1)]                # Rearranges the columns

##   Weight Length Color
## 1     23    3.8 white
## 2     21    3.7  grey
## 3     12    3.0 black
## 4     26    3.4 brown
## 5     25    3.4 black
## 6     22    3.1 white
## 7     26    3.5 black
## 8     19    3.2 white
```

Note above that the permutation was specified *after* the comma in `[ ]`.

This next command returns the rows in the order 8th, 7th, 6th, ..., 1st:

```
mice.data[8:1, ]                    # Rearranges the rows

##   Color Weight Length
## 8 white     19    3.2
## 7 black     26    3.5
## 6 white     22    3.1
## 5 black     25    3.4
## 4 brown     26    3.4
## 3 black     12    3.0
## 2  grey     21    3.7
## 1 white     23    3.8
```

Note above that the permutation was specified *before* the comma in `[ ]`.

7.5 Filtering on Data Frames

- ***Filtering*** means extracting from a data frame just the rows that satisfy some condition.

  We can filter rows from a data frame using either square brackets `[ ]` or `subset()`.

7.5.1 Filtering Rows Using Square Brackets `[ ]`

- To filter using square brackets `[ ]`, we specify a `"logical"` vector before the comma. For example here's `mice.data` again:

```
mice.data

##   Color Weight Length
## 1 white     23    3.8
## 2  grey     21    3.7
## 3 black     12    3.0
## 4 brown     26    3.4
## 5 black     25    3.4
## 6 white     22    3.1
## 7 black     26    3.5
## 8 white     19    3.2
```

We can use the `"logical"` vector:

```
mice.data$Color == "black"

## [1] FALSE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE
```

to extract the rows corresponding to **black mice** by typing:

```
mice.data[mice.data$Color == "black", ]    # Extracts rows of black mice

##   Color Weight Length
## 3 black     12    3.0
## 5 black     25    3.4
## 7 black     26    3.5
```

7.5.2 Filtering Rows Using `subset()`

- `subset()` takes two main arguments, a data frame, `x`, and a condition to be met, `subset`, and then extracts from `x` the rows for which the condition is met.

```
subset()          # Extract a subset of vector elements that satisfy a
                  # given condition
```

- For example, to extract the rows of `mice.data` the rows corresponding to *black mice*, type:

```
subset(mice.data, subset = Color == "black")

##   Color Weight Length
## 3 black     12    3.0
## 5 black     25    3.4
## 7 black     26    3.5
```

Notice that there's no need to use the dollar sign operator `$` with `Color` when `subset()` is used.

Section 7.5 Exercises

Exercise 59 The built-in R data frame `warpbreaks` contains data from a study of the strength of yarn used in weaving. There are three variables in the data set:

`breaks`	The number of breaks per loom, where a loom corresponds to a fixed length of yarn.

`wool`	The type of wool (A or B)
`tension`	The level of tension (L, M, H)

To look at the data, type:

```
warpbreaks
```

If you want to read about it, look at its help page by typing:

```
? warpbreaks
```

a) Write a command involving square brackets `[ ]` that returns just the rows of `warpbreaks` corresponding to observations made at the `"M"` level of `tension`.

b) Now write a command that uses `subset()` to do the same thing as in part *a*.

8 Factors (B.4)

8.1 Creating and Viewing Factors and Their Levels

- ***Factors***, like `"character"` vectors, are used to store categorical data. They differ from `"character"` vectors in the way R stores them internally, and they contain a bit of extra information called *levels*.
- To create and look at factors, the following are useful:

```
factor()          # Create a factor from a character vector
length()          # Returns the number of elements in a factor
levels()          # Examine the levels of the factor
nlevels()         # Returns the number of levels of the factor
is.factor()       # Indicates whether or not an object is a factor
str()             # Describes the structure of a factor
```

- `factor()` takes a `"character"` vector and converts it to a factor:

```
char.vec <- c("ctrl", "trt1", "trt2", "ctrl", "trt1", "trt2", "ctrl", "trt1", "trt2")
```

```
my.fac <- factor(char.vec)
```

```
is.factor(my.fac)

## [1] TRUE
```

- When R prints out a factor, it also indicates its ***levels***, or unique values:

```
my.fac

## [1] ctrl trt1 trt2 ctrl trt1 trt2 ctrl trt1 trt2
## Levels: ctrl trt1 trt2
```

- We'll see shortly that it's generally preferable to convert factors to `"character"` vectors. To convert a factor to a `"character"` vector, we use:

```
as.character()      # Convert a factor to a character vector
```

- For example:

```
char.vec <- as.character(my.fac)
```

```
is.vector(char.vec)

## [1] TRUE
```

- To see how many elements a factor has, use `length()`. To look at the *levels* of a factor, use `levels()`. To simply see *how many* levels there are, use `nlevels()`.

```
length(my.fac)

## [1] 9

levels(my.fac)

## [1] "ctrl" "trt1" "trt2"

nlevels(my.fac)

## [1] 3
```

Section 8.1 Exercises

Exercise 60

a) Write a command that uses `factor()` to convert the following `"character"` vector

```
x.char <- c("a", "a", "b", "b", "c", "c", "d", "d")
```

to a factor named `x.factor`:

```
x.factor

## [1] a a b b c c d d
## Levels: a b c d
```

b) After creating the factor `x.factor`, guess what the result of the following command will be, then check your answer:

```
length(x.factor)
```

c) Guess what the result of the following command will be, then check your answer:

```
levels(x.factor)
```

d) Guess what the result of the following command will be, then check your answer:

```
nlevels(x.factor)
```

Exercise 61 Guess what the result of the following commands will be, then check your answer:

```
x.fac <- factor(c("a", "a", "a", "b", "b", "c"))
levels(x.fac)
```

8.2 The Difference Between Factors and `"character"` Vectors

- Unlike `"character"` vectors, R stores factors internally as *numeric vectors*, with integer values representing the levels of the factor.

  To see, recall that `typeof()` indicates the *type* of an object, and for the factor `my.fac` and `"character"` vector `char.vec` from above, returns:

  ```
  typeof(my.fac)

  ## [1] "integer"

  typeof(char.vec)

  ## [1] "character"
  ```

  The first command above indicates that `my.fac` is stored as `"integer"` values.

- The integer values used internally to represent the levels of a factor can be viewed using:

  ```
  unclass()          # Remove the class attribute of an object.  Can be
                     # used to view the integer values used internally
                     # to represent the levels of a factor.
  ```

- For example, here's `my.fac` again:

  ```
  my.fac

  ## [1] ctrl trt1 trt2 ctrl trt1 trt2 ctrl trt1 trt2
  ## Levels: ctrl trt1 trt2
  ```

  The integers used to represent the levels of `my.fac` can be viewed by typing:

  ```
  unclass(my.fac)

  ## [1] 1 2 3 1 2 3 1 2 3
  ## attr(,"levels")
  ## [1] "ctrl" "trt1" "trt2"
  ```

  The numbering is as follows. An observation made at the jth level of the factor (i.e. the level in the jth position as returned by `levels()`) is represented by the integer value j. Above, because the levels of `my.fac` (as returned by `levels()`) are `"ctrl"`, `"trt1"`, and `"trt2"`, observations at the `"ctrl"` level are represented by the value 1, those at the `"trt1"` level by the value 2, and those at the `"trt2"` level by the value 3.

- Factors can be annoying because R can convert a factor to a numeric vector without telling you. For example, here's a factor:

  ```
  y.fac <- factor(c("a", "a", "b", "b", "c", "c"))
  ```

When we try to combine `y.fac` with a numerical value, R automatically converts `y.fac` its integer representation:

```
c(y.fac, 6)

## [1] 1 1 2 2 3 3 6
```

For this reason, it's generally preferable to convert factors to `"character"` vectors using `as.character()`.

9 Data Visualization With Base R (Not in the book)

9.1 Creating Plots

- R is equipped with built-in graphics capabilities. Here are some commonly used functions for creating plots:

```
plot()              # Scatterplot, time-series plot
hist()              # Histogram
boxplot()           # Boxplots
curve()             # Graph of a function
barplot()           # Bar chart of specified bar heights
pie()               # Pie chart of specified pie areas
stripchart()        # Dot plot, individual value plot
```

9.1.1 Scatterplots

- `plot()` takes arguments `x` and `y` and plots them in a ***scatterplot***. Some other arguments that can be passed to `plot()` are below:

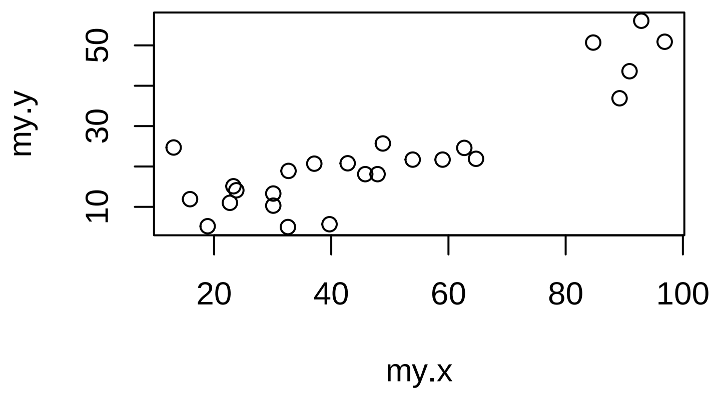
```
x, y           # Vectors containing the coordinates of points in the plot
main           # Main title (in quotation marks)
sub            # Subtitle (in quotation marks)
xlab, ylab     # Labels for the x and y axes (in quotation marks)
xlim, ylim     # Limits for the x and y axes in the plot (in the form
               # c(lower, upper) )
type           # Type of plot that should be drawn (e.g. points, lines,
               # etc.)
...            # Other arguments such as the graphical parameters that
               # can be controlled by par()
```

- Here's an example using these vectors `my.x` and `my.y`:

```
my.x <- c(18.9, 53.9, 42.8, 47.9, 37.1, 23.3, 90.9, 30.1, 96.9, 22.7, 15.9, 45.8, 59.0,
          89.2, 30.1, 92.9, 32.6, 84.7, 64.7, 48.8, 23.8, 62.7, 13.1, 39.7, 32.7)
```

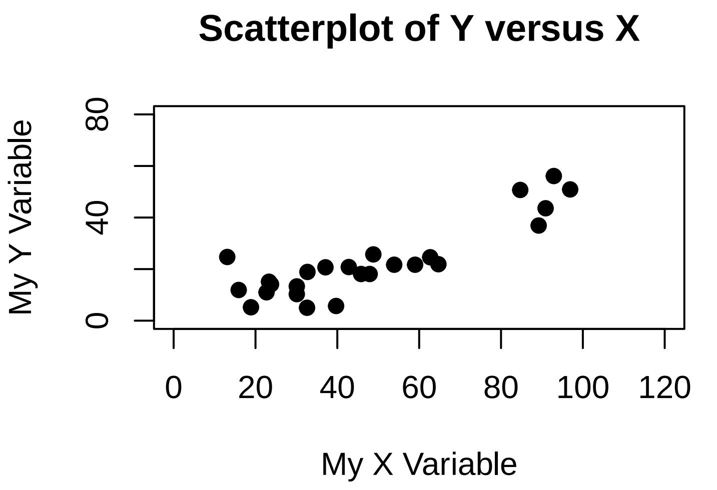
```
my.y <- c(5.2, 21.7, 20.8, 18.1, 20.7, 15.1, 43.6, 10.3, 50.9, 11.0, 11.9, 18.1, 21.7,
          36.9, 13.3, 56.1, 5.0, 50.7, 21.9, 25.7, 14.1, 24.6, 24.7, 5.7, 18.9)
```

```
plot(x = my.x, y = my.y)
```



- Here's a nicer version of the plot:

```
plot(x = my.x, y = my.y,
     main = "Scatterplot of Y versus X",
     xlab = "My X Variable",
     ylab = "My Y Variable",
     xlim = c(0, 120),
     ylim = c(0, 80),
     pch = 19)
```

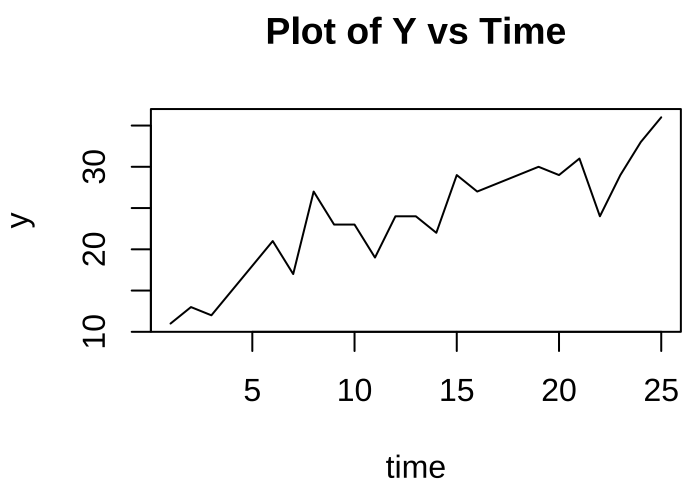


The argument `pch`, for "plot character", is one of the '`...`' arguments that can be passed to `plot()` and that can also be set by the `par()` function. Specifying `pch = 19` indicates solid circles for the points. To see a list of the available point types, look for `pch` in the help page for `par()` (type `?par`).

- In a ***time-series plot***, x represents time and the points are connected by lines. To make one, we specify `type = "l"` (the letter `"l"` for "line") in `plot()`. For example:

```
y <- c(11, 13, 12, 15, 18, 21, 17, 27, 23, 23, 19, 24, 24, 22, 29, 27, 28, 29, 30, 29,
       31, 24, 29, 33, 36)

time <- 1:25
```

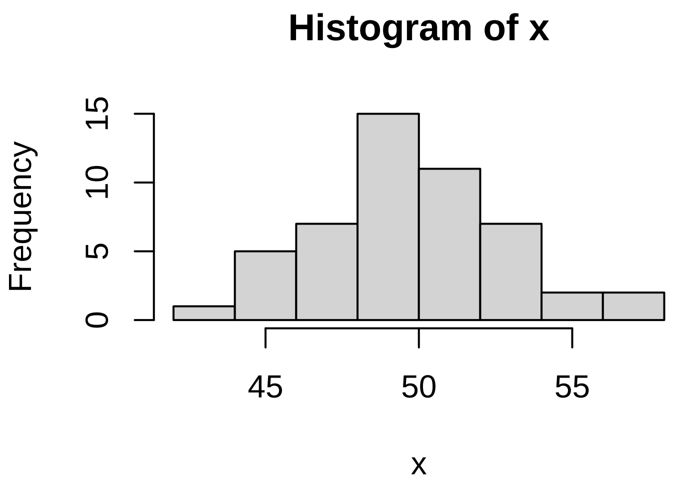
```r
plot(x = time, y = y,
     type = "l",
     main = "Plot of Y vs Time")
```



9.1.2 Histograms and Boxplots

- `hist()` takes a vector argument `x` and produces a histogram of the data. For example:

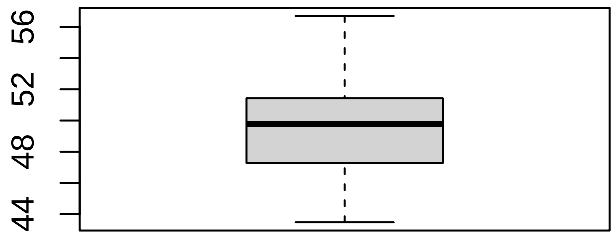
```r
# Generate 50 random observations from a normal distribution:
x <- rnorm(n = 50, mean = 50, sd = 4)

hist(x)
```



The `rnorm()` function is used above to generate a random sample from a *normal* distribution with mean $\mu = 50$ and standard deviation $\sigma = 4$.

- `boxplot()` takes one or more vector arguments and produces the boxplot(s). For example:

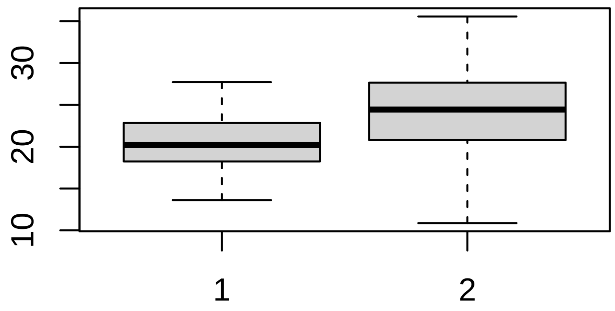
```r
boxplot(x)
```



- To produce side-by-side boxplots, we pass multiple vectors to `boxplot()`. For example:

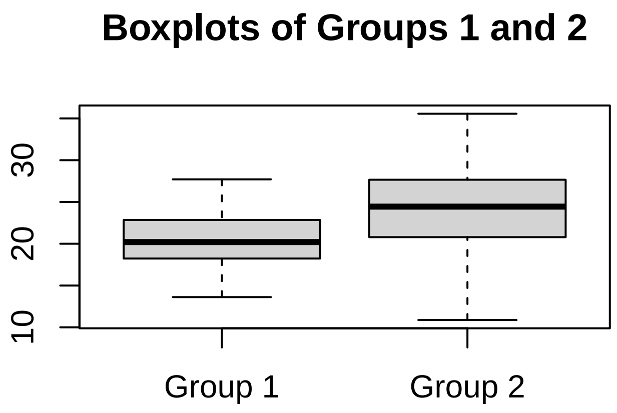
```
x1 <- rnorm(n = 50, mean = 20, sd = 3)
x2 <- rnorm(n = 40, mean = 25, sd = 5)
```

```
boxplot(x1, x2)
```



- We can include labels below the boxes via the `names` argument (and add a title via `main`):

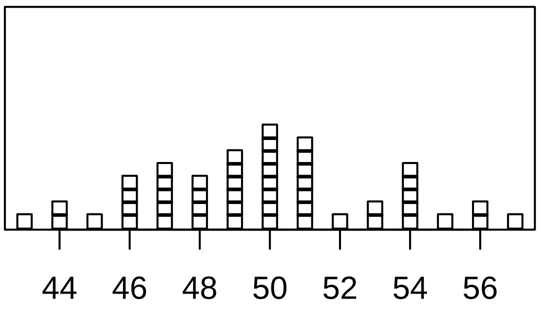
```
boxplot(x1, x2,
        names = c("Group 1", "Group 2"),
        main = "Boxplots of Groups 1 and 2")
```



9.1.3 Dot Plots

- The function `stripchart()` will produce a dot plot of a data set if we specify `method = "stack"`. It's sometimes necessary to round the data values to get them to stack on top of each other. For example (using `x` from above):

```
x <- round(x)
stripchart(x, method = "stack", at = 0)
```

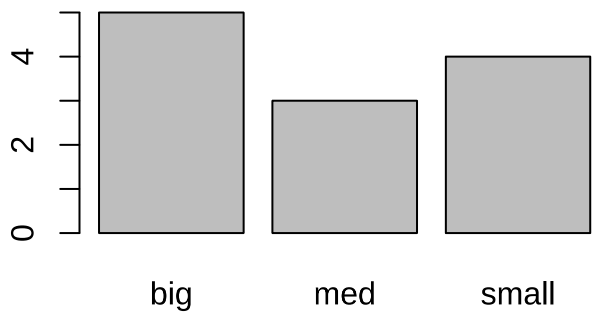


Specifying `at = 0` indicates that we want base of the stacks of dots to be "at" the horizontal axis ($y = 0$).

9.1.4 Bar Plots and Pie Charts

- *Categorical* data are may be displayed in bar plots or pie charts.
- `barplot()` takes a vector argument `height` containing bar heights and produces the bar plot. For example:

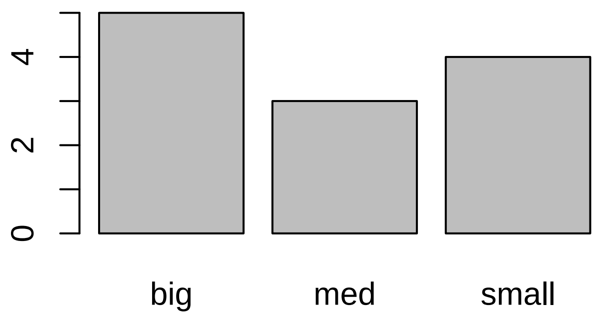
```
bar.hts <- c(5, 3, 4)

barplot(height = bar.hts, names.arg = c("big", "med", "small"))
```



The `names.arg` argument was used to the add labels below the bars.

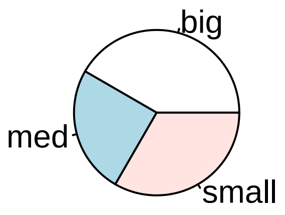
- The bar heights can be obtained from a `"character"` vector (or *factor*) using `table()`, and a *table* object can be passed directly to `barplot()`:

```
char.vec <- c("big", "big", "med", "small", "med", "big", "big", "small", "small",
              "med", "big", "small")
my.tab <- table(char.vec)
my.tab

## char.vec
##   big   med small
##     5     3     4

barplot(my.tab)
```



- `pie()` takes a vector argument `x` indicating the *relative* sizes of the pie slices, and produces a pie chart. For example:

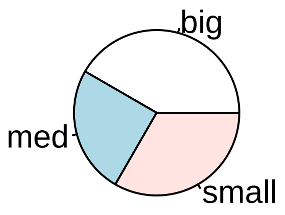
```
slice.sizes <- c(5, 3, 4)

pie(x = slice.sizes, labels = c("big", "med", "small"))
```



The `labels` argument was used to the add labels to the slices.

- The pie areas can be obtained from a `"character"` vector or factor using `table()`, and a *table* object can be passed directly to `pie()`. For example (using `char.vec` from above):

```
my.tab <- table(char.vec)
pie(my.tab)
```



Section 9.1 Exercises

Exercise 62 The data set `state.x77` comes built-in with R. You can type

```
? state.x77
```

for more information about the data. The vectors `illit` and `murder`, created from columns of `state.x77` below, contain illiteracy and murder rates for each of the 50 states in the U.S.:

```
illit <- state.x77[ , 3]
murder <- state.x77[ , 5]
```

Use `plot()` to make a scatterplot of the murder rates (y-axis) versus illiteracy rates (x-axis). Use the arguments `main`, `xlab`, `ylab`, `xlim`, and `ylim` to modify the main title, x-axis and y-axis labels, and x-axis and y-axis limits. Report your R command.

Exercise 63 The function `table()` takes a `"character"` vector (or *factor*) argument and returns a *table* of counts.

Both `barplot()` and `pie()` accept *tables* objects as arguments, and they produce plots from the table counts.

A recent Gallup poll asked people if they smiled or laughed "a lot" in a given day. Here's a representative sampling of the responses:

```
laughed <- c("Yes", "Yes", "Yes", "No", "Yes", "No", "Yes", "Yes", "Yes", "Yes", "No",
             "Yes", "Yes", "Yes", "No", "No", "Yes", "Yes", "Yes", "No", "Yes")
```

a) In words, what do the following commands do:

```
my.tab <- table(laughed)
my.tab
```

b) Pass the *table* object `my.tab` to `barplot()` to make a bar plot of the counts. Report your R command.

c) Pass the *table* object `my.tab` to `pie()` to make a pie chart of the counts. Report your R command.

9.2 Adding to an Existing Plot

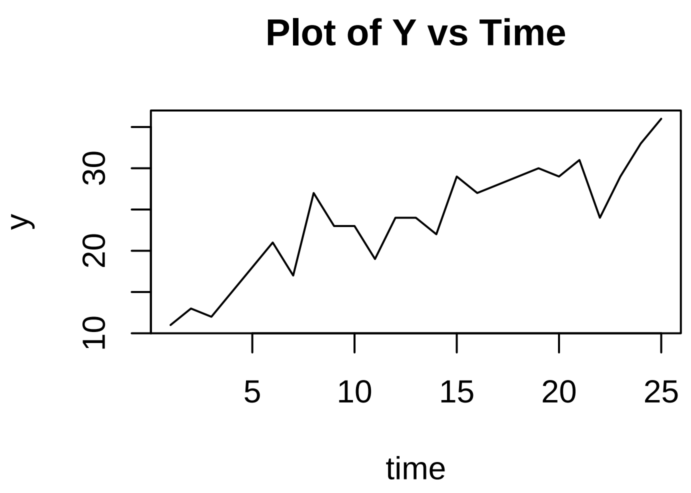
- Here are some functions for adding features to an existing plot:

```
points()      # Add points to the plot at specified coordinates
lines()       # Add a line to the plot connecting specified coordinates
polygon()     # Draw a polygon in the plot with a given set of vertices
abline()      # Add a line to the plot with given intercept and
              # slope
segments()    # Add line segments to the plot between pairs of points
arrows()      # Draw an arrow in the plot (with specified start and
              # end points)
text()        # Add text to the plot at a specified set of coordinates
legend()      # Add a legend to the plot
title()       # Add a main title to the plot (if it doesn´t already
              # have one).  Can also be used to add x and y axis labels.
curve()       # Add a curve to the plot (specify add = TRUE)
```

- In addition to their main arguments (see their help pages), these functions also accept arguments '`...`' representing graphical parameters that can be set by `par()` (e.g. `col`, `pch`, `cex`, `lwd`, etc. – see the help page for `par()`).

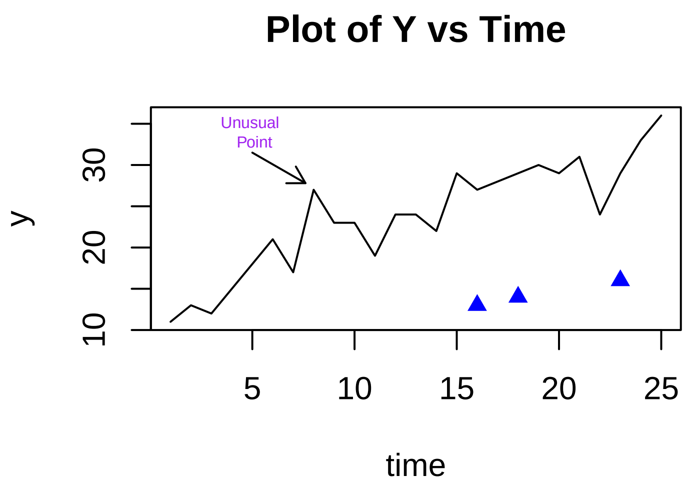
- As an example, consider the time-series plot (using `time` and `y` created earlier):

```
plot(x = time, y = y,
     type = "l",
     main = "Plot of Y vs Time")
```



Below, we use `text()`, `arrows()`, and `points()` to add to the plot:

```r
text(x = 5, y = 34,
     labels = "Unusual \n Point",
     cex = 0.5,
     col = "purple")
arrows(x0 = 5, y0 = 31.5, x1 = 7.6, y1 = 27.8,
       length = 0.1)

new.x <- c(16, 18, 23)
new.y <- c(13, 14, 16)
points(x = new.x, y = new.y,
       pch = 17,
       col = "blue")
```



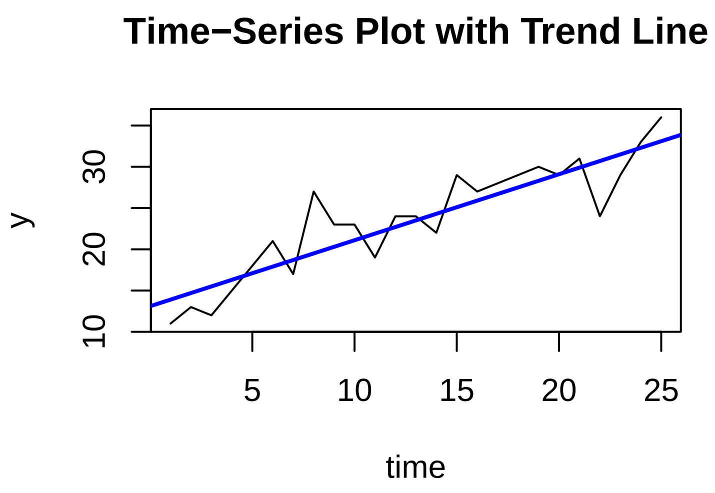
The symbol `\n` in the label passed to `text()` above is a ”new line” indicator.

- As another example, consider the time-series plot:

```r
plot(x = time, y = y,
     type = "l",
     main = "Time-Series Plot with Trend Line")
```

Below, we use `abline()` to add a trend line (with intercept $a = 13.1$ and slope $b = 0.8$):

```
abline(a = 13.1, b = 0.8,
       lwd = 2,
       col = "blue")
```



10 Objects and Classes (B.4)

- In R:
  - "Everything that **exists** is an **object**."
  - "Everything that **happens** is a **function call**."

  The quotes are from John Chambers.

- Each object belongs to a specific ***class*** of objects. The most important classes are:

`"numeric"` vectors	`"logical"` vectors
`"character"` vectors	`"matrix"`
`"array"`	`"list"`
`"data.frame"`	`"factor"`
`"function"`	

  We'll use quotes, as above, when referring to a *class* of objects.

- We can get (or set) the class of an object using:

```
class()        # Returns the class of an object.  Can also be used
               # to assign a class attribute to an object.
is.*()         # Returns TRUE if an object belongs to the class *
               # (e.g. is.numeric(), is.data.frame(), etc.) and FALSE
               # otherwise.
```

- For example:

```
x <- c(2, 4, 7, 9)
class(x)

## [1] "numeric"

is.numeric(x)

## [1] TRUE
```

- Another example:

```r
a <- matrix(1:6, nrow = 2, ncol = 3)
class(a)

## [1] "matrix" "array"

is.matrix(a)

## [1] TRUE
```

- One more example:

```r
y <- data.frame(y1 <- c(1, 4), y2 <- c("u", "v"))
class(y)

## [1] "data.frame"

is.data.frame(y)

## [1] TRUE
```

- It's important to distinguish between the *class* of an object (returned by `class()`) and the *type* of the object (returned by `typeof()`).
  - The *type* refers to how R *stores* the object internally.
  - The *class* is a property (or *attribute*) of the object that affects what functions do with the object when it's passed to them as an argument.

  Sometimes the type and class are the same, but it's not always the case.

Section 10.0 Exercises

Exercise 64 In R, everything that "exists" is an *object*, and each object belongs to a *class* of objects. Guess the class of each of the objects below, then check your answers.

a)
```r
u <- c("a", "b", "c")
class(u)
```

b)
```r
x <- c(3, 6, 1)
class(x)
```

c)
```r
y <- list(3, 5, 2)
class(y)
```

Exercise 65 I In R, everything that "exists" is an *object*, and each object belongs to a *class* of objects. Even *functions* are objects.

a) Guess the class of `sqrt()`, then check your answer.

```r
class(sqrt)
```

b) Guess what will be returned by the following command, then check your answer.

```r
is.function(sqrt)
```

c) Because functions are objects, they can be passed as arguments to other functions. The *user-defined* function `apply.fun()` below takes another function `FUN` and a value `x` as arguments and applies `FUN` to `x`:

```
apply.fun <- function(x, FUN) {
    do.call(FUN, args = list(x))
  }
```

Above, `do.call()` executes a function call by passing the value `x` to the function `FUN`. It wouldn't work to just type `FUN(x)`.

After creating `apply.fun()`, guess what the result of each of the following commands will be, then check your answers:

```
apply.fun(x = 4, FUN = sqrt)
```

```
apply.fun(x = -3, FUN = abs)
```

Exercise 66 The *type* of an object refers to how R *stores* the object internally. The *class* is a property (or *attribute*) of the object that affects what functions do with the object when it's passed to them as an argument.

Sometimes the type and class of an object are the same, but not always. Here are two objects:

```
x <- c("a", "b", "c")
y <- factor(c("a", "b", "c"))
```

a) Guess what the result of each of the following commands will be, then check your answers:

```
class(x)

typeof(x)
```

b) Guess what the result of each of the following commands will be, then check your answers:

```
class(y)

typeof(y)
```

11 Generic Functions and Methods (B.4)

11.1 Generic Functions

- For some R functions, the object passed to it as an argument has to belong to a *single*, *specific* class.

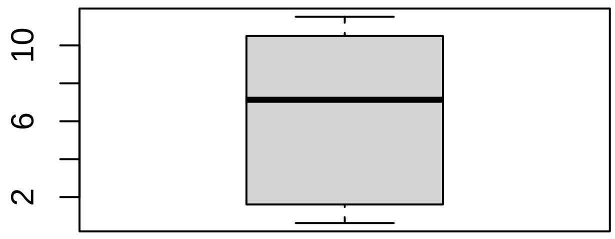
  For other functions, called ***generic*** functions, the object passed to it can belong to any of *several* classes, and the function does a different thing depending on the class of that object.

- For example, `boxplot()` is *generic* because we can pass it a `"numeric"` vector *or* a `"matrix"` (among other classes of objects), and it does something different depending on which of these we pass to it.

  Below, `boxplot()` is first passed a `"numeric"` vector and then a `"matrix"`:

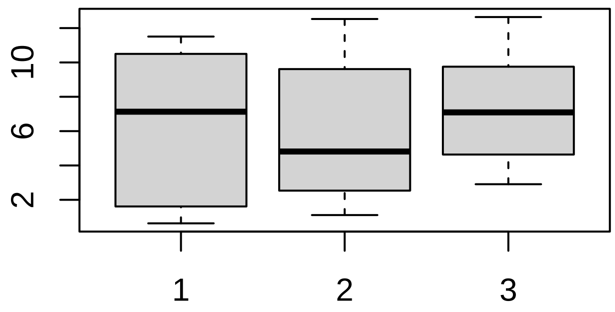
```r
# Generate 10 random numbers uniformly distributed between 0 and 15, store
# them in a vector x.vec:
x.vec <- runif(n = 10, min = 0, max = 15)
is.vector(x.vec)

## [1] TRUE
```

```r
boxplot(x.vec)                   # x.vec is a numeric vector.
```



```r
# Generate 30 random numbers between 0 and 15, store them in three
# columns of a matrix x.mat:
x.mat <- matrix(runif(n = 30, min = 0, max = 15), nrow = 10, ncol = 3)
```

```r
boxplot(x.mat)                   # x.mat is a matrix with 3 columns.
```



When passed a `"numeric"` vector, `boxplot()` produces a single boxplot, but when passed a `"matrix"`, it produces side-by-side boxplots of the matrix's columns.

11.2 Methods

- Actually, a generic function such as `boxplot()` is a whole *family* of functions, each of which is referred to as a ***method*** and is designed to handle a *specific* class of objects.

  When we pass an object to a generic function like `boxplot()`, the function determines the class of the object and then ***dispatches*** it to the appropriate *method*.

- We can view the methods associated with a generic function using:

```
methods()       # Determine the S3 methods that are associated with a
                # given generic function
```

- For example, to see the methods for `boxplot()` type:

```
methods(boxplot)

## [1] boxplot.default  boxplot.formula* boxplot.matrix
## see ?methods for accessing help and source code
```

  When we pass to `boxplot()` an object of class `"matrix"`, R passes it along to the `boxplot.matrix()` *method*, which produces side-by-side boxplots of its columns (as above).

  If we were to pass an object of class `"formula"` to `boxplot()`, R would pass it along to the `boxplot.formula()` *method*.

  All other objects, including `"numeric"` vectors, get passed along the `boxplot.default()` *method*.

- *Methods must have names of the form* `fname.cname()`, where `fname()` is the name of the generic function (e.g. `boxplot` in the example above) and `cname` is the class of objects (e.g. `"matrix"` in the example above) that the method accepts or it is `default`.

Section 11.2 Exercises

Exercise 67 The function `summary()` is a *generic* function, meaning it does something different depending on the *class* of the object passed to it.

a) Use `methods()` to look at `summary()`'s *methods*. Is there a method for `"data.frames"`? How about for `"factors"`?

b) Try the following commands, and describe the results:

```
x <- data.frame(x1 = c(4, 3, 6),
                x2 = c(4, 7, 9),
                x3 = c(1, 1, 3))
summary(x)
```

```
y <- factor(c("a", "a", "b", "c", "c", "c", "c"))
summary(y)
```

c) In part *b*, the *generic* function `summary()` *dispatched* (passed) each object (`x` and `y`) to the appropriate `summary()` *method*. Now try the following commands, which bypass the generic function and instead pass `x` and `y` to the appropriate `summary()` *method* directly. Compare the results with those of part *b*.

```
summary.data.frame(x)
```

```
summary.factor(y)
```

12 Packages and the Search Path (B.5)

12.1 Packages

- A ***package*** is a collection of specialized functions (or other R objects such as data sets) that are stored together in a bundle.

  Some packages are built into R. Others need to be downloaded and installed from the Internet.

- To see which packages are built-in, type:

```
search()

##  [1] ".GlobalEnv"        "package:stats"     "package:graphics"
##  [4] "package:grDevices" "package:utils"     "package:datasets"
##  [7] "package:methods"   "Autoloads"         "package:base"
```

  As can be seen, the packages `stats`, `graphics`, `grDevices`, etc. are built-in.

- To download and install a non-built-in package, you can use **pull down menu** in **R** or **RStudio**, or you can use:

```
install.packages()      # Install a package from the web.  You can also
                        # use the pull-down menu in the R console.
```

- For example, to download and install the `"ggplot2"` package, type:

```
install.packages("ggplot2")
```

  You'll be prompted to choose a ***CRAN mirror*** (web server) from which to download the package.

- After downloading and installing a package, you need to load it into the current R session using either of the functions:

```
library()       # Loads a package into the current R session.  Must be
                # called from the command line.
require()       # Loads a package into the current R session.  Can be
                # called from within a function.
```

- Both `library()` and `require()` load a previously installed package into the current R session. `require()` is more versatile because it can be called from within a function, whereas `library()` can only be called from the R command line.

  As an example, to load the `"ggplot2"` package, type:

```
library(ggplot2)
```

- Once a package has been loaded into the current R session, all of its functions and other objects (such as data sets) will be available for use.

- You can see what objects are contained in a package by looking at it's help page, for example by typing

```
help(package = ggplot2)
```

12.2 The Search Path

- When a package is loaded into the current R session, the objects it contains are placed into an *environment*. An ***environment*** is a ”container” that holds R objects.

  Two different objects can share the same name (e.g. `x`) as long as they’re stored in different *environments*. So placing a package’s objects into their own *environment* avoids the possibility of existing objects being over-written when the package is loaded.

- To understand how R looks for objects when it needs them, it helps to understand the *search path*. The ***search path*** is a sequence of environments through which R searches when it encounters the name of an object while executing a command.

- When R encounters the name of an object in a command, it searches one environment after another in the order specified by the *search path* until it either finds the object or reaches the end of the search path, in which case it prints an error message:

```
ffrg

## Error in eval(expr, envir, enclos):  object 'ffrg' not found
```

- Typing `search()` shows the current search path:

```
search()              # Shows the current search path.
```

- For example:

```
search()

##  [1] ".GlobalEnv"         "package:stats"     "package:graphics"
##  [4] "package:grDevices"  "package:utils"     "package:datasets"
##  [7] "package:methods"    "Autoloads"         "package:base"
```

  This indicates that when R encounters the name of an object, it first searches the ***Global Environment*** (another name for the **Workspace**). If it doesn’t find the object there, it searches the environment associated with the `"stats"` package, then the one associated with the `"graphics"` package, and so on.

- When a package is downloaded from the Internet and then loaded into the current R session, R places the package in the 2nd position in the search path, right after the Global Environment (Workspace). For example, after installing and loading the `"ggplot2"` package:

```
install.packages("ggplot2")
library(ggplot2)
```

  we get:

```
search()

##   [1] ".GlobalEnv"        "package:ggplot2"     "package:stats"     "package:graphics"
##   [5] "package:grDevices" "package:utils"     "package:datasets"   "package:methods"
##   [9] "Autoloads"         "package:base"
```

  Notice that `"ggplot2"` is in the 2nd position.

- If a user-defined object (which will be stored in the Workspace) shares the same name as one in a package, when the package is loaded R prints a message indicating that the object in the package is ***masked*** by the one in the Workspace.

  For example, the `"ggplot2"` package has a function called `qplot()`. If we create a *user-defined* function of the same name in the Workspace, and then load `"ggplot2"`, we get:

```r
qplot <- 5
```

```r
library(ggplot2)

## Attaching package:  'ggplot2'
##
## The following object is masked by '.GlobalEnv':
##
## qplot
```

This means that when R encounters the name `qplot` while executing a command, it will use the user-defined version of `qplot()`, not the one that's in the `"ggplot2"` package.

- To override the search path and access a function in a specific package, use:

```
packagename::functionname()          # Access a function in a specific package.
```

For example, to use `qplot()` from the `"ggplot2"` package (when another object named `qplot` is in the Workspace), type:

```r
ggplot2::qplot(x = my.x, y = my.y)
```

(Output not shown.)

13 Some R Programming Features (B.4)

13.1 The Logical Operations ”And”, ”Or”, and ”Not”

13.1.1 Logical Operations and Compound Logical Expressions

- ***Logical operators*** (or ***Boolean operators***) correspond to ”and”, ”or”, and ”not”, and are written in R as:

```
&               # "And"
|               # "Or"
!               # "Not"
```

- These operate on `"logical"` (`TRUE` or `FALSE`) expressions and return `TRUE` or `FALSE` values. They're listed above in order of operator precedence (highest to lowest).

13.1.2 Logical Operations on Scalar Logical Expressions

- `&` returns `TRUE` if both expressions are `TRUE`, and it returns `FALSE` if at least one expression is `FALSE`:

```r
TRUE & TRUE

## [1] TRUE
```

```r
TRUE & FALSE

## [1] FALSE
```

- `|` returns `TRUE` if at least one of the expressions is `TRUE`, and it returns `FALSE` if both expressions are `FALSE`:

```
FALSE | TRUE

## [1] TRUE
```

```
FALSE | FALSE

## [1] FALSE
```

- The negation operator, `!`, returns the ”opposite” of a logical expression:

```
!TRUE

## [1] FALSE

!FALSE

## [1] TRUE
```

- As an example, to test whether a variable `x` lies *between* two numbers (60 and 70), we type:

```
x <- 63
x > 60 & x < 70

## [1] TRUE
```

and to test whether it lies *outside* the range (60 to 70), we can type:

```
x < 60 | x > 70

## [1] FALSE
```

- The logical operators `&`, `|`, and `!` operate *elementwise* on `"logical"` vectors (i.e. they're *vectorized*). For example:

```
c(TRUE, FALSE, TRUE) & c(TRUE, TRUE, FALSE)

## [1]  TRUE FALSE FALSE
```

- As another example, here are two vectors, `Syst` and `Diast`, containing systolic and diastolic blood pressures for five people:

```
Syst <- c(110, 119, 111, 113, 128)
Diast <- c(70, 74, 88, 74, 83)
```

A blood pressure considered normal if the systolic level is less than 120 *and* the diastolic level is less than 80. To identify the people with normal blood pressures, we can type:

```
Syst < 120 & Diast < 80

## [1]  TRUE  TRUE FALSE  TRUE FALSE
```

or we can use `which()`:

```
which(Syst < 120 & Diast < 80)

## [1] 1 2 4
```

To identify the people whose blood pressures are *not* normal, we can type:

```
!(Syst < 120 & Diast < 80)

## [1] FALSE FALSE  TRUE FALSE  TRUE
```

- Pay attention to the operator precedence of the logical operators, which from highest to lowest, is `!`, `&`, and `|`. More information can be found by typing:

  ```
  ? Syntax
  ```

  Parentheses can be used to control the order of operations.

Section 13.1 Exercises

Exercise 68 Here are two variables `x` and `y`:

```
x <- 4
y <- 7
```

Guess what the result of each of the following will be, then check your answers.

a) `x > 2 & y == 7`

b) `x < 0 | y == 7`

c) `!(x < 0)`

Exercise 69 Recall that the operator precedence of the logical operators, from highest to lowest, is `!`, `&`, and `|`. The order of operations can be controlled using parentheses. Guess what the result of each of the following commands will be, then check your answers.

a) `10 < 20 | 15 < 16 & 9 == 10`

b) `(10 < 20 | 15 < 16) & 9 == 10`

c) `4 < 3 & (5 < 6 | 8 < 9)`

d) `(4 < 3 & 5 < 6) | 8 < 9`

Exercise 70 Recall that the logical operators operate *elementwise* on vectors. Guess what the result of each of the following will be, then check your answers.

a) `c(FALSE, TRUE, FALSE) & c(TRUE, TRUE, FALSE)`

b) `c(FALSE, TRUE, FALSE) | c(TRUE, TRUE, FALSE)`

c) `!c(FALSE, TRUE, FALSE)`

Exercise 71 Recall that `is.na()` will identify missing values (`NA`s) in a vector. Here's a vector `x`:

```r
x <- c(1, 2, NA, 6, 3, NA, 5)
```

Describe in words what the following command does:

```r
!is.na(x)
```

Exercise 72 Here's a vector `x`:

```r
x <- c(1, 2, 6, 5)
```

Guess what the result of each of the following commands will be, then check your answers.

a) `!(x > 4)`

b) `x > 4 & x < 6`

c) `!(x > 4 & x < 6)`

d) `x > 4 | x < 6`

Exercise 73 Here are two vectors, `Gender` and `Age`:

```r
Gender <- c("m", "f", "m", "f", "f")
Age <- c(27, 34, 55, 21, 43)
```

Guess what each of the following commands will return, then check your answers.

a) `Gender == "m" & Age > 40`

b) `Gender == "m" | Age > 40`

13.2 User-Defined Functions

- We can create a ***user-defined function*** using:

```
function()          # Used to create user-defined functions.
return()            # Used within a function definition to terminate
                    # the function call and return a value.
```

- Here's a simple (mathematical) function $f(x)$:

$$f(x) \ = \ x^2$$

In R, we can represent this as a *user-defined* function that takes an argument `x` and returns its square:

```r
my.f <- function(x) {
  return(x^2)
}
```

We use *user-defined* functions just as we would built-in functions:

```r
my.f(x = 2)

## [1] 4
```

We can look at the function definition by typing its name on the command line:

```r
my.f

## function(x) {
##   return(x^2)
## }
```

- The general format for a *user-defined function* is:

```r
my.fun <- function(arg1, arg2, ..., argk) {
     statement1
     statement2
         .
         .
         .
     statementq
     return(value)              # The value to be returned.  We could  also just
   }                            # write value (instead of return(value)).
```

Above,

  - `my.fun()` is the function name (that we're free to choose).
  - `arg1`, `arg2`, ..., `argk` are argument names (that we're free to choose) for k arguments.
  - `statement1`, `statement2`, ..., `statementq` are a set of q statements (which may involve `arg1`, `arg2`, ..., `argk`).
  - `value` is a value (or expression that gives a value) to be returned by the function. It could also be any R object (vector, list, etc.).

- If the body of the function consists of just a `value` (or expression), we can omit the curly brackets `{ }` as long as we write the entire function definition on a single line:

```r
my.fun <- function(arg1, arg2, ..., argk) return(value)
```

For example:

```r
my.f <- function(x) return(x^2)
```

- As another example, here's a (mathematical) function that computes the average absolute value of two numbers x and y:

$$f(x, y) = \frac{|x| + |y|}{2},$$

We can represent this as a *user-defined* function `AvgAbsVal()` in R using:

```r
AvgAbsVal <- function(x, y) {
  avg <- (abs(x) + abs(y)) / 2
  return(avg)
}
```

We can now call `AvgAbsVal()`, passing it values via the arguments `x` and `y`:

```
AvgAbsVal(x = -4, y = 2)

## [1] 3
```

- In fact, we don't need to use `return()`. In the absence of a `return()` statement, R returns the value of any expression that appears by itself as the last line of the function definition, just before the closing bracket `}`:

```
AvgAbsVal <- function(x, y) {
  avg <- (abs(x) + abs(y)) / 2
  avg
}
```

13.2.1 Formal Arguments and Actual Arguments

- A function's arguments are sometimes referred to as ***formal*** arguments. Values passed to the function are referred to as ***actual*** arguments.
- For example, below, `x` is a *formal* argument, and `z` is an *actual* argument:

```
g <- function(x) {
  x + 1
}
```

```
z <- 5
g(x = z)                # x is a formal argument, z is an actual argument

## [1] 6
```

13.2.2 Specifying Default Values for a Function's Arguments

- We can specify *default* values for one or more of a function's arguments by specifying `arg = expr` in the function definition:

```
my.fun <- function(arg1, arg2 = expr2, ..., argk = exprk) {
     statement1
     statement2
         .
         .
         .
     statementq
     value
  }
```

- For example, below we define `AvgAbsVal()` so that the default value for `y` is `NULL`.

```
AvgAbsVal <- function(x, y = NULL) {
  if (is.null(y)) {
    return(abs(x))
  } else {
    avg <- (abs(x) + abs(y)) / 2
    return(avg)
  }
}
```

Note the use of the `if()` statement. Note also that `return()` is necessary because the function call might terminate before the last line is encountered.

Now, if a value for `y` isn't passed to `AvgAbsVal()`, it uses `NULL` and returns `abs(x)`:

```
AvgAbsVal(x = -120)

## [1] 120
```

13.2.3 Variable Number of Arguments Using "..."

- Functions can be written to take a *variable number* of arguments. The argument name `...` in the function definition will match any number of arguments.

  Within the body of the function, we can refer to `...` as if it was the name of a variable.

- For example, here's a function that returns the mean of all the values in an arbitrary number of vectors:

```
mean.of.all <- function(...) {
    mean(c(...))
}
```

  If `us.sales`, `europe.sales`, and `other.sales` were vectors, the command

```
mean.of.all(us.sales, europe.sales, other.sales)
```

  would combine them and take the mean of the combined data. The effect of `c(...)` is as if `c()` were called with the same three arguments, `us.sales`, `europe.sales`, and `other.sales`, that were passed to `mean.of.all()`.

- Many of R's built-in functions take a variable number of arguments. For example look at the help pages for `list()` and `c()` by typing:

```
? list
? c
```

13.2.4 Printing Warning or Error Messages Using `warning()` or `stop()`

- The following functions are useful for terminating a function call or just printing a warning message:

```
stop()          # Terminate a function call and print an error message.
warning()       # Print a warning message (without terminating the
                # function call).
```

- `stop()` and `warning()` are usually used in `if()` statements within functions.
- `stop()` terminates a function call (without returning a value) and prints an error message. Here's an example:

```
my.ratio <- function(x, y) {
    if (y == 0) stop("Cannot divide by 0")
    return(x / y)
}
```

  An attempt to pass the value `0` for `y` now results in the following:

```
my.ratio(x = 3, y = 0)

## Error in my.ratio(x = 3, y = 0):  Cannot divide by 0
```

  Note that the last line, `return(x / y)`, was never encountered during the call to `my.ratio()`.

- `warning()` just prints a warning message to the screen without terminating the function call. Here's an example:

```
my.ratio <- function(x, y) {
    if (y == 0) warning("Attempt made to divide by 0")
    return(x / y)
}
```

Now when we pass the value `0` for `y`, the function call isn't terminated (`Inf` is returned), but we get a warning message:

```
my.ratio(x = 3, y = 0)

## Warning in my.ratio(x = 3, y = 0):  Attempt made to divide by 0

## [1] Inf
```

Note that in this case, the last line, `return(x / y)`, *is* encountered during the call to `my.ratio()`.

Section 13.2 Exercises

Exercise 74 Here's a function:

```
f1 <- function(x) {
    y <- x + 1
    return(y)
  }
```

If we replace `return(y)` by just `y`:

```
f2 <- function(x) {
    y <- x + 1
    y
  }
```

does the function do the same thing?

Exercise 75 In the code below, which argument (`x` or `z`) is the *formal* argument and which is the *actual* argument?

```
g <- function(x) {
  x^2 - 1
}
```

```
z <- 2
g(x = z)               # Which argument, x or z, is formal and which is actual?

## [1] 3
```

Exercise 76

a) Write a function that takes two arguments, `x` and `y`, and returns their *relative difference*, defined as

$$f(x, y) = \left|\frac{x-y}{y}\right|,$$

where $|\cdot|$ is the absolute value. Test your functions by passing it a few different values for `x` and `y`.

b) What happens when you pass it the value `y = 0`? What about when you pass it `x = 0` and `y = 0`.

c) Rewrite your function so that it specifies a *default* value of 1 for `y`.

Exercise 77 Write a function that takes a vector argument `x` and returns a *list* containing the mean, median, standard deviation, and range of `x`. Use `mean()`, `median()`, `sd()`, and `range()`.

Exercise 78 Look at the help page for `list()` by typing

```
? list
```

What do the `...`'s mean when it says `list(...)` under Usage?

Exercise 79

a) Write a function that takes two vectors `x` and `y` and returns the maximum value in the two vectors combined. **Hint**: Use `max()` with `c(x, y)`.

b) Modify your function so that it uses `...` to take a *variable number* of vectors as arguments, and returns the maximum value in all the vectors combined. **Hint**: Use `max()` with `c(...)`.