

Ecological Dynamics

Dr. Tarik C. Gouhier

Copyright © 2014-2015 Tarik C. Gouhier

PUBLISHED ONLINE

<http://www.northeastern.edu/synchrony>

Licensed under the Creative Commons Attribution-NonCommercial-NoDerivs 3.0 Unported License (the “License”). You may not use this file except in compliance with the License. You may obtain a copy of the License at <http://creativecommons.org/licenses/by-nc-nd/3.0>.

First Edition, Spring 2015

Contents

1	Introduction to Ecological Dynamics	i
1.1	Course description	i
1.2	Course goals	i
1.3	Student evaluation	i
1.3.1	Assignments	ii
1.3.2	Project proposal	ii
1.3.3	Peer-review of project proposal	ii
1.3.4	Final project	ii
1.3.5	Presentation of the final project	ii
1.4	Organization of the course manual	ii
1.5	Course manual conventions	ii
2	An R programming tutorial	1
2.1	What is programming?	1
2.2	Why learn to program in R?	2
2.3	An overview of R	3
2.4	Getting started with R	4
2.5	Components of R	6
2.6	Objects and variables	6
2.7	Compound objects	8
2.8	Combining objects	12
2.9	Manipulating compound objects	14
2.9.1	Vectors	14
2.9.2	Matrices	18
2.9.3	Arrays	23

2.9.4	Data frames	24
2.9.5	Lists	29
2.9.6	Converting compound objects	32
2.10	Manipulating text	34
2.10.1	Characters	34
2.10.2	Factors	37
2.11	Functions	39
2.12	Operators	40
2.12.1	Assignment operators	41
2.12.2	Mathematical operators	42
2.12.3	Logical operators	43
2.12.4	When operators go bad	45
2.13	Input and Output	46
2.13.1	Managing your R session	46
2.13.2	Navigating the filesystem	47
2.13.3	Importing data	49
2.13.4	Exporting data	50
2.13.5	Packages	50
2.14	Data wrangling	50
2.14.1	Combining datasets	51
2.14.2	Reshaping datasets	52
2.14.3	Computing statistics	54
2.15	Plotting	58
2.15.1	Base system	59
2.15.2	Types of plots	60
2.15.3	Customizing plots	64
2.15.4	Alternative systems	69
2.15.5	Saving plots	71
2.16	Programming	71
2.16.1	Control structures	71
2.16.2	Random numbers	74
2.17	Getting help	75
3	Reproducibility and R markdown	76
3.1	Using literate programming to combat the reproducibility crisis	76
3.2	Installing R markdown	77
3.3	Using R markdown for assignments	77
	Index	80



1 — Introduction to Ecological Dynamics

1.1 Course description

This course is designed to provide students with a comprehensive overview of the mathematical and computational concepts needed to construct (meta)population, (meta)community, and (meta)ecosystem models. The lectures will describe how to mathematically derive and model processes such as growth, trophic and non-trophic species interactions, dispersal, and environmental variability to understand patterns of population abundance and species diversity in a changing world. Special emphasis will be placed on the mathematical tools required to (1) analyze the dynamical behavior of ecological models (e.g., stability, invasion, graphical, and numerical analyses) and (2) validate model predictions using empirical data (e.g., via maximum likelihood and optimization methods). The supervised lab sessions will teach students how to derive, analyze, and test models using the free R programming environment.

1.2 Course goals

The overall goals of the course are to (1) provide an overview of the mathematical and computational approaches used to derive and construct ecological models; (2) discuss the suitability of each approach for tackling different classes of ecological and environmental issues; (3) teach the programming and numerical skills needed to test model predictions with empirical data; (4) allow students to develop the writing and presentation skills required to successfully convey complex concepts to scientists and non-scientists alike.

1.3 Student evaluation

This course will emphasize critical thinking skills and understanding over memorization. Student performance will thus be assessed using a series of practical, real-world problems selected from a range of ecological topics. Specifically, the evaluations will be based on biweekly assignments in R and a final project. Each student's final project will consist of a scientific manuscript that tackles an ecological topic of general interest using novel mathematical or computational models. Students will also be required to submit a proposal for their project and (peer)-review other students' project proposals. Finally, students will present their project at the end of the term.

1.3.1 Assignments

The assignments will consist of problem sets designed to assess comprehension of the lecture material and the R programming environment. These should be completed using the `Rmarkdown` template provided on Blackboard.

1.3.2 Project proposal

Students will write a 2-page (single-spaced) proposal for their final project containing the following sections: “Introduction & Background”, “Questions & Goals”, “Methodological Approach”, “Expected Results & Implications”. The “Introduction & Background” section should describe an open question/topic in any biological field and outline its importance. The “Questions & Goals” section should clearly delineate the questions your project will attempt to answer and your hypotheses/predictions. The “Methodological Approach” section should describe how you plan to carry out the research (e.g., what kind of model and analysis will you use?). Finally, the “Expected Results & Implications” section should describe what you expect to find and the implications of your results for scientists and the broader public.

1.3.3 Peer-review of project proposal

Students will review the proposals of their peers by ranking them based on criteria such as clarity, novelty, feasibility, and potential applicability to real-world issues.

1.3.4 Final project

For the final project, students will write a scientific manuscript based on their proposal that adheres to the format of a *Letter* for the journal *Ecology Letters*. Specifically, the manuscript will have the following structure: “Abstract”, “Introduction”, “Methods & Materials”, “Results”, “Discussion”, “References”, “Tables”, “Figures”; all in no more than 5,000 words and 6 tables/figures. The manuscript will present a novel model that tackles the topic described in the proposal. The paper will also include an appendix describing the rationale for the model derivation and analysis, along with the R code used to produce the results.

1.3.5 Presentation of the final project

Students will prepare and deliver a 30-minute presentation of their final project.

1.4 Organization of the course manual


The second chapter presents a thorough introduction to processing and plotting data in the R environment. The third chapter presents a brief introduction to `Rmarkdown`, the literal programming language that students will use to complete their assignments.

1.5 Course manual conventions

The course manual was written using the typesetting language \LaTeX , with extensive examples of R code embedded in the text using `knitr`. The text is interspersed with R code presented in gray colored blocks:

```
"hello world!"  
  
## [1] "hello world!"
```

All R output appears after the `##` symbols. The rest of the commands in the code block represent R input. Important tips and tricks about R or statistical theory are highlighted using special note blocks:

 This is a note.

Finally, to facilitate navigation, the PDF of the course manual contains bookmarks for each section and subsection of the text, along with live links in the table of contents and the index.



2 — An R programming tutorial

2.1 What is programming?

A programming language is composed of a suite of **reserved words** whose use is governed by a set of **syntactic rules**. When assembled in compliance with the syntactic rules, these words form a **program** that can be used to achieve **specific tasks**. Programming languages are thus directly analogous to natural languages, which use sets of words (i.e., **vocabulary**) and syntactic rules (i.e., **grammar**) to achieve a specific task (i.e., **convey an idea**). The main difference is that there are many fewer reserved words and syntactic rules in programming languages than there are in natural languages. There are four main reasons for learning to program:

- **Repeatability:** code that is developed once can be used to analyze subsequent datasets (i.e., code once, run many times).
- **Reproducibility:** anyone can reproduce the results using the code. This is becoming increasingly important in this era of open science.
- **Rapid turn-around:** once bugs are fixed upstream in the analytical pipeline, the results and figures downstream can be easily regenerated (i.e., fix upstream, propagate downstream).
- **Modern analyses:** many modern computer-based methods such as Monte Carlo simulations, machine learning, likelihood and Bayesian statistics require some level of programming.

Programming languages are either **compiled** or **interpreted**. Compiled languages typically take less time to complete a task because they have stricter rules (e.g., strict **typing** of data structures) and less overhead. However, program syntax must be verified by a compiler and then converted to efficient machine code (i.e., a series of zeros and ones) before being executed. This extra step (compilation) in the programming pipeline can make identifying problems (i.e., debugging) more difficult. Interpreted languages such as R are more user-friendly because programs can be evaluated immediately (no need for compilation), thus allowing analysts to identify problems more rapidly. However, that user-friendliness comes at a price: relatively slow speeds.

Choosing a programming language thus involves evaluating a trade-off between performance or run time T_R and development time T_D . Programs written in faster, compiled programming languages such as C typically take longer to develop than those written in slower, interpreted languages such as Python, R and MATLAB (Fig. 2.1). A programmer's goal should be to select the language that minimizes the total time T_T required to complete a project, knowing that $T_T = T_D + T_R$.

Hence, there is no such thing as an *ideal programming language*. Rather, the ideal language (i.e., the one that minimizes T_T) will depend on the scope and scale of the programming task. For programs that are expected to run for a very long time on large datasets, one should select fast, efficient compiled languages such as C that minimize running time T_R at the cost of development time T_D . Conversely, for programs that are expected to run relatively quickly on small datasets, one should select interpreted languages such as R that minimize development time T_D at the cost of running time T_R .

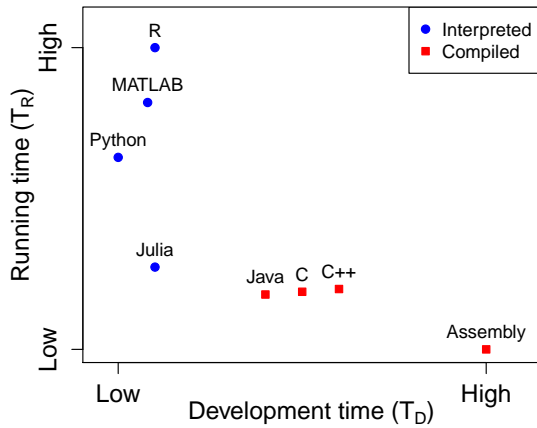


Figure 2.1: Trade-off between running time and development time

2.2 Why learn to program in R?

Given that there is no “right” or “wrong” programming language, why choose to use R for biostatistics? The first reason is purely practical: R is free and **available** for all major computing platforms including Mac OS X, Windows, and Linux. This means that you can download R on your personal computer and do your assignments from the comfort of your own home instead of having to find a free computer in the lab.

The second reason is that R was created for statisticians by statisticians¹. This means that it is highly extensible and has sensible defaults for performing statistical calculations, unlike other general computational languages such as C that were designed by computer scientists. Additionally, R is currently enjoying explosive growth in terms of its popularity (i.e., number of users) and functionality (i.e., number of statistical packages; Fig. 2.2). The explosive growth in the size of the user-base means that you can easily find help and solutions to most problems by consulting online mailing lists or forums such as **RSeek** and **StackOverflow**. The growth in R's functionality means that it is becoming the *de facto* language of quantitative biology, with tools for performing analyses across different levels of biological organization: from

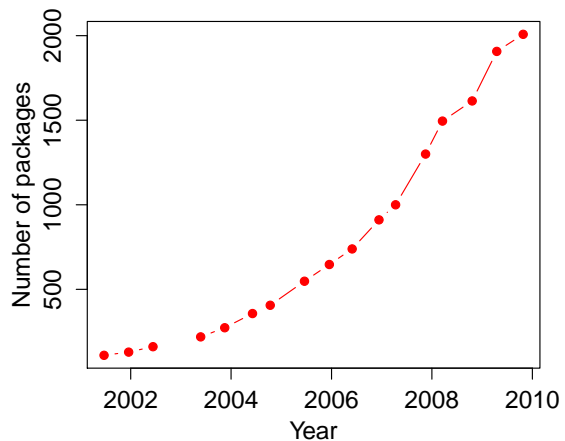


Figure 2.2: Growth of statistical packages for the R programming environment

¹University of Auckland (New Zealand) statisticians Ross Ihaka and Robert Gentleman created R, which is a free implementation of the (once) popular commercial programming language S/SPplus written by John Chambers at Bell Labs.

genomic approaches to understand processes occurring within cells (see [bioconductor](#)) to spatial statistics to identify ecological patterns across continental scales (see [CRAN](#)).

Indeed, in rapidly developing fields such as bioinformatics and genomics, technological innovations are spurring the emergence of novel statistical techniques, which are being implemented primarily in R. In these cases, R is the only game in town unless you are willing to invest a considerable amount of time reimplementing these statistical techniques in a different programming language. Hence, learning R for biostatistics is both a practical time investment and a sensible strategy to improve your career prospects.

2.3 An overview of R

Programs consist of **commands** or **expressions** that, once evaluated in R, perform specific tasks such as mathematical calculations, loading or saving files, or plotting data. Programs typically also contain embedded **comments** that describe the code in plain language. These comments are ignored by R, but critically important for avoiding mistakes, particularly when writing complex programs. In R, comments start with the special character `#`. Any line beginning with that special character will not be executed by R.

Multiple commands are typically stored in text files called **scripts**, whose names must end with a `‘.R’` extension (e.g., `myScript.R`). Scripts can be evaluated line-by-line or in their entirety. When evaluated in their entirety, the commands in the script are interpreted sequentially by R. Commands can also be placed in **functions**. Functions are sections of code (i.e., a sequence of commands) that take **parameters** or **arguments** as input, perform a series of operations and then either (i) return the results to be saved in **variables** or (ii) have side-effects (i.e., plot the results or write files to the hard drive). Hence, programming functions are directly analogous to mathematical functions in that they take an argument (i.e., input) and return a value (i.e., output). Functions are saved in text files that end with a `‘.R’` extension (e.g., `myFunction.R`).

Figure 2.3 shows how these different concepts relate to one another in a typical R workflow using RStudio. Generally, one starts to enter commands directly at the console or command line, where they are evaluated by R. Once satisfied with the results, these commands can be copied & pasted into the Editor window and saved as a script for later use. Scripts are ideal for working on a single ‘sample’ dataset, but if the script is truly useful, it may be worth moving the code to a function. Doing so will allow you to apply the code to any dataset, not just the ‘sample’ dataset used to develop the code in the first place. For example, if you develop a script to convert temperatures from Fahrenheit to Celsius for a ‘sample’ dataset, it may be useful to move that code into a function in order to be able to perform the conversion for any arbitrary dataset. In this case, the function would take the dataset as an argument (i.e., input) and return the converted temperature (i.e., output). The final step in the workflow typically consists of moving functions that perform related tasks into a package. For instance, if you create multiple functions that convert temperatures between Celsius, Fahrenheit, and Kelvin, you may

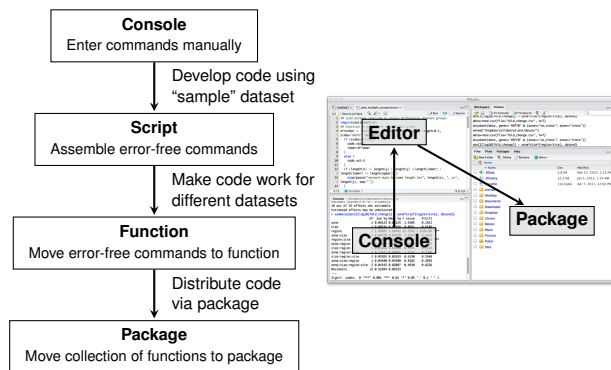


Figure 2.3: Typical workflow in R

want to move them into a ‘temperature’ package.

2.4 Getting started with R

If you have not already done so, please [download](#) and install R. Although R comes with its own Integrated Development Environment or IDE, we will be using a new and better open-source IDE called **RStudio**. The job of the IDE is to make your life easier. You will be spending a lot of time writing code and analyzing results, so you need to make sure that you have the best environment available. IDEs provide a single application to program, access help functions, download packages, and run your code. Once you have installed and launched RStudio, you will be greeted with a 4-panel window:

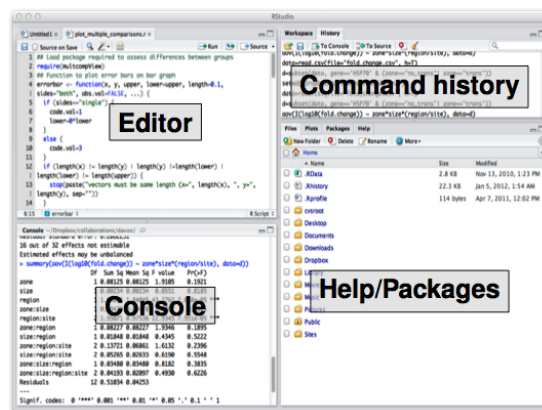


Figure 2.4: RStudio on a Mac

The different tabs in the top left panel show the content of the open **script** files. The different tabs in the top right panel show the **command history** (i.e., the commands that were run in the past) and the **workspace** (i.e., the variables and functions that were created during the R session). The bottom left panel shows the **console** where the output of the commands appears. The different tabs in the bottom right panel show plotted figures, a list of installed packages, **help** on specific functions, and the list of files in the current working directory. Keyboard shortcuts can be used to switch between the different panels (e.g., for Macs **control-1**: editor, **control-2**: console, **control-3**: help).

In RStudio, it is advisable to place commands directly in a new R script instead of typing them one at a time in the console. This is because commands placed in a script can be sent to the console for execution via RStudio’s graphical user interface (i.e., “Run” icon in the top right corner of the Editor panel) or keyboard shortcuts. Individual lines of code or commands can be sent to the console for execution by using the keyboard shortcut **command-return** on a Mac or **control-enter** on Windows and Linux. Alternatively, highlighting a few lines of code and using the same keyboard shortcut will send the whole block to the console for execution. Finally, to execute the entire script, click on the “Source” icon in the top right corner of the Editor window or use the keyboard shortcut **command-shift-return** on a Mac.

RStudio has a number of other extremely powerful features, most of which are easily discoverable. Here, I will focus on *the* killer feature: code **completion** via the keyboard shortcut **tab** in both the console and editor panels. After typing the first few leading letters of a function or variable name, hit the **tab** key to get a list of candidate functions and variables that start with those letters:

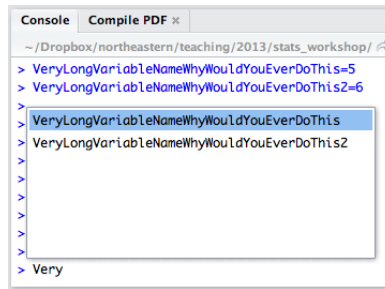


Figure 2.5: tab completion of variable name

Using the up and down keys on the keyboard will scroll through the list of candidate functions/variables and pressing **return** on a Mac or **enter** on Windows/Linux will select a command. For functions, tab completion shows the package that the function belongs to in curly braces and a small snippet of its documentation:

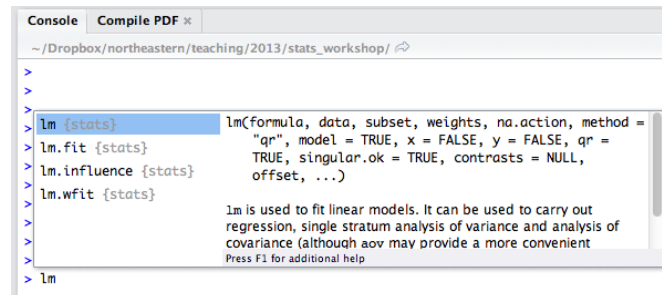


Figure 2.6: tab completion of function name

In this example, the `lm` function has quite a few arguments. To get additional details about the function, hit the **F1** key to bring-up the full documentation for the function in the help tab located in the bottom right panel of RStudio. Crucially, **tab** completion also works for function arguments. Once the list of arguments appears, use the up and down keys to scroll through the list of arguments and then press **return** to have the selection appear in the console:

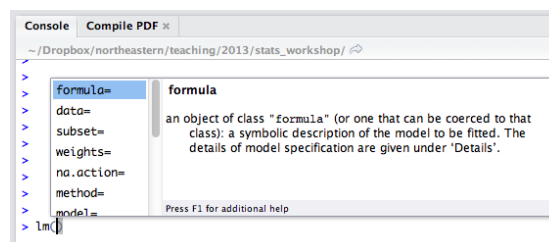


Figure 2.7: tab completion of function arguments

To get further help on any function or command, type `help.search("commandName")` or `?commandName` at the console to open the local R documentation within RStudio. To search online forums for help, type `RSiteSearch("commandName")` at the console. Doing so will open a web browser window.

N The **F1** shortcut will only work on a Mac if the “Use all F1, F2, etc. keys as standard functions” option is selected in the **Keyboard** section of the **System Preferences**.

2.5 Components of R

Programs use a combination of components such as **variables**, **operators**, and **functions** to perform specific tasks. Hence, to design effective and efficient programs, one has to master these components by understanding the rules that govern their use and implementation. The components of the R programming language are summarized in the table below and described in greater detail in the following sections:

Component	Description
Variables	Used to store results from calculations. They can contain many different types of values including numbers, characters, or collections of characters or numbers. A variable that contains a collection of values is called a vector if it is 1-Dimensional, a matrix or data frame if it is 2-Dimensional, and an array if it is $N > 2$ Dimensional. A variable is generally referred to as an object in R, regardless of whether it contains a single value or a complex, multi-dimensional collection of values.
Operators	Perform assignments (e.g., storing a value into a variable), mathematical calculations (e.g., $5+5$), and logical operations (e.g., compare two values: $6 > 5$). Mathematical operators return the result of the calculations, whereas logical operators return TRUE or FALSE (e.g., $6 > 5$ will return FALSE).
Functions	Perform a series of commands or calculations. These typically take 1 to N arguments as input(s) and return a single variable as output. Functions always adhere to the following format: <code>functionName(arg1=val1, arg2=val2, ..., argN=valN)</code> . For example <code>rep(x=5, times=2)</code> will repeat 5 twice.

2.6 Objects and variables

In R, variables are used to store objects. All objects have a specific type or **class** based on their content and structure:

Class	Subclass	Description	Example
numeric	integer	Whole number	1, -2
	double	Floating point number	1.3, -2.5
	complex	Complex number	$5+6i$
character	character	Text	"a", "word"
factor	factor	Special enumerated list	low, mid, high

Variable names can contain a combination of letters, numbers and certain special characters (e.g., `'.'` and `'_'`). However, variable names cannot **begin** with special characters or numbers. R is case-sensitive so `var1` and `Var1` represent two distinct variables. Variable names should be short but informative. Additionally, one should avoid creating variable names that 'clash' with R function names such as `mean` or `sum`². To store objects in variables, one can use the assignment operators `'='` or `'<='`. These two operators are generally equivalent for simple assignments, but the latter is strongly preferred. For our first example, let's create two variables and compute their sum:

²This is not strictly necessary as R can tease apart variables and functions that have the same name

```
var1 <- 3
Var1 <- 4 # This is a comment: anything following # is ignored
my.sum <- var1 + Var1
my.sum

## [1] 7
```

The expressions or commands in the code block above were entered at the console and evaluated in R (i.e., input into R). The lines that begin with `##` represent the evaluated commands (i.e., output from R). The first line assigned the value 3 to variable `var1` using the `<-` assignment operator and the second line assigned the value 4 to variable `Var1`. The third line computed the sum of `var1` and `Var1` and stored the result into a new variable called `my.sum`. By default, R does not print the results to the screen. To print the contents of `my.sum` (or any variable) to the screen, the variable must be placed on a separate line by itself. However, we can combine those last two steps by wrapping our assignment into parentheses³:

```
(my.sum <- var1 + Var1)

## [1] 7
```

The contents of `var1` and `Var1` are summed, stored into `my.sum` and printed to the screen, all in one line of code. Note that this command overwrote the original content of `my.sum` without issuing a warning. We can also create variables that contain text:

```
(my.text <- "hello world")

## [1] "hello world"
```

Note that in the evaluated output, the content of variable `my.text` is ‘hello world’ in double quotes, providing visual confirmation that the class of the variable is `character`. Function `class` can be used to determine the class of a variable directly:

```
class(my.text)

## [1] "character"

class(var1)

## [1] "numeric"
```

Knowing the class of a variable can be important in order to diagnose problems. For instance, certain operations are only defined for variables of the same class:

```
my.text + var1

## Error in my.text + var1: non-numeric argument to binary operator
```

The code above generates an error because the ‘+’ operation is not defined for `character` and `numeric` variables. At this point, we have seen that R can be used as a glorified scientific calculator. Let’s push forward to discover more interesting and unique functionality.

³Alternatively, `print(a)` can be used to print variable `a` to the screen

2.7 Compound objects

Variables can also be used to store **compound objects** such as collections of objects (e.g., numbers, characters, etc...). Typically, a 1-Dimensional compound object is called a **vector**, a 2-Dimensional compound object is called a **matrix** or **data frame**, and a N -Dimensional compound object is called an **array**. Compound objects can even be combined and stored in a **list**. Each object stored within a compound object is associated with an address or **index**. This index can be used to retrieve or alter the object. The following functions can be used to create compound objects of different classes:

Function	Description
<code>numeric(length=N)</code>	Create a vector of numerical values of length N initially filled with zeros.
<code>logical(length=N)</code>	Create a vector of logical values of length N initially filled with <code>FALSE</code> .
<code>character(length=N)</code>	Create a vector of character values of length N initially filled with 'empty' characters.
<code>vector(mode="logical", length=N)</code>	Create a vector of logical values of length N . This is equivalent to <code>logical(length=N)</code> . The <code>vector</code> function can also be used to create vectors of character or numerical values by setting the appropriate <code>mode</code> . For example, <code>vector(mode="numeric", length=N)</code> produces a vector of numerical values of length N .
<code>matrix(data=vals, nrow=N, ncol=M)</code>	Store <code>vals</code> in a 2-D table or matrix with N rows and M columns. Matrices can contain only one type or class of values (e.g., either numeric , character , or logical but no combination).
<code>data.frame(col1.name=vals1, col2.name=vals2, ..., colM.name=valsM, stringsAsFactors=TRUE)</code>	Store vectors <code>vals1</code> , <code>vals2</code> , ..., <code>valsM</code> in a special 2-D table with M named columns. Data frames are one of the most powerful features of R because they can be used to store columns that contain multiple different classes of data (e.g., numeric , character , and logical). By default, data frames convert characters to factors. However, this default behavior can be turned off by setting <code>stringsAsFactors=FALSE</code> .
<code>array(data=vals, dim=c(dim1, dim2, ..., dimM))</code>	Store <code>vals</code> in an M -dimensional array of length <code>dim1</code> along dimension 1, <code>dim2</code> along dimension 2, ..., <code>dimM</code> along dimension M .
<code>list(var1=vals1, var2=vals2, ..., varN=valsN)</code>	Store variables <code>var1</code> , <code>var2</code> , ..., <code>varN</code> in a list . The variables can be of any type, including numerical , character , matrix , array , or data.frame .

Let's see how this works in practice by creating a vector of numbers of length 5:

```
# Create a vector of zeros of length 5
(nums <- numeric(length = 5))

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

R created a vector with 5 zeros and stored it in the variable `nums`. Zero is the default initial value assigned to `numeric` vectors. Next, let's create a vector of characters of length 3:

```
# Create an empty character vector of length 3
(chars <- character(length = 3))

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

R created a vector with 3 'empty' characters as space-holders and stored it in the variable `chars`. Note the presence of double quotes for each element in the vector to indicate that this is a `character` vector. Let's create our first 2-D data structure: a 3-row x 2-column matrix of 4:

```
# Create a 3-row x 2-column matrix of 4
(mat <- matrix(4, nrow = 3, ncol = 2))

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

R created the matrix and stored it in variable `mat`. The matrix also has column and row headings such as `[,1]`, which reflect the index or location of each element. We will come back to this later when we learn how to access specific parts of a matrix. Next, let's create a data frame consisting of a numerical column and a logical column:

```
# Create a 2-column data frame containing numerical and
# logical values
(df <- data.frame(my.numbers = numeric(length = 5), is.one = logical(5)))

##   my.numbers is.one
## 1          0 FALSE
## 2          0 FALSE
## 3          0 FALSE
## 4          0 FALSE
## 5          0 FALSE
```

R created the data frame and stored it in variable `df`.



The column and row headings of the data frame look different than those of the matrix created earlier. We will discuss the differences between matrices and data frames in a later section.

Now, let's create our first 3-D object:

```
# Create a 3 x 4 x 2 array of 1 3 is the length along
# dimension 1 4 is the length along dimension 2 2 is the
# length along dimension 3
array(data = 1, dim = c(3, 4, 2))
```

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



The output of the array looks a little different because it is a 3-D structure represented in two dimensions (i.e., a computer monitor).

The first entry of the array along its third dimension is a matrix of ones with 3 rows and 4 columns. The second entry of the array along its third dimension is another matrix of ones with 3 rows and 4 columns. Now, let's create a new **list** to store the compound objects that we previously created, namely the data frame **df** and the matrix **mat**:

```
# Combine matrix mat and data frame df into a list
list(df, mat)

## [[1]]
##   my.numbers is.one
## 1          0 FALSE
## 2          0 FALSE
## 3          0 FALSE
## 4          0 FALSE
## 5          0 FALSE
##
## [[2]]
##      [,1] [,2]
## [1,]    4    4
## [2,]    4    4
## [3,]    4    4
```

The first line of the output shows the index of the list (i.e., `[[1]]`) and then the contents of the list at that index (i.e., data frame **df**). The second index of the list (i.e., `[[2]]`) appears after those entries, along with the contents of the list at that index (i.e., matrix **mat**).

There are a few additional functions that can be extremely useful for generating or altering patterned compound objects:

Function	Description
<code>rep(x, times=5)</code>	Create a vector by repeating the content of <code>x</code> 5 times
<code>rep(x, each=5)</code>	Create a vector by repeating each element of <code>x</code> 5 times
<code>seq(from=1, to=8, by=1)</code>	Create a vector containing a sequence of values starting from 1 and ending with 8, in increments of 1
<code>1:10</code>	Create a vector containing a sequence of values starting from 1 and ending with 10, in increments of 1. This is equivalent to <code>seq(from=1, to=10, by=1)</code>
<code>seq(from=1, to=8, length=10)</code>	Create a vector containing a sequence of 10 linearly-spaced values between 1 and 8

For instance, to repeat a simple sequence twice:

```
# Repeat the entire sequence 1,2,3 twice
rep(x = 1:3, times = 2)

## [1] 1 2 3 1 2 3

# Repeat each element in the sequence 1,2,3 twice
rep(x = 1:3, each = 2)

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

Both of these commands generated vectors of the same length ($3 \times 2 = 6$), whose elements were repeated the same number of times (twice). The only difference between these vectors is the order of the elements. To create a vector of fixed length between a minimum and a maximum value, one can use the `seq` function by specifying the `length` argument:

```
# Create a sequence of 8 linearly-spaced values between
# 1 and 4
seq(from = 1, to = 4, length = 8)

## [1] 1.000000 1.428571 1.857143 2.285714 2.714286 3.142857 3.571429 4.000000
```

These functions can also be used to create more interesting matrices:

```
# Create a 2-row x 4-column matrix filled with values
# 1-8
matrix(1:8, ncol = 4, nrow = 2)

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

In the code above, the numbers in the sequence `1:8` were used to fill the 2-row x 4-column matrix by columns. To fill the matrix by rows, we can specify the argument `byrow=TRUE`:

```
matrix(1:8, ncol = 4, nrow = 2, byrow = TRUE)
```

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

2.8 Combining objects

As we saw in the previous section, it is easy to generate compound objects using functions like `numeric` and `matrix`. However, we can also create compound objects by combining previously created objects via the following functions:

Function	Description
<code>c(obj1, obj2)</code>	combine <code>obj1</code> and <code>obj2</code> . The objects can be single values or compound objects
<code>rbind(mat1, mat2)</code>	Append the rows of <code>mat2</code> to those of <code>mat1</code> via row-bind
<code>cbind(mat1, mat2)</code>	Append the columns of <code>mat2</code> to those of <code>mat1</code> via column-bind

For example, to combine variables containing a single value or vectors, one can use the function `c`:

```
val1 <- 18
val2 <- 4
# Combine val1 and val2 into a vector of size 2
(vals.vec1 <- c(val1, val2))

## [1] 18  4

# Combine vals.vec with a new vector containing 8, 9, 5
(vals.vec2 <- c(vals.vec1, c(8, 9, 5)))

## [1] 18  4  8  9  5
```

We can also create compound objects such as matrices and data frames by combining previously created vectors. For example, to add a row of numbers to a matrix, one can use the function `rbind`:

```
(mat <- matrix(3, nrow = 3, ncol = 2))

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

# Create a new row vector, which has the same number of
# columns as the matrix
new.row <- c(4, 2)
# Add new.row at the end of mat
rbind(mat, new.row)

##      [,1] [,2]
##      3    3
##      3    3
##      3    3
## new.row  4    2
```

```
# Add new.row at the beginning of mat
rbind(new.row, mat)

##          [,1] [,2]
## new.row    4    2
##           3    3
##           3    3
##           3    3
```

Similarly, one can add a column to a matrix by using the function `cbind`:

```
(mat <- matrix(3, nrow = 3, ncol = 2))

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

# Create a new column vector, which has the same number
# of rows as the matrix
new.col <- c(5, 2, 6)
# Add new.col at the end of mat
cbind(mat, new.col)

##          new.col
## [1,] 3 3      5
## [2,] 3 3      2
## [3,] 3 3      6

# Add new.col at the beginning of mat
cbind(new.col, mat)

##          new.col
## [1,]      5 3 3
## [2,]      2 3 3
## [3,]      6 3 3
```

We can combine these steps to simultaneously add a new row and column to a matrix:

```
(mat <- matrix(3, nrow = 3, ncol = 2))

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

# Add a row vector and a column vector to mat in a
# single step
cbind(rbind(mat, c(4, 2)), c(5, 2, 6, 8))

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

Reading the last command from inside-out, we (1) created a 2-element vector containing the values 4 and 2 using the `c` function first, (2) then we appended that vector as a row to matrix `mat` using the function `rbind`, (3) then we created a 4-element vector

containing the values 5, 2, 6, 8, and (4) then we appended that 4-element vector as a column to the matrix.

2.9 Manipulating compound objects

Storing collections of objects in compound objects is only useful if they can be retrieved and altered at a later time. To facilitate the process, a number of utility functions exist to determine the structure and content of compound objects. Below, I describe how the following functions can be used to manipulate each type of object in R.

Function	Description
<code>length(x)</code>	Returns the number of elements in <code>x</code> . If <code>x</code> is a matrix or a data frame, returns the number of columns.
<code>NROW(x)</code> ; <code>NCOL(x)</code>	Return the number of rows and columns of <code>x</code> , respectively. If <code>x</code> is a vector, they both return its length.
<code>nrow(x)</code> ; <code>ncol(x)</code>	Same as their uppercase counterparts, except that they return <code>NULL</code> if <code>x</code> is not multidimensional.
<code>dim(x)</code>	Returns the number of dimensions of <code>x</code> , with dimension 1 = number of rows, dimension 2 = number of columns. Returns <code>NULL</code> if <code>x</code> is not multidimensional.
<code>colnames(x)</code> ; <code>rownames(x)</code>	column names and row names of <code>x</code> , respectively. Return <code>NULL</code> if <code>x</code> is not multidimensional. Can also be used to set column names and row names, respectively.
<code>names(x)</code>	Returns name of objects stored within a compound object. Returns <code>NULL</code> if <code>x</code> is objects are not named. Can also be used to set names.
<code>str(x)</code>	Returns the structure of <code>x</code> , which includes the name of the objects stored in <code>x</code> , along with their first values.
<code>head(x, n=6)</code> ; <code>tail(x, n=6)</code>	Return the first and last 6 rows of <code>x</code> , respectively.

2.9.1 Vectors

Vectors are used to store collections of values of the same class (e.g. numbers, characters, logicals). Each value in a vector has a unique address called an **index**, which can be **numerical** (e.g., position five in a vector) or **named** (e.g., the position called 'x' in a vector). One can also create an *ad hoc* **logical** index by evaluating whether the values of the vector meet a specific criterion (e.g., values greater than 6; more on this later). These numerical, named, and logical indices can be used to retrieve or alter the values of a vector by using the special square brackets indexing operator `[]`.

Numerical indices

All vectors possess numerical indices, which range from 1 (i.e., first position in the vector) to the length of the vector (i.e., the last position in the vector). Let's create a vector of size 4 and retrieve the third value:

```
v <- c(1, 3, 5, 7)
# Show the length of v
length(v)
```

```
## [1] 4

# Show value at index 3:
v[3]

## [1] 5
```

Now, let's replace the value at index 3 with 88:

```
# Replace value at index 3 with 88
(v[3] <- 88)

## [1] 88

v

## [1] 1 3 88 7
```

Let's replace the first two values in `v`. Because we're selecting two values in `v`, we will need to have two replacement values:

```
# Show the values in index 1, 2
v[1:2]

## [1] 1 3

# Replace the values in index 1, 2 with values 9, 2
v[1:2] <- c(9, 2)
v

## [1] 9 2 88 7
```

We can alter values that occur across a discontinuous index range. For instance, to change the values at indices 1, 2, and 4:

```
# Show the values in index 1, 2, 4
v[c(1:2, 4)]

## [1] 9 2 7

# Replace the values in index 1, 2, 4 with values 6, 45,
# 72
v[c(1:2, 4)] <- c(6, 45, 72)
v

## [1] 6 45 88 72
```

Negative indices can be used to exclude the values at specific indices:

```
# Retrieve all values in v except for the one at index 3
v[-3]

## [1] 6 45 72
```

Named indices

One can also use **named** or character indices, which are easier to remember when dealing with very large vectors, to retrieve and alter values in vectors. Named indices can be assigned when (i) vectors are created or (ii) added to preexisting vectors by using the function `names`. For example, let's add named indices to the previously created vector `v`

```
# Add named indices to v; notice that the indices are
# characters
names(v) <- c("ind1", "ind2", "ind3", "ind4")
# The named indices appear as labels above each value
v

## ind1 ind2 ind3 ind4
##    6   45   88   72
```

N If the vector of named indices is smaller than the vector of values (i.e., there are fewer named indices than values), the values without a named index will be labeled `<NA>` and impossible to access by name. If this happens, R will not produce an error or a warning, so be vigilant.

Now, we can access the values of `v` by using either numerical or named indices:

```
# Access values using numerical indices
v[c(1, 3)]

## ind1 ind3
##    6   88

# Access values using named indices
v[c("ind1", "ind3")]

## ind1 ind3
##    6   88
```

N The named indices must be enclosed in double (or single) quotes because they are characters. If they were not enclosed in double (or single) quotes, R would interpret them as variable names and issue an error.

To create a vector with named indices at the onset, we can use the `c` function:

```
# Create a vector with named indices
(v2 <- c(first = 6, second = 6, third = 109))

## first second third
##     6      6   109

# str shows that v2 is a named numerical vector
str(v2)

## Named num [1:3] 6 6 109
## - attr(*, "names")= chr [1:3] "first" "second" "third"

# Access value at third index
v2[3]

## third
##   109
```

```
# Same using named index
v2["third"]

## third
## 109
```

Logical indices

It is also possible to select elements of a vector based on values instead of indices by using **logical operators** (e.g., greater than: `>`, smaller than: `<`, equal to: `==`) to create logical indices by evaluating whether a certain criterion is met. For instance, we may want to retrieve values in a numerical vector that are greater than some arbitrary number without knowing the index or location of those values. Achieving this task involves a two-step process: (1) determine which values meet the criterion (e.g., greater than some arbitrary number) and (2) retrieve those values. The first step finds the indices of the values that we want to retrieve. For example, to find the indices of the values that are greater than 6 in vector `v2`:

```
# Find values of v2 that are greater than 6
(ind <- v2 > 6)

## first second third
## FALSE FALSE TRUE
```

The returned vector of indices `ind` is composed of logical values (i.e., `TRUE` and `FALSE`). The `TRUE` entries in `ind` correspond to indices where the values of `v2` are greater than 6 (i.e., meet the criterion), whereas the `FALSE` entries in `ind` correspond to indices where the values of `v2` are smaller than or equal to 6 (i.e., fail to meet the criterion). When we apply vector `ind` to `v2`, we will only retrieve the values in `v2` where the `ind` is `TRUE`:

```
# Retrieve the values of v2 that are greater than 6
v2[ind]

## third
## 109
```

Naturally, these two steps can be combined:

```
# Find and retrieve the values in v2 greater than 6
v2[v2 > 6]

## third
## 109
```

Sorting

Vectors can be sorted by using the following functions:

Function	Description
<code>sort(x, decreasing=FALSE)</code>	Sorts <code>x</code> in ascending order (i.e., from low to high values). Set <code>decreasing=TRUE</code> to sort in descending order.
<code>order(x, decreasing=FALSE)</code>	Returns the (ascending) order of <code>x</code> (i.e., from low to high values). Set <code>decreasing=TRUE</code> to return the descending order.
<code>rev(x)</code>	Reverses the order of <code>x</code> .

For instance, to sort a vector of characters alphabetically:

```
v <- c("a", "quick", "brown", "fox", "jumps", "over", "the",
      "lazy", "dog")
# Sort alphabetically
sort(v)

## [1] "a"      "brown" "dog"   "fox"   "jumps" "lazy"  "over"  "quick" "the"

# Get the ascending order of the elements in v
order(v)

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

We can also use function `order` to generate an index to sort `v` without using the `sort` function:

```
v[order(v)]

## [1] "a"      "brown" "dog"   "fox"   "jumps" "lazy"  "over"  "quick" "the"
```

2.9.2 Matrices

Matrices can be used to store values of any single class into a 2-D table format consisting of rows and columns. To access and retrieve values within a matrix, one can use numerical, named, or logical indices.

Numerical indices

Values within a matrix can be accessed by using either a 1-Dimensional or a 2-Dimensional (numerical) index system. By convention, the 1-Dimensional index system is incremented across rows first and then columns. This leads to the following 1-Dimensional index for an arbitrary N -row by M -column matrix:

$$\begin{matrix} & \text{column 1} & \text{column 2} & \text{column 3} & \dots & \text{column } M \\ \begin{bmatrix} 1 \\ 2 \\ \vdots \\ N \end{bmatrix} & \begin{matrix} N+1 \\ N+2 \\ \vdots \\ 2N \end{matrix} & \begin{matrix} 2N+1 \\ 2N+2 \\ \vdots \\ 3N \end{matrix} & \begin{matrix} \dots \\ \dots \\ \ddots \\ \dots \end{matrix} & \begin{matrix} (M-1)N+1 \\ (M-1)N+2 \\ \vdots \\ MN \end{matrix} \end{matrix}$$

Here's a practical example for a $N = 2$ by $M = 4$ matrix:

```
# Create a 2 x 4 matrix
(m <- matrix(c(1, 3, 2, 5), nrow = 2, ncol = 4))

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

# Get dimensions of m
dim(m)

## [1] 2 4

# Get number of rows in m
nrow(m)
```

```
## [1] 2

# Get number of columns in m
ncol(m)

## [1] 4
```

Now that we've verified the properties of matrix `m`, let's see how linear indices work:

```
# Index 2 of the matrix is row 2, column 1
m[2]

## [1] 3
```

The index moves down rows before moving across columns. In the case of this matrix, index 3 will be located in the first row of the second column:

```
# Index 3 of the matrix is row 1, column 2
m[3]

## [1] 2
```

As you can tell, it can be a little tricky to use a 1-Dimensional index system to navigate through a 2-Dimensional object. Thankfully, a much more convenient and intuitive 2-Dimensional index system exists. This system uses square brackets placed around a comma (i.e., `[row,col]`). The value that comes to the left of the comma (i.e., `row`) indicates the position along the rows and the value that comes to the right of the comma (i.e., `col`) indicates the position along the columns. For instance, `m[2,4]` would retrieve the value in row 2 and column 4 of matrix `m`:

```
# 2-Dimensional system: retrieve row 2, column 4
m[2, 4]

## [1] 5

# 1-Dimensional system: retrieve index 8
m[8]

## [1] 5
```

We can also retrieve an entire column or row by leaving either side of the comma within the square bracket blank. For example, `m[, 3]` retrieves all rows of column 3, whereas `m[2,]` retrieves all columns of row 2:

```
# Print m
m

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

# Get structure of m
str(m)

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

# Get all rows of column 3
m[, 3]
```

```
## [1] 1 3

# Get all columns of row 2
m[2, ]

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

N The default row and column labels of matrix `m` should now make sense to you. For example, to access all the rows from column 1, you would use `m[,1]`, which happens to correspond to the label for column 1. Similarly, to access all the columns from row 1, you would use `m[1,]`, which corresponds to the label for row 1. The output from `str(m)` should also be clear: `m` is a matrix of numerical values with row numbers that range from 1 to 2 and column numbers that range from 1 to 4.

We can also retrieve groups of columns or rows by using the `c` function. For example, to retrieve values in rows 1, 2 and columns 1, 3:

```
# Retrieve values in rows 1, 2 and columns 1, 3
m[1:2, c(1, 3)]

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

Negative indices can be used to exclude certain rows and columns. For instance, to exclude columns 2, 4:

```
m[, -c(2, 4)]

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

Named indices

As with vectors, we can define a named index for matrices. To name the columns and rows, we can use the functions `colnames` and `rownames`, respectively. These are analogous to the `names` function used to create named indices for vectors:

```
# Name columns: need as many names as there are columns
colnames(m) <- c("col1", "col2", "col3", "col4")
# Name rows: need as many names as there are rows
rownames(m) <- c("r1", "r2")
m

##      col1 col2 col3 col4
## r1      1    2    1    2
## r2      3    5    3    5
```

N Remember that the named indices have to be enclosed in single or double quotes because they are characters.

We can now use the named index to extract values from the matrix:

```
# Named index for row 1, column 4
m["r1", "col4"]

## [1] 2

# Numerical index for row 1, column 4
m[1, 4]

## [1] 2

# Retrieve all columns from row 1 via named index
m["r1", ]

## col1 col2 col3 col4
##    1    2    1    2

# Same via numerical index
m[1, ]

## col1 col2 col3 col4
##    1    2    1    2
```

Logical indices

Logical indices can be used to extract values from matrices along 1- or 2-Dimensions. For example, to extract the values that are equal to 5 in matrix `m`:

```
# Create matrix
(m <- matrix(c(1, 3, 2, 5), nrow = 2, ncol = 4))

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

# Index of values that equal to 5
(ind <- m == 5)

##      [,1] [,2] [,3] [,4]
## [1,] FALSE FALSE FALSE FALSE
## [2,] FALSE  TRUE FALSE  TRUE

# Retrieve those values
m[ind]

## [1] 5 5

# Or in one step
m[m == 5]

## [1] 5 5
```

To extract values that are equal to 5 in a specific column or row, we must specify an index along both dimensions. For example, to retrieve values that are equal to 5 in column 4:

```
# Index of values that are equal to 5 in column 4
(ind <- m[, 4] == 5)

## [1] FALSE  TRUE
```



```
# Retrieve those values
m[ind, 4]

## [1] 5

# Or in one step
m[m[, 4] == 5, 4]

## [1] 5
```

Similarly, to extract values that are equal to 2 in row 1:

```
# Index of values that are equal to 2 in row 1
(ind <- m[1, ] == 2)

## [1] FALSE TRUE FALSE TRUE

# Retrieve those values
m[1, ind]

## [1] 2 2

# Or in one step
m[1, m[1, ] == 2]

## [1] 2 2
```

Sorting

Matrices can be sorted based on one or many columns by using the `order` function. For instance, to sort a matrix based on column 1:

```
(m <- matrix(20:1, nrow = 5, ncol = 4))

##      [,1] [,2] [,3] [,4]
## [1,]  20  15  10   5
## [2,]  19  14   9   4
## [3,]  18  13   8   3
## [4,]  17  12   7   2
## [5,]  16  11   6   1

m[order(m[, 1]), ]

##      [,1] [,2] [,3] [,4]
## [1,]  16  11   6   1
## [2,]  17  12   7   2
## [3,]  18  13   8   3
## [4,]  19  14   9   4
## [5,]  20  15  10   5
```

To sort based on columns 1 and 2:

```
m[order(m[, 1], m[, 2]), ]

##      [,1] [,2] [,3] [,4]
## [1,]  16  11   6   1
## [2,]  17  12   7   2
## [3,]  18  13   8   3
## [4,]  19  14   9   4
## [5,]  20  15  10   5
```

In this case, `m` is sorted based on column 1 first and then column 2.

2.9.3 Arrays

The approaches described above for extracting values from 2-Dimensional matrices (namely numerical, named, and logical indices) also work for N-Dimensional arrays. For example, indexing in a 3-Dimensional array uses squared brackets with 3 terms, one for each dimension: `a[dim1, dim2, dim3]`:

```
# Create 3-Dimensional array
(a <- array(1:18, dim = c(2, 3, 3)))

## , , 1
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
##
## , , 2
##      [,1] [,2] [,3]
## [1,]    7    9   11
## [2,]    8   10   12
##
## , , 3
##      [,1] [,2] [,3]
## [1,]   13   15   17
## [2,]   14   16   18

# Get dimensions of a
dim(a)

## [1] 2 3 3

# Structure of a
str(a)

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

We can see that `a` is 3-Dimensional array of numbers, containing a 2-Dimensional matrix within each index along dimension 3. Here, dimension 1 corresponds to the rows of the matrix and dimension 2 to the columns. We can now extract values from this array:

```
# Extract matrix from index 1 along dimension 3:
a[, , 1]

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

# Extract row 1 from the matrix located at index 2 along
# dimension 3:
a[1, , 2]

## [1]  7  9 11
```

```
# Extract column 2 from matrix located at index 3 along
# dimension 3:
a[, 2, 3]

## [1] 15 16
```

2.9.4 Data frames

Data frames are special 2-D data structures that can be used to store column vectors of values belonging to different classes. They are thus ideal for storing statistical datasets, which often contain a mixture of character and numerical columns. To clarify the distinction between matrices and data frames, let's attempt to create a matrix of heights from a fictional population consisting of 2 males and 3 females:

```
# Create a vector of heights in inches
height <- c(68, 72, 65, 67, 66)
# Create vector of 2 males and 3 females
sex <- c(rep("male", 2), rep("female", 3))
# Create matrix
(mat <- cbind(sex, height))

##      sex      height
## [1,] "male"      "68"
## [2,] "male"      "72"
## [3,] "female"    "65"
## [4,] "female"    "67"
## [5,] "female"    "66"
```

That appears to have worked: we were able to create a matrix by combining a character column vector and a numerical column vector. However, closer inspection of the `mat` object shows that the heights values are all enclosed in double quotes, suggesting that they are coded as characters in the matrix even though they were created as numerical values. Let's check the structure of `mat` by using function `str`:

```
str(mat)

## chr [1:5, 1:2] "male" "male" "female" "female" "female" ...
## - attr(*, "dimnames")=List of 2
## ..$ : NULL
## ..$ : chr [1:2] "sex" "height"
```

The output of `str` clearly confirms what we suspected: the matrix `mat` consists of two character columns. This is problematic because we will not be able to use mathematical operations on the `height` column since the numerical values were converted to characters. Let's see what happens when we attempt to use a data frame to store the character and numerical vectors:

```
# Create data frame
(df <- data.frame(sex = sex, height = height))

##      sex height
## 1  male     68
## 2  male     72
## 3 female     65
## 4 female     67
## 5 female     66
```

```
# What's the structure of df?
str(df)

## 'data.frame': 5 obs. of  2 variables:
## $ sex   : Factor w/ 2 levels "female","male": 2 2 1 1 1
## $ height: num  68 72 65 67 66
```

That worked: the `str` function reports that `df` is a data frame that contains 5 observations of 2 variables. Furthermore, the class of the variables within the data frame is (close to) what we expected: the heights are numerical values and the sexes, which were originally characters, were converted to **factors** with two levels (male and female). By default, data frames convert all character vectors into factors, or enumerated lists, in order to save space. Each unique character value in the vector is replaced by a unique numerical value, and the map used to convert the numerical values into the character values (e.g., ‘male’=1 and ‘female’=2) is then ‘attached’ to the data frame as an **attribute**. This conversion is very efficient because it allows long character values (e.g., ‘female’) to be replaced by short numerical values (e.g., 2) in the column. However, because we keep a map of that conversion in the data frame, we can always back-convert from numerical values to character values if we wish to retrieve the original text. Indeed, that’s why when we print `df`, we see the values in column `sex` appear as the original text instead of the numerical representation used to store them in the data frame.

N In addition to using functions like `str` or `class` to distinguish characters from factors in an object, you can also simply print out the object and look for double quotes: if they appear around text values, then those values are represented as **characters**. However, if text entries appear without double quotes, then they are represented as **factors**.

Notice that `str` reports 5 observations and 2 variables for data frame `df`, but 5 rows and 2 columns for matrix `mat`. This is because data frames are typically used to represent statistical datasets that contain different observations along the rows and different variables along the columns. This subtle change in the structure of data frames has important implications for indexing.

Numerical indices

Indeed, 1-Dimensional indexing in data frames works very differently than it does in matrices:

```
# Get value at index 1 in the matrix
mat[1]

## [1] "male"

# Get value at index 1 in the data frame
df[1]

##      sex
## 1  male
## 2  male
## 3 female
## 4 female
## 5 female
```

Although index 1 returned the first value of the matrix, it returned the entire first column of the data frame. Hence, 1-Dimensional indexing in data frames cannot be

used to retrieve individual values, only individual columns. Thankfully, 2-Dimensional indexing works identically in data frames and matrices:

```
# Get row 1 all columns
mat[1, ]

##      sex height
## "male"    "68"

# Get observation 1 from all variables
df[1, ]

##      sex height
## 1 male     68
```

Here, both the matrix and the data frame return the same value, namely ‘male’. However, the data frame also prints the different **levels** (or unique values) of the factor.

Named indices

Data frames can also be accessed via named indices. All data frames have row and column names that can be retrieved or modified via the `rownames` and `colnames` functions. These named indices can be used to access or alter parts of a data frame:

```
# Rename the observations
rownames(df) <- c("obs1", "obs2", "obs3", "obs4", "obs5")
df

##      sex height
## obs1  male    68
## obs2  male    72
## obs3 female   65
## obs4 female   67
## obs5 female   66

# Get the sex and height for observations 1 and 3
df[c("obs1", "obs3"), c("sex", "height")]

##      sex height
## obs1  male    68
## obs3 female   65
```

Data frames also introduce a new way of accessing different variables (or columns) via the operator `$`. For example, to extract all observations from variable `sex`:

```
# Get all rows from column sex
df[, "sex"]

## [1] male  male  female female female
## Levels: female male

# Get all observations from variable sex
df$sex

## [1] male  male  female female female
## Levels: female male
```

The `$` operator returns vectors of values, which can be subset easily:


```
# Get rows 1,3 from column height
df[c(1, 3), "height"]

## [1] 68 65

# Get observations 1,3 from variable height
df$height[c(1, 3)]

## [1] 68 65
```

The `$` operator can also be used to remove or add columns to a data frame. To delete an existing column or variable from a data frame, one can simply set that column to `NULL`:

```
# Make a copy of df
df2 <- df
# Remove the height variable or column
df2$height <- NULL
df2

##           sex
## obs1   male
## obs2   male
## obs3 female
## obs4 female
## obs5 female
```

One can also use the `$` operator to add a new variable or column to a data frame. For example, to create a new column called `age`:

```
df2$age <- c(18, 17, 15, 19, 25)
df2

##           sex age
## obs1   male  18
## obs2   male  17
## obs3 female  15
## obs4 female  19
## obs5 female  25
```

Overall, `$` is merely a convenience operator that makes it easier to access and alter variables in data frames. However, there is nothing that the `$` operator does that cannot be achieved via numerical or named indices.

Logical indices

Logical indices operate identically in matrices and data frames. For example, to extract all rows with heights greater than 67 inches:

```
# Retrieve all rows with heights (i.e., column 2)
# greater than 67
df[df[, 2] > 67, ]

##           sex height
## obs1 male         68
## obs2 male         72

# Same with $ operator
df[df$height > 67, ]
```

```
##      sex height
## obs1 male     68
## obs2 male     72
```

One can also use the convenience function `subset(x, subset=crit, select=cols)` to retrieve and alter parts of a data frame. The first argument `subset` corresponds to the logical criterion that the values must satisfy in order to be retrieved and the second argument `select` specifies the columns of the data frame to retrieve. For example, to select only the sex of individuals whose heights are greater than 67 inches:

```
# Retrieve the sex of individuals whose heights > 67
subset(df, subset = height > 67, select = sex)

##      sex
## obs1 male
## obs2 male

# Same with logical indexing
df[df$height > 67, "sex"]

## [1] male male
## Levels: female male
```

Notice that `subset` returned a data frame object, whereas the logical indexing approach returned a vector.

N Technically, the `subset` function can also be used with vectors and matrices.

Sorting

Like matrices, data frames can be sorted based on one or more columns by using the `order` function. For instance, to sort a data frame based on column 1 (i.e., `sex`):

```
df <- data.frame(sex = c(rep("male", 2), rep("female", 3)),
  height = c(68, 72, 65, 67, 66), age = c(18, 17, 15,
    19, 25))
df[order(df$sex), ]

##      sex height age
## 3 female     65  15
## 4 female     67  19
## 5 female     66  25
## 1  male     68  18
## 2  male     72  17
```

To sort `df` based on columns 1 (i.e., `sex`), 2 (i.e., `height`), and 3 (i.e., `age`):

```
df[order(df$sex, df$height, df$age), ]

##      sex height age
## 3 female     65  15
## 5 female     66  25
## 4 female     67  19
## 1  male     68  18
## 2  male     72  17
```

In this case, `df` is sorted based on column 1 first, then column 2, and finally column 3. To sort the data frame in ascending order for `sex` and `age`, but descending order for `height`:

```
df[order(df$sex, -df$height, df$age), ]

##      sex height age
## 4 female     67  19
## 5 female     66  25
## 3 female     65  15
## 2  male     72  17
## 1  male     68  18
```

2.9.5 Lists

Lists can be used to store objects of different classes (e.g., matrices or vectors of numbers, characters, etc...). The objects within lists can be accessed via numerical indices using the operator `[[]]` or via named indices, including the `$` operator used for data frames.

Numerical indices

Numerical indices in lists are identical to those in vectors, matrices, and data frames. For example, let's create a list containing two vectors of numbers:

```
v1 <- c("january", "february", "march")
v2 <- 83:89
# Combine these vectors into a list
(l <- list(v1, v2))

## [[1]]
## [1] "january" "february" "march"
##
## [[2]]
## [1] 83 84 85 86 87 88 89

# Get structure of list
str(l)

## List of 2
## $ : chr [1:3] "january" "february" "march"
## $ : int [1:7] 83 84 85 86 87 88 89
```

List `l` consists of 2 elements, each of which is a numerical vector (more specifically integers or whole numbers). Here, the contents of vector `v1` are stored in index 1 of list `l` (i.e., `l[[1]]`) and those of vector `v2` are stored in index 2 of list `l` (i.e., `l[[2]]`). To access the entire vectors, we can use the list indices:

```
# Retrieve v1 stored in index 1
l[[1]]

## [1] "january" "february" "march"

# Retrieve v2 stored in index 2
l[[2]]

## [1] 83 84 85 86 87 88 89
```

We can also gain access to specific parts of each object stored in a list. For example, to access the first and the third values in vector `v1`, which is stored in index 1 of the list:

```
# Get the first and the third value of the object
# located in index 1 of the list
l[[1]][c(1, 3)]

## [1] "january" "march"
```

In this example, `l[[1]]` retrieves `v1`, a vector whose values can be accessed via `[]`. To add new objects to a list, we can use the `c` function:

```
v4 <- rep(x = c(1, 2), each = 3)
(l <- c(l, list(v4)))

## [[1]]
## [1] "january" "february" "march"
##
## [[2]]
## [1] 83 84 85 86 87 88 89
##
## [[3]]
## [1] 1 1 1 2 2 2
```

The last line of code first converts vector `v4` to a list, adds it to list `l`, and then stores the new list in `l`.

N When adding or replacing an object in a list in this manner, it is important to always convert the new object to a list first. For example, to add `vec1` to `list1`:
`list1=c(list1, list(vec1))`

However, it is much easier to add an object to a list by using list indices directly. For example, to add a matrix to the end of `l`:

```
mat <- matrix(1:8, nrow = 4, ncol = 2)
# Add matrix mat to index 4 of list l
l[[4]] <- mat
l

## [[1]]
## [1] "january" "february" "march"
##
## [[2]]
## [1] 83 84 85 86 87 88 89
##
## [[3]]
## [1] 1 1 1 2 2 2
##
## [[4]]
##      [,1] [,2]
## [1,]    1    5
## [2,]    2    6
## [3,]    3    7
## [4,]    4    8
```

We can also access parts of matrix `mat` in list `l` via indices. For example, to extract row 3 of matrix `mat`:


```
# Get row 3 of mat, which is stored in index 4 of l
l[[4]][3, ]

## [1] 3 7
```

We can remove objects from a list by using the special object `NULL`. For example, to remove the values stored in index 1:

```
l[[1]] <- NULL
l

## [[1]]
## [1] 83 84 85 86 87 88 89
##
## [[2]]
## [1] 1 1 1 2 2 2
##
## [[3]]
##      [,1] [,2]
## [1,]    1    5
## [2,]    2    6
## [3,]    3    7
## [4,]    4    8
```

 Removing an object from a list changes the indices of all remaining objects. Be very careful about such reorganizations when using numerical indices.

Named indices

Lists can also be accessed by using named indices. Named indices can be generated at the onset or added to a preexisting list. For example, to create a named list:

```
(l2 <- list(obj1 = v1, obj2 = v2))

## $obj1
## [1] "january" "february" "march"
##
## $obj2
## [1] 83 84 85 86 87 88 89
```

The objects can then be accessed directly via the `$` operator or via `[["name"]]`:

```
# Access obj2
l2$obj2

## [1] 83 84 85 86 87 88 89

l2[["obj2"]]

## [1] 83 84 85 86 87 88 89
```

To add named indices to an existing list, one can use the `names` function:

```
names(l) <- c("vec1", "vec2", "mat")
l

## $vec1
## [1] 83 84 85 86 87 88 89
##
## $vec2
## [1] 1 1 1 2 2 2
##
## $mat
```



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

l$mat

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

N Remember to specify as many names as there are indices in the list. Otherwise, the objects stored at unnamed indices will be inaccessible unless you use numerical indices.

2.9.6 Converting compound objects

The class of compound objects can be determined and altered by using utility functions `is.*` and `as.*`, respectively, where `*` represents an object class. To get a full listing of these functions, issue the following command at the R console: `methods(is)` or `methods(as)`. Below, I document just a few of these functions:

Function	Description
<code>is.character(x)</code> ; <code>x=as.character(x)</code>	Return TRUE if <code>x</code> is a character; convert <code>x</code> to a character
<code>is.numeric(x)</code> ; <code>x=as.numeric(x)</code>	Return TRUE if <code>x</code> is a number; convert <code>x</code> to a number
<code>is.matrix(x)</code> ; <code>x=as.matrix(x)</code>	Return TRUE if <code>x</code> is a matrix; convert <code>x</code> to a matrix
<code>is.data.frame(x)</code> ; <code>x=as.data.frame(x)</code>	Return TRUE if <code>x</code> is a data frame; convert <code>x</code> to a data frame
<code>is.array(x)</code> ; <code>x=as.array(x)</code>	Return TRUE if <code>x</code> is an array; convert <code>x</code> to an array

For example, to convert a vector from class `numeric` to `character`:

```
# Create a numerical vector
(v1.numb <- 1:4)

## [1] 1 2 3 4

# Is v1.numb is numerical vector?
is.numeric(v1.numb)

## [1] TRUE

# Convert from numeric to character
(v1.char <- as.character(v1.numb))

## [1] "1" "2" "3" "4"

# Is v1.char is character vector?
is.character(v1.char)

## [1] TRUE
```

Some classes of objects cannot be converted to other classes. For example, although numbers can be represented as numerical values or characters, the converse does not hold. Indeed, converting a vector of characters to a numeric class will fail and issue a warning:

```
v1.char <- c("a", "b", "c")
as.numeric(v1.char)

## [1] NA NA NA
```

Class conversion can also lead to unforeseen complications. For instance, let's create a matrix by combining two vectors of different classes:

```
# Create matrix by column binding a character and
# numeric vector
(mat <- cbind(c("a", "b", "c"), c(5, 2, 4)))

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

Because matrices can only contain values belonging to a single class, the numerical vector was converted to a character vector before the matrix was generated. This is problematic because we can no longer perform mathematical operations on what was once the numerical vector. Let's find out if we can solve this problem by converting the matrix to a data frame, since we know that the latter can handle objects of mixed classes:

```
# Convert to data frame
(df <- as.data.frame(mat))

##   V1 V2
## 1  a  5
## 2  b  2
## 3  c  4

# What's the structure of df?
str(df)

## 'data.frame': 3 obs. of  2 variables:
##  $ V1: Factor w/ 3 levels "a","b","c": 1 2 3
##  $ V2: Factor w/ 3 levels "2","4","5": 3 1 2
```

That did not work as expected. Although the print out of `df` looks fine, the output of `str` tells us that `df` contains two vectors of class **factor**. Hence, conversion to a data frame failed in this case. We should have started by using the proper object class to handle objects of mixed classes. However, can we get out of this jam by converting the second column of `df` from a factor to a numerical vector?

```
# Convert second column or variable of df to a numerical
# vector
df[, 2] <- as.numeric(df[, 2])
df

##   V1 V2
## 1  a  3
## 2  b  1
## 3  c  2
```

```
# What's the structure of df now?
str(df)

## 'data.frame': 3 obs. of 2 variables:
## $ V1: Factor w/ 3 levels "a","b","c": 1 2 3
## $ V2: num 3 1 2
```

That appears to have worked: the second column or variable of `df` is now of class `numeric`, which means that we can compute the mean, sum, etc... But a closer look at the values of the second column of data frame `df` and those of matrix `mat` reveals that they are not identical. This happened because factors encode characters by mapping each character entry in a vector to a unique numerical value. In this case, when we converted the vector of factors to a vector of numerical values (i.e., `df[,2]=as.numeric(df[,2])`), R converted the values used to encode the characters instead of the characters themselves! The moral of the story is that you should use the appropriate object type to store the data from the onset because object conversion won't always be able to save the day.

N Factor encoding is one of the biggest sources of conversion problems in R, so here is a detailed account of why it can fail. Factors encode character vectors into numerical vectors by using a mapping function f that takes each unique character c_i and attributes it a unique integer value v_i so that: $v_i = f(c_i)$. Hence, one can always retrieve the original values c_i encoded by the factor by using the inverse mapping function f^{-1} . However, conversion functions such as `as.numeric` ignore f : instead of converting the original character values c_i , they convert their encoded values v_i , thus potentially causing huge problems. This is not a bug *per se* because there is no way of systematically converting characters to numerical values.

2.10 Manipulating text

2.10.1 Characters

Statistical datasets often contain character vectors (also called strings). Unlike numerical values, character entries often need to be altered before being plotted or analyzed. These types of operations require the following utility functions, which are specific to the character class:

Function	Description
<code>grep(pattern=glob2rx("value"), x)</code>	Retrieve value in x
<code>gsub(pattern=glob2rx("value"), replacement, x)</code>	Replace all instances of value in each element of x with replacement
<code>sub(pattern=glob2rx("value"), replacement, x)</code>	Replace the first instance of value in each element of x with replacement
<code>substr(x, start, stop)</code>	Retrieve and/or replace the characters in x between index start and stop
<code>strsplit(x, split)</code>	Split character vector x based on split
<code>tolower(x), toupper(x)</code>	Change the elements of x to lower- and upper-case, respectively
<code>nchar(x)</code>	Number of characters in x
<code>format(x, ..., digits)</code>	Convert numeric x to character using 3 significant digits

One of the main purposes of these functions is to help bring some sanity to datasets. Indeed, datasets are typically polluted with extraneous characters, incorrect entries, and

other aberrations. Processing the text in datasets often consists of (1) making entries consistent by cleaning the text, (2) altering naming schemes by substituting text, (3) splitting text entries that contain distinct types of information.

Cleaning text

Datasets frequently contain incorrect or inconsistent text entries, especially when the data is collected and aggregated by several people. Some will capitalize text entries using ‘sentence case’, while others may decide to use lower case or upper case. Such inconsistencies, if left unchecked, can wreak havoc on statistical analyses. Functions `tolower` and `toupper` can alleviate any inconsistencies based on the case of the entries:

```
(bad.char.vec <- c("male", "MaLe", "mAlE", "MALE"))

## [1] "male" "MaLe" "mAlE" "MALE"

# Convert to all lower or upper case
tolower(bad.char.vec)

## [1] "male" "male" "male" "male"

toupper(bad.char.vec)

## [1] "MALE" "MALE" "MALE" "MALE"
```

Substituting text

Processing datasets often requires changing the way certain character values are encoded. For instance, to replace all ‘male’ entries with ‘man’ and all ‘female’ entries with ‘woman’:

```
vec <- c("male", "male", "female", "female")
# Replace all instances of 'male' with 'man'
vec <- gsub(vec, pattern = glob2rx("male"), replacement = "man")
# Replace all instances of 'female' with 'woman'
vec <- gsub(vec, pattern = glob2rx("female"), replacement = "woman")
vec

## [1] "man" "man" "woman" "woman"
```

N Functions `grep`, `sub`, `gsub` expect argument `pattern` to be a **regular expression**. Regular expressions are used to describe patterns of text using a powerful but complex set of rules. It is easier to use function `glob2rx` to convert patterns based on simple **wildcard** rules into the regular expressions expected by `grep`, `sub`, `gsub`.

Another frequent step when processing text is to abbreviate certain entries:

```
# Long state names
(long.states <- c("Massachusetts", "California", "Florida"))

## [1] "Massachusetts" "California" "Florida"

# Short state names
(short.states <- substr(start = 1, stop = 2, toupper(long.states)))

## [1] "MA" "CA" "FL"
```

The last line of code converted the entries in `long.states` to upper case and then selected the first and second letters to form the vector of abbreviated state names.

Splitting text

Sometimes, datasets aggregate multiple types of information into a single vector (e.g., 'short male'). To split these compound entries into separate vectors based on the type of information they represent, one can use the function `strsplit`:

```
vec <- c("short male", "short male", "tall female", "short female")
# Split entries separated by a single space, i.e., ' '
(split.vec <- strsplit(vec, " "))

## [[1]]
## [1] "short" "male"
##
## [[2]]
## [1] "short" "male"
##
## [[3]]
## [1] "tall"   "female"
##
## [[4]]
## [1] "short"   "female"

str(split.vec)

## List of 4
## $ : chr [1:2] "short" "male"
## $ : chr [1:2] "short" "male"
## $ : chr [1:2] "tall"  "female"
## $ : chr [1:2] "short" "female"
```

Function `strsplit` can thus be used to split a column containing two types of information (e.g., height and sex) into separate vectors. However, there is still some work to be done to convert the list returned by `strsplit` into distinct vectors. We can use the function `unlist` to convert the list to a vector

```
# Unlist converts the list into a single vector
unlist(split.vec)

## [1] "short" "male" "short" "male" "tall" "female" "short" "female"
```

However, we require two separate vectors, not one. Notice that the vector has a specific pattern. The first, third, fifth, and seventh entries contain information about the height of the individuals whereas the second, fourth, sixth, and eighth entries contain information about the sex of the individuals. We can create two distinct vectors by using this regular patterning (i.e., every other entry belongs to a different vector):

```
# Extract the heights from the common vector
(vec.height <- unlist(split.vec)[seq(from = 1, to = length(unlist(split.vec)),
  by = 2)])

## [1] "short" "short" "tall" "short"

# Extract the sex from the common vector
(vec.sex <- unlist(split.vec)[seq(from = 2, to = length(unlist(split.vec)),
  by = 2)])

## [1] "male" "male" "female" "female"
```

Combining text

Combining text and numerical vectors is often useful. Function `paste` can combine multiple character vectors or multiple character and numerical vectors:

```
# Character vector
chars <- c("a", "b")
# Numerical vector
pos <- 1:2
# Combine these vectors using paste
paste(chars, pos, sep = " is in position ", collapse = " of the alphabet; ")

## [1] "a is in position 1 of the alphabet; b is in position 2"
```

In this example, function `paste` combined the elements of `chars` and `pos` to form a single string. The `sep` argument specified that the elements of `chars` and `pos` should be separated by the string " is in position " and the `collapse` argument specified that the combined terms should be separated by the string " of the alphabet; ". The `paste` function can also be used to create descriptive labels for the rows and columns of matrices:

```
# Create a matrix with no unnamed columns and rows
(mat <- matrix(1:20, nrow = 4, ncol = 5))

##      [,1] [,2] [,3] [,4] [,5]
## [1,]   1   5   9  13  17
## [2,]   2   6  10  14  18
## [3,]   3   7  11  15  19
## [4,]   4   8  12  16  20

# Rename all columns
colnames(mat) <- paste("col", 1:NCOL(mat), sep = "")
# Rename all rows
rownames(mat) <- paste("row", 1:NROW(mat), sep = "")
mat

##      col1 col2 col3 col4 col5
## row1   1   5   9  13  17
## row2   2   6  10  14  18
## row3   3   7  11  15  19
## row4   4   8  12  16  20
```



R version 2.15 introduced the function `paste0(...)` as a shortcut for `paste(..., sep="")`.

2.10.2 Factors

Text is often encoded as a factor in R (e.g., in data frames). Factors are vectors that contain enumerated lists or categories. Each unique category of a factor is called a **level**. Because of their ubiquitous use in statistics, R provides a number of utility functions to make working with factors easier:

Function	Description
<code>factor(x, levels, labels)</code>	Convert character vector <code>x</code> into a factor vector with levels <code>levels</code> and labels <code>labels</code>
<code>gl(n, k, length=n*k, labels, ordered=FALSE)</code>	Create a factor vector with <code>n</code> levels and <code>k</code> replicates per level.
<code>level(x)</code>	Set or retrieve the levels of <code>x</code>
<code>nlevels(x)</code>	Get the number of levels of <code>x</code>
<code>relevel(x, ref)</code>	Reorder the levels so that <code>ref</code> is the first level
<code>reorder(x, X, FUN=mean, ...)</code>	Reorder the levels of <code>x</code> based on the mean of the values in <code>X</code>

Factors are often used to describe an experimental treatment. For example, let's say that we wanted to determine the effect of temperature on the growth of a plant. To do so, we grow 5 plants (replicates) at three distinct temperatures: 20°C, 25°C, and 30°C. Here, **temperature** is the factor and **20°C**, **25°C**, **30°C** are the levels. To encode such a factor in R:

```
(temps <- gl(n = 3, k = 5, label = c("20C", "25C", "30C")))

## [1] 20C 20C 20C 20C 20C 25C 25C 25C 25C 25C 30C 30C 30C 30C 30C
## Levels: 20C 25C 30C
```

The output of `temps` shows the different temperature levels for each replicate plant `k`. It also shows the unique levels in order. However, let's say that we want to change the order of the temperature levels so that 25C is the base level. To do so, we use function `relevel`:

```
(temp <- relevel(temps, "25C"))

## [1] 20C 20C 20C 20C 20C 25C 25C 25C 25C 25C 30C 30C 30C 30C 30C
## Levels: 25C 20C 30C
```

Doing so doesn't change the order of the vector `temps`, which would be mildly catastrophic, but it does change the base level. We can also create a factor vector by converting an existing character vector:

```
humid <- c("high", "low", "mid")
# Convert to factor
(humid <- factor(humid))

## [1] high low mid
## Levels: high low mid
```

That worked but the order of the levels is wrong. To fix the order, we can use function `levels`:

```
# Set arbitrary order for the levels
(levels(humid) <- c("low", "mid", "high"))

## [1] "low" "mid" "high"
```

2.11 Functions

Functions consist of blocks of code that take arguments as input, perform specific tasks (e.g., calculations), and then either (i) return the results in the form of a single object (i.e., output) or (ii) have side-effects (i.e., plot the results or write files to the hard drive). Functions in R are particularly powerful because they contain a lot of **syntactic sugar** or niceties that include:

- **Named arguments:** when the arguments are specified by name, they do not need to be specified in order
- **Partial argument matching:** arguments can be specified by using the first few unambiguous letters of their name
- **Default argument values:** most arguments have default values and can thus remain unspecified

Functions are typically placed in separate ‘.R’ files and called from an R script or the console. To make R “aware” of a new function (e.g., `myFun.R`), you must **source** the function by issuing the following command in your script (or the console) prior to calling your function: `source("myFun.R")`. To call a function with no arguments, you must append `()` to the function name (e.g., `myFun()`). Calling a function without `()` will print out its contents instead of execute its commands (e.g., `myFun`). Functions can take multiple arguments but can only return a maximum of one object (e.g., `var=myFun()`).

The code within functions has its own **lexical scope** and is thus isolated from other functions, scripts, and the command line. This means that variables that are defined within a function are not accessible outside the function. Similarly, variables defined outside the function are not accessible within the function⁴. Hence, the only way to access a variable created within a function is to have the function return it as output⁵.

Let’s demonstrate some of these features by creating a very simple function that takes two numerical arguments and returns their difference:

```
myFun <- function(firstLongArgumentName = 1, secondLongArgumentName = 1) {
  delta <- firstLongArgumentName - secondLongArgumentName
  return(delta)
}
```

The function, which takes two arguments, was stored in variable `myFun`. The body of the function consists of a single directive: compute the difference between the first and the second argument and store it in variable `delta`. This variable `delta` is returned as output. Each argument has a default value of 1 associated with it. This means that we can call function `myFun()` without specifying any of the arguments and it will return a result, namely $1 - 1 = 0$:

```
# Call the function without any arguments
myFun()

## [1] 0
```

⁴That’s not exactly true. One of the biggest issues with R is its lax rules when it comes to scope. The statement above regarding the scope of functions holds for most reasonable programming languages, but not R. Indeed, if a command within a function calls a variable that is not defined within that same function, R will happily look for that variable elsewhere, leading to potentially nasty surprises.

⁵That’s a bit of a lie. In later sections, we will introduce global assignment operators, which can be used to make variables created within functions available to the outside world.

We can naturally specify arguments to compute the difference between other values. Thankfully, we don't have to write the names of the arguments. Indeed, if we specify the arguments in the right order, the function will give the expected result:

```
# Compute 2-3=-1
myFun(2, 3)

## [1] -1
```

However, if we flip the order of the arguments and don't specify them by name, we will get an unexpected result:

```
myFun(3, 2)

## [1] 1
```

However, we don't need to worry about the order if we specify the arguments by their first few distinguishing letters:

```
myFun(s = 3, f = 2)

## [1] -1
```

Finally, if we are curious about the content of a function, we can print it out to the screen:

```
# Print out the content of myFun
myFun

## function(firstLongArgumentName = 1, secondLongArgumentName = 1) {
##   delta <- firstLongArgumentName - secondLongArgumentName
##   return(delta)
## }
```

2.12 Operators

Operators are used to perform **mathematical** calculations, **logical** comparisons, and **assignments**. Each type of operator is associated with a **precedence** level, as outlined in the table below (ordered from highest to lowest precedence):

Precedence	Operator	Description
1	::, :::	Access variables in different namespaces
2	\$, @	Component or slot extraction for objects
3	[, [[Indexing objects
4	^	Exponentiation
5	+, -	Unary minus, plus
6	:	Sequence operator
7	%any%, %in%	Special operators
8	*, /	Multiply, divide
9	+, -	Addition, Subtraction
10	<, >, <=, >=, ==, !=	Ordering and comparison
11	!	Logical negation (NOT)
12	&, &&	Logical AND
13	,	Logical OR
14	~	Formula
15	->, ->	Assignment (left to right)
16	<-, <-	Assignment (right to left)
17	=	Assignment (right to left)

If a single command contains multiple operators, the order in which they are evaluated by R will depend on their precedence. For example, mathematical calculations in R follow standard precedence rules in that exponentiations are done prior to multiplications and divisions, which are done prior to sums and subtractions:

```
# Exponentiation first, then multiplication, then sum
3 + 2 * 5^2

## [1] 53
```

To escape these rules, use parentheses to give precedence to the calculations you want performed first:

```
# Do sum first, then multiplication, then exponentiation
((3 + 2) * 5)^2

## [1] 625
```

2.12.1 Assignment operators

Assignment operators are used to store values in variables. The preferred local assignment operator is `<-`. Although `=` can also be used to perform local assignments, it can lead to unexpected errors because it has lower precedence and can be confused with the argument assignment operator:

```
mean(a = 1 + 3)

## Error in mean.default(a = 1 + 3): argument "x" is missing, with no default
```

Instead of creating a variable `a` containing the value 4, the code above generates an error because R thinks that we are trying to assign `1+3` to argument `a` of function `mean`. However, using the unambiguous `<-` operator yields the expected result:

```
mean(a <- 1 + 3)

## [1] 4
```

Global assignment operators allow variables to be shared across different environments (e.g., within functions, scripts). However, their use is generally discouraged because global assignments allow different functions to silently alter the same variable, thus making debugging and error detection much harder:

```
# Assign 2 to a
(a <- 2)

## [1] 2

# Define function that alters a across all environments
myFunGlobal <- function() {
  a <- 5
}
# Call function
myFunGlobal()
# Value of a changed unexpectedly
a

## [1] 5
```

It is much safer to create a function that returns a value that can be stored locally:

```
# Define function that returns a without changing it
# globally
myFunLocal <- function() {
  return(a = 5)
}
# Assign 2 to a
(a <- 2)

## [1] 2

# Call function and store results in b
(b <- myFunLocal())

## [1] 5

# a remains unaltered
a

## [1] 2
```

2.12.2 Mathematical operators

R has built-in operators for standard mathematical operations such as summation, division, multiplication, and exponentiation. These operations work on both simple and compound objects. For instance, two vectors of the same length can be summed, divided, or multiplied:

```
v1 <- 1:5
v2 <- 11:15
v1 * v2

## [1] 11 24 39 56 75
```

This is called **element-wise** multiplication because the corresponding elements of vectors `v1` and `v2` (i.e., the elements having the same indices in their respective vector) are multiplied. Naturally, there are analogous element-wise operations for divisions, sums, subtractions, etc...

N R will happily perform element-wise operations between vectors of different lengths as long as the length of one vector is a multiple of the other. In that case, the elements of the shorter vector will be recycled. This is very unlikely to be the desired behavior, so be very careful about performing operations on vectors of different lengths.

R also provides operators to perform **matrix algebra**. These types of operations are specified by enclosing the element-wise mathematical signs (e.g., `+`, `-`, `*`, `/`) between two `%` signs (e.g., `%*%`). For instance, matrix multiplication is carried out as follows:

```
a <- matrix(1:6, nrow = 3)
b <- matrix(21:24, nrow = 2)
a %*% b

##      [,1] [,2]
## [1,]  109  119
## [2,]  152  166
## [3,]  195  213
```

N There are special rules that govern matrix algebra. However, their description is beyond the scope of this course manual.

2.12.3 Logical operators

Logical operators are used to compare values and evaluate propositions; they return either `TRUE` or `FALSE`. We have covered a number of these operators in previous examples, so I will focus on the novel ones. Specifically, the inequality (`!=`), negation (`!`), and matching (`%in%`) operators, along with the **AND** (`&`, `&&`), **OR** (`|`, `||`) operators. The inequality operator is simple enough: it returns `TRUE` when two values are not equal and `FALSE` when they are. Like all other logical operators, it works on both individual values and complex objects such as vectors:

```
5 != 9

## [1] TRUE

1:5 != 5

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

The negation, AND, and OR operators can be used to build-up complex propositions for evaluation. The negation operator returns the opposite of the condition: it returns `TRUE` if the condition is `FALSE` and `FALSE` if the condition is `TRUE`. The AND operator returns `TRUE` if two conditions are `TRUE`, whereas the OR operator returns `TRUE` if either one of the two conditions is `TRUE`. For example, these operators can be used to select certain values within a range:

```
v <- 1:5
# Get values that fall between 2 and 4
v[v >= 2 & v <= 4]
```



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

Here, `v >= 2` returns `TRUE` for indices 2 through 5 whereas `v <= 4` returns `TRUE` for indices 1 through 4. The AND operator combines these conditions in element-wise fashion and returns `TRUE` only when the indices in both conditions are `TRUE`. The negation operator can be used to select values outside a range:

```
# Get values that do not fall between 2 and 4
v[!(v >= 2 & v <= 4)]

## [1] 1 5
```

The negation operator simply flips the `TRUE` and `FALSE` indices, and thus selects the complement (or opposite) set of values from vector `v`.

The OR operator can also be used to combine different conditions: if either condition is met, the OR operator will return `TRUE`. For example, to select only extreme values in `v` (e.g., smaller than 2 or greater than 4):

```
# Get values that are smaller than 2 or greater than 4
v[v < 2 | v > 4]

## [1] 1 5
```

As you might expect, these types of logical operations are extremely useful for subsetting data from matrices or data frames:

```
(df <- data.frame(sex = c(rep("male", 2), rep("female",
3)), height = c(68, 72, 65, 67, 66), age = c(18, 17,
15, 19, 25)))

##      sex height age
## 1  male     68  18
## 2  male     72  17
## 3 female     65  15
## 4 female     67  19
## 5 female     66  25

# Get observations from females whose height < 68 and
# age < 20
subset(df, sex == "female" & height < 68 & age < 20)

##      sex height age
## 3 female     65  15
## 4 female     67  19
```

The code above selects all females whose height < 68 and age < 20 from data frame `df`.

The final novel operator is `%in%`, which can be used to test whether a value is found in a vector:

```
# Is 5 in v?
5 %in% v

## [1] TRUE
```

N Although logical operators have clear precedence rules, it is wise to wrap conditions in parentheses in order to ensure that they are evaluated in the desired order.

2.12.4 When operators go bad

Missing or undefined values can occur naturally in datasets or via mathematical operations (e.g., division by zero). R uses a variety of ways to represent these non-finite values:

Value	Description
NA	Indicates missing values
NaN	Produced when calculation is undefined
Inf/-Inf	Produced when calculation generates non-finite value

Unfortunately, all of these non-finite values **pollute** any mathematical operation that they are involved in:

```
# One bad apple ...
(x <- c(1.6, 3.8, NA))

## [1] 1.6 3.8 NA

# ... spoils the whole bunch
mean(x)

## [1] NA
```

Instead of getting the average, the `mean` function generated `NA`. Perhaps we can get around this by selecting only the non-NA values in `x`? Let's try to do just that:

```
# Find indices of x that are not NA
x != NA

## [1] NA NA NA
```

That didn't work as expected because `NA` values pollute everything, including logical operations... As it turns out, we need special functions to deal with non-finite values such as `NA`, `NaN`, and `Inf/-Inf`:

Function	Description
<code>is.na(x)</code>	Returns TRUE if <code>x</code> is NA or NaN
<code>is.nan(x)</code>	Returns TRUE if <code>x</code> is NaN
<code>is.infinite(x)</code>	Returns TRUE if <code>x</code> is infinite
<code>is.finite(x)</code>	Returns TRUE if <code>x</code> is finite
<code>na.omit(x)</code>	Returns only non-NA values of <code>x</code>
<code>mean(x, na.rm=TRUE)</code>	Computes mean on non-NA values of <code>x</code>

Going back to our example, we can now compute the mean of the remaining non-NA values:

```
# Determine indices of NA values
(nas <- is.na(x))

## [1] FALSE FALSE TRUE

# Compute mean of non-NA values
mean(x[!nas])

## [1] 2.7
```

```
# Compute mean using the built-in na.rm argument to get
# rid of NA values
mean(x, na.rm = TRUE)

## [1] 2.7
```

Most functions that perform mathematical calculations (e.g., `mean`, `sum`) have a `na.rm` argument that can be used to deal with missing values.

2.13 Input and Output

There are many ways of loading (inputs) and saving (output) objects in R. Typically, these objects will only persist for the duration of an R session, defined as the time between launching and closing R/RStudio.

2.13.1 Managing your R session

We have seen that objects can be created in an R session by assigning them to variables. The following functions can be used to keep track of these objects and/or remove them:

Function	Description
<code>ls(pattern=glob2rx("obj"))</code>	Returns the name of <code>obj</code> if it is in the current R session and an empty character if it is not
<code>ls()</code>	Returns the name of all objects in the current R session
<code>rm(obj)</code>	Remove object <code>obj</code> from the current R session
<code>rm(list=ls())</code>	Remove all objects from the current R session

I have been creating a lot of (poorly named) objects. Let's see what's currently in my R session:

```
# List all objects in my session
ls()

## [1] "a"          "b"          "bad.char.vec" "cb"
## [5] "cb.black"   "cb.blue"    "cb.green"     "cb.grey"
## [9] "cb.orange"  "cb.pink"    "cb.red"       "cb.turquoise"
## [13] "cb.yellow"  "chars"      "df"           "df2"
## [17] "height"     "humid"      "ind"          "l"
## [21] "l2"         "long.states" "m"            "mat"
## [25] "my.sum"     "my.text"    "myFun"        "myFunGlobal"
## [29] "myFunLocal" "nas"        "new.col"      "new.row"
## [33] "nums"       "pos"        "sex"          "short.states"
## [37] "split.vec"  "temp"       "temps"        "tm"
## [41] "v"          "v1"         "v1.char"      "v1.numb"
## [45] "v2"         "v4"         "val1"         "val2"
## [49] "vals.vec1"  "vals.vec2"  "var1"         "Var1"
## [53] "vec"        "vec.height" "vec.sex"      "x"
```

Wow, that's a lot of stuff, most of which is useless at this point. Let's start deleting some of these objects:

```
# Remove object nas
rm(nas)
# Is nas still in the session?
ls(pattern = glob2rx("nas"))

## character(0)
```

We can also go thermonuclear and remove all objects in the session:

```
# Remove all objects in the session
rm(list = ls())
# Have any objects survived?
ls()

## character(0)
```

The functions above are very useful for managing objects within an R session. However, the **entire session** can be saved to or loaded from the hard drive. This is particularly useful if you are working on a project but have to switch to some other activity. You can save your entire session, reload it later, and pick-up right where you left off. The following functions will help you do just that:

Function	Description
<code>save(file="mySession.RData", obj)</code>	Save object <code>obj</code> to file <code>mySession.RData</code>
<code>save.image(file="mySession.RData")</code>	Save all objects in the current R session to file <code>mySession.RData</code>
<code>load("mySession.RData")</code>	Load objects in <code>mySession.RData</code> to the current R session

By default, these session files (i.e., `.RData`) will be saved in the current working directory. However, they can be saved anywhere by specifying the path in the filename:

```
# Create an object since this session is empty
a <- 5
# Save session file to desktop
save.image(file = "~/Desktop/mySession.RData")
# Create a new object after having saved the session
b <- 7
# Reload the session
load(file = "~/Desktop/mySession.RData")
ls()

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

A session can also be saved and loaded by using the icons in the **workspace** pane of RStudio.

N As long as the variable names in the old session are different from those in the current session, loading an old session will not affect the variables created in the new session. However, all variables that have identical names in the old and current sessions will be overwritten when the old session is loaded.

2.13.2 Navigating the filesystem

Three main functions are used to move around the filesystem and determine its contents:

Function	Description
<code>list.files(path=".", ...)</code>	List files in the current directory. Can specify a different directory using the <code>path</code> argument
<code>getwd()</code>	Get the current working directory
<code>setwd(dir)</code>	Change the working directory to <code>dir</code>

For example, to list the contents of the current directory:

```
# List files in current directory
list.files()

## [1] "chapter1.Rnw"
## [2] "chapter2.Rnw"
## [3] "chapter3.Rnw"
## [4] "chapter4.Rnw"
## [5] "commands.tex"
## [6] "ecological_dynamics-course_manual.pdf"
## [7] "ecological_dynamics-course_manual.Rnw"
## [8] "ecological_dynamics-course_manual.tex"
## [9] "figures"
## [10] "figures_manual"
## [11] "images"
## [12] "lab_manual_outline.txt"
## [13] "main-concordance.tex"
## [14] "manual-preamble.tex"
## [15] "reproducibility"
## [16] "rmarkdown-rstudio.pdf"
## [17] "rmarkdown-rstudio.png"
## [18] "scripts"
## [19] "StyleInd.ist"
```

Let's go to the desktop and see what's there:

```
# Go to desktop
setwd("~/Desktop")
# List files on desktop
list.files()

## [1] "mySession.RData"
```

We can even manipulate the filesystem directly by adding and removing files and directories via the following functions:

Function	Description
<code>file.create("fileName", ...)</code>	Create empty file called <code>fileName</code> in the current directory. Returns <code>TRUE</code> if successful and <code>FALSE</code> otherwise
<code>file.remove("fileName", ...)</code>	Remove <code>fileName</code> in the current directory. Returns <code>TRUE</code> if successful and <code>FALSE</code> otherwise
<code>dir.create("dirName", ...)</code>	Create subdirectory <code>dirName</code> in the current directory. Returns <code>TRUE</code> if successful and <code>FALSE</code> otherwise

For instance, to remove the R session file that I saved on the desktop:

```
# Remove session file
file.remove("~/Desktop/mySession.RData")

## [1] TRUE
```

2.13.3 Importing data

Data can be imported from binary files such as Excel spreadsheets using functions from external packages. However, it is much better and safer to load files in R that are encoded using standard ASCII formats such as tab delimited or comma separated values (i.e., CSV). If need be, Excel binary spreadsheets can be converted to ASCII files and then loaded into R using the following functions:

Function	Description
<code>read.xls(file=f, sheet=1)</code>	Read sheet 1 of Excel xls file <code>f</code> (requires package <code>gdata</code>). Again, highly discouraged
<code>read.table(file=f, header=FALSE, ...)</code>	Read ASCII file <code>f</code> . Set <code>header=TRUE</code> if the first row contains column labels. Type <code>?read.table</code> at the R command line to get more help
<code>read.csv(file=f, header=TRUE, ...)</code>	Read CSV file <code>f</code> . Type <code>?read.csv</code> at the R command line to get more help

Importantly, the functions outlined above can also be used to import files directly from the internet! This is an extremely powerful feature:

```
# Download a data file from the internet
d <- read.csv("http://faraway.neu.edu/data/pisco.csv")
# Get the structure of the data
str(d)

## 'data.frame': 2629 obs. of 25 variables:
## $ sitenum      : int  8 10 11 12 13 14 15 16 17 18 ...
## $ latitude     : num  45.9 44.8 44.8 44.7 44.2 ...
## $ longitude    : num  -124 -124 -124 -124 -124 ...
## $ species      : int  471 471 471 471 471 471 471 471 471 471 ...
## $ yearnum      : int  1 1 1 1 1 1 1 1 1 1 ...
## $ cover        : num  0 0 0 0 0 0 0 0 0 0 ...
## $ chla_mean    : num  5.25 4.36 4.76 4.89 5.63 ...
## $ chla_n       : int  10 8 10 8 9 11 11 12 14 14 ...
## $ chla_std     : num  1.66 1.33 1.43 1.58 2.06 ...
## $ filter_chla_mean : num  5.25 3.88 4.43 4.48 6.19 ...
## $ filter_chla_n  : int  10 7 9 7 8 10 10 12 12 12 ...
## $ filter_chla_std : num  1.659 0.362 1.08 1.221 1.413 ...
## $ sst_mean     : num  12.7 12.1 12.1 12 12.1 ...
## $ sst_n        : int  41 39 43 42 38 38 37 44 38 38 ...
## $ sst_std      : num  0.431 0.578 0.604 0.534 0.363 ...
## $ filter_sst_mean : num  12.8 11.9 11.9 11.8 12.1 ...
## $ filter_sst_n   : int  35 33 37 36 35 35 34 39 34 34 ...
## $ filter_sst_std : num  0.236 0.381 0.424 0.321 0.226 ...
## $ upindex_mean  : num  -35.7 -40.2 -40.2 -40.2 -39.9 ...
## $ upindex_n     : int  1 2 2 2 2 2 2 1 1 1 ...
## $ upindex_std   : num  0 4.46 4.46 4.46 4.75 ...
## $ filter_upindex_mean : num  -35.7 -40.2 -40.2 -40.2 -39.9 ...
## $ filter_upindex_n : int  1 2 2 2 2 2 2 1 1 1 ...
```

```
## $ filter_upindex_std : num  0 4.46 4.46 4.46 4.75 ...
## $ speciesname       : Factor w/ 11 levels "B. californicus",...: 1 1 1 1 1 1 1 1 1 ...
```

The commands `read.csv` and `read.table` make local copies of the file and load it into R. Hence, once those commands have been issued, an internet connection is no longer necessary to work on the data.

2.13.4 Exporting data

Data tables or model results can be written to the hard drive using the following functions:

Function	Description
<code>write.table(x, file=f, ...)</code>	Write object <code>x</code> to file <code>f</code> using <code>tab</code> ASCII format
<code>write.csv(x, file=f, ...)</code>	Write object <code>x</code> to file <code>f</code> using <code>CSV</code> ASCII format

N These functions can only write data of class `data.frame` or `matrix`

2.13.5 Packages

The 4,000+ packages created by the community greatly extend the base functionality of R. These packages typically include functions and datasets, and can be accessed via the following functions:

Function	Description
<code>install.packages("pkg")</code>	Install package <code>pkg</code> from R's central repository (CRAN). Each package needs to be installed only once.
<code>require(pkg)</code> , <code>library(pkg)</code>	Either command will load package <code>pkg</code> into the current R session.
<code>detach(package:pkg, unload=TRUE)</code>	Unload package <code>pkg</code> from the current R session.
<code>data(package="biwavelet", enviro.data)</code>	Load dataset <code>enviro.data</code> from package <code>biwavelet</code> . Must first install the package in order to access its datasets.

N There are many more R packages to conduct biological analyses available on [bioconductor](#).

2.14 Data wrangling

The first step of any analysis consists of manipulating the dataset (not the data!) so that it is in the proper format. This typically requires slicing the data in order to select only the variables of interest using the previously described `subset` function, **merging** distinct datasets together, and **reshaping** the final dataset. The following table summarizes the key functions to accomplish these tasks:

Function	Description
<code>subset(data, subset, select)</code>	Subset data by selecting only the rows that meet the criterion specified in the subset argument and the columns specified in the select argument.
<code>merge(x, y, by='z', all=FALSE, ...)</code>	Merge datasets x and y using common column z as a key.
<code>reshape(data, ...)</code>	Reshape dataset data from the long to the wide format or vice versa.

We now describe these functions in detail.

2.14.1 Combining datasets

R essentially replicates the functionality of a full database via function `merge`, making it easy to combine datasets using a common **key**. The key serves a critical purpose: it tells R how the rows or records of dataset **x** should be matched-up with those of dataset **y**. The `merge` function can combine two datasets **x** and **y** in several ways:

- **Natural join:** Combine **x** and **y** based on their common entries. This means that any entry that is found in either **x** or **y** but not both will be discarded. In set theory, this is called the **intersection** of the sets. In R, natural joins are performed by using `merge(x, y, by=c("z"))`, with **z** being the key.
- **Left join:** Combine **x** and **y** based on the entries found in **x** or both **x** and **y**. This means that any entry that is found in **y** only will be discarded. In set theory, this is called the **partial union** of both sets. In R, left joins are performed by using `merge(x, y, by=c("z"), all.x=TRUE)`, with **z** being the key.
- **Right join:** Combine **x** and **y** based on the entries found in **y** or both **x** and **y**. This means that any entry that is found in **x** only will be discarded. In set theory, this is called the **partial union** of both sets. In R, right joins are performed by using `merge(x, y, by=c("z"), all.y=TRUE)`, with **z** being the key.

Let's use the `merge` function to combine biological and environmental datasets collected in intertidal ecosystems along the West Coast of the US:

```
# Load biological data
bio <- read.csv("http://faraway.neu.edu/data/pisco_bio.csv")
# Take a look at the beginning of the dataset
head(bio)

##   sitenum latitude longitude year      species cover
## 1      8 45.93000 -123.9867 1999 B. californicus    0
## 2     10 44.82667 -124.0567 1999 B. californicus    0
## 3     11 44.80667 -124.0600 1999 B. californicus    0
## 4     12 44.74667 -124.0600 1999 B. californicus    0
## 5     13 44.24250 -124.1100 1999 B. californicus    0
## 6     14 44.21833 -124.1100 1999 B. californicus    0

# Load environmental data
env <- read.csv("http://faraway.neu.edu/data/pisco_env.csv")
# Take a look at the beginning of the dataset
head(env)

##   sitenum latitude longitude year chla_mean sst_mean upwelling_mean
## 1      18 43.30000 -124.4000 1999  4.404954 12.03498   -35.13245
## 2      17 43.31000 -124.3933 1999  4.404954 12.03498   -35.13245
```

```
## 3      16 43.33000 -124.3740 1999 3.032030 12.06844      -35.13245
## 4      19 41.06000 -124.1500 1999 3.938939 12.00153       11.02540
## 5      15 44.20000 -124.1100 1999 5.482347 12.14849      -39.88423
## 6      14 44.21833 -124.1100 1999 5.482347 12.12739      -39.88423
```

The biological dataset contains the annual mean abundance (column `cover`) of a number of sessile species (column `species`) collected from 1999-2003 (column `year`) at 49 different sites (column `sitenum`). The environmental dataset contains mean annual sea surface temperature (column `sst_mean`), chlorophyll-a concentration (column `chla_mean`), and upwelling current (column `upwelling_mean`) at these same sites and over the same time period. To understand how the biological patterns relate to the environmental conditions, we will need to merge these two datasets. However, we need to do this correctly: we want the biological and environmental data collected at the same location (`sitenum`) and time (`yearnum`) to appear on the same row in the combined dataset. In this case, `sitenum` and `year` are the keys. Additionally, since `sitenum` is associated with a unique latitude and longitude, we will add these columns as keys:

```
# Merge bio and env based on common sitenum, latitude,
# longitude, and year
bio.env <- merge(bio, env, by = c("sitenum", "latitude",
                                "longitude", "year"))
# Take a look at the result
head(bio.env)
```

```
##   sitenum latitude longitude year      species  cover chla_mean
## 1      1  48.38917  -124.65 2000    P. polymerus  4.8667  3.460111
## 2      1  48.38917  -124.65 2000    B. glandula  1.4000  3.460111
## 3      1  48.38917  -124.65 2000 M. californianus 59.7667  3.460111
## 4      1  48.38917  -124.65 2000    M. trossulus  0.0000  3.460111
## 5      1  48.38917  -124.65 2000 Chthamalus spp.  0.9333  3.460111
## 6      1  48.38917  -124.65 2000    B. nubilus   0.0000  3.460111
##   sst_mean upwelling_mean
## 1  9.555175      -57.68661
## 2  9.555175      -57.68661
## 3  9.555175      -57.68661
## 4  9.555175      -57.68661
## 5  9.555175      -57.68661
## 6  9.555175      -57.68661
```

Now that our new dataset contains both biological and environmental data, we are ready to perform our analysis.

2.14.2 Reshaping datasets

Generally speaking, datasets come in one of two formats: **long** or **wide**. In the long format, columns are used to store data of the same type. Here's an example of the long format:

sitenum	latitude	longitude	year	cover
1	42.5	-126.8	2001	0.3
...
2	42.5	-132.8	2012	0.4

In the long format above, the values of each variable are aggregated in a single column. For instance, there is a `year` column that holds all the year entries or values. Similarly,

there is a `latitude` column that holds all the latitude entries or values. Although this format is intuitive, it is not very economical because information is duplicated. For example, for each `cover` value and `year`, the same `latitude`, `longitude`, and `sitenum` information is repeated. The wide format cuts down on duplication by spreading values from a single column across multiple columns:

sitenum	latitude	longitude	2001	2002	...	2012
1	42.5	-126.8	0.3	0.5	...	0.6
...
2	42.5	-132.8	0.1	0.8	...	0.4

Here, for instance, the `cover` values are spread across multiple columns, each corresponding to a different `year`. Now, the `latitude`, `longitude`, and `sitenum` values are not repeated for each `cover` value, and the `year` column has been removed. Although more economical, this data format makes it difficult to conduct statistical analyses. Hence, it is important to be able to seamlessly shift between long and wide formats. The function `reshape` allows us to do just that. Let's begin by downloading and subsetting a dataset:

```
# Download a dataset
d <- read.csv("http://faraway.neu.edu/data/pisco_bio.csv")
# Subset it by selecting only the relevant columns
s <- subset(d, subset = species == "M. californianus", select = c("sitenum",
  "latitude", "longitude", "year", "cover"))
# Look at the dataset
head(s)

##      sitenum latitude longitude year  cover
## 2152      8 45.93000 -123.9867 1999 85.0000
## 2153     10 44.82667 -124.0567 1999 69.7333
## 2154     11 44.80667 -124.0600 1999 78.0000
## 2155     12 44.74667 -124.0600 1999 75.3333
## 2156     13 44.24250 -124.1100 1999 60.0667
## 2157     14 44.21833 -124.1100 1999 68.0000

# Size of the dataset
dim(s)

## [1] 239  5
```

Here, we have selected the cover of the dominant mussel *M. californianus*. The `head` command shows that the dataset is in the long format. We now switch it to the wide format:

```
# Convert from long to wide format
wide <- reshape(s, timevar = "year", idvar = c("sitenum",
  "latitude", "longitude"), direction = "wide")
# Take a look at the end of the dataset
tail(wide)

##      sitenum latitude longitude cover.1999 cover.2000 cover.2001
## 2210      45 33.44000 -118.4767         NA      2.8000      1.1034
## 2211      46 32.84000 -117.2800         NA     10.6000      6.1667
## 2212      47 32.82000 -117.2767         NA     46.7000     31.3667
## 2213      48 32.71167 -117.2500         NA      1.0667      0.7000
## 2214      50 45.91333 -123.9767         NA     38.8000     60.2333
## 2215      51 45.75500 -123.9650         NA      8.8333     32.8333
```

```
##      cover.2002 cover.2003 cover.2004
## 2210      0.7333      0.5333        NA
## 2211      4.4667      6.7000        NA
## 2212     35.0667     52.2000        NA
## 2213      1.5333      0.5667        NA
## 2214     32.7407     13.3667     19.6667
## 2215      3.4828     10.2667      3.7333

# Size of the dataset
dim(wide)

## [1] 49  9
```

The `reshape` function converted the dataset to the wide format by spreading the `cover` values across multiple `year` columns. One new column was created for each unique year value found in the `year` column of the long format dataset. As a result, the values in columns `sitenum`, `latitude`, and `longitude` are not replicated anymore. NA values were produced for sites and years where no cover was collected. Overall, to convert a dataset from the long to the wide format using function `reshape`, one has to (1) use the `timevar` argument to specify the column in the long format that should be split based on its unique values to create multiple new columns each containing the desired variable (e.g., `cover`) in the new wide format, (2) identify the columns that should be simplified or whose values should be de-duplicated via the `idvar` argument, and (3) specify `direction=wide`.

Naturally, we can convert a dataset from wide to long format. Let's try it with the previously created dataset `wide`:

```
# Convert from wide to long
long <- reshape(wide, timevar = "year", varying = list(4:9),
  idvar = c("sitenum", "latitude", "longitude"), v.names = "cover",
  direction = "long")
# Replace long rownames with row number
rownames(long) <- 1:NROW(long)
# See a snippet of the data
head(long)

##   sitenum latitude longitude year  cover
## 1      8 45.93000 -123.9867    1 85.0000
## 2     10 44.82667 -124.0567    1 69.7333
## 3     11 44.80667 -124.0600    1 78.0000
## 4     12 44.74667 -124.0600    1 75.3333
## 5     13 44.24250 -124.1100    1 60.0667
## 6     14 44.21833 -124.1100    1 68.0000
```

In the code above, we used (1) the `varying=list(4:9)` argument to indicate which columns in the wide format should be combined into a single column, (2) the `v.names="cover"` argument to indicate what the name of that new column should be, and (3) the `direction="long"` argument to indicate that we wanted to convert the dataset to the long format.



There is a package called `reshape2` that provides slightly more intuitive functions to perform the same dataset conversions.

2.14.3 Computing statistics

It is often useful to compute the statistical properties of different subsets of the data. This can be achieved by using the `aggregate` function, which can be invoked in two

different ways:

Function	Description
<code>aggregate(x, by, FUN, ...)</code>	Apply function FUN to x based on the unique values specified in by.
<code>aggregate(formula, data, FUN, ..., subset)</code>	Apply function FUN to x based on the unique values specified in formula.

Here is how to use the `aggregate` function to compute the mean cover of mussels at each site:

```
# Download the dataset
d <- read.csv("http://faraway.neu.edu/data/pisco_bio.csv")
# Subset the data
s <- subset(d, subset = species == "M. californianus", select = c("latitude",
  "longitude", "year", "cover"))
# Compute the mean cover for each site unique latitude
# and longitude (i.e., site)
a <- aggregate(d$cover, by = list(lat = d$latitude, lon = d$longitude),
  FUN = mean, na.rm = TRUE)
head(a)

##      lat      lon      x
## 1 48.33750 -124.6875 5.460602
## 2 48.31333 -124.6600 2.889309
## 3 47.94000 -124.6583 2.133935
## 4 48.38917 -124.6500 6.814880
## 5 47.87000 -124.6000 4.330909
## 6 47.86000 -124.5700 3.742598
```

In this case, for each unique site, `aggregate` averaged the cover values obtained in different years. We can also invoke `aggregate` using the formula operator:

```
# Compute the mean cover for each site unique latitude
# and longitude (i.e., site)
a <- aggregate(cover ~ latitude + longitude, data = d, FUN = mean,
  na.rm = TRUE)
head(a)

##  latitude longitude      cover
## 1 48.33750 -124.6875 5.460602
## 2 48.31333 -124.6600 2.889309
## 3 47.94000 -124.6583 2.133935
## 4 48.38917 -124.6500 6.814880
## 5 47.87000 -124.6000 4.330909
## 6 47.86000 -124.5700 3.742598
```

The first argument to `aggregate` is called a **formula** and consists of two text entries separated by the special operator `~`. The text entry to the left of the `~` operator corresponds to the variable of interest (often called the response variable), and the text to the right of the `~` operator specifies the explanatory variables. In this case, the formula states that we want to apply the function FUN to the response variable `cover` based on the unique values of the explanatory variables `latitude` and `longitude`.

N The formula operator `~` is one of the most powerful and ubiquitous features of R. It is used when manipulating, plotting, and fitting models to datasets.

We can also use `aggregate` to compute our own custom functions on subsets of the data. For example, here is how to compute the standard error, defined as $\frac{SD(X)}{\sqrt{n}}$, where $SD(X)$ corresponds to the standard deviation of vector X and n is its length:

```
# Function to compute standard error
stderr <- function(x) {
  sd(x, na.rm = TRUE)/sqrt(length(na.omit(x)))
}
```

Notice that we made sure to deal with potential NA contamination. Now, let's apply this function using `aggregate` to determine the standard error of mussel cover at each site:

```
# Compute standard error of the mean for each location
a <- aggregate(cover ~ latitude + longitude, data = d, FUN = stderr)
# Get snippet of data
head(a)

##   latitude longitude    cover
## 1 48.33750 -124.6875 1.4677080
## 2 48.31333 -124.6600 0.6431765
## 3 47.94000 -124.6583 0.5170729
## 4 48.38917 -124.6500 2.4446882
## 5 47.87000 -124.6000 1.0080200
## 6 47.86000 -124.5700 1.3878912
```

We can also define an **anonymous** function within `aggregate` and perform the same calculation in a single step:

```
# Compute standard error of the mean for each location
a <- aggregate(cover ~ latitude + longitude, data = d, FUN = function(x) {
  sd(x, na.rm = TRUE)/sqrt(length(na.omit(x)))
})
# Get snippet of data
head(a)

##   latitude longitude    cover
## 1 48.33750 -124.6875 1.4677080
## 2 48.31333 -124.6600 0.6431765
## 3 47.94000 -124.6583 0.5170729
## 4 48.38917 -124.6500 2.4446882
## 5 47.87000 -124.6000 1.0080200
## 6 47.86000 -124.5700 1.3878912
```



Anonymous functions are useful for one-time operations because they cannot be called beyond their very limited scope (thus the name anonymous).

The `aggregate` function is not the only game in town when it comes to performing statistical calculations on datasets. Indeed, there is a whole family of ***apply** functions dedicated to that purpose, one for each class of object:

Function	Description
<code>apply(x, MARGIN, FUN, ...)</code>	Apply function FUN to matrix x along MARGIN (MARGIN=1 means rows, MARGIN=2 means columns).
<code>lapply(x, FUN, ...)</code>	Apply function FUN to list x.
<code>mapply(FUN, ...)</code>	Apply function FUN to multiple lists or vectors.
<code>sapply(FUN, ...)</code>	Apply function FUN to vector x and simplify results.
<code>tapply(x, INDEX, FUN, ...)</code>	Apply function FUN to array or vector x along INDEX.

For example, to compute the mean mussel cover for each year, we can use the `apply` function:

```
# Download the data
d <- read.csv("http://faraway.neu.edu/data/pisco_bio.csv")
# Subset only M. californianus
s <- subset(d, subset = species == "M. californianus", select = c("sitenum",
  "latitude", "longitude", "year", "cover"))
# Convert to wide format
s.wide <- reshape(s, timevar = c("year"), idvar = c("sitenum",
  "latitude", "longitude"), direction = "wide")
# Extract only mussel cover
cover <- s.wide[, 4:NCOL(s.wide)]
# Rows = different sites, Columns = different years
head(cover)

##      cover.1999 cover.2000 cover.2001 cover.2002 cover.2003 cover.2004
## 2152    85.0000        NA        NA        NA        NA        NA
## 2153    69.7333    65.1667    47.6667    61.8621    70.5056    80.6889
## 2154    78.0000    76.4667    34.8000    70.1333    65.5667    73.3111
## 2155    75.3333    66.0667    66.3000    59.2000    78.0112    82.1395
## 2156    60.0667    37.1333    26.9000    16.8000    25.1818    49.4778
## 2157    68.0000    81.3000    34.8333    76.1333    56.6742    67.5667

# Mean mussel cover by year
apply(cover, MARGIN = 2, mean, na.rm = T)

## cover.1999 cover.2000 cover.2001 cover.2002 cover.2003 cover.2004
## 47.02918   30.44768   27.37799   25.78234   25.55849   32.91539

# Mean mussel cover by site
apply(cover, MARGIN = 1, mean, na.rm = T)

##      2152      2153      2154      2155      2156      2157      2158
## 85.000000 65.937217 66.379633 71.175117 35.926600 64.084583 41.566300
##      2159      2160      2161      2162      2163      2164      2165
## 18.522233 25.355167 54.766050 19.794833 26.230833 35.866683 6.517833
##      2166      2167      2168      2169      2170      2171      2172
## 42.072200 46.195367 63.279100 35.715620 3.491420 1.453320 0.813320
##      2173      2174      2190      2191      2192      2193      2194
## 0.133360 4.969540 39.406680 32.633340 46.258100 22.139980 11.940000
##      2195      2196      2197      2198      2199      2200      2201
## 19.913320 4.834720 19.224150 4.645400 46.791650 29.797725 47.185000
##      2202      2203      2204      2205      2206      2207      2208
## 35.695975 29.010900 43.346550 35.041675 18.050025 25.925000 0.283325
##      2209      2210      2211      2212      2213      2214      2215
## 1.083325 1.292500 6.983350 41.333350 0.966675 32.961480 11.829880
```

Using the `apply` function in this case was not very efficient because we first had to convert the dataset from long to wide format. However, we can use the `tapply` function

on the original long format dataset to obtain the same result:

```
# Download the data
d <- read.csv("http://faraway.neu.edu/data/pisco_bio.csv")
# Subset only M. californianus
s <- subset(d, subset = species == "M. californianus", select = c("sitenum",
  "latitude", "longitude", "year", "cover"))
# Mean mussel cover by year
tapply(s$cover, s$year, mean, na.rm = T)

##      1999      2000      2001      2002      2003      2004
## 47.02918 30.44768 27.37799 25.78234 25.55849 32.91539

# Mean mussel cover by site
tapply(s$cover, s$sitenum, mean, na.rm = T)

##          1          2          3          4          5          6          8
## 63.279100 35.715620 3.491420 1.453320 0.813320 0.133360 85.000000
##          9         10         11         12         13         14         15
## 4.969540 65.937217 66.379633 71.175117 35.926600 64.084583 41.566300
##         16         17         18         19         20         21         22
## 18.522233 25.355167 54.766050 19.794833 26.230833 35.866683 6.517833
##         23         24         25         26         27         28         29
## 42.072200 46.195367 39.406680 32.633340 46.258100 22.139980 11.940000
##         30         31         32         33         34         35         36
## 19.913320 4.834720 19.224150 4.645400 46.791650 29.797725 47.185000
##         37         38         39         40         41         42         43
## 35.695975 29.010900 43.346550 35.041675 18.050025 25.925000 0.283325
##         44         45         46         47         48         50         51
## 1.083325 1.292500 6.983350 41.333350 0.966675 32.961480 11.829880
```

N In this case, the functions `colMeans` and `rowMeans` could have been used to compute the mean mussel cover for each year and site more rapidly.

2.15 Plotting

It is **always important to plot the raw data first**, before fitting models and performing statistical analyses. Thankfully, plotting is one of R's strengths. As a matter of fact, there are (at least) three alternative plotting systems in R:

Package	Description
Base	Plotting system included in default installations of R. Allows you to customize every aspect of a plot
<code>lattice</code>	Alternative plotting system that started as an external package but is now included in default installations of R. It is especially useful for plotting complex figures with multiple panels
<code>ggplot2</code>	Alternative but very popular plotting system based on the Grammar of Graphics (Wilkinson, 2005). A thorough introduction to <code>ggplot2</code> 's implementation of the grammar of graphics can be found in Wickham (2009)

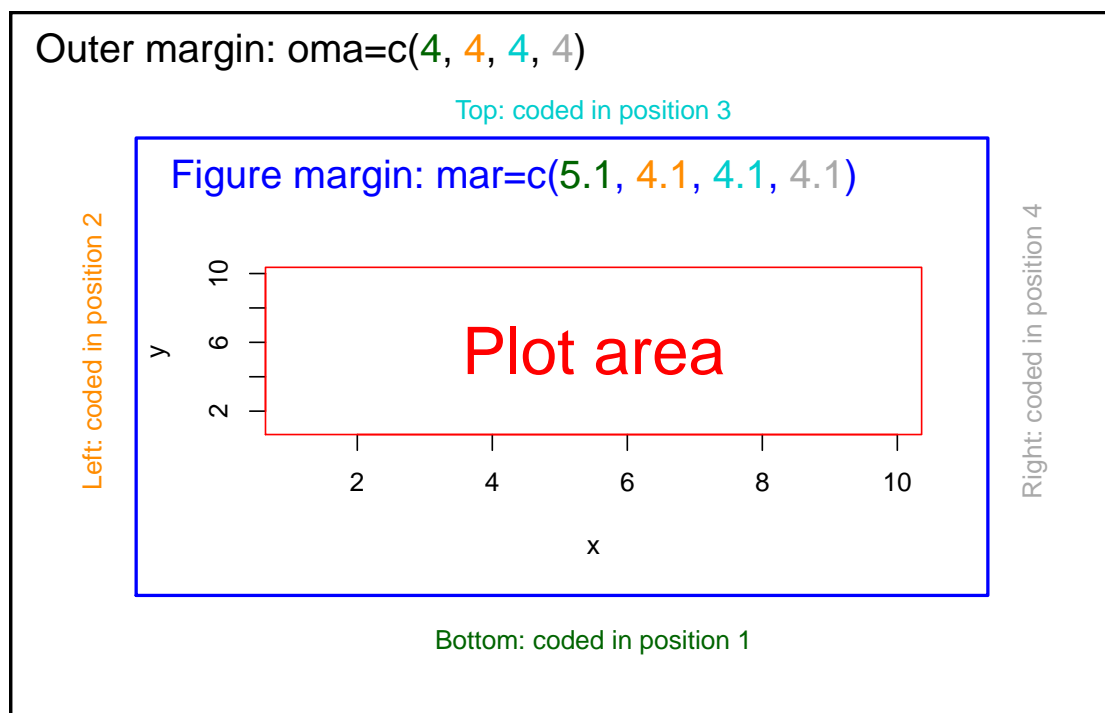
There are also many packages that can be used to generate special graphics such as 3-D plots:

Package	Description
<code>scatterplot3d</code> , <code>rgl</code> , <code>fields</code>	3-D plotting (e.g., surface, mesh, scatter)
<code>map</code> , <code>mapdata</code>	Plot maps using various projection systems

I personally use the **base** plotting system almost exclusively because it provides finer control over the aesthetics of the figures. I use **lattice** for plotting hierarchical datasets before fitting mixed-effects models and generally stay away from **ggplot2**. Below, I cover the plotting functionality in the **base** package and only provide snippets of code to produce similar results using the **lattice** and **ggplot2** packages.

2.15.1 Base system

Plots in the base system are incredibly flexible, but the price of that flexibility is complexity. Indeed, there are dozens of parameters that can be tweaked to control the way a figure looks. These parameters can be accessed or changed via the function `par` (to get more information type `?par` at the console). Overall, all plots have the following elements:



The plot above consists of a central **plot area** (depicted in red), axes around that area, a set of **figure margins** (depicted in blue), and a set of **outer margins** (depicted in black). The different colored numbers (green, orange, cyan, grey) in the figure correspond to the sizes of the inner and outer margins. The margins on each side of a figure can be altered by specifying the values using `par(mar=c(bottom, left, top, right))` for the figure margins and `par(oma=c(bottom, left, top, right))` for the outer margins. The default values for these margins are indicated in the figure and correspond to numbers of lines (specifying a fractional number of lines is fine). Note that by default, the bottom margin is specified first, followed by the left margin, top margin, and right margin. In general, one can determine the properties of a plot by issuing the following commands:

Command	Description
<code>par()\$oma</code>	Outer margin size (in number of lines). Default is <code>c(0, 0, 0, 0)</code>
<code>par()\$mar</code>	Figure margin size (in number of lines). Default is <code>c(5.1, 4.1, 4.1, 2.1)</code>
<code>par()\$mgp</code>	Distance between labels and axis. Default is <code>c(3, 1, 0)</code>
<code>par()\$pty</code>	Plot region type (<code>m</code> ="maximum size" and <code>s</code> ="square")
<code>par()\$cex</code>	Magnification value for symbols and text. Default is 1

To modify these values, one can use the function `par` before calling a plotting function:

Command	Description
<code>par(oma=c(0, 1, 0, 1))</code>	Set a margin consisting of 1 line around the left and right of the plot
<code>par(pty="s", mar=c(0, 0, 0, 0))</code>	Set a margin consisting 0 of lines around figure; square plot region
<code>par(mgp=c(1.8, 0.2, 0), cex=1.5)</code>	Set a distance of 1.8 lines between axis and axis title, 0.2 lines between axis ticks and labels; magnification of 1.5x

Here are a few additional parameters that you may want to alter:

Command	Description
<code>par(tck=0.02)</code>	Positive values indicate “internal” tick marks
<code>par(las=1)</code>	Horizontal tick labels for the x- and y-axes
<code>par(mfrow=c(3,2))</code>	Create a multi-panel figure by splitting the plot into 3-rows x 2-columns. The plots will fill the figure from left to right then top to bottom (i.e., by rows)
<code>par(mfcol=c(3,2))</code>	Same as above except that the plots will fill the figure from top to bottom first then left to right (i.e., by columns)

2.15.2 Types of plots

R can produce almost any type of plot. Below, I focus on those most commonly used plot types in statistics:

Function	Description
<code>plot(x, y, ...)</code>	Basic function used to plot lines or points
<code>points(x, ...)</code>	Add points to an existing plot
<code>lines(x, ...)</code>	Add lines to an existing plot
<code>hist(x, ...)</code>	Histograms showing the distribution of the data
<code>barplot(x, ...)</code>	Bar plot, often used to compare means across treatments in experiments
<code>boxplot(x, ...)</code>	Box plots present the min, max, median, 1st and 3rd quartile, outliers in a dataset
<code>pairs(x, ...)</code>	Plot all pairs of columns in matrix or data frame <code>x</code> against each other
<code>matplot(x, ...)</code>	Plot all columns of matrix <code>x</code>
<code>abline(...)</code>	Plot straight line (e.g., horizontal or vertical)
<code>arrows(...)</code>	Plot arrows but also error bars
<code>legend(x, ...)</code>	Add legend
<code>axis(side=1, at=x, ...)</code>	Add axis with tickmarks at locations specified by vector <code>x</code> to side 1 of plot, with 1=bottom, 2=left, 3=top, 4=right

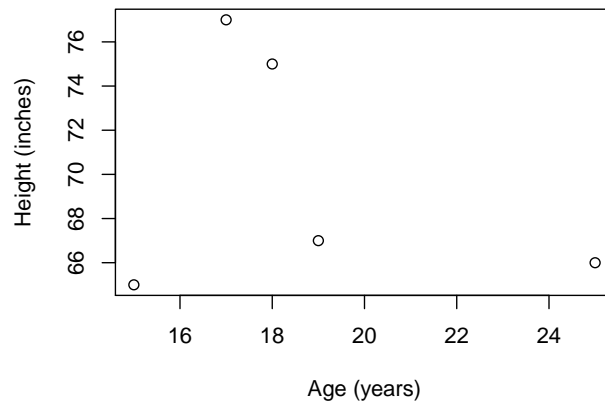
These plots can be modified by replacing the ... with some of the following arguments:

Argument	Description
<code>t="n"</code>	This argument can only be specified for function <code>plot</code> to create a plot without plotting anything
<code>t="p"</code>	This argument can only be specified for function <code>plot</code> to plot the data using points
<code>t="l"</code>	This argument can only be specified for function <code>plot</code> to plot the data using lines
<code>t="b"</code>	This argument can only be specified for function <code>plot</code> to plot the data using both lines and points
<code>lty=1</code>	Type of line to be plotted: 1=solid, 2 or higher for broken lines
<code>lwd=1</code>	Width of the line
<code>xlim=c(minVal, maxVal), ylim=c(minVal, maxVal)</code>	Range of the x- and y-axes, respectively
<code>xaxt="n", yaxt="n"</code>	Do not plot the x- or y-axes, respectively
<code>col="red"</code>	Color of points/lines
<code>xlab="X Text"</code>	Label of x-axis
<code>ylab="Y Text"</code>	Label of y-axis
<code>main="Title"</code>	Title of figure
<code>sub="Subtitle"</code>	Subtitle of figure

These arguments represent just the tip of the iceberg. To get a full description of all arguments for each plot type, use the appropriate command at the console (e.g., `?plot`). Below, I provide examples of each plot type:

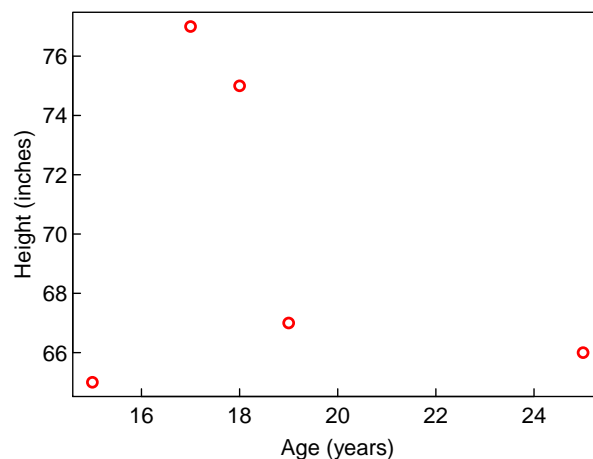
Line plots

```
# Generate a dataset
df <- data.frame(sex = c(rep("male", 2), rep("female", 3)),
  height = c(75, 77, 65, 67, 66), age = c(18, 17, 15,
    19, 25))
# Create scatterplot of height as a function of age
plot(df$age, df$height, ylab = "Height (inches)", xlab = "Age (years)")
```



The plot looks OK, but let's try to alter it a bit by playing with the `par` function:

```
# Use internal tickmarks, and change the margins
par(tck = 0.02, las = 1, mgp = c(1.4, 0.2, 0), mar = c(4.5,
  4, 1, 1))
# Create scatterplot of height as a function of age
plot(df$age, df$height, ylab = "Height (inches)", xlab = "Age (years)",
  lwd = 2, col = "red")
```



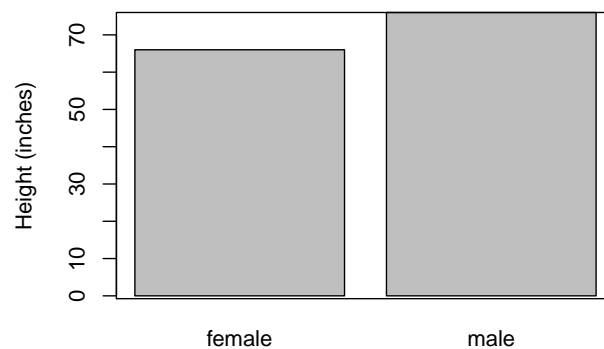
Bar plots

We can use a bar plot to represent summary statistics of the data. For instance, to plot the mean height of males and females:

```
# Compute mean height for each sex
(a <- aggregate(height ~ sex, data = df, FUN = mean))
```

```
##      sex height
## 1 female     66
## 2  male     76

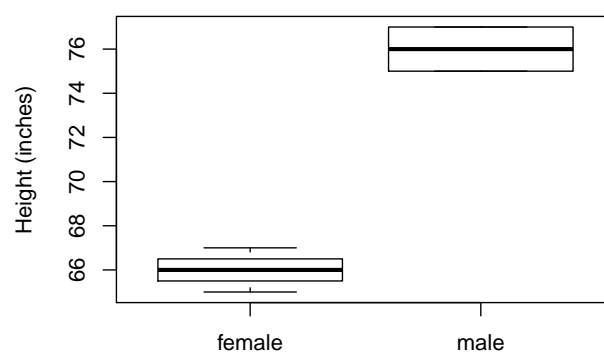
# Plot mean height for each sex
barplot(a$height, names.arg = a$sex, ylab = "Height (inches)")
# Add box around plot
box()
```



Boxplots

Boxplots provide more information about the distribution of the data by plotting the min, max, median, first quartile and third quartile:

```
boxplot(height ~ sex, data = df, ylab = "Height (inches)")
```

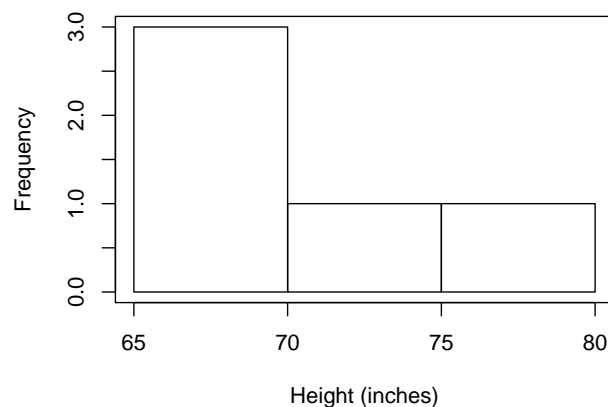


The median is represented by the horizontal black bar, the whiskers (i.e., the shorter horizontal bars that are above and below each box) represent the min and the max, and the top of the box represents the third quartile, whereas the bottom of the box represents the first quartile. Here, we can see that the range of heights is larger in females than in males (larger space between min and max). Additionally, the median height of males is much larger than that of females.

Histograms

Histograms are used to plot the raw distribution of the data:

```
hist(df$height, ylab = "Frequency", xlab = "Height (inches)",
     main = "")
# Add box around plot
box()
```



The histogram of this very small dataset shows that there is an overabundance of short heights (leftmost bar indicating the frequency of individuals with heights <70 inches). In fact, a cursory look at the data reveals that the leftmost bar represents the females and that the two other bars represent the two males.

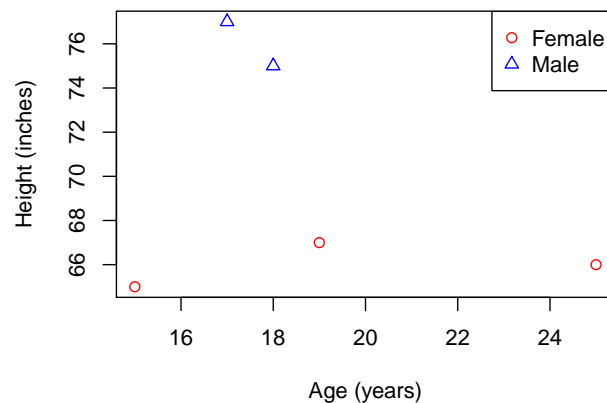
2.15.3 Customizing plots

Plots in R can be customized extensively, especially when created with the base system. These customizations can be broadly categorized as (1) plotting multiple datasets via layering, (2) adding annotations such as text to existing plots, (3) plotting datasets of varying ranges using multiple axes, and (4) plotting multiple panels per figure.

Layering

To represent multiple datasets on a single figure, one can use `plot` to initiate a figure, and then plot subsequent data using the `lines` or `points` commands. For instance, we can use this approach to plot the males and females using different colors and symbols:

```
# Plot female heights in red circles
plot(subset(df, sex == "female", select = c("age", "height")),
     col = "red", t = "p", xlab = "Age (years)", ylab = "Height (inches)",
     ylim = range(df$height), xlim = range(df$age), pch = 1)
# Plot males heights in blue triangles
points(subset(df, sex == "male", select = c("age", "height")),
       col = "blue", pch = 2)
# Add legend in top right corner
legend(x = "topright", pch = c(1, 2), col = c("red", "blue"),
       legend = c("Female", "Male"))
```

There are a few important things worth highlighting. First, the `range` function used to specify the `xlim` and `ylim` arguments is critical in layered figures. This function takes multiple vectors as input and returns the minimum and maximum value. In this case, I applied `range` to the entire dataset to make sure that the range of the plot would be adequate for both males and females. Indeed, a frequent source of frustration in layered figures is that R will set the range of the plot based on the first dataset being plotted. In this case, since male heights are about 10 inches greater than female heights, the former would have been out of range (i.e., not appeared in the figure) had I not specified the range manually. The second unique aspect of layered figures is that they must include a legend to distinguish the different sets of lines and/or points. This is done via the intuitively named `legend` function.

Adding text

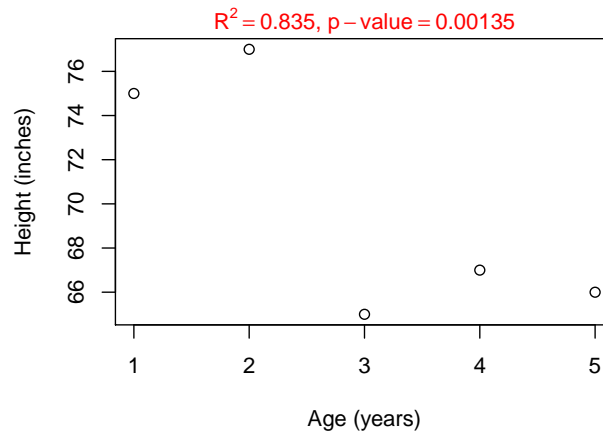
Annotations can often be useful to highlight certain important features of a plot or adding statistical details about fitted models (e.g., p-values and R^2) to a figure. Such annotations can be added with the following functions:

Function	Description
<code>mtext(side=1, text, ...)</code>	Add text to the side of a plot, with 1=bottom, 2=left, 3=top, 4=right
<code>text(x, y, text, ...)</code>	Add text to the location specified by the x, y coordinates
<code>paste("text1", val, "tex2")</code>	Utility function used to combine or concatenate text and numerical values
<code>expression(...)</code>	Used to display mathematical symbols (e.g., super- or sub-scripts)
<code>substitute(...)</code>	Replace variable name with value
<code>format(x, digits=3, ...)</code>	Print up to <code>digits=3</code> significant digits of x

Let's see a few examples of annotations using our trusty height dataset:

```
# Properties of a model fitted to the data
rsq <- 0.834543534534
pval <- 0.0013454353453
plot(df$height, ylab = "Height (inches)", xlab = "Age (years)",
     main = "")
# Add properties of a model fit to the top of the figure
```

```
# Replace text 'rsq' and 'pval' with their values
mtext(side = 3, substitute(paste(R^2 == rsq, ", ", p - value ==
  pval), list(rsq = format(rsq, digit = 3), pval = format(pval,
  digit = 3))), col = "red")
```



In the above figure, we added details about a statistical model that was fit to the data using the `mtext` function. This is a common task when displaying statistical results, but it requires a bit of code contortion. Indeed, the statistical details of models are represented using many (>10) significant digits. To trim these down to something more reasonable, we use function `format`. Additionally, we have to combine the `substitute` and the `paste` functions to concatenate these values with their text labels because the labels contain mathematical symbols such as \wedge that `paste` cannot handle by itself.

Multiple axes

Demonstrating associations between several variables or datasets sometimes requires plotting data with different units or ranges using multiple axes. For example, to visualize the relationship between mussel cover and sea surface temperature on the same plot, we would have to use multiple axes⁶ because the range of mussel cover is 0 to 100% whereas that of sea surface temperature is about 10-16°C. In this real-world example, we are going to have to deal with several difficulties: (1) axis labels contain special characters (e.g., °C for sea surface temperature) and (2) we need each y-axis to have the same color as the data that is being plotted on it. This will require a bit of work, so I broke down the steps required to attain the desired result. First, we begin by downloading the data and subsetting it:

```
# Download the data
d <- read.csv("http://faraway.neu.edu/data/pisco.csv")
# Select latitude, mussel cover and sst for year 2000
s <- subset(d, speciesname == "M. californianus" & yearnum ==
  2, select = c("latitude", "cover", "sst_mean"))
```

Next, we prepare the plotting region to accommodate the axis on the right side:

⁶That's not entirely true. We can also standardize different datasets by deducting the mean and dividing by the standard deviation, thus generating values across a common range that can be plotted using a single set of axes.

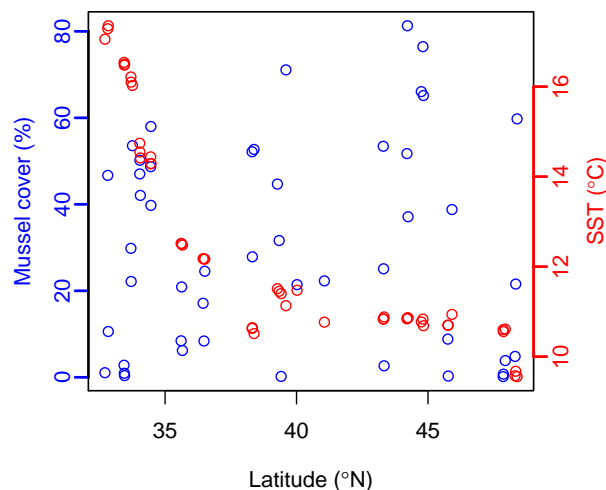
```
# Make room in margin for right axis
par(mar = c(5.1, 4.1, 0.6, 4.1))
```

Now, we plot mussel cover in blue and link it to a blue y-axis on the left of the figure:

```
# Plot the cover in blue but without a yaxis so we can
# add it later
plot(s$latitude, s$cover, t = "p", col = "blue", yaxt = "n",
     ylab = "", xlab = expression(paste("Latitude (", degree,
                                         "N)")))
# Add blue axis to the left of the plot
axis(2, col = "blue", col.ticks = "blue", label = FALSE,
     lwd = 2)
# Add the y-axis tickmark labels for mussel cover in
# blue
mtext(side = 2, at = axTicks(2), text = axTicks(2), line = 0.5,
      col = "blue")
# Add y-axis label for mussel cover
mtext("Mussel cover (%)", side = 2, line = 2, col = "blue")
```

We must now overlay a new plot on top of the existing one to represent sea surface temperature. However, if we were to simply use the `plot` command, we would replace the previous plot. To avoid crushing the previous plot, we use the `par(new=TRUE)` command to add to instead of replace the previous plot:

```
# Overlay a new plot for the temperature data
par(new = TRUE)
# Plot sea surface temperature in red
plot(s$latitude, s$sst_mean, type = "p", col = "red", yaxt = "n",
     yaxt = "n", xlab = "", ylab = "")
# Add axis in red
axis(4, col = "red", col.ticks = "red", label = FALSE, lwd = 2)
# Add axis tickmarks in red
mtext(side = 4, at = axTicks(4), text = axTicks(4), line = 0.5,
      col = "red")
# Add red axis label to the right side of the plot
mtext(text = expression(paste("SST (", degree, "C)")), side = 4,
      line = 2, col = "red")
```



Multiple panels

Another way to represent variables with different ranges and units onto a single figure is to plot them on separate panels. R allows figures to be subdivided into a multiple regions via the `mfrow(x, y)` argument to function `par`, with `x` specifying the number of rows and `y` specifying the number or columns. Additionally, each panel in a multi-panel figure must be labeled. Let's try to plot mussel cover and sea surface temperature on separate panels of a figure. We begin by splitting the plotting region into two columns:

```
# Split plot into two rows and one column
par(mfrow = c(1, 2))
```

Next, we need to keep track of the current user coordinates of the figure (see `?par` for details):

```
# Store current coordinate system
op <- par("usr")
```

Now we plot the mussel cover data first. In this case, it will be much simpler since we don't have to use different colors for the axes:

```
# Plot the cover in blue but without a yaxis so we can
# add it later
plot(s$latitude, s$cover, ylab = "Mussel cover (%)", xlab = expression(paste("Latitude (",
  degree, "N)")), t = "p", col = "blue")
```

We now add the label “(a)” for this first panel by changing the coordinate system so that it is relative. In the relative coordinate system, coordinates along the x and y axes will vary between 0 to 1 regardless of the range of the data being plotted. This allows us to position the labels in exactly the same place on each panel, regardless of the ranges of the data being plotted across the different panels. However, we must reset the coordinate system after plotting the label:

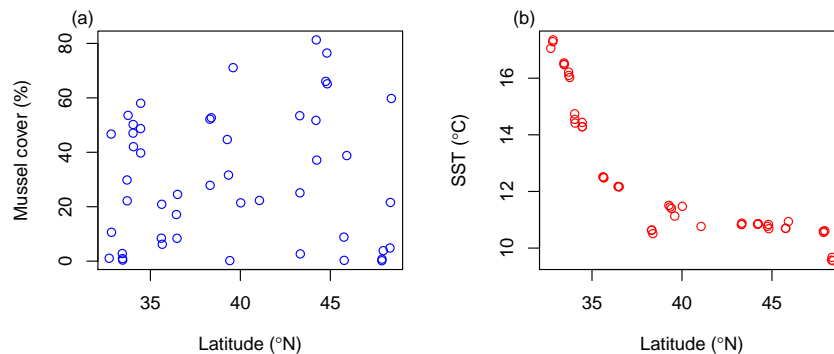
```
# Change coordinate system to relative
par(usr = c(0, 1, 0, 1))
# Add text label for first panel
text(-0.05, 1.05, "(a)", col = "black", xpd = NA)
# Reset coordinate system
par(usr = op)
```

Now, we plot the sea surface temperature data in the second panel:

```
plot(s$latitude, s$sst_mean, xlab = expression(paste("Latitude (",
  degree, "N)")), ylab = expression(paste("SST (", degree,
  "C)")), type = "p", col = "red")
```

And then add label “(b)” to the panel:

```
# Change coordinate system to relative
par(usr = c(0, 1, 0, 1))
# Add text label for first panel
text(-0.05, 1.05, "(b)", col = "black", xpd = NA)
# Reset coordinate system
par(usr = op)
```



N The argument `xpd=NA` allows text written in the margins to appear on the figure.

2.15.4 Alternative systems

There are two alternative systems for plotting figures in R. As I demonstrate below, these systems really excel when it comes to plotting multi-panel figures.

lattice

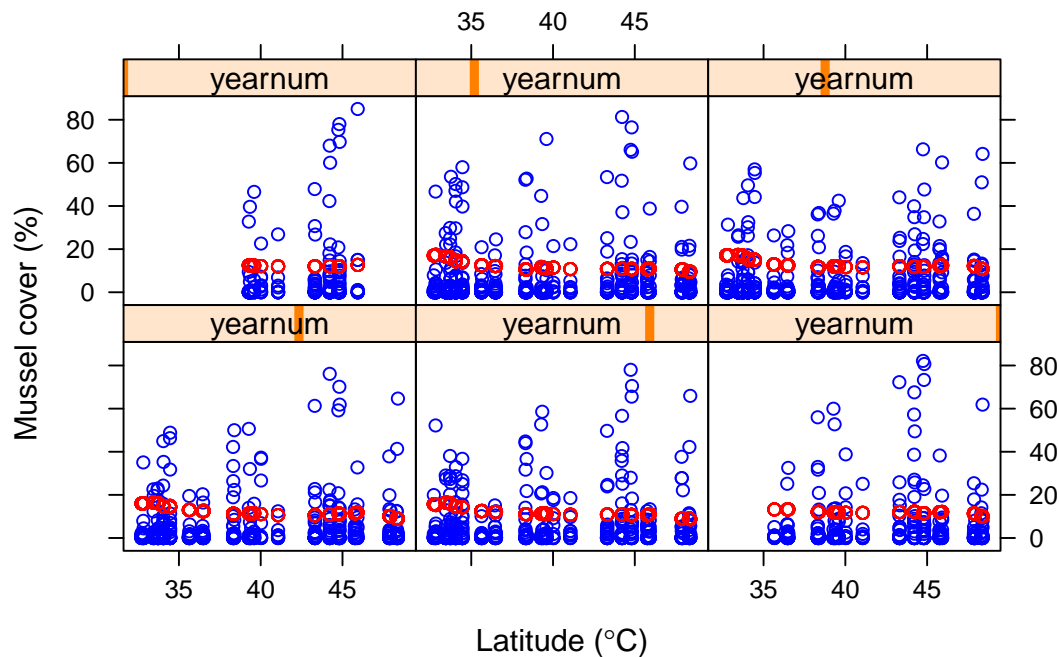
The `lattice` package makes it easy to plot multi-panel figures by exploiting the `formula` system in R. To show how this works, let's create a figure by plotting the mean annual mussel cover and sea surface temperature for each year on a separate panel. To do so, we will need to load the `lattice` package and download the data:

```
library(lattice)
d <- read.csv("http://faraway.neu.edu/data/pisco.csv")
```

Now, we can plot the data using function `xyplot`:

```
xyplot(cover + sst_mean ~ latitude | yearnum, data = d,
       layout = c(3, 2), type = "p", col = c("blue", "red"),
       xlab = expression(paste("Latitude (", degree, "C)")),
       ylab = "Mussel cover (%)", as.table = TRUE)
```

The first argument to `xyplot` is a formula: `cover + sst_mean`, which appears to the left of the `~` symbol, represents the response variables that we want to plot on the y-axis and `latitude`, which appears to the right of the `~` symbol, represents the explanatory variable that we want to show on the x-axis. The vertical bar `|` is a **grouping** symbol used to indicate that the operation appearing to the left of it needs to be applied to all unique values of the grouping variable appearing to the right of it. In this case, it means that we want to plot both mean cover and sea surface temperature as a function of latitude for each year number (the grouping variable). The function `xyplot` uses the unique values of this grouping variable to create each new panel. The `layout` argument specifies how the panels should be arranged. In this case, because argument `as.table=TRUE`, it means that we want 2 rows and 3 columns. If argument `as.table=FALSE` were set, then the layout would be 3 rows and 2 columns. The `type` argument is used to specify the type of plot, in this case points. Finally, the `col` argument specifies the colors to be used when plotting each variable in the same order that they appear in the formula. The rest of the arguments are identical to those used in the base plotting system and thus need no further explanation.

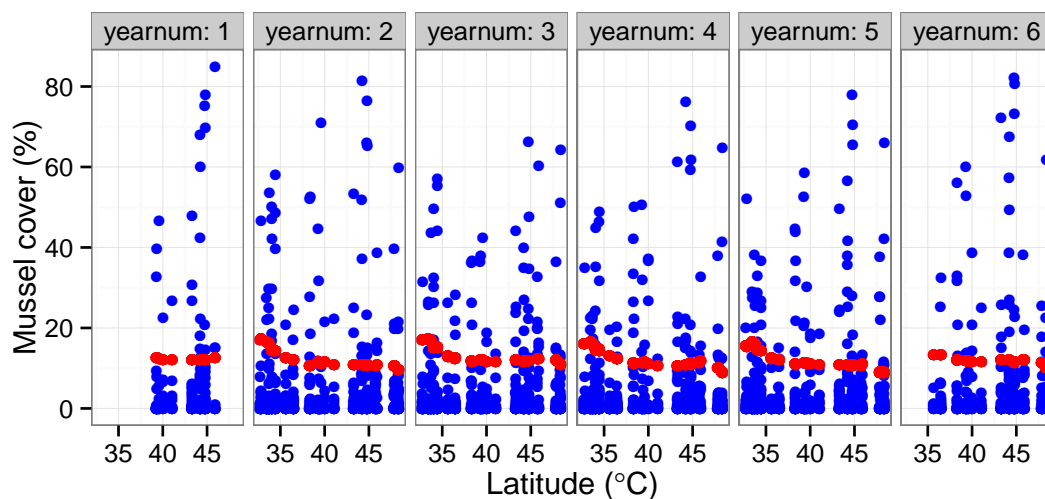


Although we could generate the same figure using the base package, it would take a whole page of code...

ggplot2

The ggplot2 system is the other alternative that provides very efficient ways of plotting multi-panel figures. For example, to plot the same figure as above:

```
# Load ggplot2
library(ggplot2)
# Plot data
p <- ggplot(d, aes(latitude)) + geom_point(aes(y = cover),
  col = "blue") + geom_point(aes(y = sst_mean), col = "red") +
  theme_bw()
p + facet_grid(. ~ yearnum, lab = label_both) + ylab("Mussel cover (%)") +
  xlab(expression(paste("Latitude (", degree, "C)")))
```



I will not dwell on the grammar of graphics used to implement `ggplot2`. However, you can use the extensive documentation provided with the package and [online](#) to make sense of the code above.

2.15.5 Saving plots

The general mechanism for saving a figure in R consists of three steps: (1) open a file to store the figure, (2) plot the results, and then (3) close the file. For example, to save a plot as a PDF:

```
# Create a PDF file to store the figure
pdf(file = "myPDF.pdf", width = 8, height = 6)
# Plot the results in the newly created file
plot(1:10)
# Close the file
dev.off()
```

In the above code block, the `pdf(file = "myPDF.pdf", width = 8, height = 6)` command created a PDF file called `myPDF.pdf`, then the command `plot(1:10)` function plotted data and stored the plot in the file, and finally command `dev.off()` closed the file. Note that if you do not close the file via `dev.off()`, the file will continue to capture subsequent plots. To close all plots, use command `graphics.off()`. The following are valid formats for saving figures:

Function	Description
<code>pdf(file="out.pdf", width=8, height=6, pointsize=12)</code>	Create a PDF called "out.pdf", with width 8 inches, height 6 inches, and point-size 12
<code>tiff(file="out.tiff", width=400, height=600, pointsize=12, res=72)</code>	Create a TIFF called "out.tiff", with width 400 pixels, height 600 pixels, and pointsize 12
<code>png(file="out.png", width=400, height=600, pointsize=12, res=72)</code>	Create a PNG called "out.png", with width 400 pixels, height 600 pixels, and pointsize 12

The `pdf` function saves the figure using a **vector format** that is ideal for publication because the quality does not change when the figure is subsequently shrunk or expanded. However, functions like `png` and `tiff` save figures using a **bitmap format**, which behaves poorly when the size is altered in any way. Hence, I recommend saving all figures using the `pdf` function.

2.16 Programming

R and its external packages provide many functions to manipulate datasets, perform calculations, and plot results. However, it is often necessary to develop code to provide additional functionality (e.g., our standard error function). In such cases, we need to know the elements of the R programming language that control the flow of programs (i.e., the evaluation of expressions).

2.16.1 Control structures

There are two main types of control structures in R that can be used to repeat commands and/or execute code conditionally. To repeat blocks of code, one can use either `for` or `while` loops. A simple example of a `for` loop is shown below:


```
for (x in 1:5) {
  print(x)
}

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

The structure of a **for** loop is similar to that of a function. The argument is a condition that is used to determine how many times the **for** loop should run. In the code above, the condition is `x in 1:5`, which creates an index variable `x` and assigns it a successive value from the vector `1:5` each time the loop is run. Hence, the **for** loop will run 5 times since the vector `1:5` is of length 5. The body of the **for** loop (the code between the curly braces) will get executed each time the **for** loop is run. In this case, the body merely prints the loop iteration `x`. However, we can replace the body of the **for** loop with something a little more useful.

For instance, let's create some code to take the \log_{10} of each value stored in vector `vals`⁷. In this case, because the **for** loop will create a new value at each iteration, we need to prepare a `log.vals` vector whose length matches the number of iterations in the **for** loop, and thus the length of the `vals` vector:

```
# Values
vals <- 1:5
# Initialize a log10.vals vector of the same length as
# vals
log.vals <- numeric(length(vals))
# For each value in vals compute the log10 and store it
# in log.vals
for (x in 1:length(vals)) {
  # Store log value in log.vals vector
  log.vals[x] <- log10(vals[x])
}
# Print logged values
log.vals

## [1] 0.0000000 0.3010300 0.4771213 0.6020600 0.6989700
```

A **while** loop performs a similar function in that it allows blocks of code to be repeated. However, the number of iterations for a **while** loop is set dynamically within the loop itself, and not fixed statically outside as it is in the **for** loop. To show this distinction, let's use a **while** loop to replicate the functionality shown in the first example of a **for** loop:

```
# Initialize the loop index variable
x <- 1
while (x <= 5) {
  print(x)
  # Increment the loop index variable
  x <- x + 1
}
```

⁷This function is not very useful and for illustrative purposes only because function `log10` can take vectors of values as input and compute the log value for each element in the vector.

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

Although the results are identical to those obtained using the **for** loop, the structure of the code is quite different. First, the index variable **x** is initialized outside the **while** loop. Indeed, the **while** loop takes a single condition as an argument but does not **update** it the way the **for** loop does. Instead, the **x** variable is updated within the body of the **while** loop (**x=x+1**). This loop is repeated until **x>5**. Hence, the main distinction between **for** loops and **while** loops is how and where the iterations are controlled.

The second important component of any programming language is the ability to execute code based on conditions. In R, this is done via **if-else** blocks and **ifelse**⁸. The syntax of an **if-else** block is:

```
x <- 3
# If x is equal to 5 set v to TRUE
if (x == 5) {
  v <- TRUE
  print("x is equal to 5")
  # If x is not 5 set v to FALSE
} else {
  v <- FALSE
  print("x is not equal to 5")
}

## [1] "x is not equal to 5"
```

The **if-else** statement takes a condition as an argument (**x==5**) and executes the first block of code between curly braces if the condition is **TRUE**, or the second block of code between curly braces (i.e., the one after **else**) if the condition is **FALSE**. Let's use conditional code evaluation in a more interesting example. Recall that in the second **for** loop example, we computed the \log_{10} of values. However, the code will produce **-Inf** if the values in the **vals** vector are smaller than or equal to zero. Here, we can use the combination of a **for** loop and an **if-else** statement to control for this:

```
(vals <- -1:4)

## [1] -1 0 1 2 3 4

# Initialize a log10.vals vector of the same length as vals
log.vals <- numeric(length(vals))

for (x in 1:length(vals)) {
  # if the value is > 0 return its log10
  if (vals[x] > 0) {
    log.vals[x] <- log10(vals[x])
  }
  # if the value <= 0 return NA
  else {
    log.vals[x] <- NA
  }
}

# Print log.vals
log.vals
```

⁸R also has a **switch** command that functions like a menu to select one among multiple alternatives.

```
## [1]      NA      NA 0.0000000 0.3010300 0.4771213 0.6020600
```

This shows how loops and conditional statements such as `if-else` blocks can be used to apply different calculations on collections of objects based on their values. The `ifelse` statement can be used to replicate the functionality of the code above with a single statement. Indeed, `ifelse` evaluates a condition for each element of a vector and then performs different calculations based on whether that condition is satisfied or not:

```
ifelse(vals > 0, log10(vals), NA)

## [1]      NA      NA 0.0000000 0.3010300 0.4771213 0.6020600
```

The first argument of `ifelse` is the condition `vals>0`, the second argument `log10(vals)` is the value that is returned if the condition is `TRUE` for each element of `vals`, and the third argument `NA` is the value that is returned if the condition is `FALSE`. Hence, this one line of code is equivalent to the entire block of code above!

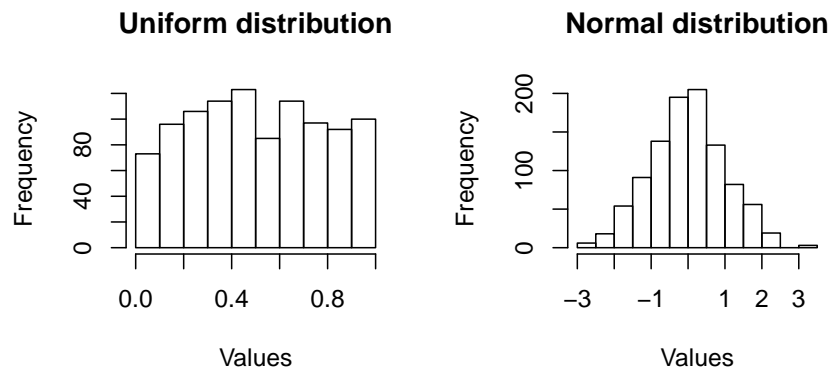
2.16.2 Random numbers

Random numbers are often used when programming. In R, random numbers can be generated from a number of different distributions (see `?Distributions`). Each type of random number is associated with four different functions used to produce the **density**, the **distribution function**, **quantile function**, and **random samples**. Below, I highlight two of the most frequently used distributions:

Function	Description
<code>dnorm(x, mean, sd)</code> ; <code>pnorm(q, mean, sd, ...)</code> ; <code>qnorm(p, mean, sd, ...)</code> ; <code>rnorm(n, mean, sd)</code>	Normal distribution: Density function; Distribution function; quantile function; random deviate
<code>dunif(x, mean, sd)</code> ; <code>punif(q, mean, sd, ...)</code> ; <code>qunif(p, mean, sd, ...)</code> ; <code>runif(n, mean, sd)</code>	Uniform distribution: Density function; Distribution function; quantile function; random deviate

For example, here is how to draw random numbers from the uniform and normal distributions:

```
# Draw 1000 random numbers from the uniform distribution
unif <- runif(1000)
# Draw 1000 random numbers from the normal distribution
norm <- rnorm(1000)
# Plot the distributions using histograms
par(mfrow = c(1, 2))
hist(unif, main = "Uniform distribution", xlab = "Values")
hist(norm, main = "Normal distribution", xlab = "Values")
```



Although the functions described above can be used to select values at random from a compound object such as a vector, it is much easier to use the convenient `sample` function. Indeed, the `sample` function can be used to sample anywhere from 1 to all elements in a vector or a matrix with or without replacement. When all values are selected from a vector or matrix, the `sample` function merely shuffles the values around (i.e., changes their order). However, if only a few values are selected, then the `sample` function lives up to its name and draws a sample from the larger population (i.e., all values in the vector or matrix). As we will see later on, the `sample` function is central to permutation-, bootstrap-, and jackknife-based Monte Carlo approaches.

2.17 Getting help

Although this introduction to R is rather thorough, it may not answer all of your questions. You can seek further assistance from a variety of sources:

From the R console:

- `?command`: Brings up help for `command` in RStudio
- `help.search("command")`: Same as above
- `RSiteSearch("command")`: Open web browser to search R vignettes and help forum

From the web:

- <http://www.rseek.org>
- <http://www.stackoverflow.com>
- [Video tutorials](#)

From reference documents:

- [R cheat sheet](#)
- [R manuals](#)
- [R inferno](#)



3 — Reproducibility and R markdown

3.1 Using literate programming to combat the reproducibility crisis

If science is defined as the search for universal truths, then scientific results ought to be reproducible and verifiable. For instance, if a lab group claims to be able to convert somatic cells into stem (pluripotent) cells by exposing them to harsh acidic conditions, then other researchers ought to be able to reproduce those results by following the same approach or protocol (Obokata et al., 2014a,b). Similarly, if a lab group claims to have found a bacterium that is able to substitute phosphorous—one of the universal building blocks of life—then other researchers ought to be able to verify that claim by conducting similar or complementary biological assays on the same organism (Wolfe-Simon et al., 2011). Finally, if leading economists claim that an analysis of Gross Domestic Product (GDP) to debt ratios reveals a “tipping-point” whereby countries with a GDP-to-debt ratio of greater than or equal to 90% have significantly lower GDP, then other researchers using the same data and methods should reach identical conclusions (Reinhart and Rogoff, 2010). However, these three examples represent some of most spectacular and high-profile cases of irreproducible results. Although the causes of this irreproducibility varied from apparent fraud to utter incompetence, these examples all generated significant controversy (Cassidy, 2013) and, in the case of the stem cell example, multiple retractions from high-impact scientific journals and the suicide of a well-known developmental biologist.

The reproducibility crisis has become so widespread that leading journals such as *Nature*¹ and *Science* have devoted entire editorial issues to the problem (McNutt, 2014). In an attempt to curb the crisis, journals and funding agencies are increasingly requiring that authors submit their raw data alongside their analyses. Doing so should allow other researchers to verify the authors’ claims as long as the methodology described in the main text is complete and accurate. To further promote reproducibility, statistical and methodological journals are going one step further by calling for the inclusion of the computer code used to conduct the analyses. One way of seamlessly weaving analytical results, code and comments together in a single executable document is to use literal programming. Literal programming is a term invented by Donald Knuth—one of the

¹<http://www.nature.com/nature/focus/reproducibility/>

(modern) gods of computer science—to refer to the practice of combining executable code and the comments describing that code into a single file. This allows the programmer or analyst to explain each command, making the code easier to interpret for the casual reader. To promote reproducibility and adopt these recommendations, you will be using a very simple but convenient literal programming language called **R markdown** to write your assignments (Baumer et al., 2014). **R markdown** will allow you to combine your code, comments and interpretation into a single executable file. Doing so will make it easier for you to complete your assignments (no need to paste code or graphics from RStudio to Word) and for the TAs to grade them.

3.2 Installing R markdown

R markdown is included in the latest version of RStudio, so there is no need to download it separately. However, you will need to download the appropriate \LaTeX distribution for your computer in order to produce a PDF from your **R markdown** assignment files. If you are a Windows user, please visit <http://miktex.org/download> to download and install the latest version of the MikTeX 64-bit Net Installer found in the “Other Downloads” section. If you are a Mac user, please visit <http://www.tug.org/mactex/> to download and install the latest version of MacTeX. These \LaTeX distributions are relatively large (> 2 GB), so make sure you have a stable (preferably wired) internet connection. Once your \LaTeX distribution has been installed, you will be ready to use RStudio to complete your assignments.

3.3 Using R markdown for assignments

To begin your assignment, download a copy of the assignment template at <http://faraway.neu.edu/data/biostatistics-assnTemplate.Rmd>. Open this file in RStudio and save it as `assnNUMBER-yourLastName.Rmd`. The file should look like this when opened in RStudio:

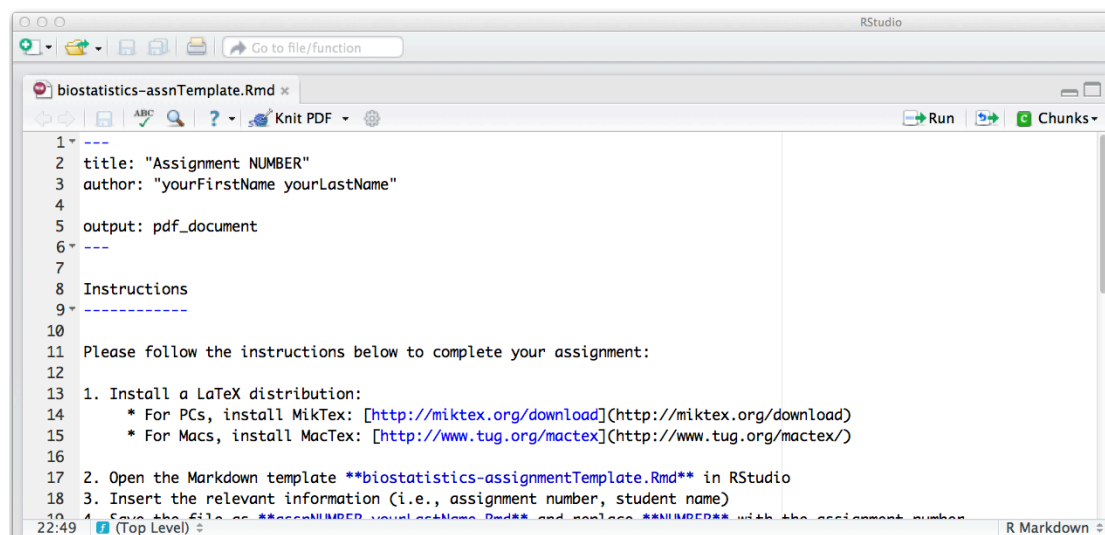


Figure 3.1: R markdown file in RStudio

The top of the file contains basic information such as the title and author of the document, along with the desired output (in this case PDF). If there are no problems with the code in your **R markdown** file, clicking on the “Knit PDF” icon at the top of

the RStudio Editor window will generate a PDF version of your file that includes your code, its output (e.g., figures, tables), along with all the text surrounding your code. You can alter the way the text around your code appears by using some basic markdown commands. For example, top-level headers are specified by placing dashes below your text or a single pound sign to the left of it (e.g., `# Top-level header`). Second-level headers are specified by placing two pound signs to the left of the text (e.g., `## Second-level header`). To emphasize text, one can use italics by placing single asterisks around the text (e.g., `*my italicized text*`) or bold by placing double asterisks around the text (e.g., `**my bold text**`). For more formatting options, please consult the help file by clicking on the “?” icon at the top of the RStudio Editor window and selecting “Markdown Quick Reference”.

To add R code, you need to create a new **chunk** by typing the following in your R markdown file:

```
```{r}
rand.unif <- runif(10)
rand.norm <- rnorm(10)
```
```

Anything between the apostrophes will be executed by R once the file is compiled into a PDF by clicking on the “Knit PDF” icon. In this case, the code above will store 10 random values drawn from a uniform distribution into variable `rand.unif` and store 10 random values drawn from a normal distribution into variable `rand.norm`. However, you can replace this code with any valid R command. You can then add your interpretation of the results below the code chunk. For instance, the assignment template shows how to answer question 1 in problem 1:



```

24 Problem 1
25 ~~~~~
26 - ### Question 1
27
28 - ```{r}
29 pisco <- read.csv(file = "http://faraway.neu.edu/data/pisco_env.csv")
30 plot(pisco$latitude, pisco$sst_mean, xlab="Latitude", ylab="Sea Surface Temperature (C)")
31 ~~~~~
32
33 There appears to be a clear latitudinal trend in Sea Surface Temperature along the West Coast of the US, with temperature decreasing with
latitude.

```

Figure 3.2: R markdown file in RStudio

Once the file is compiled, the code between the apostrophes will generate a figure that will be embedded in a PDF alongside the interpretation that follows the code (see <http://faraway.neu.edu/data/dynamics-assnTemplate.pdf>). These simple instructions should allow you to complete your assignment with minimal effort and maximal reproducibility.



Bibliography

- Baumer, B., M. Cetinkaya-Rundel, A. Bray, L. Loi, and N. J. Horton. 2014. R markdown: Integrating a reproducible analysis tool into introductory statistics. arXiv:1402.1894 [stat].
- Cassidy, J. 2013. The reinhart and rogoﬀ controversy: A summing up.
- McNutt, M. 2014. Reproducibility. *Science*, **343**:229–229.
- Obokata, H., Y. Sasai, H. Niwa, M. Kadota, M. Andrabi, N. Takata, M. Tokoro, Y. Terashita, S. Yonemura, C. A. Vacanti, and T. Wakayama. 2014*a*. Bidirectional developmental potential in reprogrammed cells with acquired pluripotency. *Nature*, **505**:676–680.
- Obokata, H., T. Wakayama, Y. Sasai, K. Kojima, M. P. Vacanti, H. Niwa, M. Yamato, and C. A. Vacanti. 2014*b*. Stimulus-triggered fate conversion of somatic cells into pluripotency. *Nature*, **505**:641–647.
- Reinhart, C. M. and K. S. Rogoﬀ. 2010. Growth in a time of debt. *American Economic Review*, **100**:573–578.
- Wickham, H. 2009. *ggplot2: Elegant Graphics for Data Analysis*. Springer, 1 edition.
- Wilkinson, L. 2005. *The Grammar of Graphics*. Springer, 2nd edition.
- Wolfe-Simon, F., J. S. Blum, T. R. Kulp, G. W. Gordon, S. E. Hoeft, J. Pett-Ridge, J. F. Stolz, S. M. Webb, P. K. Weber, P. C. W. Davies, A. D. Anbar, and R. S. Oremland. 2011. A bacterium that can grow by using arsenic instead of phosphorus. *Science*, **332**:1163–1166.

Index

- An overview of R, 3
- Combining objects, 12
- Components of R, 6
- Compound objects, 8
- Data wrangling, 50
 - Combining datasets, 51
 - Computing statistics, 54
 - Reshaping datasets, 52
- Functions, 39
- Getting help, 75
- Getting started with R, 4
- Input and Output, 46
 - Exporting data, 50
 - Importing data, 49
 - Managing your R session, 46
 - Navigating the filesystem, 47
 - Packages, 50
- Installing R markdown, 77
- Manipulating compound objects, 14
 - Arrays, 23
 - Converting compound objects, 32
 - Data frames, 24
 - Logical indices, 27
 - Named indices, 26
 - Numerical indices, 25
 - Sorting, 28
 - Lists, 29
 - Named indices, 31
 - Numerical indices, 29
 - Matrices, 18
 - Logical indices, 21
 - Named indices, 20
 - Numerical indices, 18
 - Sorting, 22
 - Vectors, 14
 - Logical indices, 17
 - Named indices, 16
 - Numerical indices, 14
 - Sorting, 17
- Manipulating text, 34
 - Characters, 34
 - Cleaning text, 35
 - Combining text, 37
 - Splitting text, 36
 - Substituting text, 35
 - Factors, 37
- Objects and variables, 6
- Operators, 40
 - Assignment operators, 41
 - Logical operators, 43
 - Mathematical operators, 42
 - When operators go bad, 45
- Plotting, 58
 - Alternative systems, 69
 - ggplot2, 70
 - lattice, 69
 - Base system, 59
 - Customizing plots, 64
 - Adding text, 65

- Layering, [64](#)
- Multiple axes, [66](#)
- Multiple panels, [68](#)
- Saving plots, [71](#)
- Types of plots, [60](#)
 - Bar plot, [62](#)
 - Boxplots, [63](#)
 - Histograms, [64](#)
 - Line plot, [61](#)
- Programming, [71](#)
 - Control structures, [71](#)
 - Random numbers, [74](#)
- Using literate programming to combat the
reproducibility crisis, [76](#)
- Using R markdown, [77](#)
- What is programming?, [1](#)
- Why learn to program in R?, [2](#)