
The RBioc Book

SEAN DAVIS

University of ColoradoAnschutz School of
Medicine

2023-06-29

Table of contents

Preface	1
Who is this book for?	1
Why this book?	1
Adult learners	1
I. Introduction	4
1. Introducing R and RStudio	5
Questions	5
Learning Objectives	5
1.1. Introduction	5
1.2. What is R?	5
1.3. Why use R?	6
1.4. Why not use R?	7
1.5. R License and the Open Source Ideal	7
1.6. RStudio	8
1.6.1. Getting started with RStudio	8
1.6.2. The RStudio Interface	8
2. R mechanics	11
2.1. Learning objectives	11
2.2. Starting R	11
2.3. <i>RStudio</i> : A Quick Tour	11
2.4. Interacting with R	12
2.4.1. Expressions	13
2.4.2. Assignment	13
2.5. Rules for Names in R	15
2.6. Resources for Getting Help	16
2.7. Further practice	17
2.8. Exercises	17

Table of contents

3. The Very Basics	19
3.1. The R User Interface	19
3.1.1. An exercise	22
3.2. Objects	23
3.3. Functions	30
3.3.1. Sample with Replacement	34
3.4. Writing Your Own Functions	37
3.4.1. The Function Constructor	38
3.5. Arguments	40
3.6. Scripts	42
3.7. Summary	44
 II. Overview of R Data Structures	 46
Chapter overview	48
 4. Vectors	 49
4.1. What is a Vector?	49
4.2. Creating Vectors	49
4.3. Accessing Vector Elements	51
4.4. Vector Operations	52
4.5. Named Vectors	54
4.6. Vector Recycling and Recycling Rule	55
4.7. Active Learning Exercises	56
4.8. Creating vectors	57
4.9. Vector Operations	58
4.10. Logical Vectors	60
4.10.1. Logical Operators	60
4.11. Indexing Vectors	61
4.12. Character Vectors, A.K.A. Strings	63
4.13. Missing Values, AKA “NA”	64
4.14. Exercises	65
 5. Matrices	 67
5.1. Creating a matrix	67
5.2. Accessing elements of a matrix	70
5.3. Changing values in a matrix	72
5.4. Calculations on matrix rows and columns	74
5.5. Exercises	76
5.5.1. Data preparation	76
5.5.2. Questions	76

Table of contents

6. Data Frames	79
6.1. Learning goals	79
6.2. Learning objectives	79
6.3. Dataset	80
6.4. Reading in data	81
6.5. Inspecting data.frames	81
6.6. Accessing variables (columns) and subsetting	85
6.6.1. Some data exploration	86
6.6.2. More advanced indexing and subsetting . .	87
6.7. Aggregating data	91
6.8. Creating a data.frame from scratch	92
6.9. Saving a data.frame	93
7. Factors	94
7.1. Factors	94
III. Exploratory data analysis	97
8. Case Study: Behavioral Risk Factor Surveillance System	99
8.1. A Case Study on the Behavioral Risk Factor Surveil- lance System	99
8.2. Loading the Dataset	100
8.3. Inspecting the Data	100
8.4. Summary Statistics	101
8.5. Data Visualization	102
8.6. Analyzing Relationships Between Variables	103
8.7. Exercises	104
8.8. Conclusion	107
8.9. Learn about the data	107
8.10. Clean data	107
8.11. Weight in 1990 vs. 2010 Females	108
8.12. Weight and height in 2010 Males	109
IV. statistics	114
9. The t-statistic and t-distribution	115
9.1. Background	115
9.2. The Z-score and probability	115
9.2.1. Small diversion: two-sided pnorm function	117

Table of contents

9.3. The t-distribution	118
9.3.1. p-values based on Z vs t	121
9.3.2. Experiment	122
9.4. Summary of t-distribution vs normal distribution	126
9.5. t.test	127
9.5.1. One-sample	127
9.5.2. two-sample	128
9.5.3. from a data.frame	128
9.5.4. Equivalence to linear model	129
10. K-means clustering	130
10.1. History of the k-means algorithm	130
10.2. The k-means algorithm	131
10.3. Pros and cons of k-means clustering	131
10.4. An example of k-means clustering	133
10.4.1. The data and experimental background	133
10.5. Getting data	134
10.6. Preprocessing	135
10.7. Clustering	137
10.8. Summary	138
V. Bioconductor	139
11. Introduction to SummarizedExperiment	140
11.1. Anatomy of a SummarizedExperiment	140
11.1.1. Assays	141
11.1.2. 'Row' (regions-of-interest) data	143
11.1.3. 'Column' (sample) data	144
11.1.4. Experiment-wide metadata	145
11.2. Common operations on SummarizedExperiment	146
11.2.1. Subsetting	146
11.2.2. Getters and setters	147
11.2.3. Range-based operations	149
11.3. Constructing a SummarizedExperiment	149
References	152

Table of contents

Appendices	153
A. Appendix	153
A.1. Data Sets	153
A.2. Swirl	153
B. Additional resources	154

List of Figures

0.1.	Why do adults choose to learn something?	2
0.2.	How to stay stuck in data science (or anything). The “Read-Do” loop tends to deliver the best results. Too much reading between doing can be somewhat effective. Reading and simply copy-paste is probably the least effective. When working through material, experiment. Try to break things. Incorporate your own experience or applications whenever possible.	3
1.1.	Google trends showing the popularity of R over time based on Google searches	6
1.2.	The RStudio interface. In this layout, the source pane is in the upper left, the console is in the lower left, the environment panel is in the top right and the viewer/help/files panel is in the bottom right.	9
3.1.	Your computer does your bidding when you type R commands at the prompt in the bottom line of the console pane. Don’t forget to hit the Enter key. When you first open RStudio, the console appears in the pane on your left, but you can change this with File > Tools > Global Options in the menu bar. . .	20
3.2.	Assignment creates an object in the environment pane.	26
3.3.	“When R performs element-wise execution, it matches up vectors and then manipulates each pair of elements independently.”	28
3.4.	“R will repeat a short vector to do element-wise operations with two vectors of uneven lengths.”	29
3.5.	“When you link functions together, R will resolve them from the innermost operation to the outermost. Here R first looks up die, then calculates the mean of one through six, then rounds the mean.” . .	31

List of Figures

3.6.	“Every function in R has the same parts, and you can use <code>function</code> to create these parts. Assign the result to a name, so you can call the function later.”	43
3.7.	“When you open an R Script (File > New File > R Script in the menu bar), RStudio creates a fourth pane (or puts a new tab in the existing pane) above the console where you can write and edit your code.”	43
3.8.	A pictorial representation of R’s most common data structures are vectors, matrices, arrays, lists, and dataframes. Figure from Hands-on Programming with R.	47
5.1.	A matrix is a collection of column vectors.	67
9.1.	t-distributions for various degrees of freedom. Note that the tails are fatter for smaller degrees of freedom, which is a result of estimating the standard deviation from the data.	120
10.1.	K-means clustering takes a dataset and divides it into k clusters.	130
10.2.	Histogram of standard deviations for all genes in the deRisi dataset.	136
10.3.	Gene expression profiles for the four clusters identified by k-means clustering. Each line represents a gene in the cluster, and each column represents a time point in the experiment. Each cluster shows a distinct trend where the genes in the cluster are potentially co-regulated.	137
11.1.	Summarized Experiment. There are three main components, the <code>colData()</code> , the <code>rowData()</code> and the <code>assays()</code> . The accessors for the various parts of a complete <code>SummarizedExperiment</code> object match the names.	142

List of Tables

4.1. Atomic (simplest) data types in R.	57
---	----

Preface

Who is this book for?

- People who want to learn data science
- People who want to teach data science
- People who want to learn how to teach data science
- People who want to learn how to learn data science

Why this book?

This book is a collection of resources for learning data science. It is meant to be largely self-directed, but for those looking to teach data science, it can also be used as a guide for structuring a course. Material is a bit variable in terms of difficulty, prerequisites, and format which is a reflection of the organic creation of the material.

Students are encouraged to work with others to learn the material. Instructors are encouraged to use the material to create a course that is tailored to the needs of their students and to spend lots of time in 1:1 and small groups to support students in their learning. See below for additional thoughts on adult learning and how it relates to this material.

Adult learners

Adult Learning Theory, also known as Andragogy, is the concept and practice of designing, developing, and delivering instructional experiences for adult learners. It is based on the belief that adults learn differently than children, and thus, require distinct approaches to engage, motivate, and retain information Center (2016). The term

Adult learners

was first introduced by Malcolm Knowles, an American educator who is known for his work in adult education.

One of the fundamental principles of Adult Learning Theory is that adults are self-directed learners. This means that we prefer to take control of our own learning process and set personal goals for themselves. We are motivated by our desire to solve problems or gain knowledge to improve our lives (see Figure 0.1). As a result, educational content for adults should be relevant and applicable to real-life situations. Furthermore, adult learners should be given opportunities to actively engage in the learning process by making choices, setting goals, and evaluating their progress.

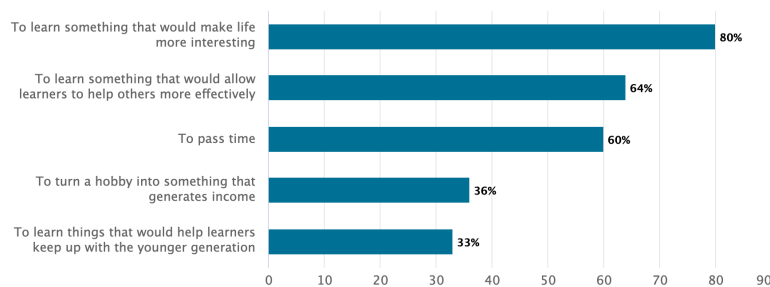


Figure 0.1.: Why do adults choose to learn something?

Another key aspect of Adult Learning Theory is the role of experience. We bring a wealth of experience to the learning process, which serves as a resource for new learning. We often have well-established beliefs, values, and mental models that can influence our willingness to accept new ideas and concepts. Therefore, it is essential to acknowledge and respect our shared and unique past experiences and create an environment where we all feel comfortable sharing our perspectives.

To effectively learn as a group of adult learners, it is crucial to establish a collaborative learning environment that promotes open communication and fosters trust among participants. We all appreciate and strive for a respectful and supportive atmosphere where we can express our opinions without fear of judgment. Instructors should help facilitate discussions, encourage peer-to-peer interactions, and incorporate group activities and collaboration to capitalize on the collective knowledge of participants.

Additionally, adult learners often have multiple responsibilities outside of the learning environment, such as work and family commitments. As a result, we require flexible learning opportunities that

Adult learners

accommodate busy schedules. Offering a variety of instructional formats, such as online modules, self-paced learning, or evening classes, can help ensure that adult learners have access to education despite any time constraints.

Adult learners benefit from a learner-centered approach that focuses on the individual needs, preferences, and interests of each participant can greatly enhance the overall learning experience. In addition, we tend to be more intrinsically motivated to learn when we have a sense of autonomy and can practice and experiment (see Figure 0.2) with new concepts in a safe environment.

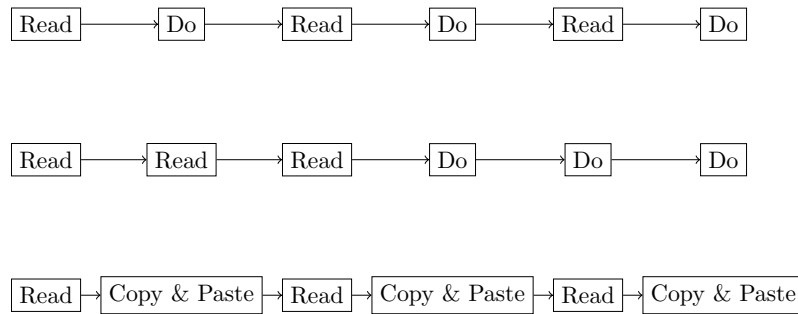


Figure 0.2.: How to stay stuck in data science (or anything). The “Read-Do” loop tends to deliver the best results. Too much reading between doing can be somewhat effective. Reading and simply copy-paste is probably the least effective. When working through material, experiment. Try to break things. Incorporate your own experience or applications whenever possible.

Understanding Adult Learning Theory and its principles can significantly enhance the effectiveness of teaching and learning as adults. By respecting our autonomy, acknowledging our experiences, creating a supportive learning environment, offering flexible learning opportunities, and utilizing diverse teaching methods, we can better cater to the unique needs and preferences of adult learners.

In practice, that means that we will not be prescriptive in our approach to teaching data science. We will not tell you what to do, but rather we will provide you with a variety of options and you can choose what works best for you. We will also provide you with a variety of resources and you can choose where to focus your time. Given that we cannot possibly cover everything, we will provide you with a framework for learning and you can fill in the gaps as you see fit. A key component of our success as adult learners is to gain the confidence to ask questions and problem-solve on our own.

Part I.

Introduction

1. Introducing R and RStudio

Questions

- What is R?
- Why use R?
- Why not use R?
- Why use RStudio and how does it differ from R?

Learning Objectives

- Know advantages of analyzing data in R
- Know advantages of using RStudio
- Be able to start RStudio on your computer
- Identify the panels of the RStudio interface
- Be able to customize the RStudio layout

1.1. Introduction

In this chapter, we will discuss the basics of R and RStudio, two essential tools in genomics data analysis. We will cover the advantages of using R and RStudio, how to set up RStudio, and the different panels of the RStudio interface.

1.2. What is R?

[R]([https://en.wikipedia.org/wiki/R_\(programming_language\)](https://en.wikipedia.org/wiki/R_(programming_language))) is a programming language and software environment designed for statistical computing and graphics. It is widely used by statisticians, data scientists, and researchers for data analysis and visualization.

Learning Objectives

R is an open-source language, which means it is free to use, modify, and distribute. Over the years, R has become particularly popular in the fields of genomics and bioinformatics, owing to its extensive libraries and powerful data manipulation capabilities.

The R language is a dialect of the S language, which was developed in the 1970s at Bell Laboratories. The first version of R was written by Robert Gentleman and Ross Ihaka and released in 1995 (see [this slide deck](#) for Ross Ihaka's take on R's history). Since then, R has been continuously developed by the R Core Team, a group of statisticians and computer scientists. The R Core Team releases a new version of R every year.



Figure 1.1.: Google trends showing the popularity of R over time based on Google searches

1.3. Why use R?

There are several reasons why R is a popular choice for data analysis, particularly in genomics and bioinformatics. These include:

1. **Open-source:** R is free to use and has a large community of developers who contribute to its growth and development. [What is “open-source”?](#)
2. **Extensive libraries:** There are thousands of R packages available for a wide range of tasks, including specialized packages for genomics and bioinformatics. These libraries have been extensively tested and are available for free.
3. **Data manipulation:** R has powerful data manipulation capabilities, making it easy (or at least possible) to clean, process, and analyze large datasets.
4. **Graphics and visualization:** R has excellent tools for creating high-quality graphics and visualizations that can be customized to meet the specific needs of your analysis. In most cases, graphics produced by R are publication-quality.
5. **Reproducible research:** R enables you to create reproducible research by recording your analysis in a script, which

Learning Objectives

can be easily shared and executed by others. In addition, R does not have a meaningful graphical user interface (GUI), which renders analysis in R much more reproducible than tools that rely on GUI interactions.

6. **Cross-platform:** R runs on Windows, Mac, and Linux (as well as more obscure systems).
7. **Interoperability with other languages:** R can interact with FORTRAN, C, and many other languages.
8. **Scalability:** R is useful for small and large projects.

I can develop code for analysis on my Mac laptop. I can then install the *same* code on our 20k core cluster and run it in parallel on 100 samples, monitor the process, and then update a database (for example) with R when complete.

1.4. Why not use R?

- R cannot do everything.
- R is not always the “best” tool for the job.
- R will *not* hold your hand. Often, it will *slap* your hand instead.
- The documentation can be opaque (but there is documentation).
- R can drive you crazy (on a good day) or age you prematurely (on a bad one).
- Finding the right package to do the job you want to do can be challenging; worse, some contributed packages are unreliable.}}
- R does not have a meaningfully useful graphical user interface (GUI).

1.5. R License and the Open Source Ideal

R is free (yes, totally free!) and distributed under GNU license. In particular, this license allows one to:

- Download the source code
- Modify the source code to your heart’s content

- Distribute the modified source code and even charge money for it, but you must distribute the modified source code under the original GNU license}}

This license means that R will always be available, will always be open source, and can grow organically without constraint.

1.6. RStudio

RStudio is an integrated development environment (IDE) for R. It provides a graphical user interface (GUI) for R, making it easier to write and execute R code. RStudio also provides several other useful features, including a built-in console, syntax-highlighting editor, and tools for plotting, history, debugging, workspace management, and workspace viewing. RStudio is available in both free and commercial editions; the commercial edition provides some additional features, including support for multiple sessions and enhanced debugging.

1.6.1. Getting started with RStudio

To get started with RStudio, you first need to install both R and RStudio on your computer. Follow these steps:

1. Download and install R from the [official R website](#).
2. Download and install RStudio from the [official RStudio website](#).
3. Launch RStudio. You should see the RStudio interface with four panels.

1.6.2. The RStudio Interface

RStudio's interface consists of four panels (see Figure 1.2):

- **Console** This panel displays the R console, where you can enter and execute R commands directly. The console also shows the output of your code, error messages, and other information.

Learning Objectives

- **Source** This panel is where you write and edit your R scripts. You can create new scripts, open existing ones, and run your code from this panel.
- **Environment** This panel displays your current workspace, including all variables, data objects, and functions that you have created or loaded in your R session.
- **Plots, Packages, Help, and Viewer** These panels display plots, installed packages, help files, and web content, respectively.

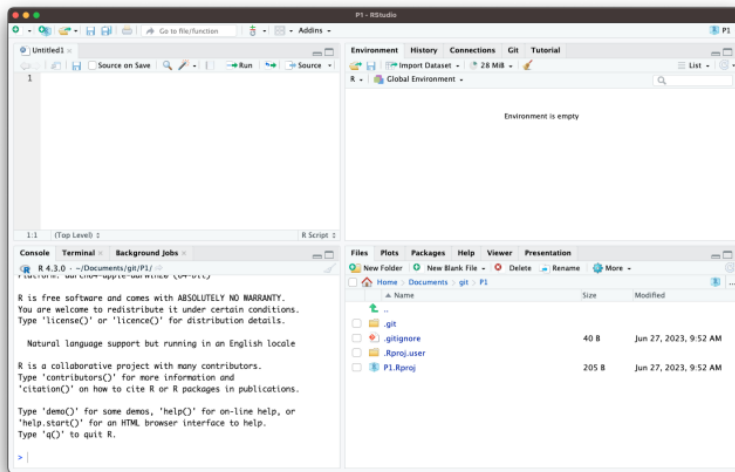


Figure 1.2.: The RStudio interface. In this layout, the **source** pane is in the upper left, the **console** is in the lower left, the **environment** panel is in the top right and the **viewer/help/files** panel is in the bottom right.

Do I need to use RStudio?

No. You can use R without RStudio. However, RStudio makes it easier to write and execute R code, and it provides several useful features that are not available in the basic R console. Note that the only part of RStudio that is actually interacting with R directly is the console. The other panels are simply providing a GUI that enhances the user experience.

Customizing the RStudio Interface

You can customize the layout of RStudio to suit your preferences. To do so, go to **Tools > Global Options > Appearance**.

Learning Objectives

ance. Here, you can change the theme, font size, and panel layout.

In summary, R and RStudio are powerful tools for genomics data analysis. By understanding the advantages of using R and RStudio and familiarizing yourself with the RStudio interface, you can efficiently analyze and visualize your data. In the following chapters, we will delve deeper into the functionality of R, Bioconductor, and various statistical methods to help you gain a comprehensive understanding of genomics data analysis.

2. R mechanics

2.1. Learning objectives

- Be able to start R and RStudio
- Learn to interact with the R console
- Know the difference between expressions and assignment
- Recognize valid and invalid R names
- Know how to access the R help system
- Know how to assign values to variables, find what is in R memory, and remove values from R memory

2.2. Starting R

How to start R depends a bit on the operating system (Mac, Windows, Linux) and interface. In this course, we will largely be using an Integrated Development Environment (IDE) called *RStudio*, but there is nothing to prohibit using R at the command line or in some other interface (and there are a few).

2.3. *RStudio*: A Quick Tour

The RStudio interface has multiple panes. All of these panes are simply for convenience except the “Console” panel, typically in the lower left corner (by default). The console pane contains the running R interface. If you choose to run R outside RStudio, the interaction will be *identical* to working in the console pane. This is useful to keep in mind as some environments, such as a computer cluster, encourage using R without RStudio.

- Panes
- Options

2. R mechanics

- Help
- Environment, History, and Files

2.4. Interacting with R

The only meaningful way of interacting with R is by typing into the R console. At the most basic level, anything that we type at the command line will fall into one of two categories:

1. Assignments

```
x = 1  
y <- 2
```

2. Expressions

```
1 + pi + sin(42)
```

```
[1] 3.225071
```

The assignment type is obvious because either the `<-` or `=` are used. Note that when we type expressions, R will return a result. In this case, the result of R evaluating `1 + pi + sin(42)` is 3.2250711.

The standard R prompt is a “>” sign. When present, R is waiting for the next expression or assignment. If a line is not a complete R command, R will continue the next line with a “+”. For example, typing the following with a “Return” after the second “+” will result in R giving back a “+” on the next line, a prompt to keep typing.

```
1 + pi +  
sin(3.7)
```

```
[1] 3.611757
```

R can be used as a glorified calculator by using R expressions. Mathematical operations include:

- Addition: +
- Subtraction: -

2. R mechanics

- Multiplication: `*`
- Division: `/`
- Exponentiation: `^`
- Modulo: `%%`

The `^` operator raises the number to its left to the power of the number to its right: for example `3^2` is 9. The modulo returns the remainder of the division of the number to the left by the number on its right, for example 5 modulo 3 or `5 %% 3` is 2.

2.4.1. Expressions

```
5 + 2
28 %% 3
3^2
5 + 4 * 4 + 4 ^ 4 / 10
```

Note that R follows order-of-operations and groupings based on parentheses.

```
5 + 4 / 9
(5 + 4) / 9
```

2.4.2. Assignment

While using R as a calculator is interesting, to do useful and interesting things, we need to assign *values* to *objects*. To create objects, we need to give it a name followed by the assignment operator `<-` (or, entirely equivalently, `=`) and the value we want to give it:

```
weight_kg <- 55
```

`<-` is the assignment operator. Assigns values on the right to objects on the left, it is like an arrow that points from the value to the object. Using an `=` is equivalent (in nearly all cases). Learn to use `<-` as it is good programming practice.

2. R mechanics

Objects can be given any name such as `x`, `current_temperature`, or `subject_id` (see below). You want your object names to be explicit and not too long. They cannot start with a number (`2x` is not valid but `x2` is). R is case sensitive (e.g., `weight_kg` is different from `Weight_kg`). There are some names that cannot be used because they represent the names of fundamental functions in R (e.g., `if`, `else`, `for`, see [here](#) for a complete list). In general, even if it's allowed, it's best to not use other function names, which we'll get into shortly (e.g., `c`, `T`, `mean`, `data`, `df`, `weights`). When in doubt, check the help to see if the name is already in use. It's also best to avoid dots (`.`) within a variable name as in `my.dataset`. It is also recommended to use nouns for variable names, and verbs for function names.

When assigning a value to an object, R does not print anything. You can force to print the value by typing the name:

```
weight_kg
```

```
[1] 55
```

Now that R has `weight_kg` in memory, which R refers to as the “global environment”, we can do arithmetic with it. For instance, we may want to convert this weight in pounds (weight in pounds is 2.2 times the weight in kg).

```
2.2 * weight_kg
```

```
[1] 121
```

We can also change a variable's value by assigning it a new one:

```
weight_kg <- 57.5  
2.2 * weight_kg
```

```
[1] 126.5
```

2. R mechanics

This means that assigning a value to one variable does not change the values of other variables. For example, let's store the animal's weight in pounds in a variable.

```
weight_lb <- 2.2 * weight_kg
```

and then change `weight_kg` to 100.

```
weight_kg <- 100
```

What do you think is the current content of the object `weight_lb`, 126.5 or 220?

You can see what objects (variables) are stored by viewing the Environment tab in Rstudio. You can also use the `ls()` function. You can remove objects (variables) with the `rm()` function. You can do this one at a time or remove several objects at once. You can also use the little broom button in your environment pane to remove everything from your environment.

```
ls()  
rm(weight_lb, weight_kg)  
ls()
```

What happens when you type the following, now?

```
weight_lb # oops! you should get an error because weight_lb no longer exists!
```

2.5. Rules for Names in R

R allows users to assign names to objects such as variables, functions, and even dimensions of data. However, these names must follow a few rules.

- Names may contain any combination of letters, numbers, underscore, and “.”
- Names may not start with numbers, underscore.
- R names are case-sensitive.

2. R mechanics

Examples of valid R names include:

```
pi
x
camelCaps
my_stuff
MY_Stuff
this.is.the.name.of.the.man
ABC123
abc1234asdf
.hi
```

2.6. Resources for Getting Help

There is extensive built-in help and documentation within R. A separate page contains a collection of [additional resources](#).

If the name of the function or object on which help is sought is known, the following approaches with the name of the function or object will be helpful. For a concrete example, examine the help for the `print` method.

```
help(print)
help('print')
?print
```

If the name of the function or object on which help is sought is *not* known, the following from within R will be helpful.

```
help.search('microarray')
RSiteSearch('microarray')
apropos('histogram')
```

There are also tons of online resources that Google will include in searches if online searching feels more appropriate.

I strongly recommend using `help("newfunction")` for all functions that are new or unfamiliar to you.

2.7. Further practice

If you are entirely new to R, you may want to complete an R tutorial to gain further experience with the basics of programming and R syntax.

One R-based system, [swirl](#), teaches you R programming and data science interactively, at your own pace and in the R console. Once you have R installed, you can install `swirl` and run it the following way:

```
install.packages("swirl")
library(swirl)
swirl()
```

Alternatively you can take the [try R](#) interactive class from Code School.

There are also many open and free resources and reference guides for R. Two examples are:

- [Quick-R](#): a quick online reference for data input, basic statistics and plots
- R reference card [PDF](#) by Tom Short
- Rstudio [cheatsheets](#)

2.8. Exercises

- Without using R, what are the values of the following?

```
mass <- 50           # mass?
age  <- 30           # age?
mass <- mass * 2     # mass?
age  <- age - 10     # age?
mass_index <- mass/age # massIndex?
```

- Use the R `help()` function to find information about the “`hist`” function. Follow up with running the example using `example("hist")`.
- Which of these is a valid R name for a variable?

2. *R mechanics*

```
x2  
2x  
.abc  
abc.123  
.123  
_my_value  
my_value  
my.value
```

3. The Very Basics

In this chapter, we’re going to get an introduction to the R language, so we can dive right into programming. We’re going to create a pair of virtual dice that can generate random numbers. No need to worry if you’re new to programming. We’ll return to many of the concepts here in more detail later.

To simulate a pair of dice, we need to break down each die into its essential features. A die can only show one of six numbers: 1, 2, 3, 4, 5, and 6. We can capture the die’s essential characteristics by saving these numbers as a group of values in the computer. Let’s save these numbers first and then figure out a way to “roll” our virtual die.

3.1. The R User Interface

The RStudio interface is simple. You type R code into the bottom line of the RStudio console pane and then click Enter to run it. The code you type is called a *command*, because it will command your computer to do something for you. The line you type it into is called the *command line*.

When you type a command at the prompt and hit Enter, your computer executes the command and shows you the results. Then RStudio displays a fresh prompt for your next command. For example, if you type `1 + 1` and hit Enter, RStudio will display:

```
> 1 + 1
[1] 2
>
```

You’ll notice that a `[1]` appears next to your result. R is just letting you know that this line begins with the first value in your result. Some commands return more than one value, and their results may

3. The Very Basics

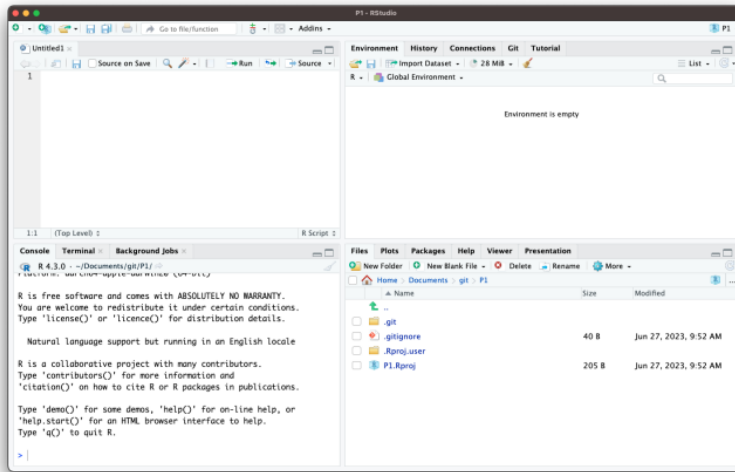


Figure 3.1.: Your computer does your bidding when you type R commands at the prompt in the bottom line of the console pane. Don't forget to hit the Enter key. When you first open RStudio, the console appears in the pane on your left, but you can change this with **File > Tools > Global Options** in the menu bar.

fill up multiple lines. For example, the command `100:130` returns 31 values; it creates a sequence of integers from 100 to 130. Notice that new bracketed numbers appear at the start of the second and third lines of output. These numbers just mean that the second line begins with the 14th value in the result, and the third line begins with the 25th value. You can mostly ignore the numbers that appear in brackets:

```
> 100:130
[1] 100 101 102 103 104 105 106 107 108 109 110 111 112
[14] 113 114 115 116 117 118 119 120 121 122 123 124 125
[25] 126 127 128 129 130
```

Tip

The colon operator (`:`) returns every integer between two integers. It is an easy way to create a sequence of numbers.

When do we compile?

In some languages, like C, Java, and FORTRAN, you have to compile your human-readable code into machine-readable code (often 1s and 0s) before you can run it. If you've programmed in such a language before, you may wonder whether

3. The Very Basics

you have to compile your R code before you can use it. The answer is no. R is a dynamic programming language, which means R automatically interprets your code as you run it.

If you type an incomplete command and press Enter, R will display a + prompt, which means R is waiting for you to type the rest of your command. Either finish the command or hit Escape to start over:

```
> 5 -  
+  
+ 1  
[1] 4
```

If you type a command that R doesn't recognize, R will return an error message. If you ever see an error message, don't panic. R is just telling you that your computer couldn't understand or do what you asked it to do. You can then try a different command at the next prompt:

```
> 3 % 5  
Error: unexpected input in "3 % 5"  
>
```

Tip

Whenever you get an error message in R, consider googling the error message. You'll often find that someone else has had the same problem and has posted a solution online. Simply cutting-and-pasting the error message into a search engine will often work

Once you get the hang of the command line, you can easily do anything in R that you would do with a calculator. For example, you could do some basic arithmetic:

```
2 * 3
```

```
[1] 6
```

3. The Very Basics

```
4 - 1
```

```
[1] 3
```

```
# this obeys order-of-operations  
6 / (4 - 1)
```

```
[1] 2
```

Tip

R treats the hashtag character, #, in a special way; R will not run anything that follows a hashtag on a line. This makes hashtags very useful for adding comments and annotations to your code. Humans will be able to read the comments, but your computer will pass over them. The hashtag is known as the *commenting symbol* in R.

Cancelling commands

Some R commands may take a long time to run. You can cancel a command once it has begun by pressing ctrl + c or by clicking the “stop sign” if it is available in Rstudio. Note that it may also take R a long time to cancel the command.

3.1.1. An exercise

That’s the basic interface for executing R code in RStudio. Think you have it? If so, try doing these simple tasks. If you execute everything correctly, you should end up with the same number that you started with:

1. Choose any number and add 2 to it.
2. Multiply the result by 3.
3. Subtract 6 from the answer.
4. Divide what you get by 3.

3. The Very Basics

```
10 + 2
```

```
[1] 12
```

```
12 * 3
```

```
[1] 36
```

```
36 - 6
```

```
[1] 30
```

```
30 / 3
```

```
[1] 10
```

3.2. Objects

Now that you know how to use R, let's use it to make a virtual die. The `:` operator from a couple of pages ago gives you a nice way to create a group of numbers from one to six. The `:` operator returns its results as a **vector** (we are going to work with vectors in more detail), a one-dimensional set of numbers:

```
1:6  
## 1 2 3 4 5 6
```

That's all there is to how a virtual die looks! But you are not done yet. Running `1:6` generated a vector of numbers for you to see, but it didn't save that vector anywhere for later use. If we want to use those numbers again, we'll have to ask your computer to save them somewhere. You can do that by creating an R *object*.

R lets you save data by storing it inside an R object. What is an object? Just a name that you can use to call up stored data. For example, you can save data into an object like `a` or `b`. Wherever R encounters the object, it will replace it with the data saved inside, like so:

3. The Very Basics

```
a <- 1  
a
```

```
[1] 1
```

```
a + 2
```

```
[1] 3
```

What just happened?

1. To create an R object, choose a name and then use the less-than symbol, <, followed by a minus sign, -, to save data into it. This combination looks like an arrow, <-. R will make an object, give it your name, and store in it whatever follows the arrow. So `a <- 1` stores 1 in an object named a.
2. When you ask R what's in a, R tells you on the next line.
3. You can use your object in new R commands, too. Since a previously stored the value of 1, you're now adding 1 to 2.

Assignment vs expressions

Everything that you type into the R console can be assigned to one of two categories:

- Assignments
- Expressions

An expression is a command that tells R to do something. For example, `1 + 2` is an expression that tells R to add 1 and 2. When you type an expression into the R console, R will evaluate the expression and return the result. For example, if you type `1 + 2` into the R console, R will return 3. Expressions can have “side effects” but they don't explicitly result in anything being added to R memory.

```
5 + 2
```

3. The Very Basics

```
[1] 7
```

```
28 %% 3
```

```
[1] 1
```

```
3^2
```

```
[1] 9
```

```
5 + 4 * 4 + 4 ^ 4 / 10
```

```
[1] 46.6
```

While using R as a calculator is interesting, to do useful and interesting things, we need to assign values to objects. To create objects, we need to give it a name followed by the assignment operator `<-` (or, entirely equivalently, `=`) and the value we want to give it:

```
weight_kg <- 55
```

So, for another example, the following code would create an object named `die` that contains the numbers one through six. To see what is stored in an object, just type the object's name by itself:

```
die <- 1:6  
die
```

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

When you create an object, the object will appear in the environment pane of RStudio, as shown in Figure 3.2. This pane will show you all of the objects you've created since opening RStudio.

You can name an object in R almost anything you want, but there are a few rules. First, a name cannot start with a number. Second, a name cannot use some special symbols, like `^`, `!`, `$`, `@`, `+`, `-`, `/`, or `*`:

3. The Very Basics

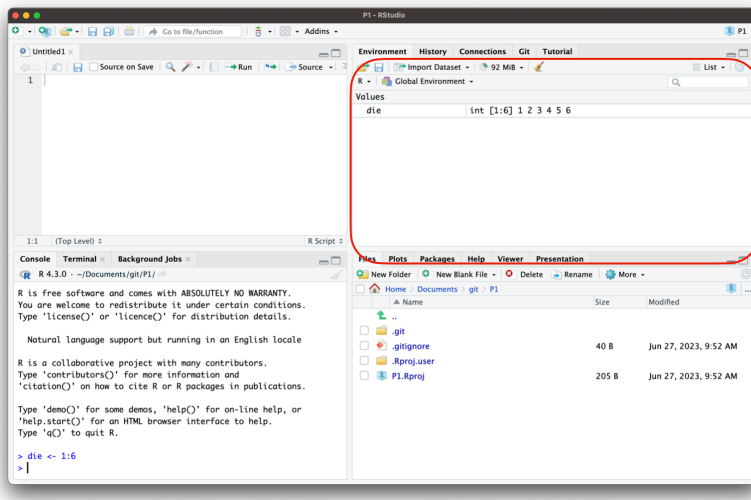


Figure 3.2.: Assignment creates an object in the environment pane.

Good names	Names that cause errors
a	1trial
b	\$
FOO	^mean
my_var	2nd
.day	!bad

⚠ Capitalization matters

R is case-sensitive, so name and Name will refer to different objects:

```
> Name = 0
> Name + 1
[1] 1
> name + 1
Error: object 'name' not found
```

The error above is a common one!

Finally, R will overwrite any previous information stored in an object without asking you for permission. So, it is a good idea to *not* use names that are already taken:

```
my_number <- 1
my_number
```

```
[1] 1
```

26

```
my_number <- 999
my_number
```

```
[1] 999
```

You can see which object names you have already used with the function `ls()`:

3. The Very Basics

```
ls()
```

Your environment will contain different names than mine, because you have probably created different objects.

You can also see which names you have used by examining RStudio's environment pane.

We now have a virtual die that is stored in the computer's memory and which has a name that we can use to refer to it. You can access it whenever you like by typing the word *die*.

So what can you do with this die? Quite a lot. R will replace an object with its contents whenever the object's name appears in a command. So, for example, you can do all sorts of math with the die. Math isn't so helpful for rolling dice, but manipulating sets of numbers will be your stock and trade as a data scientist. So let's take a look at how to do that:

```
die - 1
```

```
[1] 0 1 2 3 4 5
```

```
die / 2
```

```
[1] 0.5 1.0 1.5 2.0 2.5 3.0
```

```
die * die
```

```
[1] 1 4 9 16 25 36
```

R uses *element-wise execution* when working with a *vector* like *die*. When you manipulate a set of numbers, R will apply the same operation to each element in the set. So for example, when you run *die - 1*, R subtracts one from each element of *die*.

When you use two or more vectors in an operation, R will line up the vectors and perform a sequence of individual operations. For example, when you run *die * die*, R lines up the two *die* vectors and then multiplies the first element of vector 1 by the first element

3. The Very Basics

of vector 2. R then multiplies the second element of vector 1 by the second element of vector 2, and so on, until every element has been multiplied. The result will be a new vector the same length as the first two {Figure 3.3}.

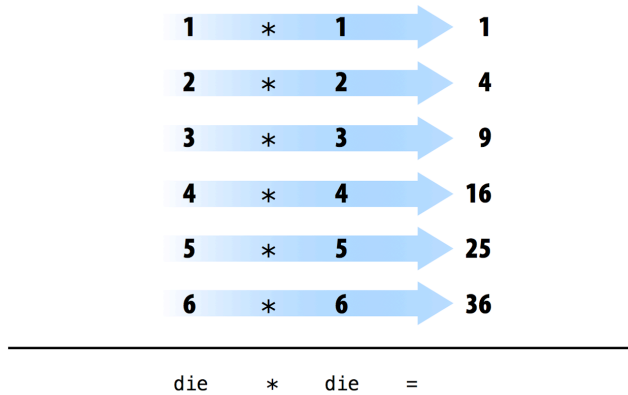


Figure 3.3.: “When R performs element-wise execution, it matches up vectors and then manipulates each pair of elements independently.”

If you give R two vectors of unequal lengths, R will repeat the shorter vector until it is as long as the longer vector, and then do the math, as shown in Figure 3.4. This isn’t a permanent change—the shorter vector will be its original size after R does the math. If the length of the short vector does not divide evenly into the length of the long vector, R will return a warning message. This behavior is known as *vector recycling*, and it helps R do element-wise operations:

```
1:2
```

```
[1] 1 2
```

```
1:4
```

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

```
die
```

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

3. The Very Basics

```
die + 1:2
```

```
[1] 2 4 4 6 6 8
```

```
die + 1:4
```

Warning in `die + 1:4`: longer object length is not a multiple of shorter object length

```
[1] 2 4 6 8 6 8
```

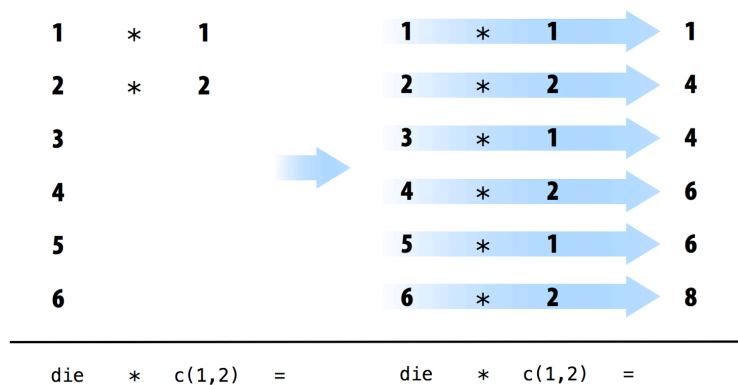


Figure 3.4.: “R will repeat a short vector to do element-wise operations with two vectors of uneven lengths.”

Element-wise operations are a very useful feature in R because they manipulate groups of values in an orderly way. When you start working with data sets, element-wise operations will ensure that values from one observation or case are only paired with values from the same observation or case. Element-wise operations also make it easier to write your own programs and functions in R.

! Element-wise operations are not matrix operations

It is important to know that operations with vectors are not the same that you might expect if you are expecting R to perform “matrix” operations. R can do inner multiplication with the `%*%` operator and outer multiplication with the `%o%` operator:

3. The Very Basics

```
# Inner product (1*1 + 2*2 + 3*3 + 4*4 + 5*5 + 6*6)
die %*% die
# Outer product
die %0% die
```

Now that you can do math with your die object, let's look at how you could “roll” it. Rolling your die will require something more sophisticated than basic arithmetic; you'll need to randomly select one of the die's values. And for that, you will need a *function*.

3.3. Functions

R has many functions and puts them all at our disposal. We can use functions to do simple and sophisticated tasks. For example, we can round a number with the `round` function, or calculate its factorial with the `factorial` function. Using a function is pretty simple. Just write the name of the function and then the data you want the function to operate on in parentheses:

```
round(3.1415)
```

```
[1] 3
```

```
factorial(3)
```

```
[1] 6
```

The data that you pass into the function is called the function's *argument*. The argument can be raw data, an R object, or even the results of another R function. In this last case, R will work from the innermost function to the outermost [Figure 3.5](#).

```
mean(1:6)
```

```
[1] 3.5
```

3. The Very Basics

```
mean(die)
```

```
[1] 3.5
```

```
round(mean(die))
```

```
[1] 4
```

```
round(mean(die))  
↓  
round(mean(1:6))  
↓  
round(3.5)  
↓  
4
```

Figure 3.5: “When you link functions together, R will resolve them from the innermost operation to the outermost. Here R first looks up `die`, then calculates the mean of one through six, then rounds the mean.”

Returning to our die, we can use the `sample` function to randomly select one of the die’s values; in other words, the `sample` function can simulate rolling the die.

The `sample` function takes *two* arguments: a vector named `x` and a number named `size`. `sample` will return `size` elements from the vector:

```
sample(x = 1:4, size = 2)
```

```
[1] 3 4
```

To roll your die and get a number back, set `x` to `die` and sample one element from it. You’ll get a new (maybe different) number each time you roll it:

```
sample(x = die, size = 1)
```

```
[1] 1
```


3. The Very Basics

```
sample(x = die, size = 1)
```

```
[1] 3
```

```
sample(x = die, size = 1)
```

```
[1] 2
```

Many R functions take multiple arguments that help them do their job. You can give a function as many arguments as you like as long as you separate each argument with a comma.

You may have noticed that I set `die` and `1` equal to the names of the arguments in `sample`, `x` and `size`. Every argument in every R function has a name. You can specify which data should be assigned to which argument by setting a name equal to data, as in the preceding code. This becomes important as you begin to pass multiple arguments to the same function; names help you avoid passing the wrong data to the wrong argument. However, using names is optional. You will notice that R users do not often use the name of the first argument in a function. So you might see the previous code written as:

```
sample(die, size = 1)
```

```
[1] 3
```

Often, the name of the first argument is not very descriptive, and it is usually obvious what the first piece of data refers to anyways.

But how do you know which argument names to use? If you try to use a name that a function does not expect, you will likely get an error:

```
round(3.1415, corners = 2)
## Error in round(3.1415, corners = 2) : unused argument(s) (corners = 2)
```

3. The Very Basics

If you're not sure which names to use with a function, you can look up the function's arguments with `args`. To do this, place the name of the function in the parentheses behind `args`. For example, you can see that the `round` function takes two arguments, one named `x` and one named `digits`:

```
args(round)
```

```
function (x, digits = 0)
NULL
```

Did you notice that `args` shows that the `digits` argument of `round` is already set to 0? Frequently, an R function will take optional arguments like `digits`. These arguments are considered optional because they come with a default value. You can pass a new value to an optional argument if you want, and R will use the default value if you do not. For example, `round` will round your number to 0 digits past the decimal point by default. To override the default, supply your own value for `digits`:

```
round(3.1415)
```

```
[1] 3
```

```
round(3.1415, digits = 2)
```

```
[1] 3.14
```

```
# pi happens to be a built-in value in R
pi
```

```
[1] 3.141593
```

```
round(pi)
```

```
[1] 3
```

3. The Very Basics

You should write out the names of each argument after the first one or two when you call a function with multiple arguments. Why? First, this will help you and others understand your code. It is usually obvious which argument your first input refers to (and sometimes the second input as well). However, you'd need a large memory to remember the third and fourth arguments of every R function. Second, and more importantly, writing out argument names prevents errors.

If you do not write out the names of your arguments, R will match your values to the arguments in your function by order. For example, in the following code, the first value, `die`, will be matched to the first argument of `sample`, which is named `x`. The next value, `1`, will be matched to the next argument, `size`:

```
sample(die, 1)
```

```
[1] 5
```

As you provide more arguments, it becomes more likely that your order and R's order may not align. As a result, values may get passed to the wrong argument. Argument names prevent this. R will always match a value to its argument name, no matter where it appears in the order of arguments:

```
sample(size = 1, x = die)
```

```
[1] 6
```

3.3.1. Sample with Replacement

If you set `size = 2`, you can *almost* simulate a pair of dice. Before we run that code, think for a minute why that might be the case. `sample` will return two numbers, one for each die:

```
sample(die, size = 2)
```

```
[1] 3 2
```

3. The Very Basics

I said this “almost” works because this method does something funny. If you use it many times, you’ll notice that the second die never has the same value as the first die, which means you’ll never roll something like a pair of threes or snake eyes. What is going on?

By default, `sample` builds a sample *without replacement*. To see what this means, imagine that `sample` places all of the values of `die` in a jar or urn. Then imagine that `sample` reaches into the jar and pulls out values one by one to build its sample. Once a value has been drawn from the jar, `sample` sets it aside. The value doesn’t go back into the jar, so it cannot be drawn again. So if `sample` selects a six on its first draw, it will not be able to select a six on the second draw; six is no longer in the jar to be selected. Although `sample` creates its sample electronically, it follows this seemingly physical behavior.

One side effect of this behavior is that each draw depends on the draws that come before it. In the real world, however, when you roll a pair of dice, each die is independent of the other. If the first die comes up six, it does not prevent the second die from coming up six. In fact, it doesn’t influence the second die in any way whatsoever. You can recreate this behavior in `sample` by adding the argument `replace = TRUE`:

```
sample(die, size = 2, replace = TRUE)
```

```
[1] 2 2
```

The argument `replace = TRUE` causes `sample` to sample *with replacement*. Our jar example provides a good way to understand the difference between sampling with replacement and without. When `sample` uses replacement, it draws a value from the jar and records the value. Then it puts the value back into the jar. In other words, `sample` *replaces* each value after each draw. As a result, `sample` may select the same value on the second draw. Each value has a chance of being selected each time. It is as if every draw were the first draw.

Sampling with replacement is an easy way to create *independent random samples*. Each value in your sample will be a sample of size one that is independent of the other values. This is the correct way to simulate a pair of dice:

3. *The Very Basics*

```
sample(die, size = 2, replace = TRUE)
```

```
[1] 4 3
```

Congratulate yourself; you’ve just run your first simulation in R! You now have a method for simulating the result of rolling a pair of dice. If you want to add up the dice, you can feed your result straight into the `sum` function:

```
dice <- sample(die, size = 2, replace = TRUE)  
dice
```

```
[1] 2 3
```

```
sum(dice)
```

```
[1] 5
```

What would happen if you call `dice` multiple times? Would R generate a new pair of dice values each time? Let’s give it a try:

```
dice
```

```
[1] 2 3
```

```
dice
```

```
[1] 2 3
```

```
dice
```

```
[1] 2 3
```

3. The Very Basics

The name `dice` refers to a *vector* of two numbers. Calling more than once does not change the value. Each time you call `dice`, R will show you the result of that one time you called `sample` and saved the output to `dice`. R won't rerun `sample(die, 2, replace = TRUE)` to create a new roll of the dice. Once you save a set of results to an R object, those results do not change.

However, it *would* be convenient to have an object that can re-roll the dice whenever you call it. You can make such an object by writing your own R function.

3.4. Writing Your Own Functions

To recap, you already have working R code that simulates rolling a pair of dice:

```
die <- 1:6
dice <- sample(die, size = 2, replace = TRUE)
sum(dice)
```

```
[1] 5
```

You can retype this code into the console anytime you want to re-roll your dice. However, this is an awkward way to work with the code. It would be easier to use your code if you wrapped it into its own function, which is exactly what we'll do now. We're going to write a function named `roll` that you can use to roll your virtual dice. When you're finished, the function will work like this: each time you call `roll()`, R will return the sum of rolling two dice:

```
roll()
## 8

roll()
## 3

roll()
## 7
```

3. The Very Basics

Functions may seem mysterious or fancy, but they are *just another type of R object*. Instead of containing data, they contain code. This code is stored in a special format that makes it easy to reuse the code in new situations. You can write your own functions by recreating this format.

3.4.1. The Function Constructor

Every function in R has three basic parts: a name, a body of code, and a set of arguments. To make your own function, you need to replicate these parts and store them in an R object, which you can do with the `function()` function. To do this, call `function()` and follow it with a pair of braces, `{}`:

```
my_function <- function() {}
```

This function, as written, doesn't do anything (yet). However, it is a valid function. You can call it by typing its name followed by an open and closed parenthesis:

```
my_function()
```

NULL

`function` will build a function out of whatever R code you place between the braces. For example, you can turn your dice code into a function by calling:

```
roll <- function() {  
  die <- 1:6  
  dice <- sample(die, size = 2, replace = TRUE)  
  sum(dice)  
}
```

Indentation and readability

Notice each line of code between the braces is indented. This makes the code easier to read but has no impact on how the

3. The Very Basics

code runs. R ignores spaces and line breaks and executes one complete expression at a time. Note that in other languages like python, spacing is extremely important and part of the language.

Just hit the Enter key between each line after the first brace, {. R will wait for you to type the last brace, }, before it responds.

Don't forget to save the output of function to an R object. This object will become your new function. To use it, write the object's name followed by an open and closed parenthesis:

```
roll()
```

```
[1] 8
```

You can think of the parentheses as the “trigger” that causes R to run the function. If you type in a function's name *without* the parentheses, R will show you the code that is stored inside the function. If you type in the name *with* the parentheses, R will run that code:

```
roll
```

```
function() {  
  die <- 1:6  
  dice <- sample(die, size = 2, replace = TRUE)  
  sum(dice)  
}
```

```
roll()
```

```
[1] 9
```

The code that you place inside your function is known as the *body* of the function. When you run a function in R, R will execute all of the code in the body and then return the result of the last line of code. If the last line of code doesn't return a value, neither will your function, so you want to ensure that your final line of code returns

3. The Very Basics

a value. One way to check this is to think about what would happen if you ran the body of code line by line in the command line. Would R display a result after the last line, or would it not?

Here's some code that would display a result:

```
dice
1 + 1
sqrt(2)
```

And here's some code that would not:

```
dice <- sample(die, size = 2, replace = TRUE)
two <- 1 + 1
a <- sqrt(2)
```

Again, this is just showing the distinction between expressions and assignments.

3.5. Arguments

What if we removed one line of code from our function and changed the name `die` to `bones` (just a name—don't think of it as important), like this?

```
roll2 <- function() {
  dice <- sample(bones, size = 2, replace = TRUE)
  sum(dice)
}
```

Now I'll get an error when I run the function. The function **needs** the object `bones` to do its job, but there is no object named `bones` to be found (you can check by typing `ls()` which will show you the names in the environment, or memory).

```
roll2()
## Error in sample(bones, size = 2, replace = TRUE) :
## object 'bones' not found
```

3. The Very Basics

You can supply bones when you call `roll2` if you make bones an argument of the function. To do this, put the name bones in the parentheses that follow function when you define `roll2`:

```
roll2 <- function(bones) {  
  dice <- sample(bones, size = 2, replace = TRUE)  
  sum(dice)  
}
```

Now `roll2` will work as long as you supply bones when you call the function. You can take advantage of this to roll different types of dice each time you call `roll2`.

Remember, we're rolling pairs of dice:

```
roll2(bones = 1:4)
```

```
[1] 6
```

```
roll2(bones = 1:6)
```

```
[1] 7
```

```
roll2(1:20)
```

```
[1] 26
```

Notice that `roll2` will still give an error if you do not supply a value for the bones argument when you call `roll2`:

```
roll2()  
## Error in sample(bones, size = 2, replace = TRUE) :  
##   argument "bones" is missing, with no default
```

You can prevent this error by giving the bones argument a default value. To do this, set bones equal to a value when you define `roll2`:

3. The Very Basics

```
roll2 <- function(bones = 1:6) {  
  dice <- sample(bones, size = 2, replace = TRUE)  
  sum(dice)  
}
```

Now you can supply a new value for bones if you like, and roll2 will use the default if you do not:

```
roll2()
```

```
[1] 5
```

You can give your functions as many arguments as you like. Just list their names, separated by commas, in the parentheses that follow function. When the function is run, R will replace each argument name in the function body with the value that the user supplies for the argument. If the user does not supply a value, R will replace the argument name with the argument's default value (if you defined one).

To summarize, function helps you construct your own R functions. You create a body of code for your function to run by writing code between the braces that follow function. You create arguments for your function to use by supplying their names in the parentheses that follow function. Finally, you give your function a name by saving its output to an R object, as shown in Figure 3.6.

Once you've created your function, R will treat it like every other function in R. Think about how useful this is. Have you ever tried to create a new Excel option and add it to Microsoft's menu bar? Or a new slide animation and add it to Powerpoint's options? When you work with a programming language, you can do these types of things. As you learn to program in R, you will be able to create new, customized, reproducible tools for yourself whenever you like.

3.6. Scripts

Scripts are code that are saved for later reuse or editing. An R script is just a plain text file that you save R code in. You can open an

3. The Very Basics

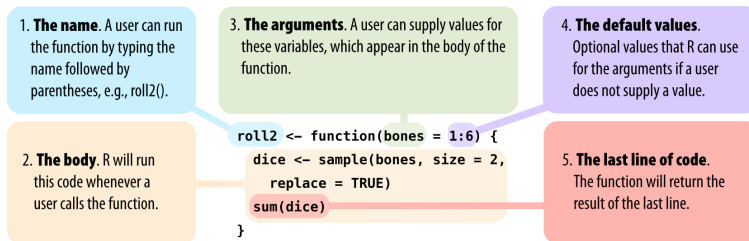


Figure 3.6.: “Every function in R has the same parts, and you can use `function` to create these parts. Assign the result to a name, so you can call the function later.”

R script in RStudio by going to **File > New File > R script** in the menu bar. RStudio will then open a fresh script above your console pane, as shown in Figure 3.7.

I strongly encourage you to write and edit all of your R code in a script before you run it in the console. Why? This habit creates a reproducible record of your work. When you’re finished for the day, you can save your script and then use it to rerun your entire analysis the next day. Scripts are also very handy for editing and proofreading your code, and they make a nice copy of your work to share with others. To save a script, click the scripts pane, and then go to **File > Save As** in the menu bar.

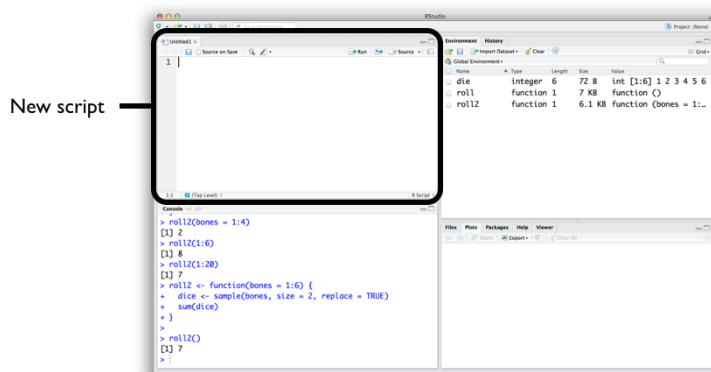


Figure 3.7.: “When you open an R Script (File > New File > R Script in the menu bar), RStudio creates a fourth pane (or puts a new tab in the existing pane) above the console where you can write and edit your code.”

RStudio comes with many built-in features that make it easy to work with scripts. First, you can automatically execute a line of code in a script by clicking the Run button at the top of the editor panel.

R will run whichever line of code your cursor is on. If you have a whole section highlighted, R will run the highlighted code. Alternatively, you can run the entire script by clicking the Source but-

3. The Very Basics

ton. Don't like clicking buttons? You can use Control + Return as a shortcut for the Run button. On Macs, that would be Command + Return.

If you're not convinced about scripts, you soon will be. It becomes a pain to write multi-line code in the console's single-line command line. Let's avoid that headache and open your first script now before we move to the next chapter.

Tip

Extract function

RStudio comes with a tool that can help you build functions. To use it, highlight the lines of code in your R script that you want to turn into a function. Then click Code > Extract Function in the menu bar. RStudio will ask you for a function name to use and then wrap your code in a function call. It will scan the code for undefined variables and use these as arguments.

You may want to double-check RStudio's work. It assumes that your code is correct, so if it does something surprising, you may have a problem in your code.

3.7. Summary

You've covered a lot of ground already. You now have a virtual die stored in your computer's memory, as well as your own R function that rolls a pair of dice. You've also begun speaking the R language.

As you've seen, R is a language that you can use to talk to your computer. You write commands in R and run them at the command line for your computer to read. Your computer will sometimes talk back—for example, when you commit an error—but it usually just does what you ask and then displays the result.

The two most important components of the R language are objects, which store data, and functions, which manipulate data. R also uses a host of operators like +, -, *, /, and <- to do basic tasks. As a data scientist, you will use R objects to store data in your computer's

3. *The Very Basics*

memory, and you will use functions to automate tasks and do complicated calculations. We will examine objects in more depth later in [Project 2: Playing Cards] and dig further into functions in [Project 3: Slot Machine]. The vocabulary you have developed here will make each of those projects easier to understand. However, we're not done with your dice yet.

In [Packages and Help Pages](#), you'll run some simulations on your dice and build your first graphs in R. You'll also look at two of the most useful components of the R language: R *packages*, which are collections of functions written by R's talented community of developers, and R documentation, which is a collection of help pages built into R that explains every function and data set in the language.

Part II.

Overview of R Data Structures

Welcome to the section on R data structures! As you begin your journey in learning R, it is essential to understand the fundamental building blocks of this powerful programming language. R offers a variety of data structures to store and manipulate data, each with its unique properties and capabilities. In this section, we will cover the core data structures in R, including:

- Vectors
- Matrices
- Lists
- Data.frames

By the end of this section, you will have a solid understanding of these data structures, and you will be able to choose and utilize the appropriate data structure for your specific data manipulation and analysis tasks.

In each chapter, we will delve into the properties and usage of each data structure, starting with their definitions and moving on to their practical applications. We will provide examples, exercises, and active learning approaches to help you better understand and apply these concepts in your work.

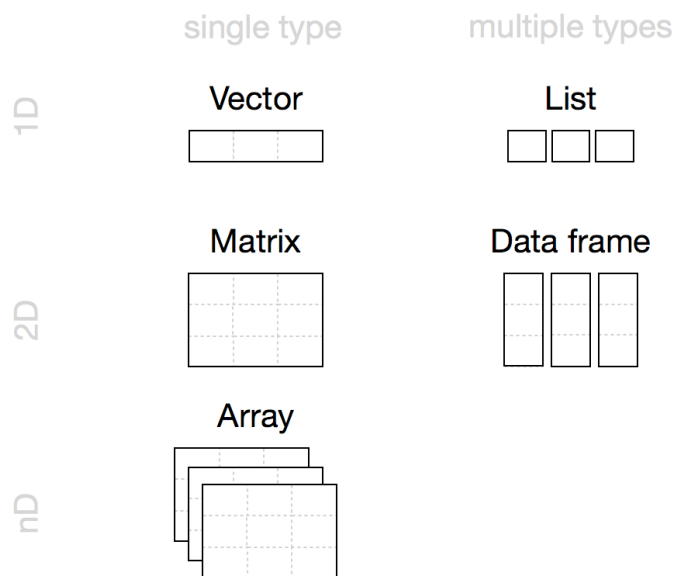


Figure 3.8.: A pictorial representation of R's most common data structures are vectors, matrices, arrays, lists, and dataframes. Figure from [Hands-on Programming with R](#).

Chapter overview

- **Vectors** : In this chapter, we will introduce you to the simplest data structure in R, the vector. We will cover how to create, access, and manipulate vectors, as well as discuss their unique properties and limitations.
- **Matrices** Next, we will explore matrices, which are two-dimensional data structures that extend vectors. You will learn how to create, access, and manipulate matrices, and understand their usefulness in mathematical operations and data organization.
- **Lists** The third chapter will focus on lists, a versatile data structure that can store elements of different types and sizes. We will discuss how to create, access, and modify lists, and demonstrate their flexibility in handling complex data structures.
- **Data.frames** Finally, we will examine `data.frames`, a widely-used data structure for organizing and manipulating tabular data. You will learn how to create, access, and manipulate `data.frames`, and understand their advantages over other data structures for data analysis tasks.
- **Arrays** While we will not focus directly on the array data type, which are multidimensional data structures that extend matrices, they are very similar to matrices, but with a third dimension.

As you progress through these chapters, we encourage you to practice the examples and exercises provided, engage in discussion, and collaborate with your peers to deepen your understanding of R data structures. This solid foundation will serve as the basis for more advanced data manipulation, analysis, and visualization techniques in R.

4. Vectors

4.1. What is a Vector?

A vector is the simplest and most basic data structure in R. It is a one-dimensional, ordered collection of elements, where all the elements are of the same data type. Vectors can store various types of data, such as numeric, character, or logical values.

In this chapter, we will provide a comprehensive overview of vectors, including how to create, access, and manipulate them. We will also discuss some unique properties and rules associated with vectors, and explore their applications in data analysis tasks.

4.2. Creating Vectors

In this section, we will cover different ways of creating vectors. The most common method for creating vectors is by using the `c()` function, which stands for “concatenate.” The `c()` function combines its arguments into a single vector.

Here’s an example of creating a numeric vector:

```
numeric_vector <- c(3, 5, 7, 9)
print(numeric_vector)
```

```
[1] 3 5 7 9
```

Similarly, you can create a character vector and a logical vector:

```
character_vector <- c("apple", "banana", "cherry")
print(character_vector)
```

4. Vectors

```
[1] "apple" "banana" "cherry"
```

A logical vector is a one-dimensional array of logical values in R. Logical values are either TRUE or FALSE, which represent the outcomes of logical expressions or conditions. Logical vectors are used for a variety of purposes, such as filtering data, controlling the flow of a program, or performing element-wise comparisons between vectors.

In R, you can create a logical vector using the `c()` function by specifying logical values as its arguments:

```
logical_vector <- c(TRUE, FALSE, TRUE, FALSE)
```

Note

The values TRUE and FALSE are reserved words in R, which means that they cannot be used as variable names. If you try to assign a value to TRUE or FALSE, R will throw an error. Also, notice that there are no quotation marks around TRUE and FALSE. The value “TRUE” with quotes is a character vector of length 1.

Logical vectors can also be generated by applying logical or comparison operators on other vectors or values:

```
numeric_vector <- c(1, 2, 3, 4, 5)
logical_vector <- numeric_vector > 3
```

In this example, `logical_vector` will contain the logical outcomes of the comparison `numeric_vector > 3`, resulting in a logical vector with the values (FALSE, FALSE, FALSE, TRUE, TRUE).

```
print(logical_vector)
```

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

You can also generate sequences using the `seq()` function or the `:` operator:

4. Vectors

```
sequence1 <- seq(from = 1, to = 10, by = 2)
print(sequence1)
```

```
[1] 1 3 5 7 9
```

```
sequence2 <- 1:5
print(sequence2)
```

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

The `rep()` function allows you to create a vector by repeating elements:

```
repeated_vector <- rep(1:3, times = 2)
print(repeated_vector)
```

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

4.3. Accessing Vector Elements

You can access and extract elements from vectors using indexing. In R, indexing starts at 1. To access a single element, use square brackets `[]` and provide the index of the desired element:

```
numeric_vector <- c(3, 5, 7, 9)
third_element <- numeric_vector[3]
print(third_element)
```

```
[1] 7
```

To access multiple elements, provide a vector of indices:

```
elements <- numeric_vector[c(1, 3)]
print(elements)
```

```
[1] 3 7
```

4. Vectors

You can also modify and update vector elements using indexing:

```
numeric_vector[2] <- 10  
print(numeric_vector)
```

```
[1] 3 10 7 9
```

```
\begin{tikzpicture}  
\foreach \x in {0,1,2,3,4,5,6,7,8,9}  
\draw (1,0) rectangle (\x,1);  
\end{tikzpicture}
```



4.4. Vector Operations

You can perform various vector operations, such as arithmetic, logical, and comparison operations. The operations are performed element-wise, which means that the operation is applied to each corresponding pair of elements in the input vectors.

Here are some examples of arithmetic operations:

```
vector1 <- c(1, 2, 3)  
vector2 <- c(4, 5, 6)  
  
sum_vectors <- vector1 + vector2  
print(sum_vectors)
```

```
[1] 5 7 9
```

```
product_vectors <- vector1 * vector2  
print(product_vectors)
```

```
[1] 4 10 18
```

4. Vectors

Logical and comparison operations also work element-wise:

```
vector1 <- c(1, 2, 3)
vector2 <- c(3, 2, 1)

equal_elements <- vector1 == vector2
print(equal_elements)
```

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

```
greater_elements <- vector1 > vector2
print(greater_elements)
```

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

Operations on a single vector are typically done element-by-element. For example, we can add 2 to a vector, 2 is added to each element of the vector and a new vector of the same length is returned.

```
x = 1:10
x + 2
```

```
[1] 3 4 5 6 7 8 9 10 11 12
```

If the operation involves two vectors, the following rules apply. If the vectors are the same length: R simply applies the operation to each pair of elements.

```
x + x
```

```
[1] 2 4 6 8 10 12 14 16 18 20
```

If the vectors are different lengths, but one length a multiple of the other, R reuses the shorter vector as needed.

4. Vectors

```
x = 1:10
y = c(1,2)
x * y
```

```
[1] 1 4 3 8 5 12 7 16 9 20
```

If the vectors are different lengths, but one length *not* a multiple of the other, R reuses the shorter vector as needed *and* delivers a warning.

```
x = 1:10
y = c(2,3,4)
x * y
```

Warning in x * y: longer object length is not a multiple of shorter object length

```
[1] 2 6 12 8 15 24 14 24 36 20
```

Typical operations include multiplication (“*”), addition, subtraction, division, exponentiation (“^”), but many operations in R operate on vectors and are then called “vectorized”.

4.5. Named Vectors

Named vectors are vectors with labels or names assigned to their elements. These names can be used to access and manipulate the elements in a more meaningful way.

To create a named vector, use the `names()` function:

```
fruit_prices <- c(0.5, 0.75, 1.25)
names(fruit_prices) <- c("apple", "banana", "cherry")
print(fruit_prices)
```

```
apple banana cherry
0.50    0.75    1.25
```

4. Vectors

You can also access and modify elements using their names:

```
banana_price <- fruit_prices["banana"]  
print(banana_price)
```

```
banana  
0.75
```

```
fruit_prices["apple"] <- 0.6  
print(fruit_prices)
```

```
apple banana cherry  
0.60    0.75    1.25
```

4.6. Vector Recycling and Recycling Rule

Vector recycling is a unique feature of R that allows for element-wise operations on vectors of different lengths. When performing operations on vectors of unequal length, R will recycle the shorter vector by repeating its elements until it matches the length of the longer vector.

Here's an example of vector recycling:

```
vector1 <- c(1, 2, 3)  
vector2 <- c(4, 5)  
  
sum_vectors <- vector1 + vector2
```

Warning in vector1 + vector2: longer object length is not a multiple of shorter object length

```
print(sum_vectors)
```

```
[1] 5 7 7
```


4. Vectors

In this example, `vector2` is shorter than `vector1`, so its elements are recycled to match the length of `vector1`. The result is equivalent to adding `vector1` to `c(4, 5, 4)`.

Be aware of the recycling rule when working with vectors of different lengths, as it may lead to unexpected results if you're not careful.

4.7. Active Learning Exercises

Now that we have covered the basics of vectors, let's put your knowledge into practice with some active learning exercises.

In R, even a single value is a vector with `length=1`.

```
z = 1
z
```

```
[1] 1
```

```
length(z)
```

```
[1] 1
```

In the code above, we “assigned” the value 1 to the variable named `z`. Typing `z` by itself is an “expression” that returns a result which is, in this case, the value that we just assigned. The `length` method takes an R object and returns the R length. There are numerous ways of asking R about what an object represents, and `length` is one of them.

Vectors can contain numbers, strings (character data), or logical values (`TRUE` and `FALSE`) or other “atomic” data types (**tab-simpletypes?**). *Vectors cannot contain a mix of types!* We will introduce another data structure, the R `list` for situations when we need to store a mix of base R data types.

4. Vectors

Data type	Stores
numeric	floating point numbers
integer	integers
complex	complex numbers
factor	categorical data
character	strings
logical	TRUE or FALSE
NA	missing
NULL	empty
function	function type

Table 4.1.: Atomic (simplest) data types in R.

4.8. Creating vectors

Character vectors (also sometimes called “string” vectors) are entered with each value surrounded by single or double quotes; either is acceptable, but they must match. They are always displayed by R with double quotes. Here are some examples of creating vectors:

```
# examples of vectors
c('hello', 'world')
```

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

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

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

```
c(1.12341e7, 78234.126)
```

```
[1] 11234100.00    78234.13
```

```
c(TRUE, FALSE, TRUE, TRUE)
```

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

4. Vectors

```
# note how in the next case the TRUE is converted to "TRUE"
# with quotes around it.
c(TRUE, 'hello')
```

```
[1] "TRUE" "hello"
```

We can also create vectors as “regular sequences” of numbers. For example:

```
# create a vector of integers from 1 to 10
x = 1:10
# and backwards
x = 10:1
```

The seq function can create more flexible regular sequences.

```
# create a vector of numbers from 1 to 4 skipping by 0.3
y = seq(1,4,0.3)
```

And creating a new vector by concatenating existing vectors is possible, as well.

```
# create a sequence by concatenating two other sequences
z = c(y,x)
z
```

```
[1] 1.0 1.3 1.6 1.9 2.2 2.5 2.8 3.1 3.4 3.7 4.0 10.0 9.0 8.0 7.0
[16] 6.0 5.0 4.0 3.0 2.0 1.0
```

4.9. Vector Operations

Operations on a single vector are typically done element-by-element. For example, we can add 2 to a vector, 2 is added to each element of the vector and a new vector of the same length is returned.

4. Vectors

```
x = 1:10  
x + 2
```

```
[1] 3 4 5 6 7 8 9 10 11 12
```

If the operation involves two vectors, the following rules apply. If the vectors are the same length: R simply applies the operation to each pair of elements.

```
x + x
```

```
[1] 2 4 6 8 10 12 14 16 18 20
```

If the vectors are different lengths, but one length a multiple of the other, R reuses the shorter vector as needed.

```
x = 1:10  
y = c(1,2)  
x * y
```

```
[1] 1 4 3 8 5 12 7 16 9 20
```

If the vectors are different lengths, but one length *not* a multiple of the other, R reuses the shorter vector as needed *and* delivers a warning.

```
x = 1:10  
y = c(2,3,4)  
x * y
```

Warning in `x * y`: longer object length is not a multiple of shorter object length

```
[1] 2 6 12 8 15 24 14 24 36 20
```

Typical operations include multiplication ("`*`"), addition, subtraction, division, exponentiation ("`^`"), but many operations in R operate on vectors and are then called "vectorized".

4.10. Logical Vectors

Logical vectors are vectors composed on only the values TRUE and FALSE. Note the all-upper-case and no quotation marks.

```
a = c(TRUE, FALSE, TRUE)

# we can also create a logical vector from a numeric vector
# 0 = false, everything else is 1
b = c(1, 0, 217)
d = as.logical(b)
d
```

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

```
# test if a and d are the same at every element
all.equal(a, d)
```

```
[1] TRUE
```

```
# We can also convert from logical to numeric
as.numeric(a)
```

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

4.10.1. Logical Operators

Some operators like <, >, ==, >=, <=, != can be used to create logical vectors.

```
# create a numeric vector
x = 1:10
# testing whether x > 5 creates a logical vector
x > 5
```

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

4. Vectors

```
x <= 5
```

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

```
x != 5
```

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

```
x == 5
```

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

We can also assign the results to a variable:

```
y = (x == 5)  
y
```

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

4.11. Indexing Vectors

In R, an index is used to refer to a specific element or set of elements in an vector (or other data structure). [R uses [and] to perform indexing, although other approaches to getting subsets of larger data structures are common in R.

```
x = seq(0,1,0.1)  
# create a new vector from the 4th element of x  
x[4]
```

```
[1] 0.3
```

We can even use other vectors to perform the “indexing”.

4. Vectors

```
x[c(3,5,6)]
```

```
[1] 0.2 0.4 0.5
```

```
y = 3:6  
x[y]
```

```
[1] 0.2 0.3 0.4 0.5
```

Combining the concept of indexing with the concept of logical vectors results in a very power combination.

```
# use help('rnorm') to figure out what is happening next  
myvec = rnorm(10)  
  
# create logical vector that is TRUE where myvec is >0.25  
gt1 = (myvec > 0.25)  
sum(gt1)
```

```
[1] 6
```

```
# and use our logical vector to create a vector of myvec values that are >0.25  
myvec[gt1]
```

```
[1] 0.5912198 1.6688328 0.8054312 1.1016043 1.6594198 0.5902489
```

```
# or <=0.25 using the logical "not" operator, "!"  
myvec[!gt1]
```

```
[1] -0.077542669 -0.991719274 -0.009135827 0.068190291
```

```
# shorter, one line approach  
myvec[myvec > 0.25]
```

```
[1] 0.5912198 1.6688328 0.8054312 1.1016043 1.6594198 0.5902489
```

4.12. Character Vectors, A.K.A. Strings

R uses the `paste` function to concatenate strings.

```
paste("abc", "def")
```

```
[1] "abc def"
```

```
paste("abc", "def", sep="THISSEP")
```

```
[1] "abcTHISSEPdef"
```

```
paste0("abc", "def")
```

```
[1] "abcdef"
```

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

```
paste(c("X", "Y"), 1:10)
```

```
[1] "X 1" "Y 2" "X 3" "Y 4" "X 5" "Y 6" "X 7" "Y 8" "X 9" "Y 10"
```

```
paste(c("X", "Y"), 1:10, sep="_")
```

```
[1] "X_1" "Y_2" "X_3" "Y_4" "X_5" "Y_6" "X_7" "Y_8" "X_9" "Y_10"
```

We can count the number of characters in a string.

```
nchar('abc')
```

```
[1] 3
```

```
nchar(c('abc', 'd', 123456))
```

```
[1] 3 1 6
```

Pulling out parts of strings is also sometimes useful.

4. Vectors

```
substr('This is a good sentence.',start=10,stop=15)
```

```
[1] " good "
```

Another common operation is to replace something in a string with something (a find-and-replace).

```
sub('This','That','This is a good sentence.')
```

```
[1] "That is a good sentence."
```

When we want to find all strings that match some other string, we can use `grep`, or “grab regular expression”.

```
grep('bcd',c('abcdef','abcd','bcde','cdef','defg'))
```

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

```
grep('bcd',c('abcdef','abcd','bcde','cdef','defg'),value=TRUE)
```

```
[1] "abcdef" "abcd"   "bcde"
```

Read about the `grep1` function (`?grep1`). Use that function to return a logical vector (TRUE/FALSE) for each entry above with an `a` in it.

4.13. Missing Values, AKA “NA”

R has a special value, “NA”, that represents a “missing” value, or *Not Available*, in a vector or other data structure. Here, we just create a vector to experiment.

```
x = 1:5  
x
```

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

4. Vectors

```
length(x)
```

```
[1] 5
```

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

```
[1] FALSE FALSE FALSE FALSE FALSE
```

```
x[2] = NA  
x
```

```
[1] 1 NA 3 4 5
```

The length of `x` is unchanged, but there is one value that is marked as “missing” by virtue of being `NA`.

```
length(x)
```

```
[1] 5
```

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

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

We can remove `NA` values by using indexing. In the following, `is.na(x)` returns a logical vector the length of `x`. The `!` is the logical *NOT* operator and converts `TRUE` to `FALSE` and vice-versa.

```
x[!is.na(x)]
```

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

4.14. Exercises

1. Create a numeric vector called `temperatures` containing the following values: 72, 75, 78, 81, 76, 73.

4. Vectors

```
temperatures <- c(72, 75, 78, 81, 76, 73, 93)
```

2. Create a character vector called `days` containing the following values: "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday".

```
days <- c("Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday")
```

3. Calculate the average temperature for the week and store it in a variable called `average_temperature`.

```
average_temperature <- mean(temperatures)
```

4. Create a named vector called `weekly_temperatures`, where the names are the days of the week and the values are the temperatures from the `temperatures` vector.

```
weekly_temperatures <- temperatures  
names(weekly_temperatures) <- days
```

5. Create a numeric vector called `ages` containing the following values: 25, 30, 35, 40, 45, 50, 55, 60.

```
ages <- c(25, 30, 35, 40, 45, 50, 55, 60)
```

6. Create a logical vector called `is_adult` by checking if the elements in the `ages` vector are greater than or equal to 18.

```
is_adult <- ages >= 18
```

7. Calculate the sum and product of the `ages` vector.

```
sum_ages <- sum(ages)  
product_ages <- prod(ages)
```

8. Extract the ages greater than or equal to 40 from the `ages` vector and store them in a variable called `older_ages`.

```
older_ages <- ages[ages >= 40]
```

5. Matrices

A *matrix* is a rectangular collection of the same data type (see Figure 5.1). It can be viewed as a collection of column vectors all of the same length and the same type (i.e. numeric, character or logical) OR a collection of row vectors, again all of the same type and length. A *data.frame* is *also* a rectangular array. All of the columns must be the same length, but they **may be** of *different* types. The rows and columns of a matrix or data frame can be given names. However these are implemented differently in R; many operations will work for one but not both, often a source of confusion.

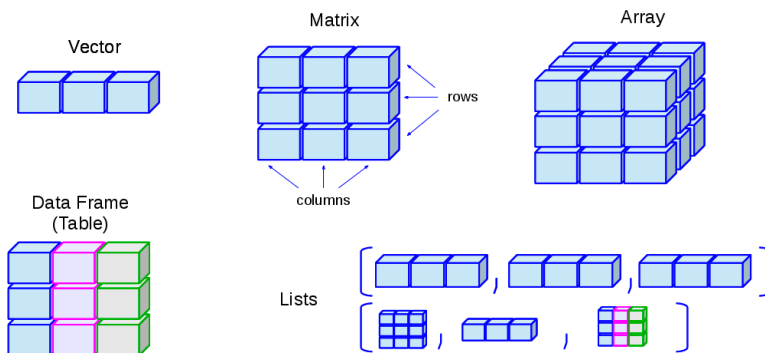


Figure 5.1.: A matrix is a collection of column vectors.

5.1. Creating a matrix

There are many ways to create a matrix in R. One of the simplest is to use the `matrix()` function. In the code below, we'll create a matrix from a vector from 1:16.

```
mat1 <- matrix(1:16, nrow=4)
mat1
```

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

5. Matrices

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

The same is possible, but specifying that the matrix be “filled” by row.

```
mat1 <- matrix(1:16,nrow=4,byrow = TRUE)
mat1
```

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

Notice the subtle difference in the order that the numbers go into the matrix.

We can also build a matrix from parts by “binding” vectors together:

```
x <- 1:10
y <- rnorm(10)
```

Each of the vectors above is of length 10 and both are “numeric”, so we can make them into a matrix. Using `rbind` binds rows (**r**) into a matrix.

```
mat <- rbind(x,y)
mat
```

	[,1]	[,2]	[,3]	[,4]	[,5]	[,6]	[,7]
x	1.000000	2.000000	3.000000	4.000000	5.000000	6.000000	7.000000
y	1.067529	0.07571403	-0.364383	-0.4984749	-1.154545	-1.259663	-0.4787461
	[,8]	[,9]	[,10]				
x	8.000000	9.000000	10.000000				
y	1.226782	-0.03506617	-1.085638				

5. Matrices

The alternative to `rbind` is `cbind` that binds columns (**c**) together.

```
mat <- cbind(x,y)
mat
```

```
      x      y
[1,]  1 1.06752879
[2,]  2  0.07571403
[3,]  3 -0.36438304
[4,]  4 -0.49847495
[5,]  5 -1.15454461
[6,]  6 -1.25966295
[7,]  7 -0.47874610
[8,]  8  1.22678244
[9,]  9 -0.03506617
[10,] 10 -1.08563803
```

Inspecting the names associated with rows and columns is often useful, particularly if the names have human meaning.

```
rownames(mat)
```

```
NULL
```

```
colnames(mat)
```

```
[1] "x" "y"
```

We can also change the names of the matrix by assigning *valid* names to the columns or rows.

```
colnames(mat) = c('apples', 'oranges')
colnames(mat)
```

```
[1] "apples" "oranges"
```

5. Matrices

```
mat
```

	apples	oranges
[1,]	1	1.06752879
[2,]	2	0.07571403
[3,]	3	-0.36438304
[4,]	4	-0.49847495
[5,]	5	-1.15454461
[6,]	6	-1.25966295
[7,]	7	-0.47874610
[8,]	8	1.22678244
[9,]	9	-0.03506617
[10,]	10	-1.08563803

Matrices have dimensions.

```
dim(mat)
```

```
[1] 10  2
```

```
nrow(mat)
```

```
[1] 10
```

```
ncol(mat)
```

```
[1] 2
```

5.2. Accessing elements of a matrix

Indexing for matrices works as for vectors except that we now need to include both the row and column (in that order). We can access elements of a matrix using the square bracket `[]` indexing method. Elements can be accessed as `var[r, c]`. Here, `r` and `c` are vectors describing the elements of the matrix to select.

5. Matrices

! Important

The indices in R start with one, meaning that the first element of a vector or the first row/column of a matrix is indexed as one.

This is different from some other programming languages, such as Python, which use zero-based indexing, meaning that the first element of a vector or the first row/column of a matrix is indexed as zero.

It is important to be aware of this difference when working with data in R, especially if you are coming from a programming background that uses zero-based indexing. Using the wrong index can lead to unexpected results or errors in your code.

```
# The 2nd element of the 1st row of mat
mat[1,2]
```

```
oranges
1.067529
```

```
# The first ROW of mat
mat[1,]
```

```
apples oranges
1.000000 1.067529
```

```
# The first COLUMN of mat
mat[,1]
```

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

```
# and all elements of mat that are > 4; note no comma
mat[mat>4]
```

```
[1] 5 6 7 8 9 10
```


5. Matrices

```
## [1] 5 6 7 8 9 10
```

Caution

Note that in the last case, there is no “,” so R treats the matrix as a long vector (length=20). This is convenient, sometimes, but it can also be a source of error, as some code may “work” but be doing something unexpected.

We can also use indexing to exclude a row or column by prefixing the selection with a - sign.

```
mat[,-1] # remove first column
```

```
[1] 1.06752879 0.07571403 -0.36438304 -0.49847495 -1.15454461 -1.25966295
[7] -0.47874610 1.22678244 -0.03506617 -1.08563803
```

```
mat[-c(1:5),] # remove first five rows
```

	apples	oranges
[1,]	6	-1.25966295
[2,]	7	-0.47874610
[3,]	8	1.22678244
[4,]	9	-0.03506617
[5,]	10	-1.08563803

5.3. Changing values in a matrix

We can create a matrix filled with random values drawn from a normal distribution for our work below.

```
m = matrix(rnorm(20),nrow=10)
summary(m)
```

	V1	V2
Min.	:-1.00782	Min. :-1.3058
1st Qu.	:-0.05989	1st Qu.:-0.3886

5. Matrices

Median :	0.38975	Median :	0.3012
Mean :	0.24393	Mean :	0.2810
3rd Qu.:	0.55160	3rd Qu.:	0.9771
Max. :	1.03408	Max. :	1.7060

Multiplication and division works similarly to vectors. When multiplying by a vector, for example, the values of the vector are reused. In the simplest case, let's multiply the matrix by a constant (vector of length 1).

```
# multiply all values in the matrix by 20
m2 = m*20
summary(m2)
```

	V1		V2
Min. :	-20.156	Min. :	-26.115
1st Qu.:	-1.198	1st Qu.:	-7.772
Median :	7.795	Median :	6.024
Mean :	4.879	Mean :	5.620
3rd Qu.:	11.032	3rd Qu.:	19.543
Max. :	20.682	Max. :	34.120

By combining subsetting with assignment, we can make changes to just part of a matrix.

```
# and add 100 to the first column of m
m2[,1] = m2[,1] + 100
# summarize m
summary(m2)
```

	V1		V2
Min. :	79.84	Min. :	-26.115
1st Qu.:	98.80	1st Qu.:	-7.772
Median :	107.80	Median :	6.024
Mean :	104.88	Mean :	5.620
3rd Qu.:	111.03	3rd Qu.:	19.543
Max. :	120.68	Max. :	34.120

5. Matrices

A somewhat common transformation for a matrix is to transpose which changes rows to columns. One might need to do this if an assay output from a lab machine puts samples in rows and genes in columns, for example, while in Bioconductor/R, we often want the samples in columns and the genes in rows.

```
t(m2)
```

```
      [,1] [,2] [,3] [,4] [,5] [,6] [,7]
[1,] 110.858518 106.50114 109.08903 95.822400 102.673212 97.51182 111.08978
[2,]  5.611247 -26.11541  32.06543  6.599318 -3.634014 34.12030 -13.59299
      [,8] [,9] [,10]
[1,] 120.68152 79.84366 114.71482
[2,]  23.85726 -9.15092  6.43664
```

5.4. Calculations on matrix rows and columns

Again, we just need a matrix to play with. We'll use `rnorm` again, but with a slight twist.

```
m3 = matrix(rnorm(100,5,2),ncol=10) # what does the 5 mean here? And the 2?
```

Since these data are from a normal distribution, we can look at a row (or column) to see what the mean and standard deviation are.

```
mean(m3[,1])
```

```
[1] 6.341979
```

```
sd(m3[,1])
```

```
[1] 2.207763
```

```
# or a row
mean(m3[1,])
```

```
[1] 5.966922
```

5. Matrices

```
sd(m3[1,])
```

```
[1] 2.680003
```

There are some useful convenience functions for computing means and sums of data in **all** of the columns and rows of matrices.

```
colMeans(m3)
```

```
[1] 6.341979 5.144662 3.912735 5.022973 5.050446 4.885206 5.162700 5.007736  
[9] 5.104758 4.230469
```

```
rowMeans(m3)
```

```
[1] 5.966922 4.423758 5.317974 4.156230 4.986355 5.383235 5.615903 4.186546  
[9] 5.278553 4.548189
```

```
rowSums(m3)
```

```
[1] 59.66922 44.23758 53.17974 41.56230 49.86355 53.83235 56.15903 41.86546  
[9] 52.78553 45.48189
```

```
colSums(m3)
```

```
[1] 63.41979 51.44662 39.12735 50.22973 50.50446 48.85206 51.62700 50.07736  
[9] 51.04758 42.30469
```

We can look at the distribution of column means:

```
# save as a variable  
cmeans = colMeans(m3)  
summary(cmeans)
```

Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
3.913	4.916	5.037	4.986	5.135	6.342

5. Matrices

Note that this is centered pretty closely around the selected mean of 5 above.

How about the standard deviation? There is not a `colSd` function, but it turns out that we can easily apply functions that take vectors as input, like `sd` and “apply” them across either the rows (the first dimension) or columns (the second) dimension.

```
csds = apply(m3, 2, sd)
summary(csds)
```

Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
1.382	1.926	1.959	2.085	2.333	2.762

Again, take a look at the distribution which is centered quite close to the selected standard deviation when we created our matrix.

5.5. Exercises

5.5.1. Data preparation

For this set of exercises, we are going to rely on a dataset that comes with R. It gives the number of sunspots per month from 1749-1983. The dataset comes as a `ts` or time series data type which I convert to a matrix using the following code.

Just run the code as is and focus on the rest of the exercises.

```
data(sunspots)
sunspot_mat <- matrix(as.vector(sunspots), ncol=12, byrow = TRUE)
colnames(sunspot_mat) <- as.character(1:12)
rownames(sunspot_mat) <- as.character(1749:1983)
```

5.5.2. Questions

- After the conversion above, what does `sunspot_mat` look like? Use functions to find the number of rows, the number of columns, the class, and some basic summary statistics.

5. Matrices

```
ncol(sunspot_mat)
nrow(sunspot_mat)
dim(sunspot_mat)
summary(sunspot_mat)
head(sunspot_mat)
tail(sunspot_mat)
```

- Practice subsetting the matrix a bit by selecting:
 - The first 10 years (rows)
 - The month of July (7th column)
 - The value for July, 1979 using the rowname to do the selection.

```
sunspot_mat[1:10,]
sunspot_mat[,7]
sunspot_mat['1979',7]
```

1. These next few exercises take advantage of the fact that calling a univariate statistical function (one that expects a vector) works for matrices by just making a vector of all the values in the matrix. What is the highest (max) number of sunspots recorded in these data?

```
max(sunspot_mat)
```

2. And the minimum?

```
min(sunspot_mat)
```

3. And the overall mean and median?

```
mean(sunspot_mat)
median(sunspot_mat)
```

4. Use the `hist()` function to look at the distribution of all the monthly sunspot data.

```
hist(sunspot_mat)
```

5. Read about the `breaks` argument to `hist()` to try to increase the number of breaks in the histogram to increase the resolution slightly. Adjust your `hist()` and `breaks` to your liking.

5. Matrices

```
hist(sunspot_mat, breaks=40)
```

6. Now, let's move on to summarizing the data a bit to learn about the pattern of sunspots varies by month or by year. Examine the dataset again. What do the columns represent? And the rows?

```
# just a quick glimpse of the data will give us a sense  
head(sunspot_mat)
```

7. We'd like to look at the distribution of sunspots by month. How can we do that?

```
# the mean of the columns is the mean number of sunspots per month.  
colMeans(sunspot_mat)  
  
# Another way to write the same thing:  
apply(sunspot_mat, 2, mean)
```

8. Assign the month summary above to a variable and summarize it to get a sense of the spread over months.

```
monthmeans = colMeans(sunspot_mat)  
summary(monthmeans)
```

9. Play the same game for years to get the per-year mean?

```
ymeans = rowMeans(sunspot_mat)  
summary(ymeans)
```

10. Make a plot of the yearly means. Do you see a pattern?

```
plot(ymeans)  
# or make it clearer  
plot(ymeans, type='l')
```

6. Data Frames

While R has many different data types, the one that is central to much of the power and popularity of R is the `data.frame`. A `data.frame` looks a bit like an R matrix in that it has two dimensions, rows and columns. However, `data.frames` are usually viewed as a set of columns representing variables and the rows representing the values of those variables. Importantly, a `data.frame` may contain *different* data types in each of its columns; matrices **must** contain only one data type. This distinction is important to remember, as there are *specific* approaches to working with R `data.frames` that may be different than those for working with matrices.

6.1. Learning goals

- Understand how `data.frames` are different from matrices.
- Know a few functions for examining the contents of a `data.frame`.
- List approaches for subsetting `data.frames`.
- Be able to load and save tabular data from and to disk.
- Show how to create a `data.frames` from scratch.

6.2. Learning objectives

- Load the yeast growth dataset into R using `read.csv`.
- Examine the contents of the dataset.
- Use subsetting to find genes that may be involved with nutrient metabolism and transport.
- Summarize data measurements by categories.

6.3. Dataset

The data used here are borrowed directly from the [fantastic Bioconductor tutorials](#) and are a cleaned up version of the data from [Brauer et al. Coordination of Growth Rate, Cell Cycle, Stress Response, and Metabolic Activity in Yeast \(2008\) Mol Biol Cell 19:352-367](#). These data are from a gene expression microarray, and in this paper the authors examine the relationship between growth rate and gene expression in yeast cultures limited by one of six different nutrients (glucose, leucine, ammonium, sulfate, phosphate, uracil). If you give yeast a rich media loaded with nutrients except restrict the supply of a single nutrient, you can control the growth rate to any rate you choose. By starving yeast of specific nutrients you can find genes that:

1. Raise or lower their expression in response to growth rate. Growth-rate dependent expression patterns can tell us a lot about cell cycle control, and how the cell responds to stress. The authors found that expression of >25% of all yeast genes is linearly correlated with growth rate, independent of the limiting nutrient. They also found that the subset of negatively growth-correlated genes is enriched for peroxisomal functions, and positively correlated genes mainly encode ribosomal functions.
2. Respond differently when different nutrients are being limited. If you see particular genes that respond very differently when a nutrient is sharply restricted, these genes might be involved in the transport or metabolism of that specific nutrient.

The dataset can be downloaded directly from:

- [brauer2007_tidy.csv](#)

We are going to read this dataset into R and then use it as a playground for learning about data.frames.

6.4. Reading in data

R has many capabilities for reading in data. Many of the functions have names that help us to understand what data format is to be expected. In this case, the filename that we want to read ends in `.csv`, meaning comma-separated-values. The `read.csv()` function reads in `.csv` files. As usual, it is worth reading `help('read.csv')` to get a better sense of the possible bells-and-whistles.

The `read.csv()` function can read directly from a URL, so we do not need to download the file directly. This dataset is relatively large (about 16MB), so this may take a bit depending on your network connection speed.

```
options(width=60)

url = paste0(
  'https://raw.githubusercontent.com',
  '/bioconnector/workshops/master/data/brauer2007_tidy.csv'
)
ydat <- read.csv(url)
```

Our variable, `ydat`, now “contains” the downloaded and read data. We can check to see what data type `read.csv` gave us:

```
class(ydat)
```

```
[1] "data.frame"
```

6.5. Inspecting data.frames

Our `ydat` variable is a `data.frame`. As I mentioned, the dataset is fairly large, so we will not be able to look at it all at once on the screen. However, R gives us many tools to inspect a `data.frame`.

- Overviews of content
 - `head()` to show first few rows
 - `tail()` to show last few rows

6. Data Frames

- Size
 - `dim()` for dimensions (rows, columns)
 - `nrow()`
 - `ncol()`
 - `object.size()` for power users interested in the memory used to store an object
- Data and attribute summaries
 - `colnames()` to get the names of the columns
 - `rownames()` to get the “names” of the rows—may not be present
 - `summary()` to get per-column summaries of the data in the data.frame.

```
head(ydat)
```

	symbol	systematic_name	nutrient	rate	expression
1	SFB2	YNL049C	Glucose	0.05	-0.24
2	<NA>	YNL095C	Glucose	0.05	0.28
3	QRI7	YDL104C	Glucose	0.05	-0.02
4	CFT2	YLR115W	Glucose	0.05	-0.33
5	SSO2	YMR183C	Glucose	0.05	0.05
6	PSP2	YML017W	Glucose	0.05	-0.69

bp

1	ER to Golgi transport
2	biological process unknown
3	proteolysis and peptidolysis
4	mRNA polyadenylation*
5	vesicle fusion*
6	biological process unknown

mf

1	molecular function unknown
2	molecular function unknown
3	metalloendopeptidase activity
4	RNA binding
5	t-SNARE activity
6	molecular function unknown

6. Data Frames

```
tail(ydat)
```

	symbol	systematic_name	nutrient	rate	expression
198425	DOA1	YKL213C	Uracil	0.3	0.14
198426	KRE1	YNL322C	Uracil	0.3	0.28
198427	MTL1	YGR023W	Uracil	0.3	0.27
198428	KRE9	YJL174W	Uracil	0.3	0.43
198429	UTH1	YKR042W	Uracil	0.3	0.19
198430	<NA>	YOL111C	Uracil	0.3	0.04

bp

198425	ubiquitin-dependent protein catabolism*
198426	cell wall organization and biogenesis
198427	cell wall organization and biogenesis
198428	cell wall organization and biogenesis*
198429	mitochondrion organization and biogenesis*
198430	biological process unknown

mf

198425	molecular function unknown
198426	structural constituent of cell wall
198427	molecular function unknown
198428	molecular function unknown
198429	molecular function unknown
198430	molecular function unknown

```
dim(ydat)
```

```
[1] 198430      7
```

```
nrow(ydat)
```

```
[1] 198430
```

```
ncol(ydat)
```

```
[1] 7
```

6. Data Frames

```
colnames(ydat)
```

```
[1] "symbol"      "systematic_name" "nutrient"  
[4] "rate"        "expression"      "bp"  
[7] "mf"
```

```
summary(ydat)
```

symbol	systematic_name	nutrient
Length:198430	Length:198430	Length:198430
Class :character	Class :character	Class :character
Mode :character	Mode :character	Mode :character

rate	expression	bp
Min. :0.0500	Min. : -6.500000	Length:198430
1st Qu.:0.1000	1st Qu.: -0.290000	Class :character
Median :0.2000	Median : 0.000000	Mode :character
Mean :0.1752	Mean : 0.003367	
3rd Qu.:0.2500	3rd Qu.: 0.290000	
Max. :0.3000	Max. : 6.640000	

```
mf  
Length:198430  
Class :character  
Mode :character
```

In RStudio, there is an additional function, `View()` (note the capital “V”) that opens the first 1000 rows (default) in the RStudio window, akin to a spreadsheet view.

```
View(ydat)
```

6.6. Accessing variables (columns) and subsetting

In R, `data.frames` can be subset similarly to other two-dimensional data structures. The `[` in R is used to denote subsetting of any kind. When working with two-dimensional data, we need two values inside the `[]` to specify the details. The specification is `[rows, columns]`. For example, to get the first three rows of `ydat`, use:

```
ydat[1:3, ]
```

	symbol	systematic_name	nutrient	rate	expression
1	SFB2	YNL049C	Glucose	0.05	-0.24
2	<NA>	YNL095C	Glucose	0.05	0.28
3	QRI7	YDL104C	Glucose	0.05	-0.02

	bp
1	ER to Golgi transport
2	biological process unknown
3	proteolysis and peptidolysis

	mf
1	molecular function unknown
2	molecular function unknown
3	metalloendopeptidase activity

Note how the second number, the columns, is blank. R takes that to mean “all the columns”. Similarly, we can combine rows and columns specification arbitrarily.

```
ydat[1:3, 1:3]
```

	symbol	systematic_name	nutrient
1	SFB2	YNL049C	Glucose
2	<NA>	YNL095C	Glucose
3	QRI7	YDL104C	Glucose

Because selecting a single variable, or column, is such a common operation, there are two shortcuts for doing so *with data.frames*. The first, the `$` operator works like so:

6. Data Frames

```
# Look at the column names, just to refresh memory
colnames(ydat)
```

```
[1] "symbol"          "systematic_name" "nutrient"
[4] "rate"            "expression"       "bp"
[7] "mf"
```

```
# Note that I am using "head" here to limit the output
head(ydat$symbol)
```

```
[1] "SFB2" NA      "QRI7" "CFT2" "SSO2" "PSP2"
```

```
# What is the actual length of "symbol"?
length(ydat$symbol)
```

```
[1] 198430
```

The second is related to the fact that, in R, data.frames are also lists. We subset a list by using `[[]]` notation. To get the second column of `ydat`, we can use:

```
head(ydat[[2]])
```

```
[1] "YNL049C" "YNL095C" "YDL104C" "YLR115W" "YMR183C"
[6] "YML017W"
```

Alternatively, we can use the column name:

```
head(ydat[["systematic_name"]])
```

```
[1] "YNL049C" "YNL095C" "YDL104C" "YLR115W" "YMR183C"
[6] "YML017W"
```

6.6.1. Some data exploration

There are a couple of columns that include numeric values. Which columns are numeric?

6. Data Frames

```
class(ydat$symbol)
```

```
[1] "character"
```

```
class(ydat$rate)
```

```
[1] "numeric"
```

```
class(ydat$expression)
```

```
[1] "numeric"
```

Make histograms of: - the expression values - the rate values

What does the `table()` function do? Could you use that to look at the rate column given that that column appears to have repeated values?

What rate corresponds to the most nutrient-starved condition?

6.6.2. More advanced indexing and subsetting

We can use, for example, logical values (TRUE/FALSE) to subset data.frames.

```
head(ydat[ydat$symbol == 'LEU1', ])
```

	symbol	systematic_name	nutrient	rate	expression	bp
NA	<NA>	<NA>	<NA>	NA	NA	<NA>
NA.1	<NA>	<NA>	<NA>	NA	NA	<NA>
NA.2	<NA>	<NA>	<NA>	NA	NA	<NA>
NA.3	<NA>	<NA>	<NA>	NA	NA	<NA>
NA.4	<NA>	<NA>	<NA>	NA	NA	<NA>
NA.5	<NA>	<NA>	<NA>	NA	NA	<NA>
	mf					
NA	<NA>					
NA.1	<NA>					

6. Data Frames

```
NA.2 <NA>
NA.3 <NA>
NA.4 <NA>
NA.5 <NA>
```

```
tail(ydat[ydat$symbol == 'LEU1', ])
```

```
      symbol systematic_name nutrient rate expression
NA.47244  <NA>             <NA>    <NA>    NA        NA
NA.47245  <NA>             <NA>    <NA>    NA        NA
NA.47246  <NA>             <NA>    <NA>    NA        NA
NA.47247  <NA>             <NA>    <NA>    NA        NA
NA.47248  <NA>             <NA>    <NA>    NA        NA
NA.47249  <NA>             <NA>    <NA>    NA        NA
      bp    mf
NA.47244 <NA> <NA>
NA.47245 <NA> <NA>
NA.47246 <NA> <NA>
NA.47247 <NA> <NA>
NA.47248 <NA> <NA>
NA.47249 <NA> <NA>
```

What is the problem with this approach? It appears that there are a bunch of NA values. Taking a quick look at the symbol column, we see what the problem.

```
summary(ydat$symbol)
```

```
Length      Class      Mode
198430 character character
```

Using the `is.na()` function, we can make filter further to get down to values of interest.

```
head(ydat[ydat$symbol == 'LEU1' & !is.na(ydat$symbol), ])
```

```
      symbol systematic_name nutrient rate expression
1526    LEU1             YGL009C  Glucose 0.05      -1.12
```

6. Data Frames

7043	LEU1	YGL009C	Glucose	0.10	-0.77
12555	LEU1	YGL009C	Glucose	0.15	-0.67
18071	LEU1	YGL009C	Glucose	0.20	-0.59
23603	LEU1	YGL009C	Glucose	0.25	-0.20
29136	LEU1	YGL009C	Glucose	0.30	0.03

bp

1526	leucine biosynthesis
7043	leucine biosynthesis
12555	leucine biosynthesis
18071	leucine biosynthesis
23603	leucine biosynthesis
29136	leucine biosynthesis

mf

1526	3-isopropylmalate dehydratase activity
7043	3-isopropylmalate dehydratase activity
12555	3-isopropylmalate dehydratase activity
18071	3-isopropylmalate dehydratase activity
23603	3-isopropylmalate dehydratase activity
29136	3-isopropylmalate dehydratase activity

Sometimes, looking at the data themselves is not that important. Using `dim()` is one possibility to look at the number of rows and columns after subsetting.

```
dim(ydat[ydat$expression > 3, ])
```

```
[1] 714 7
```

Find the high expressed genes when leucine-starved. For this task we can also use `subset` which allows us to treat column names as R variables (no `$` needed).

```
subset(ydat, nutrient == 'Leucine' & rate == 0.05 & expression > 3)
```

	symbol	systematic_name	nutrient	rate	expression
133768	QDR2	YIL121W	Leucine	0.05	4.61
133772	LEU1	YGL009C	Leucine	0.05	3.84
133858	BAP3	YDR046C	Leucine	0.05	4.29
135186	<NA>	YPL033C	Leucine	0.05	3.43

6. Data Frames

135187	<NA>	YLR267W	Leucine	0.05	3.23
135288	HXT3	YDR345C	Leucine	0.05	5.16
135963	TPO2	YGR138C	Leucine	0.05	3.75
135965	YRO2	YBR054W	Leucine	0.05	4.40
136102	GPG1	YGL121C	Leucine	0.05	3.08
136109	HSP42	YDR171W	Leucine	0.05	3.07
136119	HXT5	YHR096C	Leucine	0.05	4.90
136151	<NA>	YJL144W	Leucine	0.05	3.06
136152	MOH1	YBL049W	Leucine	0.05	3.43
136153	<NA>	YBL048W	Leucine	0.05	3.95
136189	HSP26	YBR072W	Leucine	0.05	4.86
136231	NCA3	YJL116C	Leucine	0.05	4.03
136233	<NA>	YBR116C	Leucine	0.05	3.28
136486	<NA>	YGR043C	Leucine	0.05	3.07
137443	ADH2	YMR303C	Leucine	0.05	4.15
137448	ICL1	YER065C	Leucine	0.05	3.54
137451	SFC1	YJR095W	Leucine	0.05	3.72
137569	MLS1	YNL117W	Leucine	0.05	3.76

bp

133768	multidrug transport
133772	leucine biosynthesis
133858	amino acid transport
135186	meiosis*
135187	biological process unknown
135288	hexose transport
135963	polyamine transport
135965	biological process unknown
136102	signal transduction
136109	response to stress*
136119	hexose transport
136151	response to dessication
136152	biological process unknown
136153	<NA>
136189	response to stress*
136231	mitochondrion organization and biogenesis
136233	<NA>
136486	biological process unknown
137443	fermentation*
137448	glyoxylate cycle
137451	fumarate transport*
137569	glyoxylate cycle

6. Data Frames

```
mf
133768      multidrug efflux pump activity
133772 3-isopropylmalate dehydratase activity
133858      amino acid transporter activity
135186      molecular function unknown
135187      molecular function unknown
135288      glucose transporter activity*
135963      spermine transporter activity
135965      molecular function unknown
136102      signal transducer activity
136109      unfolded protein binding
136119      glucose transporter activity*
136151      molecular function unknown
136152      molecular function unknown
136153      <NA>
136189      unfolded protein binding
136231      molecular function unknown
136233      <NA>
136486      transaldolase activity
137443      alcohol dehydrogenase activity
137448      isocitrate lyase activity
137451 succinate:fumarate antiporter activity
137569      malate synthase activity
```

6.7. Aggregating data

Aggregating data, or summarizing by category, is a common way to look for trends or differences in measurements between categories. Use `aggregate` to find the mean expression by gene symbol.

```
head(aggregate(ydat$expression, by=list( ydat$symbol), mean))
```

```
Group.1      x
1  AAC1  0.52888889
2  AAC3 -0.21628571
3  AAD10 0.43833333
4  AAD14 -0.07166667
5  AAD16 0.24194444
6  AAD4 -0.79166667
```

6. Data Frames

```
# or
head(aggregate(expression ~ symbol, mean, data=ydat))
```

```
      symbol expression
1    AAC1  0.52888889
2    AAC3 -0.21628571
3   AAD10  0.43833333
4   AAD14 -0.07166667
5   AAD16  0.24194444
6    AAD4 -0.79166667
```

6.8. Creating a data.frame from scratch

Sometimes it is useful to combine related data into one object. For example, let's simulate some data.

```
smoker = factor(rep(c("smoker", "non-smoker"), each=50))
smoker_numeric = as.numeric(smoker)
x = rnorm(100)
risk = x + 2*smoker_numeric
```

We have two variables, risk and smoker that are related. We can make a data.frame out of them:

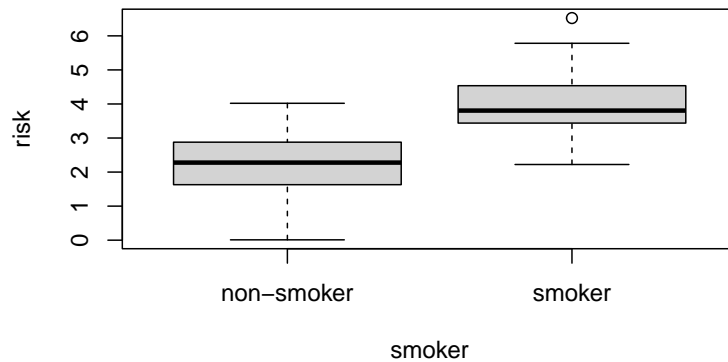
```
smoker_risk = data.frame(smoker = smoker, risk = risk)
head(smoker_risk)
```

```
      smoker      risk
1 smoker 3.056532
2 smoker 3.558159
3 smoker 4.158129
4 smoker 2.981418
5 smoker 4.052849
6 smoker 3.976635
```

R also has plotting shortcuts that work with data.frames to simplify plotting

6. Data Frames

```
plot( risk ~ smoker, data=smoker_risk)
```



6.9. Saving a data.frame

Once we have a data.frame of interest, we may want to save it. The most portable way to save a data.frame is to use one of the write functions. In this case, let's save the data as a .csv file.

```
write.csv(smoker_risk, "smoker_risk.csv")
```

7. Factors

7.1. Factors

A factor is a special type of vector, normally used to hold a categorical variable—such as smoker/nonsmoker, state of residency, zipcode—in many statistical functions. Such vectors have class “factor”. Factors are primarily used in Analysis of Variance (ANOVA) or other situations when “categories” are needed. When a factor is used as a predictor variable, the corresponding indicator variables are created (more later).

Note of caution that factors in R often *appear* to be character vectors when printed, but you will notice that they do not have double quotes around them. They are stored in R as numbers with a key name, so sometimes you will note that the factor *behaves* like a numeric vector.

```
# create the character vector
citizen<-c("uk","us","no","au","uk","us","us","no","au")

# convert to factor
citizenf<-factor(citizen)
citizenf
```

```
[1] "uk" "us" "no" "au" "uk" "us" "us" "no" "au"
```

```
citizenf
```

```
[1] uk us no au uk us us no au
Levels: au no uk us
```

7. Factors

```
# convert factor back to character vector
as.character(citizenf)
```

```
[1] "uk" "us" "no" "au" "uk" "us" "us" "no" "au"
```

```
# convert to numeric vector
as.numeric(citizenf)
```

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

R stores many data structures as vectors with “attributes” and “class” (just so you have seen this).

```
attributes(citizenf)
```

```
$levels
```

```
[1] "au" "no" "uk" "us"
```

```
$class
```

```
[1] "factor"
```

```
class(citizenf)
```

```
[1] "factor"
```

```
# note that after unclassing, we can see the
# underlying numeric structure again
unclass(citizenf)
```

```
[1] 3 4 2 1 3 4 4 2 1
attr(,"levels")
[1] "au" "no" "uk" "us"
```

Tabulating factors is a useful way to get a sense of the “sample” set available.

7. Factors

```
table(citizenf)
```

```
citizenf  
au no uk us  
  2  2  2  3
```

Part III.

Exploratory data analysis

Imagine you're on an adventure, about to embark on a journey into the unknown. You've just been handed a treasure map, with the promise of valuable insights waiting to be discovered. This map is your data set, and the journey is exploratory data analysis (EDA).

As you begin your exploration, you start by getting a feel for the terrain. You take a broad, bird's-eye view of the data, examining its structure and dimensions. Are you dealing with a vast landscape or a small, confined area? Are there any missing pieces in the map that you'll need to account for? Understanding the overall context of your data set is crucial before venturing further.

With a sense of the landscape, you now zoom in to identify key landmarks in the data. You might look for unusual patterns, trends, or relationships between variables. As you spot these landmarks, you start asking questions: What's causing that spike in values? Are these two factors related, or is it just a coincidence? By asking these questions, you're actively engaging with the data and forming hypotheses that could guide future analysis or experiments.

As you continue your journey, you realize that the map alone isn't enough to fully understand the terrain. You need more tools to bring the data to life. You start visualizing the data using charts, plots, and graphs. These visualizations act as your binoculars, allowing you to see patterns and relationships more clearly. Through them, you can uncover the hidden treasures buried within the data.

EDA isn't a linear path from start to finish. As you explore, you'll find yourself circling back to previous points, refining your questions, and digging deeper. The process is iterative, with each new discovery informing the next. And as you go, you'll gain a deeper understanding of the data's underlying structure and potential.

Finally, after your thorough exploration, you'll have a solid foundation to build upon. You'll be better equipped to make informed decisions, test hypotheses, and draw meaningful conclusions. The insights you've gained through EDA will serve as a compass, guiding you towards the true value hidden within your data. And with that, you've successfully completed your journey through exploratory data analysis.

8. Case Study: Behavioral Risk Factor Surveillance System

8.1. A Case Study on the Behavioral Risk Factor Surveillance System

The Behavioral Risk Factor Surveillance System (BRFSS) is a large-scale health survey conducted annually by the Centers for Disease Control and Prevention (CDC) in the United States. The BRFSS collects information on various health-related behaviors, chronic health conditions, and the use of preventive services among the adult population (18 years and older) through telephone interviews. The main goal of the BRFSS is to identify and monitor the prevalence of risk factors associated with chronic diseases, inform public health policies, and evaluate the effectiveness of health promotion and disease prevention programs. The data collected through BRFSS is crucial for understanding the health status and needs of the population, and it serves as a valuable resource for researchers, policy makers, and healthcare professionals in making informed decisions and designing targeted interventions.

In this chapter, we will walk through an exploratory data analysis (EDA) of the Behavioral Risk Factor Surveillance System dataset using R. EDA is an important step in the data analysis process, as it helps you to understand your data, identify trends, and detect any anomalies before performing more advanced analyses. We will use various R functions and packages to explore the dataset, with a focus on active learning and hands-on experience.

8.2. Loading the Dataset

First, let's load the dataset into R. We will use the `read.csv()` function from the base R package to read the data and store it in a data frame called `brfss`. Make sure the CSV file is in your working directory, or provide the full path to the file.

First, we need to get the data. Either download the data from [THIS LINK](https://raw.githubusercontent.com/seandavi/ITR/master/BRFSS-subset.csv) or have R do it directly from the command-line (preferred):

```
download.file('https://raw.githubusercontent.com/seandavi/ITR/master/BRFSS-subset.csv',  
             destfile = 'BRFSS-subset.csv')
```

```
path <- file.choose()    # look for BRFSS-subset.csv
```

```
stopifnot(file.exists(path))  
brfss <- read.csv(path)
```

8.3. Inspecting the Data

Once the data is loaded, let's take a look at the first few rows of the dataset using the `head()` function:

```
head(brfss)
```

	Age	Weight	Sex	Height	Year
1	31	48.98798	Female	157.48	1990
2	57	81.64663	Female	157.48	1990
3	43	80.28585	Male	177.80	1990
4	72	70.30682	Male	170.18	1990
5	31	49.89516	Female	154.94	1990
6	58	54.43108	Female	154.94	1990

This will display the first six rows of the dataset, allowing you to get a feel for the data structure and variable types.

Next, let's check the dimensions of the dataset using the `dim()` function:

8. Case Study: Behavioral Risk Factor Surveillance System

```
dim(brfss)
```

```
[1] 20000      5
```

This will return the number of rows and columns in the dataset, which is important to know for subsequent analyses.

8.4. Summary Statistics

Now that we have a basic understanding of the data structure, let's calculate some summary statistics. The `summary()` function in R provides a quick overview of the main statistics for each variable in the dataset:

```
summary(brfss)
```

Age	Weight	Sex	Height
Min. :18.00	Min. : 34.93	Length:20000	Min. :105.0
1st Qu.:36.00	1st Qu.: 61.69	Class :character	1st Qu.:162.6
Median :51.00	Median : 72.57	Mode :character	Median :168.0
Mean :50.99	Mean : 75.42		Mean :169.2
3rd Qu.:65.00	3rd Qu.: 86.18		3rd Qu.:177.8
Max. :99.00	Max. :278.96		Max. :218.0
NA's :139	NA's :649		NA's :184

Year
Min. :1990
1st Qu.:1990
Median :2000
Mean :2000
3rd Qu.:2010
Max. :2010

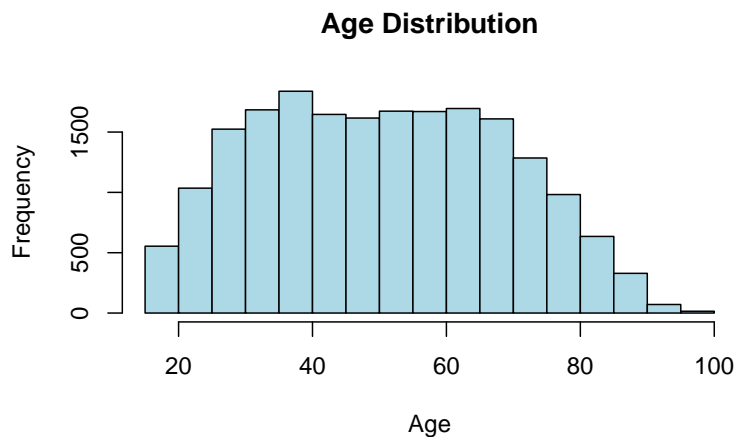
This will display the minimum, first quartile, median, mean, third quartile, and maximum for each numeric variable, and the frequency counts for each factor level for categorical variables.

8.5. Data Visualization

Visualizing the data can help you identify patterns and trends in the dataset. Let's start by creating a histogram of the Age variable using the `hist()` function.

This will create a histogram showing the frequency distribution of ages in the dataset. You can customize the appearance of the histogram by adjusting the parameters within the `hist()` function.

```
hist(brfss$Age, main = "Age Distribution",  
     xlab = "Age", col = "lightblue")
```



💡 What are the options for a histogram?

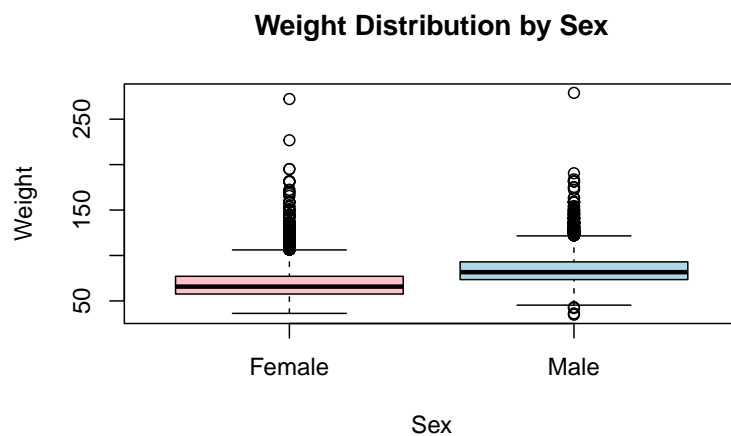
The `hist()` function has many options. For example, you can change the number of bins, the color of the bars, the title, and the x-axis label. You can also add a vertical line at the mean or median, or add a normal curve to the histogram. For more information, type `?hist` in the R console.

More generally, it is important to understand the options available for each function you use. You can do this by reading the documentation for the function, which can be accessed by typing `?function_name` or `help("function_name")` in the R console.

8. Case Study: Behavioral Risk Factor Surveillance System

Next, let's create a boxplot to compare the distribution of Weight between males and females. We will use the `boxplot()` function for this. This will create a boxplot comparing the weight distribution between males and females. You can customize the appearance of the boxplot by adjusting the parameters within the `boxplot()` function.

```
boxplot(brfss$Weight ~ brfss$Sex, main = "Weight Distribution by Sex",  
        xlab = "Sex", ylab = "Weight", col = c("pink", "lightblue"))
```



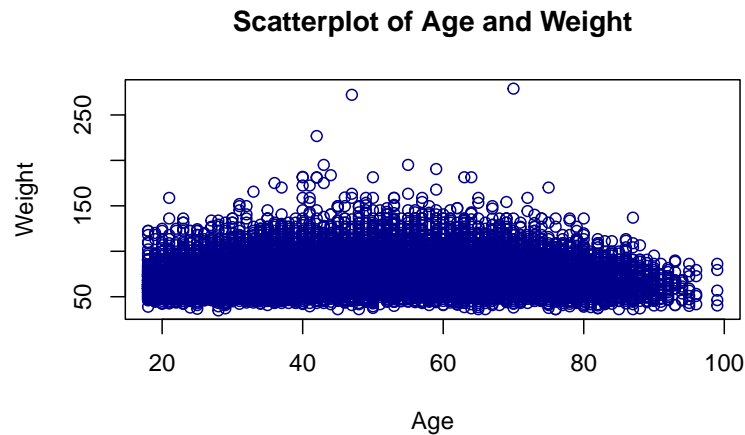
8.6. Analyzing Relationships Between Variables

To further explore the data, let's investigate the relationship between age and weight using a scatterplot. We will use the `plot()` function for this:

This will create a scatterplot of age and weight, allowing you to visually assess the relationship between these two variables.

```
plot(brfss$Age, brfss$Weight, main = "Scatterplot of Age and Weight",  
      xlab = "Age", ylab = "Weight", col = "darkblue")
```


8. Case Study: Behavioral Risk Factor Surveillance System



To quantify the strength of the relationship between age and weight, we can calculate the correlation coefficient using the `cor()` function:

This will return the correlation coefficient between age and weight, which can help you determine whether there is a linear relationship between these variables.

```
cor(brfss$Age, brfss$Weight)
```

```
[1] NA
```

Why does `cor()` give a value of NA? What can we do about it? A quick glance at `help("cor")` will give you the answer.

```
cor(brfss$Age, brfss$Weight, use = "complete.obs")
```

```
[1] 0.02699989
```

8.7. Exercises

1. What is the mean weight in this dataset? How about the median? What is the difference between the two? What does this tell you about the distribution of weights in the dataset?

8. Case Study: Behavioral Risk Factor Surveillance System

```
mean(brfss$Weight, na.rm = TRUE)
```

```
[1] 75.42455
```

```
median(brfss$Weight, na.rm = TRUE)
```

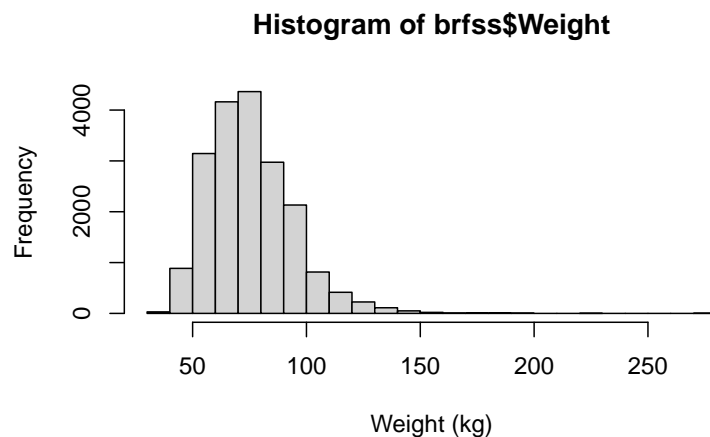
```
[1] 72.57478
```

```
mean(brfss$Weight, na.rm=TRUE) - median(brfss$Weight, na.rm = TRUE)
```

```
[1] 2.849774
```

- Given the findings about the mean and median in the previous exercise, use the `hist()` function to create a histogram of the weight distribution in this dataset. How would you describe the shape of this distribution?

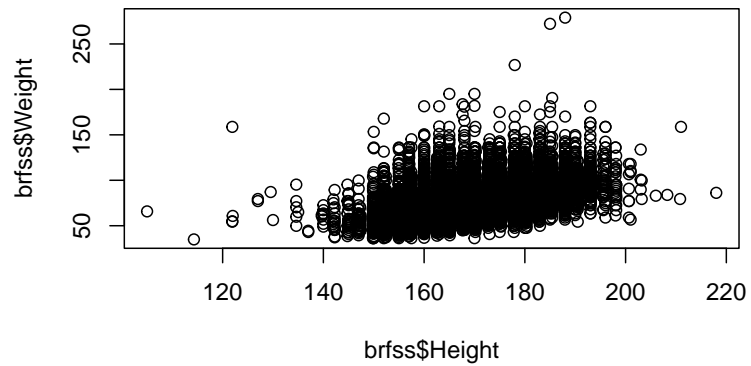
```
hist(brfss$Weight, xlab="Weight (kg)", breaks = 30)
```



- Use `plot()` to examine the relationship between height and weight in this dataset.

```
plot(brfss$Height, brfss$Weight)
```

8. Case Study: Behavioral Risk Factor Surveillance System



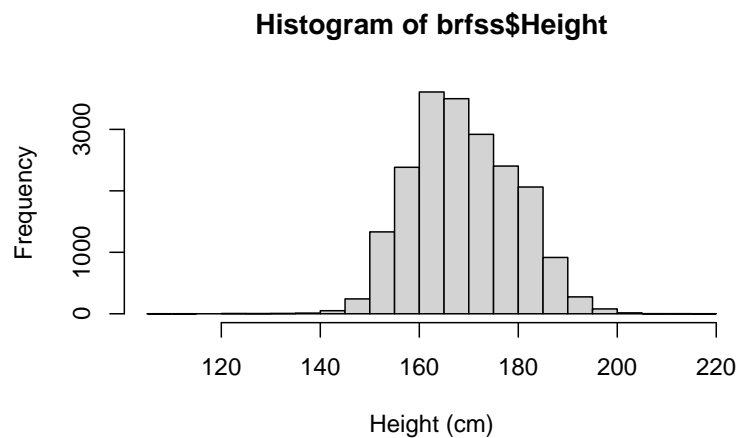
4. What is the correlation between height and weight? What does this tell you about the relationship between these two variables?

```
cor(brfss$Height, brfss$Weight, use = "complete.obs")
```

```
[1] 0.5140928
```

5. Create a histogram of the height distribution in this dataset. How would you describe the shape of this distribution?

```
hist(brfss$Height, xlab="Height (cm)", breaks = 30)
```



8.8. Conclusion

In this chapter, we have demonstrated how to perform an exploratory data analysis on the Behavioral Risk Factor Surveillance System dataset using R. We covered data loading, inspection, summary statistics, visualization, and the analysis of relationships between variables. By actively engaging with the R code and data, you have gained valuable experience in using R for EDA and are well-equipped to tackle more complex analyses in your future work.

Remember that EDA is just the beginning of the data analysis process, and further statistical modeling and hypothesis testing will likely be necessary to draw meaningful conclusions from your data. However, EDA is a crucial step in understanding your data and informing your subsequent analyses.

8.9. Learn about the data

Using the data exploration techniques you have seen to explore the brfss dataset.

- `summary()`
- `dim()`
- `colnames()`
- `head()`
- `tail()`
- `class()`
- `View()`

You may want to investigate individual columns visually using plotting like `hist()`. For categorical data, consider using something like `table()`.

8.10. Clean data

R read Year as an integer value, but it's really a factor

8. Case Study: Behavioral Risk Factor Surveillance System

```
brfss$Year <- factor(brfss$Year)
```

8.11. Weight in 1990 vs. 2010 Females

- Create a subset of the data

```
brfssFemale <- brfss[brfss$Sex == "Female",]  
summary(brfssFemale)
```

Age	Weight	Sex	Height
Min. :18.00	Min. : 36.29	Length:12039	Min. :105.0
1st Qu.:37.00	1st Qu.: 57.61	Class :character	1st Qu.:157.5
Median :52.00	Median : 65.77	Mode :character	Median :163.0
Mean :51.92	Mean : 69.05		Mean :163.3
3rd Qu.:67.00	3rd Qu.: 77.11		3rd Qu.:168.0
Max. :99.00	Max. :272.16		Max. :200.7
NA's :103	NA's :560		NA's :140
Year			
1990:5718			
2010:6321			

- Visualize

```
plot(Weight ~ Year, brfssFemale)
```

8. Case Study: Behavioral Risk Factor Surveillance System



- Statistical test

```
t.test(Weight ~ Year, brfssFemale)
```

Welch Two Sample t-test

data: Weight by Year

t = -27.133, df = 11079, p-value < 2.2e-16

alternative hypothesis: true difference in means between group 1990 and group 2010 is not equal to 0

95 percent confidence interval:

-8.723607 -7.548102

sample estimates:

mean in group 1990 mean in group 2010

64.81838 72.95424

8.12. Weight and height in 2010 Males

- Create a subset of the data

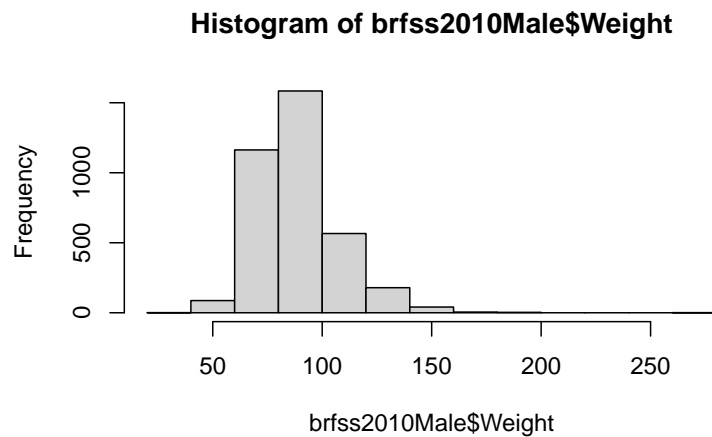
```
brfss2010Male <- subset(brfss, Year == 2010 & Sex == "Male")  
summary(brfss2010Male)
```

8. Case Study: Behavioral Risk Factor Surveillance System

Age	Weight	Sex	Height	Year
Min. :18.00	Min. : 36.29	Length:3679	Min. :135	1990: 0
1st Qu.:45.00	1st Qu.: 77.11	Class :character	1st Qu.:173	2010:3679
Median :57.00	Median : 86.18	Mode :character	Median :178	
Mean :56.25	Mean : 88.85		Mean :178	
3rd Qu.:68.00	3rd Qu.: 99.79		3rd Qu.:183	
Max. :99.00	Max. :278.96		Max. :218	
NA's :30	NA's :49		NA's :31	

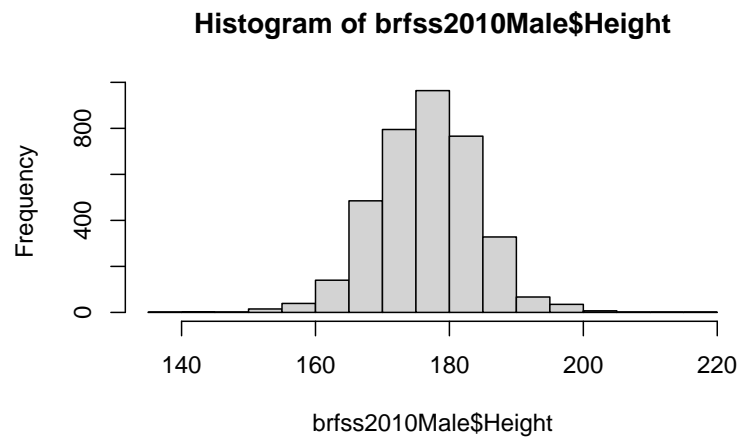
- Visualize the relationship

```
hist(brfss2010Male$Weight)
```

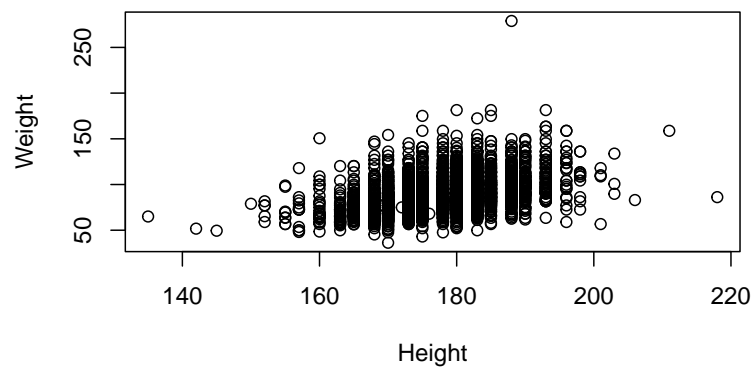


```
hist(brfss2010Male$Height)
```

8. Case Study: Behavioral Risk Factor Surveillance System



```
plot(Weight ~ Height, brfss2010Male)
```



- Fit a linear model (regression)

```
fit <- lm(Weight ~ Height, brfss2010Male)
fit
```

Call:

8. Case Study: Behavioral Risk Factor Surveillance System

```
lm(formula = Weight ~ Height, data = brfss2010Male)
```

Coefficients:

(Intercept)	Height
-86.8747	0.9873

Summarize as ANOVA table

```
anova(fit)
```

Analysis of Variance Table

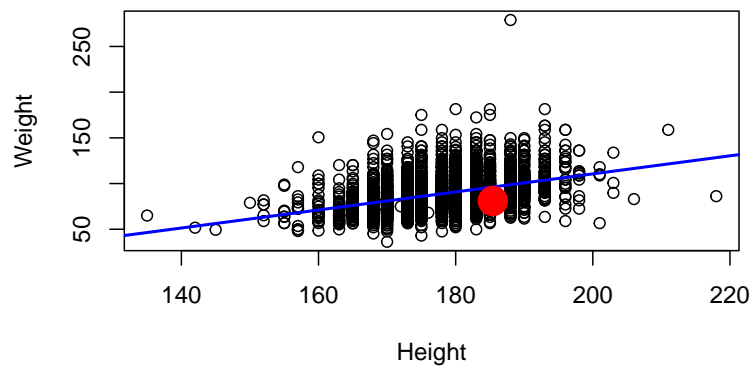
Response: Weight

	Df	Sum Sq	Mean Sq	F value	Pr(>F)
Height	1	197664	197664	693.8	< 2.2e-16 ***
Residuals	3617	1030484	285		

Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

- Plot points, superpose fitted regression line; where am I?

```
plot(Weight ~ Height, brfss2010Male)
abline(fit, col="blue", lwd=2)
# Substitute your own weight and height...
points(73 * 2.54, 178 / 2.2, col="red", cex=4, pch=20)
```



8. Case Study: Behavioral Risk Factor Surveillance System

- Class and available ‘methods’

```
class(fit)           # 'noun'
methods(class=class(fit)) # 'verb'
```

- Diagnostics

```
plot(fit)
# Note that the "plot" above does not have a ".lm"
# However, R will use "plot.lm". Why?
?plot.lm
```

Part IV.
statistics

9. The t-statistic and t-distribution

9.1. Background

The t-test is a [statistical hypothesis test](#) that is commonly used when the data are normally distributed (follow a normal distribution) if the value of the population standard deviation were known. When the population standard deviation is not known and is replaced by an estimate based on the data, the test statistic follows a Student's t distribution.

T-tests are handy hypothesis tests in statistics when you want to compare means. You can compare a sample mean to a hypothesized or target value using a one-sample t-test. You can compare the means of two groups with a two-sample t-test. If you have two groups with paired observations (e.g., before and after measurements), use the paired t-test.

A t-test looks at the t-statistic, the t-distribution values, and the degrees of freedom to determine the statistical significance. To conduct a test with three or more means, we would use an analysis of variance.

The distribution that the t-statistic follows was described in a famous paper (Student 1908) by "Student", a pseudonym for [William Sealy Gosset](#).

9.2. The Z-score and probability

Before talking about the t-distribution and t-scores, let's review the Z-score, its relation to the normal distribution, and probability.

The Z-score is defined as:

9. The *t*-statistic and *t*-distribution

$$Z = \frac{x - \mu}{\sigma} \quad (9.1)$$

where μ is a the population mean from which x is drawn and σ is the population standard deviation (taken as known, not estimated from the data).

The probability of observing a Z score of z or greater can be calculated by `pnorm(z, μ , σ)`.

For example, let's assume that our "population" is known and it truly has a mean 0 and standard deviation 1. If we have observations drawn from that population, we can assign a probability of seeing that observation by random chance *under the assumption that the null hypothesis is **TRUE***.

```
zscore = seq(-5,5,1)
```

For each value of `zscore`, let's calculate the p-value and put the results in a `data.frame`.

```
df = data.frame(  
  zscore = zscore,  
  pval    = pnorm(zscore, 0, 1)  
)  
df
```

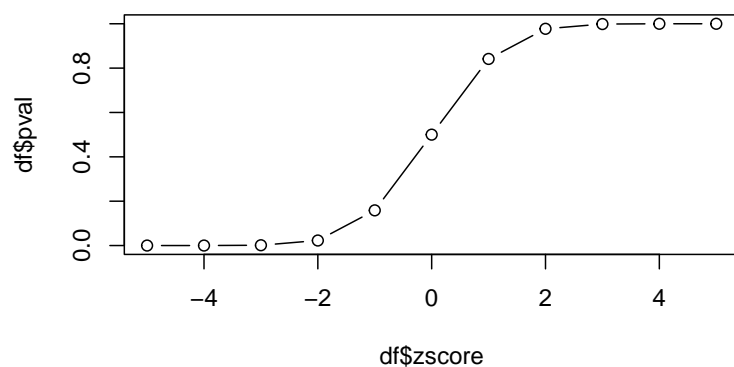
	zscore	pval
1	-5	2.866516e-07
2	-4	3.167124e-05
3	-3	1.349898e-03
4	-2	2.275013e-02
5	-1	1.586553e-01
6	0	5.000000e-01
7	1	8.413447e-01
8	2	9.772499e-01
9	3	9.986501e-01
10	4	9.999683e-01
11	5	9.999997e-01

9. The *t*-statistic and *t*-distribution

Why is the p-value of something 5 population standard deviations away from the mean (zscore=5) nearly 1 in this calculation? What is the default for `pnorm` with respect to being one-sided or two-sided?

Let's plot the values of probability vs z-score:

```
plot(df$zscore, df$pval, type='b')
```



This plot is the *empirical* cumulative density function (cdf) for our data. How can we use it? If we know the z-score, we can look up the probability of observing that value. Since we have constructed our experiment to follow the standard normal distribution, this cdf also represents the cdf of the standard normal distribution.

9.2.1. Small diversion: two-sided `pnorm` function

The `pnorm` function returns the “one-sided” probability of having a value at least as extreme as the observed x and uses the “lower” tail by default. Let's create a function that computes two-sided p-values.

1. Take the absolute value of x
2. Compute `pnorm` with `lower.tail=FALSE` so we get lower p-values with larger values of x .

9. The t-statistic and t-distribution

3. Since we want to include both tails, we need to multiply the area (probability) returned by `pnorm` by 2.

```
twosidedpnorm = function(x,mu=0,sd=1) {  
  2*pnorm(x,mu,sd,lower.tail=FALSE)  
}
```

And we can test this to see how likely it is to be 2 or 3 standard deviations from the mean:

```
twosidedpnorm(2)
```

```
[1] 0.04550026
```

```
twosidedpnorm(3)
```

```
[1] 0.002699796
```

9.3. The t-distribution

We spent time above working with z-scores and probability. An important aspect of working with the normal distribution is that we **MUST** assume that we know the standard deviation. Remember that the Z-score is defined as:

$$Z = \frac{x - \mu}{\sigma}$$

The formula for the *population* standard deviation is:

$$\sigma = \sqrt{\frac{1}{N} \sum_{i=1}^N (x_i - \mu)^2} \quad (9.2)$$

In general, the population standard deviation is taken as “known” as we did above.

If we do not but only have a *sample* from the population, instead of using the Z-score, we use the t-score defined as:

9. The *t*-statistic and *t*-distribution

$$t = \frac{x - \bar{x}}{s} \quad (9.3)$$

This looks quite similar to the formula for Z-score, but here we have to *estimate* the standard deviation, s from the data. The formula for s is:

$$s = \sqrt{\frac{1}{N-1} \sum_{i=1}^N (x_i - \bar{x})^2} \quad (9.4)$$

Since we are estimating the standard deviation from the data, this leads to extra variability that shows up as “fatter tails” for smaller sample sizes than for larger sample sizes. We can see this by comparing the *t*-distribution for various numbers of degrees of freedom (sample sizes).

We can look at the effect of sample size on the distributions graphically by looking at the densities for 3, 5, 10, 20 degrees of freedom and the normal distribution:

```
library(dplyr)
library(ggplot2)
t_values = seq(-6,6,0.01)
df = data.frame(
  value = t_values,
  t_3    = dt(t_values,3),
  t_6    = dt(t_values,6),
  t_10   = dt(t_values,10),
  t_20   = dt(t_values,20),
  Normal= dnorm(t_values)
) |>
  tidyr::gather("Distribution", "density", -value)
ggplot(df, aes(x=value, y=density, color=Distribution)) +
  geom_line()
```


9. The *t*-statistic and *t*-distribution

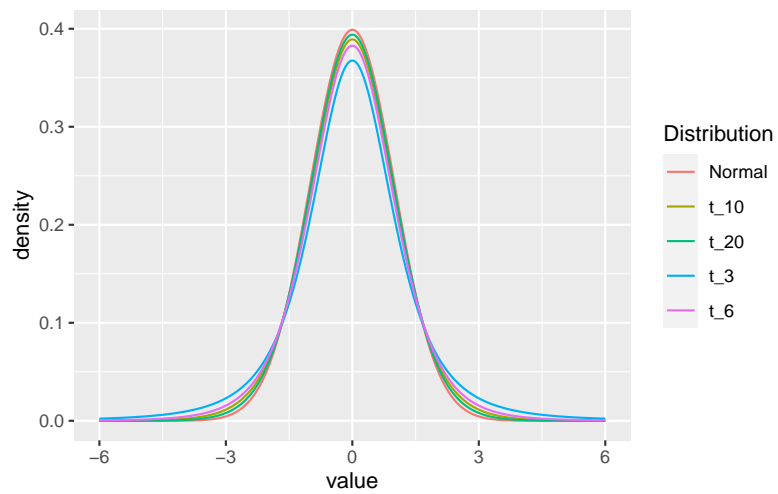
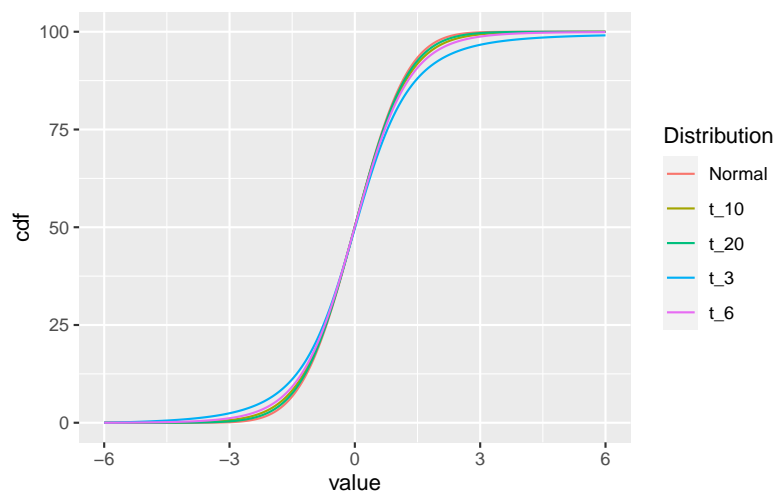


Figure 9.1: *t*-distributions for various degrees of freedom. Note that the tails are fatter for smaller degrees of freedom, which is a result of estimating the standard deviation from the data.

The `dt` and `dnorm` functions give the density of the distributions for each point.

```
df2 = df |>
  group_by(Distribution) |>
  arrange(value) |>
  mutate(cdf=cumsum(density))
ggplot(df2, aes(x=value, y=cdf, color=Distribution)) +
  geom_line()
```



9. The *t*-statistic and *t*-distribution

9.3.1. p-values based on Z vs t

When we have a “sample” of data and want to compute the statistical significance of the difference of the mean from the population mean, we calculate the standard deviation of the sample means (standard error).

$$z = \frac{x - \mu}{\sigma / \sqrt{n}}$$

Let's look at the relationship between the p-values of Z (from the normal distribution) vs t for a **sample** of data.

```
set.seed(5432)
samp = rnorm(5)
z = sqrt(length(samp)) * mean(samp) #simplifying assumption (sigma=1, mu=0)
```

And the p-value if we assume we know the standard deviation:

```
pnorm(z)
```

```
[1] 0.8035432
```

```
ts = sqrt(length(samp)) * mean(samp) / sd(samp)
pnorm(ts)
```

```
[1] 0.8215048
```

```
pt(ts,5)
```

```
[1] 0.800373
```

9.3.2. Experiment

When sampling from a normal distribution, we often calculate p-values to test hypotheses or determine the statistical significance of our results. The p-value represents the probability of obtaining a test statistic as extreme or more extreme than the one observed, under the null hypothesis.

In a typical scenario, we assume that the population mean and standard deviation are known. However, in many real-life situations, we don't know the true population standard deviation, and we have to estimate it using the sample standard deviation (Equation 9.4). This estimation introduces some uncertainty into our calculations, which affects the p-values. When we include an estimate of the standard deviation, we switch from using the standard normal (z) distribution to the t-distribution for calculating p-values.

What would happen if we used the normal distribution to calculate p-values when we use the sample standard deviation? Let's find out!

1. Simulate a bunch of samples of size n from the standard normal distribution
2. Calculate the p-value distribution for those samples based on the normal.
3. Calculate the p-value distribution for those samples based on the normal, but with the *estimated* standard deviation.
4. Calculate the p-value distribution for those samples based on the t-distribution.

Create a function that draws a sample of size n from the standard normal distribution.

```
zf = function(n) {  
  samp = rnorm(n)  
  z = sqrt(length(samp)) * mean(samp) / 1 #simplifying assumption (sigma=1, mu=0)  
  z  
}
```

And give it a try:

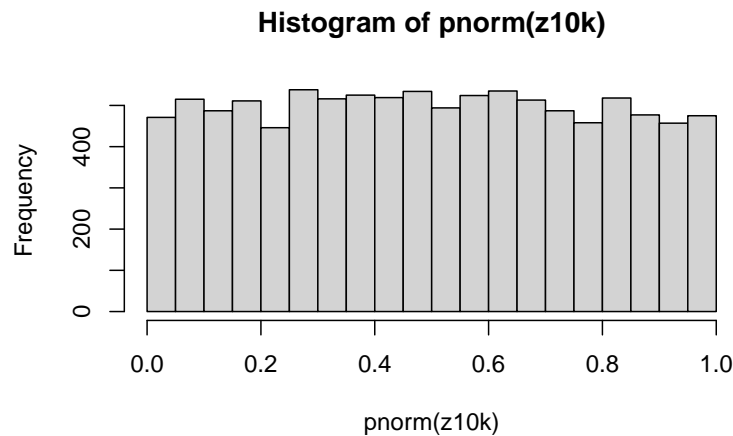
9. The *t*-statistic and *t*-distribution

```
zf(5)
```

```
[1] 0.7406094
```

Perform 10000 replicates of our sampling and z-scoring. We are using the assumption that we know the population standard deviation; in this case, we do know since we are sampling from the standard normal distribution.

```
z10k = replicate(10000, zf(5))  
hist(pnorm(z10k))
```



And do the same, but now creating a *t*-score function. We are using the assumption that we *don't* know the population standard deviation; in this case, we must estimate it from the data. Note the difference in the calculation of the *t*-score (*ts*) as compared to the *z*-score (*z*).

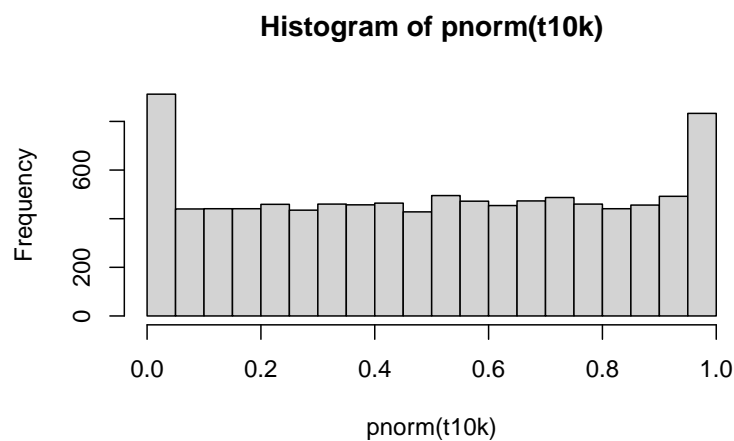
```
tf = function(n) {  
  samp = rnorm(n)  
  # now, using the sample standard deviation since we  
  # "don't know" the population standard deviation  
  ts = sqrt(length(samp)) * mean(samp) / sd(samp)  
}
```

9. The *t*-statistic and *t*-distribution

```
ts  
}
```

If we use those *t*-scores and calculate the *p*-values based on the normal distribution, the histogram of those *p*-values looks like:

```
t10k = replicate(10000,tf(5))  
hist(pnorm(t10k))
```

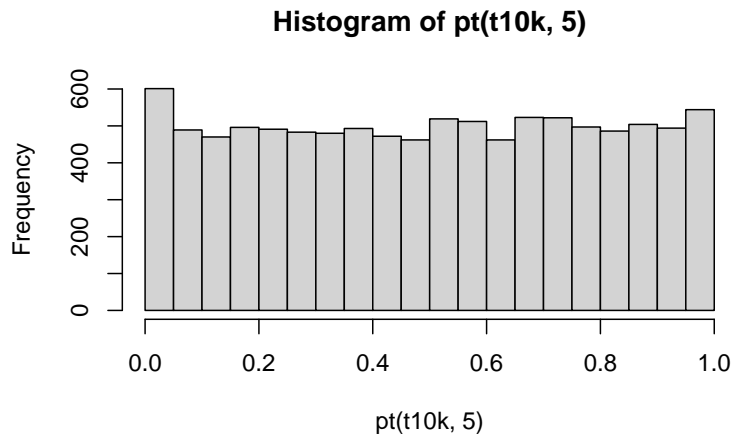


Since we are using the normal distribution to calculate the *p*-values, we are, in effect, assuming that we know the population standard deviation. This assumption is incorrect, and we can see that the *p*-values are not uniformly distributed between 0 and 1.

If we use those *t*-scores and calculate the *p*-values based on the *t*-distribution, the histogram of those *p*-values looks like:

```
hist(pt(t10k,5))
```

9. The *t*-statistic and *t*-distribution

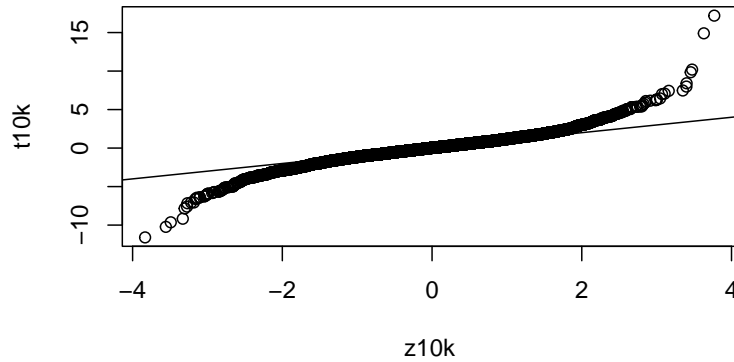


Now, the p-values are uniformly distributed between 0 and 1, as expected.

What is a qqplot and how do we use it? A qqplot is a plot of the quantiles of two distributions against each other. If the two distributions are identical, the points will fall on a straight line. If the two distributions are different, the points will deviate from the straight line. We can use a qqplot to compare the t-distribution to the normal distribution. If the t-distribution is identical to the normal distribution, the points will fall on a straight line. If the t-distribution is different from the normal distribution, the points will deviate from the straight line. In this case, we can see that the t-distribution is different from the normal distribution, as the points deviate from the straight line. What would happen if we increased the sample size? The t-distribution would approach the normal distribution, and the points would fall closer and closer to the straight line.

```
qqplot(z10k, t10k)
abline(0, 1)
```

9. The *t*-statistic and *t*-distribution



9.4. Summary of *t*-distribution vs normal distribution

The *t*-distribution is a family of probability distributions that depends on a parameter called degrees of freedom, which is related to the sample size. The *t*-distribution approaches the standard normal distribution as the sample size increases but has heavier tails for smaller sample sizes. This means that the *t*-distribution is more conservative in calculating *p*-values for small samples, making it harder to reject the null hypothesis. Including an estimate of the standard deviation changes the way we calculate *p*-values by switching from the standard normal distribution to the *t*-distribution, which accounts for the uncertainty introduced by estimating the population standard deviation from the sample. This adjustment is particularly important for small sample sizes, as it provides a more accurate assessment of the statistical significance of our results.

9.5. t.test

9.5.1. One-sample

```
x = rnorm(20,1)
# small sample
t.test(x[1:5])
```

One Sample t-test

```
data:  x[1:5]
t = 0.97599, df = 4, p-value = 0.3843
alternative hypothesis: true mean is not equal to 0
95 percent confidence interval:
 -1.029600  2.145843
sample estimates:
mean of x
0.5581214
```

Increase sample size:

```
t.test(x[1:20])
```

One Sample t-test

```
data:  x[1:20]
t = 3.8245, df = 19, p-value = 0.001144
alternative hypothesis: true mean is not equal to 0
95 percent confidence interval:
 0.3541055 1.2101894
sample estimates:
mean of x
0.7821474
```


9. The *t*-statistic and *t*-distribution

9.5.2. two-sample

```
x = rnorm(10,0.5)
y = rnorm(10,-0.5)
t.test(x,y)
```

Welch Two Sample t-test

```
data:  x and y
t = 3.4296, df = 17.926, p-value = 0.003003
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 0.5811367 2.4204048
sample estimates:
mean of x  mean of y
0.7039205 -0.7968502
```

9.5.3. from a data.frame

```
df = data.frame(value=c(x,y),group=as.factor(rep(c('g1','g2'),each=10)))
t.test(value ~ group, data=df)
```

Welch Two Sample t-test

```
data:  value by group
t = 3.4296, df = 17.926, p-value = 0.003003
alternative hypothesis: true difference in means between group g1 and group g2 is not equal to 0
95 percent confidence interval:
 0.5811367 2.4204048
sample estimates:
mean in group g1 mean in group g2
 0.7039205      -0.7968502
```

9.5.4. Equivalence to linear model

```
t.test(value ~ group, data=df, var.equal=TRUE)
```

Two Sample t-test

```
data: value by group
t = 3.4296, df = 18, p-value = 0.002989
alternative hypothesis: true difference in means between group g1 and group g2 is not equal to 0
95 percent confidence interval:
 0.5814078 2.4201337
sample estimates:
mean in group g1 mean in group g2
    0.7039205      -0.7968502
```

This is *equivalent* to:

```
res = lm(value ~ group, data=df)
summary(res)
```

Call:

```
lm(formula = value ~ group, data = df)
```

Residuals:

	Min	1Q	Median	3Q	Max
	-1.9723	-0.5600	0.2511	0.5252	1.3889

Coefficients:

	Estimate	Std. Error	t value	Pr(> t)
(Intercept)	0.7039	0.3094	2.275	0.03538 *
groupg2	-1.5008	0.4376	-3.430	0.00299 **

Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.9785 on 18 degrees of freedom
Multiple R-squared: 0.3952, Adjusted R-squared: 0.3616
F-statistic: 11.76 on 1 and 18 DF, p-value: 0.002989

10. K-means clustering

10.1. History of the k-means algorithm

The k-means clustering algorithm was first proposed by Stuart Lloyd in 1957 as a technique for pulse-code modulation. However, it was not published until 1982. In 1965, Edward W. Forgy published an essentially identical method, which became widely known as the k-means algorithm. Since then, k-means clustering has become one of the most popular unsupervised learning techniques in data analysis and machine learning.

K-means clustering is a method for finding patterns or groups in a dataset. It is an unsupervised learning technique, meaning that it doesn't rely on previously labeled data for training. Instead, it identifies structures or patterns directly from the data based on the similarity between data points (see Figure 10.1).

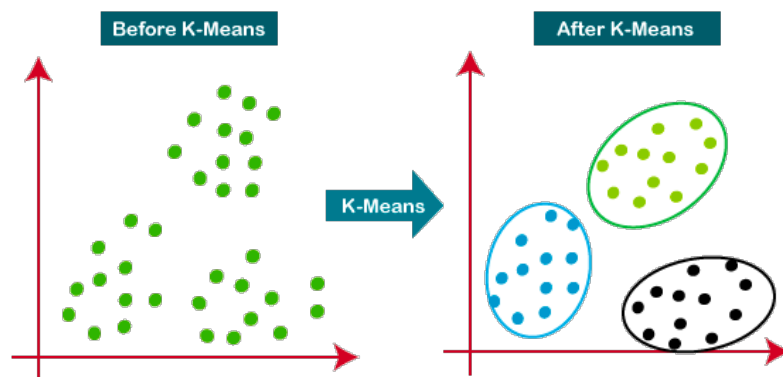


Figure 10.1.: K-means clustering takes a dataset and divides it into k clusters.

In simple terms, k-means clustering aims to divide a dataset into k distinct groups or clusters, where each data point belongs to the cluster with the nearest mean (average). The goal is to minimize the

10. K-means clustering

variability within each cluster while maximizing the differences between clusters. This helps to reveal hidden patterns or relationships in the data that might not be apparent otherwise.

10.2. The k-means algorithm

The k-means algorithm follows these general steps:

1. Choose the number of clusters k .
2. Initialize the cluster centroids randomly by selecting k data points from the dataset.
3. Assign each data point to the nearest centroid.
4. Update the centroids by computing the mean of all the data points assigned to each centroid.
5. Repeat steps 3 and 4 until the centroids no longer change or a certain stopping criterion is met (e.g., a maximum number of iterations).

The algorithm converges when the centroids stabilize or no longer change significantly. The final clusters represent the underlying patterns or structures in the data. Advantages and disadvantages of k-means clustering

10.3. Pros and cons of k-means clustering

Compared to other clustering algorithms, k-means has several advantages:

- **Simplicity and ease of implementation** The k-means algorithm is relatively straightforward and can be easily implemented, even for large datasets.
- **Scalability** The algorithm can be adapted for large datasets using various optimization techniques or parallel processing.
- **Speed** K-means is generally faster than other clustering algorithms, especially when the number of clusters k is small.

10. *K-means clustering*

- **Interpretability** The results of k-means clustering are easy to understand, as the algorithm assigns each data point to a specific cluster based on its similarity to the cluster's centroid.

However, k-means clustering has several disadvantages as well:

- **Choice of k** Selecting the appropriate number of clusters can be challenging and often requires domain knowledge or experimentation. A poor choice of k may yield poor results.
- **Sensitivity to initial conditions** The algorithm's results can vary depending on the initial placement of centroids. To overcome this issue, the algorithm can be run multiple times with different initializations and the best solution can be chosen based on a criterion (e.g., minimizing within-cluster variation).
- **Assumes spherical clusters** K-means assumes that clusters are spherical and evenly sized, which may not always be the case in real-world datasets. This can lead to poor performance if the underlying clusters have different shapes or densities.
- **Sensitivity to outliers** The algorithm is sensitive to outliers, which can heavily influence the position of centroids and the final clustering result. Preprocessing the data to remove or mitigate the impact of outliers can help improve the performance of k-means clustering.

Despite limitations, k-means clustering remains a popular and widely used method for exploring and analyzing data, particularly in biological data analysis, where identifying patterns and relationships can provide valuable insights into complex systems and processes.

10.4. An example of k-means clustering

10.4.1. The data and experimental background

The data we are going to use are from DeRisi, Iyer, and Brown (1997). From their abstract:

DNA microarrays containing virtually every gene of *Saccharomyces cerevisiae* were used to carry out a comprehensive investigation of the temporal program of gene expression accompanying the metabolic shift from fermentation to respiration. The expression profiles observed for genes with known metabolic functions pointed to features of the metabolic re-programming that occur during the diauxic shift, and the expression patterns of many previously uncharacterized genes provided clues to their possible functions.

These data are available from NCBI GEO as [GSE28](#).

In the case of the baker's or brewer's yeast *Saccharomyces cerevisiae* growing on glucose with plenty of aeration, the diauxic growth pattern is commonly observed in batch culture. During the first growth phase, when there is plenty of glucose and oxygen available, the yeast cells prefer glucose fermentation to aerobic respiration even though aerobic respiration is the more efficient pathway to grow on glucose. This experiment profiles gene expression for 6400 genes over a time course during which the cells are undergoing a [diauxic shift](#).

The data in deRisi et al. have no replicates and are time course data. Sometimes, seeing how groups of genes behave can give biological insight into the experimental system or the function of individual genes. We can use clustering to group genes that have a similar expression pattern over time and then potentially look at the genes that do so.

Our goal, then, is to use kmeans clustering to divide highly variable (informative) genes into groups and then to visualize those groups.

10.5. Getting data

These data were deposited at NCBI GEO back in 2002. GEOquery can pull them out easily.

```
library(GEOquery)
gse = getGEO("GSE28")[[1]]
class(gse)
```

```
[1] "ExpressionSet"
attr(,"package")
[1] "Biobase"
```

GEOquery is a little dated and was written before the Summarized-Experiment existed. However, Bioconductor makes a conversion from the old ExpressionSet that GEOquery uses to the Summarized-Experiment that we see so commonly used now.

```
library(SummarizedExperiment)
gse = as(gse, "SummarizedExperiment")
gse
```

```
class: SummarizedExperiment
dim: 6400 7
metadata(3): experimentData annotation protocolData
assays(1): exprs
rownames(6400): 1 2 ... 6399 6400
rowData names(20): ID ORF ... FAILED IS_CONTAMINATED
colnames(7): GSM887 GSM888 ... GSM892 GSM893
colData names(33): title geo_accession ... supplementary_file
                  data_row_count
```

Taking a quick look at the colData(), it might be that we want to reorder the columns a bit.

```
colData(gse)$title
```

10. *K-means clustering*

```
[1] "diauxic shift timecourse: 15.5 hr" "diauxic shift timecourse: 0 hr"  
[3] "diauxic shift timecourse: 18.5 hr" "diauxic shift timecourse: 9.5 hr"  
[5] "diauxic shift timecourse: 11.5 hr" "diauxic shift timecourse: 13.5 hr"  
[7] "diauxic shift timecourse: 20.5 hr"
```

So, we can reorder by hand to get the time course correct:

```
gse = gse[, c(2,4,5,6,1,3,7)]
```

10.6. Preprocessing

In gene expression data analysis, the primary objective is often to identify genes that exhibit significant differences in expression levels across various conditions, such as diseased vs. healthy samples or different time points in a time-course experiment. However, gene expression datasets are typically large, noisy, and contain numerous genes that do not exhibit substantial changes in expression levels. Analyzing all genes in the dataset can be computationally intensive and may introduce noise or false positives in the results.

One common approach to reduce the complexity of the dataset and focus on the most informative genes is to subset the genes based on their standard deviation in expression levels across the samples. The standard deviation is a measure of dispersion or variability in the data, and genes with high standard deviations have more variation in their expression levels across the samples.

By selecting genes with high standard deviations, we focus on genes that show relatively large changes in expression levels across different conditions. These genes are more likely to be biologically relevant and involved in the underlying processes or pathways of interest. In contrast, genes with low standard deviations exhibit little or no change in expression levels and are less likely to be informative for the analysis. It turns out that applying filtering based on criteria such as standard deviation can also increase power and reduce false positives in the analysis (Bourgon, Gentleman, and Huber 2010).

To subset the genes for analysis based on their standard deviation, the following steps can be followed: Calculate the standard deviation of each gene's expression levels across all samples. Set a threshold for the standard deviation, which can be determined based on

10. K-means clustering

domain knowledge, data distribution, or a specific percentile of the standard deviation values (e.g., selecting the top 10% or 25% of genes with the highest standard deviations). Retain only the genes with a standard deviation above the chosen threshold for further analysis.

By subsetting the genes based on their standard deviation, we can reduce the complexity of the dataset, speed up the subsequent analysis, and increase the likelihood of detecting biologically meaningful patterns and relationships in the gene expression data. The threshold for the standard deviation cutoff is rather arbitrary, so it may be beneficial to try a few to check for sensitivity of findings.

```
sds = apply(assays(gse)[[1]], 1, sd)
hist(sds)
```

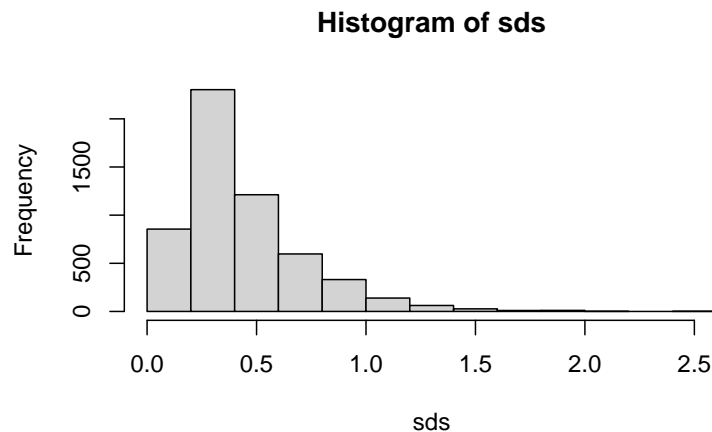


Figure 10.2.: Histogram of standard deviations for all genes in the deRisi dataset.

Examining the plot, we can see that the most highly variable genes have an $sd > 0.8$ or so (arbitrary). We can, for convenience, create a new SummarizedExperiment that contains only our most highly variable genes.

```
idx = sds > 0.8 & !is.na(sds)
gse_sub = gse[idx,]
```

10.7. Clustering

Now, `gse_sub` contains a subset of our data.

The `kmeans` function takes a matrix and the number of clusters as arguments.

```
k = 4
km = kmeans(assays(gse_sub)[[1]], 4)
```

The `km` `kmeans` result contains a vector, `km$cluster`, which gives the cluster associated with each gene. We can plot the genes for each cluster to see how these different genes behave.

```
expression_values = assays(gse_sub)[[1]]
par(mfrow=c(2,2), mar=c(3,4,1,2)) # this allows multiple plots per page
for(i in 1:k) {
  matplot(t(expression_values[km$cluster==i, ]), type='l', ylim=c(-3,3),
          ylab = paste("cluster", i))
}
```

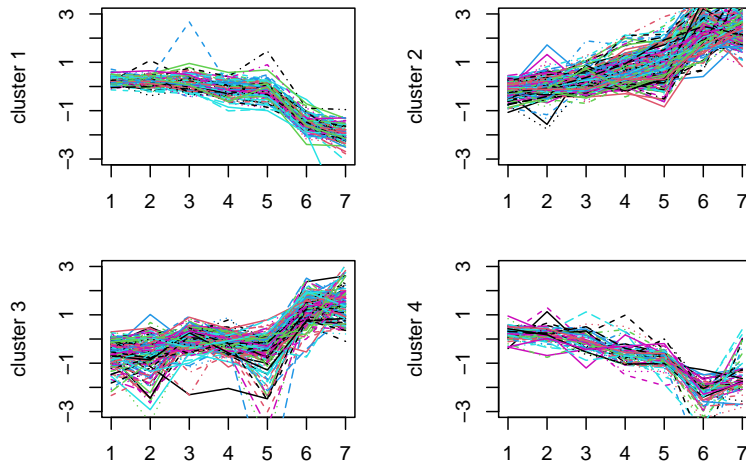


Figure 10.3.: Gene expression profiles for the four clusters identified by k-means clustering. Each line represents a gene in the cluster, and each column represents a time point in the experiment. Each cluster shows a distinct trend where the genes in the cluster are potentially co-regulated.

Try this with different size `k`. Perhaps go back to choose more genes (using a smaller cutoff for `sd`).

10.8. Summary

In this lesson, we have learned how to use k-means clustering to identify groups of genes that behave similarly over time. We have also learned how to subset our data to focus on the most informative genes.

Part V.

Bioconductor

11. Introduction to SummarizedExperiment

The `SummarizedExperiment` class is used to store rectangular matrices of experimental results, which are commonly produced by sequencing and microarray experiments. Each object stores observations of one or more samples, along with additional meta-data describing both the observations (features) and samples (phenotypes).

A key aspect of the `SummarizedExperiment` class is the coordination of the meta-data and assays when subsetting. For example, if you want to exclude a given sample you can do for both the meta-data and assay in one operation, which ensures the meta-data and observed data will remain in sync. Improperly accounting for meta and observational data has resulted in a number of incorrect results and retractions so this is a very desirable property.

`SummarizedExperiment` is in many ways similar to the historical `ExpressionSet`, the main distinction being that `SummarizedExperiment` is more flexible in its row information, allowing both `GRanges` based as well as those described by arbitrary `DataFrames`. This makes it ideally suited to a variety of experiments, particularly sequencing based experiments such as RNA-Seq and ChIP-Seq.

```
BiocManager::install('airway')
BiocManager::install('SummarizedExperiment')
```

11.1. Anatomy of a SummarizedExperiment

The *SummarizedExperiment* package contains two classes: `SummarizedExperiment` and `RangedSummarizedExperiment`.

11. Introduction to SummarizedExperiment

SummarizedExperiment is a matrix-like container where rows represent features of interest (e.g. genes, transcripts, exons, etc.) and columns represent samples. The objects contain one or more assays, each represented by a matrix-like object of numeric or other mode. The rows of a SummarizedExperiment object represent features of interest. Information about these features is stored in a DataFrame object, accessible using the function `rowData()`. Each row of the DataFrame provides information on the feature in the corresponding row of the SummarizedExperiment object. Columns of the DataFrame represent different attributes of the features of interest, e.g., gene or transcript IDs, etc.

RangedSummarizedExperiment is the “child” of the SummarizedExperiment class which means that all the methods on SummarizedExperiment also work on a RangedSummarizedExperiment.

The fundamental difference between the two classes is that the rows of a RangedSummarizedExperiment object represent genomic ranges of interest instead of a DataFrame of features. The RangedSummarizedExperiment ranges are described by a GRanges or a GRangesList object, accessible using the `rowRanges()` function.

Figure 11.1 displays the class geometry and highlights the vertical (column) and horizontal (row) relationships.

11.1.1. Assays

The airway package contains an example dataset from an RNA-Seq experiment of read counts per gene for airway smooth muscles. These data are stored in a RangedSummarizedExperiment object which contains 8 different experimental and assays 64,102 gene transcripts.

Loading required package: airway

```
library(SummarizedExperiment)
data(airway, package="airway")
se <- airway
se
```

11. Introduction to SummarizedExperiment

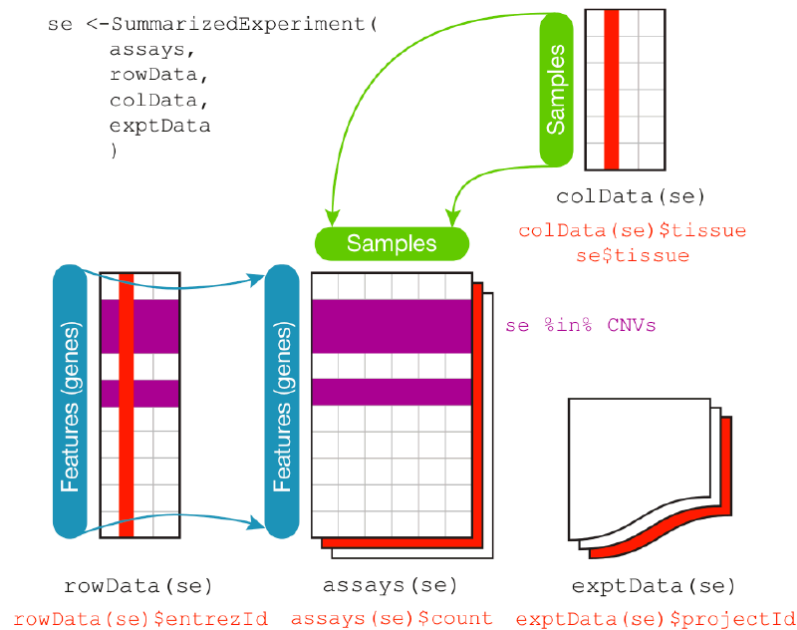


Figure 11.1.: Summarized Experiment. There are three main components, the colData(), the rowData() and the assays(). The accessors for the various parts of a complete SummarizedExperiment object match the names.

```
class: RangedSummarizedExperiment
dim: 63677 8
metadata(1): ''
assays(1): counts
rownames(63677): ENSG000000000003 ENSG000000000005 ... ENSG00000273492
ENSG00000273493
rowData names(10): gene_id gene_name ... seq_coord_system symbol
colnames(8): SRR1039508 SRR1039509 ... SRR1039520 SRR1039521
colData names(9): SampleName cell ... Sample BioSample
```

To retrieve the experiment data from a SummarizedExperiment object one can use the assays() accessor. An object can have multiple assay datasets each of which can be accessed using the \$ operator. The airway dataset contains only one assay (counts). Here each row represents a gene transcript and each column one of the samples.

```
assays(se)$counts
```

11. Introduction to SummarizedExperiment

[illegible]

11.1.2. 'Row' (regions-of-interest) data

The `rowRanges()` accessor is used to view the range information for a `RangedSummarizedExperiment`. (Note if this were the parent `SummarizedExperiment` class we'd use `rowData()`). The data are stored in a `GRangesList` object, where each list element corresponds to one gene transcript and the ranges in each `GRanges` correspond to the exons in the transcript.

```
rowRanges(se)
```

```
GRangesList object of length 63677:
```

\$ENSG00000000003

GRanges object with 17 ranges and 2 metadata columns:

seqnames		ranges	strand	exon_id	exon_name
<Rle>	<IRanges>	<Rle>	<integer>	<character>	
[1]	X 99883667-99884983	-	667145	ENSE00001459322	
[2]	X 99885756-99885863	-	667146	ENSE00000868868	
[3]	X 99887482-99887565	-	667147	ENSE00000401072	
[4]	X 99887538-99887565	-	667148	ENSE00001849132	
[5]	X 99888402-99888536	-	667149	ENSE00003554016	
...
[13]	X 99890555-99890743	-	667156	ENSE00003512331	
[14]	X 99891188-99891686	-	667158	ENSE00001886883	
[15]	X 99891605-99891803	-	667159	ENSE00001855382	
[16]	X 99891790-99892101	-	667160	ENSE00001863395	

11. Introduction to SummarizedExperiment

```
[17]      X 99894942-99894988      - |    667161 ENSE00001828996
-----
seqinfo: 722 sequences (1 circular) from an unspecified genome

...
<63676 more elements>
```

11.1.3. 'Column' (sample) data

Sample meta-data describing the samples can be accessed using `colData()`, and is a `DataFrame` that can store any number of descriptive columns for each sample row.

```
colData(se)
```

`DataFrame` with 8 rows and 9 columns

	SampleName	cell	dex	albut	Run	avgLength
	<factor>	<factor>	<factor>	<factor>	<factor>	<integer>
SRR1039508	GSM1275862	N61311	untrt	untrt	SRR1039508	126
SRR1039509	GSM1275863	N61311	trt	untrt	SRR1039509	126
SRR1039512	GSM1275866	N052611	untrt	untrt	SRR1039512	126
SRR1039513	GSM1275867	N052611	trt	untrt	SRR1039513	87
SRR1039516	GSM1275870	N080611	untrt	untrt	SRR1039516	120
SRR1039517	GSM1275871	N080611	trt	untrt	SRR1039517	126
SRR1039520	GSM1275874	N061011	untrt	untrt	SRR1039520	101
SRR1039521	GSM1275875	N061011	trt	untrt	SRR1039521	98

	Experiment	Sample	BioSample
	<factor>	<factor>	<factor>
SRR1039508	SRX384345	SRS508568	SAMN02422669
SRR1039509	SRX384346	SRS508567	SAMN02422675
SRR1039512	SRX384349	SRS508571	SAMN02422678
SRR1039513	SRX384350	SRS508572	SAMN02422670
SRR1039516	SRX384353	SRS508575	SAMN02422682
SRR1039517	SRX384354	SRS508576	SAMN02422673
SRR1039520	SRX384357	SRS508579	SAMN02422683
SRR1039521	SRX384358	SRS508580	SAMN02422677

This sample metadata can be accessed using the `$` accessor which makes it easy to subset the entire object by a given phenotype.

11. Introduction to SummarizedExperiment

```
# subset for only those samples treated with dexamethasone
se[, se$dex == "trt"]
```

```
class: RangedSummarizedExperiment
dim: 63677 4
metadata(1): ''
assays(1): counts
rownames(63677): ENSG000000000003 ENSG000000000005 ... ENSG00000273492
               ENSG00000273493
rowData names(10): gene_id gene_name ... seq_coord_system symbol
colnames(4): SRR1039509 SRR1039513 SRR1039517 SRR1039521
colData names(9): SampleName cell ... Sample BioSample
```

11.1.4. Experiment-wide metadata

Meta-data describing the experimental methods and publication references can be accessed using `metadata()`.

```
metadata(se)
```

```
[[1]]
```

Experiment data

 Experimenter name: Himes BE

 Laboratory: NA

 Contact information:

 Title: RNA-Seq transcriptome profiling identifies CRISPLD2 as a glucocorticoid responsive gene that mo

 URL: <http://www.ncbi.nlm.nih.gov/pubmed/24926665>

 PMIDs: 24926665

Abstract: A 226 word abstract is available. Use 'abstract' method.

Note that `metadata()` is just a simple list, so it is appropriate for *any* experiment wide metadata the user wishes to save, such as storing model formulas.

```
metadata(se)$formula <- counts ~ dex + albut
```

```
metadata(se)
```

11. Introduction to SummarizedExperiment

```
[[1]]
```

Experiment data

 Experimenter name: Himes BE

 Laboratory: NA

 Contact information:

 Title: RNA-Seq transcriptome profiling identifies CRISPLD2 as a glucocorticoid responsive gene that mo

 URL: <http://www.ncbi.nlm.nih.gov/pubmed/24926665>

 PMIDs: 24926665

Abstract: A 226 word abstract is available. Use 'abstract' method.

```
$formula
```

```
counts ~ dex + albut
```

11.2. Common operations on SummarizedExperiment

11.2.1. Subsetting

- [Performs two dimensional subsetting, just like subsetting a matrix or data frame.

```
# subset the first five transcripts and first three samples
se[1:5, 1:3]
```

```
class: RangedSummarizedExperiment
dim: 5 3
metadata(2): '' formula
assays(1): counts
rownames(5): ENSG00000000003 ENSG00000000005 ENSG00000000419
             ENSG00000000457 ENSG00000000460
rowData names(10): gene_id gene_name ... seq_coord_system symbol
colnames(3): SRR1039508 SRR1039509 SRR1039512
colData names(9): SampleName cell ... Sample BioSample
```

- \$ operates on colData() columns, for easy sample extraction.

11. Introduction to SummarizedExperiment

```
se[, se$cell == "N61311"]
```

```
class: RangedSummarizedExperiment
dim: 63677 2
metadata(2): ' ' formula
assays(1): counts
rownames(63677): ENSG000000000003 ENSG000000000005 ... ENSG00000273492
               ENSG00000273493
rowData names(10): gene_id gene_name ... seq_coord_system symbol
colnames(2): SRR1039508 SRR1039509
colData names(9): SampleName cell ... Sample BioSample
```

11.2.2. Getters and setters

- `rowRanges()` / `(rowData()), colData(), metadata()`

```
counts <- matrix(1:15, 5, 3, dimnames=list(LETTERS[1:5], LETTERS[1:3]))

dates <- SummarizedExperiment(assays=list(counts=counts),
                             rowData=DataFrame(month=month.name[1:5], day=1:5))

# Subset all January assays
dates[rowData(dates)$month == "January", ]
```

```
class: SummarizedExperiment
dim: 1 3
metadata(0):
assays(1): counts
rownames(1): A
rowData names(2): month day
colnames(3): A B C
colData names(0):
```

- `assay()` versus `assays()` There are two accessor functions for extracting the assay data from a `SummarizedExperiment` object. `assays()` operates on the entire list of assay data as a whole, while `assay()` operates on only one assay at a time. `assay(x, i)` is simply a convenience function which is equivalent to `assays(x)[[i]]`.

11. Introduction to SummarizedExperiment

```
assays(se)
```

```
List of length 1  
names(1): counts
```

```
assays(se)[[1]][1:5, 1:5]
```

	SRR1039508	SRR1039509	SRR1039512	SRR1039513	SRR1039516
ENSG000000000003	679	448	873	408	1138
ENSG000000000005	0	0	0	0	0
ENSG000000000419	467	515	621	365	587
ENSG000000000457	260	211	263	164	245
ENSG000000000460	60	55	40	35	78

```
# assay defaults to the first assay if no i is given  
assay(se)[1:5, 1:5]
```

	SRR1039508	SRR1039509	SRR1039512	SRR1039513	SRR1039516
ENSG000000000003	679	448	873	408	1138
ENSG000000000005	0	0	0	0	0
ENSG000000000419	467	515	621	365	587
ENSG000000000457	260	211	263	164	245
ENSG000000000460	60	55	40	35	78

```
assay(se, 1)[1:5, 1:5]
```

	SRR1039508	SRR1039509	SRR1039512	SRR1039513	SRR1039516
ENSG000000000003	679	448	873	408	1138
ENSG000000000005	0	0	0	0	0
ENSG000000000419	467	515	621	365	587
ENSG000000000457	260	211	263	164	245
ENSG000000000460	60	55	40	35	78

11.2.3. Range-based operations

- `subsetByOverlaps()` `SummarizedExperiment` objects support all of the `findOverlaps()` methods and associated functions. This includes `subsetByOverlaps()`, which makes it easy to subset a `SummarizedExperiment` object by an interval.

In the next code block, we define a region of interest (or many regions of interest) and then subset our `SummarizedExperiment` by overlaps with this region.

```
# Subset for only rows which are in the interval 100,000 to 110,000 of
# chromosome 1
roi <- GRanges(seqnames="1", ranges=100000:1100000)
sub_se = subsetByOverlaps(se, roi)
sub_se
```

```
class: RangedSummarizedExperiment
dim: 74 8
metadata(2): '' formula
assays(1): counts
rownames(74): ENSG00000131591 ENSG00000177757 ... ENSG00000272512
           ENSG00000273443
rowData names(10): gene_id gene_name ... seq_coord_system symbol
colnames(8): SRR1039508 SRR1039509 ... SRR1039520 SRR1039521
colData names(9): SampleName cell ... Sample BioSample
```

```
dim(sub_se)
```

```
[1] 74 8
```

11.3. Constructing a SummarizedExperiment

Often, `SummarizedExperiment` or `RangedSummarizedExperiment` objects are returned by functions written by other packages. However it is possible to create them by hand with a call to the `SummarizedExperiment()` constructor. The code below is simply

11. Introduction to SummarizedExperiment

to illustrate the mechanics of creating an object from scratch. In practice, you will probably have the pieces of the object from other sources such as Excel files or csv files.

Constructing a RangedSummarizedExperiment with a GRanges as the *rowRanges* argument:

```
nrows <- 200
ncols <- 6
counts <- matrix(runif(nrows * ncols, 1, 1e4), nrows)
rowRanges <- GRanges(rep(c("chr1", "chr2"), c(50, 150)),
                     IRanges(floor(runif(200, 1e5, 1e6)), width=100),
                     strand=sample(c("+", "-"), 200, TRUE),
                     feature_id=sprintf("ID%03d", 1:200))
colData <- DataFrame(Treatment=rep(c("ChIP", "Input"), 3),
                     row.names=LETTERS[1:6])

SummarizedExperiment(assays=list(counts=counts),
                     rowRanges=rowRanges, colData=colData)
```

```
class: RangedSummarizedExperiment
dim: 200 6
metadata(0):
assays(1): counts
rownames: NULL
rowData names(1): feature_id
colnames(6): A B ... E F
colData names(1): Treatment
```

A SummarizedExperiment can be constructed with or without supplying a DataFrame for the *rowData* argument:

```
SummarizedExperiment(assays=list(counts=counts), colData=colData)
```

```
class: SummarizedExperiment
dim: 200 6
metadata(0):
assays(1): counts
rownames: NULL
rowData names(0):
```

11. Introduction to SummarizedExperiment

```
colnames(6): A B ... E F  
colData names(1): Treatment
```


References

- Bourgon, Richard, Robert Gentleman, and Wolfgang Huber. 2010. “Independent Filtering Increases Detection Power for High-Throughput Experiments.” *Proceedings of the National Academy of Sciences* 107 (21): 9546–51. <https://doi.org/10.1073/pnas.0914005107>.
- Center, Pew Research. 2016. “Lifelong Learning and Technology.” *Pew Research Center: Internet, Science & Tech*. <https://www.pewresearch.org/internet/2016/03/22/lifelong-learning-and-technology/>.
- DeRisi, J. L., V. R. Iyer, and P. O. Brown. 1997. “Exploring the Metabolic and Genetic Control of Gene Expression on a Genomic Scale.” *Science (New York, N.Y.)* 278 (5338): 680–86. <https://doi.org/10.1126/science.278.5338.680>.
- Student. 1908. “The Probable Error of a Mean.” *Biometrika* 6 (1): 1–25. <https://doi.org/10.2307/2331554>.

A. Appendix

A.1. Data Sets

- [BRFSS subset](#)
- [ALL clinical data](#)
- [ALL expression data](#)

A.2. Swirl

The following is from the [swirl website](#).

The swirl R package makes it fun and easy to learn R programming and data science. If you are new to R, have no fear.

To get started, we need to install a new package into R.

```
install.packages('swirl')
```

Once installed, we want to load it into the R workspace so we can use it.

```
library('swirl')
```

Finally, to get going, start swirl and follow the instructions.

```
swirl()
```

B. Additional resources

- [Base R Cheat Sheet](#)

Index

RStudio, [5](#)