# RPC 519 R and Bioconductor

Lori Kern

2025-10-28

# Table of contents

# Preface

This is a selection of material from **The RBioc Book** created by Sean Davis. The original full content may be viewed here. The contents of this book may have minor modifications or additions.

The material is modified and redistributed in accordance with the original Licensing.

Select modifications were inspired by RPC 520 content originally distributed by Martin Morgan.

# Part I

# Introduction

# 1 About R

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.

## 1.1 What is R?

R 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. 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.

## 1.2 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 ara 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 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 interfact 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. In other words, R is a powerful tool that can be used for a wide range of tasks, from small-scale data analysis to large-scale genomics and omics data science projects.

## 1.3 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).
- Additional languages are becoming increasingly popular for bioinformatics and biological data science, such as Python, Julia, and Rust.

## 1.4 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.

## 1.5 Working with R

R is a programming language, and as such, it requires you to write code to perform tasks. This can be intimidating for beginners, but it is also what makes R so powerful. In R, you can write scripts to automate tasks, create functions to encapsulate complex operations, and use packages to extend the functionality of R.

R can be used interactively or as a scripting language. In interactive mode, you can enter commands directly into the R console and see the results immediately. In scripting mode, you can write a series of commands in a script file and then execute the entire script at once. This allows you to save your work, reuse code, and share your analysis with others.

In the next section, we will discuss how to set up RStudio, an integrated development environment (IDE) for R that makes it easier to write and execute R code. However, you can use R without RStudio if you prefer to work in the R console or another IDE. RStudio is not required to use R, but it does provide a more user-friendly interface and several useful features that can enhance your R programming experience.

# 2 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.

## 2.1 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.

> **i** R versions
>
> RStudio works with all versions of R, but it is recommended to use the latest version of R to take advantage of the latest features and improvements. You can check your R version by running `version` (no parentheses)in the R console.
> You can check the latest version of R on the R-project website.

## 2.2 The RStudio Interface

RStudio's interface consists of four panels (see Figure 2.1):

- **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.
- **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.

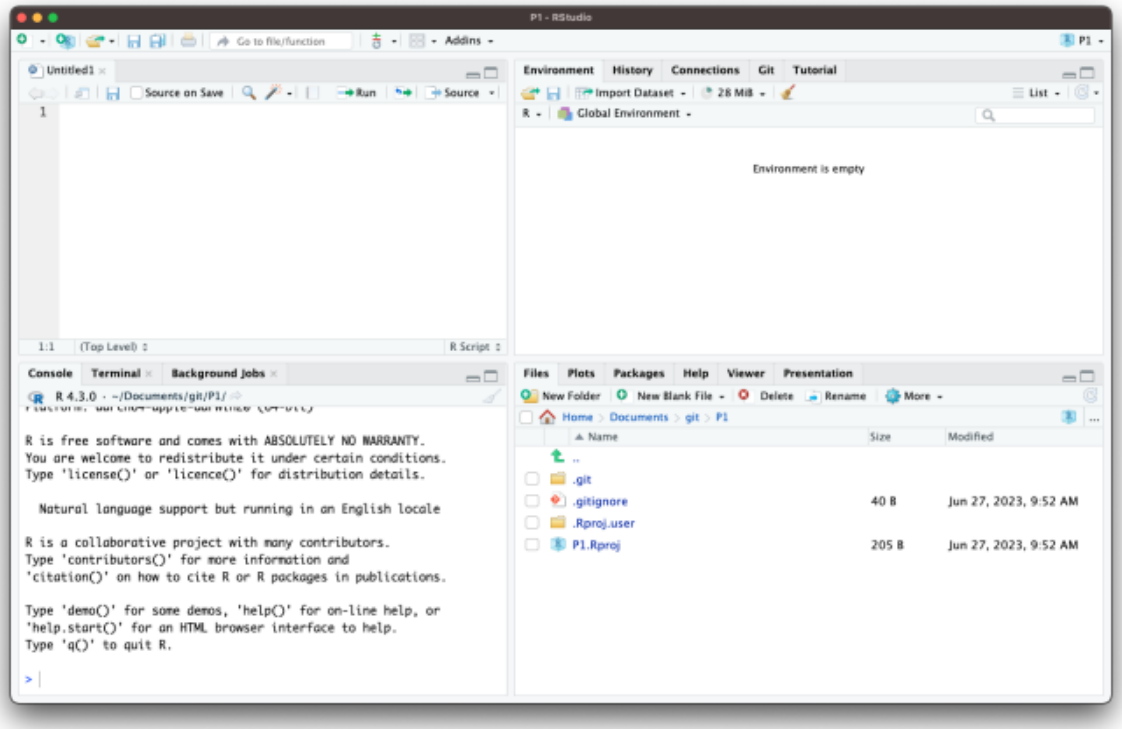



Figure 2.1: The RStudio interface. In this layout, the **source** pane is in the upper left, the **console** is in the lower left, the **environment** panel is in the top right and the **viewer/help/files** panel is in the bottom right.

> **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**. Here, you can change the theme, font size, and panel layout. You can also resize the panels as needed to gain screen real estate (see Figure 2.2).

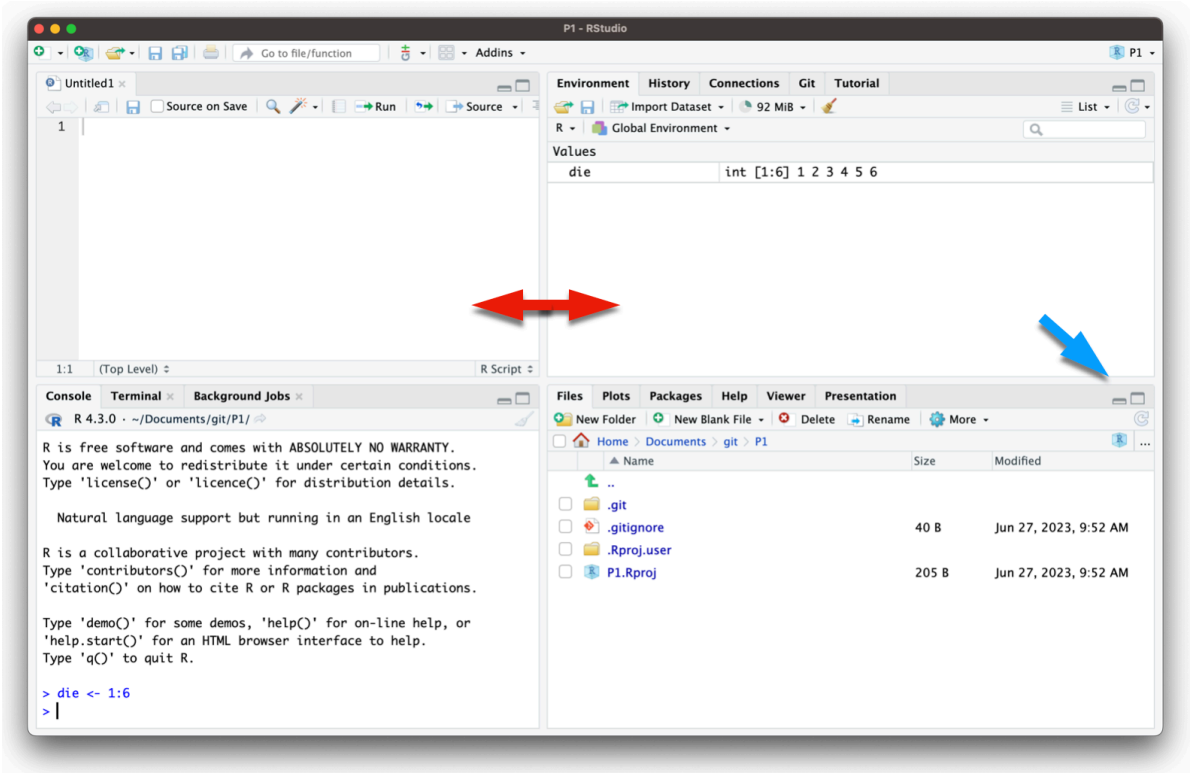



Figure 2.2: Dealing with limited screen real estate can be a challenge, particularly when you want to open another window to, for example, view a web page. You can resize the panes by sliding the center divider (red arrows) or by clicking on the minimize/maximize buttons (see blue arrow).

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.

## 2.3 Alternatives to RStudio

While RStudio is a popular choice for R development, there are several alternatives you can consider:

1. **Jupyter Notebooks**: Jupyter Notebooks provide an interactive environment for writing and executing R code, along with rich text support for documentation. You can use the IRKernel to run R code in Jupyter.

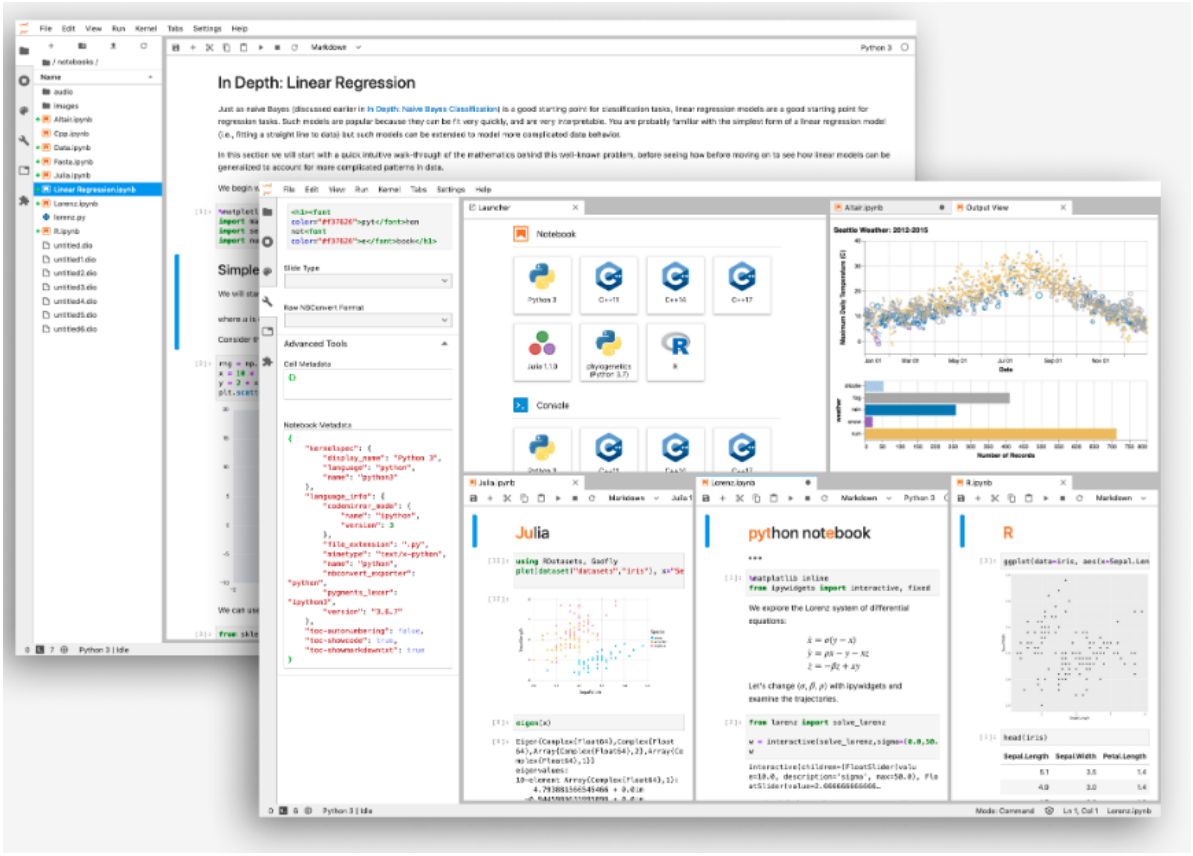



Figure 2.3: Jupyter Notebook interface. This is an interactive environment for writing and executing R code, along with rich text support for documentation.

2. **Visual Studio Code**: With the R extension for Visual Studio Code, you can write and execute R code in a lightweight editor. This setup provides features like syntax highlighting, code completion, and integrated terminal support.

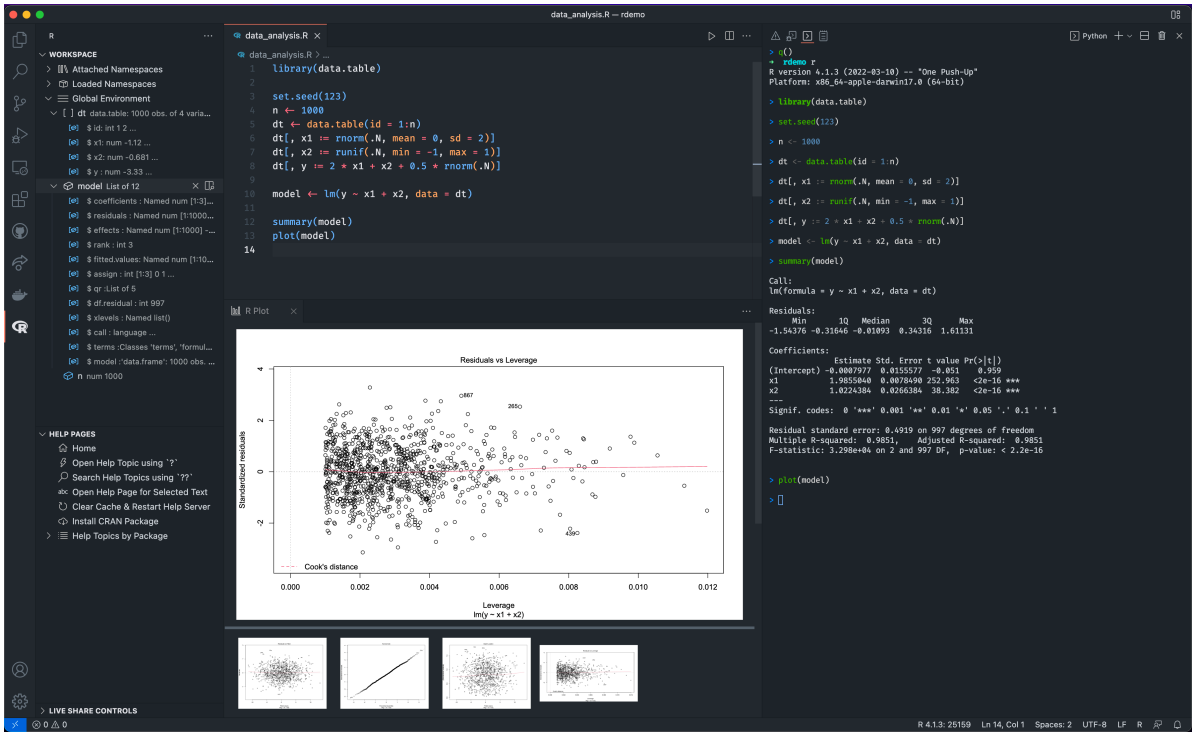



Figure 2.4: Visual Studio Code (VSCode) with the R extension. This is a lightweight alternative to RStudio that provides syntax highlighting, code completion, and integrated terminal support.

3. Positron Workbench: This is a commercial IDE that supports R and Python. It provides a similar interface to RStudio but with additional features for data science workflows, including support for multiple languages and cloud integration.

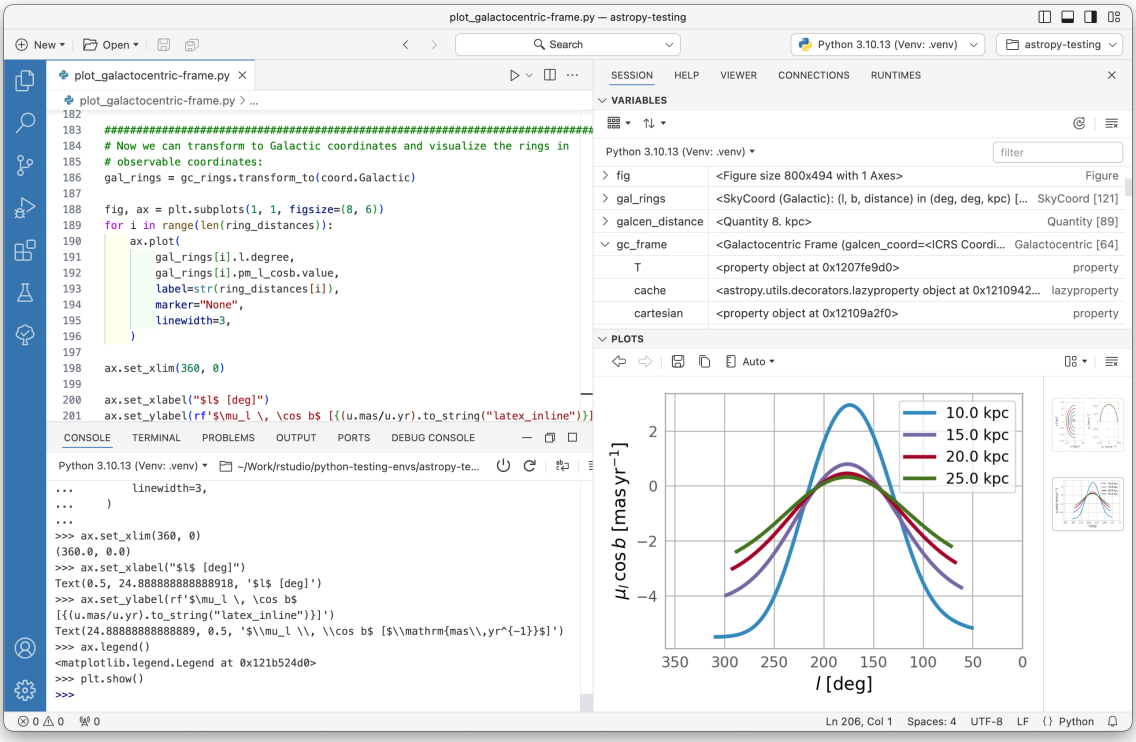



Figure 2.5: Positron Workbench interface. This IDE supports R and Python, providing a similar interface to RStudio with additional features for data science workflows.

4. **Command Line R**: For those who prefer a minimalistic approach, you can use R directly from the command line. This method lacks the GUI features of RStudio but can be efficient for quick tasks, scripting, automation, or when working on remote servers.

Each of these alternatives has its own strengths and weaknesses, so you may want to try a few to see which one best fits your workflow. All are available for free, and you can install them alongside RStudio if you wish to use multiple environments. Each can be installed in Windows, Mac, and Linux.

# 3 R mechanics

## 3.1 Starting R

We've installed R and RStudio. Now, let's start R and get going. 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).

## 3.2 *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
- Help
- Environment, History, and Files

## 3.3 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 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: `-`
- 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.

### 3.3.1 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
```

### 3.3.2 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.

> **What about `<-` and `=` for assignment?**
>
> The `<-` and `=` both work fine for assignment. You'll see both used and it is up to you to choose a standard for yourself. However, some programming communities, such as Bioconductor, will strongly suggest using the `<-` as it is clearer that it represents an *assignment* operation.

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
```

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!
```

## 3.4 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.

Examples of valid R names include:

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

## 3.5 About R functions

When you see a name followed by parentheses `()`, you are likely looking a name that represents an R function (or method, but we'll sidestep that distinction for now). Examples of R functions include `print()`, `help()`, and `ls()`. We haven't seen examples yet, but when a name is followed by `[]`, that name represents a variable of some kind and the `[]` are used for "subsetting" the variable. So:

- Name followed by `()` is a function.
- Name with `[]` means a variable that is being subset.

In many cases, when you see a new function used, you may not know what it does. The R `help()` function takes the name of another function and gives back the R help document for that function if there is one. The next section reviews that technique.

## 3.6 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
```

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.

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

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

## 3.7 Reflection

- Can you recognize the difference between *assignment* and *expressions* when interacting with R?
- Can you demonstrate an assignment to a variable?
- Do you know the rules for "names" in R?
- Are you able to get help using the R `help()` function?
- Do you know that functions are recognizable as names followed by `()`?

# 4 Up and Running with R

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.

## 4.1 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 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:

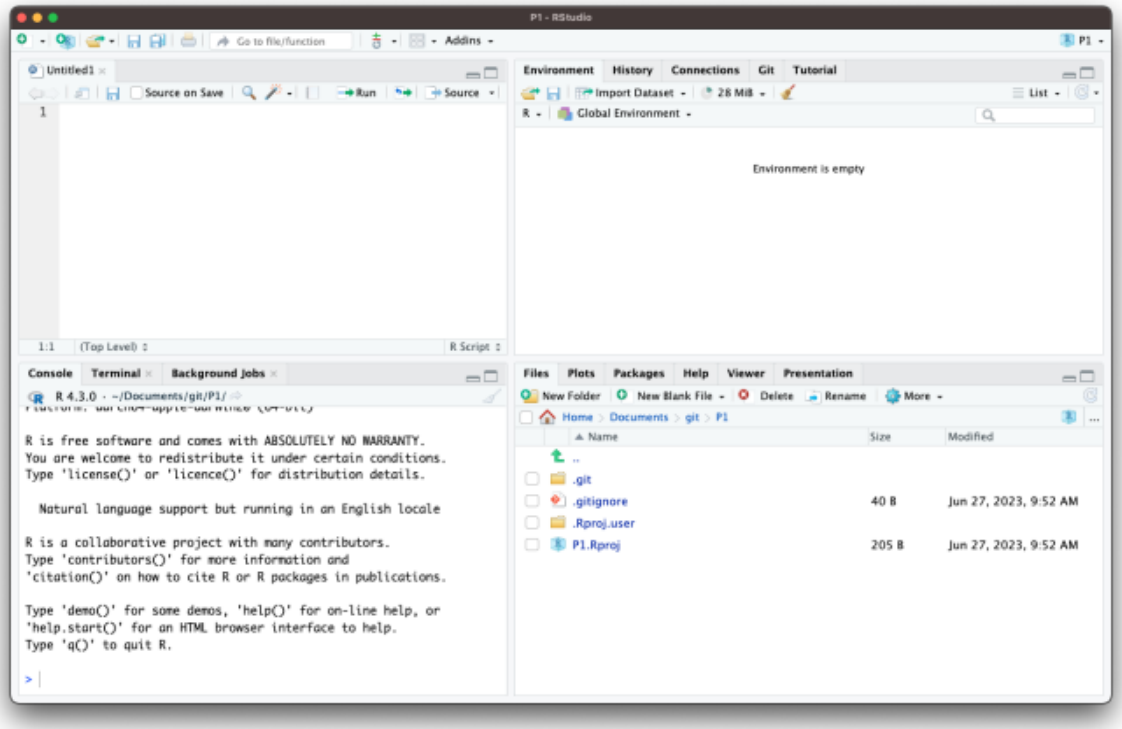



Figure 4.1: Your computer does your bidding when you type R commands at the prompt in the bottom line of the console pane. Don't forget to hit the Enter key. When you first open RStudio, the console appears in the pane on your left, but you can change this with **File > Tools > Global Options** in the menu bar.

```
> 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 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
```

```
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.

### 4.1.1 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.

```
10 + 2
```

```
[1] 12
```

```
12 * 3
```

```
[1] 36
```

```
36 - 6
```

```
[1] 30
```

```
30 / 3
```

```
[1] 10
```

## 4.2 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:

```
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
```

```
[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 4.2. 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 `*`:

| 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!

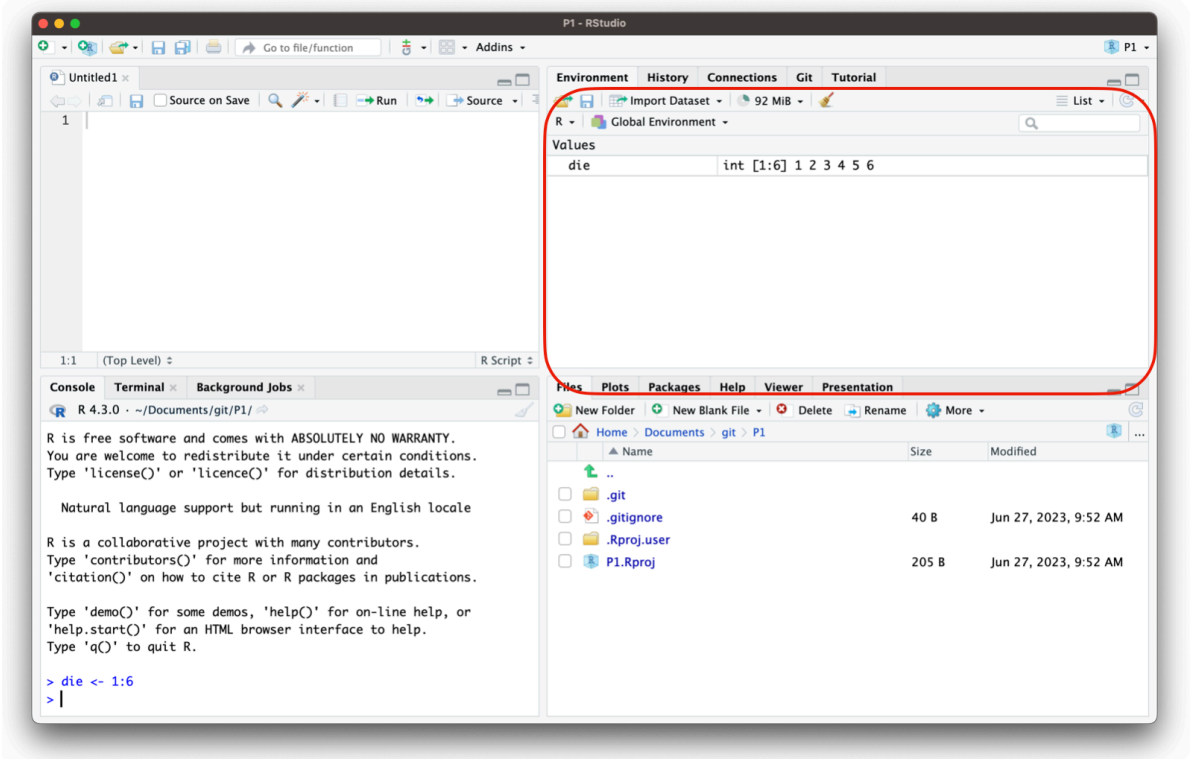



Figure 4.2: Assignment creates an object in the environment pane.

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
```

```
my_number <- 999
my_number
```

```
[1] 999
```

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

```
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 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 4.3}.

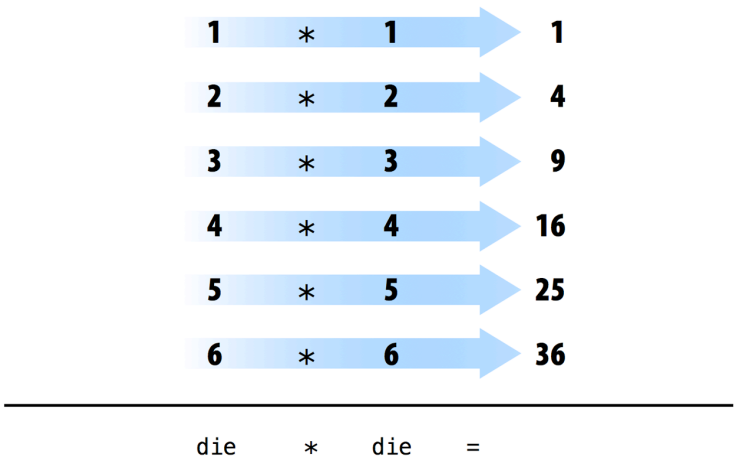



Figure 4.3: "When R performs element-wise execution, it matches up vectors and then manipulates each pair of elements independently."

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 4.4. 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
```

```
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
```

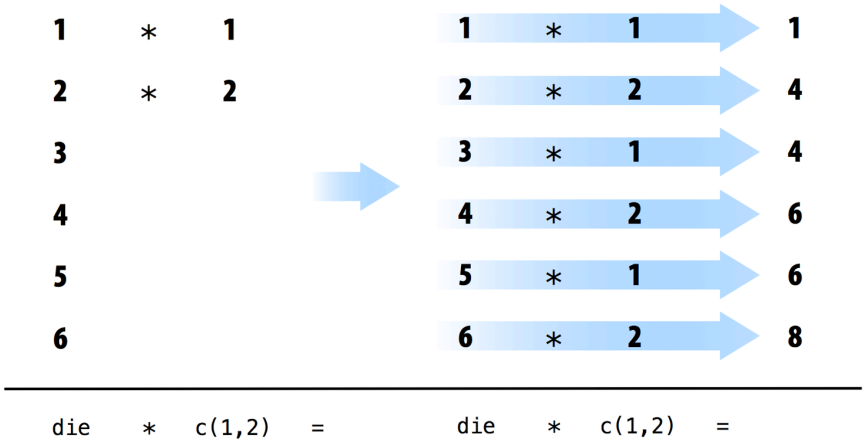



Figure 4.4: "R will repeat a short vector to do element-wise operations with two vectors of uneven lengths."

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:

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

## 4.3 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 4.5.

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

```
[1] 3.5
```

```
mean(die)
```

```
[1] 3.5
```

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

```
[1] 4
```

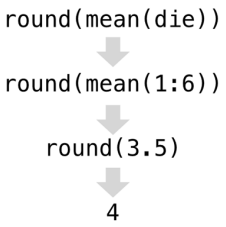



Figure 4.5: "When you link functions together, R will resolve them from the innermost operation to the outermost. Here R first looks up die, then calculates the mean of one through six, then rounds the mean."

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] 1 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] 5
```

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

```
[1] 1
```

```
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)
```

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
```

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] 2
```

### 4.3.1 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] 6 3
```

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] 4 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:

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

```
[1] 2 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] 5 1
```

```
sum(dice)
```

```
[1] 6
```

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] 5 1
```

```
dice
```

```
[1] 5 1
```

```
dice
```

```
[1] 5 1
```

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.

## 4.4 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] 6
```

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
```

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.

### 4.4.1 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 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] 10
```

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 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.

## 4.5 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
```

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] 5
```

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

```
[1] 4
```

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

```
[1] 4
```

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`:

```
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] 9
```

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 4.6.

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.

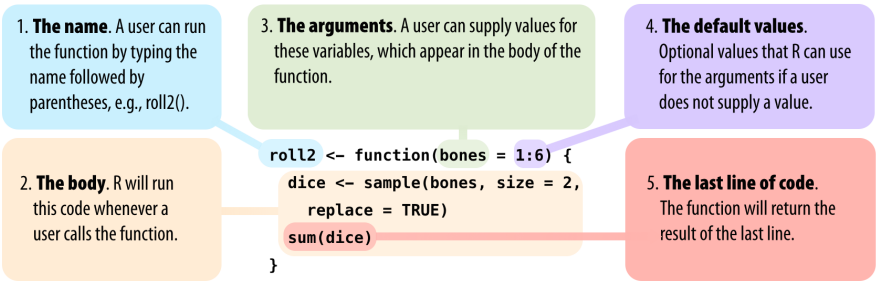



Figure 4.6: "Every function in R has the same parts, and you can use `function` to create these parts. Assign the result to a name, so you can call the function later."

## 4.6 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 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 4.7.

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.

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 button. 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.

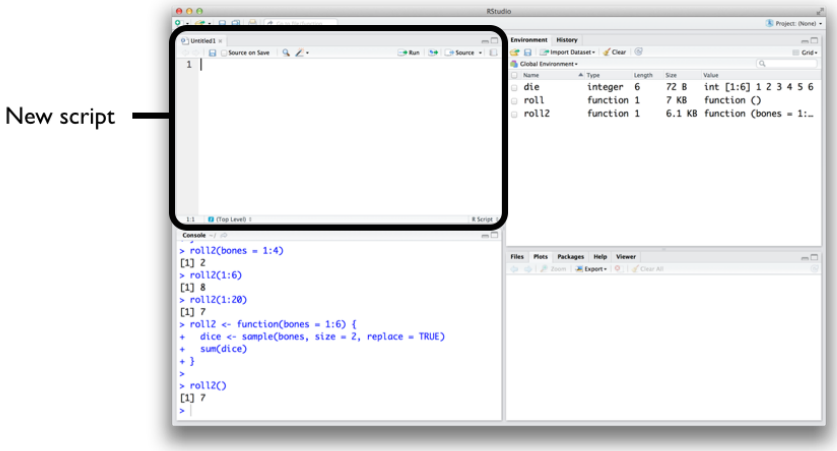



Figure 4.7: "When you open an R Script (File > New File > R Script in the menu bar), RStudio creates a fourth pane (or puts a new tab in the existing pane) above the console where you can write and edit your code."

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.

## 4.7 Summary

We'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.

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 memory, and you will use functions to automate tasks and do complicated calculations.

# 5 Packages

R is a powerful language for data science and programming, allowing beginners and experts alike to manipulate, analyze, and visualize data effectively. One of the most appealing features of R is its extensive library of packages, which are essential tools for expanding its capabilities and streamlining the coding process.

An R package is a collection of reusable functions, datasets, and compiled code created by other users and developers to extend the functionality of the base R language. These packages cover a wide range of applications, such as data manipulation, statistical analysis, machine learning, and data visualization. By utilizing existing R packages, you can leverage the expertise of others and save time by avoiding the need to create custom functions from scratch.

Using others' R packages is incredibly beneficial as it allows you to take advantage of the collective knowledge of the R community. Developers often create packages to address specific challenges, optimize performance, or implement popular algorithms or methodologies. By incorporating these packages into your projects, you can enhance your productivity, reduce development time, and ensure that you are using well-tested and reliable code.

## 5.1 Installing R packages

To install an R package, you can use the `install.packages()` function in the R console or script. For example, to install the popular data manipulation package "dplyr," simply type install.packages("dplyr"). This command will download the package from the Comprehensive R Archive Network (CRAN) and install it on your local machine. Keep in mind that you only need to install a package once, unless you want to update it to a newer version.

For those who are going to be using R for bioinformatics or biological data science, you will also want to install packages from Bioconductor, which is a repository of R packages specifically designed for bioinformatics and computational biology. To install Bioconductor packages, you can use the `BiocManager::install()` function.

To use this recommended approach, you first need to install the `BiocManager` package, which is the package manager for Bioconductor.

```
install.packages('BiocManager')
```

This is a one-time installation. After that, you can install any R, Bioconductor, rOpenSci, or even GitHub package using the `BiocManager::install()` function. For example, to install the `ggplot2` package, which is widely used for data visualization, you would run:

```
BiocManager::install("ggplot2")
```

## 5.2 Installing vs loading (library) R packages

After installing an R package, you will need to load it into your R session before using its functions. To load a package, use the `library()` function followed by the package name, such as `library(dplyr)`. Loading a package makes its functions and datasets available for use in your current R session. Note that you need to load a package every time you start a new R session.

```
library(ggplot2)
```

Now, the functionality of the *ggplot2* package is available in our R session.

Installing vs loading packages

The main thing to remember is that you only need to install a package once, but you need to load it with library each time you wish to use it in a new R session. R will unload all of its packages each time you close RStudio.


Figure 5.1: Installing vs loading R packages.

As in {Figure 5.1}, screw in the lightbulb (eg., `BiocManager::install`) only once and

then to use it, you need to turn on the switch each time you want to use it (`library`).

## 5.3 Finding R packages

Finding useful R packages can be done in several ways. First, browsing CRAN (https://cran.r-project.org/) and Bioconductor (https://bioconductor.org) are an excellent starting points, as they host thousands of packages categorized by topic. Additionally, online forums like Stack Overflow and R-bloggers can provide valuable recommendations based on user experiences. Social media platforms such as Twitter, where developers and data scientists often share new packages and updates, can also be a helpful resource. Finally, don't forget to ask your colleagues or fellow R users for their favorite packages, as they may have insights on which ones best suit your specific needs.

## 5.4 Creating a package

While it may seem overwhelming, creating a package can be fairly simple with the assistance of R packages that provide tips and templates. Some good starting points:

- devtools
- usethis
- biocthis
- roxygen2

# 6 Saving and Loading Workspaces and Objects

## 6.1 Rstudio Projects: Organizing Your Work

Before diving into reading and writing files, it's essential to understand how to organize your work effectively. RStudio Projects provide a powerful way to keep your files, scripts, and data organized in a self-contained workspace.

### 6.1.1 What are RStudio Projects?

An RStudio Project is a special folder that contains all the files associated with a particular analysis or research project. When you create a project, RStudio creates a `.Rproj` file that serves as the anchor for your project workspace. This approach offers several key benefits:

- **Consistent working directory**: The project folder automatically becomes your working directory
- **File organization**: All related files (scripts, data, outputs) are kept together
- **Reproducibility**: Others can easily run your code without worrying about file paths
- **Version control integration**: Projects work seamlessly with Git and GitHub

### 6.1.2 Creating an RStudio Project

You can create a new RStudio Project in several ways:

1. **File menu**: Go to `File > New Project...`
2. **Project dropdown**: Click the project dropdown in the top-right corner and select "New Project"
3. **Choose New Directory**: Create a project in a new folder.

When creating a project, you have three main options:

- **New Directory**: Create a fresh project folder
- **Existing Directory**: Turn an existing folder into a project
- **Version Control**: Clone a repository from GitHub or other version control systems

### 6.1.3 Project Structure Best Practices

A well-organized project typically follows a consistent structure (that YOU define). Here's a common structure that you might consider:

```
my_analysis_project/
  my_analysis_project.Rproj
  data/
      raw/
      processed/
  scripts/
  notebooks/
  outputs/
      figures/
      tables/
  README.md
  .gitignore
```

This structure separates raw data from processed data, keeps scripts organized, and provides clear locations for outputs.

### 6.1.4 Working Directories and File Paths

One of the most significant advantages of using RStudio Projects is that they solve the common problem of file path management. When you open a project, RStudio automatically sets the working directory to the project folder. This means:

```r
# Instead of using absolute paths like this:
df <- read.csv("/Users/username/Documents/my_analysis/data/dataset.csv")

# You can use relative paths like this:
df <- read.csv("data/dataset.csv")
```

Relative paths make your code portable—anyone who opens your project will be able to run your scripts without modifying file paths.

### 6.1.5 Projects and Reproducibility

RStudio Projects can play a key (but optional) role in creating reproducible analyses. When you share a project folder (or push it to GitHub), collaborators can:

1. Download/clone the entire project
2. Open the `.Rproj` file
3. Run your scripts without any setup or path modifications

This seamless workflow is essential for collaborative research and makes your work more credible and verifiable.

## 6.2 Saving and Loading Workspace

If you are not in Rstudio and want to save your workspace there are a few options. If you choose to save your session when you quit out of R with `q("yes")` or `q()` and selecting `yes`, it is also equivalent to the following using save

```
save(list = ls(all.names = TRUE), file =".RData", envir = .GlobalEnv)
```

This saves your workspace and command history. Any object assignments and command history (viewable with `history()`) will be available in your next R session. Your session will load automatically when you start up R in the same directory.

You can use the `save.image` option

```
save.image()
# or
save.image(file="descriptiveFileName.RData")
```

If you do not give `save.image` a file name, it will also load on default if you start an R session in that directory but with default settings will not have your command history.

If you give `save.image` a file name, it will not automatically load when you start R and will not have command history. You will have to load the image manually to see the objects in the new R session.

```
load("descriptiveFileName.RData")
```

## 6.3 Saving and Loading Objects

If you want to selectively save objects, the `save` function is utilized. This allows you to choose the objects you want saved. The `load` function would then load the selected objects into a new R session.

```r
ages = 1:4
months = c("may", "june", "july", "august")
vec = c(TRUE, FALSE, TRUE)
save(months, ages, file="subset.RData")
```

## 6.4 Saving and Loading Command History

When quitting out of R, if you save, it also saves your command history. To do this manually you can use the `savehistory`/`loadhistory` functions in R.

Now that we understand how to organize our work with RStudio Projects and how to save and load workspaces, let's explore how to read and write the data files that will live within these organized project structures.

# 7 Reading and writing data files

## 7.1 Introduction

In this chapter, we will discuss how to read and write data files in R. Data files are essential for storing and sharing data across different platforms and applications. R provides a variety of functions and packages to read and write data files in different formats, such as text files, CSV files, Excel files. By mastering these functions, you can efficiently import and export data in R, enabling you to perform data analysis and visualization tasks effectively.

## 7.2 CSV files

Comma-Separated Values (CSV) files are a common file format for storing tabular data. They consist of rows and columns, with each row representing a record and each column representing a variable or attribute. CSV files are widely used for data storage and exchange due to their simplicity and compatibility with various software applications. In R, you can read and write CSV files using the `read.csv()` and `write.csv()` functions, respectively. A commonly used alternative is to use the `readr` package, which provides faster and more user-friendly functions for reading and writing CSV files.

### 7.2.1 Writing a CSV file

Since we are going to use the `readr` package, we need to install it first. You can install the `readr` package using the following command:

```
install.packages("readr")
```

Once the package is installed, you can load it into your R session using the `library()` function:

```
library(readr)
```

Since we don't have a CSV file sitting around, let's create a simple data frame to write to a CSV file. Here's an example data frame:

```
df <- data.frame(
  id = c(1, 2, 3, 4, 5),
  name = c("Alice", "Bob", "Charlie", "David", "Eve"),
  age = c(25, 30, 35, 40, 45)
)
```

Now, you can write this data frame to a CSV file using the `write_csv()` function from the `readr` package. Here's how you can do it:

```
write_csv(df, "data.csv")
```

You can check the current working directory to see if the CSV file was created successfully. If you want to specify a different directory or file path, you can provide the full path in the `write_csv()` function.

```
# see what the current working directory is
getwd()
```

```
[1] "/home/lorikern/Projects/Papers_Reporting_Conferences/RBiocBook-book/RPC519RBioc"
```

```
# and check to see that the file was created
dir(pattern = "data.csv")
```

```
[1] "data.csv"
```

### 7.2.2 Reading a CSV file

Now that we have a CSV file, let's read it back into R using the `read_csv()` function from the `readr` package. Here's how you can do it:

```
df2 <- read_csv("data.csv")
```

```
Rows: 5 Columns: 3
-- Column specification --------------------------------------------------------
Delimiter: ","
chr (1): name
dbl (2): id, age

i Use `spec()` to retrieve the full column specification for this data.
i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

You can check the structure of the data frame `df2` to verify that the data was read correctly:

```
df2
```

```
# A tibble: 5 x 3
     id name      age
  <dbl> <chr>   <dbl>
1     1 Alice      25
2     2 Bob        30
3     3 Charlie    35
4     4 David      40
5     5 Eve        45
```

The `readr` package can read CSV files with various delimiters, headers, and data types, making it a versatile tool for handling tabular data in R. It can also read CSV files directly from web locations like so:

```
df3 <- read_csv("https://data.cdc.gov/resource/pwn4-m3yp.csv")
```

```
Rows: 1000 Columns: 10
-- Column specification --------------------------------------------------------
Delimiter: ","
chr  (1): state
dbl  (6): tot_cases, new_cases, tot_deaths, new_deaths, new_historic_cases, ...
dttm (3): date_updated, start_date, end_date

i Use `spec()` to retrieve the full column specification for this data.
i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

The dataset that you just downloaded is described here: Covid-19 data from CDC

## 7.3 Excel files

Microsoft Excel files are another common file format for storing tabular data. Excel files can contain multiple sheets, formulas, and formatting options, making them a popular choice for data storage and analysis. In R, you can read and write Excel files using the `readxl` package. This package provides functions to import and export data from Excel files, enabling you to work with Excel data in R.

### 7.3.1 Reading an Excel file

To read an Excel file in R, you need to install and load the `readxl` package. You can install the `readxl` package using the following command:

```
install.packages("readxl")
```

Once the package is installed, you can load it into your R session using the `library()` function:

```
library(readxl)
```

Now, you can read an Excel file using the `read_excel()` function from the `readxl` package. We don't have an excel file available, so let's download one from the internet. Here's an example:

```
download.file('https://www.w3resource.com/python-exercises/pandas/excel/SaleData.xlsx', 'Sale
```

Now, you can read the Excel file into R using the `read_excel()` function:

```
df_excel <- read_excel("SaleData.xlsx")
```

You can check the structure of the data frame `df_excel` to verify that the data was read correctly:

```
df_excel
```

```
# A tibble: 45 x 8
   OrderDate           Region  Manager SalesMan  Item  Units Unit_price Sale_amt
   <dttm>              <chr>   <chr>   <chr>     <chr> <dbl>      <dbl>    <dbl>
 1 2018-01-06 00:00:00 East    Martha  Alexander Tele~    95       1198   113810
 2 2018-01-23 00:00:00 Central Hermann Shelli    Home~    50        500    25000
 3 2018-02-09 00:00:00 Central Hermann Luis      Tele~    36       1198    43128
 4 2018-02-26 00:00:00 Central Timothy David     Cell~    27        225     6075
 5 2018-03-15 00:00:00 West    Timothy Stephen   Tele~    56       1198    67088
 6 2018-04-01 00:00:00 East    Martha  Alexander Home~    60        500    30000
 7 2018-04-18 00:00:00 Central Martha  Steven    Tele~    75       1198    89850
 8 2018-05-05 00:00:00 Central Hermann Luis      Tele~    90       1198   107820
 9 2018-05-22 00:00:00 West    Douglas Michael   Tele~    32       1198    38336
10 2018-06-08 00:00:00 East    Martha  Alexander Home~    60        500    30000
# i 35 more rows
```

The `readxl` package provides various options to read Excel files with multiple sheets, specific ranges, and data types, making it a versatile tool for handling Excel data in R.

### 7.3.2 Writing an Excel file

To write an Excel file in R, you can use the `write_xlsx()` function from the `writexl` package. You can install the `writexl` package using the following command:

```
install.packages("writexl")
```

Once the package is installed, you can load it into your R session using the `library()` function:

```
library(writexl)
```

The `write_xlsx()` function allows you to write a data frame to an Excel file. Here's an example:

```
write_xlsx(df, "data.xlsx")
```

You can check the current working directory to see if the Excel file was created successfully. If you want to specify a different directory or file path, you can provide the full path in the `write_xlsx()` function.

```
# see what the current working directory is
getwd()
```

```
[1] "/home/lorikern/Projects/Papers_Reporting_Conferences/RBiocBook-book/RPC519RBioc"
```

```
# and check to see that the file was created
dir(pattern = "data.xlsx")
```

```
[1] "data.xlsx"
```

## 7.4 Additional options

- Google Sheets: You can read and write data from Google Sheets using the `googlesheets4` package. This package provides functions to interact with Google Sheets, enabling you to import and export data from Google Sheets to R.
- JSON files: You can read and write JSON files using the `jsonlite` package. This package provides functions to convert R objects to JSON format and vice versa, enabling you to work with JSON data in R.
- Database files: You can read and write data from database files using the `DBI` and `RSQLite` packages. These packages provide functions to interact with various database systems, enabling you to import and export data from databases to R.

# Part II

# 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.

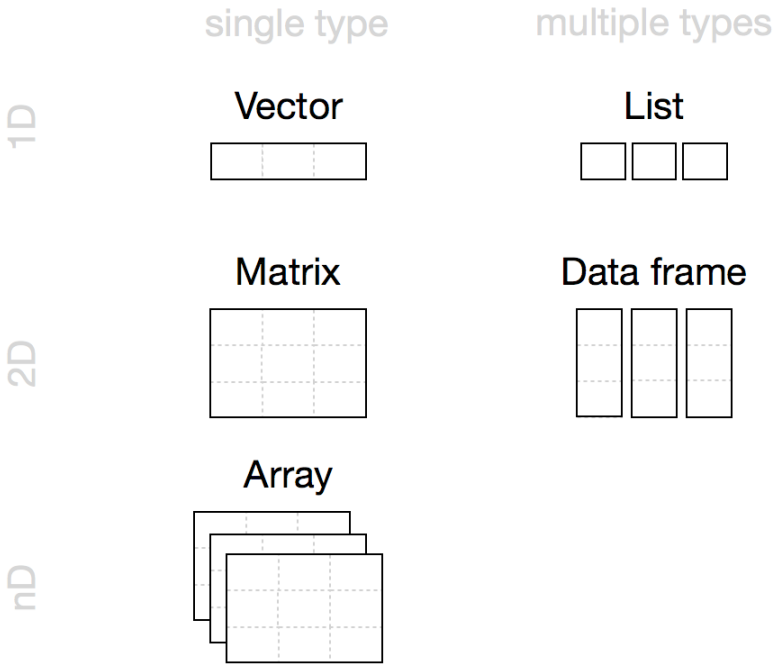



Figure 7.1: A pictorial representation of R's most common data structures are vectors, matrices, arrays, lists, and dataframes. Figure from Hands-on Programming with R.

## 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, 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.

# 8 Vectors

## 8.1 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. Figure 8.1 shows a pictorial representation of three vector examples.

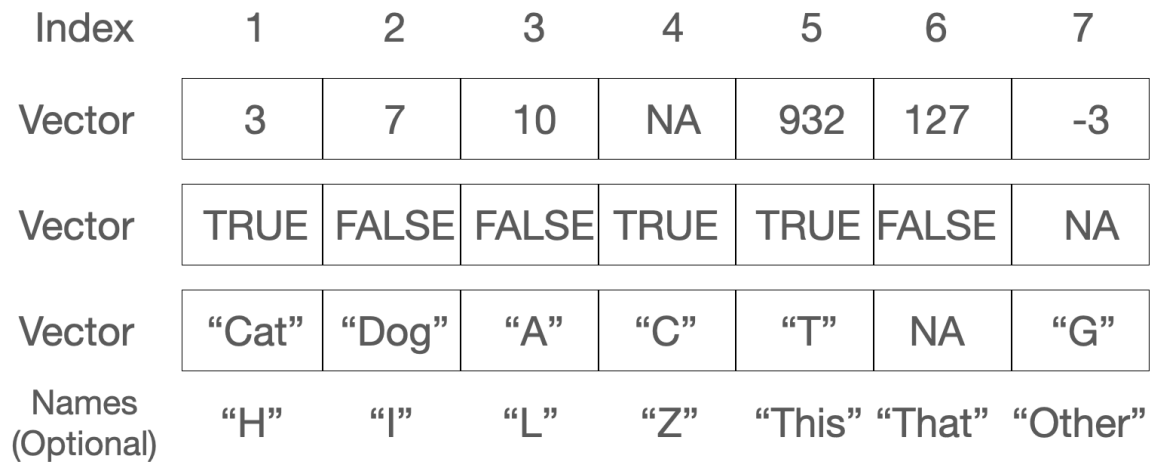



Figure 8.1: “Pictorial representation of three vector examples. The first vector is a numeric vector. The second is a ‘logical’ vector. The third is a character vector. Vectors also have indices and, optionally, names.”

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.

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 Table 8.1. *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.

Table 8.1: Atomic (simplest) data types in R.

| 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 |

## 8.2 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
```

```
# 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
```

## 8.3 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.

```
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".

> ⚠ Warning
>
> 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.

## 8.4 Logical Vectors

Logical vectors are vectors composed on only the values `TRUE` and `FALSE`. Note the all-uppercase 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
```

### 8.4.1 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
```

```
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
```

## 8.5 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".

```
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] 3
```

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

```
[1] 0.8374351 2.1755444 0.4164355
```

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

```
[1] -0.7322452 -0.6779004 -1.3999144  0.1521377 -0.4009818  0.1829415 -0.6192756
```

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

```
[1] 0.8374351 2.1755444 0.4164355
```

## 8.6 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
```

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
```

## 8.7 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.

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

## 8.8 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
```

```
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
```

## 8.9 Exercises

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

   ```
   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]
```

# 9 Matrices

A *matrix* is a rectangular collection of the same data type (see Figure 9.1). 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.

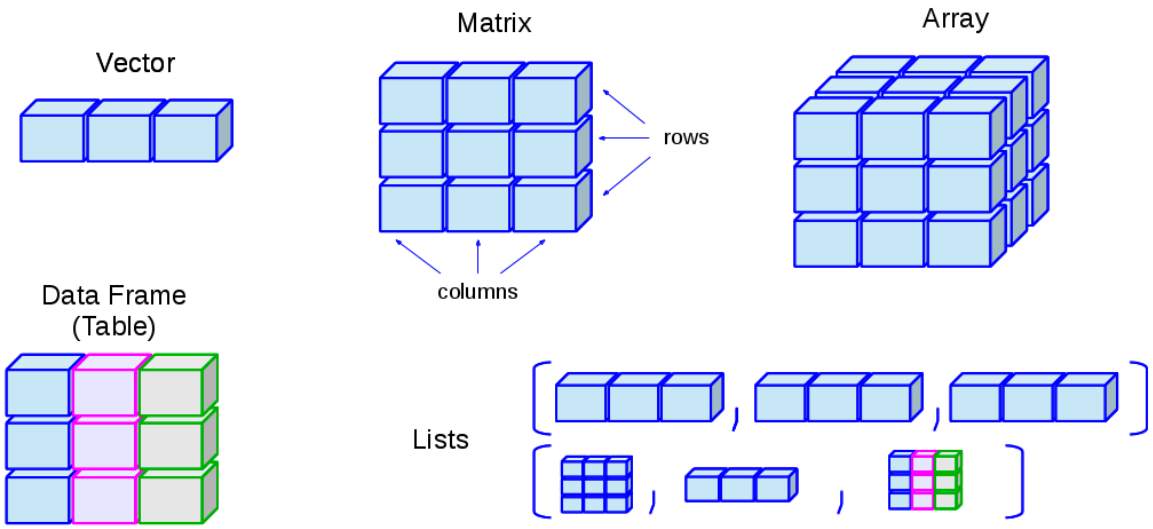



Figure 9.1: A matrix is a collection of column vectors.

## 9.1 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]
[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  1.00000  2.0000000 3.0000000  4.0000000  5.000000  6.0000000 7.000000
y -1.30655 -0.2849439 0.5445569 -0.5404574 -1.269572 -0.2017896 1.681029
      [,8]       [,9]     [,10]
x 8.000000  9.0000000 10.000000
y 1.462597 -0.6817514 -1.632114
```

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

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

```
       x          y
 [1,]  1 -1.3065504
 [2,]  2 -0.2849439
 [3,]  3  0.5445569
 [4,]  4 -0.5404574
 [5,]  5 -1.2695718
 [6,]  6 -0.2017896
 [7,]  7  1.6810294
 [8,]  8  1.4625973
 [9,]  9 -0.6817514
[10,] 10 -1.6321137
```

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"
```

```
mat
```

```
      apples    oranges
 [1,]      1 -1.3065504
 [2,]      2 -0.2849439
 [3,]      3  0.5445569
 [4,]      4 -0.5404574
 [5,]      5 -1.2695718
 [6,]      6 -0.2017896
 [7,]      7  1.6810294
 [8,]      8  1.4625973
 [9,]      9 -0.6817514
[10,]     10 -1.6321137
```

Matrices have dimensions.

```
dim(mat)
```

```
[1] 10  2
```

```
nrow(mat)
```

```
[1] 10
```

```
ncol(mat)
```

```
[1] 2
```

## 9.2 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.

> **!** 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
-1.30655
```

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

```
  apples  oranges
 1.00000 -1.30655
```

```
# 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
```

```
## [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] -1.3065504 -0.2849439  0.5445569 -0.5404574 -1.2695718 -0.2017896
 [7]  1.6810294  1.4625973 -0.6817514 -1.6321137
```

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

```
     apples    oranges
[1,]      6 -0.2017896
[2,]      7  1.6810294
[3,]      8  1.4625973
[4,]      9 -0.6817514
[5,]     10 -1.6321137
```

## 9.3 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.   :-1.7287   Min.   :-2.1252  
 1st Qu.:-1.0217   1st Qu.:-0.2659  
 Median :-0.7183   Median : 0.4366  
 Mean   :-0.3965   Mean   : 0.3845  
 3rd Qu.: 0.4372   3rd Qu.: 1.2448  
 Max.   : 1.2721   Max.   : 2.7633  
```

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.   :-34.573   Min.   :-42.504  
 1st Qu.:-20.433   1st Qu.: -5.319  
 Median :-14.367   Median :  8.731  
 Mean   : -7.930   Mean   :  7.691  
 3rd Qu.:  8.744   3rd Qu.: 24.895  
 Max.   : 25.443   Max.   : 55.265  
```

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.   : 65.43   Min.   :-42.504  
 1st Qu.: 79.57   1st Qu.: -5.319  
 Median : 85.63   Median :  8.731  
 Mean   : 92.07   Mean   :  7.691  
 3rd Qu.:108.74   3rd Qu.: 24.895  
 Max.   :125.44   Max.   : 55.265  
```

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,]  82.60753 84.09010  65.42690 87.17645 114.365134 74.39922 116.755042
[2,] -19.99175 55.26513 -42.50417 30.37719  -0.662172 15.96026   2.494758
           [,8]     [,9]    [,10]
[1,] 125.442777 78.55289 91.87916
[2,]  -6.871007 14.96736 27.87335
```

## 9.4 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] 5.25246
```

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

```
[1] 2.351132
```

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

```
[1] 4.073799
```

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

```
[1] 1.762386
```

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] 5.252460 5.253362 5.743336 4.150106 3.995761 2.906782 4.905868 4.617782
 [9] 3.670924 5.157113
```

```
rowMeans(m3)
```

```
 [1] 4.073799 4.559017 4.321635 4.559505 4.033766 4.850547 4.292942 5.116671
 [9] 5.428672 4.416941
```

```
rowSums(m3)
```

```
 [1] 40.73799 45.59017 43.21635 45.59505 40.33766 48.50547 42.92942 51.16671
 [9] 54.28672 44.16941
```

```
colSums(m3)
```

```
 [1] 52.52460 52.53362 57.43336 41.50106 39.95761 29.06782 49.05868 46.17782
 [9] 36.70924 51.57113
```

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.
  2.907   4.034   4.762   4.565   5.229   5.743
```

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.
  1.421   1.721   1.977   1.995   2.291   2.612
```

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

## 9.5 Exercises

### 9.5.1 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)
```

### 9.5.2 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.

```
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.

```
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')
```

# 10 Lists

## 10.1 The Power of a "Catch-All" Container

So far in our journey through R's data structures, we've dealt with vectors and matrices. These are fantastic tools, but they have one strict rule: all their elements must be of the *same data type*. You can have a vector of numbers or a matrix of characters, but you can't mix and match.

But what about real-world biological data? A single experiment can generate a dizzying variety of information. Imagine you're studying a particular gene. You might have:

- The gene's name (text).
- Its expression level across several samples (a set of numbers).
- A record of whether it's a known cancer-related gene (a simple TRUE/FALSE).
- The raw fluorescence values from your qPCR machine (a matrix of numbers).
- Some personal notes about the experiment (a paragraph of text).

How could you possibly store all of this related, yet different, information together? You could create many separate variables, but that would be clunky and hard to manage. This is exactly the problem that **lists** are designed to solve.

A list in R is like a flexible, multi-compartment container. It's a single object that can hold a collection of *other* R objects, and those objects can be of any type, length, or dimension. You can put vectors, matrices, logical values, and even other lists inside a single list. This makes them one of the most fundamental and powerful data structures for bioinformatics analysis.

The key features of lists are:

- **Flexibility**: They can contain a mix of any data type.
- **Organization**: You can and should *name* the elements of a list, making your data self-describing.
- **Hierarchy**: Because lists can contain other lists, you can create complex, nested data structures to represent sophisticated relationships in your data.

## 10.2 Creating a List

You create a list with the `list()` function. The best practice is to name the elements as you create them. This makes your code infinitely more readable and your data easier to work with.

Let's create a list to store the information for our hypothetical gene study.

```
# An experiment tracking list for the gene TP53
experiment_data <- list(
  experiment_id = "EXP042",
  gene_name = "TP53",
  read_counts = c(120, 155, 98, 210),
  is_control = FALSE,
  sample_matrix = matrix(1:4, nrow = 2, dimnames = list(c("Treated", "Untreated"), c("Replica
)

# --- Function Explainer: print() ---
# The print() function displays the contents of an R object in the console.
# For a list, it shows each element and its contents. It's the default action
# when you just type the variable's name and hit Enter.
print(experiment_data)
```

```
$experiment_id
[1] "EXP042"

$gene_name
[1] "TP53"

$read_counts
[1] 120 155  98 210

$is_control
[1] FALSE

$sample_matrix
          Replicate1 Replicate2
Treated            1          3
Untreated          2          4
```

## 10.3 Inspecting Your List: What's Inside?

When someone hands you a tube in the lab, the first thing you do is look at the label. When R gives you a complex object like a list, you need to do the same. R provides several "introspection" functions to help you understand the contents and structure of your lists.

### 10.3.1 `str()`: The Structure Function

This is arguably the most useful function for inspecting any R object, especially lists.

```
# --- Function Explainer: str() ---
# The str() function provides a compact, human-readable summary of an
# object's internal "str"ucture. It's your best friend for understanding
# what's inside a list, including the type and a preview of each element.
str(experiment_data)
```

```
List of 5
 $ experiment_id: chr "EXP042"
 $ gene_name    : chr "TP53"
 $ read_counts  : num [1:4] 120 155 98 210
 $ is_control   : logi FALSE
 $ sample_matrix: int [1:2, 1:2] 1 2 3 4
  ..- attr(*, "dimnames")=List of 2
  .. ..$ : chr [1:2] "Treated" "Untreated"
  .. ..$ : chr [1:2] "Replicate1" "Replicate2"
```

The output of `str()` tells us everything we need to know: it's a "List of 5", and for each of the 5 elements, it shows the name (e.g., `experiment_id`), the data type (e.g., `chr` for character, `num` for numeric), and a preview of the content.

### 10.3.2 `length()`, `names()`, and `class()`

These functions give you more specific information about the list itself.

```
# --- Function Explainer: length() ---
# For a list, length() tells you how many top-level elements it contains.
length(experiment_data)
```

```
[1] 5
```

```
# --- Function Explainer: names() ---
# The names() function extracts the names of the elements in a list as a
# character vector. It's a great way to see what you can access.
names(experiment_data)
```

```
[1] "experiment_id" "gene_name"     "read_counts"   "is_control"
[5] "sample_matrix"
```

```
# --- Function Explainer: class() ---
# The class() function tells you the type of the object itself.
# This is useful to confirm you are indeed working with a list.
class(experiment_data)
```

```
[1] "list"
```

## 10.4 Accessing List Elements: Getting Things Out

Okay, you've packed your experimental data into a list. Now, how do you get specific items out? This is a critical concept, and R has a few ways to do it, each with a distinct purpose.

### 10.4.1 The Mighty `[[...]]` and `$` for Single Items

To pull out a *single element* from a list in its original form, you use either double square brackets `[[...]]` or the dollar sign `$` (for named lists). Think of this as carefully reaching into a specific compartment of your container and taking out the item itself.

Let's use our `experiment_data` list.

```
# Get the gene name using [[...]]
gene <- experiment_data[["gene_name"]]
print(gene)
```

```
[1] "TP53"
```

```
class(gene) # It's a character vector, just as it was when we put it in.
```

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

```
# Get the read counts using the $ shortcut. This is often easier to read.
reads <- experiment_data$read_counts
print(reads)
```

```
[1] 120 155  98 210
```

```
class(reads) # It's a numeric vector.
```

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

```
# The [[...]] has a neat trick: you can use a variable to specify the name.
element_to_get <- "read_counts"
experiment_data[[element_to_get]]
```

```
[1] 120 155  98 210
```

The key takeaway is that `[[...]]` and `$` **extract the element**. The result is the object that was stored inside the list.

### 10.4.2 The Subsetting `[...]` for New Lists

The single square bracket `[...]` behaves differently. It always returns a *new, smaller list* that is a subset of the original list. It's like taking a whole compartment, label and all, out of your larger container.

```
# Get the gene name using [...]
gene_sublist <- experiment_data["gene_name"]

print(gene_sublist)
```

```
$gene_name
[1] "TP53"
```

```
# --- Note the class! ---
# The result is another list, which contains the gene_name element.
class(gene_sublist)
```

```
[1] "list"
```

This distinction is vital. If you want to perform a calculation on an element (like finding the `mean()` of `read_counts`), you must extract it with `[[...]]` or `$`. If you tried `mean(experiment_data["read_counts"])`, R would give you an error because you can't calculate the mean of a list!

## 10.5 Modifying Lists

Your data is rarely static. You can easily add, remove, or update elements in a list after you've created it.

### 10.5.1 Adding and Updating Elements

You can add a new element or change an existing one by using the `$` or `[[...]]` assignment syntax.

```
# Add the date of the experiment
experiment_data$date <- "2024-06-05"

# Add some notes using the [[...]] syntax
experiment_data[["notes"]] <- "Initial pilot experiment. High variance in read counts."

# Let's update the control status
experiment_data$is_control <- TRUE

# Let's look at the structure now
str(experiment_data)
```

```
List of 7
 $ experiment_id: chr "EXP042"
 $ gene_name    : chr "TP53"
 $ read_counts  : num [1:4] 120 155 98 210
 $ is_control   : logi TRUE
 $ sample_matrix: int [1:2, 1:2] 1 2 3 4
  ..- attr(*, "dimnames")=List of 2
  .. ..$ : chr [1:2] "Treated" "Untreated"
  .. ..$ : chr [1:2] "Replicate1" "Replicate2"
 $ date         : chr "2024-06-05"
 $ notes        : chr "Initial pilot experiment. High variance in read counts."
```

### 10.5.2 Removing Elements

To remove an element from a list, you simply assign `NULL` to it. `NULL` is R's special object representing nothingness.

```
# We've decided the matrix isn't needed for this summary object.
experiment_data$sample_matrix <- NULL

# See the final structure of our list
str(experiment_data)
```

```
List of 6
 $ experiment_id: chr "EXP042"
 $ gene_name    : chr "TP53"
 $ read_counts  : num [1:4] 120 155 98 210
 $ is_control   : logi TRUE
 $ date         : chr "2024-06-05"
 $ notes        : chr "Initial pilot experiment. High variance in read counts."
```

## 10.6 A Biological Example: A Self-Contained Gene Record

Let's put this all together. Lists are perfect for creating self-contained records that you can easily pass to functions or combine into larger lists.

```
# --- Function Explainer: log2() ---
# The log2() function calculates the base-2 logarithm. It's very common in
# gene expression analysis to transform skewed count data to make it more
# symmetric and easier to model.

brca1_gene <- list(
  gene_symbol = "BRCA1",
  full_name = "BRCA1 DNA repair associated",
  chromosome = "17",
  expression_log2 = log2(c(45, 50, 30, 88, 120)),
  related_diseases = c("Breast Cancer", "Ovarian Cancer")
)

# Now we can easily work with this structured information

# --- Function Explainer: cat() ---
```

```r
# The cat() function concatenates and prints its arguments to the console.
# Unlike print(), it allows you to seamlessly join text and variables, and
# the "\n" character is used to add a newline (a line break).
cat("Analyzing gene:", brca1_gene$gene_symbol, "\n")
```

```
Analyzing gene: BRCA1
```

```r
cat("Located on chromosome:", brca1_gene$chromosome, "\n")
```

```
Located on chromosome: 17
```

```r
# Calculate the average log2 expression
# --- Function Explainer: mean() ---
# The mean() function calculates the arithmetic average of a numeric vector.
avg_expression <- mean(brca1_gene$expression_log2)
cat("Average log2 expression:", avg_expression, "\n")
```

```
Average log2 expression: 5.881784
```

This simple `brca1_gene` list is now a complete, portable record. You could imagine creating a list of these gene records, creating a powerful, hierarchical database for your entire project.

# 11 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.

## 11.1 Learning goals

- Understand how data.frames are different from matrices.
- Know a few functions for examing 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.

## 11.2 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.

## 11.3 Dataset

The data used here are borrowed directly from the fantastic Bioconnector 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.

## 11.4 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"
```

## 11.5 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
- 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 polyadenylylation*
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
```

```
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
```

```
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)
```

## 11.6 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:

```
# 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"
```

### 11.6.1 Some data exploration

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

```
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 a the `rate` column given that that column appears to have repeated values?

What `rate` corresponds to the most nutrient-starved condition?

### 11.6.2 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>
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
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
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
                                           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
```

## 11.7 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
```

```
# 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
```

## 11.8 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
```

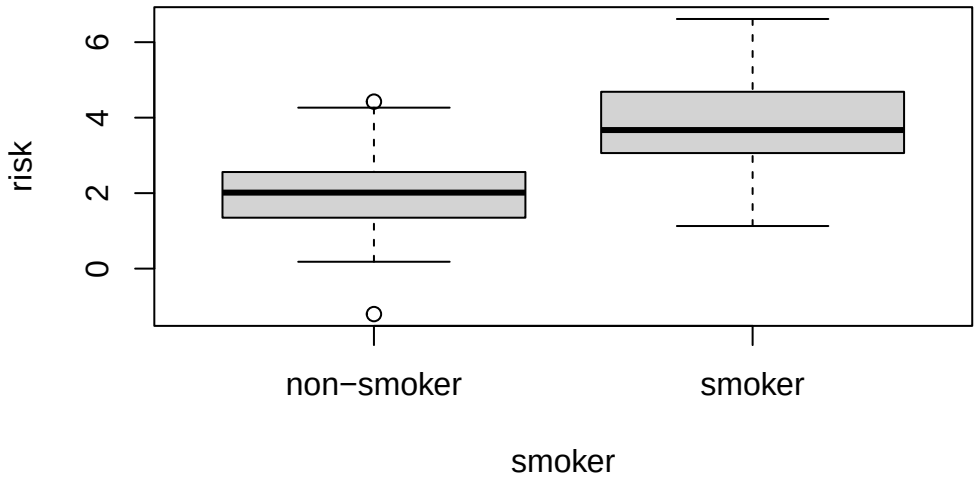
We have two varibles, `risk` and `smoker` that are related. We can make a data.frame out of them:

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

```
  smoker     risk
1 smoker 3.460632
2 smoker 2.811295
3 smoker 4.010997
4 smoker 4.243112
5 smoker 2.746992
6 smoker 2.990194
```

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

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



## 11.9 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")
```

# 12 Factors

## 12.1 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)
citizen
```

```
[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
```

```
# 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.

```
table(citizenf)
```

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

The default factor levels are the unique set of possible values. It is possible to specify a subset of factor levels. Note how missing values are introduced if a value is not included.

```r
citizenf2 <- factor(citizen, levels=c("us", "uk"))
citizenf2
```

```
[1] uk   us   <NA> <NA> uk   us   us   <NA> <NA>
Levels: us uk
```

```r
table(citizenf2)
```

```
citizenf2
us uk
 3  2
```

Missing values are exlcuded by default. There is an option to override this setting.

```r
addNA(citizenf2)
```

```
[1] uk   us   <NA> <NA> uk   us   us   <NA> <NA>
Levels: us uk <NA>
```

```r
table(addNA(citizenf2))
```

```
  us   uk <NA>
   3    2    4
```

> **Caution**
>
> This emphasizes that default settings may or may not be appropriate for your analysis. It's important to know what those settings are and choose alternatives as necessary.

# 13 Classes

All the data structures discussed are also known as an object's class. While R has many predefined classes, it is flexible to create new classes. Many packages define their own classes of objects. This can be beneficial for organizational purposes as well as to build custom functions for specialized analysis.

This is a high level concept to be aware of. Bioconductor for example has many specialized classes related to genomic research. An overview of some of those classes can be found at Introduction to Bioconductor Classes.

If time permits we'll come back to showing how custom and more complex classes, while intimidating to learn at first, can be really beneficial and help prevent careless errors if designed correctly.

# 14 Control Statements

Control statements help determine the flow and execution of commands based on conditional statements. This chapter will cover a brief overview of the following:

- Conditional Statements
  - if
  - if-else
  - ifelse
- Loops
  - for
  - while
  - repeat-break
- Special
  - break
  - return
  - next
- Other
  - nested
  - try / tryCatch
  - vectorization and apply functions

> **Important**
>
> Pay attention to syntax. It is important to include all parenthesis and brackets. In general if you have an open parenthesis or bracket, you will need a closed parenthesis or bracket.

> **Tip**
>
> Good coding practices involve consistent indenting and spacing. Once a control statement is initialized, all code in its brackets are indented to show clearly what code/statements are being executed for that section. This becomes especially important if you start nesting

control statements.

## 14.1 Conditional Statements

### 14.1.1 if

Let's start with an `if` statement. An if statement evaluates an expression and depending on its result performs a sub-section of select code.

**Syntax:**

```
if (expression){
  # additional code to run if expression is TRUE
  ...
}
```

The expression contained in parenthesis will result in a boolean (TRUE/FALSE) value used to determine if the code in the braces should be executed.

**Example:**

```
x <- 12
if (x > 0){
  message(x, " is greater than 0")
  x <- 0
}
```

```
12 is greater than 0
```

```
x
```

```
[1] 0
```

> **Tip**
>
> Notice all lines in the braces are executed including a assignment that changes our original value

### 14.1.2 if-else

An `if-else` statement adds additional code to be executed if the expression is FALSE.

**Syntax:**

```
if (expression){
  # additional code to run if expression is TRUE
  ...
} else {
  # additional code to run if expression if FALSE
  ...
}
```

We could read this allowed as if the expression is true execute this code else if the expression is false execute this other code.

**Example:**

```
if (x > 0){
  message(x, " is greater than 0")
  x <- 0
} else {
  message(x, " is not greater than 0")
  x <- x + 2
}
```

```
0 is not greater than 0
```

What is x now?

### 14.1.3 ifelse

A specialized if-else statement is the `ifelse`. It is a simplified version where an object can be coerced into logical form and return values for true/false.

**Syntax:**

```
ifelse(test_expression, yes_value, no_value)
```

**Example**

```
num_vec <- -3:3
ifelse(num_vec >= 0, "positive", "negative")
```

```
[1] "negative" "negative" "negative" "positive" "positive" "positive" "positive"
```

## 14.2 Loops

Loops are control statements that allow for repeated code execution either for a set number of times, over a certain set of elements, or until a conditional statement is met.

### 14.2.1 for

A `for` loop will execute commands over a certain set of elements.

**Syntax:**

```
for(value in vector){
  ## code to execute for each item in vector
  ...
}
```

value can be utilized in the executed code.

**Examples**

In this example, the vector of names is looped over, printing the number of characters in each name.

```
names <- c("Donna", "John", "Bradley", "Kara")
for(nm in names){
  print(paste(nm, "has", nchar(nm), "letters"))
}
```

```
[1] "Donna has 5 letters"
[1] "John has 4 letters"
[1] "Bradley has 7 letters"
[1] "Kara has 4 letters"
```

In this example, for each value 1 to 5 (1,2,3,4,5), we take that value and add to the current value of x. Notice how this updates x each time.

```
x <- 0
for(i in 1:5){
  print(paste("add", i, "to", x))
  x <- x + i
}
```

```
[1] "add 1 to 0"
[1] "add 2 to 1"
[1] "add 3 to 3"
[1] "add 4 to 6"
[1] "add 5 to 10"
```

In this example, we loop over the elements of a list. For each list element we get the name of the item in the list and how many items that list element contains.

```
my_list <- list(people=names,
                ages=c(54, 78, 40, 5, 25),
        animals=c("dog", "fish"))
for(i in seq_along(my_list)){
  list_element <- my_list[[i]]
  print(paste("List element", names(my_list)[i], "contains",
  length(my_list[[i]]), "values"))
}
```

```
[1] "List element people contains 4 values"
[1] "List element ages contains 5 values"
[1] "List element animals contains 2 values"
```

### 14.2.2 while

A `while` loop will execute until an expression is met.

**Syntax:**

```
while(expression){
  ## code to execute until the expression is met
  ## be sure to update variable used in expression
  ...
}
```

In this example we will start at the value 1 and as long as that value stays less than or equal to 5, we will print the value, incrementing by 1 each loop:

**Example**

```
value <- 1
while (value <= 5){
  print(value)
  value = value + 1
}
```

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

> **!** Important
>
> Notice how we have to update the value that is being checked each time and that it is logical that it should eventually reach a point where the loop exists. Be cautious of **infinite loops**. These occur when the loop will never reach a stopping point because the expression will never be FALSE

### 14.2.3 repeat

`repeat` is a indefinite loop. A `break` statement must be used to terminate the loop.

**Syntax:**

```
repeat{
  ## code to be evaluated
  if (condition){
    break
  }
}
```

In this example we will repeat ourself until the breaking conditions reaches the number of times we set to repeat. Notice how the code inside the loop alters the variable used in the conditional statement.

**Example**

```r
i <- 0
times <- 3
repeat{
  print("I am repeating myself")
  i <- i + 1
  if (i == times){
    break
  }
}
```

```
[1] "I am repeating myself"
[1] "I am repeating myself"
[1] "I am repeating myself"
```

## 14.3 Special

There are some special other options used to customize control statements. These are used within control statements and can be especially useful in nested statements.

**break**

We have already seen `break` usage with repeat. Break will stop and exit the control statement immediately when it hits.

**return**

`return` is similar to break in that it will stop and ext the control statement immediately when it is hit, however it will return the result of the given executed function or variable value upon exiting. It is generally used within functions.

**Syntax:**

```r
return(expression)
```

This example creates a function that takes argument x. If x is equal to 0 it returns the string "zero". If it does not equal 0 it continues to execute code. It will add 4 to the value. If that value is less than 0 it returns the value, otherwise it returns the value multipled by 2.

**Example**

```
func <- function(x){
  if(x == 0){
    return("zero")
  }
  x <- x+4
  if (x <= 0){
    return(x)
  }else{
    return(x*2)
  }
}

func(0)
```

```
[1] "zero"
```

```
func(-8)
```

```
[1] -4
```

```
func(6)
```

```
[1] 20
```

**next**

`next` will skip the current iteration of a loop without executing any further statements without terminating the loop.

**Example**

```
for(i in 1:10){
  if(i%%2 != 0){
    next
  }
  print(i)
}
```

```
[1] 2
[1] 4
[1] 6
[1] 8
[1] 10
```

## 14.4 Other

### 14.4.1 nesting

We already saw an example of a nesting built into the design of repeat-break; nesting an if statement inside the repeat. All control statements can have multiple nesting. Nesting multiple for loops to loop over the row and columns of a matrix is such an example.

**Example:**

In this example we create a numeric matrix with 5 rows and 3 columns and fill the numbers 1 through 15 by column. Let us loop over by row and print out each cell of the matrix.

```
mat <- matrix(1:15, ncol=3)
mat
```

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

```
for (r in seq(nrow(mat))) {
  for (c in seq(ncol(mat))) {
    print(mat[r, c])
  }
}
```

```
[1] 1
[1] 6
[1] 11
[1] 2
[1] 7
[1] 12
[1] 3
[1] 8
[1] 13
[1] 4
[1] 9
[1] 14
[1] 5
```

```
[1] 10
[1] 15
```

### 14.4.2 try / tryCatch

This isn't necessarily a control statement but fits well enough to disucss. A `tryCatch` statement is a way to handle code that may produce errors, warnings, or other conditions that may arise.

**Syntax:**

```
tryCatch(expr,
         error = function(e) {
         ## error handling code
     },
         warning = function(w) {
         ## warning handling code
     },
         finally = {
        ## code to execute regardless of conditions
     }
)
```

It can include all error, warning, finally or any subset of the three. expr is an expression or block of code that is attempted to be executed. The error if a function that is triggered if running the expr resulted in an error. The argument e is generally an error object that contains details about the error for reference or parsing if necessary. warning similar to error but handles warnings with argument w. finally is code that executed regardless of what happens. This is particularly useful for when the expr opens a connection such as a database or file; finally can be used to close or clean up these open connections.

### 14.4.3 vectorization and apply functions

Some most common and simply implementations of control functions are to perform functions over a vector or applying a function over the columns or rows of a matrix. While control statement can be utilized, they may not be the most efficient way to accomplish these tasks. It is encouraged to invesetigate if an already existing vectorized function exists for common tasks (mean, average, etc are already vectorized) or to utilized apply functions. Apply functions include: `apply()`, `lapply()`, `sapply()`, `mapply()`, and `tapply()`.

# Part III

# Exploratory Data Analysis

# 15 Introduction to dplyr: mammal sleep dataset

The dataset we will be using to introduce the *dplyr* package is an updated and expanded version of the mammals sleep dataset. Updated sleep times and weights were taken from V. M. Savage and G. B. West. A quantitative, theoretical framework for understanding mammalian sleep[1].

## 15.1 Learning goals

- Know that `dplyr` is just a different approach to manipulating data in data.frames.
- List the commonly used `dplyr` verbs and how they can be used to manipulate data.frames.
- Show how to aggregate and summarized data using `dplyr`
- Know what the piping operator, `|>`, is and how it can be used.

## 15.2 Learning objectives

- Select subsets of the mammal sleep dataset.
- Reorder the dataset.
- Add columns to the dataset based on existing columns.
- Summarize the amount of sleep by categorical variables using `group_by` and `summarize`.

## 15.3 What is dplyr?

The *dplyr* package is a specialized package for working with `data.frames` (and the related `tibble`) to transform and summarize tabular data with rows and columns. For another explanation of dplyr see the dplyr package vignette: Introduction to dplyr

[1] A quantitative, theoretical framework for understanding mammalian sleep. Van M. Savage, Geoffrey B. West. Proceedings of the National Academy of Sciences Jan 2007, 104 (3) 1051-1056; DOI: 10.1073/pnas.0610080104

## 15.4 Why Is dplyr userful?

dplyr contains a set of functions–commonly called the dplyr "verbs"–that perform common data manipulations such as filtering for rows, selecting specific columns, re-ordering rows, adding new columns and summarizing data. In addition, dplyr contains a useful function to perform another common task which is the "split-apply-combine" concept.

Compared to base functions in R, the functions in dplyr are often easier to work with, are more consistent in the syntax and are targeted for data analysis around data frames, instead of just vectors.

## 15.5 Data: Mammals Sleep

The msleep (mammals sleep) data set contains the sleep times and weights for a set of mammals and is available in the dagdata repository on github. This data set contains 83 rows and 11 variables. The data happen to be available as a `data`set in the *ggplot2* package. To get access to the `msleep` dataset, we need to first install the ggplot2 package.

```
install.packages('ggplot2')
```

Then, we can load the library.

```
library(ggplot2)
data(msleep)
```

As with many datasets in R, "help" is available to describe the dataset itself.

```
?msleep
```

The columns are described in the help page, but are included here, also.

| column name | Description |
|---|---|
| name | common name |
| genus | taxonomic rank |
| vore | carnivore, omnivore or herbivore? |
| order | taxonomic rank |
| conservation | the conservation status of the mammal |
| sleep_total | total amount of sleep, in hours |
| sleep_rem | rem sleep, in hours |

| column name | Description |
|---|---|
| sleep_cycle | length of sleep cycle, in hours |
| awake | amount of time spent awake, in hours |
| brainwt | brain weight in kilograms |
| bodywt | body weight in kilograms |

## 15.6 dplyr verbs

The dplyr verbs are listed here. There are many other functions available in dplyr, but we will focus on just these.

| dplyr verbs | Description |
|---|---|
| `select()` | select columns |
| `filter()` | filter rows |
| `arrange()` | re-order or arrange rows |
| `mutate()` | create new columns |
| `summarise()` | summarise values |
| `group_by()` | allows for group operations in the "split-apply-combine" concept |

## 15.7 Using the dplyr verbs

The two most basic functions are `select()` and `filter()`, which selects columns and filters rows respectively. What are the equivalent ways to select columns without dplyr? And filtering to include only specific rows?

Before proceeding, we need to install the dplyr package:

```
install.packages('dplyr')
```

And then load the library:

```
library(dplyr)
```

```
Attaching package: 'dplyr'
```

```
The following objects are masked from 'package:stats':

    filter, lag

The following objects are masked from 'package:base':

    intersect, setdiff, setequal, union
```

### 15.7.1 Selecting columns: `select()`

Select a set of columns such as the `name` and the `sleep_total` columns.

```
sleepData <- select(msleep, name, sleep_total)
head(sleepData)
```

```
# A tibble: 6 x 2
  name                       sleep_total
  <chr>                            <dbl>
1 Cheetah                           12.1
2 Owl monkey                        17
3 Mountain beaver                   14.4
4 Greater short-tailed shrew        14.9
5 Cow                                4
6 Three-toed sloth                  14.4
```

To select all the columns *except* a specific column, use the "-" (subtraction) operator (also known as negative indexing). For example, to select all columns except `name`:

```
head(select(msleep, -name))
```

```
# A tibble: 6 x 10
  genus      vore  order    conservation sleep_total sleep_rem sleep_cycle awake
  <chr>      <chr> <chr>    <chr>              <dbl>     <dbl>       <dbl> <dbl>
1 Acinonyx   carni Carnivo~ lc                  12.1      NA        NA      11.9
2 Aotus      omni  Primates <NA>                17         1.8      NA       7
3 Aplodontia herbi Rodentia nt                  14.4       2.4      NA       9.6
4 Blarina    omni  Soricom~ lc                  14.9       2.3       0.133   9.1
5 Bos        herbi Artioda~ domesticated         4         0.7       0.667  20
6 Bradypus   herbi Pilosa   <NA>                14.4       2.2       0.767   9.6
# i 2 more variables: brainwt <dbl>, bodywt <dbl>
```

To select a range of columns by name, use the ":" operator. Note that dplyr allows us to use the column names without quotes and as "indices" of the columns.

```
head(select(msleep, name:order))
```

```
# A tibble: 6 x 4
  name                        genus      vore  order
  <chr>                       <chr>      <chr> <chr>
1 Cheetah                     Acinonyx   carni Carnivora
2 Owl monkey                  Aotus      omni  Primates
3 Mountain beaver             Aplodontia herbi Rodentia
4 Greater short-tailed shrew Blarina    omni  Soricomorpha
5 Cow                         Bos        herbi Artiodactyla
6 Three-toed sloth            Bradypus   herbi Pilosa
```

To select all columns that start with the character string "sl", use the function `starts_with()`.

```
head(select(msleep, starts_with("sl")))
```

```
# A tibble: 6 x 3
  sleep_total sleep_rem sleep_cycle
        <dbl>     <dbl>       <dbl>
1        12.1      NA         NA
2        17         1.8       NA
3        14.4       2.4       NA
4        14.9       2.3        0.133
5         4         0.7        0.667
6        14.4       2.2        0.767
```

Some additional options to select columns based on a specific criteria include:

1. `ends_with()` = Select columns that end with a character string
2. `contains()` = Select columns that contain a character string
3. `matches()` = Select columns that match a regular expression
4. `one_of()` = Select column names that are from a group of names

### 15.7.2 Selecting rows: `filter()`

The `filter()` function allows us to filter rows to include only those rows that *match* the filter. For example, we can filter the rows for mammals that sleep a total of more than 16 hours.

```
filter(msleep, sleep_total >= 16)
```

```
# A tibble: 8 x 11
  name    genus vore  order conservation sleep_total sleep_rem sleep_cycle awake
  <chr>   <chr> <chr> <chr> <chr>              <dbl>     <dbl>       <dbl> <dbl>
1 Owl mo~ Aotus omni  Prim~ <NA>                17         1.8      NA       7
2 Long-n~ Dasy~ carni Cing~ lc                  17.4       3.1       0.383   6.6
3 North ~ Dide~ omni  Dide~ lc                  18         4.9       0.333   6
4 Big br~ Epte~ inse~ Chir~ lc                  19.7       3.9       0.117   4.3
5 Thick-~ Lutr~ carni Dide~ lc                  19.4       6.6      NA       4.6
6 Little~ Myot~ inse~ Chir~ <NA>                19.9       2         0.2     4.1
7 Giant ~ Prio~ inse~ Cing~ en                  18.1       6.1      NA       5.9
8 Arctic~ Sper~ herbi Rode~ lc                  16.6      NA        NA       7.4
# i 2 more variables: brainwt <dbl>, bodywt <dbl>
```

Filter the rows for mammals that sleep a total of more than 16 hours *and* have a body weight of greater than 1 kilogram.

```
filter(msleep, sleep_total >= 16, bodywt >= 1)
```

```
# A tibble: 3 x 11
  name    genus vore  order conservation sleep_total sleep_rem sleep_cycle awake
  <chr>   <chr> <chr> <chr> <chr>              <dbl>     <dbl>       <dbl> <dbl>
1 Long-n~ Dasy~ carni Cing~ lc                  17.4       3.1       0.383   6.6
2 North ~ Dide~ omni  Dide~ lc                  18         4.9       0.333   6
3 Giant ~ Prio~ inse~ Cing~ en                  18.1       6.1      NA       5.9
# i 2 more variables: brainwt <dbl>, bodywt <dbl>
```

Filter the rows for mammals in the Perissodactyla and Primates taxonomic order. The `%in%` operator is a logical operator that returns `TRUE` for values of a vector that are present *in* a second vector.

```
filter(msleep, order %in% c("Perissodactyla", "Primates"))
```

```
# A tibble: 15 x 11
   name   genus vore  order conservation sleep_total sleep_rem sleep_cycle awake
   <chr>  <chr> <chr> <chr> <chr>               <dbl>     <dbl>       <dbl> <dbl>
 1 Owl m~ Aotus omni  Prim~ <NA>                 17         1.8      NA       7
 2 Grivet Cerc~ omni  Prim~ lc                   10         0.7      NA      14
 3 Horse  Equus herbi Peri~ domesticated          2.9       0.6       1      21.1
 4 Donkey Equus herbi Peri~ domesticated          3.1       0.4      NA      20.9
 5 Patas~ Eryt~ omni  Prim~ lc                   10.9       1.1      NA      13.1
 6 Galago Gala~ omni  Prim~ <NA>                  9.8       1.1       0.55   14.2
 7 Human  Homo  omni  Prim~ <NA>                  8         1.9       1.5    16
 8 Mongo~ Lemur herbi Prim~ vu                    9.5       0.9      NA      14.5
 9 Macaq~ Maca~ omni  Prim~ <NA>                 10.1       1.2       0.75   13.9
10 Slow ~ Nyct~ carni Prim~ <NA>                 11        NA        NA      13
11 Chimp~ Pan   omni  Prim~ <NA>                  9.7       1.4       1.42   14.3
12 Baboon Papio omni  Prim~ <NA>                  9.4       1         0.667  14.6
13 Potto  Pero~ omni  Prim~ lc                   11        NA        NA      13
14 Squir~ Saim~ omni  Prim~ <NA>                  9.6       1.4      NA      14.4
15 Brazi~ Tapi~ herbi Peri~ vu                    4.4       1         0.9    19.6
# i 2 more variables: brainwt <dbl>, bodywt <dbl>
```

You can use the boolean operators (e.g. >, <, >=, <=, !=, %in%) to create the logical tests.

## 15.8 "Piping" " with |>

It is not unusual to want to perform a set of operations using dplyr. The pipe operator |> allows us to "pipe" the output from one function into the input of the next. While there is nothing special about how R treats operations that are written in a pipe, the idea of piping is to allow us to read multiple functions operating one after another from left-to-right. Without piping, one would either 1) save each step in set of functions as a temporary variable and then pass that variable along the chain or 2) have to "nest" functions, which can be hard to read.

Here's an example we have already used:

```
head(select(msleep, name, sleep_total))
```

```
# A tibble: 6 x 2
  name                        sleep_total
  <chr>                             <dbl>
1 Cheetah                            12.1
2 Owl monkey                         17
3 Mountain beaver                    14.4
```

```
4 Greater short-tailed shrew        14.9
5 Cow                                4
6 Three-toed sloth                  14.4
```

Now in this case, we will pipe the msleep data frame to the function that will select two columns (`name` and `sleep\_total`) and then pipe the new data frame to the function `head()`, which will return the head of the new data frame.

```
msleep |>
    select(name, sleep_total) |>
    head()
```

```
# A tibble: 6 x 2
  name                        sleep_total
  <chr>                             <dbl>
1 Cheetah                            12.1
2 Owl monkey                         17
3 Mountain beaver                    14.4
4 Greater short-tailed shrew         14.9
5 Cow                                 4
6 Three-toed sloth                   14.4
```

You will soon see how useful the pipe operator is when we start to combine many functions.

Now that you know about the pipe operator (|>), we will use it throughout the rest of this tutorial.

### 15.8.1 Arrange Or Re-order Rows Using `arrange()`

To arrange (or re-order) rows by a particular column, such as the taxonomic order, list the name of the column you want to arrange the rows by:

```
msleep |> arrange(order) |> head()
```

```
# A tibble: 6 x 11
  name    genus vore  order conservation sleep_total sleep_rem sleep_cycle awake
  <chr>   <chr> <chr> <chr> <chr>              <dbl>     <dbl>       <dbl> <dbl>
1 Tenrec  Tenr~ omni  Afro~ <NA>                15.6       2.3      NA       8.4
2 Cow     Bos   herbi Arti~ domesticated         4         0.7       0.667  20
3 Roe de~ Capr~ herbi Arti~ lc                   3        NA        NA      21
4 Goat    Capri herbi Arti~ lc                   5.3       0.6      NA      18.7
```

```
5 Giraffe Gira~ herbi Arti~ cd                   1.9       0.4      NA      22.1
6 Sheep   Ovis  herbi Arti~ domesticated         3.8       0.6      NA      20.2
# i 2 more variables: brainwt <dbl>, bodywt <dbl>
```

Now we will select three columns from msleep, arrange the rows by the taxonomic order and then arrange the rows by sleep_total. Finally, show the head of the final data frame:

```
msleep |>
    select(name, order, sleep_total) |>
    arrange(order, sleep_total) |>
    head()
```

```
# A tibble: 6 x 3
  name     order        sleep_total
  <chr>    <chr>              <dbl>
1 Tenrec   Afrosoricida        15.6
2 Giraffe  Artiodactyla         1.9
3 Roe deer Artiodactyla         3
4 Sheep    Artiodactyla         3.8
5 Cow      Artiodactyla         4
6 Goat     Artiodactyla         5.3
```

Same as above, except here we filter the rows for mammals that sleep for 16 or more hours, instead of showing the head of the final data frame:

```
msleep |>
    select(name, order, sleep_total) |>
    arrange(order, sleep_total) |>
    filter(sleep_total >= 16)
```

```
# A tibble: 8 x 3
  name                   order           sleep_total
  <chr>                  <chr>                 <dbl>
1 Big brown bat          Chiroptera             19.7
2 Little brown bat       Chiroptera             19.9
3 Long-nosed armadillo   Cingulata              17.4
4 Giant armadillo        Cingulata              18.1
5 North American Opossum Didelphimorphia        18
6 Thick-tailed opposum   Didelphimorphia        19.4
7 Owl monkey             Primates               17
8 Arctic ground squirrel Rodentia               16.6
```

For something slightly more complicated do the same as above, except arrange the rows in the sleep_total column in a descending order. For this, use the function `desc()`

```
msleep |>
    select(name, order, sleep_total) |>
    arrange(order, desc(sleep_total)) |>
    filter(sleep_total >= 16)
```

```
# A tibble: 8 x 3
  name                   order           sleep_total
  <chr>                  <chr>                 <dbl>
1 Little brown bat       Chiroptera             19.9
2 Big brown bat          Chiroptera             19.7
3 Giant armadillo        Cingulata              18.1
4 Long-nosed armadillo   Cingulata              17.4
5 Thick-tailed opposum   Didelphimorphia        19.4
6 North American Opossum Didelphimorphia        18
7 Owl monkey             Primates               17
8 Arctic ground squirrel Rodentia               16.6
```

## 15.9 Create New Columns Using `mutate()`

The `mutate()` function will add new columns to the data frame. Create a new column called rem_proportion, which is the ratio of rem sleep to total amount of sleep.

```
msleep |>
    mutate(rem_proportion = sleep_rem / sleep_total) |>
    head()
```

```
# A tibble: 6 x 12
  name    genus vore  order conservation sleep_total sleep_rem sleep_cycle awake
  <chr>   <chr> <chr> <chr> <chr>              <dbl>     <dbl>       <dbl> <dbl>
1 Cheetah Acin~ carni Carn~ lc                  12.1      NA        NA      11.9
2 Owl mo~ Aotus omni  Prim~ <NA>                17         1.8      NA       7
3 Mounta~ Aplo~ herbi Rode~ nt                  14.4       2.4      NA       9.6
4 Greate~ Blar~ omni  Sori~ lc                  14.9       2.3       0.133   9.1
5 Cow     Bos   herbi Arti~ domesticated         4         0.7       0.667  20
6 Three-~ Brad~ herbi Pilo~ <NA>                14.4       2.2       0.767   9.6
# i 3 more variables: brainwt <dbl>, bodywt <dbl>, rem_proportion <dbl>
```

You can add many new columns using mutate (separated by commas). Here we add a second column called bodywt_grams which is the bodywt column in grams.

```
msleep |>
    mutate(rem_proportion = sleep_rem / sleep_total,
           bodywt_grams = bodywt * 1000) |>
    head()
```

```
# A tibble: 6 x 13
  name    genus vore  order conservation sleep_total sleep_rem sleep_cycle awake
  <chr>   <chr> <chr> <chr> <chr>              <dbl>     <dbl>       <dbl> <dbl>
1 Cheetah Acin~ carni Carn~ lc                  12.1      NA        NA      11.9
2 Owl mo~ Aotus omni  Prim~ <NA>                17         1.8      NA       7
3 Mounta~ Aplo~ herbi Rode~ nt                  14.4       2.4      NA       9.6
4 Greate~ Blar~ omni  Sori~ lc                  14.9       2.3       0.133   9.1
5 Cow     Bos   herbi Arti~ domesticated         4         0.7       0.667  20
6 Three-~ Brad~ herbi Pilo~ <NA>                14.4       2.2       0.767   9.6
# i 4 more variables: brainwt <dbl>, bodywt <dbl>, rem_proportion <dbl>,
#   bodywt_grams <dbl>
```

Is there a relationship between `rem_proportion` and `bodywt`? How about `sleep_total`?

### 15.9.1 Create summaries: `summarise()`

The `summarise()` function will create summary statistics for a given column in the data frame such as finding the mean. For example, to compute the average number of hours of sleep, apply the `mean()` function to the column sleep_total and call the summary value avg_sleep.

```
msleep |>
    summarise(avg_sleep = mean(sleep_total))
```

```
# A tibble: 1 x 1
  avg_sleep
      <dbl>
1      10.4
```

There are many other summary statistics you could consider such `sd()`, `min()`, `max()`, `median()`, `sum()`, `n()` (returns the length of vector), `first()` (returns first value in vector), `last()` (returns last value in vector) and `n_distinct()` (number of distinct values in vector).

```
msleep |>
    summarise(avg_sleep = mean(sleep_total),
              min_sleep = min(sleep_total),
              max_sleep = max(sleep_total),
              total = n())
```

```
# A tibble: 1 x 4
  avg_sleep min_sleep max_sleep total
      <dbl>     <dbl>     <dbl> <int>
1      10.4       1.9      19.9    83
```

## 15.10 Grouping data: `group_by()`

The `group_by()` verb is an important function in dplyr. The `group_by` allows us to use the concept of "split-apply-combine". We literally want to split the data frame by some variable (e.g. taxonomic order), apply a function to the individual data frames and then combine the output. This approach is similar to the `aggregate` function from R, but `group_by` integrates with dplyr.

Let's do that: split the msleep data frame by the taxonomic order, then ask for the same summary statistics as above. We expect a set of summary statistics for each taxonomic order.

```
msleep |>
    group_by(order) |>
    summarise(avg_sleep = mean(sleep_total),
              min_sleep = min(sleep_total),
              max_sleep = max(sleep_total),
              total = n())
```

```
# A tibble: 19 x 5
   order           avg_sleep min_sleep max_sleep total
   <chr>               <dbl>     <dbl>     <dbl> <int>
 1 Afrosoricida        15.6       15.6      15.6     1
 2 Artiodactyla         4.52       1.9       9.1     6
 3 Carnivora           10.1        3.5      15.8    12
 4 Cetacea              4.5        2.7       5.6     3
 5 Chiroptera          19.8       19.7      19.9     2
 6 Cingulata           17.8       17.4      18.1     2
 7 Didelphimorphia     18.7       18        19.4     2
 8 Diprotodontia       12.4       11.1      13.7     2
```

| | | | | | |
|---|---|---|---|---|---|
| 9 | Erinaceomorpha | 10.2 | 10.1 | 10.3 | 2 |
| 10 | Hyracoidea | 5.67 | 5.3 | 6.3 | 3 |
| 11 | Lagomorpha | 8.4 | 8.4 | 8.4 | 1 |
| 12 | Monotremata | 8.6 | 8.6 | 8.6 | 1 |
| 13 | Perissodactyla | 3.47 | 2.9 | 4.4 | 3 |
| 14 | Pilosa | 14.4 | 14.4 | 14.4 | 1 |
| 15 | Primates | 10.5 | 8 | 17 | 12 |
| 16 | Proboscidea | 3.6 | 3.3 | 3.9 | 2 |
| 17 | Rodentia | 12.5 | 7 | 16.6 | 22 |
| 18 | Scandentia | 8.9 | 8.9 | 8.9 | 1 |
| 19 | Soricomorpha | 11.1 | 8.4 | 14.9 | 5 |

# 16 Case Study: Behavioral Risk Factor Surveillance System

## 16.1 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.

## 16.2 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 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)
```

## 16.3 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:

```
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.

## 16.4 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
```

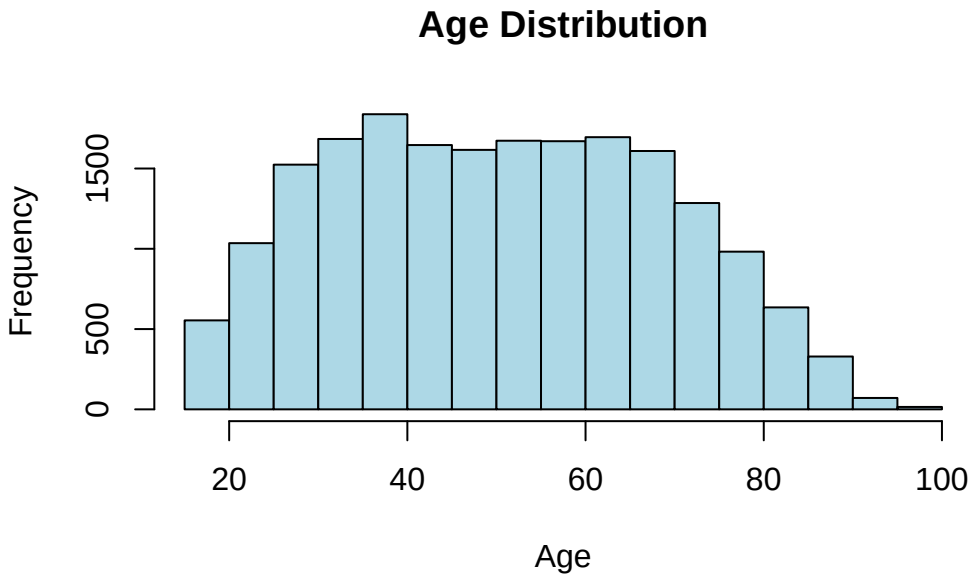
This will display the minimum, first quartile, median, mean, third quartile, and maximum for each numeric variable, and the frequency counts for each factor level for categorical variables.

## 16.5 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")
```

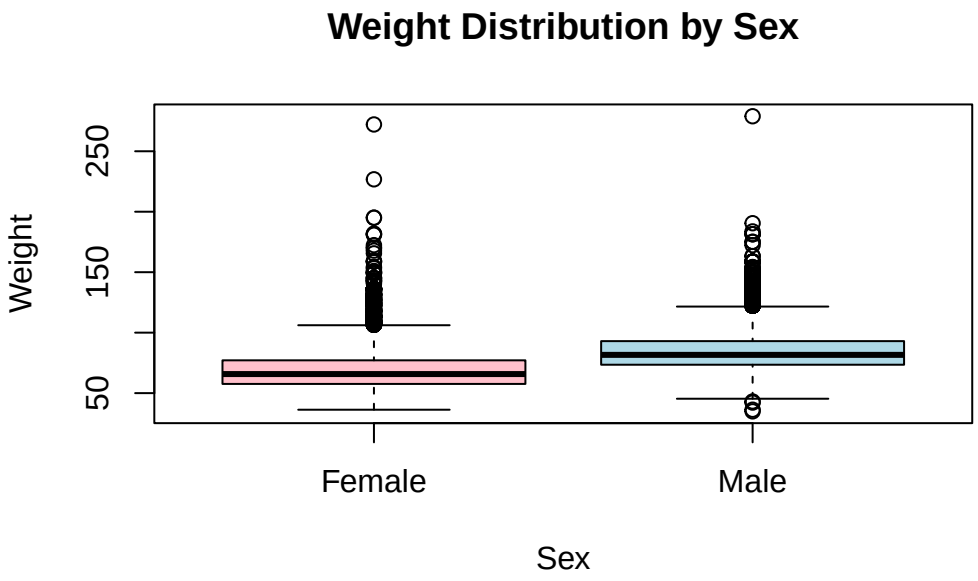



What are the options for a histogram?

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.

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"))
```

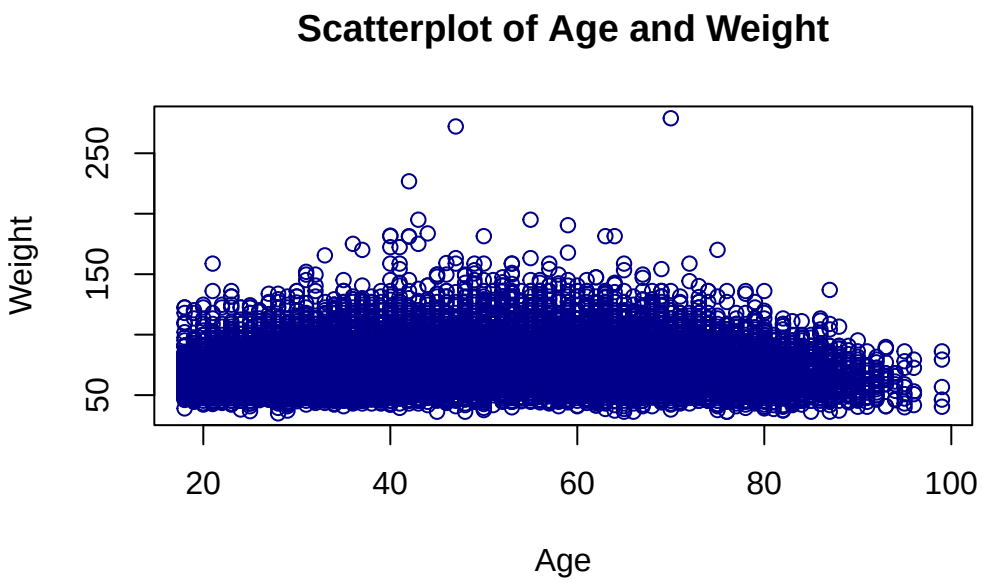



## 16.6 Analyzing Relationships Between Variables

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")
```




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
```

## 16.7 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?

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

```
[1] 75.42455
```

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

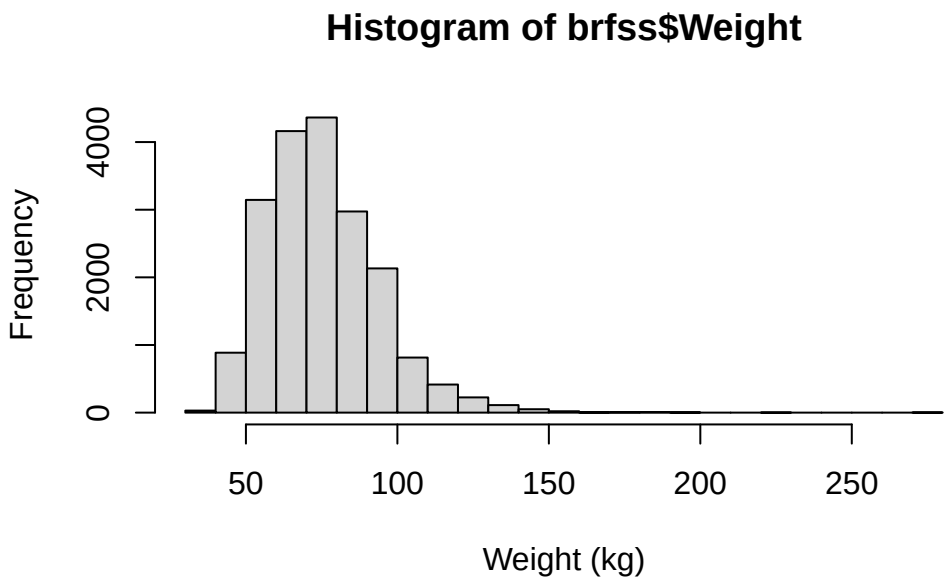
```
[1] 72.57478
```

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

```
[1] 2.849774
```

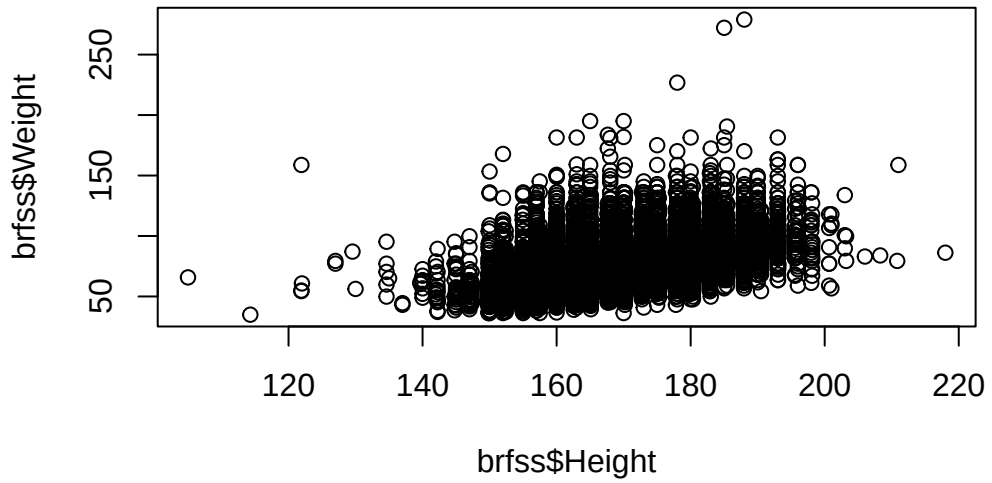
2. Given the findings about the `mean` and `median` in the previous exercise, use the `hist()` function to create a histogram of the weight distribution in this dataset. How would you describe the shape of this distribution?

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




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

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



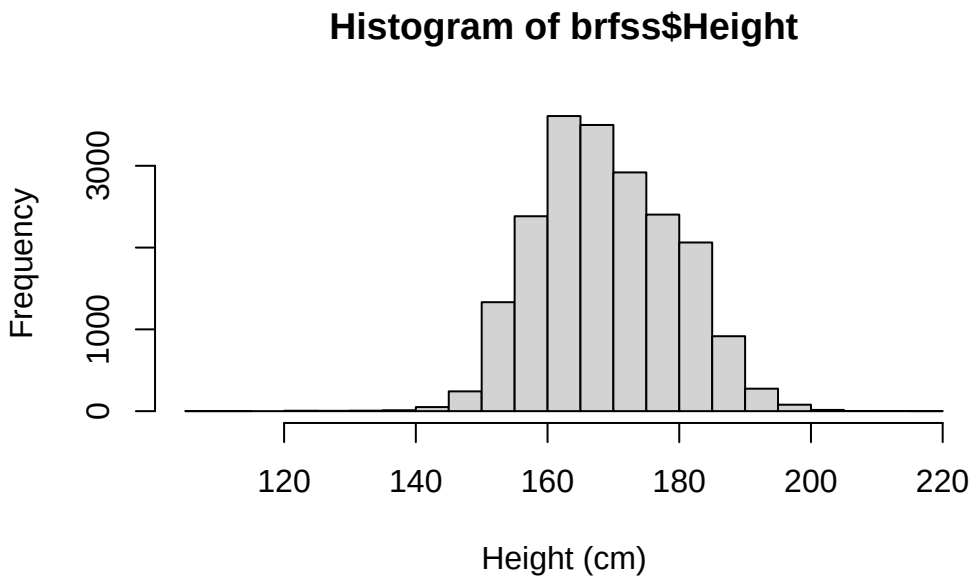

4. What is the correlation between height and weight? What does this tell you about the relationship between these two variables?

```
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)
```



## 16.8 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.

## 16.9 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()`.

## 16.10 Clean data

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

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

## 16.11 Weight in 1990 vs. 2010 Females

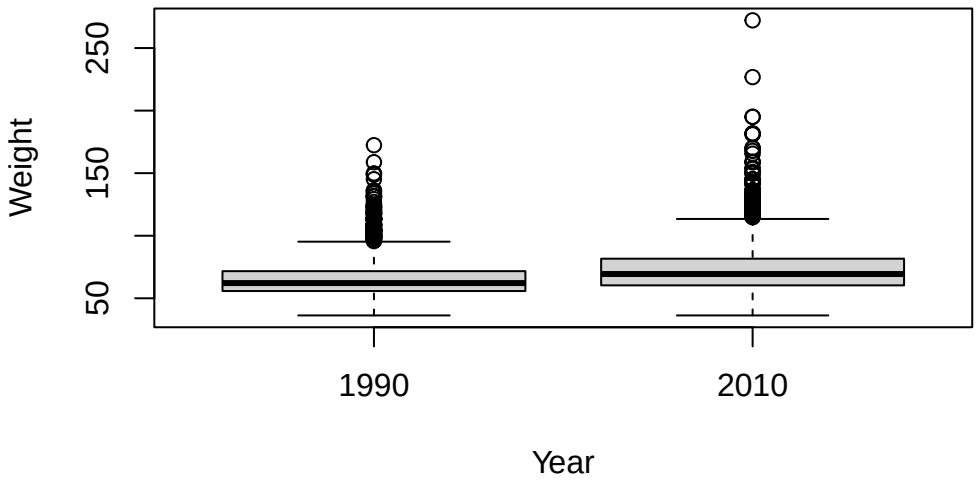
- Create a subset of the data

```
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)
```




- 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 equ
95 percent confidence interval:
 -8.723607 -7.548102
sample estimates:
mean in group 1990 mean in group 2010
          64.81838           72.95424
```

## 16.12 Weight and height in 2010 Males

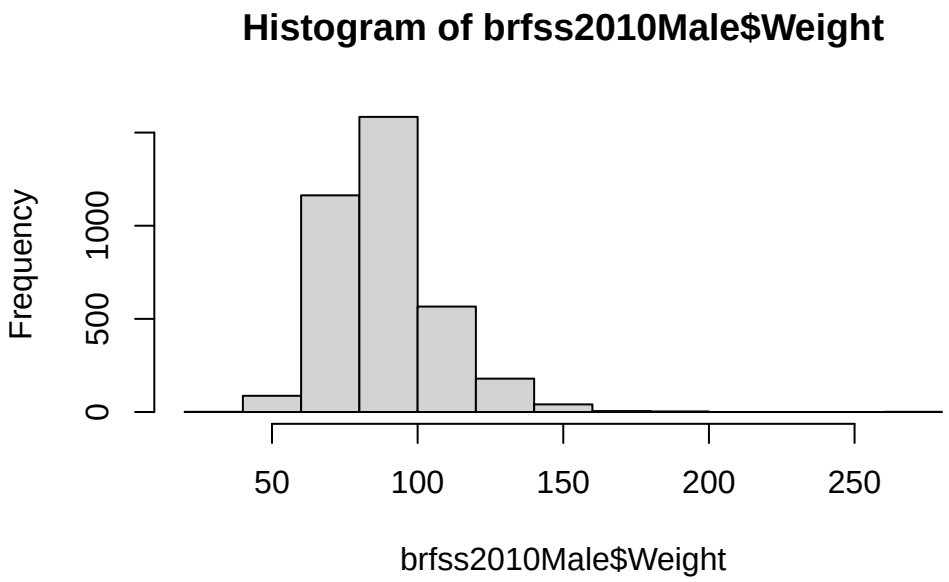
- Create a subset of the data

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

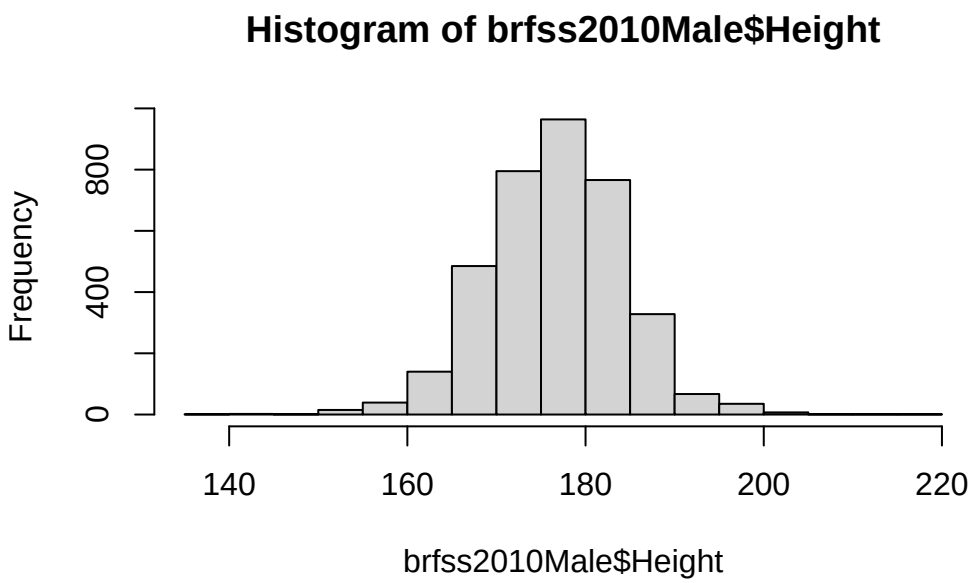
```
      Age             Weight           Sex                 Height       Year
 Min.   :18.00   Min.   : 36.29   Length:3679        Min.   :135   1990:   0
 1st Qu.:45.00   1st Qu.: 77.11   Class :character   1st Qu.:173   2010:3679
 Median :57.00   Median : 86.18   Mode  :character   Median :178
 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

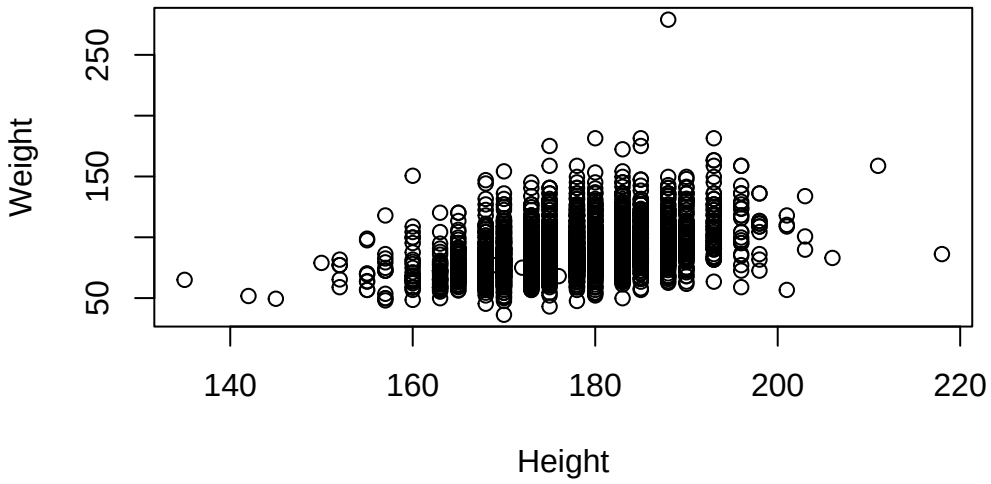
```
hist(brfss2010Male$Weight)
```




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



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




- Fit a linear model (regression)

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

```

Call:
lm(formula = Weight ~ Height, data = brfss2010Male)

Coefficients:
(Intercept)       Height
   -86.8747       0.9873
```

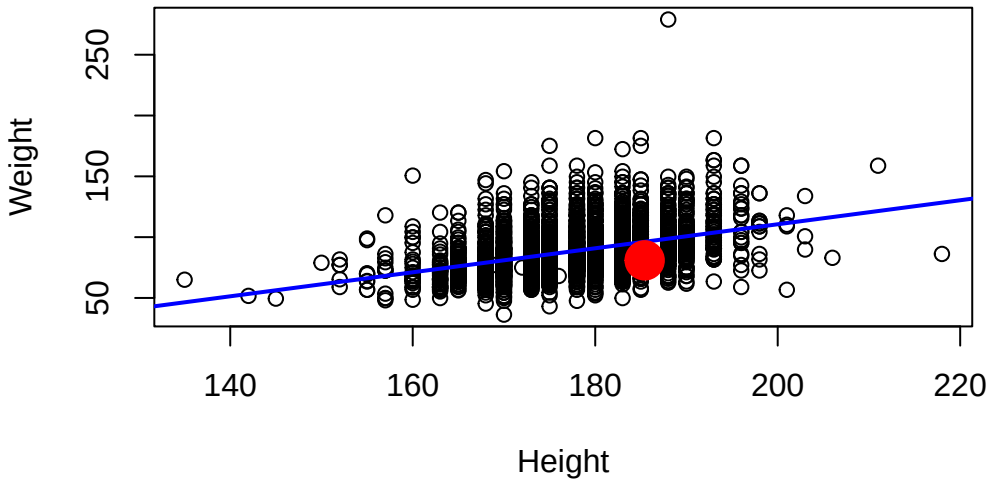
Summarize as ANOVA table

```
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)
```




- 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
```

# 17 Exploring data with ggplot2

This chapter is based on the Intro to ggplot2 chapter from the book Modern Data Visualization with R by Robert Kabacoff, which is licensed under the Creative Commons Attribution-NonCommercial 4.0 International License. The original chapter has been modified to fit the context of this book.

The insurance dataset is described in the book Machine Learning with R by Brett Lantz. A cleaned version of the dataset is also available on Kaggle. The dataset describes medical information and costs billed by health insurance companies in 2013, as compiled by the United States Census Bureau. Variables include age, sex, body mass index, number of children covered by health insurance, smoker status, US region, and individual medical costs billed by health insurance for 1338 individuals.

In this chapter, we will explore the dataset using `ggplot2`, a powerful visualization package in R.

To get started, we need to install and load the `ggplot2` package. If you haven't installed it yet, you can do so using the following command:

```
install.packages("ggplot2")
```

Once installed, load the package:

```
library(ggplot2)
```

Next, we will read the insurance dataset into R. We'll use a convenient online version of the dataset and use the `read.csv` function to load it:

```
# load the data
url <- "https://tinyurl.com/mtktm8e5"
insurance <- read.csv(url)
```

In Rstudio, you can use the `View()` function to inspect the dataset:

```
# view the dataset
View(insurance)
```

Next, we'll add a variable indicating if the patient is obese or not. Obesity will be defined as a body mass index greater than or equal to 30.

```
# create an obesity variable
insurance$obese <- ifelse(insurance$bmi >= 30,
                          "obese", "not obese")
```

In building a ggplot2 graph, only the first two functions described below are required. The others are optional and can appear in any order.

## 17.1 ggplot()

The `ggplot()` function initializes the plot. It takes a data frame as its first argument and can also include aesthetic mappings (aes) that define how variables in the data are mapped to visual properties of the plot, such as x and y axes, color, size, etc.

```
# initialize the plot
ggplot(data = insurance, aes(x = age, y = expenses))
```

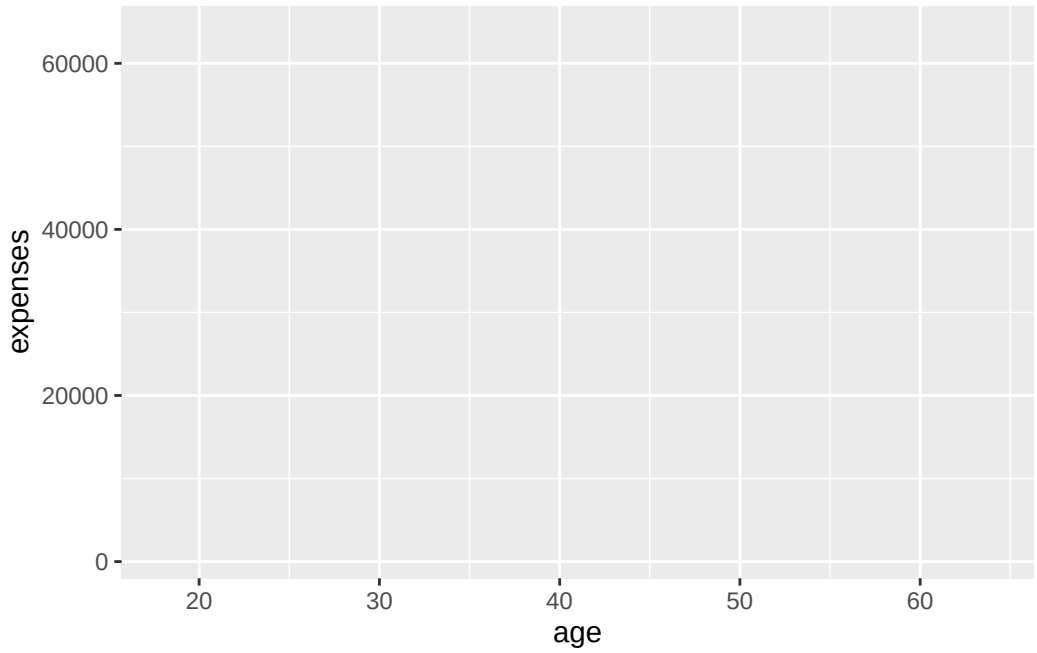



Figure 17.1: Initialize the plot with ggplot()

Why is Figure 17.1 not showing a plot? Because we have not added any layers to the plot yet. The `ggplot()` function only initializes the plot; it does not display anything until we add layers. We specified that the age variable should be mapped to the x-axis and that the expenses should be mapped to the y-axis, but we haven't yet specified what we wanted placed on the graph.

## 17.2 `geom_*()`

The `geom_*()` functions add layers to the plot. Each `geom_*()` function corresponds to a specific type of geometric object, such as points, lines, bars, etc. For example, `geom_point()` adds points to the plot, while `geom_line()` adds lines.

```
# add points to the plot
ggplot(data = insurance,
       mapping = aes(x = age, y = expenses)) +
  geom_point()
```

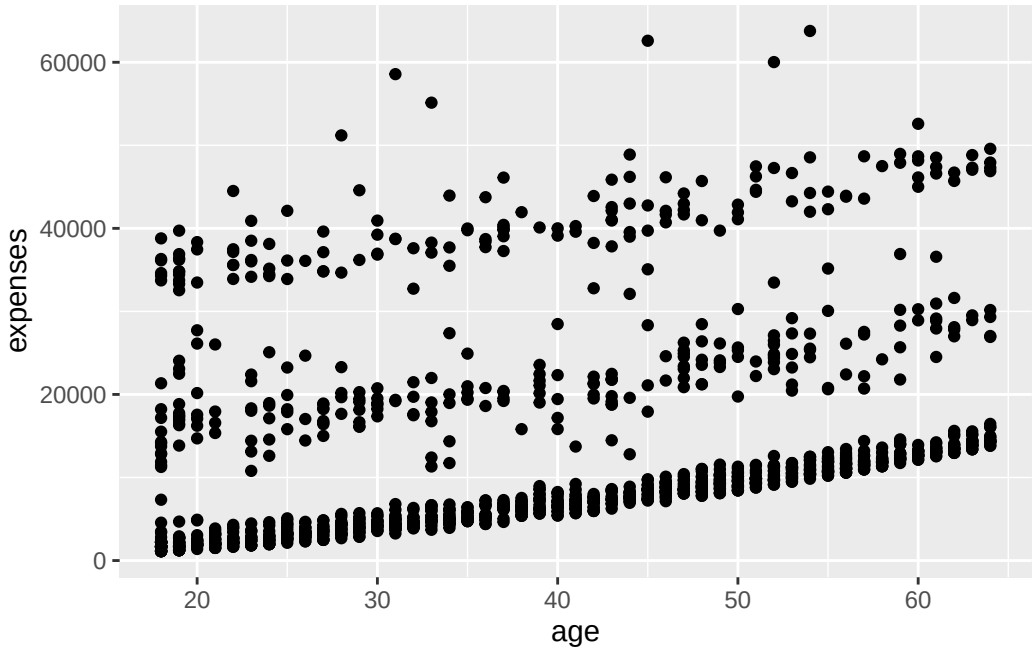



Figure 17.2: Add points to the plot with geom_point()

In Figure 17.2, we added points to the plot using `geom_point()`. The `+` operator is used to add layers to the plot. The `mapping` argument in `aes()` specifies how variables in the data are mapped to visual properties of the plot.

We can see in Figure 17.2 that insurance expenses increase with age, but there is a lot of variability in the data.

A number of parameters (options) can be specified in a geom_ function. Options for the geom_point function include color, size, and alpha. These control the point color, size, and transparency, respectively. Transparency ranges from 0 (completely transparent) to 1 (completely opaque). Adding a degree of transparency can help visualize overlapping points.

```
# make points blue, larger, and semi-transparent
ggplot(data = insurance,
       mapping = aes(x = age, y = expenses)) +
  geom_point(color = "cornflowerblue",
             alpha = .7,
             size = 2)
```

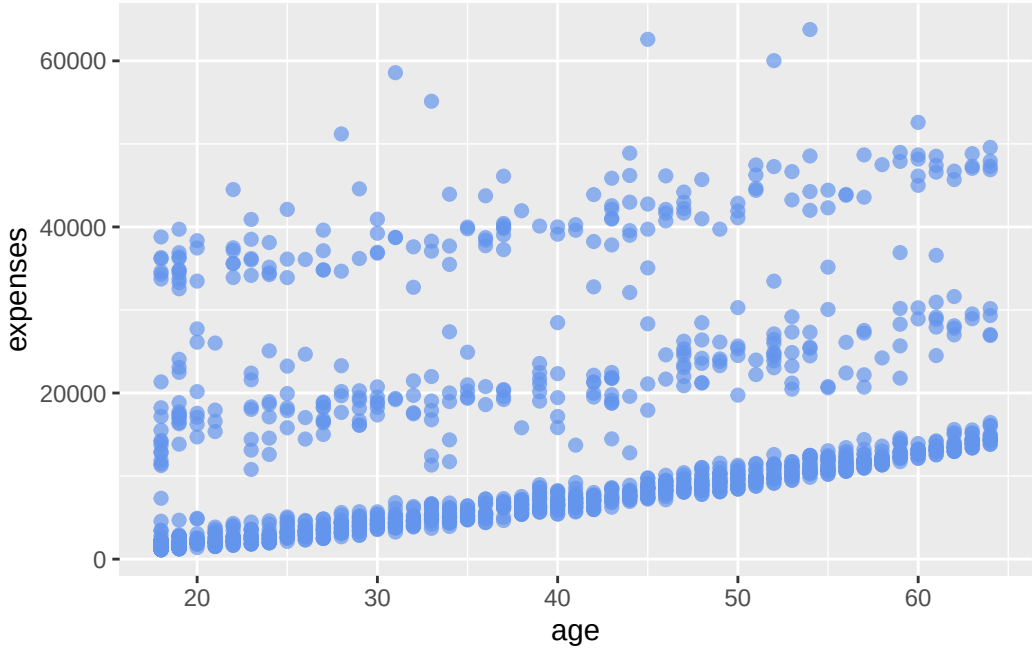



Figure 17.3: Modify point color, size, and transparency.

Next, we can add a line of best fit; in essence, we will layer on a regression fit. We can do this with the geom_smooth function. Options control the type of line (linear, quadratic, nonparametric), the thickness of the line, the line's color, and the presence or absence of a confidence interval. Here we request a linear regression (method = lm) line (where lm stands for linear model).

```
# add a line of best fit
ggplot(data = insurance,
         mapping = aes(x = age, y = expenses)) +
    geom_point(color = "cornflowerblue",
                  alpha = .7,
                  size = 2) +
  geom_smooth(method = "lm")
```

```
`geom_smooth()` using formula = 'y ~ x'
```

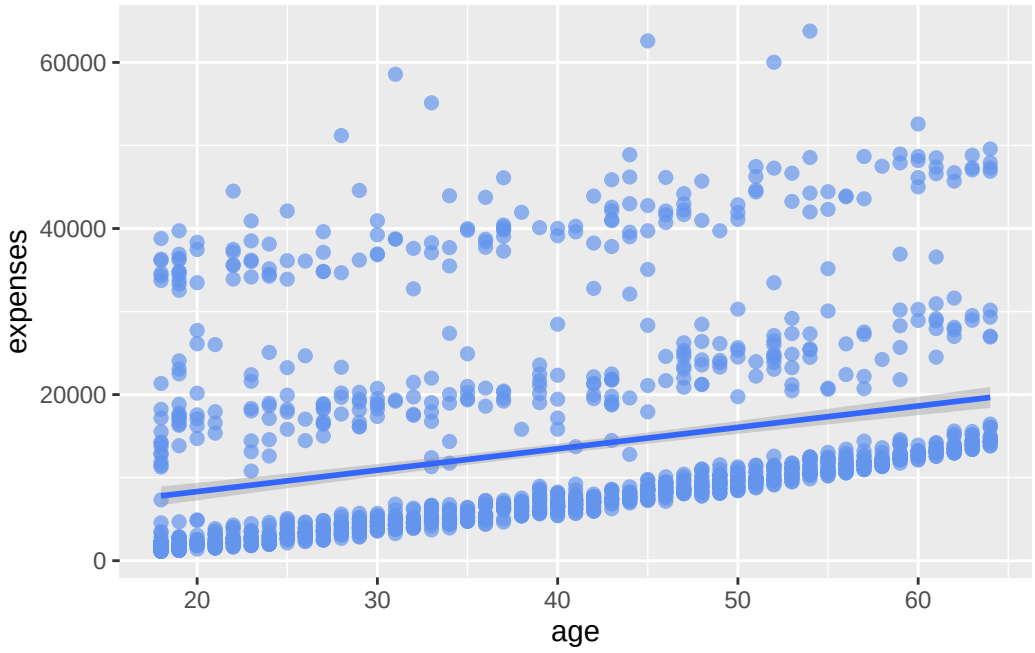



Figure 17.4: Add a line of best fit with geom_smooth()

In Figure 17.4, we added a line of best fit using `geom_smooth(method = "lm")`. The `method` argument specifies the type of smoothing to apply. In this case, we used a linear model (lm) to fit the line.

## 17.3 Grouping

In addition to mapping variables to the x and y axes, groups of observations can be mapped to the color, shape, size, transparency, and other visual characteristics of geometric objects. This allows groups of observations to be superimposed in a single graph.

Let's add smoker status to the plot and represent it by color.

```
# group points by smoker status
ggplot(data = insurance,
       mapping = aes(x = age, y = expenses, color = smoker)) +
  geom_point(alpha = .7, size = 2) +
  geom_smooth(method = "lm", se = FALSE)
```

```
`geom_smooth()` using formula = 'y ~ x'
```

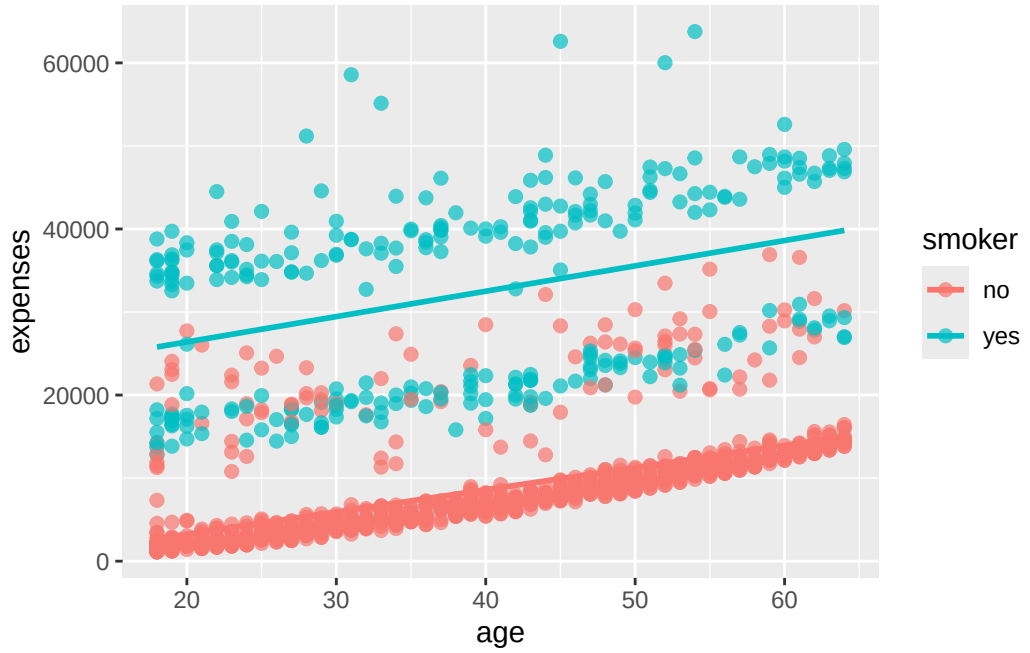



Figure 17.5: Group points by smoker status.

In Figure 17.5, we added the `color` aesthetic to the `aes()` function to map the `smoker` variable to the color of the points and the line of best fit. This allows us to see how the relationship between age and expenses differs for smokers and non-smokers. It probably comes as no surprise that smokers appear to incur greater expenses than non-smokers.

## 17.4 Scales

Scales control how variables are mapped to the visual characteristics of the plot. Scale functions (which start with `scale_`) allow you to modify this mapping. In the next plot, we'll change the x and y axis scaling, and the colors employed.

```
# modify scales for x and y axes, and colors
# modify the x and y axes and specify the colors to be used
ggplot(data = insurance,
       mapping = aes(x = age,
                     y = expenses,
                     color = smoker)) +
  geom_point(alpha = .5,
             size = 2) +
  geom_smooth(method = "lm",
              se = FALSE,
              size = 1.5) +
  scale_x_continuous(breaks = seq(0, 70, 10)) +
  scale_y_continuous(breaks = seq(0, 60000, 20000),
                     label = scales::dollar) +
  scale_color_manual(values = c("indianred3",
                                "cornflowerblue"))
```

```
Warning: Using `size` aesthetic for lines was deprecated in ggplot2 3.4.0.
i Please use `linewidth` instead.
```

```
`geom_smooth()` using formula = 'y ~ x'
```

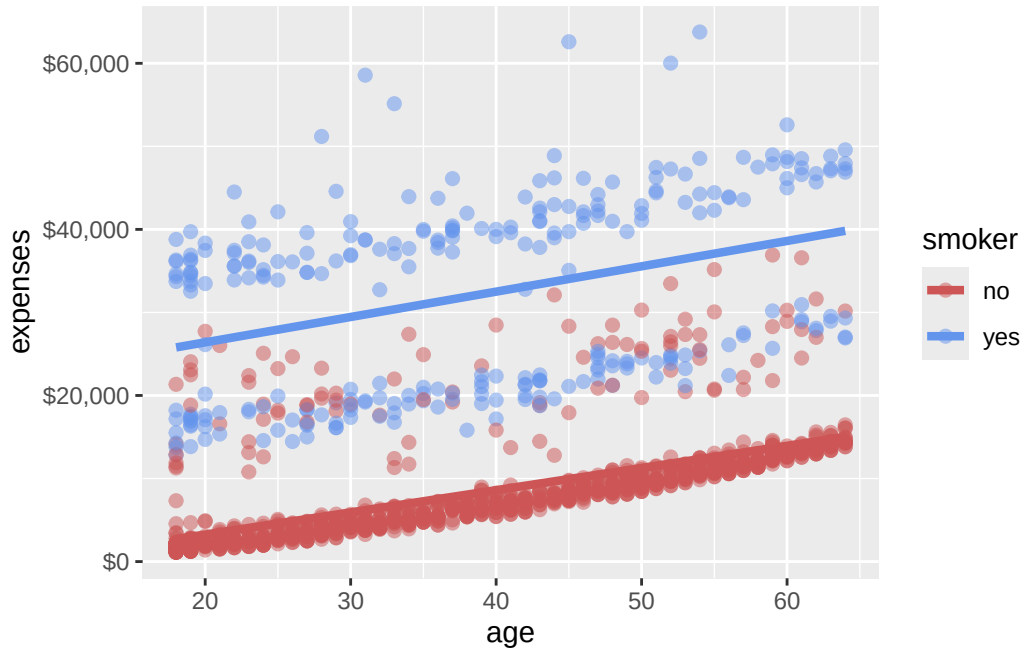



Figure 17.6: Modify scales for x and y axes, and colors.

In Figure 17.6, we used `scale_x_continuous()` and `scale_y_continuous()` to modify the x and y axes, respectively. The `breaks` argument specifies the tick marks on the axes, and the `label` argument in `scale_y_continuous()` formats the y-axis labels as dollar amounts using the `scales::dollar` function. We also used `scale_color_manual()` to specify custom colors for the points based on smoker status.

## 17.5 Facets

Faceting allows you to create multiple plots based on a categorical variable. This is useful for comparing distributions or relationships across different groups. The `facet_wrap()` function is commonly used for this purpose.

```
# create facets based on the obese variable
# reproduce plot for each obsese and non-obese individuals
ggplot(data = insurance,
       mapping = aes(x = age,
                     y = expenses,
                     color = smoker)) +
  geom_point(alpha = .5) +
  geom_smooth(method = "lm",
              se = FALSE) +
  scale_x_continuous(breaks = seq(0, 70, 10)) +
  scale_y_continuous(breaks = seq(0, 60000, 20000),
                     label = scales::dollar) +
  scale_color_manual(values = c("indianred3",
                                "cornflowerblue")) +
  facet_wrap(~obese)
```

```
`geom_smooth()` using formula = 'y ~ x'
```

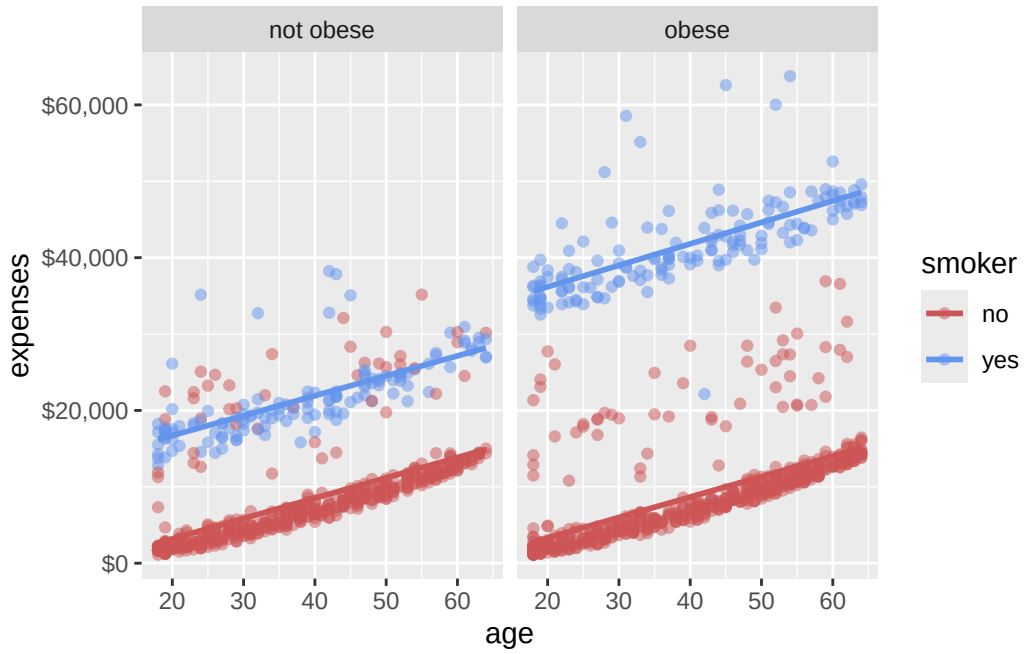



Figure 17.7: Create facets based on the obese variable.

In Figure 17.7, we used `facet_wrap(~obese)` to create separate plots for obese and non-obese individuals. This allows us to compare the relationship between age and expenses for these two groups side by side. Pretty cool, right? We have now created a plot that shows the relationships among age, smoking status, obesity, and annual medical expenses. In essence, we have placed four dimensions of data into a two-dimensional plot!

## 17.6 Labels and Titles

Labels and titles are important for making your plots informative and easy to understand. The `labs()` function is used to add labels to the x and y axes, as well as a title for the plot.

```
# add labels and title to the plot
# add informative labels
ggplot(data = insurance,
       mapping = aes(x = age,
                     y = expenses,
                     color = smoker)) +
  geom_point(alpha = .5) +
  geom_smooth(method = "lm",
              se = FALSE) +
```

```
  scale_x_continuous(breaks = seq(0, 70, 10)) +
  scale_y_continuous(breaks = seq(0, 60000, 20000),
                     label = scales::dollar) +
  scale_color_manual(values = c("indianred3",
                                "cornflowerblue")) +
  facet_wrap(~obese) +
  labs(title = "Relationship between patient demographics and medical costs",
       subtitle = "US Census Bureau 2013",
       caption = "source: http://mosaic-web.org/",
       x = " Age (years)",
       y = "Annual expenses",
       color = "Smoker?")
```

```
`geom_smooth()` using formula = 'y ~ x'
```

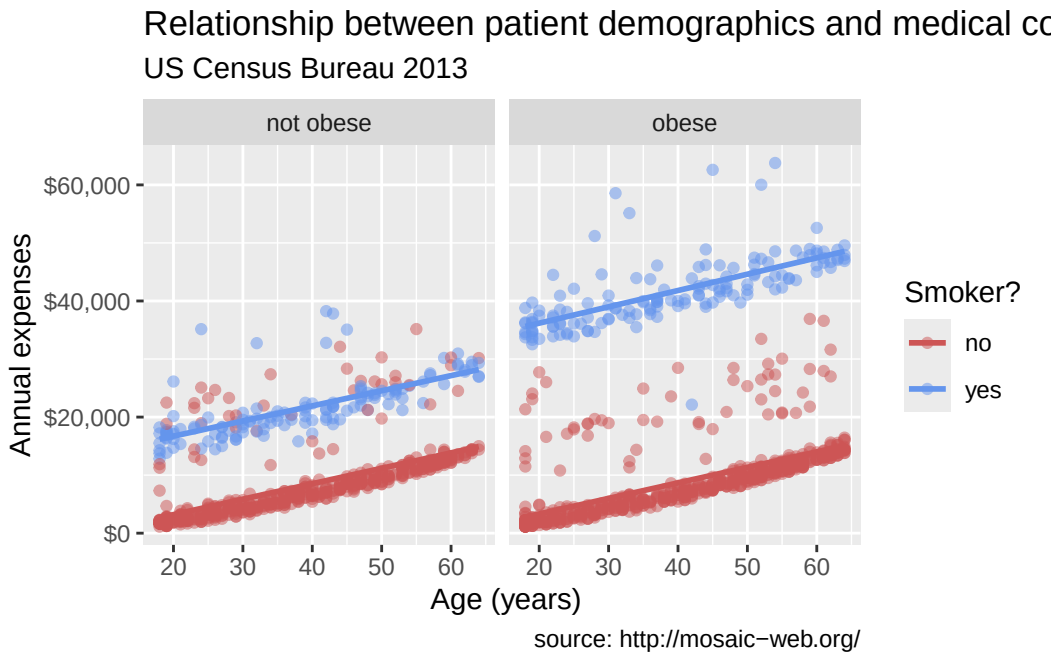



Figure 17.8: Add labels and title to the plot.

In Figure 17.8, we used the `labs()` function to add a title, subtitle, caption, and labels for the x and y axes. This makes the plot more informative and easier to interpret.

## 17.7 Theming

Finally, we can fine tune the appearance of the graph using themes. Theme functions (which start with theme_) control background colors, fonts, grid-lines, legend placement, and other non-data related features of the graph. Let's use a cleaner theme.

```
# customize the plot's appearance with themes
# use a minimalist theme
ggplot(data = insurance,
       mapping = aes(x = age,
                     y = expenses,
                     color = smoker)) +
  geom_point(alpha = .5) +
  geom_smooth(method = "lm",
              se = FALSE) +
  scale_x_continuous(breaks = seq(0, 70, 10)) +
  scale_y_continuous(breaks = seq(0, 60000, 20000),
                     label = scales::dollar) +
  scale_color_manual(values = c("indianred3",
                                "cornflowerblue")) +
  facet_wrap(~obese) +
  labs(title = "Relationship between age and medical expenses",
       subtitle = "US Census Data 2013",
       caption = "source: https://github.com/dataspelunking/MLwR",
       x = " Age (years)",
       y = "Medical Expenses",
       color = "Smoker?") +
  theme_minimal()
```

```
`geom_smooth()` using formula = 'y ~ x'
```

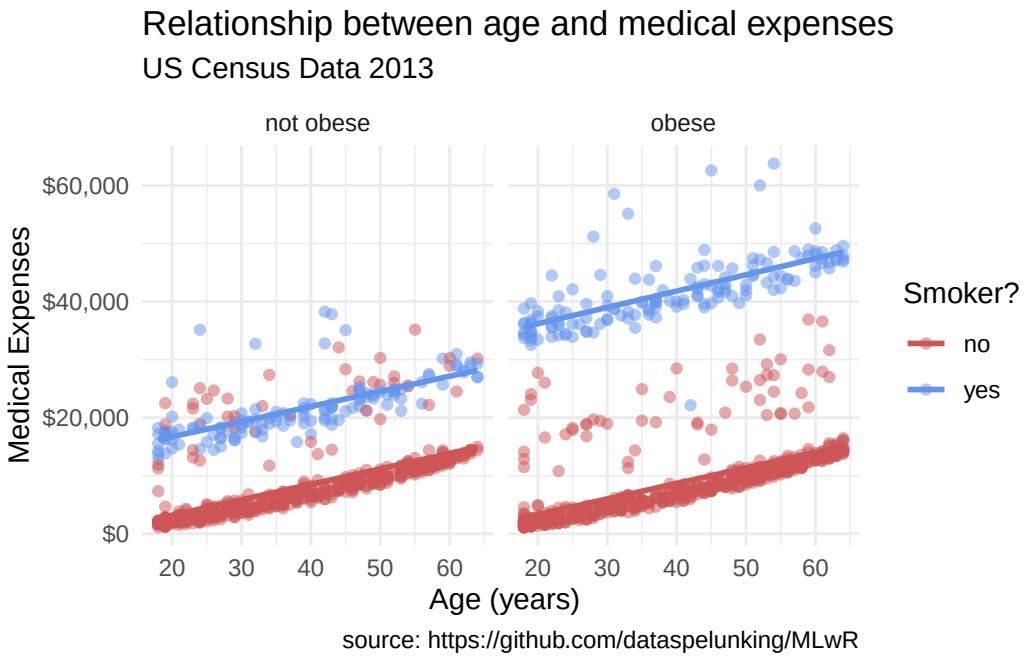



Figure 17.9: Customize the plot's appearance with themes.

## 17.8 Conclusion

In this chapter, we explored the insurance dataset using `ggplot2`, a powerful visualization package in R. We learned how to initialize a plot with `ggplot()`, add layers with `geom_*()` functions, group observations, modify scales, create facets, add labels and titles, and customize the plot's appearance with themes.

Our final plot:

Now we have something. From Figure 17.9 it appears that:

- There is a positive linear relationship between age and expenses. The relationship is constant across smoking and obesity status (i.e., the slope doesn't change).
- Smokers and obese patients have higher medical expenses.
- There is an interaction between smoking and obesity. Non-smokers look fairly similar across obesity groups. However, for smokers, obese patients have much higher expenses.
- There are some very high outliers (large expenses) among the obese smoker group.

These findings are tentative. They are based on a limited sample size and do not involve statistical testing to assess whether differences may be due to chance variation.

# 18 Base R vs tidy

Sean Davis, Martin Morgan, Lori Kern

R is flexible and often has multiple ways of accomplishing the same or similar tasks. Find the options, packages, styles, etc. that work best for you. In this chapter, we will recap some of the previous BRFSS study that utilized base R objects and graphing, and compare doing the same exact anaysis using tidyverse and ggplot2.

## 18.1 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. We will also use the `read_csv()` function from the readr package to load the tidyverse tibble data.frame. Make sure the CSV file is in your working directory, or provide the full path to the file.

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

# We will be using dplyr throughout so let's load now
library(dplyr)
```

```
# loading using base R
stopifnot(file.exists(path))
brfss_DF <- read.csv(path)
```

```
# loading using readr
library(readr)
brfss_tbl <- readr::read_csv(path)
```

Let's examine our objects:

```r
# Classic data frame
head(brfss_DF)
```

```
  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
```

```r
# Tidyverse tibble
head(brfss_tbl)
```

```
# A tibble: 6 x 5
    Age Weight Sex    Height  Year
  <dbl>  <dbl> <chr>   <dbl> <dbl>
1    31   49.0 Female   157.  1990
2    57   81.6 Female   157.  1990
3    43   80.3 Male     178.  1990
4    72   70.3 Male     170.  1990
5    31   49.9 Female   155.  1990
6    58   54.4 Female   155.  1990
```

```r
# Classic data frame
class(brfss_DF)
```

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

```r
# Tidyverse tibble
class(brfss_tbl)
```

```
[1] "spec_tbl_df" "tbl_df"      "tbl"         "data.frame"
```

> **Note**
>
> **Note:** A tidyverse tibble object inherits a data.frame class. This means that most data.frame operations like `dim()`, `colnames()`, `$`, `[`, etc. will work on the tibble object as well.

## 18.2 Clean data

Both 'Sex' and 'Year' are really `factor` values (each can only take on specific levels, 'Female' and 'Male' for 'Sex', and '1990' and '2010' for 'Year').

```
# base R / data.frame
brfss_DF$Year <- factor(brfss_DF$Year)
brfss_DF$Sex <- factor(brfss_DF$Sex)
```

```
# dplyr / tibble
brfss_tbl <-
brfss_tbl |>
    mutate(
        Sex = factor(Sex,
          levels = c("Female", "Male")),
        Year = factor(Year,
          levels = c("1990", "2010"))
    )
```

## 18.3 Data Exploration

Let's execute some basic exploration. `summary()` works the same for both objects but lets looks at some summary tables and counts. They produce the same results but in different formats.

We'll start with basic table of a single variable:

```
# base R / data.frame
table(brfss_DF$Year)
```

```
# dplyr / tibble
brfss_tbl |> count(Year)
```

```
 1990  2010
10000 10000
```

```
# A tibble: 2 x 2
  Year      n
  <fct> <int>
```

```
1 1990  10000
2 2010  10000
```

```
# base R / data.frame
table(brfss_DF$Sex)
```

```
# dplyr / tibble
brfss_tbl |> count(Sex)
```

```
Female   Male
 12039   7961
```

```
# A tibble: 2 x 2
  Sex        n
  <fct>  <int>
1 Female 12039
2 Male    7961
```

Now let's look at contingency table

```
# base R / data.frame
table(brfss_DF$Sex, brfss_DF$Year)
```

```
        1990 2010
  Female 5718 6321
  Male   4282 3679
```

```
# dplyr / tibble
brfss_tbl |> count(Sex, Year)
```

```
# A tibble: 4 x 3
  Sex    Year      n
  <fct>  <fct> <int>
1 Female 1990   5718
2 Female 2010   6321
3 Male   1990   4282
4 Male   2010   3679
```

We can get the tidy table to look even more similar to the base R table with the help of the tidyr package's function `pivot_wider`

```
# base R / data.frame
table(brfss_DF$Sex, brfss_DF$Year)
```

```
        1990 2010
  Female 5718 6321
  Male   4282 3679
```

```
# dplyr / tibble
library(tidyr)
brfss_tbl |> count(Sex, Year) |>
    tidyr::pivot_wider(names_from = "Year", values_from = "n")
```

```
# A tibble: 2 x 3
  Sex    `1990` `2010`
  <fct>   <int>  <int>
1 Female   5718   6321
2 Male     4282   3679
```

What about some summary statistics on the columns of data? `summarize()` will create the new data.frame automatically; base R you have to create your own.

```
# base R / data.frame
data.frame(
  avg_age = mean(brfss_DF$Age, na.rm = TRUE),
  ave_wt  = mean(brfss_DF$Weight, na.rm = TRUE),
  ave_ht  = mean(brfss_DF$Height, na.rm = TRUE)
)
```

```
# dplyr / tibble
brfss_tbl |>
    summarize(
        avg_age = mean(Age, na.rm = TRUE),
        ave_wt = mean(Weight, na.rm = TRUE),
        ave_ht = mean(Height, na.rm = TRUE)
    )
```

```
   avg_age   ave_wt   ave_ht
1 50.99164 75.42455 169.2131
```

```
# A tibble: 1 x 3
  avg_age ave_wt ave_ht
    <dbl>  <dbl>  <dbl>
1    51.0   75.4   169.
```

If we want to get more complex with groupings by Year and Sex, dlpyr uses `group_by` where base R would use `aggregate`.

```
# base R / data.frame
aggregate(
  cbind(Age, Weight, Height) ~ Sex + Year,
  data = brfss_DF,
  FUN = function(x) mean(x, na.rm = TRUE)
)
```

```
# dplyr / tibble
brfss_tbl |>
    group_by(Sex, Year) |>
    summarize(
        avg_age = mean(Age, na.rm = TRUE),
        ave_wt = mean(Weight, na.rm = TRUE),
        ave_ht = mean(Height, na.rm = TRUE)
    )
```

```
     Sex Year      Age   Weight   Height
1 Female 1990 46.09153 64.84333 163.2914
2   Male 1990 43.87574 81.19496 178.2242
3 Female 2010 57.07807 73.03178 163.2469
4   Male 2010 56.25465 88.91136 178.0139
```

```
# A tibble: 4 x 5
# Groups:   Sex [2]
  Sex    Year  avg_age ave_wt ave_ht
  <fct>  <fct>   <dbl>  <dbl>  <dbl>
1 Female 1990     46.2   64.8   163.
2 Female 2010     57.1   73.0   163.
3 Male   1990     43.9   81.2   178.
4 Male   2010     56.2   88.8   178.
```

## 18.4 Visualization

Before we start visualizing, lets create a few different subsets of data.

```
# base R / data.frame
brfss_female_DF <-
    brfss_DF[brfss_DF$Sex == "Female",]
brfss_male_DF <-
    brfss_DF[brfss_DF$Sex == "Male",]
brfss_2010_DF <-
    brfss_DF[brfss_DF$Year == "2010",]
```

```
# dplyr / tibble
brfss_male_tbl <-
    brfss_tbl |> filter(Sex == "Male")
brfss_female_tbl <-
    brfss_tbl |> filter(Sex == "Female")
brfss_2010_tbl <-
    brfss_tbl |> filter(Year == "2010")
```

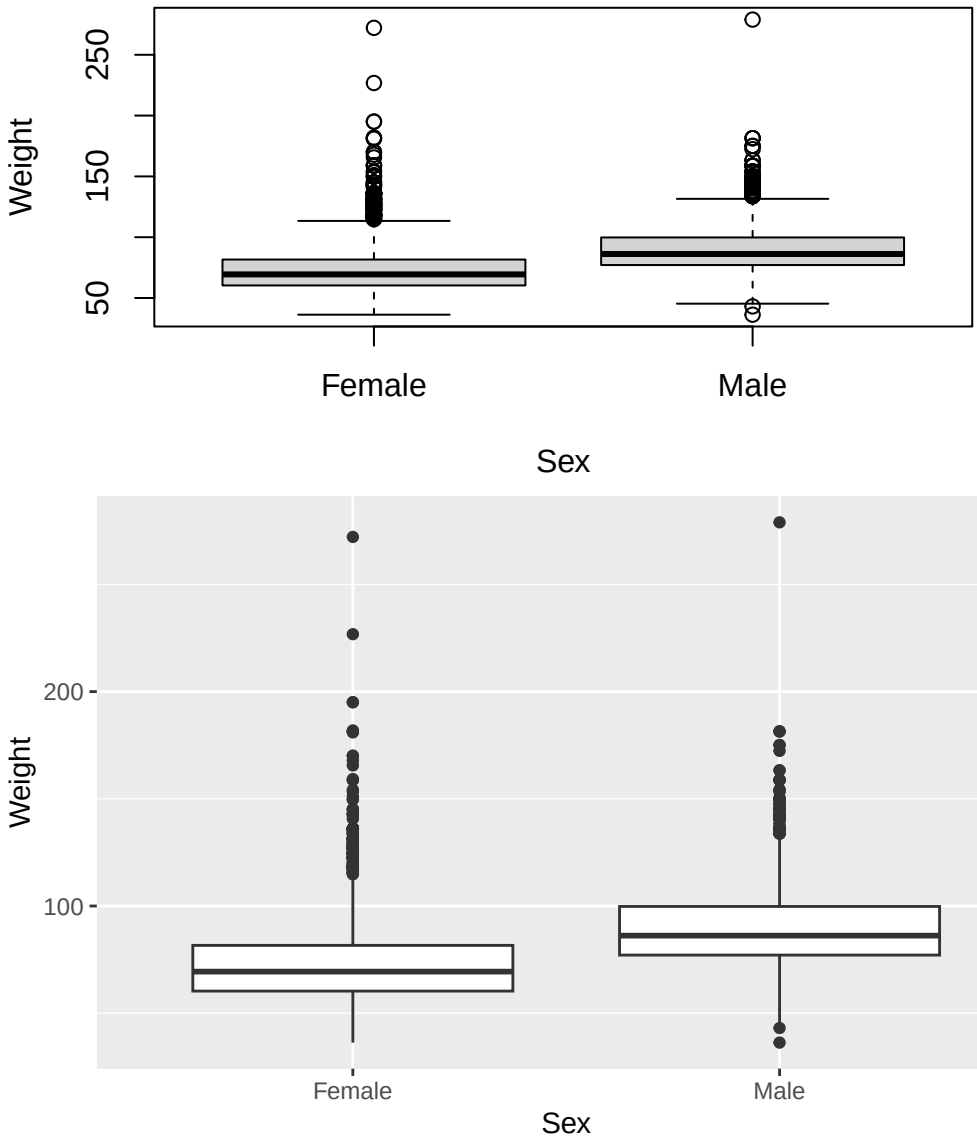
We should also load the ggplot2 package so we can compare base R graphics vs ggplot2

```
library(ggplot2)
```

Let's start with a boxplot that compares the Weights of Males vs Females for the 2010 dataset.

```
# base R
plot(Weight ~ Sex, brfss_2010_DF)
```

```
# ggplot2
ggplot(brfss_2010_tbl) +
    aes(x = Sex, y = Weight) +
    geom_boxplot()
```





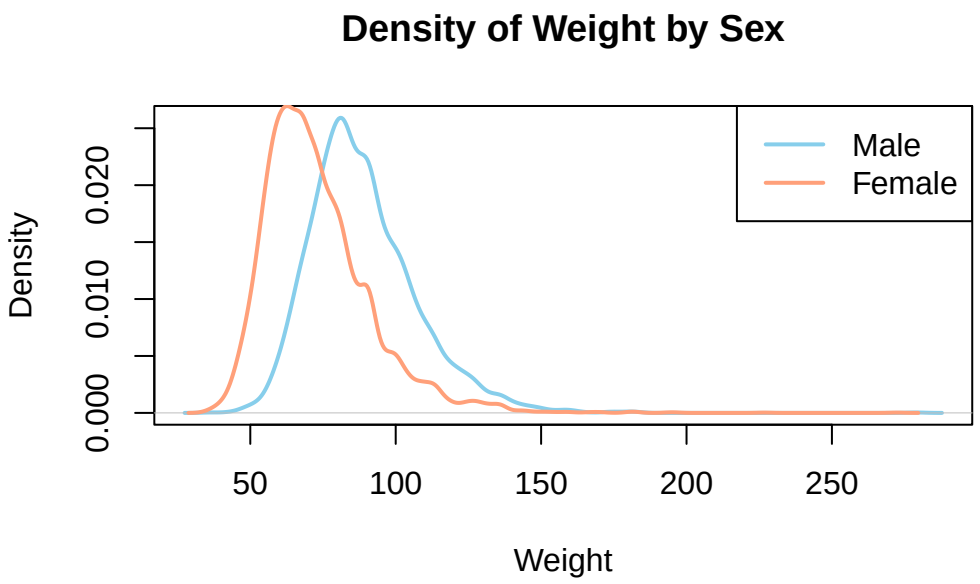


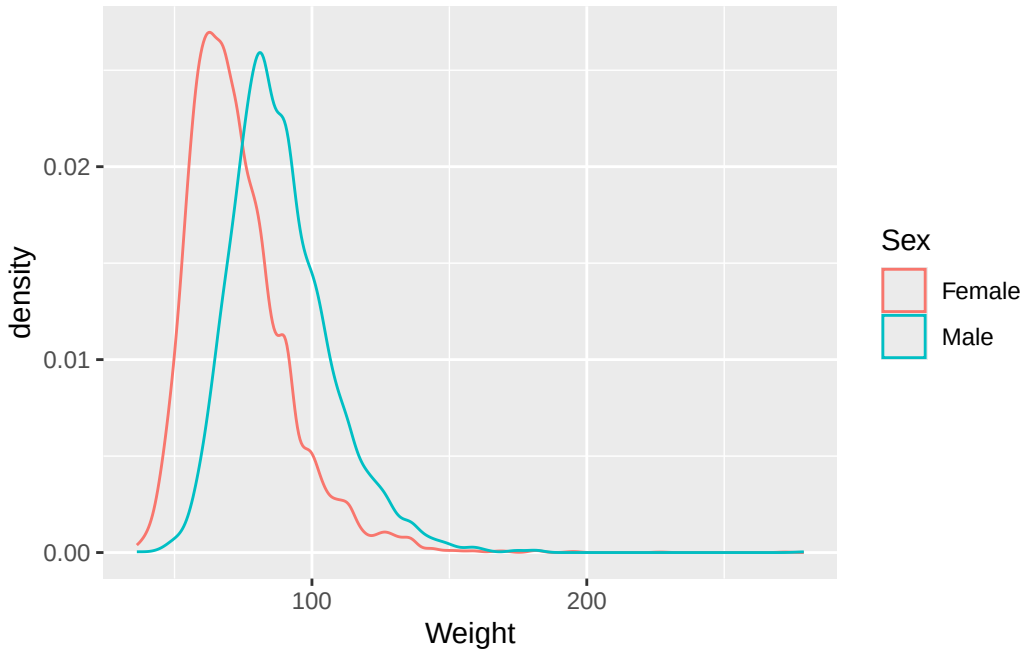
Let's look at some density and scatterplots.

```
# base R
den_male <- density(brfss_2010_DF$Weight[brfss_2010_DF$Sex == "Male"], na.rm = TRUE)
den_female <- density(brfss_2010_DF$Weight[brfss_2010_DF$Sex == "Female"], na.rm = TRUE)
plot(den_male,
     col = "skyblue", lwd = 2,
     main = "Density of Weight by Sex",
     xlab = "Weight")

lines(den_female,
```

```
      col = "lightsalmon", lwd = 2)

legend("topright",
    legend = c("Male", "Female"),
    col = c("skyblue", "lightsalmon"), lwd = 2)
```

```
# ggplot2
brfss_2010_tbl |>
    ggplot() +
    aes(x = Weight, color= Sex) +
    geom_density()
```

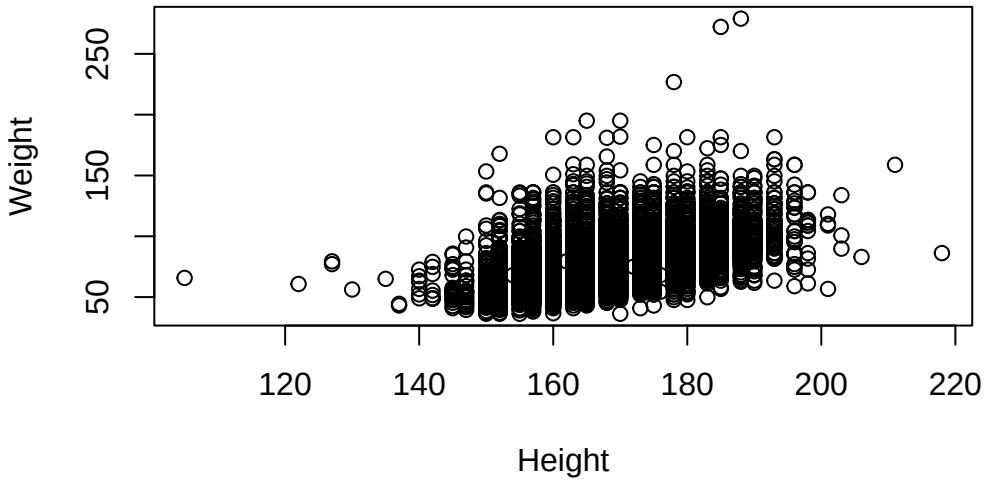


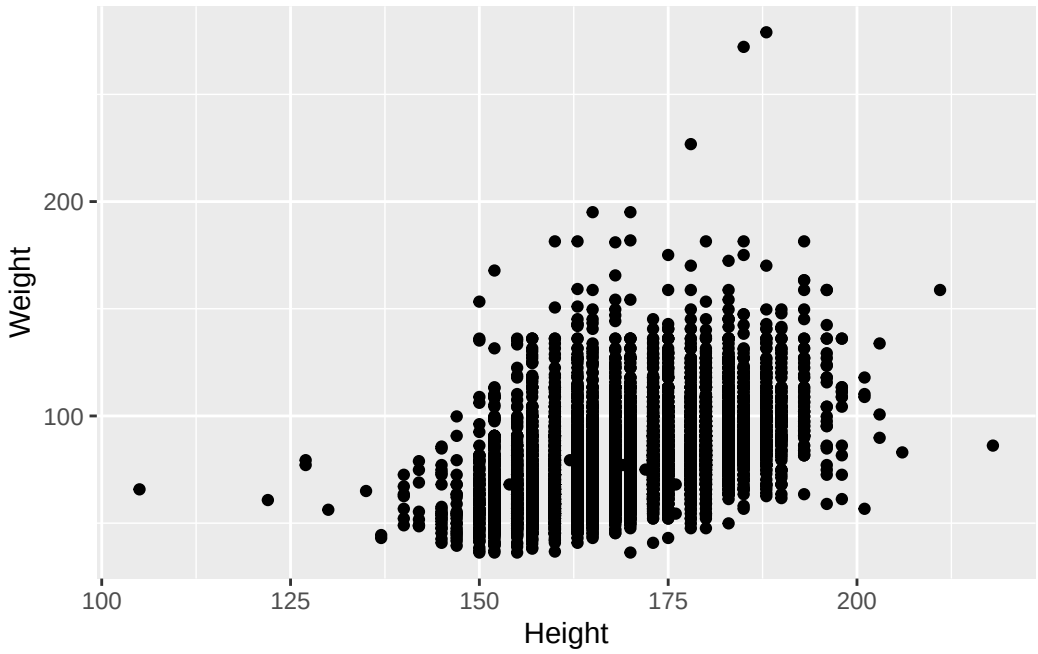



Presumably taller people are heavier than shorter people. Let's examine this relationship.

```
# base R
plot(Weight ~ Height, brfss_2010_DF)
```

```
# ggplot2
brfss_2010_tbl |>
    ggplot() +
    aes(x = Height, y = Weight) +
    geom_point()
```

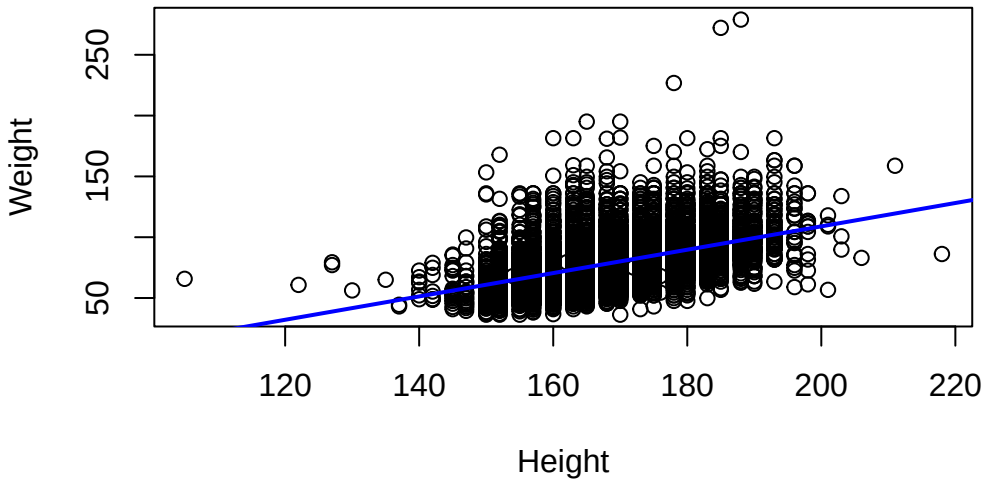


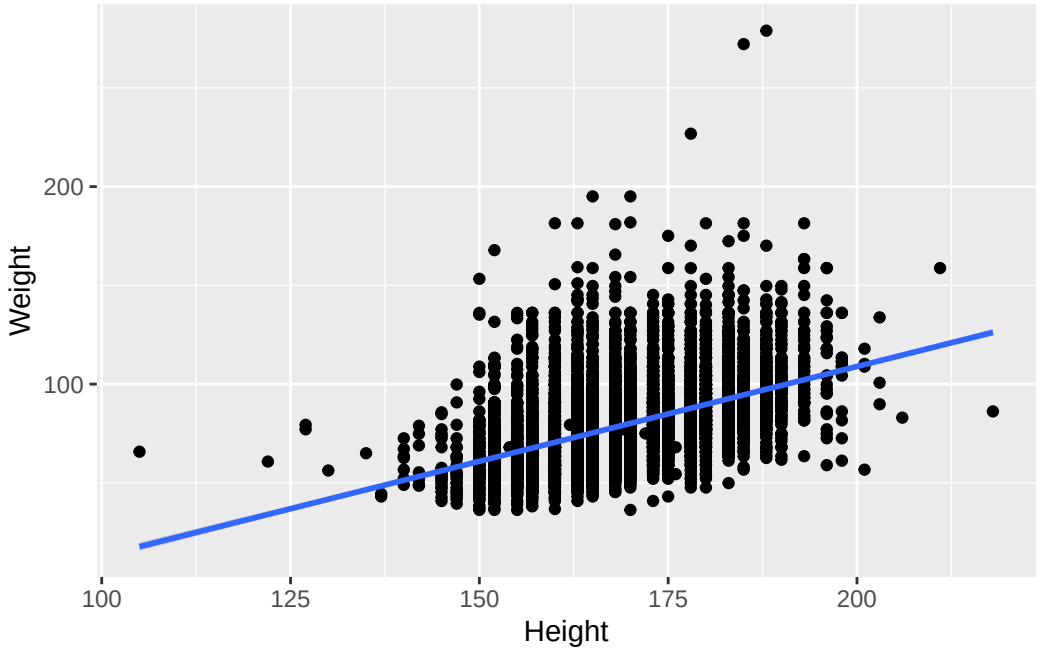



Let's fit the linear regression

```
# base R
plot(Weight ~ Height, brfss_2010_DF)
fit <- lm(Weight ~ Height, brfss_2010_DF)
abline(fit, col="blue", lwd=2)
```

```
# ggplot2
brfss_2010_tbl |>
    ggplot() +
    aes(x = Height, y = Weight) +
    geom_point() +
    geom_smooth(method = "lm")
```



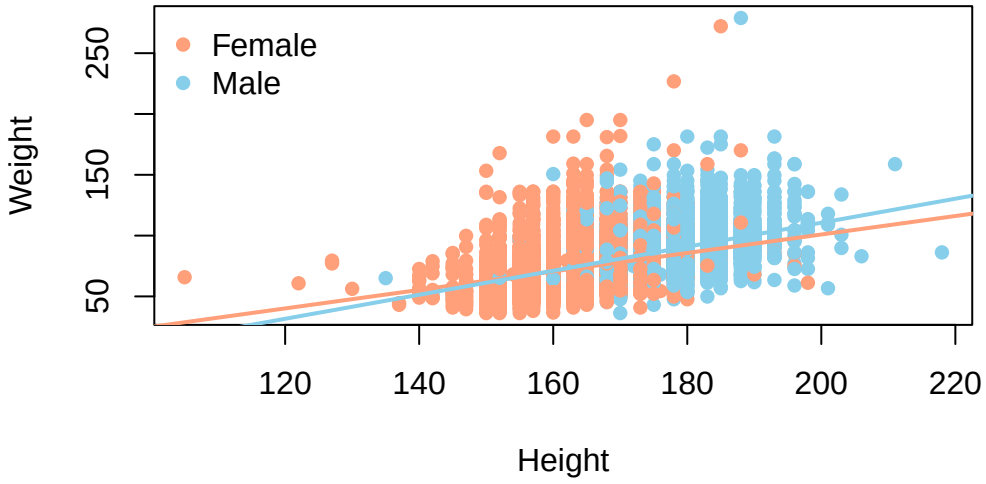



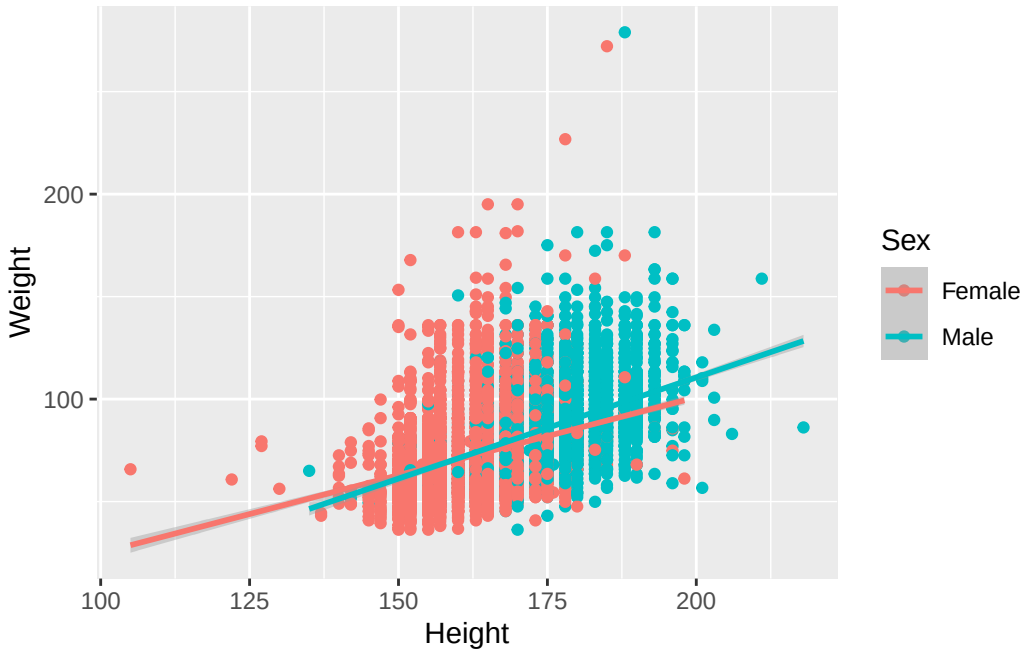

We saw that there could be a difference based on Sex. Let's add color to the points

```
# base R
colors <- c("Female" = "lightsalmon", "Male" = "skyblue")
plot(Weight ~ Height, brfss_2010_DF,
  col = colors[Sex], pch = 16)
for (sex in levels(brfss_2010_DF$Sex)) {
  subset_data <- subset(brfss_2010_DF, Sex == sex)
  fit <- lm(Weight ~ Height, data = subset_data)
  abline(fit, col = colors[sex], lwd = 2)
```

```
}
legend("topleft", legend = levels(brfss_2010_DF$Sex),
       col = colors, pch = 16, bty = "n")
```

```
# ggplot2
brfss_2010_tbl |>
    ggplot() +
    aes(x = Height, y = Weight, color = Sex) +
    geom_point() +
    geom_smooth(method = "lm")
```

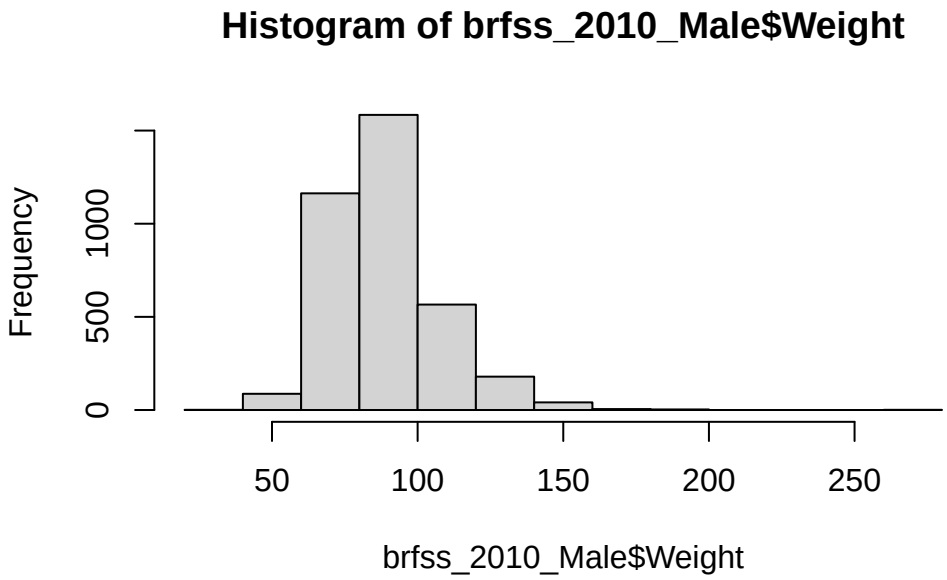


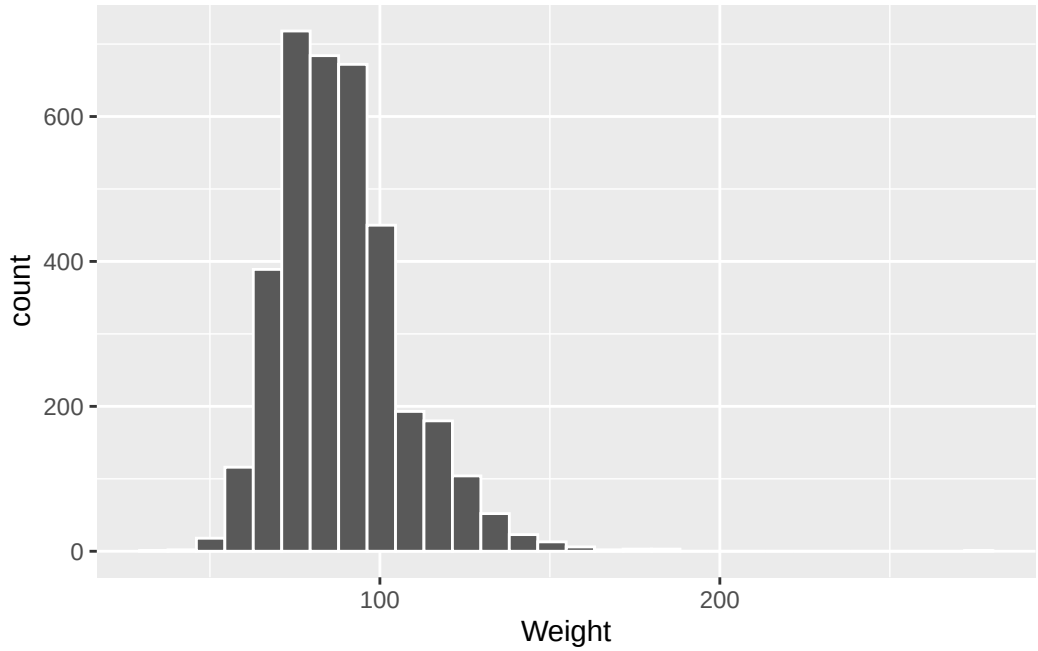



Let's dig into some visualizations of the 2010 Males and recreate the histograms of Weight but we did not create that subset.

```
# base R
brfss_2010_Male <- subset(brfss_DF,
    Year == 2010 & Sex == "Male")
hist(brfss_2010_Male$Weight)
```

```
# ggplot2
brfss_2010_tbl |> filter(Sex == "Male") |>
    ggplot() +
    aes(x = Weight) +
    geom_histogram(col = "white")
```



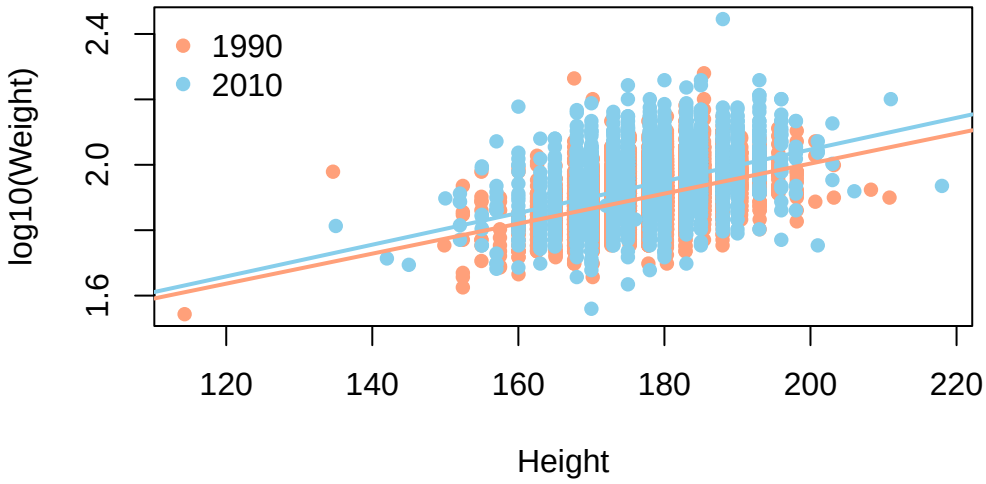



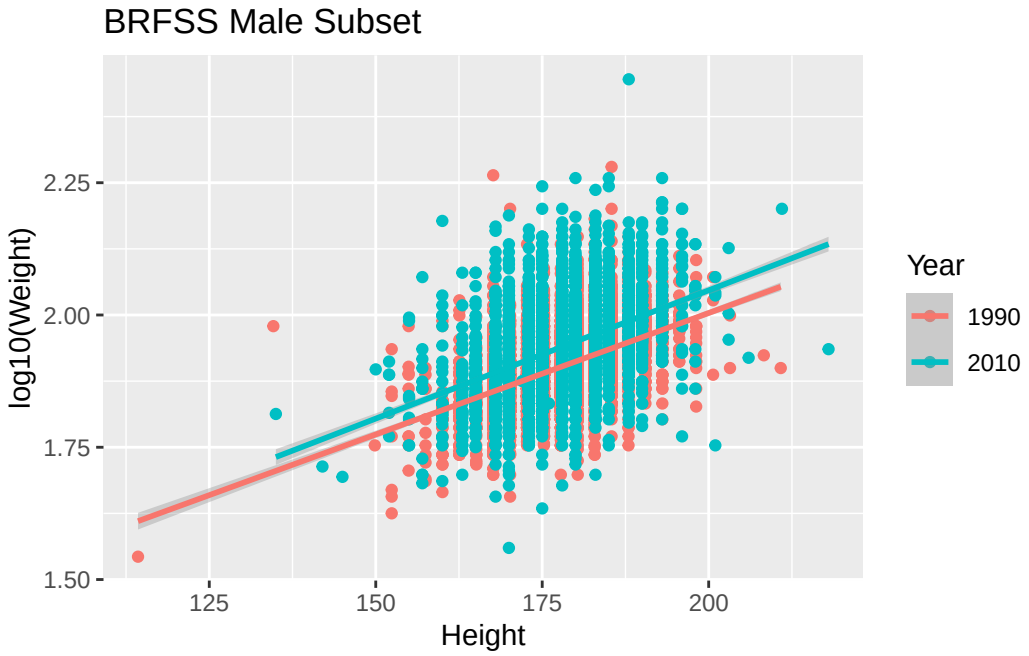

What if we took all the Males and looked to see if the relationship of Height and Weight changed between 1990 and 2010.

```
# base R
colors <- c("1990" = "lightsalmon","2010" = "skyblue")
plot(log10(Weight) ~ Height, brfss_male_DF,
  col = colors[Year], pch = 16, ylab = "log10(Weight)")
for (yr in levels(brfss_male_DF$Year)) {
```

```r
  subset_data <- subset(brfss_male_DF, Year == yr)
  fit <- lm(log10(Weight) ~ Height, data = subset_data)
  abline(fit, col = colors[yr], lwd = 2)
}
legend("topleft", legend = levels(brfss_male_DF$Year),
       col = colors, pch = 16, bty = "n")
```

```r
# ggplot2
ggplot(brfss_male_tbl) +
    aes(x = Height, y = log10(Weight), color = Year) +
    geom_point() +
    geom_smooth(method = "lm") +
    labs(title = "BRFSS Male Subset")
```






## 18.5 Summary

There are many visualization packages in R. You can explore the many options and what each has to offer to design high quality, customized plots for reporting.

# 19 Self-Guided Data Visualization in R

Data visualization is a critical skill in the data scientist's toolkit. It's the bridge between raw data and human understanding. Effective visualizations can reveal patterns, trends, and outliers that might be missed in a table of numbers. In the R programming language, the `ggplot2` package stands as the gold standard for creating beautiful, flexible, and powerful graphics.

This document will guide you through the principles of effective data visualization and show you how to apply them using `ggplot2`. We'll cover best practices, common plot types, and the "grammar of graphics" methodology that makes `ggplot2` so intuitive.

Before diving in, though, there are a some truly amazing online resources that showcase what can be done with R graphics and also stimulate your imagination. Two of the best are:

- The R Graph Gallery: A comprehensive collection of R graphics examples, covering a wide range of plot types and customization options.
- From Data to Viz: A guide that helps you choose the right type of visualization for your data and provides examples in R.

After walking through this document, go back to these resources and explore the examples. You'll see how the principles we discuss here are applied in real-world scenarios, and you'll gain inspiration for your own visualizations.

## 19.1 Getting Started with ggplot2

To get started, we need to load the `ggplot2` package, which is part of the `tidyverse`.

```
# Load the necessary R packages for data visualization
library(ggplot2)
library(dplyr)
```

## 19.2 Core Principles of Effective Data Visualization

Before we start plotting, it's essential to understand what makes a visualization effective. Two key principles are maximizing the data-ink ratio and using clear labels.

### 19.2.1 The "Least Ink" Principle

Coined by the statistician Edward Tufte, the **data-ink ratio** is the proportion of a graphic's ink devoted to the non-redundant display of data information. The goal is to maximize this ratio. In simpler terms, **every single pixel should have a reason to be there.**

Avoid chart junk like:

- Redundant grid lines
- Unnecessary backgrounds or colors
- 3D effects on 2D plots
- Shadows and other decorative elements

Let's look at an example. The first plot has a lot of "chart junk," while the second one is cleaner and focuses on the data.

```
# Using the built-in 'mpg' dataset
ggplot(mpg, aes(x = displ, y = hwy, color = class)) +
  geom_point() +
  theme_gray() + # A theme with a lot of non-data ink
  labs(title = "Fuel Efficiency vs. Engine Displacement",
       subtitle = "This is a very busy plot",
       x = "Engine Displacement (Liters)",
       y = "Highway Miles per Gallon",
       color = "Vehicle Class")
```

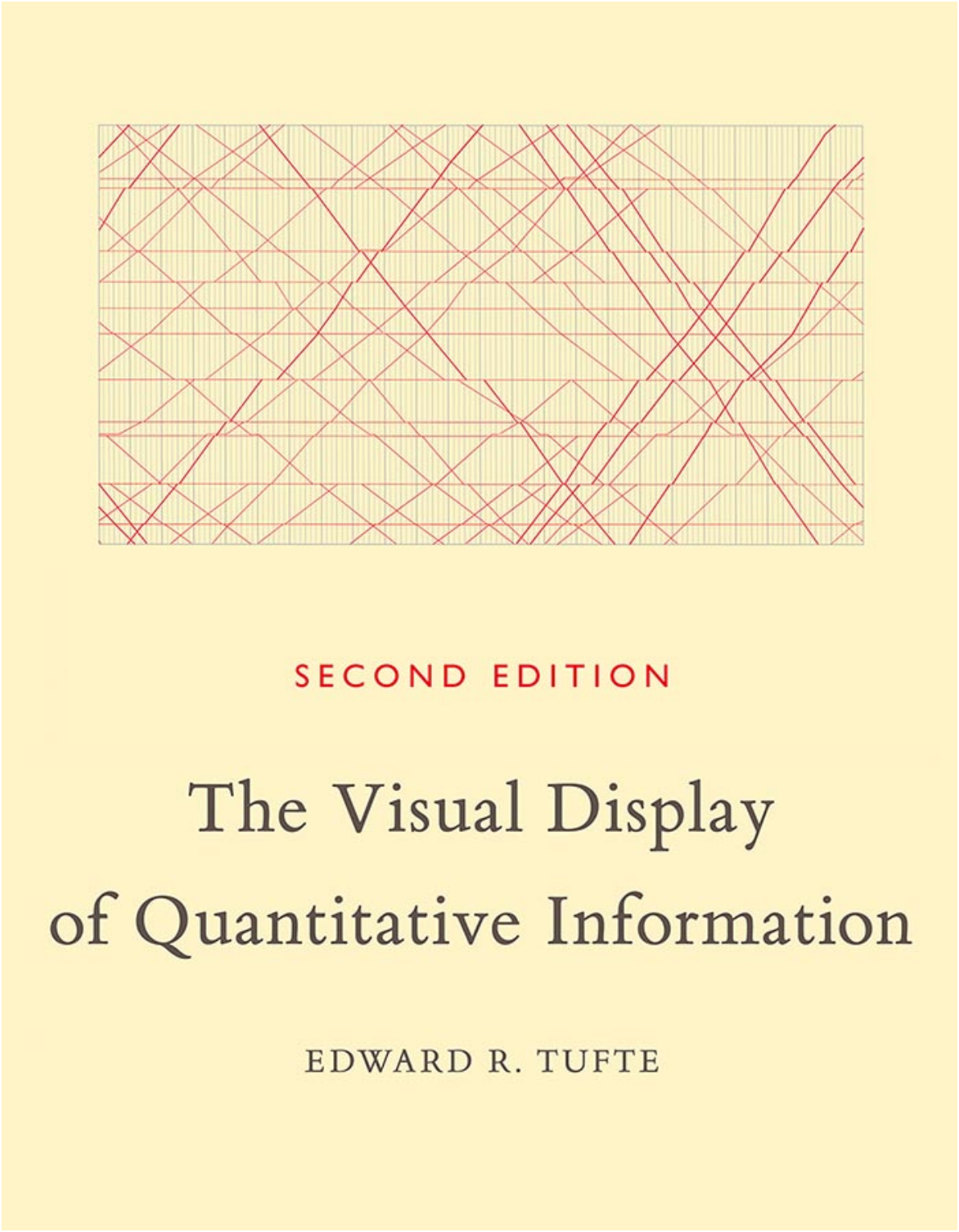



Figure 19.1: Edward Tufte's landmark book, "The Visual Display of Quantitative Information," emphasizes the importance of maximizing the data-ink ratio.

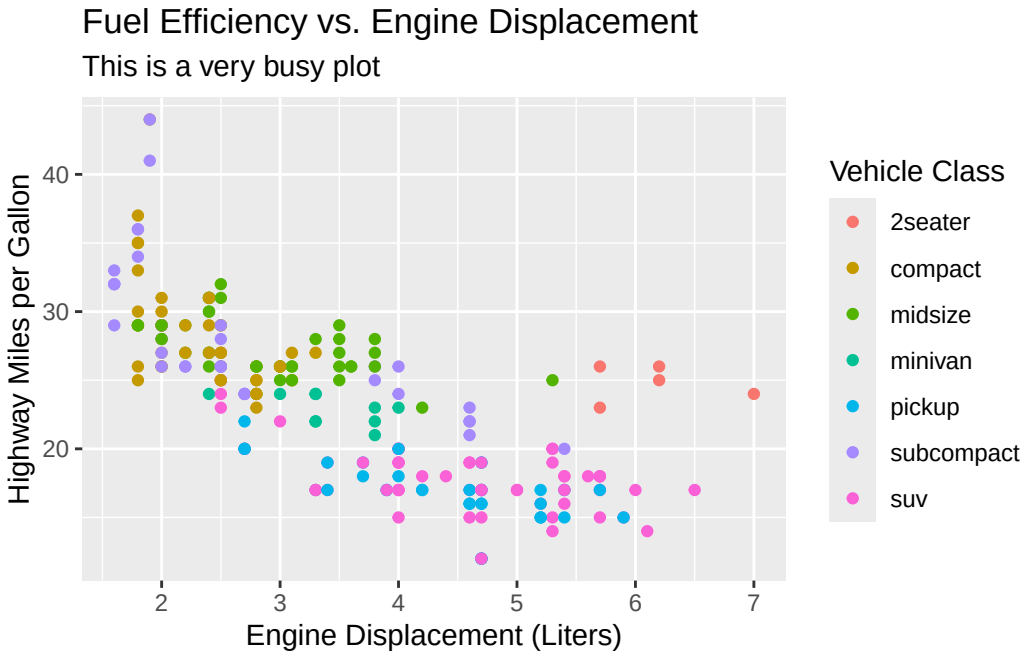



Figure 19.2: A cluttered plot with low data-ink ratio. The gray background, heavy gridlines, and legend title are all unnecessary.

Here's a cleaner version of the same plot that follows the "least ink" principle. Notice how it removes unnecessary elements while still conveying the same information.

```
ggplot(mpg, aes(x = displ, y = hwy, color = class)) +
  geom_point() +
  theme_minimal() + # A cleaner theme
  labs(title = "Fuel Efficiency vs. Engine Displacement",
       x = "Engine Displacement (Liters)",
       y = "Highway Miles per Gallon",
       color = "Class") # Simpler legend title
```

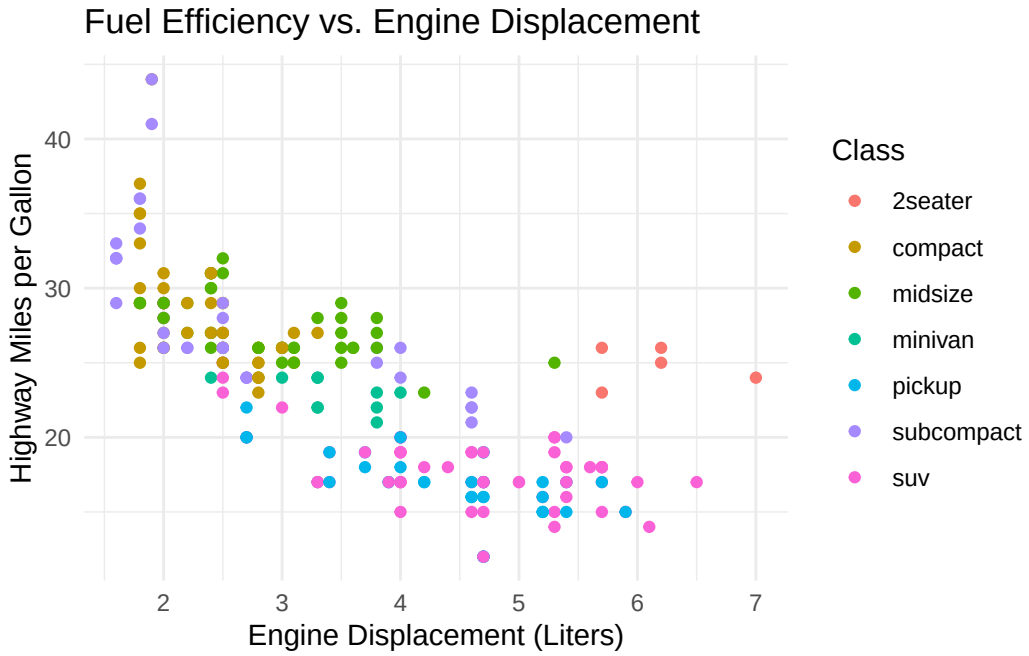



Figure 19.3: A clean, minimalist plot that follows the 'least ink' principle. The focus is entirely on the relationship between the data points.

While this is a subjective topic, the goal is to make your plots as clear and informative as possible. The "least ink" principle is a guideline, not a rule, but it can help you create more effective visualizations.

### 19.2.2 The Importance of Clear Labeling

A plot without labels is just a picture. To be a useful piece of analysis, it needs to communicate context. Always ensure your plots have:

- A clear and descriptive title.
- Labeled axes with units (*e.g.*, "Temperature (°C)").
- An informative legend if you're using color, shape, or size to encode data.

### 19.2.3 Color and Contrast

Color is a powerful tool in data visualization, but it can also be misused. R provides several built-in color palettes, and you can also use packages like `RColorBrewer` for more options. Think in terms of colorblind-friendly palettes, and avoid using too many colors in a single plot.

Color palettes can be roughly categorized into:

- Sequential palettes (first list of colors), which are suited to ordered data that progress from low to high (gradient).
- Qualitative palettes (second list of colors), which are best suited to represent nominal or categorical data. They not imply magnitude differences between groups.
- Diverging palettes (third list of colors), which put equal emphasis on mid-range critical values and extremes at both ends of the data range.

```
library(RColorBrewer)

display.brewer.all()
```

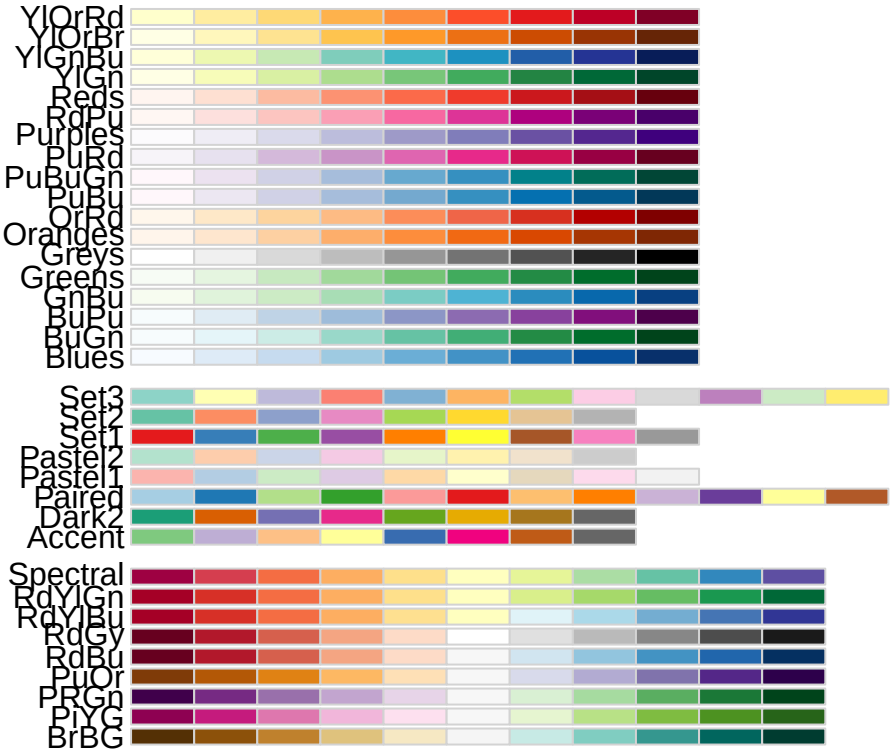



Figure 19.4: Examples of color palettes in R. The first row shows sequential palettes, the second row shows qualitative palettes, and the third row shows diverging palettes.

It is also important to consider colorblindness when choosing colors for your plots. The most common types of color blindness are red-green and blue-yellow. You can use tools like the `colorspace` package to check how your plots will look to people with different types of color vision deficiencies.

```
display.brewer.all(colorblindFriendly=TRUE)
```

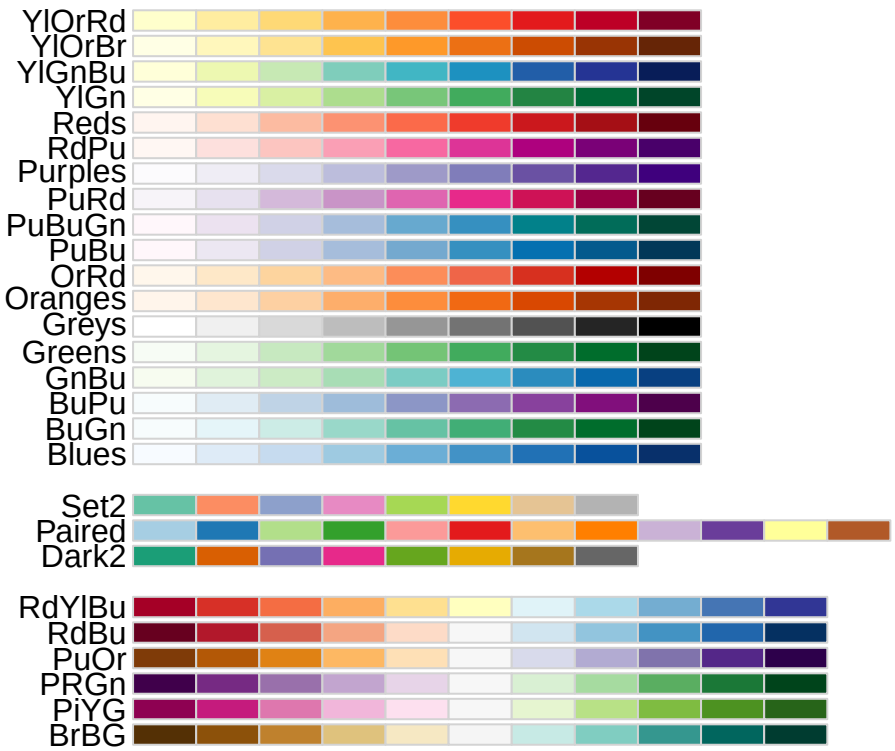



Figure 19.5: A colorblind-friendly palette from the 'colorspace' package. This palette is designed to be distinguishable for people with various types of color vision deficiencies.

## 19.3 Introduction to ggplot2: The Grammar of Graphics

Rather than reproducing excellent online resources, for this section, pick any or all of the following resources to learn about the grammar of graphics and how to use `ggplot2`:

- R for Data Science: Data Visualization
- ggplot2 documentation
- Modern Data Visualization with R

## 19.4 Sets and Intersections: UpSet Plots

When dealing with multiple sets, visualizing their intersections can be challenging. Traditional Venn diagrams become cluttered and hard to read with more than three sets. An **UpSet plot** is a powerful alternative for visualizing the intersections of multiple sets. It consists of two main parts: a matrix that shows which sets are part of an intersection, and a bar chart that shows the size of each intersection. This makes it far more scalable and easier to interpret than a complex Venn diagram.

To create an UpSet plot, we use the `UpSetR` package. It takes a specific input format where 0s and 1s indicate the absence or presence of an element in a set.

```
# install.packages("UpSetR")
library(UpSetR)
movies <- read.csv(system.file("extdata", "movies.csv", package = "UpSetR"),
    header = T, sep = ";")

# Use the 'movies' dataset that comes with UpSetR
# This dataset is already in the correct binary format
upset(movies,
      nsets = 5, # Show the 5 most frequent genres
      order.by = "freq",
      mainbar.y.label = "Intersection Size",
      sets.x.label = "Total Movies in Genre")
```

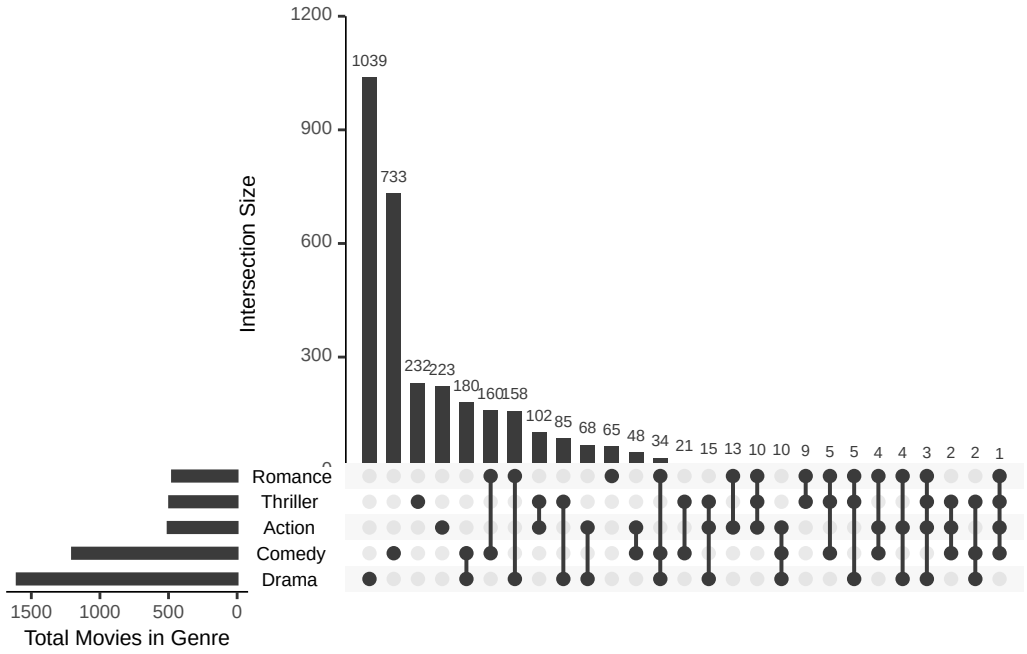



Figure 19.6: An UpSet plot visualizing movie genres. The main bar chart shows the size of intersections (e.g., how many movies are both 'Comedy' and 'Romance'). The bottom-left matrix indicates which genres are part of each intersection. This is much clearer than a 5-set Venn diagram.

The UpSet plot clearly shows us, for instance, the number of movies that are exclusively "Drama" versus those that are a combination of "Drama," "Comedy," and "Romance." This level of detail is difficult to achieve with a Venn diagram.

## 19.5 Complex Heatmaps

Heatmaps are a powerful way to visualize complex data matrices, especially when dealing with large datasets. They allow you to see patterns and relationships in the data at a glance.

What is a heatmap? It's a graphical representation of data where individual values are represented as colors. The color intensity indicates the magnitude of the value, making it easy to spot trends and outliers. The underlying data is typically a matrix of numbers.

> **Note**
>
> A matrix is a two-dimensional array of numbers, where each element is identified by its row and column indices. Matrices can include only ONE data type.

There are many ways to create heatmaps in R including the base R `heatmap()` function, the `ggplot2` package, and specialized packages like `ComplexHeatmap`.

Feel free to explore the following resources for some of the most popular heatmap packages in R:

- ggplot2 Heatmaps allows you to create basic heatmaps using the `geom_tile()` function. This is a good starting point for simple heatmaps.
- The heatmap() function in base R is a simple way to create heatmaps. It automatically scales the data and provides options for clustering rows and columns.
- The pheatmamp package is a popular package for creating heatmaps with more customization options. It allows you to add annotations, customize colors, and control clustering.
- Complex Heatmaps is a powerful R package for creating complex heatmaps. It allows you to visualize data matrices with multiple annotations, making it ideal for genomic data analysis.
- Interactive Complex Heatmaps is an extension of the Complex Heatmaps package that allows you to create interactive heatmaps. This can be useful for exploring large datasets and identifying patterns.

## 19.6 Genome and Genomic Data Visualization

The `Gviz` package is a powerful tool for visualizing genomic data in R. It allows you to create publication-quality plots of genomic features, such as gene annotations, sequence alignments, and expression data.

The GenomicDistributions package is another useful package for visualizing genomic data. It provides functions for creating distribution plots of genomic features, such as coverage, chip-seq or atac-seq distributions relative to genomic features, etc.

## 19.7 Conclusion

This document has provided a comprehensive foundation for creating effective data visualizations in R with `ggplot2`. We've covered the core principles of good design, explored a wide range of common plot types including heatmaps, and seen how `ggplot2`'s layered grammar allows for the creation of complex, insightful graphics by mapping multiple data dimensions to aesthetics. We also discussed why certain plots like Venn diagrams can be problematic and introduced powerful alternatives like UpSet plots.

The key to mastering data visualization is practice. Experiment with different datasets, try new `geoms`, and always think critically about the story your plot is telling and the best way to tell it.

# References

# A Interactive Intro to R

## A.1 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 Git and GitHub

Git is a version control system that allows you to track changes in your code and collaborate with others. GitHub is a web-based platform that hosts Git repositories, making it easy to share and collaborate on projects. Github is NOT the only place to host Git repositories, but it is the most popular and has a large community of users.

You can use git by itself locally for version control. However, if you want to collaborate with others, you will need to use a remote repository, such as GitHub. This allows you to share your code with others, track changes, and collaborate on projects.

> **Note**
>
> It can be confusing to understand the difference between Git and GitHub. In short, Git is the version control system that tracks changes in your code, while GitHub is a platform that hosts your Git repositories and provides additional features for collaboration.

## B.1 install Git and GitHub CLI

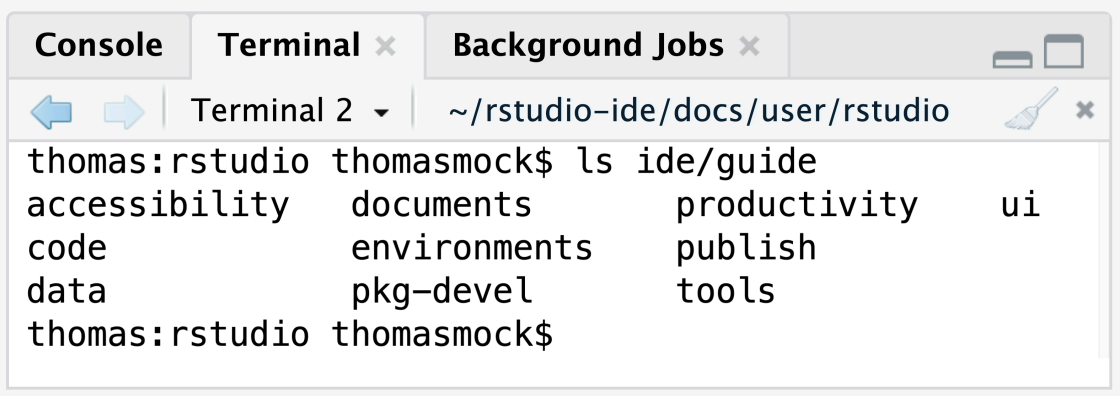
To use Git and GitHub, you need to have Git installed on your computer. You can download it from git-scm.com. After installation, you can check if Git is installed correctly by running the following command in your terminal:

```
git --version
```

We also need the `gh` command line tool to interact with GitHub. You can install it from cli.github.com. To install, go to the releases page and download the appropriate version for your operating system. For the Mac, it is the file named something like "Macos Universal" and the file will have a .pkg extension. You can install it by double-clicking the file after downloading it.

> **Using the RStudio Terminal**
>
> If you are using RStudio, you can use the built-in terminal to run Git commands. To open the terminal, go to the "Terminal" tab in the bottom pane of RStudio. This allows you to run Git commands directly from RStudio without needing to switch to a separate

terminal application.




For more details, see the RStudio terminal documentation.

## B.2 Configure Git

After installing Git, you need to configure it with your name and email address. This information will be used to identify you as the author of the commits you make. Run the following commands in your terminal, replacing "Your Name" and "you@example.com" with your actual name and email address:

```
git config --global user.name "Your Name"
git config --global user.email "you@example.com"
```

## B.3 Create a GitHub account

If you don't already have a GitHub account, you can create one for free at github.com.

## B.4 Login to GitHub CLI

After installing the GitHub CLI, you need to log in to your GitHub account. Run the following command in your terminal:

```
gh auth login
```

## B.5 Introduction to Version Control with Git

Welcome to the world of version control! Think of Git as a "save" button for your entire project, but with the ability to go back to previous saves, see exactly what you changed, and even work on different versions of your project at the same time. It's an essential tool for reproducible and collaborative research.

In this tutorial, we'll learn the absolute basics of Git using the command line directly within RStudio.

### B.5.1 Key Git Commands We'll Learn Today:

- **`git init`**: Initializes a new Git repository in your project folder. This is the first step to start tracking your files.
- **`git add`**: Tells Git which files you want to track changes for. You can think of this as putting your changes into a "staging area."
- **`git commit`**: Takes a snapshot of your staged changes. This is like creating a permanent save point with a descriptive message.
- **`git restore`**: Discards changes in your working directory. It's a way to undo modifications you haven't committed yet.
- **`git branch`**: Allows you to create separate timelines of your project. This is useful for developing new features without affecting your main work.
- **`git merge`**: Combines the changes from one branch into another.

---

## B.6 The Toy Example: An R Script

First, let's create a simple R script that we can use for our Git exercise. In RStudio, create a new R Script and save it as `data_analysis.R`.

```
# data_analysis.R

# Load necessary libraries
library(ggplot2)
library(dplyr)

# Create some sample data
data <- data.frame(
  x = 1:10,
```

```
  y = (1:10) ^ 2
)

# Initial data summary
summary(data)
```

---

## B.7 Let's Get Started with Git!

Open the **Terminal** in RStudio (you can usually find it as a tab next to the Console). We'll be typing all our Git commands here.

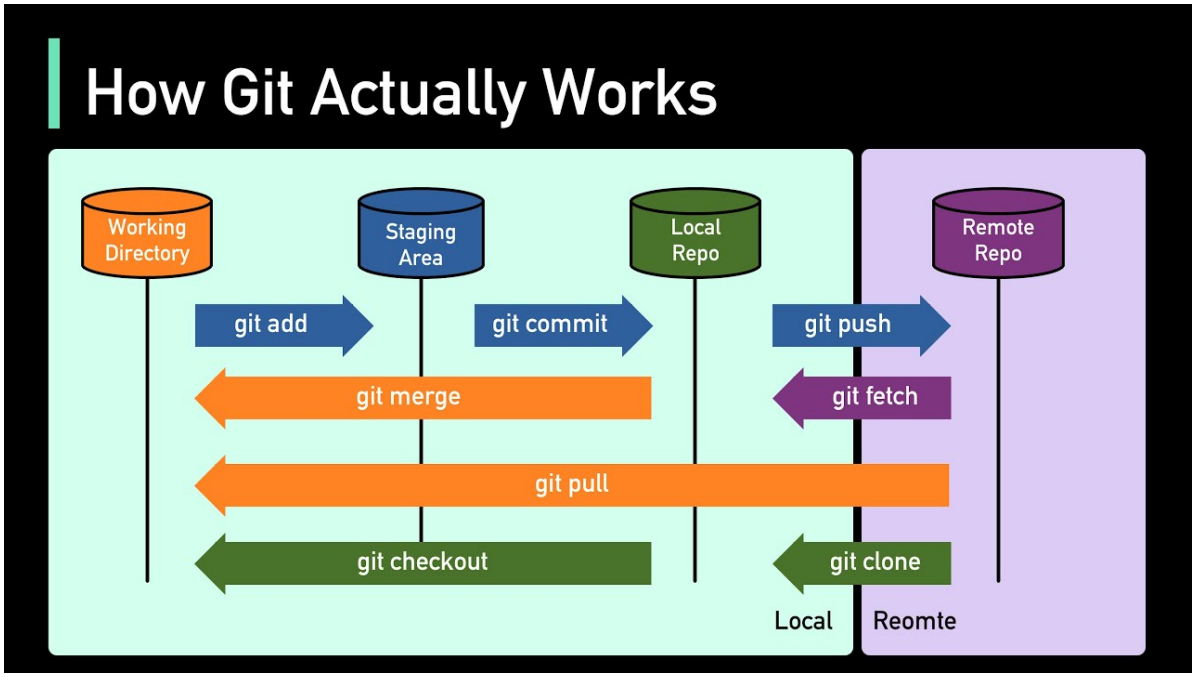



Figure B.1: This is an overview of how git works along with the commands that make it tick. See this video

### B.7.1 Step 1: Initialize Your Git Repository

First, we need to tell Git to start tracking our project folder.

```
git init
```

You'll see a message like `Initialized empty Git repository in...`. You might also notice a new `.git` folder in your project directory (it might be hidden). This is where Git stores all its tracking information. Your default branch is automatically named `main`.

### B.7.2 Step 2: Your First Commit

Now, let's add our `data_analysis.R` script to Git's tracking and make our first "commit."

1. **Add the file to the staging area:**
   ```
   git add data_analysis.R
   ```
2. **Commit the staged file with a message:**
   ```
   git commit -m "Initial commit: Add basic data script"
   ```
   The `-m` flag lets you write your commit message directly in the command. Good commit messages are short but descriptive!

### B.7.3 Step 3: Making and Undoing a Change

Let's modify our R script. Add a plotting section to the end of `data_analysis.R`.

```
# ... (keep the previous code)

# Create a plot
ggplot(data, aes(x = x, y = y)) +
  geom_point() +
  ggtitle("A Simple Scatter Plot")
```

Now, what if we decided we didn't want this change after all? We can use `git restore` to go back to our last committed version.

```
git restore data_analysis.R
```

If you look at your `data_analysis.R` file now, the plotting code will be gone!

### B.7.4 Step 4: Branching Out

Branches are a powerful feature. Let's create a new branch to add our plot without messing up our `main` branch.

1. **Create a new branch and switch to it:**

   ```
   git checkout -b add-plot
   ```

   *This is a shortcut for `git branch add-plot` and `git checkout add-plot`.*

Now, re-add the plotting code to `data_analysis.R`.

```
# ... (keep the previous code)

# Create a plot
ggplot(data, aes(x = x, y = y)) +
  geom_point() +
  ggtitle("A Simple Scatter Plot")
```

Let's commit this change on our new `add-plot` branch.

```
git add data_analysis.R
git commit -m "feat: Add scatter plot"
```

### B.7.5 Step 5: Seeing Branches in Action

Now for the magic of branches. Let's switch back to our `main` branch.

```
git checkout main
```

Now, open your `data_analysis.R` script in the RStudio editor. **The plotting code is gone!** That's because the change only exists on the `add-plot` branch. The `main` branch is exactly as we last left it.

Let's switch back to our feature branch.

```
git checkout add-plot
```

Check the `data_analysis.R` script again. **The plotting code is back!** This demonstrates how branches allow you to work on different versions of your project in isolation.

### B.7.6 Step 6: Merging Your Work

Our plot is complete and we're happy with it. It's time to merge it back into our `main` branch to incorporate the new feature.

1. **Switch back to the main branch, which is our target for the merge:**

   ```
   git checkout main
   ```

2. **Merge the `add-plot` branch into `main`:**

   ```
   git merge add-plot
   ```

You'll see a message indicating that the merge happened. Now, your `main` branch has the updated `data_analysis.R` script with the plotting code!

# C Additional resources

## C.1 R Cheatsheets and Reference material

- Base R Cheat Sheet
- Modern Data Visualization with R

## C.2 RMarkdown and Quarto

- RMarkdown Cheatsheet
- Rstudio 1 page RMarkdown Cheatsheet
- Rstudio RMarkdown Cheetsheet
- Quarto
- Quarto Books. This course material is a quarto book

## C.3 AI

- chatGPT
- Gemini
- Claude
- DeepSeek
- Perplexity

# D Data Visualization with ggplot2

Start with this worked example to get a feel for the `ggplot2` package.

- https://rkabacoff.github.io/datavis/IntroGGPLOT.html

Then, for more detail, I refer you to this excellent ggplot2 tutorial.

Finally, for more R graphics inspiration, see the R Graph Gallery.

# E Installation of Packages

---

To install packages needed to run and compile this book you can use the installation script provided.

```
source("https://raw.githubusercontent.com/lshep/RPC519RBioc/refs/heads/main/installationScri
```

This will install required packages as well as optional packages utilized in the appendix and quarto used to compile this book. To choose what packages to install you may reference this script and optionally select installs.

# F Class Notes

---

Helpful Notes from Questions during 519 Class held Oct 2025.

Thursday's class was an interactive session on Rmarkdown. This will briefly highly some of the formating and packages that were covered in class.

See also Additional Resources

### F.0.1 Random

Bold: `**`

Italics: `*`

Line Space: `---`

Inline Code: Single Backticks `` ` `` Inline Code `` ` ``

Note: If using html as output than can use html syntax as well (e.g `<br>`, `<a href = https:///>tag</a>`

### F.0.2 Code Blocks

Code Block: Backticks to open and close section: ```` ``` ````

Code Block Options: After opening backticks `{}` can define options for the code chunk. It is always a good idea to give it the programming language and an id for the code chunk for debugging `{r myid, options ...}` Common Options:

- eval
- echo
- results
- message
- warning
- figure options: fig.cap, out.width, etc

### F.0.3 Tables

- library(knitr) for kable
- library(kableExtra) for cool styling and built in themes
- library(DT) for interactive tables

### F.0.4 Lists

Ordered uses numbers

Unordered uses Dash `-`

Checklist use Dash space brackets `- [ ]` or `- [X]`

### F.0.5 Figures

- Build it with `![image caption](path to image)`
- library(knitr) include_graphics
- html `<figure>`

### F.0.6 Table of Contents

- In rmarkdown header

```
output:
  html_document:
    toc: true
```

- Special: `[Back to top](#top)`
- You can anchor any section header `# Coding Section{#codechunk}` and then reference `[link to coding](#codechunk)`
- With html `<a href="#top" style="color:red; float:right;">Back to top</a>`

### F.0.7 Plotting

- Code blocks have options for plotting
- Can include directly in code block or save to png, jpeg, etc and include elsewhere
- library(knitr) include_graphics

### F.0.8 Tabulars

```
## Results {.tabset}

### Text
Some Overview Text

### Plots
Anoter section

### tables
Anoter section
```

### F.0.9 Columns

- Can specify for the entire document through options and layouts but ad hoc with columns

```
::: {.columns}

::: {.column width="50%"}
something

:::
::: {.column width="50%"}
Second column

:::
:::
```

# G Contact

---

Lori Kern lori.shepherd@roswellpark.org RSC-400 MTHF 7:00-2:30