Basic Computing 1 — Introduction to R*

Stefano Allesina and Peter Carbonetto *University of Chicago*

The aim of this workshop is to introduce R and RStudio, and show how it can be used to analyze data in an automated, replicable way. We will illustrate the notion of assignment and present the main data structures available in R. We will learn how to read, inspect, manipulate, analyze and write data, and how to execute simple programs. This workshop is intended for biologists with little to no background in programming.

About this tutorial

We will work through the tutorial examples all together as a class, as well as in smaller groups. Feel to free to try the examples on your own, before or after the in-class tutorial. To run the examples before the data have been collected in class, you may use the data from previous years, stored in files `laptop_2019.csv` and `laptop_2020.csv`. To use these data, rename one of them as `laptops.csv`.

Setup

To complete the tutorial, you will need to install R and RStudio. To install R, follow instructions at `cran.rstudio.com`. Then install Rstudio following the instructions at `bit.ly/2JARd4v`.

Before starting up RStudio, it is helpful to quit applications that are not needed, and other "clutter", to reduce distractions.

Launch RStudio. It is best if you start with a fresh workspace; you can refresh your environment by selecting **Session > Clear Workspace** from the RStudio menu. Also, make sure your R working directory is the same directory containing the tutorial materials; you can run `getwd()` and `list.files()` to check this.

Motivation

When it comes to analyzing data, there are two competing paradigms. One could use point-and-click software with a graphical user interface, such as Excel, to perform calculations and draw graphs; or one could write programs that can be run to perform the analysis of the data, the generation of tables and statistics, and the production of figures automatically.

This latter approach is preferred because it allows for the automation of analysis, it requires a good documentation of the procedures, and is completely replicable.

A few motivating examples:

*This document is included as part of the Basic Computing 1—Introduction to R tutorial packet for the BSD qBio Bootcamp, University of Chicago, 2021. **Current version**: August 13, 2021; **Corresponding author**: pcarbo@uchicago.edu. Thanks to John Novembre, Stephanie Palmer and Matthew Stephens for their guidance.

- You have written code to analyze your data. You receive from your collaborators a new batch of data. With simple modifications of your code, you can update your results, tables and figures automatically.
- A new student joins the laboratory. The new student can read the code and understand the analysis without the need of a labmate showing the procedure step-by-step.
- The reviewers of your manuscript ask you to slightly alter the analysis. Rather than having to start over, you can modify a few lines of code and satisfy the reviewers.

Here we introduce R, which can help you write simple programs to analyze your data, perform statistical analysis, and draw beautiful figures.

What is R?

R is a statistical software that is completely programmable. This means that one can write a program ("script") containing a series of commands for the analysis of data, and execute them automatically. This approach is especially good as it makes the analysis of data well-documented, and completely replicable.

R is free software: anyone can download its source code, modify it, and improve it. The R community of users is vast and very active. In particular, scientists have enthusiastically embraced the program, creating thousands of packages to perform specific types of analysis, and adding many new capabilities. See www.r-pkg.org for a listing of official packages (which have been vetted by R core developers); many more are available on Bioconductor, GitHub and other websites.

The main hurdle new users face when approaching R is that it is based on a command-line interface; when you launch R or RStudio, you open a console with the character > signaling that R is ready to accept an input. When you write a command and press Enter, the command is interpreted by R, and the result is printed immediately after the command. For example, this calculates the sum $1 + 2 + \cdots + 99$ and prints the result:

```
sum(1:99)
# [1] 4950
```

A little history

R was modeled after the commercial statistical software S by Robert Gentleman and Ross Ihaka. The project was started in 1992, first released in 1994, and the first stable version appeared in 2000. Today, R is managed by the *R Core Team*.

A "data-centric" view of programming

For those of you who already have experience with a programming language—interactive (e.g., MATLAB, Python) or otherwise (e.g., C++, Java)—you will find that there is much to learn and discover in R. One distinguishing feature of R is that it promotes a *data-centric* view of programming practice; that is, it is not the proedures (actions) that are the primary focus, but instead the *data structures* (objects). This is not accidental—analysis of data is central to R. Appreciating this emphasis on data can help you navigate R, particularly if you are more familiar with procedural programming langauges such as C++.

RStudio

For this introduction, we're going to use RStudio, an Integrated Development Environment (IDE) for R. One advantage of RStudio is that the environment will look identical irrespective of your computer architecture (Linux, Windows, Mac). Also, RStudio makes writing code much easier by automatically completing commands and file names (simply type the beginning of the name and press Tab), and allowing you to easily inspect data and code.

The main RStudio interface is split up into "panels". The most important panels are:

1. **Console:** This is a panel containing an instance of R. For this tutorial, we will work mainly in this panel.
2. **Source:** In this panel, you can write a program and save it to a file. The code and also be run from this panel, but the actual results show up in the Console.
3. **Environment:** This panel lists all the variables you created (more on this later).
4. **History:** This gives you the history of the commands you typed.
5. **Plots:** This panel shows you all the plots you drew. Other tabs allow you to access the list of packages you have loaded, and the help page for commands (e.g., type `help(p.adjust)` in the Console).

What is an R "program"?

An R program is simply a list of commands, which are executed one after the other (for this reason, it is sometimes called a "script"). The commands are written in a text file (with extension .R). When R executes the program, it will start from the beginning of the file and proceed toward the end of the file. Every time R encounters a command, it will execute it. Special commands can modify this basic flow of the program by, for example, executing a series of commands only when a condition is met, or repeating the execution of a series of commands multiple times.

Note that if you were to copy and paste (or type) the code into the **Console** you would obtain exactly the same result. The advantage of a program, or "script", is that it allows for automating the computations; that is, it eliminates the manual copying and pasting. Moreover, after a while you will have accumulated a lot of code, at which point you can reuse much of your code for new projects.

Tutorial outline

In this tutorial, we will learn about the basic elements of R through analysis of a small data set. This will involve:

1. Collecting the data.
2. Deciding on some analysis goals.
3. Importing the data into R.

4. Inspecting the data.
5. Manipulating the data.
6. Automating the analysis.

In-class activity: Collaborative data collection

We begin this tutorial by collecting the data we will analyze. Specifically, we will collect data about the laptops (or desktops) of incoming BSD graduate students. We will use these data to learn about R, and how it can be used to analyze a data set; we will write some R code that can be used to answer basic questions such as, what is the most common operating system among the population (where "population" is all the current incoming BSD grad students)?

We will collect the data in Etherpad. Specifically, we will collect four data points: first name, operating system, amount of memory (in GB), and number of processors ("cpus"). We will enter these data into a table with four columns. We will use commas to separate the columns in the table. The first few lines of the table will look something like this:

```
name,os,mem,cpus
rishi,mac,8,4
anna,windows,4,1
peter,mac,16,4
```

If you are unsure how to enter some of this information, ask your teammates or peers for help.

Once we have collected the data from the entire class, we will create a new file in `RStudio`, copy & paste the data (including table header) into the new file, and save the file as `laptops.csv`.

In-class activity: Formulate analysis aims

Before writing any R code to explore the data, it is helpful to have a focus to the exploration. Suppose you are in charge of designing the QBio programamaing tutorials for next year's incoming BSD graduate cohort. What information about the students laptops would help in designing the course? Here are some basic questions we might want to answer:

- What are the different operating systems used?
- Which operating system is used most?
- Are all the laptops multicore (more than one processor)?

Write down a few more guiding questions in the space below. Try to state your questions using plain (non-technical) language, avoiding statistical terms such as "mean", "correlation" and "significant".

Import the data into R

You should now have a CSV file on your computer, `laptops.csv`, containing the data we collected. (CSV is an abbreviation of "comma-separated values".) The standard `R` function for importing data from a CSV file is `read.csv`:

```
read.csv("laptops.csv", comment = "#", stringsAsFactors = FALSE)
```

In time, we will understand what this code is doing. For now, we will jump ahead.

This command will only work if your `R` working directory is the same as the directory containing `laptops.csv`. You can check your working directory with `getwd()` and `list.files()`. If you are in the wrong directory, use **Session > Set Working Directory** from the `RStudio` menu bar to move to the right directory.

This line of code, on its own, is a dead-end; all it does is print the result to the screen. What is missing from this expression is *assignment of the output to an object.* A common and frustrating mistake—even experienced programmers make this mistake—is forgetting to assign the output.

```
laptops <- read.csv("laptops.csv", comment = "#", stringsAsFactors = FALSE)
```

A good name for the new object helps remind you of its contents (and it is particularly important when you have created many objects).

Assuming you started with a clean workspace, your workspace, or "environment", should now contain a single object, `laptops`:

```
ls()
```

This object stores the output of our `read.csv` call. Invoking its name will print its contents:

```
laptops
```

Or you can explicitly type

```
print(laptops)
```

which does the exact same thing.

What *kind* of object is it?

```
class(laptops)
```

It is a "data frame". This is R's term for "table" or "spreadsheet". (You will find that R has a habit of renaming things you are already familiar with.)

If you are used to the "point-and-click" way of doing things in software such as Excel, it may seem burdensome to have arrived at this point where we have written a bunch of code, and all we have done is printed the contents of the table to the screen. But now that we have created a data frame, laptops, there are many powerful things we can do it. Also bear in mind that the advantages of the programmatic approach to data analysis are less obvious when working with small data sets, but the techniques we use here also work with large data sets that are not so easily analyzed by point-and-click (e.g., tables containing millions of rows).

Inspect the data

R has many commands that are easy to use and quickly give you insight into the data. Let's try some of the more commonly used commands for inspecting and summarizing the contents of a data frame. You might want to add a note next to each line of code to remind yourself what it does:

```
nrow(laptops)
ncol(laptops)
head(laptops)
tail(laptops)
str(laptops)
summary(laptops)
```

Some of these commands, like head, str and summary, are "generic" functions, meaning that they work for many object types, from basic types to complex data structures. Generic functions such as these are routinely used, and they will likely become part of your go-to data analysis toolkit.

Data subviews

This data set is small enough that you can easily inspect all of it by eye. However, when you are working with a much larger data set you need a strategy to inspect manageable subsets of the data.

Each of the examples below print a subset of the data. As before, add a note next to each line of code to remind yourself what it does.

```
laptops[ ,2]
laptops[[2]]
laptops[ ,"os"]
laptops["os"]
laptops$os
laptops[4, ]
laptops[4, 2]
laptops[4, "os"]
laptops$os[4]
laptops[4, ]$os
```

Here are a few slightly more complex examples:

```
laptops[1:4, ]
laptops[ ,2:3]
laptops[ ,c("os", "mem")]
laptops[c("os", "mem")]
```

Once you are comfortable with the basic elements of selecting subsets, you can combine these elements in an endless variety of ways, e.g.,

```
laptops[c(1:3, 5:7), c("name", "os")]
```

Creating new data sets from subviews

The result of almost any calculation in R can be saved to an object. This includes selecting subsets. For example,

```
x <- laptops[1:10, ]
print(x)
class(x)
```

creates a new table (a data frame) from the first 10 rows of full `laptops`. There should now be two data frames in your environment:

```
ls()
```

Objects can also be *overwritten*:

```
x <- laptops[1:5, ]
```

The 10-row table you created is now gone.

Observe that both assignment (creating new objects) and overwriting both use `<-`. So be careful—*there is no undo command in R!*

This ability to have multiple data sets floating around in your environment underscores the importance of naming your objects well, and keep track of what is in your environment; names such as "x" or "temp" or "dat", while commonly used, should only be used for temporary or "throw-away" calculations. It isn't unusual to have half a dozen different copies of a data set over the course of an analysis. This is where the `rm` function also comes in handy for cleaning up your workspace. Another important practice is to document your code in case you do accidentally overwrite something important!

Conditional subviews

One powerful way to inspect subsets is by condition. For example, to view all the laptops with exactly two processors, do

```r
subset(laptops, cpus == 2)
```

There are at least a couple other ways to achieve the same thing:

```r
laptops[laptops$cpus == 2, ]
```

and

```r
rows <- which(laptops$cpus == 2)
laptops[rows, ]
```

The last approach is interesting because it involves creating a new *numeric* object containing the numbers of the rows we are interested in.

We will learn more about logical expressions below.

Question: Which way is best?

Exercise: How would you select the subset of laptops that have Windows?

A side note: "`<-`" versus "`=`"

Let's take a moment to discuss a common point of confusion. In `R`, the equality symbol (`=`) can also be used for creating and overwriting objects. Many people prefer to use `=`, but we recommend against using it because:

1. The `=` is easily confused with `==`. (How are `=` and `==` different?)
2. The `=` is also used for named arguments to functions (see the `read.csv` call above for an example). This is not assignment—no objects are created or overwritten. (Some people do incorrectly all this "assignment"—feel free to correct them.)

Deconstructing the "laptops" data frame

We will continue to explore the laptops data frame. A data frame is an example of a *composite* data structure—it is made of simpler data objects. These simpler ("atomic") data objects are `R`'s "building blocks" for all other data structures. In a data frame, the *columns* are the atomic data objects; in the next sections, we will take a close look at the columns of the `laptops` data frame.

Text data

Let's begin with the "os" column:

```
x <- laptops$os
print(x)
```

In `R`, the `character` type is used to store text data:

```
class(x)
```

You can access individual elements by their index: the first element is indexed at 1, the second at 2, *etc.*

```
x[1]
x[2]
```

`R` has many built-in functions for operating on text data. Here are some examples (add notes explaining what these lines of code do):

```
nchar(x)
toupper(x)
sort(x)
unique(x)
table(x)
paste(x[1], x[2], x[3])
```

If you want to learn more about a particular function, `R` has extensive documentation that can accessed directly from the Console (no need to Google it). For example, to learn more about the `sort` function, type

```
help(sort)
```

From this help page, you will learn, among other things, that you can control the order of the sorting using the `decreasing` option, and by default entries are sorted in increasing order.

Just as data types can be combined to form more complex data structures, operations can also be combined (provided the combination makes sense, of course!). For example,

```
unique(toupper(x))
```

is effectively equivalent to

```
y <- toupper(x)
unique(y)
```

Numeric data

The "mem" column is an example of numeric data:

```
x <- laptops$mem
class(x)
```

Specifically, it is an integer type since all the numbers are whole numbers. There is another data type, `numeric`, used to store real numbers (or rather their approximations, as computers have limited memory and thus cannot store numbers like π, or even 0.2), e.g.,

```
y <- x / 100
class(y)
```

Very small or very large numbers can be represented in `R` (any idea what the ^ operator is doing?):

```
x^10
```

Like the `character` data type, a numeric object is a vector, and individual elements can be accessed their index,

```
x[1]
x[2]
```

Also, like the `character` data type, `R` has many built-in functions for working with numbers, such as

```
abs(x)
sqrt(x)
cos(x)
log(x)
```

Notice that, for example, `log(x)` computes the logarithm for *each data point in* `x`, and the result is *a vector of the same length as* `x`. This is idea of automatically applying an operation to each entry of a vector is sometimes called "vectorization". In `R`, vectorization is quite natural because vectors are one of the elemental data structures. Vectorization allows complex operations to be accomplished very simply; for example, if `x` contains 10 million numbers, `y <- sqrt(x)` will compute the square root of these 10 million numbers, and store the result in vector `y`.

Some functions are more versatile, and can work with different data types:

```
sort(laptops$mem)
sort(laptops$os)
table(laptops$mem)
table(laptops$os)
```

Given that R was born for statistics, there are many statistical operations—from basic to sophisticated—that you can perform on numeric data (add notes next to these lines explaining what they do):

```
min(x)
max(x)
range(x)
sum(x)
median(x)
quantile(x, 0.5)
mean(x)
summary(x)
```

Finally, standard mathematical operations such as + (addition), - (subtraction), * (multiplication), / (division), and ^ (exponentiation) can be applied to numbers. A less familiar operator is the modulo (%%), which gives the remainder from integer division:

```
x %% 3
```

For more complex calculations, the mathematical and statistical operations can be combined together, for example,

```
y <- log(x / mean(x))
```

Exercise: Combine some of the functions and/or operators mentioned above to compute the standard deviation of x. Compare your answer against R's built-in solution, y <- sd(x). *Hint:* Use the definition of variance. Write your answer in the box below.

Logical data

The laptops data set does not contain logical data. But we can *generate* logical data from one of the columns:

```
x <- laptops$os == "mac"
print(x)
class(x)
```

The logical data type takes only two values, `TRUE` and `FALSE`.

Exercises:

1. Besides equality, other comparison operators include > (greater than), < (less than), `!=` (differs) and >= and <=. Write an expression using one or more of these operators to create some new logical data.

2. You can also formulate more complicated logical statements (perhaps using multiple variables or columns of a data frame) using `&` (and), `|` (or), and `!` (not). Write an expression using these operators, as well as the ones above, to create some new logical data.

Manipulating data

Up until now, we have treated the laptops data frame as if it were a static object. But like almost any other object in `R`, a data frame can be modified and overwritten. This is very powerful but also obviously dangerous! Here we will illustrate a few of the many kinds of data manipulations we can make, building on the elements we learned above. But before doing so, it is good practice to create a copy of the data frame in case we make a mistake along the way:

```
laptops2 <- laptops
```

Note that `laptops` and `laptops2` are two *entirely independent* copies of the data. Although they are identical at first, as soon you a make change to `laptop2`, you will have two different data sets. Again, when you have multiple versions of a data set present in your environment, it is your responsibility to keep your environment organized.

You can overwrite individual entries of the data frame, or several entries at once:

```
laptops2[c(1, 2), "mem"] <- 2
```

Using a conditional subview, we can replace all instances of "mac" values in the "os" column with "macOS":

```
rows <- which(laptops2$os == "mac")
laptops2[rows, "os"] <- "macOS"
```

If you don't like the column names, you can also change them, too:

```
colnames(laptops2) <- c("name", "OS", "GB", "CPUs")
```

You can reorder the rows or columns:

```
rows      <- order(laptops2$name)
laptops2 <- laptops2[rows, ]
laptops2 <- laptops2[c("name", "CPUs", "GB", "OS")]
```

Or you can even create new columns:

```
n <- nrow(laptops2)
laptops2$id <- seq(1, n)
laptops2$MB <- 1000 * laptops2$GB
```

Building a data frame

Above, we deconstructed the `laptops` data frame, and we found that the constituent parts are the table columns (data vectors). We also go in the reverse direction—we can build a data frame by joining together data vectors. Here's a small illustration of this:

```
x <- laptops$os
y <- laptops$cpu
```

Given text data `x` and numeric data `y`, construct a new data frame:

```
dat <- data.frame(os = x, cpu = y)
```

In-class activity: what is a "factor"?

So far, we have gotten acquainted with three basic data types: `character`, `numeric` and `logical`. Here we introduce a fourth: the `factor`. Factors are vectors, like the other atomic data types we have seen. What is unusual about factors is that there is no equivalent of factors in other popular programming languages, at least not as a native data type. Here we will explore factors and understand how and when they could be useful.

None of the columns in our `laptops` table are a factor, but we can create a factor from one of the columns using a function appropriately called "factor". Let's try this with the "mem" column:

```
x <- laptops$mem
y <- factor(x)
class(x)
class(y)
```

Let's now inspect the contents of `x` and `y`:

```
print(x)
print(y)
summary(x)
summary(y)
```

How should we interpret the second data summary?

Let's try the same with the "os" column:

```
x <- laptops$os
y <- factor(x)
class(x)
class(y)
summary(x)
summary(y)
```

Question: Based on the summaries of the two factors, what do you think a factor is?

More questions: Which representation to do you prefer for "mem", numeric or factor? What about the "os" column? Are other columns good candidates for being factors?

These questions—should I convert my data to a factor—touch on the broader question of *data representation* or *data encoding*. Choosing the right representation of your data can be important to your analysis. Judicious use of factors can also help you perform complex calculations very simply. (In fact you may find that factors are useful in the programming challenge.)

If you prefer the "os" column to be a factor, for example, you can change it inside the data frame by doing

```
laptops$os <- factor(laptops$os)
```

Save your work

Since are nearing the end of the tutorial, so this is a good point to save our work. To save your results, go to **Session > Save Workspace As** in `RStudio`, or run

```
save.image("basic_computing_1.RData")
```

Later, to restore your environment, select **Session > Load Workspace** in RStudio, or run

```
load("basic_computing_1.RData")
```

Question: What is the difference between **File > Save As** and **Session > Save Workspace As**?

Group activity: Analysis of laptops data

In this activity, your team will pick two or three of the guiding questions that you and others have formulated, and use R to answer these questions.

Before writing any code, discuss with your team what steps you will take to answer a question. Points of discussion may include: (1) What column(s) will you need? (2) What calculations or operations will you need to perform? (3) Will you need to perform your calculations on all rows, or a subset of the rows?

For each question, record your answer, as well as the code you wrote to generate the answer.

Use this code to create a *script* or a *program*—a text file containing the code necessary to run the data analysis, starting with reading the data into R, and ending with the calculations that produce the answers.

Once you have written your script, verify that it runs, and save the file somewhere on your computer. Give your script a memorable name such as `analyze_laptops_data.R`. Also, a few comments (these are lines starting with "`#`") explaining in plain language what the code does. We will use Etherpad to share our programs.

Programming Challenge

Instructions

Work with your team to solve the following exercise. When you have found the solution, go to the `jnovembre.github.io/BSD-QBio7` and follow the link "Submit solution to challenge 1" to submit your answer.

At the end of the boot camp, the group with the highest number of correct solutions will be declared the winner. If you need extra time, you can work with your group during free time or in the breaks in the schedule.

Collaboration strategy

Before diving into the problems, first agree on a collaboration strategy with your teammates. Important aspects include communication and co-ordination practices, and setting goals and deadlines. How will your team collaborate on code, and share solutions? (Consider online resources such as Etherpad or the UofC-hosted Google Drive.) The aim is not just to complete the challenges, but also to do collaboratively; all team members should be included, and should have the opportunity to contribute and learn from each other.

Nobel nominations

The file `nobel_nominations.csv` contains data on Nobel Prize nominations from 1901 to 1964. There are three columns (the file has no "header"): (1) the field (e.g., "Phy" for physics), (2) the year of the nomination, and (3) the id and name of the nominee.

1. Take Chemistry (`Che`). Who received the most nominations?
2. Find all researchers who received nominations in more than one field.
3. Take Physics (`Phy`). Which year had the largest number of nominees?
4. For each field, what is the average number of nominees per year? Calculate the number of nominees in each field and year, and take the average across all years.

Hints

- You will need to subset the data. To make your calculations more clear, it may be helpful to give names to the columns. For example, suppose you imported the data into a data frame called `nobel`. Then `colnames(nobel) <- c("field", "year", "nominee")` should do the trick.
- The simplest way to obtain a count from a vector is to use the `table` function. For example, the command `sort(table(x))` produces a table of the occurrences in `x`, in which the counts are sorted from smallest to largest.
- You can also use the same function, `table`, to build a table using more than one vector. For example, suppose `x` and `y` are vectors of the same length. Then `table(x,y)` will build a table with counts for each unique pair of occurrences in `x` and `y`.
- Some other functions you may find useful for the challenge: `colMeans`, `factor`, `head`, `length`, `max`, `read.csv`, `subset`, `tail`, `tapply`, `unique`, `which` and `which.max`.
- Save your solution code for each exercise in a file.

Additional topics

Creating data

Above, we imported data into `R`. In addition to viewing and manipulating existing data, `R` also has many facilities for creating data structures, either from scratch, or from existing data. (We already saw some examples of this.)

The most basic tool is the `c` function, short for "combine". It can combine multiple objects or values:

```
x      <- c(2, 3, 5, 27, 31, 13, 17, 19)
sex    <- c("M", "M", "F", "M", "F")           # Sex of Drosophila
weight <- c(0.230, 0.281, 0.228, 0.260, 0.231) # Weight (in mg)
```

You can generate sequences of numbers using `seq`. For example, this generates all odd numbers from 1 to 100:

```
x <- seq(from = 1, to = 100, by = 2)
```

For simpler number sequences, use the colon operator:

```
x <- 1:10
```

To repeat a value (or values) several times, use `rep`:

```
x <- rep("treated", 5)  # Treatment status
x <- rep(c(1, 2, 3), 4)
```

Finally, there are many functions for generating random numbers. For example,

```
x <- runif(100)
```

Question: What is the result of running `runif(100)`?

Matrices

A matrix is like a data frame—it also has rows and columns. A key difference is that all the columns must be of the same type. Here's an example of a 2 x 2 matrix:

```
A <- matrix(c(1, 2, 3, 4), 2, 2)  # Inputs are values, nrows, ncols.
```

In the case of numeric values, you can perform the usual operations on matrices (product, inverse, decomposition, *etc*).

```
A %*% A        # Matrix product
solve(A)       # Matrix inverse
t(A)           # Transpose
A %*% solve(A) # This should return the identity matrix.
```

Determine the dimensions of a matrix:

```
nrow(A)
ncol(A)
dim(A)
```

Use indices to access a particular row or column of a matrix:

```
Z <- matrix(1:9, 3, 3)
Z[1, ]              # First row.
Z[ ,2]              # Second column.
Z[1:2, 2:3]         # Submatrix with coefficients in rows 1 & 2, and columns 2 & 3.
Z[c(1, 3), c(1, 3)] # Indexing non-adjacent rows and columns.
```

Some operations apply to all elements of the matrix:

```
sum(Z)
mean(Z)
```

Question: When is it better to store data in a matrix instead of a data frame?

Arrays

If you need tables with more than two dimensions, use arrays:

```
A <- array(1:24, c(4, 3, 2))
```

A matrix is a special case of an array with two dimensions.

You can still determine the dimensions with `dim`:

```
dim(A)
```

And you can access the elements as for matrices. One thing to be careful about: `R` drops dimensions that are not needed. So, if you access a "slice" of a 3-dimensional array, you should obtain a matrix:

```
A[ ,2, ]
dim(A[ ,2, ])
class(A[ ,2, ])
```

Lists

You have already worked with a list object (perhaps without realizing it). The laptops data frame is also a list:

```
is.list(laptops)
```

A data frame is a special kind of list—it is a list in which every list item is a vector of the same length. Lists also allow vectors of different lengths, for example, here we add an additional item ("date") that has a length of 1:

```
mylist <- as.list(laptops)
mylist$last_modified <- "September 9, 2019"
```

We could not do this in a data frame.

A list is a very general and very widely used `R` data structure for storing complex data.

Missing data

Finally, `R` allows most types of data—text, numeric, factors, *etc*—to take on a special value, `NA`. This is short for "not available" or "not assigned", and is commonly called "missing data". One special feature of `R` is that it often gracefully handles data sets containing missing data.

Exercise: Set a few entries in the `laptops` data frame to the value `NA`; for example, to set the first row and column to `NA` you could do `laptops[1,1] <- NA`. (You might want to make these changes to a copy of the data frame instead.) Then run `summary(laptops)`. How are the missing data reported in the summary?

Optional group activity: Analyzing genetic data

In this activity, you will apply the tools we have developed so far to look at an example data set from human genetics: genotype data on chromosome 6 collected from European individuals. (Data adapted from the Human Genome Diversity Project by John Novembre.)

```
chr6 <- read.table("H938_Euro_chr6.geno", header = TRUE)
```

Setting `header = TRUE` means that we want to take the first line to be a header containing the column names. How big is this table?

```
nrow(chr6)
ncol(chr6)
```

It contains 7 columns and over 40,000 rows!

The table reports the number of homozygotes (`nA1A1`, `nA2A2`) and heterozygotes (`nA1A2`), for 43,141 single nucleotide polymorphisms (SNPs) obtained by sequencing European individuals. The other columns are:

- `CHR`: The chromosome (in this case, 6).
- `SNP`: The identifier of the Single Nucleotide Polymorphism.
- `A1`: One of the observed alleles.
- `A2`: The other allele.

Write `R` code to answer the following questions:

1. How many individuals were sampled? Find the maximum of the sum `nA1A1 + nA1A2 + nA2A2`. *Hint:* Recall that you can access the columns by index (e.g., `chr6[ ,5]`), or by name (e.g., `chr6$nA1A1`, or `chr6[ ,"nA1A1"]`).
2. Use the `rowSums` function to obtain the same answer.
3. How many SNPs have no heterozygotes (*i.e.*, no "A1A2")?
4. How many SNPs have less than 1% heterozygotes?