# Introduction to R

Luce Skrabanek, Jason Banfelder
Paul Zumbo
Applied Bioinformatics Core

July, 2023

# I Prologue

## i. What is R?

R is a free software environment for statistical computing and graphics (`www.r-project.org`). It can effectively analyze large-scale datasets, such as those resulting from high-throughput sequencing experiments. It promotes automated and reproducible analyses of scientific data, creates a wide spectrum of publication quality figures, and has an extensive library of add-on packages to facilitate many complex statistical analyses. Because it is free and ubiquitously available (it runs on Windows, Mac, and Linux computers), your investment in learning R will pay dividends for years to come.

## ii. What is RStudio?

While R is very powerful, it is essentially a command line program and is thus not the friendliest thing to use. Especially when learning R, a friendlier environment is helpful, and RStudio provides this, giving you things you expect in a modern interface like integrated file editing, syntax highlighting, code completion, smart indentation, tools to manage plots, browse files and directories, visualize object structures, etc.

From your computer, choose the RStudio application. This will start R under the hood for you.

# II Introduction

## i. R is a calculator

1. The Console panel (lower left panel) is where you type commands to be run immediately. When R is waiting for a new command, you will see a prompt character, `>`.
2. To add two numbers, type after the prompt:

   ```
   1 + 2
   ```

   When you hit return, you should see ...

   ```
   [1] 3
   ```

3. The answer is of course 3, but what is the `[1]` before it? It turns out that all numbers in R are vectors, and the `[1]` is a hint about how the vector is indexed. To see a long vector of random numbers, type:

   ```
   rnorm(100)
   ```

   For now we can ignore the vector indexing; we will learn more about vectors and indexing shortly.

4. R understands basic math. Try typing:

   ```
   3 - 4
   5 * 6
   7 / 8
   ```

5. The order of operations is kept (PEMDAS). Note the difference between ...

   ```
   1 + 2 * 3
   ```

   and ...

   ```
   (1 + 2) * 3
   ```

6. You can force R to do integer division using the `%/%` operator (division symbol inside two percent signs):

   ```
   17 %/% 4
   ```

   and to get the remainder (i.e., modulo):

   ```
   17 %% 4
   ```

7. You can also compute powers:

   ```
   2 ^ 4
   ```

   even with fractional exponents.

   ```
   2 ^ 4.3
   ```

8. R comes with an extensive library of built-in functions.

   ```
   log(4)       # natural log
   log10(4)     # log in base 10
   log(4, 10)   # same as above
   sqrt(9)      # square root
   abs(3-4)     # absolute value
   exp(1)       # exponential
   ```

9. Note in the examples above, we have used comments (preceded by the `#` character). You can type them if you want but they do not add anything to the work that R does. Comments are not usually used when interactively typing commands into the Console, but are essential when writing scripts - stay tuned!

## ii. R has variables

1. It can be really useful to assign values to variables, so they can be referred to later. This is done using the assignment operator (`<-`).

```
us_population <- 3.31e8 # From Wolfram|Alpha, 2020 estimate
us_area <- 3719000      # From Wolfram|Alpha
us_pop_density <- us_population / us_area
us_pop_density
( us_pop_density <- us_population / us_area )
```

   Some notes:

   (a) Once a variable is defined, you will see it show up in the environment panel in RStudio.

   (b) R will not automatically print out the value of an assigned variable. Type the name of the variable by itself to see it. Alternatively, wrapping the assignment in parentheses executes the assignment and prints the result.

   (c) Case matters: `US_area` is not the same as `us_area`.

   (d) Word separation in R was traditionally done with periods, but this is slowly losing favor. Other options include snake_case (separated by underscores, as seen here) or camelCase (capitalize each new word).

2. Often, "quick and dirty" variable names that you will be using often in the Console are named with single letter variables, whereas variables in a script are long enough to be self-explanatory.

   **Tip**: Note that in RStudio, the `Tab` key will attempt to autocomplete the variable or function name that your cursor is currently on.

3. Use the `rm()` function to get rid of a variable from your environment.

```
rm(us_pop_density) # gets rid of the us_pop_density variable
```

   Note that removing variables from your environment can help reduce clutter and is essential when dealing with large objects.

## iii. Working with environments and history

1. You can save your environment (the set of variables you have defined). To do so, click the Save icon in the Environment tab (top right). Once you have saved your environment, the actual R command that was run pops up in your Console. Note that RStudio automatically adds the traditional file extension of `.RData`.

2. To clear your current environment, click the broom icon on the Environment panel. You can also achieve this by typing:

```
rm(list = ls(all = TRUE))
```

   Note that all the variables you just defined have disappeared!

 

3. To load an environment, click the Load icon and select the `.RData` file that you saved earlier. Again, you'll see the corresponding R command in the Console panel. Note that loading an environment does not empty your existing environment, but it will overwrite any existing variables.

4. It is good practice to have a separate directory for each project or analysis that you are working on. If you tell R about this directory, it will, by default, load and save files from it. We call this the working directory. You can browse files and directories from the Files tab of the lower right panel. Set the working directory using the gear icon (the More button). Alternatively, you can use the `ctrl-shift-H` shortcut. Once run, the R command to set the working directory is also shown in the Console tab.

5. When you quit R, you will be asked if you want to save your workspace image (meaning your environment) in your working directory in a file called `.RData`.

6. RStudio always saves your history in your working directory. This can be a problem when restarting RStudio just by clicking on the Dock or the Start menu as your working directory will be your home directory and you will not see the history saved from your last session. On Macs, an easy way to specify your project directory when starting R is to drag the folder you want onto the RStudio icon. For Windows, the icon in your Toolbar does not work; you will have to use an alias on your Desktop. This will also load any saved `.Rdata` and .Rhistory files from that directory.

7. An even better practice is to create a new RStudio project for every analysis. This is RStudio's way of supporting and streamlining the common practice of keeping all the input data, R scripts, and other files associated with that analysis together. This will create a .Rproj file in your project directory. Now, whenever you want to work on that analysis again, opening that .Rproj file will bring you back to exactly where you left off (the same working directory, the same files open, the same command history), although you'll have to re-populate your environment.

8. Note that you can easily copy a line from your history to the Console by double-clicking it, or using the To Console icon.

## iv. Getting help

1. Much work has gone into making R self-documenting. There are extensive built-in help pages for all R commands, which are accessible with the `help()` function. Thus, to see how `sqrt()` works, type:

   ```
   help(sqrt)
   ```

   The help page will show up in the Help section of the RStudio window. In case typing `help` is too long, there is a shortcut.

   ```
   ?sqrt
   ```

2. It should be noted that some special characters or reserved words have to be quoted when using either of the above help functions.

   ```
   ?"+"
   ```

   will show the help page for the arithmetic operators. Note that since the `+` function is just one of a group of similar operators, the help page explains all of them in a single page, rather than having separate pages for `+`, `-`, `*`, `/` etc. The help pages quite often will group similar functions together this way (e.g., the related functions `log()` and `exp()` are found on the same page).

3. Another very useful command is the `example()` function. Almost all R commands will include a series of examples on their help pages (accessible using the `help()` or `?` functions). You can run these

 

examples directly from your console by using the `example()` function. To see the examples for the `sqrt()` function, type:

```
example(sqrt)
```

This runs the set of examples that are listed at the bottom of the help page, exactly as if you had typed them out yourself.

4. The R help is not always as transparent as one would like and StackOverflow (`stackoverflow.com`) may be a better bet for answering your questions.

## v. Data types

1. So far, we have only been dealing with numerical data, but in the real world, data takes many forms. R has several basic data types that you can use.

```
has_diabetes    <- TRUE         # logical  (note case!)
patient_name    <- "Jane Doe"   # character
moms_age        <- NA           # used to represent an unknown ("missing") value
NY_socialite_iq <- NULL         # used to represent something that does not exist
```

2. When working with truth values, you can use the logical operators:

```
AND (&)
OR (|)
NOT (!)
```

```
is_enrolled  <- FALSE
is_candidate <- has_diabetes & ! is_enrolled
```

3. R uses tri-state logic when working with truth values.

```
TRUE & FALSE
T | F
T & NA
F & NA
TRUE | NA
FALSE | NA
TRUE & ! FALSE & NA
```

4. R can convert among datatypes using a series of `as.()` methods.

```
as.numeric(has_diabetes)
as.numeric(is_enrolled)
as.character(us_population)
as.character(moms_age) # still NA - we still don't know!
```

 

# III Keeping your code

Up to this point, we have been using the interactive console panel of RStudio. We have seen that the history panel keeps a record of what we have done, but what would happen if we quit RStudio and did not save the .Rdata? The objects in our environment would disappear, and we would have to trawl through the command history, remember those commands that worked, and re-run them to recreate our environment.

There are two ways in which we can keep track of which commands we want to run to re-generate our analyses, both of which use the Source panel.

## i. R scripts

A best practice is to store R commands in a script file. Choose *File* ⇒ *New File* ⇒ *R Script*, or from the green cross icon, choose R Script. An empty tab should appear, which you can populate with R commands, exactly as you would type them in the Console panel. You can use the Console panel to engineer the command to do exactly what you want, then copy it to the Source window. An easy way to do this is to highlight the command in the History panel and click the 'To Source' button.

From the Source panel, you can run a single line (that the cursor is on), or a selection of highlighted lines (Mac users press cmd-return). Clicking the Source button will run all the commands in the script, from start to finish. Note that nothing will be printed to the Console window unless explicitly declared, or you use the 'Source With Echo' option, which will print both the command and any output.

A good practice is to periodically empty your environment (but not your history - you might need it!), and try to re-create it using just the script. This will let you know if you have inadvertently missed adding a crucial command!

## ii. R Markdown

You can use RMarkdown documents as a way of combining not just the code, but also notes and comments about the code and project. It can be thought of as akin to a lab notebook, which you can use to capture scientific ideas and the associated analysis results and communicate these with colleagues.

To start a new R Markdown document, choose *File* ⇒ *New File* ⇒ *R Markdown*, or select R Markdown from the green cross icon. You will be asked to fill in some basic metadata (name of document and author) and what you want the default output format to be.

There are three parts to an R Markdown document:

1. a YAML header, surrounded by triple dashes, which is automatically generated for you when you open a new document

```
---
title: "Some title"
author: "Luce"
output: html_document
---
```

 

2. the chunks of R code that will be run, surrounded by triple backticks (```)

   ```
   ``` {r optional_name, optional_options}
   R code here
   ```
   ```

3. text, with Markdown formatting.

To produce a complete report including code and text, click the Knit button. If you haven't already saved the file, it will ask you to do so now. Try this now, with the default content given to you on creation.

By default, an HTML file will be produced, with a preview either in the Preview panel, or externally (which preference can be set from the Options menu (the cog icon)). The HTML file can be sent to colleagues or your PI, and viewed in a browser. Other output options include PDF and Word, although you will need to install separate packages for that functionality. We will talk more about external packages in a later section.

Each R code chunk can be run separately. The output from each chunk can include both the code and the results (use `echo=TRUE` as a chunk option) or just the results (use `echo=FALSE`). The output, including figures, may be shown inline (in the .Rmd file), but many people find this distracting, and prefer to see the output in the Console and Plot panels, which can be set from the Options menu (select Chunk Output in Console).

It is also possible to include mini-code chunks inline, surrounded by single backticks, and prefixed by the letter `r`. These will get evaluated, and get filled in, whenever the document is knitted.

The formatting of the text is accomplished with standard Markdown formatting. Examples include:

- Different level headings are prefixed by varying numbers of `#`.
- Italic or bold text is surrounded by one or two `*`, respectively,
- Bulleted list items are prefixed by `*`,
- Numbered list items are prefixed by numbers.

There are a number of guides to help you construct your R Markdown documents directly from within RStudio:

- an external R Markdown cheatsheet PDF, which lists all the most commonly used commands and syntax, accessed from *Help* ⇒ *Cheatsheets* ⇒ *R Markdown Cheat Sheet*,
- an external markdown reference guide PDF, accessed from *Help* ⇒ *Cheatsheets* ⇒ *R Markdown Reference Guide*, also found inline at *Help* ⇒ *Markdown Quick Reference*.

 

# IV Data Structures

## i. Overview

R has several different types of data structures and knowing what each is used for and when they are appropriate is fundamental to the efficient use of R. The ones that we are going to examine in detail here are: vectors, matrices, lists and data frames.

A quick summary of the four main data structures:

**Vectors** are ordered collections of elements, where each of the objects must be of the same data type or mode, but can be any mode.

A **matrix** is a rectangular array, having some number of columns and some number of rows. Matrices can only comprise one data type (if you want multiple data types in a single structure, use a data frame).

**Lists** are like vectors, but whereas elements in vectors must all be of the same type, a single list can include elements from any data type. Elements in lists can be named. A common use of lists is to combine multiple values into a single object that can then be passed to, or returned by, a function.

**Data frames** are similar to matrices, in that they can have multiple rows and multiple columns, but in a data frame, each of the columns can be of a different data type; within a column, all elements must be of the same data type. You can think of a data frame as being like a list, where each element corresponds to a complete vector, and all elements are the same length.

## ii. Vectors

1. We've already seen a vector when we ran the `rnorm()` command. Let's run that again, but this time assigning the result to a variable.

   ```
   x <- rnorm(100)
   ```

2. Many commands in R take a vector as input.

   ```
   sum(x)
   max(x)
   summary(x)
   plot(x)
   hist(x)
   ```

   Don't get too excited about the plotting yet; we will be making prettier plots soon!

3. There are many ways of creating vectors. The most common way is using the `c()` function, where `c` stands for concatenation. Here we assign a vector of characters (character strings must be quoted).

   ```
   my_colors <- c("red", "orange", "yellow", "green", "blue", "indigo", "violet")
   ```

4. The `c()` function can combine vectors.

   ```
   my_colors <- c("infrared", my_colors, "ultraviolet")
   # remember that "infrared" and "ultraviolet" are one-element vectors
   ```

   By assigning the result back to the `my_colors` variable, we are updating its value. The net effect is to both prepend and append new colors to the original `my_colors` vector.

5. You can get the length of a vector using the `length()` function.

 

```
length(my_colors)
```

6. You can access an individual element of a vector by its position (or "index"). In R, the first element has an index of 1.

   ```
   my_colors[7]
   ```

7. You can also change the elements of a vector using the same notation as you use to access them.

   ```
   my_colors[7] <- "purple"
   ```

   **Tip**: Appending an element is a slow operation because it actually creates a new vector. If you do this a limited number of times, this is fine, but if you are doing this 1000s of times, it is more efficient to create an empty vector of a pre-determined size, and then change the elements. You can create a blank vector using the `vector()` function.

   ```
   a_numeric_vector <- vector(mode="numeric", length=1000)
   a_numeric_vector[50] <- 5
   a_numeric_vector[750] <- 10
   plot(a_numeric_vector)
   ```

8. You can access multiple elements of a vector by specifying a vector of element indices.

9. R has many built-in datasets for us to play with. You can view these datasets using the `data()` function. Two examples of vector datasets are `state.name` and `state.area`.

10. We can get the last ten states (alphabetically) by using R's convenient way of making a vector of sequential numbers, with the ":" operator.

    ```
    indices <- 41:50
    indices[1]
    indices[2]
    length(indices)
    state.name[indices]
    ```

    **Exercise:**

    a. We've seen how to list the last 10 states (alphabetically). How would you list the first 10 states?

    b. How would you list the first 10 **and** last 10 states (alphabetically)?

    c. Can you generalize this so that it works for any arbitrary length vector?

11. We can test all the elements of a vector at once using logical expressions. Let's use this to get a list of small states. First, how do we determine what a small state is?

    ```
    summary(state.area)
    ```

    Next, figure out which states are in the bottom quartile.

    ```
    cutoff <- 37317
    cutoff <- summary(state.area)[2]
    state.area < cutoff
    ```

    Note that this returns a vector of logical elements. We have seen that we can access vector elements using their indices, but we can also access them using logical vectors.

    ```
    small_states <- state.name[state.area < cutoff]
    ```

12. We can test for membership in a vector using the `%in%` operator. To see if a state is among the smallest:

 

```
"New York" %in% small_states
"Rhode Island" %in% state.name[state.area < cutoff]
```

13. You can also get the positions of elements that meet your criteria using the `which()` function.

    ```
    which(state.area < cutoff)
    state.name[which(state.area < cutoff)]
    ```

    Techniques like this can be useful for removing outliers from your data.

14. Let's get the area of Wyoming:

    ```
    state.area[state.name == "Wyoming"]
    ```

    Notes:

    (a) The `==` is a test for equality. This is different from assignment.

    (b) The indexing vector here is a logical vector.

15. While this works, it can be a little long-winded. Luckily, R lets us name every element of a vector using the `names()` function.

    ```
    names(state.area) <- state.name
    ```

16. And now we can access Wyoming directly:

    ```
    state.area["Wyoming"]
    ```

17. Here the indexing vector we are using to access elements is a character vector.

    ```
    state.area[c("Wyoming", "Alaska")]
    ```

18. Now we can see all the small states and their areas in one shot:

    ```
    state.area[small_states]
    ```

19. Sadly, not all functions that fetch an element from a vector keep the associated name.

    ```
    min(state.area)
    ```

    But you can find the index at which the minimum occurs, and use that.

    ```
    state.area[which.min(state.area)]
    ```

20. In addition to using the ":" notation to create vectors of sequential numbers, there are a handful of useful functions for generating vectors with systematically created elements.

    ```
    seq(1, 10)     # same as 1:10
    seq(1, 4, 0.5) # shows all numbers from 1 to 4, incrementing by 0.5 each time
    ```

    Let's look carefully at the help page for the `seq()` function.

    ```
    ?seq
    ```

21. The `seq()` function can take five different arguments, but not all of them make sense at the same time. In particular, it would not make sense to give the `from`, `to`, `by`, and `length` arguments, since you can figure out the length using the values of `from`, `to`, and `by`. You can pass argments by name rather than position; this is helpful for skipping arguments.

    ```
    seq(0, 1, length.out = 10)
    ```

    **Tip**: In scripts, it is often good form to use named arguments, even when not necessary, as it makes your intent clearer.

 

```
seq(from = 1, to = 4, by = 0.5)
seq(from = 0, to = 1, length.out = 10)
```

22. Take a look at the help again: note that all of the arguments have default values, which will be used if you don't specify them.

    ```
    seq(to = 99)
    ```

23. Another commonly used function for making regular vectors is `rep()`. This repeats the values in the argument vector as many times as specified. This can be used with character and logical vectors as well as numeric.

    ```
    rep(my_colors, 2)
    rep(my_colors, times = 2) # same as above
    rep(my_colors, each = 2)
    rep(my_colors, each = 2, times = 2)
    ```

24. When using the `length.out` argument, you may not get a full cycle of repetition.

    ```
    rep(my_colors, length.out = 10)
    ```

25. In many cases, R will implicitly "recycle" vector elements as needed to get operations on vectors to make sense. When vector operations align, results are as you would expect:

    ```
    x <- 0:9
    y <- seq(from = 0, to = 90, by = 10)
    x + y
    ```

    Here, the first element of `x` has the first element of `y` added to it, the second element of `x` has the second element of `y` added to it, etc. What happens when the vectors are not the same length?

    ```
    (1:5) + y
    ```

26. Here the elements of the first vector were recycled (your linear algebra professor would be horrified). When one vector is shorter than the other, the elements of that entire vector get recycled, starting from the first element and getting repeated as often as necessary. Note that if this mechanism does not use a complete cycle, you'll get a warning.

    ```
    (1:4) + y
    ```

27. Finally, note that using a single value (i.e., a scalar) is just a special case of recycling the same value over and over.

    ```
    y * 2
    ```

    **Exercise:**

    a. `0:10 / 10` yields the same result as `seq(from = 0, to = 1, by = 0.1)`. Can you understand why? Which do you think is more efficient?

    b. Can you predict what this command does?

    ```
    10 ^ (0:5)
    ```

28. R supports sorting, using the `sort()` and `order()` functions.

    ```
    sort(state.area)               # sort the areas of the states from smallest to largest
    order(state.area)              # return a vector of the positions of the sorted elements
    state.name[order(state.area)]  # sort the state names by state size
    state.name[order(state.area, decreasing = TRUE)]  # sort the state names by state size
    ```

 

29. We can also randomly sample elements from a vector, using `sample()`.

```
sample(state.name, 4)              # randomly picks four states
sample(state.name)                 # randomly permute the entire vector of state names
sample(state.name, replace = TRUE) # selection with replacement
```

This is frequently used in bootstrapping techniques.

30. Other miscellaneous useful commands on vectors include

```
rev(x)    # reverses the vector
sum(x)    # sums all the elements in a numeric or logical vector
cumsum(x) # returns a vector of cumulative sums (or a running total)
diff(x)   # returns a vector of differences between adjacent elements
max(x)    # returns the largest element
min(x)    # returns the smallest element
range(x)  # returns a vector of the smallest and largest elements
mean(x)   # returns the arithmetic mean
```

**Summary**: Vector elements are accessed using indexing vectors, which can be numeric, character or logical vectors.

**Summary**: List of logical expression functions:
`<` `>` `<=` `>=` `!=` `==` `%in%`

**Summary**: Methods of generating regular vectors:
1. Numeric vector, from scratch, shortcut:
   `from:to`
2. Numeric vector, from scratch:
   `seq(from, to, by, length.out, along.with)`
3. Any type of vector, derived from an existing one (x):
   `rep(x, times, length.out, each)`

## iii. Factors

Factors are similar to vectors, but they have another tier of information. A factor keeps track of all the distinct values in that vector, and notes the positions in the vector where each distinct value can be found. Factors are R's preferred way of storing **categorical data**.

The set of distinct allowed values are called levels. To see (and set) the levels of a factor, you can use the `levels()` function, which will return the levels as a vector.

1. R has an example factor built in:

```
state.division
levels(state.division)
```

2. To get a hint about how R stores factors (or any other object), we can use the `str()` function to view the structure of that object. You can also use the `class()` function to learn the class of an object, without having to see all the details.

```
str(state.division)
class(state.division)
```

 

Note the list of integers corresponds to the level at each position. While factors may behave like character vectors in many ways, they are much more efficient because they are internally represented as integers and computers are good at working with integers.

3. You can convert a vector to a factor using the `factor()` function. Let's wish for some ponies.

```
pony_colors <- sample(my_colors, size = 500, replace = TRUE)
str(pony_colors)
```

Note that we are storing each color as a character string. This is not ideal. Let's convert this vector to a factor.

```
pony_colors_f <- factor(pony_colors)
str(pony_colors_f)
```

4. You can plot a factor to see how frequently each level appears.

```
plot(pony_colors_f)
```

The levels are plotted in the order they are returned by `levels()`. By default, levels will be listed alphabetcially, unless you specify otherwise. Let's see how you can control the order of the levels when you create the factor.

```
pony_colors_f <- factor(pony_colors, levels = my_colors)
str(pony_colors_f)
plot(pony_colors_f)
```

5. You can make a factor from a factor, reordering its levels as you go.

```
plot(state.division)
state.division <- factor(state.division, levels = sort(levels(state.division)))
plot(state.division)
```

6. You can rename the levels in a factor by assignment to its `levels()`. This only changes the labels, not the underlying integer representation. In this case, the labels we have are quite long; let's abbreviate them.

```
levels(state.division)
levels(state.division) <- c("ENC", "ESC", "MA", "MT", "NE", "PAC", "SA", "WNC", "WSC")
plot(state.division)
```

7. In most cases, you can treat a factor as a character vector, and R will do the appropriate conversions. Here we list the states in the North East, and then compare the sizes of various groups of states.

```
state.name[state.division == "NE"]
mean(state.area[state.division == "NE"]) / mean(state.area[state.division == "WSC"])
t.test(state.area[state.division == "SA"], state.area[state.division == "MT"])
```

## iv. Matrices

Multi-dimensional structures in R, where all the elements are of the same data type, are called arrays. Arrays have three dimensions: number of rows, number of columns and number of layers. Matrices are a special type of array using only two of those dimensions (rows and columns) and can be thought of as tables.

1. Let's use one of R's built-in datasets, the `USPersonalExpenditure` dataset, which describes how much Americans spent in five categories from 1940-1960.

 

```
?USPersonalExpenditure
USPersonalExpenditure
```

2. Notice that the rows and columns of the matrix are named, using the categories and years as row names and column names, respectively. You can access (and set) the row names and column names using the `rownames()` and `colnames()` functions. There is a third function, `dim()`, which tells you the number of rows and columns making up your matrix.

   ```
   rownames(USPersonalExpenditure)
   colnames(USPersonalExpenditure)
   dim(USPersonalExpenditure)
   ```

   These functions return vectors, which can be accessed with all the usual access methods.

3. Accessing (and assigning) elements of a matrix is analogous to accessing (and assigning) elements of a vector, except that matrices need two indexing vectors, the first specifying rows and second specifying columns.

4. Let's say we wanted to see the Food and Tobacco expenditure for 1950:

   ```
   USPersonalExpenditure[1, 3]
   USPersonalExpenditure["Food and Tobacco", "1950"]
   USPersonalExpenditure[1, "1950"]
   ```

5. To get this expenditure for several years, we pass a vector for the column index.

   ```
   USPersonalExpenditure[1, c(5, 3, 1)]
   USPersonalExpenditure["Food and Tobacco", c("1960", "1950", "1940")]
   ```

6. Omitting a row or column index implies that all of the elements are wanted along that dimension.

   ```
   USPersonalExpenditure[1, ]
   USPersonalExpenditure["Food and Tobacco", ]
   USPersonalExpenditure["Food and Tobacco", , drop = FALSE]
   USPersonalExpenditure[ , c("1940","1950")]
   USPersonalExpenditure[1:3, c("1940","1950")]
   ```

7. Observe that if your result is one-dimensional, it is by default returned as a vector. All the vector operations you learned can be used on this output. If you don't want this behavior, you can use the `drop=FALSE` option, as was shown in the third example above.

   ```
   sum(USPersonalExpenditure[ , "1940"])
   ```

8. If you access an element of a matrix using only one index, R will treat the elements of the matrix as a vector of stacked columns.

   ```
   USPersonalExpenditure[1]
   USPersonalExpenditure[2]
   USPersonalExpenditure[7]
   ```

9. This is usually not the clearest way to access elements, but can be useful when you want to work with all of the elements.

   ```
   length(USPersonalExpenditure)
   sum(USPersonalExpenditure)
   ```

10. There are a few ways to make a new matrix. The `matrix()` function takes as arguments a vector of all the elements, and then some information about how many rows and columns there are. By default,

 

the matrix is filled in column order, but you can change this behaviour with the `by.rows=TRUE` option.

```
game1 <- matrix(c("X","","O","","X","O","","",""), ncol = 3)
game2 <- matrix(c("X","","O","","X","O","","",""), ncol = 3, byrow = TRUE)
```

Notes:

a. The elements of this matrix are characters. As with vectors, all elements in a matrix must be of the same datatype.

b. In this matrix, the rows and columns are not named, and can only be accessed using numeric indices.

11. Assignment of values is just like for vectors, except using two dimensions.

```
game1[3, 3] <- "X" # I win!!!!!
```

12. When making a fresh game, we can set the size in advance, and fill it in piecemeal. This is much more efficient than extending the matrix each time a new row or column is added.

```
new_game <- matrix(data = "", ncol = 3, nrow = 3)
```

13. We can assemble a matrix by combining vectors or matrices. They can be stacked by row or column, using the `rbind()` or `cbind()` functions, respectively. Here, we build a chess board from the component pieces.

```
pieces <- c("rook", "knight", "bishop", "queen", "king", "bishop", "knight", "rook")
pawns <- rep("pawn", 8)
board <- rbind(pieces, pawns, matrix("", nrow = 4, ncol = 8), pawns, pieces)
rownames(board) <- 8:1
colnames(board) <- letters[1:8]
```

`letters` is a very handy built-in vector of all 26 lowercase letters. See the help for `Constants` for a few more handy built-ins.

14. Remember all of the elements in a matrix must be of the same data type. If you assign one element of a different data type, R will convert (or coerce) elements as necessary.

```
USPersonalExpenditure["Personal Care", "1955"] <- "Unknown"
USPersonalExpenditure
```

All the numbers have been converted to characters. How should we have done this to avoid this coercion?

```
rm(USPersonalExpenditure) # revert to built-in dataset
```

15. R coerces elements up the data type chain, only far enough to satisfy the rule that all elements remain the same type. If you add an integer to a matrix of logicals, you'll get a matrix of integers, not characters, as in the example above.

```
NULL < raw < logical < integer < double < complex < character < list < expression
```

The same coercion strategy applies to vectors.

The next section introduces a data structure that allows mixed types.

 

## v. Lists

Lists are similar to vectors, but with some differences. Lists can hold data structures of different types, and of different sizes. Each component in a list can be (optionally, but commonly) separately named (a list in R is analogous to a hash or associative array in most other programming languages). In fact, one list can be a member of another list, allowing for deeply nested and arbitrarily complex data structures to be modeled.

1. The results of many high-level analyses in R are packaged as lists. Let's use R to fit a linear model to some data.

   ```
   (edu_spend <- unname(USPersonalExpenditure["Private Education", ]))
   (edu_yr <- seq(from = 0, to = 20, by = 5))
   plot(edu_yr, edu_spend)
   my_model <- lm(edu_spend ~ edu_yr)
   my_model
   summary(my_model)
   abline(my_model)
   plot(my_model)
   ```

   The `lm()` function fits data to a linear model (i.e., performs a linear regression), and packages up all of the results into an object we have named `my.model`. The `my.model` object is of class `lm` (linear model), which is a special kind of `list`. You may have to `rm()` USPersonalExpediture from your environment before running the above commands, if your object is still coerced into character strings from the previous section.

2. Remember that we can look into any object using the `str()` function. Let's do that now with our linear model. Brace yourself!

   ```
   str(my_model)
   ```

   Take a moment to get a sense of what is packaged in the linear model object (but don't stress over the details).

   Now look carefully at the very first line of the output and note that this object is a "List of 12". This object has 12 components, and many of those have sub-components (an element of a list can be another list).

   We'll need to learn about lists so we can access the subcomponents of the model object; for example, you'd probably want to extract the slope and intercept of the best-fit line from this object. Let's start with the basics...

3. Lists can be created using the `list()` function. When using the `list()` function, you can optionally give names to the components (component names are called tags).

   ```
   ad_mouse_colony <- list("9.1",  FALSE)
   ad_mouse_colony <- list(room = "9.1",  bsl3 = FALSE)
   ```

   Note that we have different data types in the same list (character, and logical). Note also the difference between the first and second attempt to model a mouse colony. In the first case, you need to know what the position of each component is; in the second, each component is named (tagged) with the tag we provided at creation.

4. We can add a component to an existing list

   ```
   ad_mouse_colony$conditions <- list(bedding = "straw", light_hrs = 12)
   ad_mouse_colony$count <- c(male = 10, female = 0)
   ```

 

```
ad_mouse_colony[["variants"]] <- c("APP695swe", "PS1-dE9")
```

Note that we have used two different notations to refer to components of a list.

5. There are several ways to refer to the components of a list; the differences can be subtle, but important.

   The most straightforward way to refer to a single component is using the `[[` notation. You can always provide an integer to access the component by position, or you can provide a character string if the component is tagged.

   ```
   ad_mouse_colony[[1]]
   ad_mouse_colony[["room"]]
   ```

   For tagged components where the tag is a literal character string, you can use the `$` notation. This is less flexible, but looks clearer. Use this when you can.

   ```
   ad_mouse_colony$room
   ```

6. Just as for vectors, to get the names of tagged components, use `names()`.

   ```
   names(ad_mouse_colony)
   ```

7. If you want to access multiple components, you can use the standard `[` notation, which takes the usual indexing vector. Note that this will always return a list (this has to be so, because the components returned could be of different datatypes).

   ```
   ad_mouse_colony[c(1,4)]
   ```

   **Exercise:**

   a. What does this return?

      ```
      ad_mouse_colony[[3]]
      ```

   b. How about this?

      ```
      ad_mouse_colony[[3]]$bedding
      ```

   c. Why does this NOT work?

      ```
      ad_mouse_colony[3]$bedding
      ```

   d. How would you add a new element (say, temp = 36.5) to the conditions list?

8. To remove a component, assign it the value `NULL`.

   ```
   ad_mouse_colony$bsl3 <- NULL
   ```

   Note that this changes the index positions of all subsequent components (yet another reason to use tags!)

9. All objects can have arbitrary additional attributes, which can be used to store metadata about the object, which are stored as a named list (with unique names). Attributes can be accessed individually with `attr()` or all at once (as a list) with `attributes()`.

   ```
   x <- c(0, 1, 1, 2, 3, 5, 8, 13, 21)
   attr(x, "description") <- "Fibonacci sequence"
   attr(x, "description")
   attributes(x)
   str(attributes(x))
   ```

 

10. Remember our linear model?

    **Exercise:**

    a. Can you extract the slope and intercept of the best-fit line?

    b. Can you extract the value of $R^2$ reported by the `summary()` function we used? *Hint: save the result of this function, and look into its structure.*

## vi. Data frames

Data frames are two-dimensional data structures like matrices, but, unlike matrices, they can contain multiple different data types. You can think of a data frame as a list of vectors, where all the vector lengths are the same. Data frames are commonly used to represent tabular data.

1. When we were learning about vectors, we used several parallel vectors, each with length 50 to represent information about US states. The collection of vectors really belongs together, and a data frame is the tool for doing this.

   ```
   state_db <- data.frame(state.name, state.abb, state.area, state.center,
                          stringsAsFactors = FALSE)
   state_db
   ```

   The `data.frame()` function combines the four data sets into a single data frame. Note that the first three data sets are vectors (two character, one numeric), but the last data set is a list with two components.

   Note the `stringsAsFactors = FALSE` argument. Some of the vectors that we are using are character vectors, but will be automatically converted to factors if this option is not set. Since we will want to work with our character data as vectors, not as factors, we want to set this argument to `FALSE`.

2. In addition to the `str()` function, you can glean useful information about a data frame (and other data structures) using the `summary()` and `head()` functions.

   ```
   summary(state_db)
   head(state_db)
   ```

3. Data frames have a split personality. They behave both like a tagged list of vectors, and like a matrix! This gives you many options for accessing elements. Fortunately, you know them all already!

   When accessing a single column, the list notation is preferred.

   ```
   state_db$state.abb
   state_db[[ "state.abb" ]]
   state_db[[ 2 ]]
   ```

   When accessing multiple columns or a subset of rows, the matrix notation is used (rows and columns are selected by indexing vectors).

   ```
   state_db[ , 1:2]
   state_db[41:50, 1:2]
   state_db[c(50, 1), c("state.abb", "x", "y")]
   state_db[order(state_db$state.area)[1:5], ]
   state_db[order(state_db$state.area), ][1:5, ]
   ```

   The last two examples produce the same output; which is more efficient?

 

4. As with matrices, the rows can be given names as well. This makes picking out specific rows less error-prone. Column names are accessed with the `names()` or `colnames()` functions.

   ```
   rownames(state_db) <- state.abb
   state_db[c("NY", "NJ", "CT", "RI"), c("x", "y")]
   names(state_db) <- c("name", "abb", "area", "long", "lat")
   ```

   Note that if you only fetch data from one column, you'll get a vector back. If you want a one-column data frame, use the `drop = FALSE` option.

5. You can add a new column the same way you would add one to a list.

   ```
   state_db$division <- state.division # Remember, this is a factor

   state_db$z.size <- (state_db$area - mean(state_db$area))/sd(state_db$area)
   state_db[ , "z.size", drop = FALSE]
   ```

6. Remember that you can pass logical vectors as indices.

   ```
   state_db[state_db$area < median(state_db$area), "name"]
   state_db[state_db$area < median(state_db$area), "name", drop = FALSE]
   coastal <- state_db[ state_db$division %in%
     c("New England", "Middle Atlantic", "South Atlantic", "Pacific"), ]
   ```

7. Another way to select data from a data frame is using the `subset()` function. You can choose rows based on a logical expression and can choose columns with the `select` option. With this function, we only have to type the dataset name once.

   ```
   subset(state_db, area < median(area), select = name)
   coastal <- subset(state_db, division %in%
     c("New England", "Middle Atlantic", "South Atlantic", "Pacific") )
   ```

   The logical condition is optional, and you can specify columns to omit instead of columns to include.

   ```
   subset(state_db, select = c(name, abb))
   subset(state_db, select = -c(long, lat))
   ```

8. Many tools in R work naturally with data frames. For example, visualizing the size distribution of state within each division could not be easier once the data is in a well-designed data frame.

   ```
   plot(area ~ division, data = state_db)
   plot(log(area) ~ division, data = state_db)
   plot(lat ~ long, data = state_db)
   text(lat ~ long, data = state_db, rownames(state_db))
   ```

   Here we are using R's function notation. Read the first argument in the first `plot()` example as "area as a function of division".

9. **Exercise:**

   a. Can you add Puerto Rico to our data frame? [abb = "PR", area = 3515 sq mi, long = -66.1, lat = 18.45, division = "South Atlantic"]

   b. How about Greenland? [abb = "GL", area = 836330 sq mi, long = -51.73, lat = 64.17, division = "Arctic Circle"]

 

# V Importing (and exporting) data

1. In most cases in the lab, you won't be typing the data in by hand but rather importing it. R provides tools for importing a variety of text file formats. If you receive data in Excel format, you'll want to save it as tab-delimited or CSV (comma separated values) text. The `read.delim()`, `read.csv()` or the very general `read.table()` functions can then be used to import the data into a data frame. Of course, you should also tell your colleagues that R is the preferred tool for data wrangling!

   We have prepared an Excel file for you to import.

   a. Go to `http://chagall.med.cornell.edu/Rcourse/` and download the Ablation.xlsx file into your project folder.

   b. Open the file using Microsoft Excel and save it in CSV format. You can quit Excel now!

   c. Return to RStudio.

   d. Use the Files tab (bottom right) to take a look at the .csv file.

   e. Import the data into a data frame:

      ```
      ablation <- read.csv("Ablation.csv", header = TRUE, stringsAsFactors = TRUE)
      ```

      Note that we explicitly asked that strings be converted to factors. In cases where you are importing string data, you would want to set this to `FALSE`, and after import, convert appropriate columns to factors.

   f. Rename the `SCORE` column to be consistent with the other column names.

      ```
      names(ablation)[names(ablation) == "SCORE"] <- "Score"
      ```

      Note that we did not use `names(ablation)[6] <- "Score"`. Why do you think that is?

   There is another more general import function, `read.table()`, that gives you exquisite control over how to import your data. One of the defaults of this function is `header = FALSE`. For this reason, we suggest that you always explicitly use the `header` option (you don't want to accidentally miss your first data point).

2. If your uninitiated colleagues insist on an Excel-compatible format, you can also export a data frame using `write.table()`.

   ```
   write.table(ablation, file = "ablation.txt",
               quote = FALSE, col.names = NA,
               sep = "\t")

   write.table(ablation, file = "ablation.txt",
               quote = FALSE, row.names = FALSE,
               sep = "\t")
   ```

 

# VI Miscellaneous

## i. User-defined functions

Although R has a wide array of in-built functions, as well as many other functions that can be accessed via third party packages (more on that later), we can also create our own functions, using `function()`. We define not only the arguments, or inputs, to the function, but also what it does, and what it returns.

1. Within the body of a function, all of the input arguments become variables that you can use, including passing them to other functions.

   ```
   mySummary <- function(x) {
     my_mean <- mean(x)
     my_sd <- sd(x)
     list(mean = my_mean, sd = my_sd)
   }
   mySummary(rnorm(100))
   ```

   Here, our function, called `mySummary`, takes a single argument called `x`. This function assumes that `x` is a numeric vector, and computes the mean and standard deviation of that vector. The function returns a list with two tagged components; the return value of a function is the last expression evaluated in the function body. The code executed by our function is enclosed in curly braces `{}`.

2. We can also create functions which can have multiple arguments, and default values for arguments.

   ```
   raiseNumber <- function(x, power = 1) {
     x ^ power
   }
   raiseNumber(10)
   raiseNumber(10, 3)
   ```

## ii. Loops

Computers are very good at dealing with repetitive tasks. Often we would like to perform a task for some number of objects, or a certain number of times. While we could just repeat the same command over and over again, changing one parameter each time, a better and less error-prone way is to write a loop.

Here we will look at using a 'for' loop, which takes some number of elements and runs a series of commands for each element. The 'for' loop starts by assigning the first element to a user-defined variable, and executing the statements in the body of the code. Then, if there are more elements, it goes back to step 1, assigns the next element to the variable, and so on, until there are no more elements, at which point it exits the 'for' loop and goes on to the next statement. Note that we pre-allocate the size of the vector that we store our results in, and fill in as we go.

```
num_iterations <- 100
my_means <- numeric(length = num_iterations)
for (i in 1:num_iterations) {
  x <- rnorm(10000)
  my_means[i] <- mean(x)
}
hist(my_means)
```

 