Analysis of Next Generation Sequencing Data
Introduction to R

Luce Skrabanek

12 February, 2019

1 Prologue

1.1 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.

1.2 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.

2 Introduction

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, `>`.

2.1 R as a calculator

R understands all the usual mathematical operators, keeping the usual order of operations, including forcing an order with parentheses (PEMDAS):

```
1 + 2
3 - 4
5 * 6
7 / 8
1 + 2 * 3
(1 + 2) * 3
```

When adding `1 + 2`, you should see . . .

```
[1] 3
```

The answer is of course 3, but what is the `[1]` before it? 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)
```

2.2 R has variables

Just as in UNIX, 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 <- 3.22e8 # From Wolfram|Alpha
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:

1. Once a variable is defined, you will see it show up in the environment panel in RStudio.
2. 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.
3. Case matters: `US.area` is not the same as `us.area`.
4. Word separation in R is traditionally done with periods, but this is slowly losing favor. Other options include snake_case (separated by underscores) or camelCase (capitalize each new word).

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

2.3 Getting help

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. To see how `sqrt()` works, type:

```
help(sqrt)
?sqrt
```

The help page will show up in the Help section of the RStudio window.

The help pages will often group similar functions together (e.g., the related functions `log()` and `exp()` are found on the same page).

2.4 Data types

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         # represents an unknown ("missing") value
NY.socialite.iq <- NULL       # represents something that doesn't exist
```

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

```
as.numeric(has.diabetes)
as.character(us.population)
as.character(moms.age)         # still NA - we still don't know!
```

3 Data Structures

3.1 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.

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, but can be any data type.

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 variable 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, although within a column, all elements must be of the same data type. You can think of a data frame as being like a list, but instead of each element in the list being just one value, each element corresponds to a complete vector.

3.2 Vectors

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. Many commands in R take a vector as input.

```r
x <- rnorm(100)
sum(x)
max(x)
summary(x)
length(x)
plot(x)
hist(x)
```

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

3.2.1 Creating vectors

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) to a variable `colors`.

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

The `c()` function can combine vectors.

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

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

While `c()` can be used to create vectors of any data type, an easy way to make a <u>numerical</u> vector of sequential numbers is with the ":" operator

```r
indices <- 41:50
```

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. Here we will look at `seq()` and `rep()`.

The `seq()` function can take five different arguments (`from`, `to`, `by`, `length`, `along.with`), although some are mutually exclusive with others. You can pass argments by name rather than position; this is helpful for skipping arguments. Note that all of the arguments have default values, which will be used if you don't specify them.

```
seq(1, 10)     # same as 1:10
seq(1, 4, 0.5) # all numbers from 1 to 4, incrementing by 0.5
seq(0, 1, length.out = 10)
seq(from = 1, to = 4, by = 0.5)
seq(from = 0, to = 1, length.out = 10)
seq(to = 99)
```

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

```
rep(colors, 2)
rep(colors, times = 2) # same as above
rep(colors, each = 2)
rep(colors, each = 2, times = 2)
rep(colors, length.out = 10)
```

3.2.2 Accessing, and assigning, individual vector elements

Individual vector elements are accessed using indexing vectors, which can be numeric, character or logical vectors.

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

```
colors[1]
colors[7]
```

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

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

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

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`.

We will get the last ten states (alphabetically) by using the ":" operator, and use that numerical vector to access the elements at those positions.

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

We can test all the elements of a vector at once using logical expressions, generating a logical vector. Let's use this to get a list of small states. First figure out which states are in the bottom quartile, and then compare every element to that number. This returns a vector of logical elements indicating, for every state, whether or not that state is smaller than the cutoff. We use that logical vector as our indexing vector, returning only those elements which correspond to a TRUE value at that position.

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

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

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

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

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

R also lets us name every element of a vector using the `names()` function, which will allow us to use a character vector to access individual elements directly.

```
names(state.area) <- state.name
state.area["Wyoming"]
state.area[c("Wyoming", "Alaska")]
```

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

```
sort(state.area)               # sorts the areas of the states from
    smallest to largest
order(state.area)              # returns 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
```

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
sample(state.name, replace = TRUE) # selection with replacement
```

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
```

3.3 Factors

Factors are similar to vectors, but they have another tier of information. A factor is used to store categorical data. They keep track of all the distinct values in that vector, and note the positions in the vector where each distinct value can be found.

The set of distinct 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.

R has an example factor built in:

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

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.

You can convert a vector to a factor using the `factor()` function. Here we create a vector using the `sample()` function, where we are storing each color as a character string, and then convert it into a factor.

```
bingo.balls <- sample(colors, size = 40, replace = TRUE)
str(bingo.balls)
bingo.balls.f <- factor(bingo.balls)
str(bingo.balls.f)
```

In most cases, you can treat a factor as a character vector, and R will do the appropriate conversions.

3.4 Matrices and lists

We are going to say very little about matrices and lists here.

Matrices can be thought of a two-dimensional vectors, where every element has to be of the same data type, and elements are accessed by two indexing vectors, one for rows, and one for columns.

Lists are like ragged tables, where every column can be of a different data type, and can have different numbers of elements. Each “column” of a list can be accessed in one of two ways: if the column is not named, we can access via a double bracket `[[ ]]` notation, and if it is named, we can use a `$` syntax.

```r
str(USPersonalExpenditure)
# access rows 1-3 and the 2nd and 4th column of a matrix
USPersonalExpenditure[1:3, c(2,4)]

str(state.center)
# access the x-axis coordinates of the center of each US state
state.center$x
state.center[[ 1 ]]
```

For more information about matrices and lists, see `http://chagall.med.cornell.edu/Rcourse/`.

3.5 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.

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.

```r
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 (since each string appears only once), we want to set this argument to `FALSE`.

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.

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

```r
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 given indexing vectors, which can be of any type (numeric, character, logical)).

```
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?

We can give names to both the rows and the columns of a data frame. This makes picking out specific rows less error-prone. Column names are accessed with the `names()` or `colnames()` functions; row names are accessed using `rownames()`.

```
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.

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]
```

3.6 Importing (and exporting) data

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 `read.table()` functions can then be used to import the data into a data frame.

The `read.table()` function is the most general, giving 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).

```
ablation <- read.table("ablation.csv", header = TRUE, sep = ",")
```

You can export a data frame using `write.table()`.

```
write.table(ablation, "my_ablation.txt", quote = FALSE, row.names =
    FALSE)
```

4 CRAN and Libraries

One of the major advantages of using R for data analysis is the rich and active community that surrounds it. There is a rich ecosystem of libraries (or packages) to the base R system. Some of these provide general functionality while others address very specific tasks.

The main hub for this ecosystem is known as CRAN (Comprehensive R Archive Network). CRAN can be accessed from `http://cran.r-project.org/` (where you also download the R software).

Follow the **Packages** link to browse the 5000+ packages currently available.

Since we will be usingthe **ggplot2** package for preparing publication-quality figures, here we will download and install the **tidyverse** package, which includes `ggplot2`, as well as `tidyr` which we will be using shortly.

```
install.packages("tidyverse") # Library name is given as a string
```

If this is the first time a package is being installed on your computer, R may ask you to select a CRAN mirror. Pick something geographically close by. Note that you only have to install a package once (per version of R).

Depending on how your computer (and R installation) is set up, you may receive a message indicating that the central location for packages is not writable; in this case R will ask if you want to use a personalized collection of packages stored in your home directory.

Installing a package does not make it ready for use in your current R session. To do this, use the `library()` function. You need to do this in every session or script that will use functions from this library.

```
library(tidyverse)     # Library name is an object (not a string)
```

5 Plotting

Although R has some basic plotting functionality which we have seen hints of, the **ggplot2** package is more comprehensive and consistent.

ggplot2 is written by Hadley Wickham (`http://hadley.nz/`). He maintains a number of other libraries; they are of excellent quality, and are very well documented. However, they are updated frequently, so make sure that you are reading the current documentation. For ggplot2, this can be found at `https://ggplot2.tidyverse.org/reference/`.

ggplot2 relies entirely on data frames for input.

Let's make our first ggplot with the `ablation` data that we imported earlier.

```
ggplot(ablation, aes(x = Time, y = Score)) + geom_point()
```

At a minimum, the two things that you need to give ggplot are:

1. The dataset (which must be a data frame), and the variable(s) you want to plot
2. The type of plot you want to make.

ggplot gives you exquisite control over plotting parameters. Here, we'll change the color and size of the points.

```r
ggplot(ablation, aes(x = Time, y = Score)) + geom_point(color = "red",
    size = 4)
```

Aesthetics are used to bind plotting parameters to your data.

```r
ggplot(ablation, aes(x = Time, y = Score)) +
  geom_point(aes(color = Experiment), size = 4)
ggplot(ablation, aes(x = Time, y = Score)) +
  geom_point(aes(color = Experiment, shape = CellType), size = 4)
```

ggplot objects are built up by adding layers. You can add as many layers as you like.

```r
ggplot(ablation, aes(x = Time, y = Score)) +
  geom_point(aes(color = Experiment), size = 4) +
  geom_text(aes(label = CellType), hjust = 0, size = 3)
```

It is sometimes useful to save off the base ggplot object and add layers in separate commands. The plot is only rendered when R "prints" the object. This is useful for several reasons:

1. We don't need to create one big huge command to create a plot, we can create it piecemeal.
2. The plot will not get rendered until it has received all of its information, and therefore allows ggplot2 to be more intelligent than R's built-in plotting commands when deciding how large a plot should be, what the best scale is, etc.

```r
p <- ggplot(ablation, aes(x = Time, y = Score))
p <- p + geom_point(aes(color = Experiment, shape = Measurement), size
    = 4)
p <- p + geom_line(aes(group = interaction(Experiment, Measurement,
    CellType),
                        color = Experiment,
                        linetype = CellType))
print(p) # plot gets rendered now
```

Here we've added a layer that plots lines, with a separate line for each unique combination of Experiment, Measurement, and CellType. The `interaction()` function takes a set of factors, and computes a composite factor. Try running...

```r
interaction(ablation$Experiment, ablation$Measurement, ablation$CellType)
```

...to see what this does. This composite factor is passed to the group aesthetic of `geom_line()` to inform ggplot2 which data values go together.

We have also added a new binding to `geom_point()`. The shape of each point is determined by the corresponding Measurement. Note that ggplot prefers six or fewer distinct shapes (i.e., there are no more than six levels in the corresponding factor). You can, however, use more adding a layer like `scale_shape_manual(values = 1:11)`.

Here we specify the shapes we want to use, as well as jittering the points slightly so they no longer sit directly on top of one another.

```
p + geom_point(aes(color = Experiment, shape = Measurement),
                     size = 4, position = position_dodge(0.5)) +
  scale_shape_manual(values = c(1,16))
```

For more details on using other `scale_()` layers to modify your plots, see `http://chagall.med.cornell.edu/Rcourse/` or the online documentation for ggplot (`https://ggplot2.tidyverse.org/reference/`).

This plot is probably showing too much data at once. One approach to resolve this would be to make separate plots for the LDLR and TfR measurements. You can make multiple plots at once using facets. Here are a few options.

```
p + facet_grid(Measurement ~ .)
p + facet_grid(. ~ Measurement)
p + facet_grid(Experiment ~ Measurement)
p + facet_grid(Measurement ~ Experiment)
```

6 Data wrangling

You may have noticed that the format of the `ablation` data frame is a bit peculiar. It is, however, in the canonical format for storing and manipulating data that you should be using. The hallmark of this canonical (tidy) format is that there is only one (set of) independently observed value(s) in each row. All of the other columns are identifying values. They explain what exactly was measured, and can be thought of as metadata.

More specifically, a tidy dataset is defined as one where:

- Each variable forms a column.
- Each observation forms a row.

When your data is in this format, it is straightfoward to subset, transform, and aggregate it by any combination of factors of the identifying variables. That is why, for example, the **ggplot** package essentially requires that your data is in tidy format.

The tidyverse that Hadley Wickham has been instrumental in creating has this format at its core, and his **tidyr** package includes functions to help coerce your data into this format.

6.1 Going long

If you are given data in non-canonical format, you can use the `gather()` function to fix it. This will convert a data frame with several measurement columns (i.e., "fat" or "wide") into a "skinny" or "long" data frame which has one row for every observed (measured) value. The `gather()` function takes multiple columns that all have the same measurement type, and collapses them into key-value pairs, duplicating all other columns as needed.

Let's start with a "fat" data frame that contains data about mouse weights.

```
set.seed(1)
mouse.weights.sim <- data.frame(
  time = seq(as.Date("2017/1/1"), by = "month", length.out = 12),
  mickey = rnorm(12, 20, 1),
  minnie = rnorm(12, 20, 2),
  mighty = rnorm(12, 20, 4)
)
```

This dataset consists of only one type of measurement - mouse weights - where each column in this dataset represents the weights of a given mouse over a year. The columns 'mickey', 'minnie' and 'mighty' are the names of each mouse, and each of the three columns contain weight data for that respective mouse. The tidy version of this data would have all the weight measurements in one column (the "values") with another column detailing which mouse (or column) that measurement came from (the "keys"). In addition to the data frame that we want to manipulate, and the names of the key and value columns that will be created, the `gather()` function also needs the columns that it will operate on. Names of columns are implicitly converted to column positions.

```
mouse.weights <- gather(data = mouse.weights.sim,
          key = mouse, value = weight, mickey, minnie, mighty)
mouse.weights <- gather(data = mouse.weights.sim, key = mouse, value =
    weight, -time)
mouse.weights <- gather(data = mouse.weights.sim,
          key = mouse, value = weight, mickey:mighty)
```

After gathering our data, each variable forms a column. Our three variables are `time`, `mouse`, and `weight`. Each row is now an observation. Before tidying our data, each row represented three observations. Note that the arguments to the key and value options become the names of the new columns. Now that the data have been tidied, it is trivial to use as input to ggplot.

```
ggplot(mouse.weights, aes(x = mouse, y = weight)) +
   geom_boxplot(aes(fill = mouse))
ggplot(mouse.weights, aes(x = time, y = weight)) +
   geom_boxplot(aes(group = time))
ggplot(mouse.weights, aes(x = time, y = weight)) +
   geom_boxplot(aes(group = time)) + geom_point(aes(color = mouse))
ggplot(mouse.weights, aes(x = time, y = weight)) +
   geom_point(aes(color = mouse)) + geom_line(aes(group = mouse, color
      = mouse))
```

6.2 Going wide

The complement of the `gather()` function is the `spread()` function. We can reshape our mouse weights to their original format, or reshape our ablation dataset into a dataframe where there is one row per time point and one column per CellType. For the ablation dataset, note that all of the experimentally measured values in new data frame will come from the original `Score` column (indicated by the `value` option).

```
spread(data = mouse.weights, key = mouse, value = weight)
spread(ablation, key = CellType, value = Score)
```

It is also possible to have columns that are combinations of identifiers, but you will need to include an extra step of manually combining those columns first. Say we wanted a wide table where each of the measurement columns showed the value for a specific combination of Experiment and CellType. We would use another function from the **tidyr** package, `unite()`, `col` is the name of the new column, and we need to supply the columns to paste together. Here, `ExptCell` is the new column that we are defining, as a combination of Experiment and CellType, where the names of the identifiers will be separated by a period.

```
abl.united <- unite(ablation, col = ExptCell,
                    Experiment, CellType, sep = ".")
spread(abl.united, ExptCell, Score)
```

Finally, the opposite of the `unite()` function is `separate()`.

```
separate(abl.united, ExptCell, c("Expt", "Cell"), sep = "\\.")
```

Note that here, if the separator is a character string, it is interpreted as a regular expression, so we have to escape out the period character. The `separate()` function can be used to split any single column which captures multiple variables.