Week_2_IntroToRmd

Seema Plaisier

06/06/22

What you need to know about R for this course

This research course will be taught using R statistal programming environment because the analysis used done in the research paper about sex differentially expressed genes in the placenta was done in R. Like any programming language, experience is the key to proficiency, but we have designed this tutorial to give you some key concepts that you will need to understand in order to do the analysis required to achieve our research aims. General R tutorials are listed at the end if you are interested in learning more.

Things we will cover in this Rmd are:

1) Basics of using R/Rmd in RStudio

a) chunks
b) code
c) line numbers
d) comments
e) libraries/packages
f) knitting

2) Data types in R

a) basic data types: string, int, float, boolean
b) sets of data points: list, vector, matrix, data frame
c) accessing data in lists (indices, subsetting)
d) iterating over lists (for loop)

3) Working with files and directories

a) working directory
b) other directories
c) file reading into variables
d) saving output files

4) Specialty data types used in this course

a) DEGList

6) Fancier plotting (ggplot)

Rmd format

This is an R Markdown document. Markdown is a simple formatting syntax that allows you to run and print R code into HTML, PDF, and MS Word documents using a tool called Knitr. For more details on using R Markdown see http://rmarkdown.rstudio.com. It contains descriptive text with specific symbols that correspond to text formmatting options and R code in sensible pieces (called "chunks"). You can run each chunk individually in RStudio to develop and correct code and then run/print the whole Rmd into a report.

Each Rmd has a header at the top. This can be used to set the author of the document and the date it was written as well as specify the output format: html_document, pdf_document, word_document are the most commonly used, see https://rmarkdown.rstudio.com/lesson-9.html for the full list.

In this Rmd, the 'date' field of the header is set to fill automatically with the date that the report is being run. This is meant to serve as a time stamp to let people know when your report was run last to help in case there are multiple versions of the code out there run at different times in different stages of development.

You will see in this document that there are headings for sections of the code marked with "##" at the beginning of the lines between the code. These allow the creation of quick-link table of contents for the report that you can print out (explained more below). The principle heading is marked with a "#" and using more hashtags indicates secondary subheadings. So if you have one heading marked with "#" and the next marked "##", the "##" heading will be accessible as a pull-down in the table of contents under the first section marked with a "#". You can play around with this if you are interested; this tutorial uses "##" for all the headings so the size of the header text is a smaller size and all headings are at equal level.

Chunks

This is the first chunk in an Rmd file. This chunk tells R to print out useful messages as it runs through the chunks of code you write. You will see them in the Render tab down below as reports are printed. Each chunk is surrounded by two sets of three single quotes (the one that goes with the ~ symbol under the escape key of your keyboard). Each chunk has a header which is surrounded in curly braces, starts with 'r' indicating that this is R code, a name which comes just after the r, and optional parameters you can pass in. Some important ones to know are "include" which when set to FALSE prevents the display of this chunk in the report, "eval" which when TRUE (by default if not included) indicates to run the code and can be set to FALSE if you want to skip think chunk when generating a report (see next section). You will see other outputting parameters to make the reports neater looking in code we will provide during this course. You will notice that the chunk is not displayed in the report.

An example of a simple R chunk:

```
print("Refreshing my memory with R")
```

```
## [1] "Refreshing my memory with R"
```

If you run specific lines of code, the output will show in the Console and Plots tables (below and right). If you run the whole chunk it will show up below the code in the Rmd tab.

Knitting a report

You should see a **Knit** button in R Studio. If you don't, copy and paste this command into the Console or press the Play button on the right side which does exactly that. 'eval' is set to FALSE because we will only need to do this once (not regularly) and 'include' is set to true so the code shows up on the report. After Knitr is installed, you should be able to click on the Knit button in RStudio and find a report document in

the same directory as this code. The document generated should include both descriptive content as well as the output of any embedded R code chunks within the document.

```
install.packages('knitr', dependencies = TRUE)
tinytex::install_tinytex()
```

Things to notice in RStudio

Look around at your coding environment in RStudio.

Notice that each line is numbered to the left. These line numbers are referenced in error messages that appear in the Console or Render tabs if something goes wrong and can be used to help figure out what happened.

Notice that this code is opened in a tab. You can open multiple Rmd files so you can easily copy code from file to file for similar applications.

The Console tab below is where you will see code that is run. The Render tab is used for this purpose if you run the code during report generation using Knitr. You can write code directly into the console to test it out and then copy and paste it into your Rmd when you have it outputing what you want in the way you like.

To the right, you should see an Environment tab. This will contain all the variables you store data into. Everything in the Data section of the Environment should be viewable by double clicking; the contents will be opened in a tab along side your Rmd tab.

This same section on the right contains a History tab which lists all the commands entered in the Console, as individual commands or by running a chunk of code. Click on any command and you can use the To **Console** button to send it to the Console tab to press enter and run or the **To Source** button to send it to wherever your cursor is currently in your Rmd.

Notice below that is a panel where you can view the Files in your current working directory (see below for how to change that). Plots is used to display any plots generated by your code (they will be displayed in the report if run with Knitr). Packages lists all the packages currently installed in your environment that you can load with the 'library' function. Help can be used to search for specific functions.

Code and comments

Inside your chunks, you are writing R code and can describe that code using comments. Comments are lines that have a hashtag symbol at the beginning. These lines are not interpretted as code and basically ignored by the R engine. To execute lines of code, you can copy and paste them into the Console, or selected using the Code menu option Run Selected Line(s) (or matching key commands that it will tell you) or use the Run button (see pull-down for specific options) both of which do exactly that.

Code consists of creating variables and passing those variables into functions. Variable names have to be a single word (no spaces or punctuation marks). They can be set to have values using the arrow assignment operator '<-' or the equal sign '='. You will see both on the internet; code from this class primarily uses the equal sign.

```
# this is a comment, indicated by a # symbol at the beginning of the line
# comments are not interpreted as code
# they are used to help explain code to others as they go through it

# below this line is real code
print ("This is code-- print is a function and this message is passed into it")
```

```
## [1] "This is code-- print is a function and this message is passed into it"
```

```
# we can use variables to store information
# here the variable called print_this_line is created and set to store specific test
# variables in R can be set using the arrow operator '<-' or an equal sign '='
print_this_line = "Hello World!"

# the value of this variable can then be passed into a function
# here we use the function 'print' that comes with R by default
print (print_this_line)
```

```
## [1] "Hello World!"
```

```
# you can change the value of variables
print_this_line = "And now print this!"

# and then pass those into a function
print (print_this_line)
```

```
## [1] "And now print this!"
```

Basic data types and structures in R

The basic data types in R that you will work with in this course are: 1) character: single character such as "a" or string of characters such as "abc" 2) double: numbers including whole numbers like 1 and numbers with fractions/decimals like 1.5 3) logical: boolean values TRUE or FALSE

The basic data structures you will work with in this course are: 1) vector: 1-dimensional collection of elements of one type, usually character, logical, integer or numeric 2) matrix: multi-dimensional collection of elements of one type (specify num rows and num cols) 3) list: multi-dimensional collection of elements that can be multiple data types 4) data frame: special type of list where every element has the same length ("rectangular" list) 5) factors: data objects used to categorize and order repeated elements in a vector or list

For multi-dimensional data structures, you can use indices to access and modify specific elements of the list. Indices are noted with square braces, [ ]. Indices in R start at 1 (first value in list/vector) and end at the length of the list/vector (last index of a list of length 3 is 3 which holds the last value in the list/vector).

The basic control structures you will work with in this course are: 1) conditional: if, else statements to test if something is true before doing something 2) loops: for loops to run through elements of a list

TutorialsPoint is a great resource for more information on these topics: https://www.tutorialspoint.com/r/r_overview.htm Click on the topic on the left side of the page for whatever you'd like to see more examples of.

```
s = "this_is_a_string"  # set variable called s to a character string "this_is_a_string"
typeof(s)  # print out what type of data structure s holds
```

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

```
print(s) # prints the value stored in the variable s
```

```
## [1] "this_is_a_string"
```

```r
n = 1 # set variable n to 1
print(n) # show the current value of the variable
```

```
## [1] 1
```

```r
typeof(n) # displays data type of current value of variable n
```

```
## [1] "double"
```

```r
n = 1.5 # set same variable n to a new value overwriting the old one
print(n) # look to see that it is a new value of 1.5
```

```
## [1] 1.5
```

```r
typeof(n) # look to see what data type it is now with the new value
```

```
## [1] "double"
```

```r
b = TRUE # set variable b to a logical state TRUE
typeof(b) # display data type of the value of variable b
```

```
## [1] "logical"
```

```r
if (n == 1) { # use a conditional statement to see if the variable n is currently set to 1
  b = TRUE  # if it is, set variable b equal to the logical value of TRUE
  print (b) # print the value of b
} else { # if the condition in the if statement above is not true, do the following code
  b = FALSE # set the variable b to logical value of FALSE
  print (b) # print the value of b
}
```

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

```r
# this code does something very similar to the if statement above
#   but in far fewer lines of code
print (n == 1) # print whether it is true or false that n is equal to 1
```

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

```r
# this shows that you can do the same thing in multiple ways
#   as long as you think through what you want to do
#   and know the coding tools available in R to do it

v = c("this","is","a","vector","of","character","strings")  # set variable v to a vector of strings
typeof(v) # shows the data type of a vector is the data type of its contents
```

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

```r
print(v) # print the contents of v
```

```
## [1] "this"      "is"        "a"         "vector"    "of"        "character"
## [7] "strings"
```

```r
length(v) # print the length of vector v
```

```
## [1] 7
```

```r
v2 = c(1,2,3.4,5.1,8,10) # set the variable v2 to a vector of numbers
typeof(v2) # shows the data type of the vector is the data type of its contents
```

```
## [1] "double"
```

```r
length(v2) # show the length of variable v2
```

```
## [1] 6
```

```r
l = list("here","we","have","a","list","with",7,"elements") # set variable l to list of strings and num
typeof(l) # shows the data type of a list is 'list' as it can have many data types inside
```

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

```r
print(l) # print contents of l
```

```
## [[1]]
## [1] "here"
##
## [[2]]
## [1] "we"
##
## [[3]]
## [1] "have"
##
## [[4]]
## [1] "a"
##
## [[5]]
## [1] "list"
##
## [[6]]
## [1] "with"
##
## [[7]]
## [1] 7
##
## [[8]]
## [1] "elements"
```

```r
length(l) # print length of list l
```

```
## [1] 8
```

```r
m = matrix(v2,nrow=2,ncol=3) # set variable m to values of vector v2 filling rows and columns
print(m) # print contents of variable m
```

```
##      [,1] [,2] [,3]
## [1,]    1  3.4    8
## [2,]    2  5.1   10
```

```r
# create data.frame d containing the data from matrix m
# dimensions of m conserved in data frame d
# can set the row names at the same time as creating the data frame
d = data.frame(mat = m, row.names = c("row1","row2"))
# can set the names outside of when you are creating the data frame too
colnames(d) = c("col1","col2","col3")
print(d) # print contents of d
```

```
##      col1 col2 col3
## row1    1  3.4    8
## row2    2  5.1   10
```

```r
dim(d) # show the dimensions of the data frame, length and width for 2 dimensional
```

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

```r
# can also set a data frame to hold differnt types of data, each given a name
emp.data <- data.frame(
   emp_id = c (1:5),
   emp_name = c("Rick","Dan","Michelle","Ryan","Gary"),
   salary = c(623.3,515.2,611.0,729.0,843.25)
)
print(emp.data) # show the contents of employee data frame
```

```
##   emp_id emp_name salary
## 1      1     Rick 623.30
## 2      2      Dan 515.20
## 3      3 Michelle 611.00
## 4      4     Ryan 729.00
## 5      5     Gary 843.25
```

```r
str(emp.data) # show the elements of the data frame with the first few entries
```

```
## 'data.frame':    5 obs. of  3 variables:
##  $ emp_id  : int  1 2 3 4 5
##  $ emp_name: chr  "Rick" "Dan" "Michelle" "Ryan" ...
##  $ salary  : num  623 515 611 729 843
```

```r
summary(emp.data) # show the range of each element of the data frame
```

```
##      emp_id    emp_name            salary
##  Min.   :1   Length:5           Min.   :515.2
##  1st Qu.:2   Class :character   1st Qu.:611.0
##  Median :3   Mode  :character   Median :623.3
##  Mean   :3                      Mean   :664.4
##  3rd Qu.:4                      3rd Qu.:729.0
##  Max.   :5                      Max.   :843.2
```

```r
# different ways to access the data in a data frame

# can use the $ symbol to get all of a specific column/element
print(emp.data$emp_name)
```

```
## [1] "Rick"     "Dan"      "Michelle" "Ryan"     "Gary"
```

```r
# can use a conditional to set which row you are interested in
# indices are given as column,row
#    so indicating the column, comma, no row specified gives you the whole row
emp.data[emp.data$emp_id == 1,]
```

```
##   emp_id emp_name salary
## 1      1     Rick  623.3
```

```r
# if the row has multiple entries, you can use the index number to access a specific entry
emp.data[emp.data$emp_id == 1,2]
```

```
## [1] "Rick"
```

```r
# you can also use the row name
emp.data[emp.data$emp_id == 1,"emp_name"]
```

```
## [1] "Rick"
```

```r
# you can use index number for both row and column
emp.data[1,2]
```

```
## [1] "Rick"
```

```r
# you can assign a specific position in the data frame to a value as long as you access it correctly
emp.data[1,2] = "Morty"

# you can print the contents after making changes to see if everything looks right
print(emp.data)
```

```
##   emp_id emp_name salary
## 1      1    Morty 623.30
## 2      2      Dan 515.20
## 3      3 Michelle 611.00
## 4      4     Ryan 729.00
## 5      5     Gary 843.25
```

```r
# if you make a mistake, you can also go back and run the original
# statement that filled the data frame to reset the data frame variable
# to its original contents and then try again

# this for loop runs through all the values of vector v2 assigning to the variable 'value'
for (value in v2) {
  print(value) # print the value of the variable 'value'
}
```

```
## [1] 1
## [1] 2
## [1] 3.4
## [1] 5.1
## [1] 8
## [1] 10
```

```r
# this for loop makes a list starting at 1 and continuing through the length of vector v2
# then the variable 'i' is filled with the list generated
for (i in 1:length(v2)) {
  # can use 'i' to access a specific index of the vector v2
  print (paste0(i,",",v2[i])) # paste0 function makes a string of what you pass it
}
```

```
## [1] "1,1"
## [1] "2,2"
## [1] "3,3.4"
## [1] "4,5.1"
## [1] "5,8"
## [1] "6,10"
```

Installing packages

One of the main reasons for learning to use R is that there is an enormous collection of prewritten code written in R that you can use to do analysis. Functions are included in bundles called packages and made available for use in the program by using the 'library' command in R. This included code will extend the functionality of R beyond the basic structures. In order to include a library, it has to be installed (code has to be downloaded to your local R environment). In this next chunk, the 'require' function is used to check to see if the package 'tidyverse' is currently installed, returning TRUE if it is and FALSE if it is not. If 'require' returns FALSE, the '!' ("not") operator returns the opposite value of TRUE and calls the 'install.packages' function to install the package 'tidyverse'. For packages in the large biology focused set of packages Bioconductor, there is a special install function called BiocManager::install.

```r
if(!require(tidyverse)){
    install.packages("tidyverse")
}
```

```
## Loading required package: tidyverse
```

```
## -- Attaching packages --------------------------------------- tidyverse 1.3.1 --
```

```
## v ggplot2 3.3.5     v purrr   0.3.4
## v tibble  3.1.6     v dplyr   1.0.9
## v tidyr   1.2.0     v stringr 1.4.0
## v readr   2.1.2     v forcats 0.5.1

## -- Conflicts ---------------------------------------- tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

```
if(!require(BiocManager)){
    install.packages("BiocManager")
}
```

```
## Loading required package: BiocManager

## Bioconductor version '3.14' is out-of-date; the current release version '3.15'
##   is available with R version '4.2'; see https://bioconductor.org/install
```

```
if(!require(ggplot2)){
    BiocManager::install("ggplot2")
}
if(!require(edgeR)){
  BiocManager::install("edgeR")
}
```

```
## Loading required package: edgeR

## Loading required package: limma
```

```
library(tidyverse)
library(BiocManager)
library(ggplot2)
library(edgeR)
```

Custom data types: DGEList

Using a combination of basic data types in R, specific packages use specialized data types to pass into functions. The construction of these data types should be covered in the documentation for the package. As an example, we will use the DGEList data structure from the package edgeR which we included in the last chunk. Here we will go over it in detail so you understand how to work with it since will use it to identify differentially expressed genes.

```
# ? or ?? can be used to look up documentation for classes
# and functions
`?`(`?`(edgeR::DGEList))
# first entry in the Help tab or browser tab is
# edgeR::DGEList-class, click on it to read

# DGEList objects contains a matrix of read counts and
# other information that we wil need to normalize and
```

```r
# process the read count data

# You will recall that read counts are the output from
# programs like featureCounts, which take the sequences
# read that align to particular genes and count them up as
# a measure of how much those genes are expressed


# there are two main element of a DGEList object: 1) a
# matrix of counts 2) a data frame of sample information:
# row for each sample, columns for groups those samples
# belong to, lib.size (number of genes that have counts),
# and and normalization factors (which start as all 1, but
# we can fill in using the appropriate functions)

# in this example, we are going to create a matrix of
# random numbers to stand in for counts data and numbers 1
# and 2 to represent groups for our fake samples
fake_counts = matrix(rnbinom(10000, mu = 5, size = 2), ncol = 4)
fake_groups = rep(1:2, each = 2)

# now we are going to construct a DGEList object with our
# fake counts and groups as features and set the lib.size
# feature to the sum of all the counts in each column
d <- DGEList(counts = fake_counts, group = fake_groups, lib.size = colSums(fake_counts))
dim(d)  # will show you the dimensions of the counts matrix
```

```
## [1] 2500    4
```

```r
colnames(d)  # will show you the sample names (filled in by default since we did not specify here)
```

```
## [1] "Sample1" "Sample2" "Sample3" "Sample4"
```

```r
d$samples  # will show you the sample information data.frame created from the parameters we passed in
```

```
##         group lib.size norm.factors
## Sample1     1    12089            1
## Sample2     1    12545            1
## Sample3     2    12245            1
## Sample4     2    12317            1
```

```r
# you should see that the norm.factors are 1 by default we
# will use the calcNormFactors function to get factors that
# we can multiply by the counts to keep the range of counts
# comparable between samples

# in our real data, we will also pass in some of the
# optional elements such as genes (name of rows)

# once all this data is filled in, you can pass the DGEList
# variable d into functions from the edgeR package that are
```

```
# designed to take DGEList objects as input parameters,
# keeping the code short and tidy using this data type
# allows the functions in edgeR to assume the name of the
# elements to be counts, samples, etc
```

Set working directory and data directories

Your working directory is where all your output files will be stored by default. You can set it by copying the path of the directory from the explorer windows. Make sure you change the slashes to forward slashes '/' if needed.

The 'setwd' function sets your working directory to the path you specify. The 'list.files' and 'list.dirs' functions can be used to print what's inside that directory.

```
# store the path to the working directory in a variable
# make sure to include the / at the end of the directory path if you plan to put it together with the p
working_directory = "/scratch/splaisie/test_CURE"

# set the working directory to where you want your output to go
setwd(working_directory)

# see what's currently in your working directory
# files:
list.files(working_directory)
```

```
## [1] "emp_salary_plot.pdf"    "subset.csv"             "subset.txt"
## [4] "Week_2_IntroToRmd.docx" "Week_2_IntroToRmd.html" "Week_2_IntroToRmd.log"
## [7] "Week_2_IntroToRmd.Rmd"  "Week_2_IntroToRmd.tex"
```

```
# directories:
list.dirs(working_directory)
```

```
## [1] "/scratch/splaisie/test_CURE"
```

```
# set the path to other directories you might need to reference
untrimmed_directory = "C:/Users/splaisie/Dropbox (ASU)/GenomicsCURE/Week 3/"
print(untrimmed_directory)
```

```
## [1] "C:/Users/splaisie/Dropbox (ASU)/GenomicsCURE/Week 3/"
```

```
# don't need to set it to working directory

# can instead use paste0 command to add the file path of a directory to the file name
# make sure you have the / at the end to make sure the file name is not added to the deepest directory
file_path = paste0(untrimmed_directory, "DE_Pipeline_UntrimmedData.Rmd")
print (file_path)
```

```
## [1] "C:/Users/splaisie/Dropbox (ASU)/GenomicsCURE/Week 3/DE_Pipeline_UntrimmedData.Rmd"
```

Reading data files

Reading data into variables is a very important task in R. For most scientific problems, you will read a table of data commonly separated by commas or tabs. The 'read.csv' function can be used to read tabular data from files; the delimiter is set to be a comma by default in this function. If you have tab-delimited data, you can set the delimiter to a tab. Running these commands will put variables containing your data in the workspace.

```
variants = read.csv("subset.csv")

variants2 = read.csv("subset.txt", sep = "\t")
```

Ways to look at the data

Once data is read in from a file (or created with code, see below), you can see them in the Environment tab and should be able to check if everything worked by clicking on the variable name. You can also use the 'head' function to see the top rows of the data table or the 'summary' function to see a summary of the counts and ranges of the data variable or the 'str' function to show the structure of the data.

```
head(variants)
```

```
##   X.1 X X.3 X.2       ALT
## 1   1 1   1   1         G
## 2   2 2   2   2         T
## 3   3 3   3   3         T
## 4   4 4   4   4 CTTTTTTTT
## 5   5 5   5   5    CCGCGC
## 6   6 6   6   6         T
```

```
summary(variants)
```

```
##       X.1             X             X.3           X.2          ALT
##  Min.   :  1   Min.   :  1   Min.   :  1   Min.   :  1   Length:801
##  1st Qu.:201   1st Qu.:201   1st Qu.:201   1st Qu.:201   Class :character
##  Median :401   Median :401   Median :401   Median :401   Mode  :character
##  Mean   :401   Mean   :401   Mean   :401   Mean   :401
##  3rd Qu.:601   3rd Qu.:601   3rd Qu.:601   3rd Qu.:601
##  Max.   :801   Max.   :801   Max.   :801   Max.   :801
```

```
str(variants)
```

```
## 'data.frame':    801 obs. of  5 variables:
##  $ X.1: int  1 2 3 4 5 6 7 8 9 10 ...
##  $ X  : int  1 2 3 4 5 6 7 8 9 10 ...
##  $ X.3: int  1 2 3 4 5 6 7 8 9 10 ...
##  $ X.2: int  1 2 3 4 5 6 7 8 9 10 ...
##  $ ALT: chr  "G" "T" "T" "CTTTTTTTT" ...
```

Subset the data

One major task in analyzing genome or transcriptome level data is to select specific information of interest for your research aims. In this example, we are taking a subset of the variants data frame loaded in the previous chunk. We are selecting specific columns containing specific information, alternative alleles, which are points in the sequence that are different from the sequence in the reference genome. We are dividing them up by what nucleotide is actually present and using basic plotting tools in R to make a bar graph so we can see which nucleotide is most often present in that position as an alternative allele. Alternative alleles or SNPs (single nucleotide polymorphisms) are the first line of suspects for variations that cause different phenotypes in a cell or tissue.

```
# select specific columns and make a new data frame
subset = data.frame(variants[,c(1:3,5)])

# look at our data to make sure we got the columns we wanted
head(subset) # look at the top rows
```

```
##   X.1 X X.3       ALT
## 1   1 1   1         G
## 2   2 2   2         T
## 3   3 3   3         T
## 4   4 4   4 CTTTTTTTT
## 5   5 5   5    CCGCGC
## 6   6 6   6         T
```

```
str(subset) # look at the components and the first few elements of each
```

```
## 'data.frame':    801 obs. of  4 variables:
##  $ X.1: int  1 2 3 4 5 6 7 8 9 10 ...
##  $ X  : int  1 2 3 4 5 6 7 8 9 10 ...
##  $ X.3: int  1 2 3 4 5 6 7 8 9 10 ...
##  $ ALT: chr  "G" "T" "T" "CTTTTTTTT" ...
```

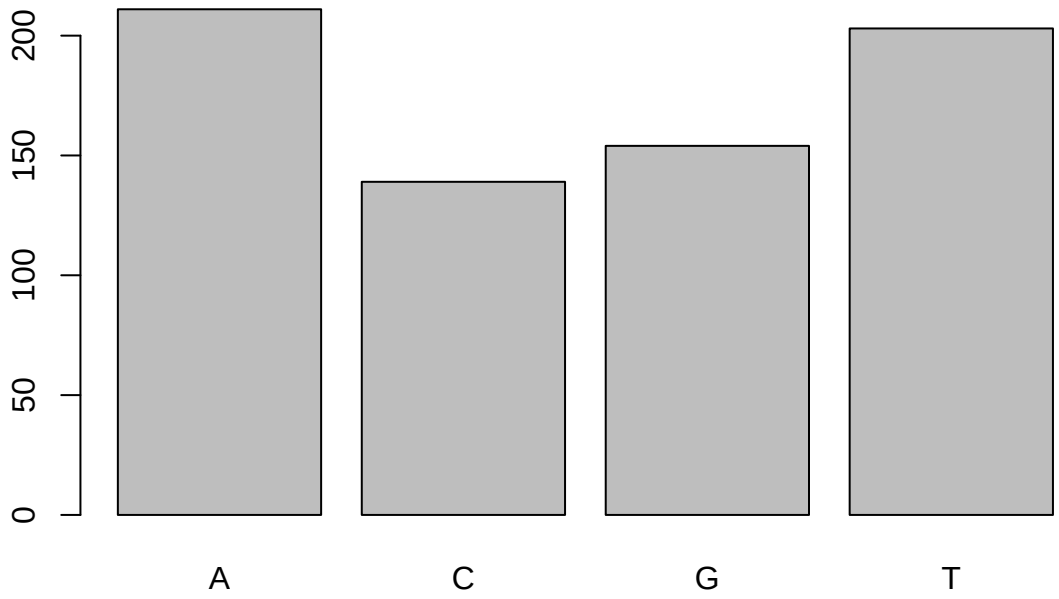
```
alt_alleles = subset$ALT # select 1 specific column of the data

# divide up the data
# use a conditional statement to determine the indices of the data frame
#   are have alleles that are a specific nucleotide
#  those indices are used to subset the column you pulled
# put the 4 subsets of data together in a list with the c() command
snps <- c(alt_alleles[alt_alleles=="A"],
  alt_alleles[alt_alleles=="T"],
  alt_alleles[alt_alleles=="G"],
  alt_alleles[alt_alleles=="C"])

factor_snps <- factor(snps) # tell R to treat this data as categories
str(factor_snps)
```

```
##  Factor w/ 4 levels "A","C","G","T": 1 1 1 1 1 1 1 1 1 1 ...
```

```r
summary(factor_snps) # take a final look at what you got out to make sure it looks right
```

```
##   A   C   G   T
## 211 139 154 203
```

```r
plot(factor_snps) # basic plot of the number of each nucleotide
```



Save data to a file

The 'write.csv' function writes data to a file. The file will be saved in the working directory unless otherwise specified.

```r
write.csv(subset, file = "subset.csv")
```

Fancier Data Plotting with ggplot

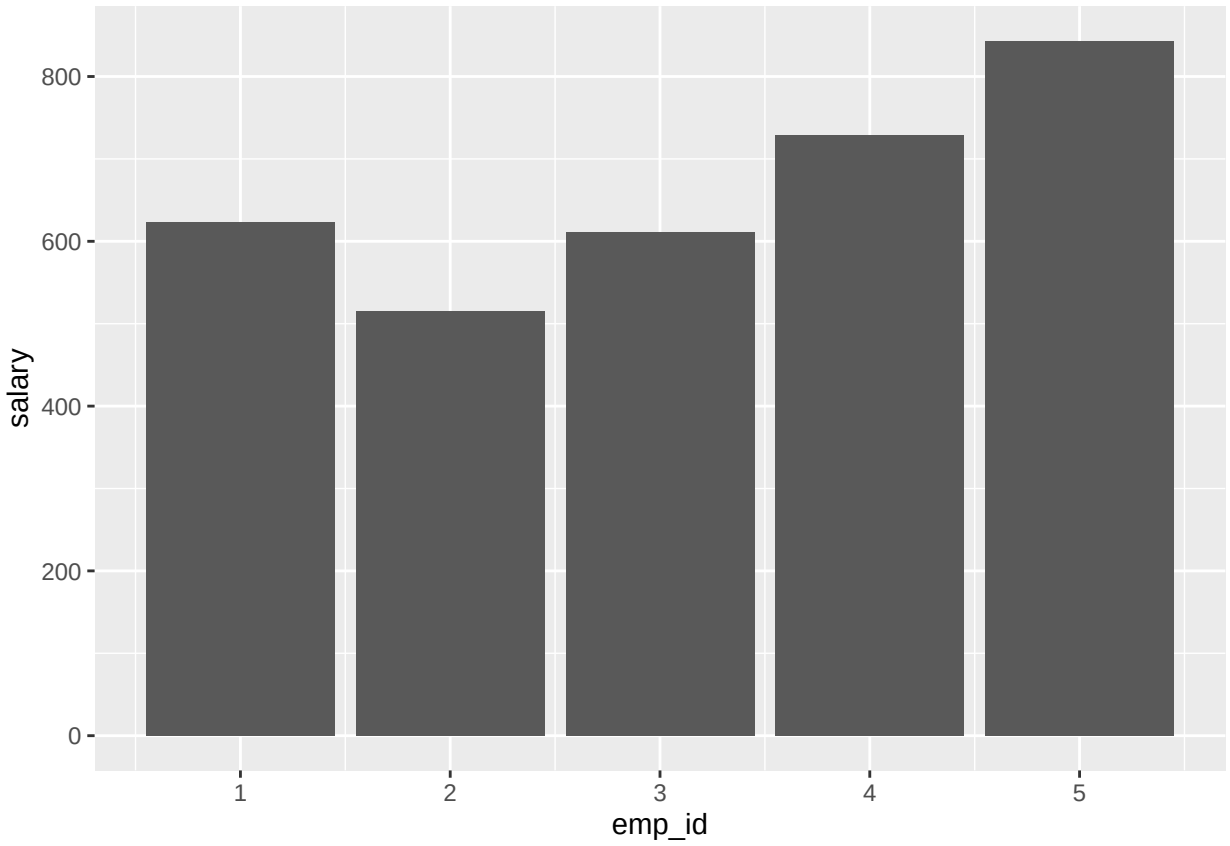
In addition to basic plotting functions in R, there are a lot of awesome packages for making fancier plots. One of the most widely used is ggplot or newer version ggplot2.

Here is a gallery of examples to give you a taste of how many beautiful types of figures/plots you can make: https://r-graph-gallery.com/

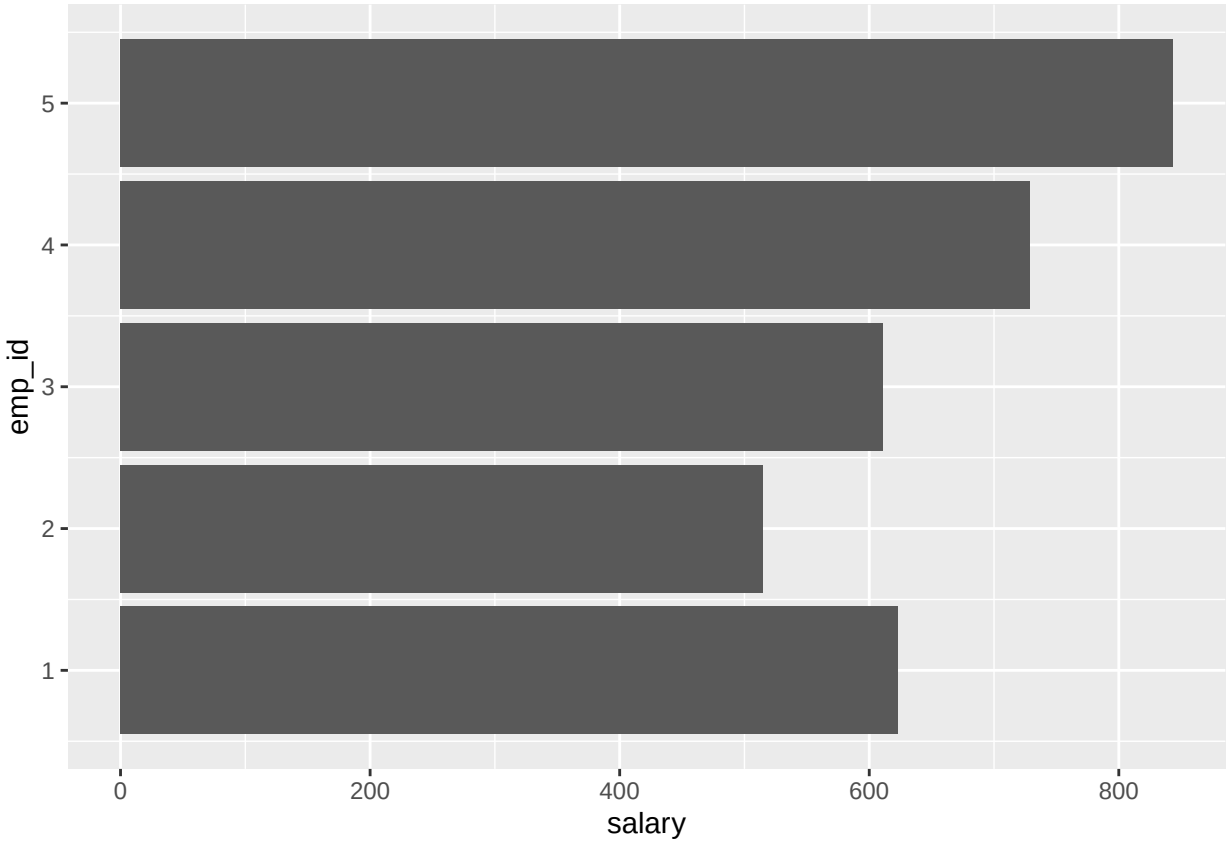
The way that ggplot works is that you set the data you are wanting to plot to a ggplot variable and then add in all the features you would like to use to customize your plot. There are a ton of options for plots and customization, you just have to go step by step figuring out how to get what you want.

In this introductory example, we are adding features one at a time so you can get a better idea of what the different modifications do. If you look at ggplot code other people have written, you will often see all the modifications added together in one line before plotting. Either way works.

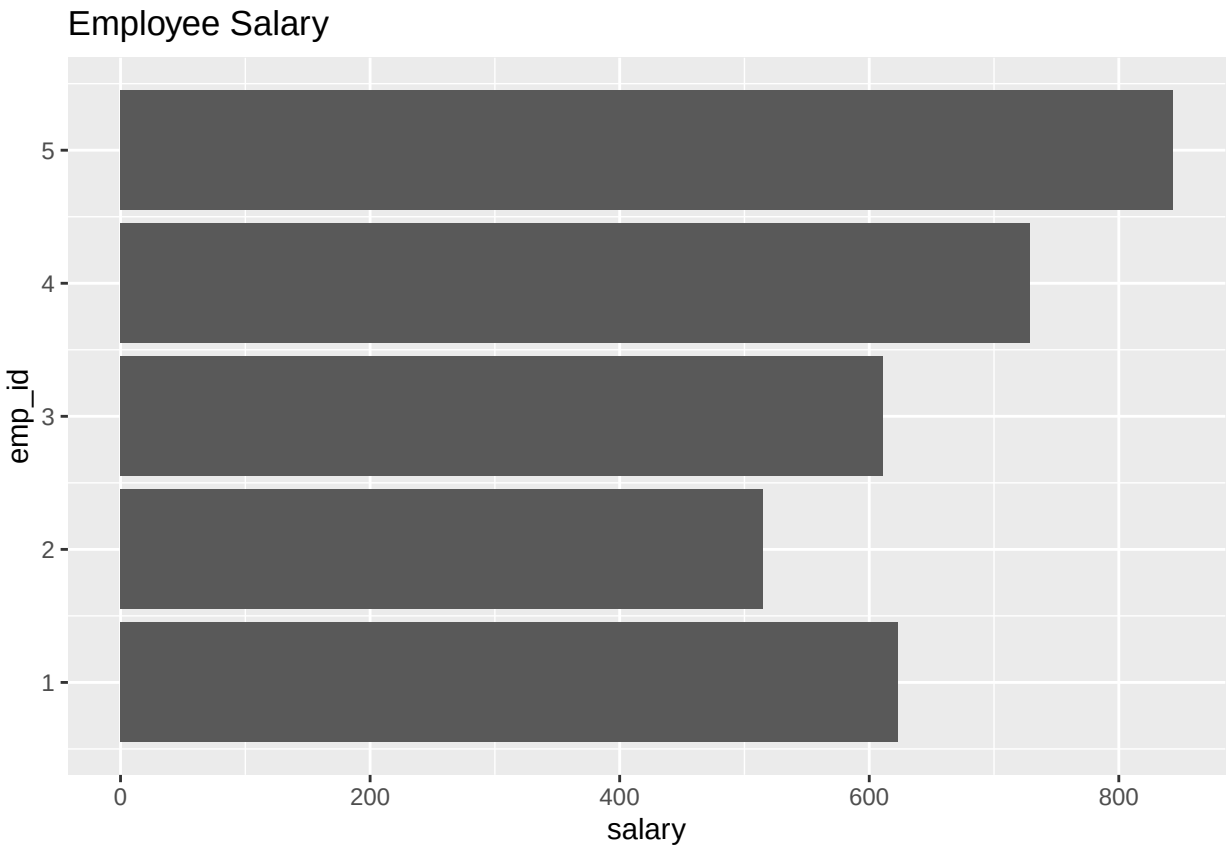
```
# have to call library(ggplot2) which we did above

# Initialize plot variable with data and basic type
p = ggplot(data = emp.data, aes(x = emp_id, y = salary)) + geom_bar(stat = "identity")

# plot the ggplot variable p
plot(p)
```



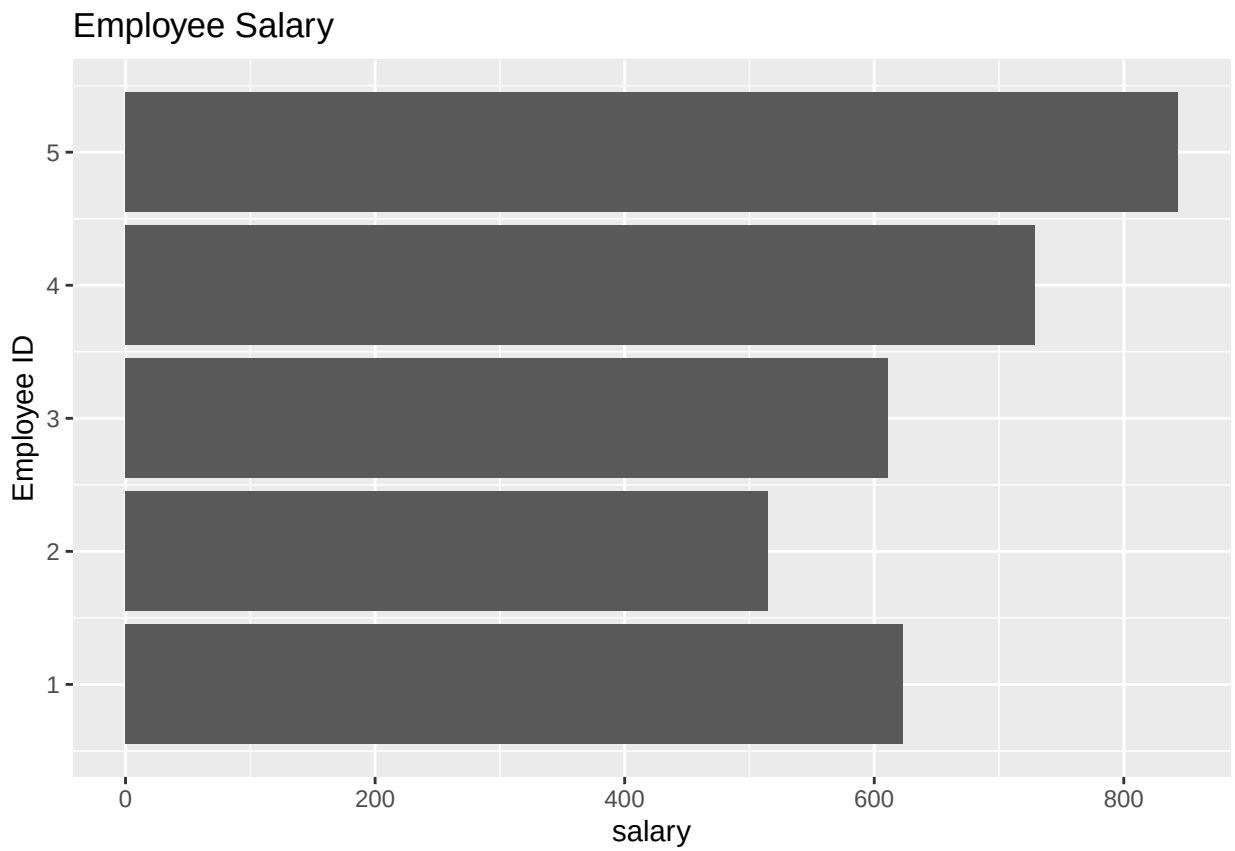

```
# let's say we wanted the plot to be going left to right
p = p + coord_flip()
plot(p)
```



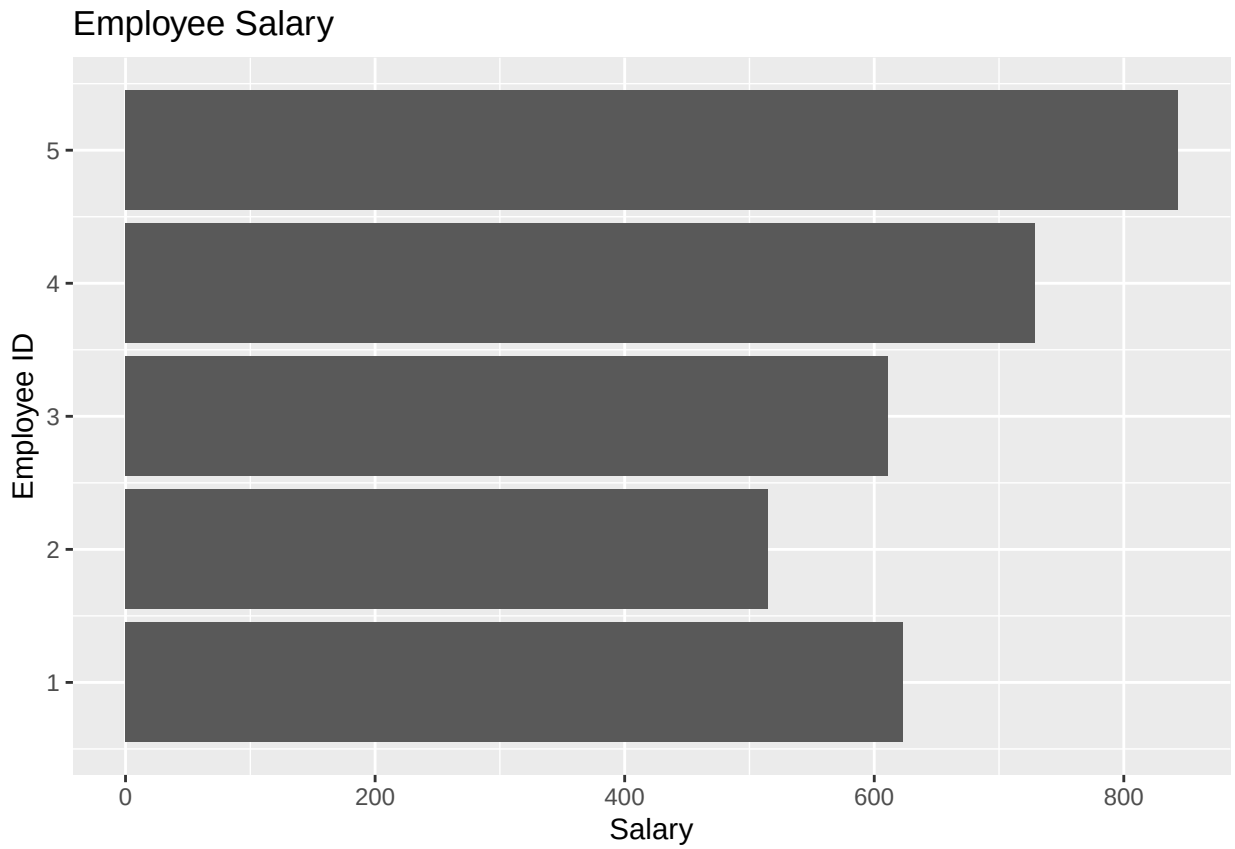

```
# add a title to the plot
p = p + ggtitle("Employee Salary")
plot(p)
```



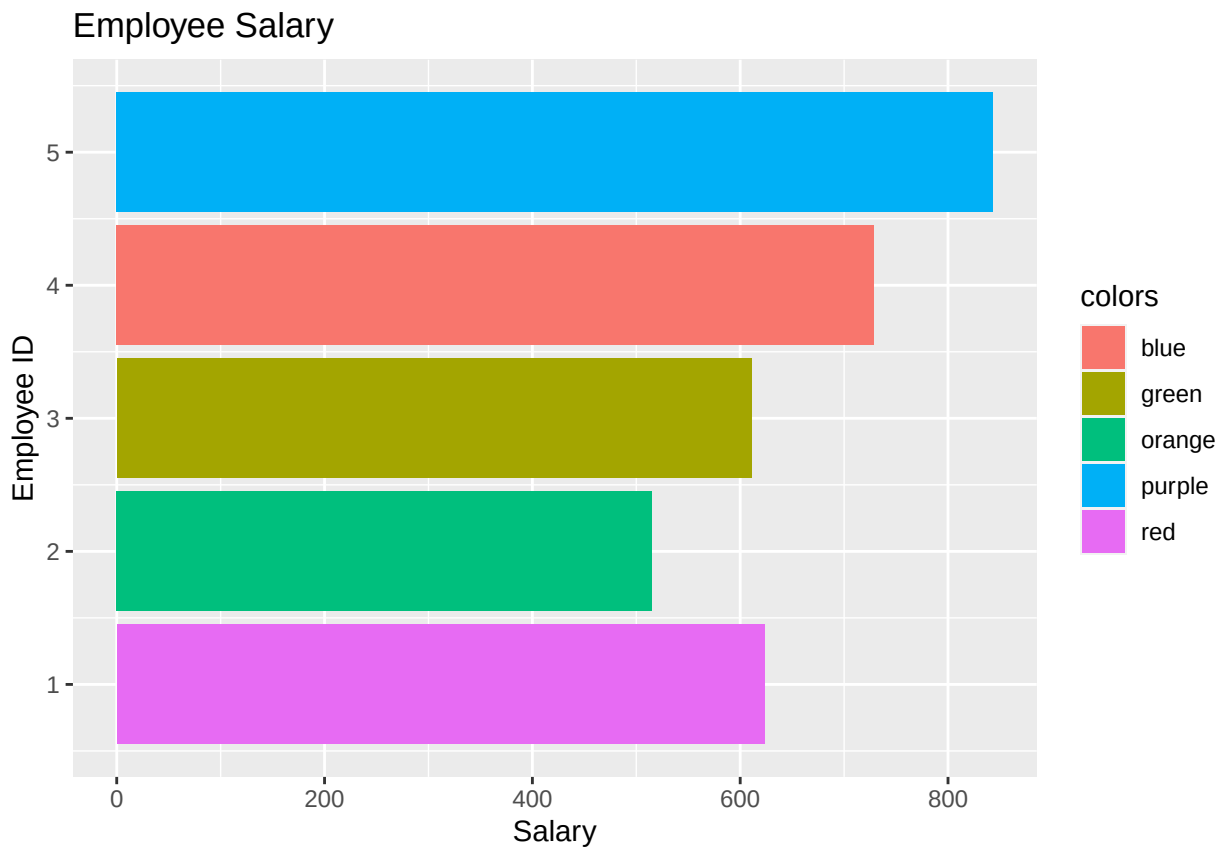

```
# change the axis labels
p = p + xlab("Employee ID")
plot(p)
```



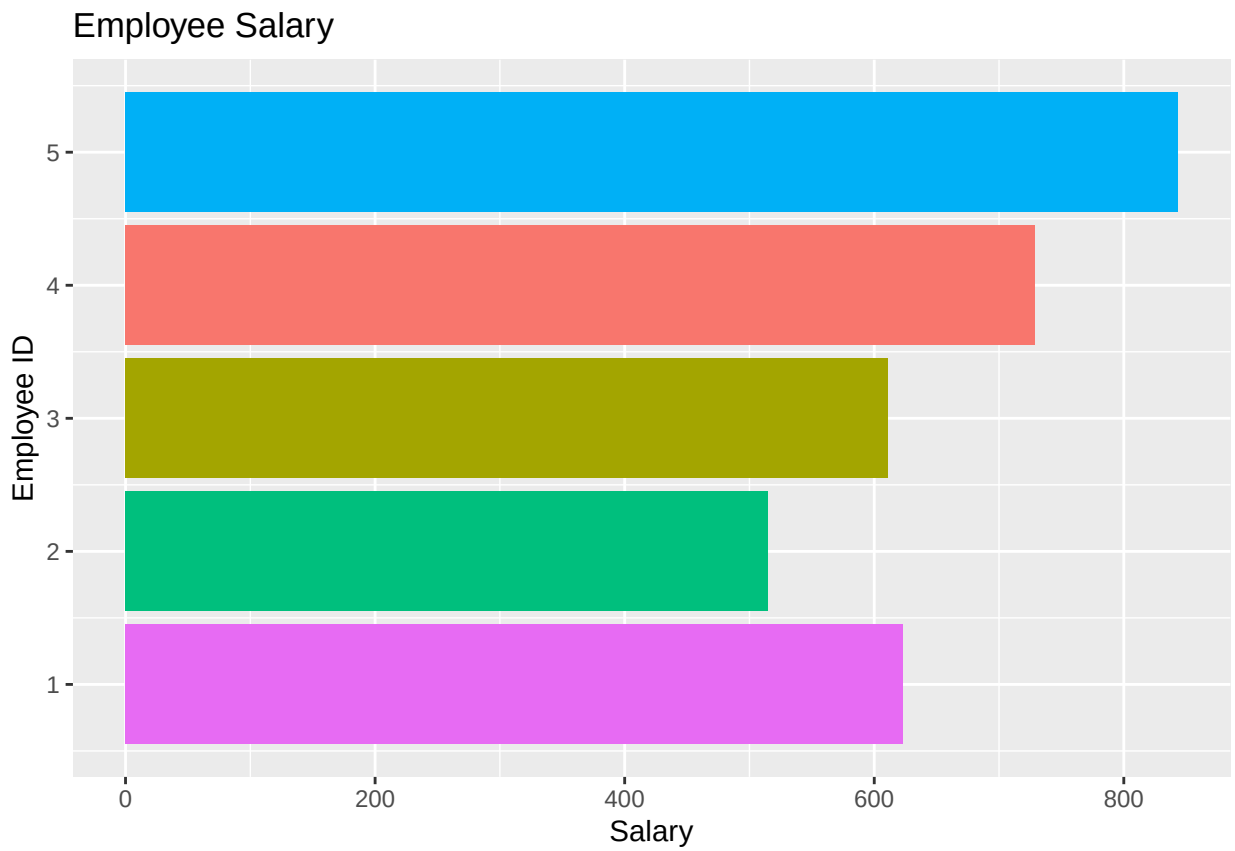

```
p = p + ylab("Salary")
plot(p)
```




```
# if you want to use specific colors, for the bars, you can
# use the fill parameter when setting your ggplot variable
# setting the colors automatically adds a legend, I am
# showing how to remove it in our case here I will show you
# how to use ggplot all in one line

# specific colors for the bars
colors = c("red", "orange", "green", "blue", "purple")

# construct ggplot object all in one line
p = ggplot(data = emp.data, aes(x = emp_id, y = salary, fill = colors)) +
    geom_bar(stat = "identity") + coord_flip() + ggtitle("Employee Salary") +
    xlab("Employee ID") + ylab("Salary")
plot(p)
```




```
# remove the legend because it's not saying anything
# important
p = ggplot(data = emp.data, aes(x = emp_id, y = salary, fill = colors)) +
    geom_bar(stat = "identity") + coord_flip() + ggtitle("Employee Salary") +
    xlab("Employee ID") + ylab("Salary") + theme(legend.position = "none")
plot(p)
```



Save plot

To save a figure, we use the 'ggsave' function. The type of file saved is determined by the file type extension you give. That is, if you give a file name ending in ".pdf", your output file will be saved as a PDF with the plot in it. If you give it a file name ending in ".png", you will get the figure in a PNG image file. The output file will be saved in your working directory by default.

```
ggsave("emp_salary_plot.pdf", plot = p)
```

```
## Saving 6.5 x 4.5 in image
```

```
# check to make sure you can find your output file after it is saved
# when you Knit a report, this save function will be run, so you
#    should see your output files and report in your working directory
```

List all the packages used for future reference

It's good practice to include this chunk at the end of all of your Rmd files. It lists out all the packages used in when this code was run including all the version numbers. This type of information should be included in any publications or technical reports to ensure reproducibility. It can explain differences in output when installing packages after some time has passed and can be used to address problems involving specific versions of packages are listed as dependencies for other packages.

```
sessionInfo()
```

```
## R version 4.1.2 (2021-11-01)
## Platform: x86_64-pc-linux-gnu (64-bit)
## Running under: CentOS Linux 7 (Core)
##
## Matrix products: default
## BLAS/LAPACK: /packages/7x/openblas/3.10.0-gcc-9.2.0/lib/libopenblas_haswellp-r0.3.10.dev.so
##
## locale:
##  [1] LC_CTYPE=en_US.UTF-8       LC_NUMERIC=C
##  [3] LC_TIME=en_US.UTF-8        LC_COLLATE=en_US.UTF-8
##  [5] LC_MONETARY=en_US.UTF-8    LC_MESSAGES=en_US.UTF-8
##  [7] LC_PAPER=en_US.UTF-8       LC_NAME=C
##  [9] LC_ADDRESS=C               LC_TELEPHONE=C
## [11] LC_MEASUREMENT=en_US.UTF-8 LC_IDENTIFICATION=C
##
## attached base packages:
## [1] stats     graphics  grDevices utils     datasets  methods   base
##
## other attached packages:
##  [1] edgeR_3.36.0        limma_3.50.3        BiocManager_1.30.18
##  [4] forcats_0.5.1       stringr_1.4.0       dplyr_1.0.9
##  [7] purrr_0.3.4         readr_2.1.2         tidyr_1.2.0
## [10] tibble_3.1.6        ggplot2_3.3.5       tidyverse_1.3.1
##
## loaded via a namespace (and not attached):
##  [1] Rcpp_1.0.8.3     locfit_1.5-9.5   lubridate_1.8.0  lattice_0.20-45
##  [5] assertthat_0.2.1 digest_0.6.29    utf8_1.2.2       R6_2.5.1
##  [9] cellranger_1.1.0 backports_1.4.1  reprex_2.0.1     evaluate_0.15
## [13] httr_1.4.2       highr_0.9        pillar_1.7.0     rlang_1.0.2
## [17] readxl_1.4.0     rstudioapi_0.13  rmarkdown_2.14   labeling_0.4.2
## [21] munsell_0.5.0    broom_0.7.12     compiler_4.1.2   modelr_0.1.8
## [25] xfun_0.31        pkgconfig_2.0.3  htmltools_0.5.2  tidyselect_1.1.2
## [29] fansi_1.0.3      crayon_1.5.1     tzdb_0.3.0       dbplyr_2.2.0
## [33] withr_2.5.0      grid_4.1.2       jsonlite_1.8.0   gtable_0.3.0
## [37] lifecycle_1.0.1  DBI_1.1.2        magrittr_2.0.2   formatR_1.12
## [41] scales_1.1.1     cli_3.3.0        stringi_1.7.6    farver_2.1.0
## [45] fs_1.5.2         xml2_1.3.3       ellipsis_0.3.2   generics_0.1.2
## [49] vctrs_0.4.1      tools_4.1.2      glue_1.6.2       hms_1.1.1
## [53] fastmap_1.1.0    yaml_2.2.2       colorspace_2.0-2 rvest_1.0.2
## [57] knitr_1.37       haven_2.5.0
```