Intermediate R Workshop Notes

Hena R. Ramay

2020-05-23

Contents

1 Introduction **5**
- 1.1 Workshop Schedule . . . 5
- 1.2 Important links . . . 6
- 1.3 Setting up a project . . . 6
- 1.4 Learning Objectives . . . 6
- 1.5 Finding your location . . . 6
- 1.6 Downloading files from the command line . . . 7
- 1.7 Organizing a larger project . . . 8
- 1.8 Create a README file . . . 8

2 Conditionals and Loops **9**
- 2.1 Setup . . . 9
- 2.2 Conditional statements . . . 9
- 2.3 Loops . . . 11

3 Functions **17**
- 3.1 Parts of a function . . . 17
- 3.2 The principle of encapsulation . . . 18
- 3.3 The return statement . . . 19

4 The apply family **21**
- 4.1 apply() . . . 21
- 4.2 lapply() . . . 22
- 4.3 sapply() . . . 25
- 4.4 vapply() . . . 27
- 4.5 Review of ggplot2 basics . . . 28
- 4.6 Statistics . . . 31
- 4.7 Scales . . . 34
- 4.8 Faceting . . . 39
- 4.9 Themes . . . 39
- 4.10 Color palettes . . . 40
- 4.11 Multiple plots . . . 41
- 4.12 Additional Resources . . . 42

5 Rmarkdown **43**
5.1 Set up new R Markdown file 43
5.2 Knitr Chunk Options 48

6 Project **51**
6.1 Setup 51
6.2 Task 1 51
6.3 Task 2 51
6.4 Task3 52

7 Interesting Packages **53**

Chapter 1

Introduction

This notebook is created so that the particpants of the workshop can follow along with the instructors. We will constantly update this notebook during the workshop to include answers to the questions asked during the workshop. So Please ask a lot of questions!!

1.1 Workshop Schedule

We will try to cover the following material in the course:

Day 1

Time	Topic
9:00	Introductions and workshop overview
9:30	Conditionals and Loops
10:45	Functions
13:00	Apply family
14:00	Advanced plotting ggplot2

Day 2

Time	Topic
9:00	R markdown basics
10:45	Knitr chunks
13:00	Rmarkdown themes, adding html and custom css styling

Time	Topic
14:00	A short project to bring it all together!

1.2 Important links

Overview and setup

Lets get started!

1.3 Setting up a project

1.4 Learning Objectives

- Create directories and files
- Understand the difference between absolute and relative paths

Before we start using R, we will first review the basics of the Unix shell by setting up our project. Open the terminal in OS X or Linux, or Git Bash in Windows.

Our project will explore the citations and alternative metrics (altmetrics) for articles published in the PLOS family of journals between 2003 and 2010. The data set was compiled by Priem et al. 2012 (publication, code).

First create a new directory to store the project files called `altmetrics` and then change to that directory.

```
mkdir altmetrics
cd altmetrics
```

Repeat this process to create a subdirectory to store the data files.

```
mkdir data
cd data
```

1.5 Finding your location

Recall that you can always determine where you are by running the command `pwd`, which stands for "print working directory". Also, if you run `cd` with no arguments, it takes you to your home directory.

Download the two data files using the links below. Save them to the data subdirectory.

- counts-raw
- counts-norm

1.6 Downloading files from the command line

When possible, it is best to download files from the command line because that makes the analysis more reproducible. Different systems have different tools installed, which is why we manually downloaded the files for this lesson. Some options include `wget`, `curl`, and `rsync`. As an example, here is how to perform the download using `wget`.

```
wget https://raw.githubusercontent.com/jdblischak/r-intermediate-altmetrics/gh-pages/data/counts-
wget https://raw.githubusercontent.com/jdblischak/r-intermediate-altmetrics/gh-pages/data/counts-
```

If `wget` isn't installed on your machine, try `curl`:

```
curl -O "https://raw.githubusercontent.com/jdblischak/r-intermediate-altmetrics/gh-pages/data/cou
curl -O  "https://raw.githubusercontent.com/jdblischak/r-intermediate-altmetrics/gh-pages/data/cou
```

The first file, `counts-raw.txt.gz`, contains the raw counts for each of the articles across all the metrics. The second file, `counts-norm.txt.gz`, contains the counts for each of the articles across all the metrics after they have been normalized across disciplines and years.

Confirm that the files downloaded by listing the files in `data`.

```
ls
```

```
counts-norm.txt.gz counts-raw.txt.gz
```

Now move back up a directory to `altmetrics`. One option would be to specify the absolute path to this directory, e.g. `~/altmetrics/` if you created the directory in your home folder. However, an easier option is to use a relative path, which is dependent on the directory you are currently in. The shortcut to move to the directory above is two periods: `..`

```
cd ..
```

Now you should be in the `altmetrics` directory. If you ran the above command again, you would be moved to the directory where you created `altmetrics`.

To create files, we'll use the simple text editor `nano`. As an argument, you provide the name of an existing file to edit or the name of a new file to create. If you call `nano` without specifying a filename, it will prompt you for a filename when saving. Create a file to practice.

```
nano example-file
```

The commands are listed at the bottom of the screen. The ^ stands for `Ctrl`, thus to save type `Ctrl-W` and to exit type `Ctrl-X`.

Remove the file with `rm`.

```
rm example-file
```

1.7 Organizing a larger project

Projects can grow quickly. For ideas on organizing your own projects, see A Quick Guide to Organizing Computational Biology Projects by William Noble.

1.8 Create a README file

It is a convention to have a file named `README` in a project directory to explain what it contains (both for others and your future self). Use `nano` to create a README file. Include the date and explain that this directory was created for a Software Carpentry workshop.

Software Carpentry Source Contact License

Chapter 2

Conditionals and Loops

2.1 Setup

1. Download data.

We will be using a dataset containing citation and alternative metrics for articles published in the PLOS family of journals between 2003 and 2010. The data set was compiled by Priem et al 2012 (publication).

Download the data onto your computer from this dropbox link and move it into a directory on your computer that makes sense.

2. Read in data into R.

```
counts <- read.delim("data/counts-raw.txt")
```

2.2 Conditional statements

Decision making is an important part of programming. This can be achieved in R programming using conditional statements such as `if` and `if...else`.

if

The syntax of an if statement is:

```
if (test_expression) {
  do_this
}
```

```
x <- 5
if (x > 0) {
  print("positive number")
}
```

```
## [1] "positive number"
```

if...else

The syntax of an if...else statement is:

```
if (test_expression) {
  do_this
} else {
    do_that
}
```

The else part is optional and is only evaluated if test_expression is FALSE. It is important that the `else` word be in the same line as the closing brace of the `if` statement.

```
x <- 1
if (x > 0){
  print("positive number")
} else {
    print("negative number")
}
```

```
## [1] "positive number"
```

Nested if...else statements

You can have more than two test expressions:

```
if (test_expression1) {
  statement1
} else if (test_expression2) {
    statement2
} else {
      statement4
}
```

```r
x <- 0
if (x < 0) {
   print("negative number")
} else if (x > 0) {
   print("positive number")
} else {
   print("zero")
}
```

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

EXERCISE 2.1

Write a simple if...else statement to check if 5 is an odd number and if it is print "I am odd", otherwise print "I am even".

2.3 Loops

Conceptually, a loop is a way to repeat a sequence of instructions under certain conditions. They allow you to automate parts of your code that are in need of repetition.

`for` loop

The easiest and most frequently used loop in R is a for loop. Here is a demonstration of using loops.

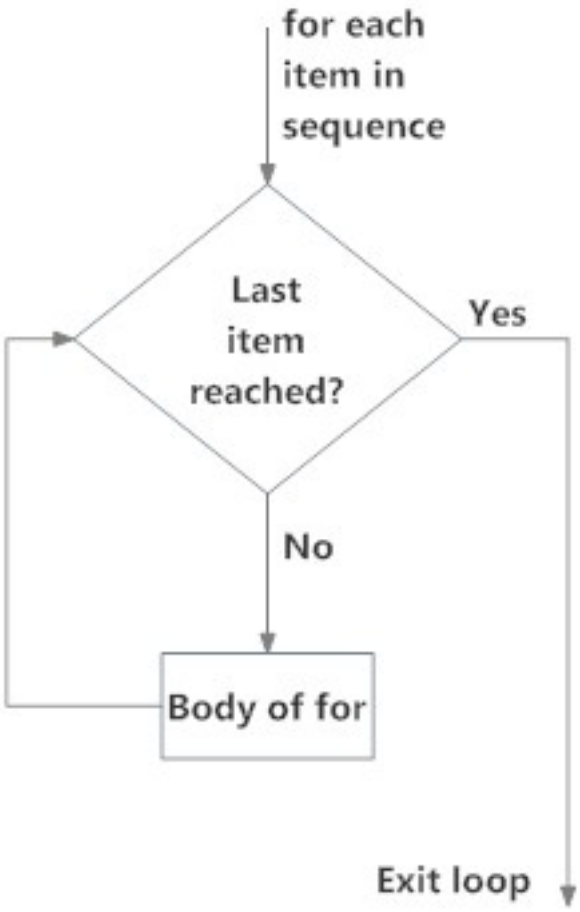


Fig: operation of for loop

```
year <- c(2015,2016,2017,2018)
for(i in 1:length(year)) {
  print(year[i])
}
```

```
## [1] 2015
## [1] 2016
## [1] 2017
## [1] 2018
```

```
for(i in 1:length(year)) {
  print(paste0("the year is ",year[i]))
}
```

```
## [1] "the year is 2015"
## [1] "the year is 2016"
## [1] "the year is 2017"
## [1] "the year is 2018"
```

`while` loop

In contrast to a **for** lop, **while** loops are used to loop until a specific conditional statement is no longer true.

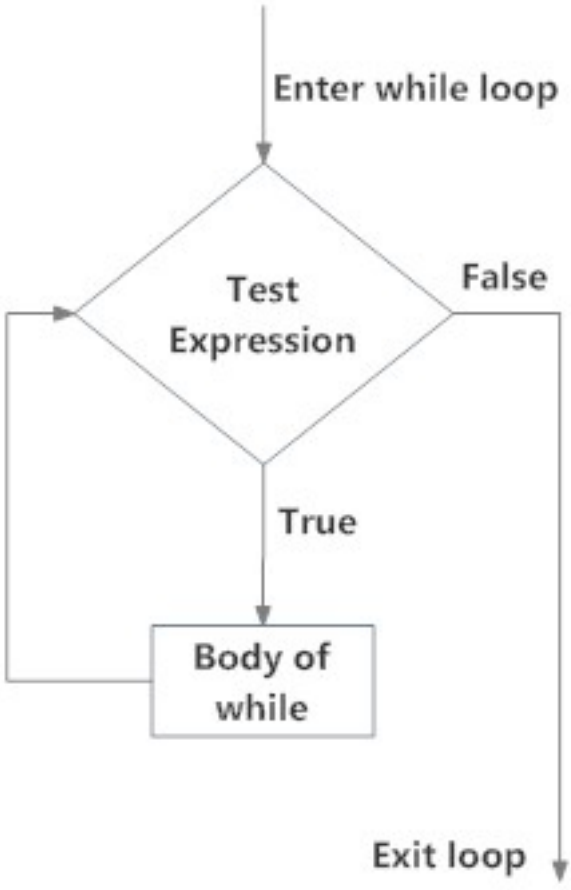


Fig: operation of while loop

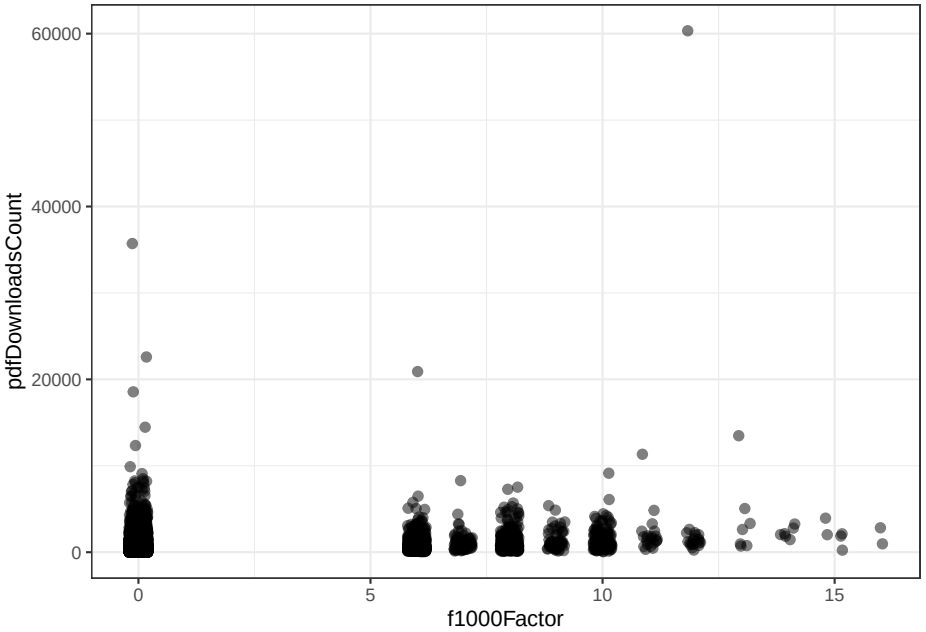
```
while (test_expression) {
  do_this
}
```

```
i <- 1
while (i < 6) {
  print(i)
  i <- i + 1
}
```

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

EXERCISE 2.2

Challenging. From the PLOS journal publication data we read into R above, here is a plot showing the impact factor according to the F1000 (Faculty of 1000) versus the number of times the PDF was downloaded.

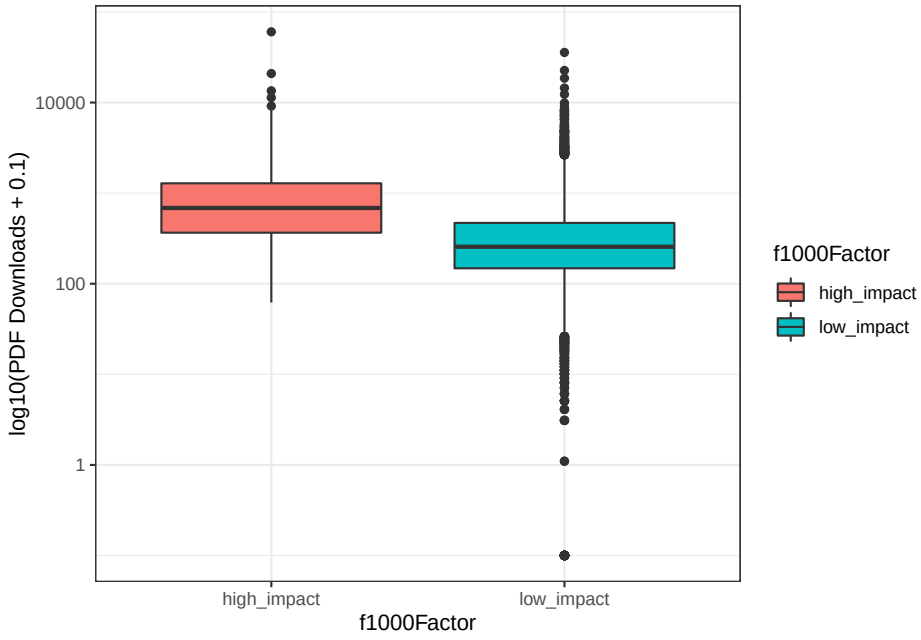


Using this dataset, write a `for` loop containing an `if...else` statement to change the f1000Factor column into categorical variable with two levels: high impact and low impact.

Do this by translating the following sentence into R code: for every element in the f1000Factor variable, if the value is greater than zero, change it to

"high_impact", otherwise, change it to "low_impact".

Bonus. Create a box plot (like the one below) showing the number of PDF downloads for high versus low impact articles.



Chapter 3

Functions

"Intelligence is the ability to avoid doing work, yet getting the work done."
Linus Torvalds

Learning Objectives

- Functions have two parts: arguments and body
- Functions have their own environment
- Functions help you repeat code chunks

In the last lesson we learnt to write loops to avoid writing the same code multiple time. Now lets learn how to make our code even more efficient and re-use the same code multiple times without copy-paste!

3.1 Parts of a function

We've already been using built-in R functions: `read.csv`, `mean`, `length`, etc. These functions allow us to run the same routine with different inputs.

Let's explore `read.csv` further. All functions in R have two parts: the input arguments and the body. We can see the arguments of a function with the `args`.

```
args(read.csv)
```

```
function (file, header = TRUE, sep = ",", quote = "\"", dec = ".",
    fill = TRUE, comment.char = "", ...)
NULL
```

So when we pass a character vector like `"data/counts-raw."`, this gets assigned to the argument `file`. All the other arguments have defaults set, so we do not need to assign them a value.

After the arguments have been assigned values, then the body of the function is executed. We can view the body of a function with `body`.

```
body(read.csv)
```

```
read.table(file = file, header = header, sep = sep, quote = quote,
    dec = dec, fill = fill, comment.char = comment.char, ...)
```

`read.csv` is very short. It just calls another function, `read.table`, using the input file and the default arguments as the arguments passed to `read.table`.

When we define our own functions, we use the syntax below. We list the arguments, separated by commas, within the parentheses. The body follows, contained within curly brackets `{}`.

```
function_name <- function(args) {
  body
}
```

3.2 The principle of encapsulation

An important feature of functions is the principle of encapsulation: the environment inside the function is distinct from the environment outside the function. In other words, variables defined inside a function are separate from variables defined outside the function.

Here's an small example to demonstrate this idea. The function `ex_fun` takes two input arguments, `x` and `y`. It calculates `z` and returns its value.

```
ex_fun <- function(x, y) {
  z <- x - y
  return(z)
}
```

When we run `ex_fun`, the only thing returned to the global environment is the value that was assigned to `z`. The variable `z` itself was only defined in the function environment, and does not exist in the global environment.

```
ex_fun(3, 10)
```

```
[1] -7
```

```
z
```

```
Error in eval(expr, envir, enclos): object 'z' not found
```

3.2.1 Environments

The situation presented above is a simplified version of environments which will serve you well if you treat functions as truly encapsulated. In reality, things are more complicated. For example, if inside a function you have a variable that has not been defined in the function, it will actually search the global environment for this variable. To learn the advanced details, see the chapter Environments in Advanced R by Hadley Wickham.

3.3 The return statement

R provides the shortcut of not needing to use `return` at the end of the function. Instead, the variable on the last line of the body of the function is returned. This is useful for writing very small functions, but in these lessons we will use `return` to be more explicit about what is happening.

Exercise

- Write your own function called `select_first`. It should take a vector as input and return the first element of the list.
- Now use articleType column from counts dataframe and use it as input to select_first function.

Chapter 4

The apply family

Learning Objectives

Learn to use apply family functions in place of loops.

Whenever you're using a for loop, you might want to revise your code and see whether you can use the lapply function instead. Learn all about this intuitive way of applying a function over a list or a vector, and its variants sapply and vapply.

4.1 apply()

apply() takes Data frame or matrix as an input and gives output in vector, list or array. apply() Function is primarily used to avoid explicit uses of loop constructs. It is the most basic of all collections can be used over a matrice.

This function takes 3 arguments:

`apply(X, MARGIN, FUN)`

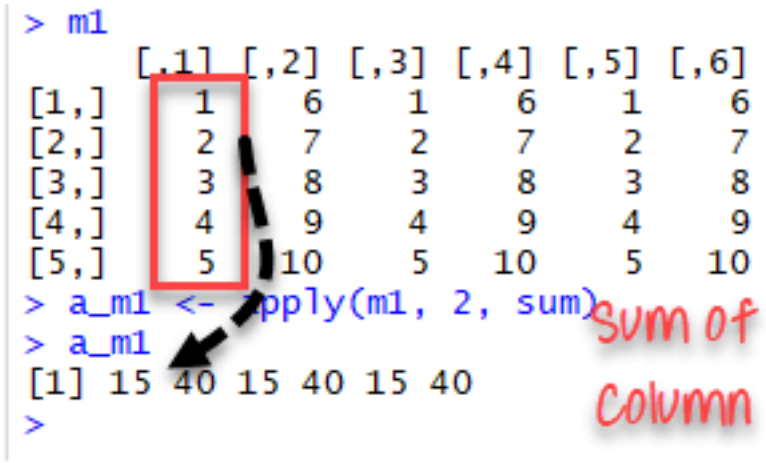
apply() takes an array or matrix `X`, a `Margin` that takes a value 1 for rows or 2 for columns and applies a functoin `FUN` to it.

```
m1 <- matrix(C<-(1:10),nrow=5, ncol=6)
m1
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6]
## [1,]    1    6    1    6    1    6
## [2,]    2    7    2    7    2    7
## [3,]    3    8    3    8    3    8
## [4,]    4    9    4    9    4    9
## [5,]    5   10    5   10    5   10
```

```
a_m1 <- apply(m1, 2, sum)
a_m1
```

```
## [1] 15 40 15 40 15 40
```



Exercise

Select the columns from count.raw file from pdfDownloadsCount to hmtlDownloadsCount and using the apply function find the mean of each column.

4.2 lapply()

Before you go about solving the exercises below, have a look at the documentation of the `lapply()` function. The Usage section shows the following expression:

```
lapply(X, FUN, ...)
```

To put it generally, `lapply` takes a vector or list `X`, and applies the function `FUN` to each of its members. If `FUN` requires additional arguments, you pass them after you've specified `X` and `FUN` (`...`). The output of `lapply()` is a list, the same length as `X`, where each element is the result of applying `FUN` on the corresponding element of `X`.

Now that you are truly brushing up on your data science skills, let's revisit some of the most relevant figures in data science history. We've compiled a vector of famous mathematicians/statisticians and the year they were born. Up to you to extract some information!

4.2.0.1 Example

```
# The vector pioneers has already been created for you
pioneers <- c("GAUSS:1777", "BAYES:1702", "PASCAL:1623", "PEARSON:1857")

# Split names from birth year to make split_math a list
split_math <- strsplit(pioneers, split = ":")
# Convert to lowercase strings: split_low
split_low <- lapply(split_math, tolower)
# Take a look at the structure of split_low
str(split_low)
```

```
## List of 4
##  $ : chr [1:2] "gauss" "1777"
##  $ : chr [1:2] "bayes" "1702"
##  $ : chr [1:2] "pascal" "1623"
##  $ : chr [1:2] "pearson" "1857"
```

4.2.1 Use lapply with your own function

As Filip explained in the instructional video, you can use `lapply()` on your own functions as well. You just need to code a new function and make sure it is available in the workspace. After that, you can use the function inside `lapply()` just as you did with base R functions.

In the previous exercise you already used `lapply()` once to convert the information about your favorite pioneering statisticians to a list of vectors composed of two character strings. Let's write some code to select the names and the birth years separately.

The sample code already includes code that defined `select_first()`, that takes a vector as input and returns the first element of this vector.

4.2.1.1 Example

```
select_first <- function(x) {
  x[1]
}
# Apply select_first() over split_low: names
names <- lapply(split_low, select_first)
```

4.2.2 lapply and anonymous functions

Writing your own functions and then using them inside `lapply()` is quite an accomplishment! But defining functions to use them only once is kind of overkill, isn't it? That's why you can use so-called **anonymous functions** in R.

Previously, you learned that functions in R are objects in their own right. This means that they aren't automatically bound to a name. When you create a function, you can use the assignment operator to give the function a name. It's perfectly possible, however, to not give the function a name. This is called an anonymous function:

```
# Named function
select_first <- function(x) { x[1] }

# Use anonymous function inside lapply()
lapply(list(1,2,3), function(x) {x[1]})
```

4.2.2.1 Example

```
# Definition of split_low
pioneers <- c("GAUSS:1777", "BAYES:1702", "PASCAL:1623", "PEARSON:1857")
split <- strsplit(pioneers, split = ":")
split_low <- lapply(split, tolower)
# Transform: use anonymous function inside lapply
names <- lapply(split_low, function(x) { x[1] })
```

4.2.3 Use lapply with additional arguments

In the video, the `triple()` function was transformed to the `multiply()` function to allow for a more generic approach. `lapply()` provides a way to handle functions that require more than one argument, such as the `multiply()` function:

```
multiply <- function(x, factor) {
  x * factor
}

lapply(list(1,2,3), multiply, factor = 3)
```

On the right we've included a generic version of the select functions that you've coded earlier: `select_el()`. It takes a vector as its first argument, and an index as its second argument. It returns the vector's element at the specified index.

4.2.3.1 Example

```
# Generic select function
select_el <- function(x, index) {
  x[index]
}
# Use lapply() twice on split_low: names and years
names <- lapply(split_low, select_el, index = 1)
years <- lapply(split_low, select_el, index = 2)
```

4.3 sapply()

You can use `sapply()` similar to how you used `lapply()`. sapply is a user-friendly version and wrapper of lapply by default returning a vector, matrix .The first argument of `sapply()` is the list or vector `X` over which you want to apply a function, `FUN`. Potential additional arguments to this function are specified afterwards (`...`):

```
sapply(X, FUN, ...)
```

4.3.0.1 Example

```
# Constructing temp variable
names <- sapply(split_low, select_el, index = 1)
```

4.3.1 sapply with your own function

Like `lapply()`, `sapply()` allows you to use self-defined functions and apply them over a vector or a list:

```
sapply(X, FUN, ...)
```

Here, `FUN` can be one of R's built-in functions, but it can also be a function you wrote. This self-written function can be defined before hand, or can be inserted directly as an anonymous function.

4.3.2 sapply with function returning vector

In the previous exercises, you've seen how `sapply()` simplifies the list that `lapply()` would return by turning it into a vector. But what if the function you're applying over a list or a vector returns a vector of length greater than 1?

If you don't remember from the video, don't waste more time in the valley of ignorance and head over to the instructions!

4.3.2.1 Example

```
temp=list(c(1,3,4,4,6),c(3,4,6,8,9))
# Create a function that returns min and max of a vector: extremes
extremes <- function(x) {
  c(min = min(x), max = max(x))
}
# Apply extremes() over temp with lapply()
lapply(temp, extremes)
```

```
## [[1]]
## min max
##   1   6
##
## [[2]]
## min max
##   3   9
```

```
# Apply extremes() over temp with sapply()
sapply(temp, extremes)
```

```
##     [,1] [,2]
## min    1    3
## max    6    9
```

4.3.3 sapply can't simplify, now what?

It seems like we've hit the jackpot with `sapply()`. On all of the examples so far, `sapply()` was able to nicely simplify the rather bulky output of `lapply()`. But, as with life, there are things you can't simplify. How does `sapply()` react?

We already created a function, `below_zero()`, that takes a vector of numerical values and returns a vector that only contains the values that are strictly below zero.

4.3.3.1 Example

```
# temp is already prepared for you in the workspace
# Definition of below_zero()
below_zero <- function(x) {
  return(x[x < 0])
```

```
}
# Apply below_zero over temp using lapply(): freezing_l
freezing_l <- lapply(temp, below_zero)
# Apply below_zero over temp using sapply(): freezing_s
freezing_s <- sapply(temp, below_zero)
# Are freezing_l and freezing_s identical?
identical(freezing_l, freezing_s)
```

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

4.4 vapply()

Before you get your hands dirty with the third and last apply function that you'll learn about in this intermediate R course, let's take a look at its syntax. The function is called `vapply()`, and it has the following syntax:

```
vapply(X, FUN, FUN.VALUE, ..., USE.NAMES = TRUE)
```

Over the elements inside `X`, the function `FUN` is applied. The `FUN.VALUE` argument expects a template for the return argument of this function `FUN`. `USE.NAMES` is `TRUE` by default; in this case `vapply()` tries to generate a named array, if possible.

For the next set of exercises, you'll be working on the `temp` list again, that contains 7 numerical vectors of length 5. We also coded a function `basics()` that takes a vector, and returns a named vector of length 3, containing the minimum, mean and maximum value of the vector respectively.

4.4.0.1 Example

```
# temp is already available in the workspace
# Definition of basics()
basics <- function(x) {
  c(min = min(x), mean = mean(x), max = max(x))
}
# Apply basics() over temp using vapply()
vapply(temp, basics, numeric(3))
```

```
##      [,1] [,2]
## min   1.0    3
## mean  3.6    6
## max   6.0    9
```

4.4.0.2 Exercise

```
<br>
Use the table() function and one of the apply functions to see how many counts of each year a
Explain which function you used and what does the output look like?

</div>


<!--chapter:end:04-Apply_family.Rmd-->

# Advanced Plotting

"The greatest value of a picture is when it forces us to notice what we never expected to see
-John Tukey

---

## Setup

**1. Install the `tidyverse` package.**


```r
library(tidyverse)
```

2. Filter data.

We will be using the publication dataset that we read into R in Chapter 2 as `counts`.

```r
research <- filter(counts, articleType == "Research Article")
```

4.5 Review of ggplot2 basics

`ggplot2` is a plotting package that makes it simple to create complex plots from data in a data frame. Graphics are built step by step by adding new elements. Adding layers in this fashion allows for extensive flexibility and customization of plots.

A plot can be divided into different fundamental parts:

Plot = data + aesthetics + geom

Required building blocks:

- data
```

- aesthetics - describe how data are mapped to colour, size, shape, location
- geoms - geometric objects like points, lines, shapes

Optional building blocks:

- facets - describes how panel plots should be constructed
- stats - statistical transformations like binning, quantiles, smoothing
- coordinates - describes the system in which the locations of the geoms will be drawn
- scales - what scale an aesthetic map uses (ex. male = red, female = blue)

To build a ggplot, we will use the following basic template that can be used for different types of plots:

```
ggplot(data = <DATA>, mapping = aes(<MAPPINGS>)) + <GEOM_FUNCTION>()
```

1. Specify which data set to use for the plot using the `data` argument.

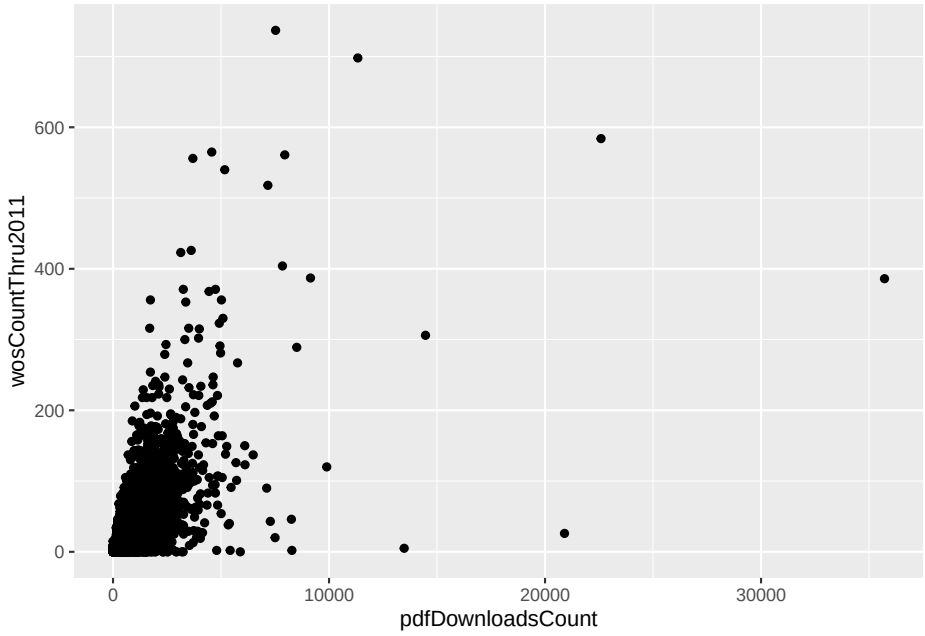
2. Define a "mapping" (using the aesthetic (`aes`) function), by selecting the variables to be plotted and specifying how to present them in the graph, e.g. as x/y positions or characteristics such as size, shape, color, etc.

3. Add "geoms" – graphical representations of the data in the plot (points, lines, bars). `ggplot2` offers many different geoms; common ones include:

- `geom_point()` for scatter plots, dot plots, etc.
- `geom_boxplot()` for boxplots.
- `geom_histogram()` for histograms.
- `geom_barplot()` for barplots.
- `geom_line()` for trend lines, time series, etc.

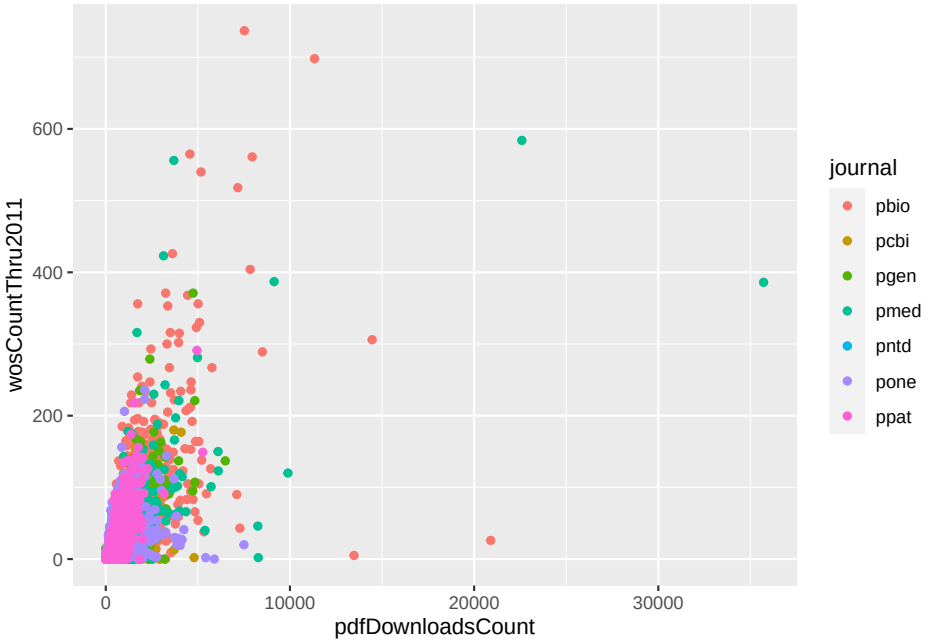
Example:

```
p <- ggplot(research, aes( x = pdfDownloadsCount, y = wosCountThru2011)) + geom_point()
p
```



Adding aesthetics:

```
p <- ggplot(research, aes(x = pdfDownloadsCount,
                          y = wosCountThru2011)) +
  geom_point(aes(color = journal))
p
```

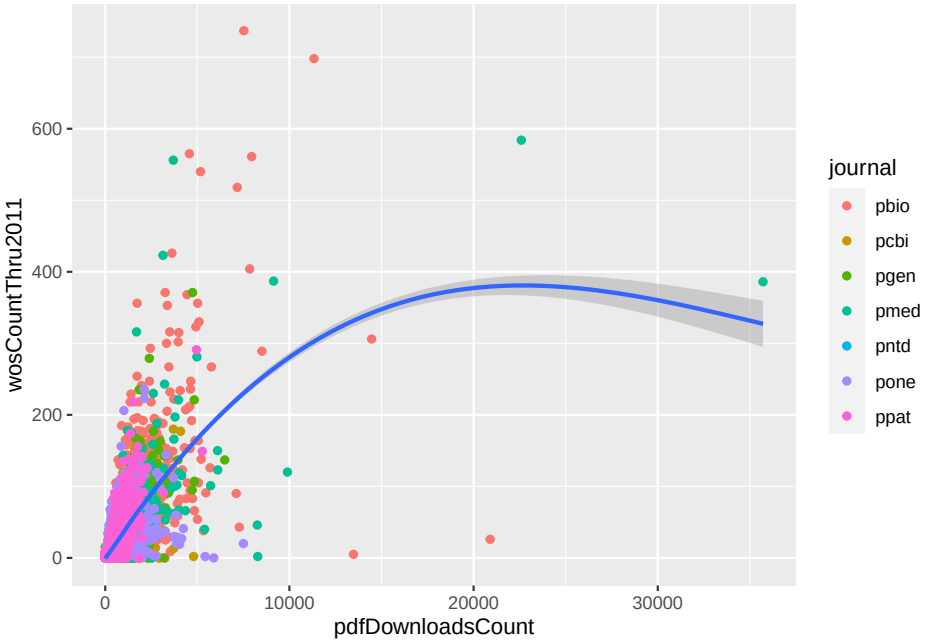


4.6 Statistics

The function `geom_smooth()` adds a loess curve to the data along with a 95% confidence interval.

```
p <- ggplot(research, aes(x = pdfDownloadsCount,
                          y = wosCountThru2011)) +
  geom_point(aes(color = journal)) +
  geom_smooth()
p
```

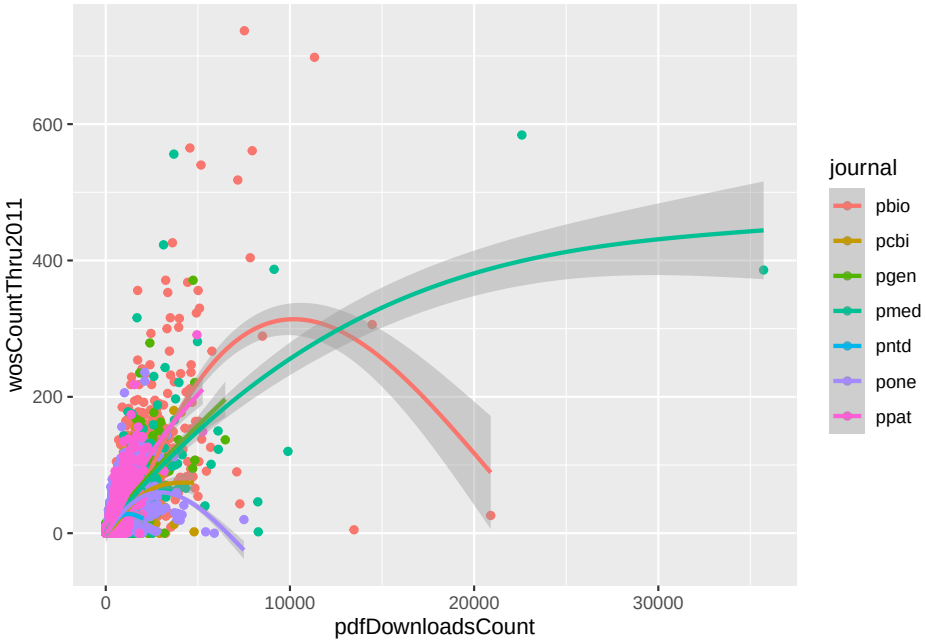
```
## `geom_smooth()` using method = 'gam' and formula 'y ~ s(x, bs = "cs")'
```



If we move the colour to the base ggplot call, we get loess curves for each level of that factor.

```
p <- ggplot(research, aes(x = pdfDownloadsCount,
                          y = wosCountThru2011,
                          color = journal)) +
  geom_point() +
  geom_smooth()
p
```

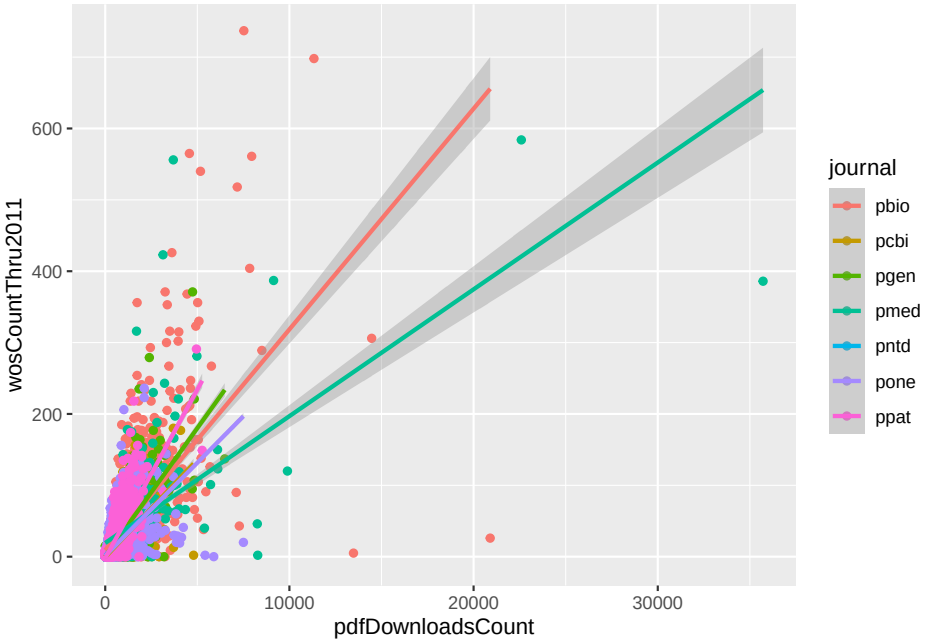
```
## `geom_smooth()` using method = 'gam' and formula 'y ~ s(x, bs = "cs")'
```



Check the help page for the function `geom_smooth()` for more information about how the curve is made. For example, to map a linear model onto the plot, you can choose `method = "lm"`.

```
p <- ggplot(research, aes(x = pdfDownloadsCount,
                          y = wosCountThru2011,
                          color = journal)) +
  geom_point() +
  geom_smooth(method = "lm")
p
```

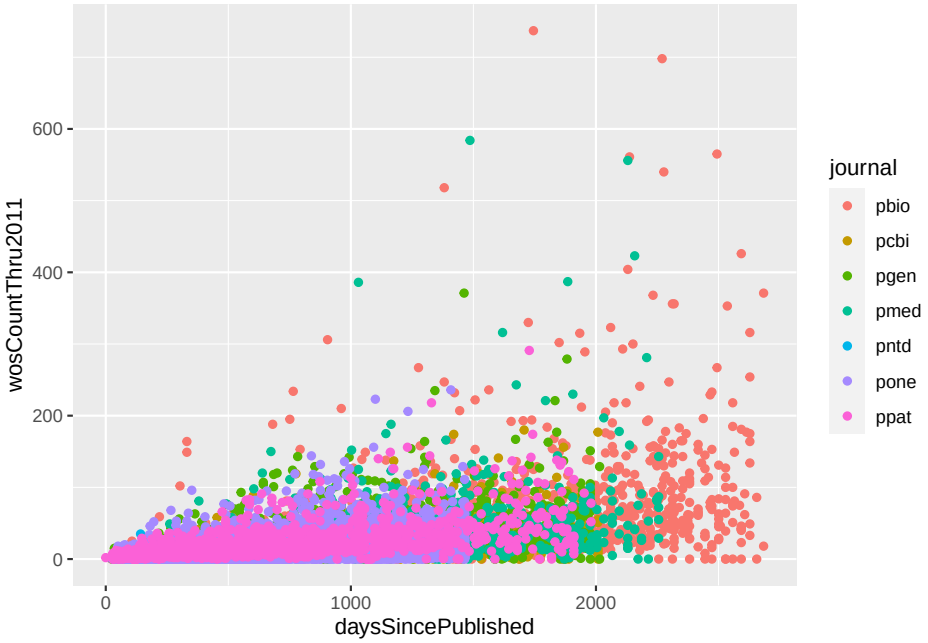
```
## `geom_smooth()` using formula 'y ~ x'
```



4.7 Scales

Now let's look at the relationship between days since published and citation count.

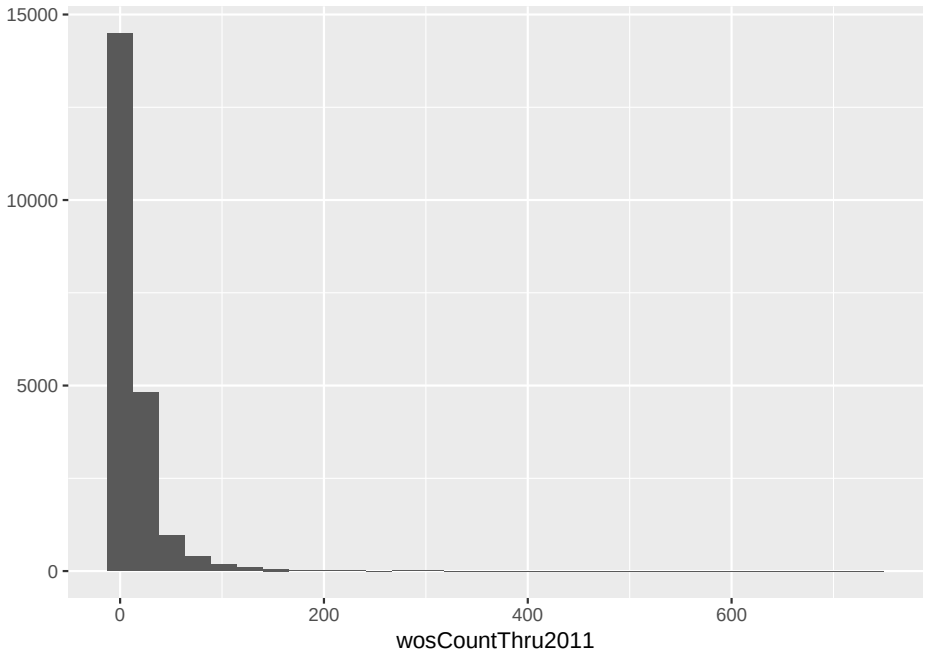
```
p <- ggplot(research, aes(x = daysSincePublished,
                          y = wosCountThru2011)) +
  geom_point(aes(color= journal))
p
```



It looks like most of the citation counts are close to 0. We can quickly check the distribution of this variable using a qplot.

```
qplot(data = research, x = wosCountThru2011)
```

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



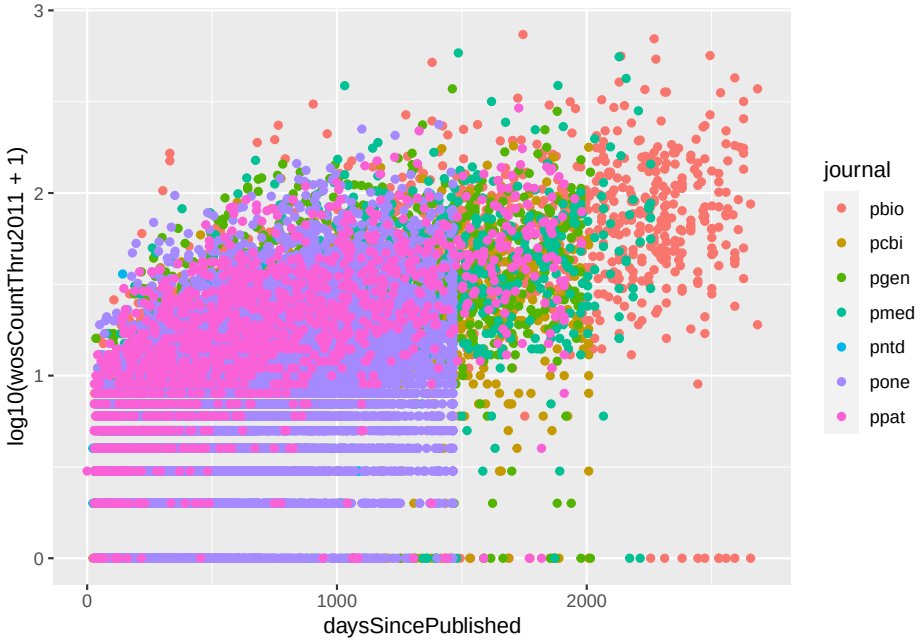
To control the plot axes, we use variants of the functions `scale_x_*` and `scale_y_*`.

```
p <- ggplot(research, aes(x = daysSincePublished,
                          y = wosCountThru2011)) +
  geom_point(aes(color= journal)) +
  scale_y_log10()
p
```

```
## Warning: Transformation introduced infinite values in continuous y-axis
```

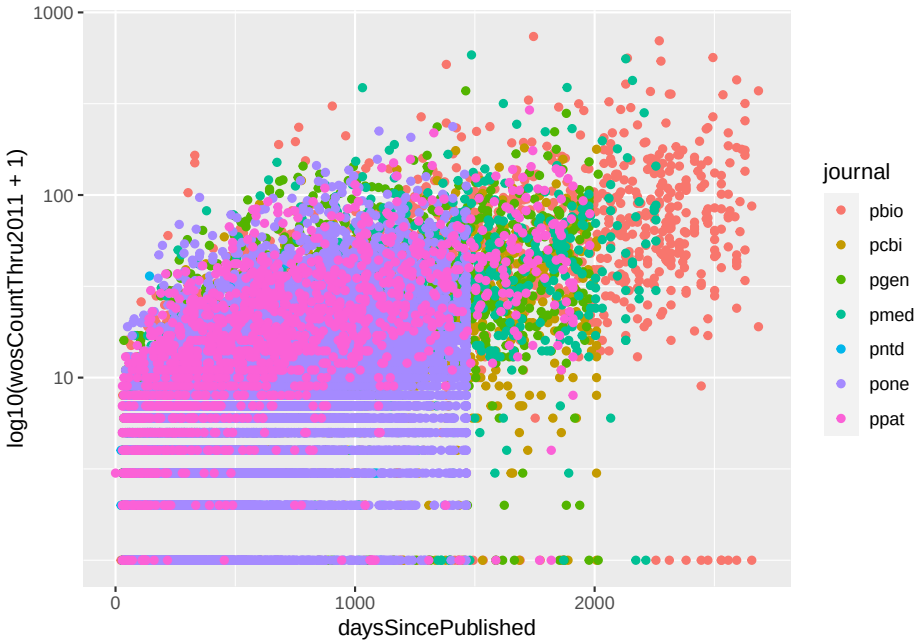
How can we fix this?

```
p <- ggplot(research, aes(x = daysSincePublished,
                          y = log10(wosCountThru2011 + 1))) +
  geom_point(aes(color= journal))
p
```



Notice what this fix has done to the way the y-axis is labelled. To manually update the axis label, we can use the `scale_y_continuous()` function.

```
p <- ggplot(research, aes(x = daysSincePublished,
                          y = log10(wosCountThru2011 + 1))) +
  geom_point(aes(color= journal)) +
  scale_y_continuous(breaks = c(1,2,3), labels = c(10, 100, 1000))
p
```



4.8 Faceting

There are two functions to control how plots are divided into facets: `facet_wrap()` and `facet_grid()`.

4.9 Themes

In addition to `theme_bw()`, which changes the plot background to white, `ggplot2` comes with several other themes which can be useful to quickly change the look of your visualization. The complete list of themes is available at https://ggplot2.tidyverse.org/reference/ggtheme.html. `theme_minimal()` and `theme_light()` are popular, and `theme_void()` can be useful as a starting point to create a new hand-crafted theme.

The `ggthemes` package provides a wide variety of options (including an Excel 2003 theme). The `ggplot2` extensions website provides a list of packages that extend the capabilities of `ggplot2`, including additional themes.

4.10 Color palettes

You can create your own color palettes using the `colorRamp()` or `colorRampPalette()` function.

`colorRamp()` returns a function that takes values between 0 and 1, indicating the extremes of the color palette.

`colorRampPalette()` returns a function that takes integer arguments and returns a vector of colours.

```
cols <- colorRamp(c("red", "blue"))
cols(0)
```

```
##      [,1] [,2] [,3]
## [1,]  255    0    0
```

```
cols(0.5)
```

```
##       [,1] [,2]  [,3]
## [1,] 127.5    0 127.5
```

```
cols(1)
```

```
##      [,1] [,2] [,3]
## [1,]    0    0  255
```

```
cols <- colorRampPalette(c("red", "blue"))
cols(2)
```

```
## [1] "#FF0000" "#0000FF"
```

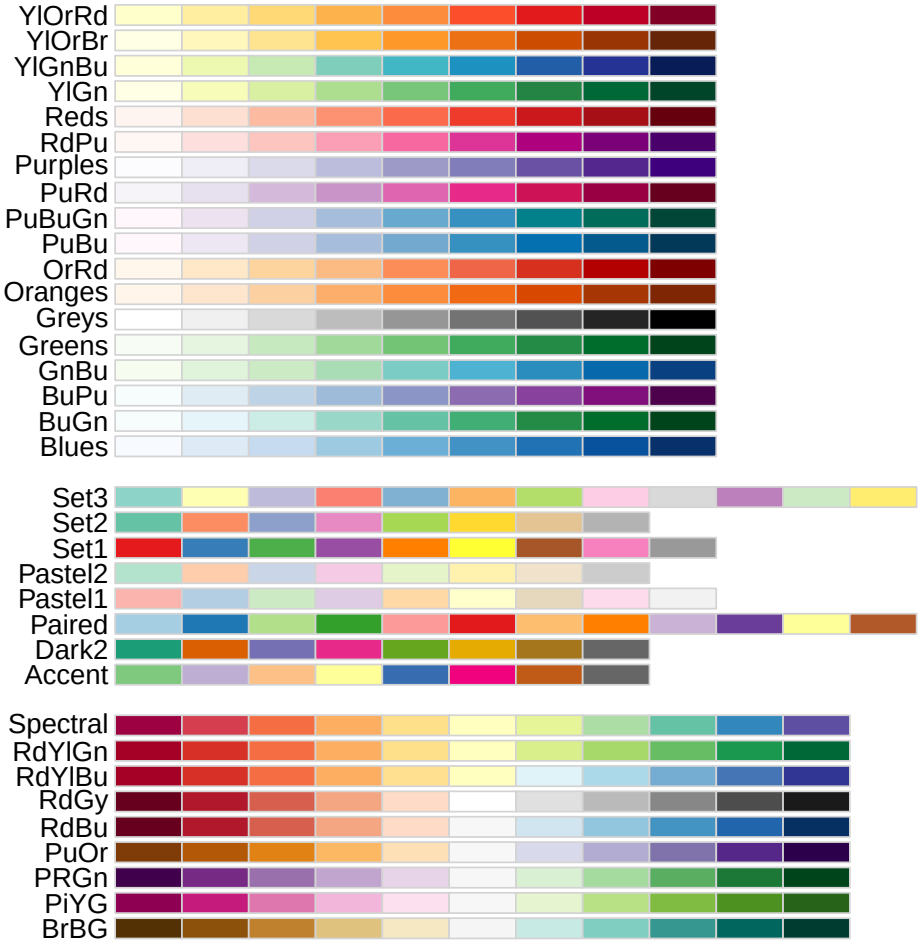
```
cols(10)
```

```
##  [1] "#FF0000" "#E2001C" "#C60038" "#AA0055" "#8D0071" "#71008D" "#5500AA"
##  [8] "#3800C6" "#1C00E2" "#0000FF"
```

Or, you can use the `RColorBrewer` package to get a premade colour palette.

There are three types of palettes: * Sequential * Diverging * Qualitative

```
library(RColorBrewer)
display.brewer.all()
```



4.11 Multiple plots

There are two useful packages for combining multiple plots: `gridExtra` and `cowplot`.

`gridExtra` has two useful functions: `grid.arrange()` and `arrangeGrob()`. However, these functions make no attempt at aligning the plot panels; instead, the plots are simply placed into the grid as they are, so the axes are not aligned. If axis alignment is required, the `plot_grid()` function of the `cowplot` package is better. We will try using both here.

4.11.1 The `gridExtra` package

4.11.2 The `cowplot` package

4.11.3 Saving plots

The easiest way to save a plot is using the `ggsave()` function.

4.12 Additional Resources

http://www.sthda.com/english/wiki/be-awesome-in-ggplot2-a-practical-guide-to-be-highly-effective-r-software-and-data-visualization

http://r-statistics.co/Top50-Ggplot2-Visualizations-MasterList-R-Code.html

- Mailing list: http://groups.google.com/group/ggplot2
- Wiki: https://github.com/hadley/ggplot2/wiki
- Website: http://had.co.nz/ggplot2/
- StackOverflow: http://stackoverflow.com/questions/tagged/ggplot

Cheatsheet

Chapter 5

Rmarkdown

"If I went back to college again, I'd concentrate on two areas: learning to write and to speak before an audience. Nothing in life is more important than the ability to communicate effectively." – Gerald R. Ford

Learning Objectives

Learn how to generate reproducible reports that display your code and results.

5.1 Set up new R Markdown file

When you perform wet lab experiments, what information do you put in your lab notebook? You probably include the protocol you used to run the experiment, information about the samples and reagents used in the protocol, and at the end you'll likely include your results (for instance, a picture of a gel). This essentially creates a report of your experiment.

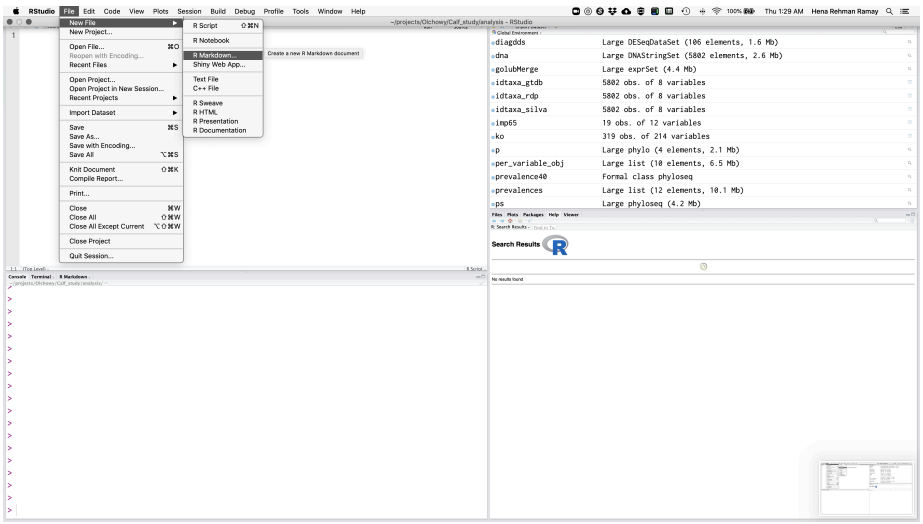
You can do the same with your dry lab analyses using a tool called R Markdown. Why would we want to do this?

- Your method, results, and interpretation are stored in one place
- If you update your methodology, you can easily update your results with the click of a button, rather than copying and pasting.
- You *could* cut and paste your code and results into Word or Power Point, but that will make rerunning your code challenging, as Word often introduces hidden characters.

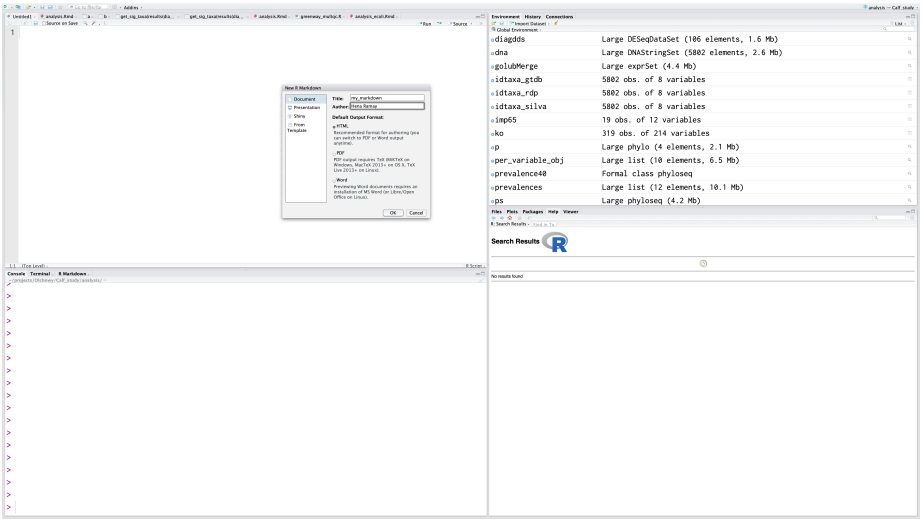
R Markdown is a fairly simple language you can use to generate reports that incorporate bits of R code along with the output they produce. There are two steps to generating reports with R Markdown and RStudio:

1. Write your code in R Markdown.
2. Assemble your report as either HTML or a PDF using the package rmarkdown.

Next, let's run through the demo R Markdown file to see some of the options. Go up to `File` -> `New File` -> `R Markdown`.



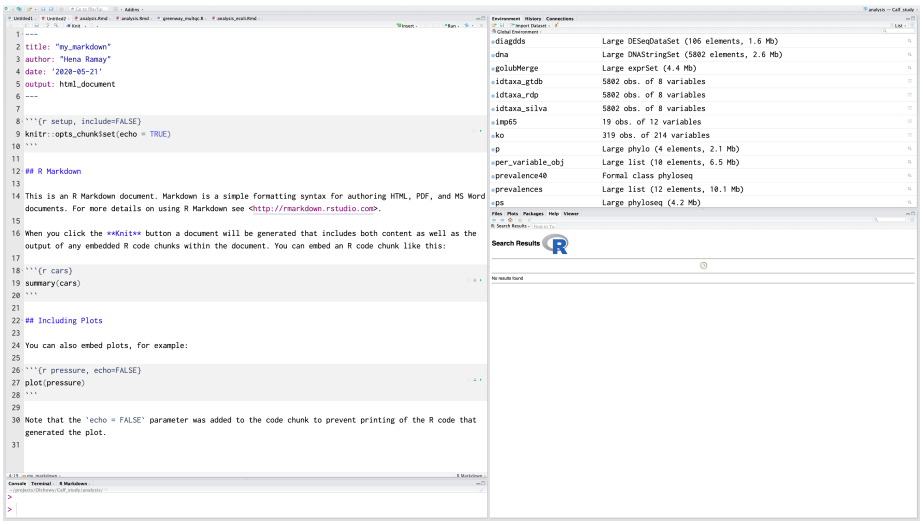
A screen will pop up asking us what kind of document we wish to create. Let's name our demo report "Trial Report" and fill in your name. Ensure that "Document" is highlighted to the left and that "HTML" is chosen. Click "Ok".



Now we have the example R Markdown file open. The first thing you'll notice at the top is a header which includes your name, the title of the document, the date, and a field called output. This header tells the package rmarkdown some information it might need about your document, including what format you want the final report rendered in.

The next thing you'll notice is white space with some text describing an R Markdown document. White space in this document represents text of the report you would like to display. You can put anything here describing your analysis, results, etc. and it will be recognized as text and not R code. This white space is interpreted as Markdown language, so you can use any of the tricks we learned in the last lesson to make lists, bold certain words, or create headers in your document.

In this trial script, you'll see how some of these markdown elements are used. For example, the word knit is in bolded (using asterisks), and there are code chucks near the bottom that say echo = FALSE.



5.1.1 R Markdown componenets

Notice that the file contains three types of content:

1. An (optional) YAML header surrounded by —s
2. R code chunks surrounded by “‘s
3. text mixed with simple text formatting

In addition to the white space, you’ll gray blocks that have “‘ at the top and bottom. These are called chunks. If the start of a chunk has {r} at the end of the ticks, the code will be run and both it and its output will be displayed in the rendered HTML. In your R Markdown, the code will look like:

In your final report, the code will look like:

```
summary(cars)
```

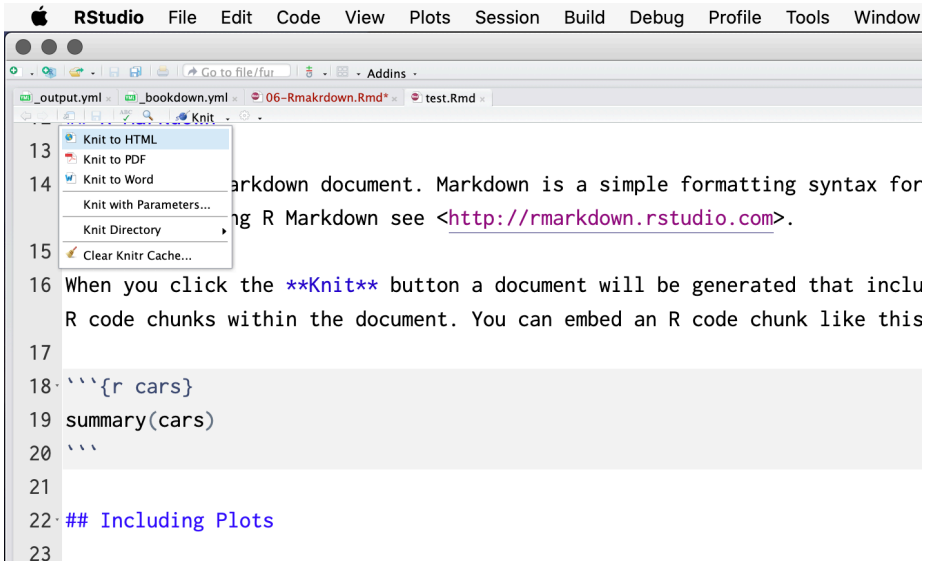
Let’s add a new chunk to end this demo document. To do so, either you can enter three backticks in a row, followed by {r}, or you can click on the green **Chunks**button and chose**Insert Chunk**. Additionally, there’s a keyboard short cut which is **ctrl**+**alt**+**i**which will also pop up a chunk in an R Markdown document.

In the chunk, let’s just examine the dimensions of the **car**mdataset:

You can actually send the code straight from the chunks over to console to be evaluated in two ways. First, you can highlight the code you want to run in the chunk and hit the **Run** button, which is located in the top right corner of the pane.

5.1.2 Knit R Markdown

These are the basics of writing R Markdown, but we still need to generate a report. To do this, click on the button on the top bar that says "Knit HMTL". This will prompt you to save the file. Go ahead and save this file as `Rmarkdown_demo.Rmd` in the altmetrics directory. The ending of the file `.Rmd` indicates that this is an R Markdown file.



When you click on this link, you see in the console that RStudio is running and rendering your R Markdown file. What is actually happening is RStudio is running the function `render`, which is part of the rmarkdown package. There are two things the command `render` does. First, it converts the R Markdown file to a Markdown file using the command `knit` from the knitr package (hence why rendering is called knitting). The second step is then the Markdown file is converted to the final file format (HTML, PDF, or Word).

The final result is that an HMTL file will pop up where you'll see the report. You can see the header has been rendered, there are code and results chunks displayed, and even plots are shown right in the report.

Also, if you now look in the altmetrics folder, you'll see an HTML file of the name Rmarkdown_demo.html. When `render` is run, it saves the current version of the .Rmd file and the generated HTML file in the directory it is stored in.

5.2 Knitr Chunk Options

Learning Objectives

Learn how to format chunks in R Markdown to display only the information you want to display.

You've learned the basics of how to incorporate markdown syntax with code chunks in an R Markdown file. Let's explore some additional options for making your code chunks appear the way you want them to appear in your reports. There are many ways to customize your chunks and you can explore all of the options by examining the documentation. Here, we'll introduce you to some of the most useful options that you might use frequently.

Chunk output can be customized with knitr options , arguments set in the `{}` of a chunk header. Above, we use five arguments:

- `include = FALSE` prevents code and results from appearing in the finished file. R Markdown still runs the code in the chunk, and the results can be used by other chunks.
- `echo = FALSE` prevents code, but not the results from appearing in the finished file. This is a useful way to embed figures.
- `message = FALSE` prevents messages that are generated by code from appearing in the finished file.
- `warning = FALSE` prevents warnings that are generated by code from appearing in the finished.
- `fig.cap = "..."` adds a caption to graphical results.

The first thing you may want to consider is naming your code chunks, which makes degubbing easier, especially if you have a long script. Chunk names must be unique to each chunk.

Write the name of your chunk after the {r}, like: `{r chunk_name}`

R Markdown:

Rendered:

```
summary(cars)
```

```
##      speed           dist
##  Min.   : 4.0   Min.   :  2.00
##  1st Qu.:12.0   1st Qu.: 26.00
##  Median :15.0   Median : 36.00
##  Mean   :15.4   Mean   : 42.98
##  3rd Qu.:19.0   3rd Qu.: 56.00
##  Max.   :25.0   Max.   :120.00
```

You can use RStudio to navigate to chunks based on their names, which can be especially useful as your script gets long. Click on the bottom left bar where it

says `(Top Level)` and you'll see all of the chunk names in your script appear. Additionally, naming your chunks will be beneficial to identify errors in your code or slow sections when knitting your report.

Sometimes you may not want to see the code that produced a particular result in your report. You can have codeblocks in your R Markdown that are evaluated, but the code is not displayed in the final report by including `echo=FALSE` after the `{r chunk_name}`.

R Markdown:

Rendered:

```
##      speed           dist
##  Min.   : 4.0   Min.   :  2.00
##  1st Qu.:12.0   1st Qu.: 26.00
##  Median :15.0   Median : 36.00
##  Mean   :15.4   Mean   : 42.98
##  3rd Qu.:19.0   3rd Qu.: 56.00
##  Max.   :25.0   Max.   :120.00
```

Conversely, sometimes you may want to see the code, but not the output once the code is evalutated. To do so, you can include `results="hide"` after the chunk_name:

R Markdown:

Rendered:

```
summary(cars)
```

Sometimes, you may want to write a report where both the code and the output are suppressed. Why would you want to do that? Perhaps you're sending a report to a collaborator and you only want them to see the final figures, but not any data manipulation steps in the middle. To include a chunk that is evaluated, but no output is displayed - neither the code nor the results - put `include=FALSE` after the chunk_name:

R Markdown:

Rendered (I swear there's a chunk after this! It's just invisible!):

There are tons of other options you can include in your chunks: sizing your figures and whether or not to display error or warning messages. As you write reports of your own data analysis, you can look up these options to create a report formatted in the way you want.

In addition to code chunks, you may want to include the results of an evaluation in line with regular text. For instance, you may want to describe the data in a paragraph, and include the number of individuals in that paragraph. To do so, you can indicate that a code box should be evaluated as R by including a lowercase r.

R Markdown:

```
The _cars_ dataset included in this analysis contains records for `r dim(cars)[1]` cars.
```

Rendered:

```
The _cars_ dataset included in this analysis contains records for 50 cars.
```

Chapter 6

Project

6.1 Setup

Please download the data folder from Dropbox

6.2 Task 1

1. Create a new R Project for this case study.
2. Create a new Rmd file to complete your project.
3. Change the auther name of the .Rmd file to your name and add today's date.
4. Read in all the files in the data folder into single dataframe. Hint: Use for loops to read in all the files.Functions that can be helpful : list.files,read_csv,rbind,data.frame(NULL)

6.3 Task 2

1. Write a function that takes a dataframe as input and if present, converts Monthin to 4 yearly quaters and add an extra column to your dataframe/tibble called Quarters with 4 levels: Q1,Q2,Q3 and, Q4.

- When Month is Jan-Mar, Quarter = "Q1"
- When Month is Apr-Jun, Quarter = "Q2"
- When Month is Jul-Sep, Quarter = "Q3"
- When Month is Oct-Dec, Quarter = "Q4"

Hint: Use mutate and case_when(), %in% and stop() function

6.4 Task3

Chapter 7

Interesting Packages

1. Cowplot
2. forcats
3. biobroom
4.