```
---
title: "Intro to Data Analysis & Visualization in R"
date: "2025-03-12"
author:
  - name: "Jeffrey K. Bye, Ph.D."
    id: jkbye
    orcid: 0000-0002-2636-3657
    email: jkbye@csudh.edu
    affiliation:
      - name: Cal State Dominguez Hills
        url: https://www.csudh.edu/psychology/
format: html
toc: true
number-sections: true
editor: visual
---
```

## Intro

Welcome to my demonstration of a few cool things you can do in R! There is no way to cover everything I want to in just one short hour. So I tried to balance a bit of breadth with a bit of depth. After the workshop, the OUR team will email you instructinos for how to download R and RStudio (for free!), and a copy of this document to use in RStudio to explore more on your own time -- please feel free to reach out to me with any questions! My email is at the top of this document (in the "YAML" section, which specifies the metadata like title/author/date).

R

R is an open-source (free as in gratis *and* libre) programming language designed specifically for people doing data analysis (statistical modeling, visualization, data wrangling, etc.) I will be teaching a class on R in Fall 2025: **PSY 495 - 01 (42613) Social Data Science**. Please email me if you have questions!

### Quarto

Quarto is a special type of document (`.qmd` format) that allows us to combine text, images, and code together into a single document. (It is similar to a Jupyter Notebook, which we used for our online demo today). Opening a Quarto document up inside of RStudio, we can also run bits of code one at a time, using the 'play' button (green triangle) in each code chunk. (To learn more about Quarto see <https://quarto.org>. I even built my personal website entirely in Quarto! It's pretty cool.)

## Running Code

Quarto is a special kind of R environment. It can be a bit friendlier to new learners, but it's far from the only way to use R. We'll use it here just because it's a short demo, and you can do a lot in Quarto with only a little learning. So that's cool! Let's learn together :)

### Commands

~~~
Fundamentally, R works like this:

- you give a clearly defined command to R (think: question!)
- R gives you the response (think: answer!)

```{r}
2 + 2
```

Congratulations! You have a calculator.

But unlike a calculator, you can also leave notes for yourself:

```{r}
2 + 2 # yay math
```

R will ignore anything after the `#` symbol. This is called a 'comment'
in programming lingo. It's human language meant for humans, not for
computers (they ignore it entirely!).

Of course, R is a lot more than a calculator that lets you leave
comments; but today we only have time explore a small fraction of what R
can do. Or should I say fRaction? Eh?

Above, we used the `+` (addition) operator to perform... addition! But we
are quickly going to run out of special symbols on the keyboard to do
most of the things we want. How would we find the square root? There's no
key on my keyboard for that like there is on a calculator.

Here, we rely on something called a ***function***.

### Functions

```{r}
sqrt(12)
```

Functions take *inputs* and give us *outputs*. We can think of a function
as a 'black box' that we use to transform input to output, like alchemy
(or math):

![Function as a black box](function_as_a_black_box.png)

(There are fancier terms for these components, but we're going to keep it
simple for now.)

#### A bit of syntax

Note that the function is called `sqrt`. But we need the parentheses! So
we'll often remind ourselves by referring to the function as `sqrt()`. If
we don't include parentheses...
~~~

```{r}
sqrt12 # comment this out later by adding a # at the beginning of the
line
```

ERROR! That's okay, we'll see a lot of those. Computers are very, very *literal*. They can only follow very specific instructions, and those instructions have a very specific form called **syntax**. We'll learn a bit of syntax today!

(By the way, when you get an error like above, you might want to 'comment it out' by putting a `#` sign before it, so R will ignore the code, thinking it's a comment.)

Syntax is a similar construct in natural language (like English, Spanish) and artificial languages (like R, Python). It is the *structure* of the language (remember doing sentence trees?).

For example:

- In English: "green turtle" follows the syntax {adjective}{noun}.
- Whereas in Spanish: "tortuga verde" follows the syntax {noun}{adjective}.

Notice neither is inherently correct. It is a choice made (collectively) by humans via language evolution. Natural language is cool! (Take Linguistics classes!)

With humans, we could probably figure out what someone meant if they said "I saw a turtle green" -- because we understand the world and can make inferences in context. But computers can't do that, they will follow rules very strictly. So when you get an error message, it's often from very small deviations from the strict rules.

What was the deviation here? The problem with `sqrt12` is R doesn't know when the function ends and the input begins. Unlike a human, R can't tell what you mean by context clues.

Parentheses tell R when to START listening for input, and when to STOP listening for input. Thus, for every function we need parentheses, *and* for every `(` (start) we must have a `)` (stop).

All functions have parentheses that separate the function name from its ***input***. (Given this, we often will refer to functions with parentheses too, `sqrt()`, to remind ourselves).

Okay okay, that was a lot about language, I thought we were gonna do cool data plotting stuff?

Yes yes, back to programming. There's one more fundamental to cover before we get to the cool data plotting stuff.

#### Sequential functions

What if I don't want all those extra decimal places for `sqrt(12)`?
Thankfully there is a `round()` function:

```{r}
round(3.464102)
```

Okay but here I copy-pasted `3.464102` from above. That's fine, but it's
not a systematic or dynamic approach. That is, what if I wanted to change
to round the square root of 13 instead? I'd have to do multiple steps.

One way to streamline this is by embedding one function in the other.
This is where parentheses *really* come in handy.

```{r}
round(sqrt(12))
```

Remember order of operations? We must resolve inside-out of parentheses.
First, we must take the square root of 12. The *output* of `sqrt(12)`
then becomes the *input* to `round()`, like this:

![Sequence of functions](sequence_of_functions.png)

Here, I'm using the orange rectangle to symbolize the output of the first
part.

So we can read `round(sqrt(12))` as "round the square root of 12".

But notice the figure shows it in a slightly different order, like "take
12, find the square root of it, and then round *that*" to get 3.

A lot of people prefer thinking about this sequentially, rather than
having to deal with inside-out parentheses (which can get messy!). This
sequential method is often called 'piping' or 'chaining', and is common
in other data languages like SQL. R can do piping too!

##### Going to the library

First we need to load some extra functionality in. We do this by
installing a package of functions, downloaded from the internet, then .
We are going to explore the increasingly popular `tidyverse` packages.
While we're at it, let's also install `ggformula` which will make our
plotting a bit simpler for this demo.

```{r echo=FALSE}
install.packages('tidyverse') # once you have run this one time, you can
comment this out (put # before 'install') or delete the line
install.packages('ggformula') # once you have run this one time, you can
comment this out (put # before 'install') or delete the line
install.packages('mosaic') # once you have run this one time, you can
comment this out (put # before 'install') or delete the line
library(tidyverse) # load into library (this part you keep and run each
time you open up R)
```

~~~
library(ggformula) # load into library (this part you keep and run each time you open up R)
library(mosaic) # load into library (this part you keep and run each time you open up R)
```
~~~

You only have to run `install.packages()` *once* on a given computer, until you upgrade your version of R or the package itself. Think of it like an app on your phone! (Packages do sometimes get updated, in which case running `install.packages()` again will update for you. Again, think of it like manually updating an app on your phone.)

You *do* have to run the `library()` function each time you start a new R session (open up the app again). The reason for this is really to save memory on your computer -- you manually tell R which functions you want to load in, that way when you first open R it doesn't load in all the possible packages you could have (which would slow it down and cause some other potential issues).

We can see the benefit of piping here:

~~~
```{r}
12 %>% sqrt() %>% round() # note we keep the () to show which are functions
```
~~~

"Take 12 *and then* find its square root *and then* round it", where `%>%` is the pipe symbol, which we read as "and then" (there's a longer story about this involving a Belgian surrealist painter; ask me sometime).

And we will often hit the 'enter' key to break these sequences into meaningful steps:

~~~
```{r}
# take the number 12
12 %>% # AND THEN (don't stop yet)
  sqrt() %>% # find its square root AND THEN (don't stop yet!)
  round() # round *that*
```
~~~

Here each function takes as its input whatever comes before it -- so we create a chain of events. In other words, we 'pipe' the previous line's output as input to the current line. That may seem a bit odd at first, but we'll see its usefulness more clearly below. We are just practicing with a simple case for now.

I can hear you saying "Okay, that's cool and all, but you promised us cool plots!"

Well, we now have enough of a foundation that we can at least see the 'big picture' of the following code. If we had more time (like, ahem, in a class this Fall!), I'd explain every step, but for now, we are going to start moving a bit faster.

## Penguin time

First, let's get some cool data. One more package:

```{r}
install.packages('palmerpenguins') # once you have run this one time, you
can comment this out (put # before 'install') or delete the line
library(palmerpenguins)
```

We have data!

```{r}
penguins
```

Notice we have a special kind of output here. Specifically, `penguins` is called a `data.frame`, which is a special type of format in R where data is stored like a well-organized spreadsheet. Here, each column is a variable and each row is a specific penguin. You can read more about this amazing data by running this:

```{r}
?penguins
```

### Exploring

We can explore the kind of data in a few ways. Here are some handy descriptive statistics for each column of `penguins`:

```{r}
summary(penguins)
```

Note that above we see there are three levels of `species` (Adelie, Chinstrap, Gentoo -- see the drawings!) and they're found on 3 `island`s. The numbers represent how many observations (rows) there are of each. Cool!

For numeric columns (most), we get an overall summary of their distribution, with the minimum (Q0), 1st quartile (Q1), median (Q2), 3rd quartile (Q3), and maximum (Q4), along with the arithmetic mean and how many missing values there are (2 NA's for the measurements).

#### Piping Palmer's penguins

Let's take a look at a couple other handy things we can do with our data. What if I wanted to know how many penguins of each species had a bill length above 45 mm?

Well, we saw above the frequency of each species. We can also get this using:

```{r}
penguins %>% # take penguins AND THEN
    count(species) # count up each species
```

But how would we do this for only those with bill length over 45 mm? This is where piping gives us a lot of power to explore our data. First, let's see how `filter()` works:

```{r}
# we can start with this:
penguins %>% # take penguins AND THEN
    filter(bill_length_mm > 45) # filter only the rows meeting this
criterion
```

Okay that shows us the **rows** that meet our criterion (`bill_length_mm > 45`) and excludes the rest (filters *out*!)

So we saw how to `filter()` and how to `count()`. Now we can *chain* these two commands together into a sequential function call:

![Sequence of functions: Penguin](sequence_of_functions_penguins.png)

```{r}
penguins %>% # take penguins AND THEN
    filter(bill_length_mm > 45) %>% # filter by our criterion AND THEN
    count(species) # count up each species FROM THOSE FILTERED ROWS
```

Interesting! Not so many Adelie with bills > 45 mm...

Okay but what about plotting?

### Visualizing

Let's visualize how flipper length (`flipper_length_mm`) relates to bill length (`bill_length_mm`):

```{r}
# we read this as:
#   take penguins AND THEN
#   make a point plot (scatterplot)
#   of bill_length_mm on the y-axis
#   and flipper_length_mm on the x-axis
#    (the ~ symbol is for a formula, y ~ x)
penguins %>%
    gf_point(bill_length_mm ~ flipper_length_mm)
```

The `gf_point()` function combines the very popular `ggplot2` method of visualizing data with a convenient "formula" interface (indicated by the `~`), thus "gf".

What do you notice about the plot?

-   What's the typical bill length? How much variation?
-   What's the typical flipper length? How much variation?
-   Is there a trend (relationship) between bill and flipper lengths?
-   Are the points 'clustered' in certain areas?

### Describing

What's the typical bill length? How much variation?

If I told you I saw a penguin, and I wanted you to guess its bill length (the closer your guess, the more candy you get!), what would your guess be?

One way to 'minimize error' (maximize candy) in your guess is to find the mean.

```{r}
# we read this as:
#   print out some of our favorite stats
#   for bill_length_mm
#   from the data.frame penguins
favstats(~bill_length_mm, data = penguins)
```

If you guessed around 44, yay, you have the best chance at maximizing your candy intake! (We'll leave the proof of this to a stats class.)

But of course, bill length varies a lot! (Notice *SD* $\approx$ 4.5.)

We can see the variation by plotting the mean as a horizontal (red) line **on top of** our scatterplot. We use piping to add plots on plots (on plots):

```{r}
# take the plot from before AND THEN
penguins %>%
  gf_point(bill_length_mm ~ flipper_length_mm, data = penguins) %>%
  # add the mean of bill length as a horizontal line (hline) that is
dashed and red
  gf_hline(yintercept = 43.92, linetype = 'dashed', color = 'darkred')
```

Means can be deceiving, but it's a start!

What's the typical flipper length? How much variation?

```{r}
# copy paste favstats() from above and replace with flipper length...
```

~~~
favstats(~flipper_length_mm, data = penguins)
```

And we can also add that as a vertical line (change "MEANHERE" below to
the mean we found):

```{r}
# take the plot from before AND THEN
penguins %>%
  gf_point(bill_length_mm ~ flipper_length_mm, data = penguins) %>%
  gf_hline(yintercept = 43.92, linetype = 'dashed', color = 'darkred')
%>%
  # add the mean of flipper length as a vertical line (vline) that is
dashed and orange
  gf_vline(xintercept = 200.92, linetype = 'dashed', color =
'darkorange')
```

Notice that we don't really have many penguins around these means!
Interesting...

### Modeling

*Is there a trend (relationship) between bill and flipper lengths?*

We see most dots are in the lower-left and upper-right quadrants, with
some in upper-left but almost none in lower-right. Broadly, it looks like
a positive relationship (as flipper length increases, so does bill
length).

Of course, R can help us by adding a line of best fit:

```{r}
# take the plot from before AND THEN
penguins %>%
  gf_point(bill_length_mm ~ flipper_length_mm, data = penguins) %>%
  gf_hline(yintercept = 43.92, linetype = 'dashed', color = 'darkred')
%>%
  gf_vline(xintercept = 200.92, linetype = 'dashed', color =
'darkorange') %>%
  # add a line of best fit ('lm' for linear model)
  gf_lm()
```

Cool! Notice where that line goes?

Does the line look like it has a positive slope? Is it meaningfully
different from a flat (horizontal) slope?

We already have a horizontal line at the mean of `bill_length_mm`. Notice
it's quite different from the blue line!
~~~

~~~
Of course, if we wanted to quantify how different these lines are from
each other, that's where ***inferential statistics*** comes in!  But I'm
going to skip that for time -- take a stats class (or R class or both!)


### More modeling

*Are the points 'clustered' in certain areas?*

You know, it's really bugging me that those dots in the upper-left feel
'off'. And remember how few Adelie penguins had longer bills? And
remember how few penguins were at the means?  These patterns suggest
there may be a 'hidden' structure in our data.

We can examine bill length *by* species with our old `favstats()`
function:

```{r}
# remember that the formula is y ~ x
favstats(bill_length_mm ~ species, data = mm)
```

Ah, Adelie looks pretty different from the others.

Numbers are cool, I love them, but let's see the species on our *plot*:

```{r}
#  here let's just modify the scatterplot
penguins %>%
  gf_point(bill_length_mm ~ flipper_length_mm,
         color = ~species, # set color and shape to vary by species
         shape = ~species) # set shape to vary by species
```

Ohhhhhhhhhhhhh. I see. This explains the cluster in the upper-left AND
why so few Adelie penguins had longer bills. There is ***speciation*** on
those islands! (Cue suspenseful music.)

#### Fancier plot

I get a bit bored of ggplot's default colors, let's use fun colors like
the rad drawing above.  Here's how we can customize our plots, to give
you a small taste of the visual power of R!

```{r}
penguins %>% # take the penguins data frame and
  # make a scatterplot with flipper_length_mm on x-axis and
bill_length_mm on y-axis
  gf_point(bill_length_mm ~ flipper_length_mm,
         color = ~species, # set color and shape to vary by species
         shape = ~species) %>% # set shape to vary by species
  # customize (refine) the plot!
  gf_refine(
    scale_color_manual(values =
~~~

~~~~
    c(Chinstrap = "#D41876", Gentoo = "#006C65", Adelie = "#F27200"))
     #   (notice I use hex codes here, you can find your own!)
  ) %>%
  # and make it look a bit cleaner by using theme_bw(), which I like
  gf_theme(theme_bw()) %>%
  # and use some better labels
  gf_labs(
    x = "Flipper length (mm)", # x axis
    y = "Bill length (mm)", # y axis
    title = "Yay penguins!", # figure title
    subtitle = "R @ OUR!", # subtitle
    caption = "We made this at OUR Workshop." # and caption
  )
```

Now that's pretty!

But wait! One more thing...

Does each species seem to have the same relationship between bill and
flipper lengths?  That is, is the slope we found above equally
appropriate for all 3 species? What if we estimated the slope for each?

#### Different slopes for different folks (penguins)

```{r}
penguins %>% # take the penguins data frame and
  # make a scatterplot with flipper_length_mm on x-axis and
bill_length_mm on y-axis
  gf_point(bill_length_mm ~ flipper_length_mm,
          color = ~species, # set color and shape to vary by species
          shape = ~species) %>% # set shape to vary by species
  # customize (refine) the plot!
  gf_refine(
    scale_color_manual(values =
    c(Chinstrap = "#D41876", Gentoo = "#006C65", Adelie = "#F27200"))
     #   (notice I use hex codes here, you can find your own!)
  ) %>%
  # and make it look a bit cleaner by using theme_bw(), which I like
  gf_theme(theme_bw()) %>%
  # and use some better labels
  gf_labs(
    x = "Flipper length (mm)", # x axis
    y = "Bill length (mm)", # y axis
    title = "Yay penguins!", # figure title
    subtitle = "R @ OUR!", # subtitle
    caption = "We made this at OUR Workshop." # and caption
  ) %>%
  gf_lm()
```

Now we see that with very simple R code, we can find *separate* lines of
best fit for each of the 3 species.
~~~~

Looking at the plot above: Is the relationship between flipper and bill length different across species?

We can visually see that each slope looks positive, and the slopes look different from each other. But the latter part is hard to really know for sure. That is, until we get numbers involved... and for that, you really will need statistics!

Maybe we can do a Part 2? Let us know!

## A data report

Putting it all together, in this document have a mixture of R code, its output (including images!), and some formatted text (this is called `markdown`, which is a simple way of marking the format of text -- you've seen it in online message boards like in Canvas). We even used some $\LaTeX$ formatting (anything I put in between `$` symbols above, which renders it as 'fancy math').

We will send you a downloadable version of this file in a "Quarto" format which works well in the free RStudio program.

You can then use and edit this Quarto `.qmd` file to your heart's content (heaRt's content? OK I'll stop.)

But what if you want to show it off to your best friends (or enemies who are afraid of penguins) who may not know anything about R (yet)?

Well, one more handy thing about *both* Jupyter Notebooks and Quarto documents (perhaps the handiest part!) is they can easily export to different file formats, like HTML and PDF.

At the very tippy-top of this document (the first 15ish lines), you'll see a section of code that isn't R or markdown. It's something else called YAML (don't worry about it, but you can read more here: <https://quarto.org/docs/authoring/front-matter.html>). I've customized it for me, but one important part you'll see is that the `format` is set to HTML (the language of websites), which means it'll make a nicely formatted interactive webpage (note: one that is stored locally, not on the internet).

That means we can turn our data exploration into a *data report* that anyone can read on their computer (because everyone has a browser) even if they don't have R.

NOTE: before doing this, you'll want to make sure you commented out the syntax error I left toward the top (`sqrt12`) and all of the `install.packages()` calls. The reason for this is when you go to make the HTML document, R will re-run the entire document -- when it sees an error it will stop, and when it sees `install.packages()` it will re-download the packages each time, and that's unnecessary.

Ready? Okay, let's go ahead and hit the **Render** button in RStudio (top-left-ish with the blue arrow pointing rightward).

Once you do, you'll get something that I think is pretty cool! Try it out in your browser of choice, share it with a friend/enemy/frenemy, and thanks for having me demo R today!