Intro to R

Cody Martin

5/18/2021

Software	Version
R	4.0.4
Rstudio	1.2.1335

Contents

  * ggplot boxplots
  * ggplot barplots
    - simple ggplot barplots
    - stacked ggplot barplots
    - grouped ggplot barplots
  * ggplot histograms

Setup

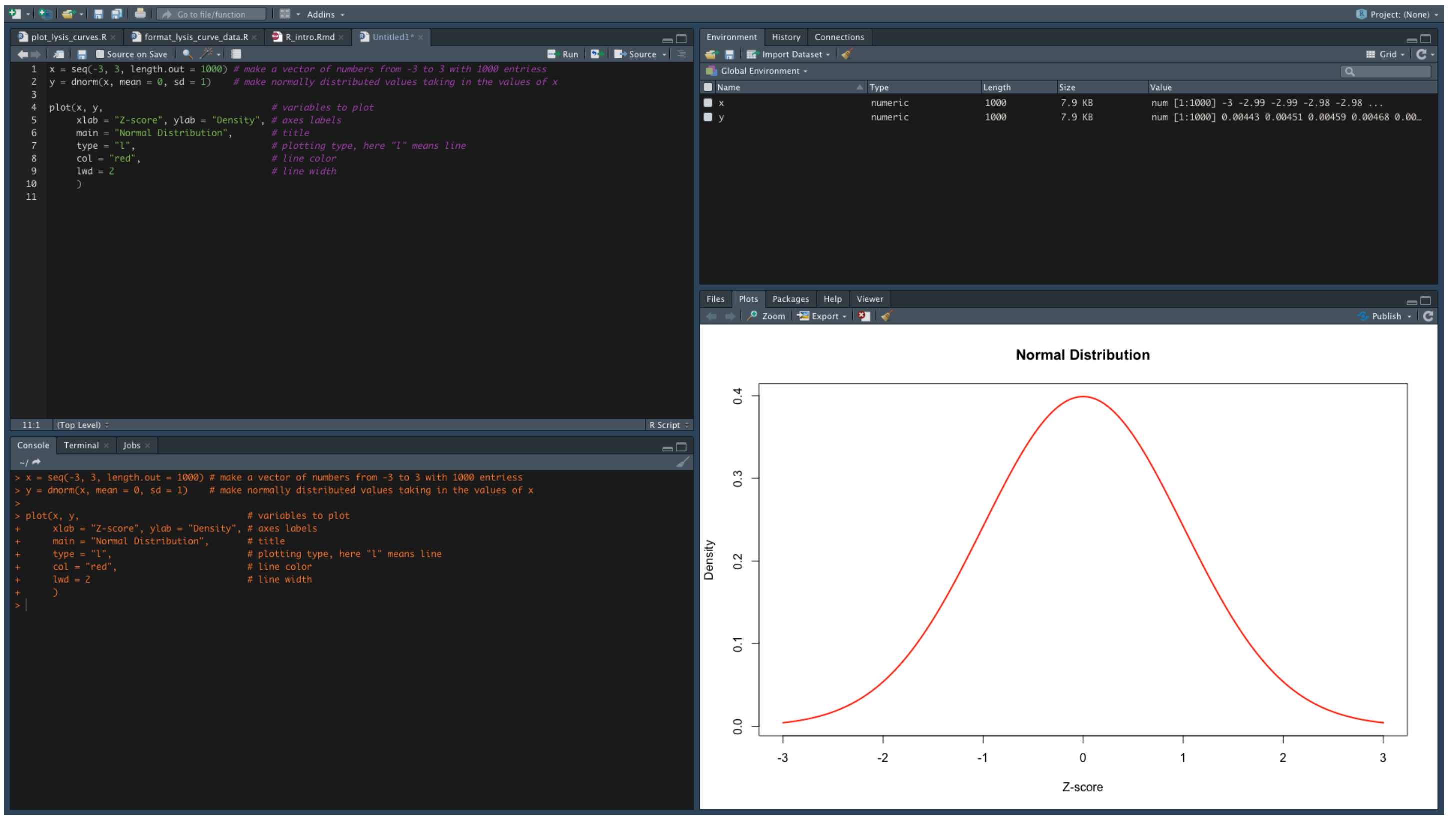
```r
library(tidyverse)
```

Introduction

Downloads

You can download the latest version of R here: https://cran.microsoft.com.
The best way to use R is with Rstudio, which you can download here: https://www.rstudio.com/products/rstudio/download/.

Here is why Rstudio is so powerful:



```r
x = seq(-3, 3, length.out = 1000) # make a vector of numbers from -3 to 3 with 1000 entriess
y = dnorm(x, mean = 0, sd = 1)    # make normally distributed values taking in the values of x

plot(x, y,                             # variables to plot
     xlab = "Z-score", ylab = "Density", # axes labels
     main = "Normal Distribution",       # title
     type = "l",                         # plotting type, here "l" means line
     col = "red",                        # line color
     lwd = 2                             # line width
     )
```

```
Name  Type     Length  Size    Value
x     numeric  1000    7.9 KB  num [1:1000] -3 -2.99 -2.99 -2.98 -2.98 ...
y     numeric  1000    7.9 KB  num [1:1000] 0.00443 0.00451 0.00459 0.00468 0.00…
```

```
> x = seq(-3, 3, length.out = 1000) # make a vector of numbers from -3 to 3 with 1000 entriess
> y = dnorm(x, mean = 0, sd = 1)    # make normally distributed values taking in the values of x
>
> plot(x, y,                             # variables to plot
+      xlab = "Z-score", ylab = "Density", # axes labels
+      main = "Normal Distribution",       # title
+      type = "l",                         # plotting type, here "l" means line
+      col = "red",                        # line color
+      lwd = 2                             # line width
+      )
> 
```

Figure 1: Rstudio_screenshot

- **Top left: Scripting** You can write scripts or import scripts to use with your own data. Here I've made a simple script to plot a normal distribution.

- **Bottom left: Console** Where you run R commands. If you are using a script, you can just run commands from the script itself. If you are a MacOS user, you will also have access to your UNIX Terminal.

- **Top right: Environment** Your environment variables will be displayed. Here you can see an `x` and `y` variable.
- **Bottom right: Multipurpose** You can see:
  - Files in a given directory
  - **Plots / outputs**
  - Installed packages

  - Help instances for a given R commmand
  - Markdown outputs in "Viewer"

Having all of these windows combined together in a compact display makes using R super powerful and efficient.

Basic Commands

R is based on UNIX, so its syntax is **case-sensitive**. Additionally, to run commands in your scripting window you can do a few things:

1. Move the cursor to a line / field of code. `CMD+RET` (or `CTRL+ENTER`) will run an entire line or block.
2. There is also a `run` button at the top of the scripting window.
3. You can also highlight blocks of code you want to run and use the above 2 methods.

Help

If you ever need to know more about the usage of a function, there are two options:

1. `?function`
2. `help(function)`

Thus, if I wanted to know all the details of the `mean` function to get the arithmetic mean, I could do `?mean`.

Math

Since R was initially developed for statistics, it works very well with standard mathematical operations.

```r
1 + 2
```

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

```r
33 / 11
```

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

```r
8 * 15
```

```
## [1] 120
```

Variable assignment

Like other object-oriented programming languages, you can assign values to variables. Most languages use the `=` symbol to assign variables. You will likely find that many R tutorials use the `<-` operator, but `=` will also work in 99% of the cases. Personally, I use `=` since it is only one keystroke.

Note: there is a shortcut for `<-`

- MacOS: `Alt+-`
- Windows: `Option+-`

```r
x = 12
y <- "apple"
print(x)
```

```
## [1] 12
```

```r
print(y)
```

```
## [1] "apple"
```

The only things that `<-` can do that `=` can't do are:

1. Directional assignment
2. Multiple assignment in one line

```r
# Directional assignment...why would you do this
"a" -> z
print(z)
```

```
## [1] "a"
```

```
# Assign both x and y the value of 2
x <- y <- 2
print(x)
```

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

```
print(y)
```

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

Note: Most variable names containing alphanumeric symbols, `-`, and `.` are valid. However, the variable name cannot start with a number. There are also special commands that you should avoid naming variables like `if`, `for`, `c`, `T`, `mean`, etc.

Data structures

Common data structures you will find in R include:

- Vectors of type:
  - `Character` / String /
  - `Factor` (categorical variable)
  - `Numeric` (`float` or `double` in other languages)
  - `Integer`
  - `Logical` or `boolean`
- Matrices
- Data frames
- Lists

Vectors

Vectors are collections of values of the *same* type. For example, you can have a vector of only integers, numbers with decimals, or strings. You can think of both data frames and matrices are collections of vectors, so it is imperative that you are comfortable with R vectors.

You can create vectors using the `c()`. Think *concatenate* when joining individual elements into a single vector.

You can think of vectors like lists, but that is not technically correct since R also has a list data structure. They are more akin to single column/row matrices. It is more like a linear algebra vector.

```
int_vec = c(1, 2, 3, 4, 5) # Notes commas in between entries
char_vec = c("a", "b", "tomato", "Cody") # single letters and words - all must be in quotes
num_vec = c(1.2, exp(1), 8/3)

print(int_vec)
```

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

```
print(char_vec)
```

```
## [1] "a"      "b"      "tomato" "Cody"
```

```
print(num_vec)
```

```
## [1] 1.200000 2.718282 2.666667
```

Vector Operations

When you have numeric vectors, you can extend math operations to the vector elements. These operations will be performed elementwise.

```
# Multiply each element by a number
x = c(1, 2, 3, 4)
x * 2
```

```
## [1] 2 4 6 8
```

```r
# Add 2 vectors of the same length elementwise
y = c(3.2, 1.9, 7.8, 2.2)
x + y
```

```
## [1]  4.2  3.9 10.8  6.2
```

```r
# Multiply elements of 2 vectors
x * y
```

```
## [1]  3.2  3.8 23.4  8.8
```

You can also start to statistical operations on numeric vectors. Most functions in R are aptly named, and Rstudio will suggest functions for you to use as you type.

```r
x = c(1, 2, 3, 4)
mean(x)
```

```
## [1] 2.5
```

```r
sd(x)
```

```
## [1] 1.290994
```

```r
median(x)
```

```
## [1] 2.5
```

It is very important in R to make sure your vectors are of the type that you intended. Since vectors can only be of one type, R will coerce vectors to a class that is compatible with all the entries.

```r
x = c(1, 2, 3, "4") # I added a character "4" instead of the integer 4
print(x)
```

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

```r
class(x)
```

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

Because `x` is a `character` vector, an operation like `x * 2` will not work! Fortunately, you can also force R to coerce vectors to a different type if it is able.

```r
x = c(1, 2, 3, "4")
x = as.integer(x)
class(x)
```

```
## [1] "integer"
```

```r
x * 2
```

```
## [1] 2 4 6 8
```

Subsetting vectors

You can access different elements of a vector, which is subsetting. There are 2 major points to note with subsetting R vectors:

1. R starts counting at 1! Other languages will start at 0, so keep this in mind.
2. The subsetting notation for **vectors** is `vector[#]`. Note the single `[]`. Subsetting R lists uses the double `[[]]`.

```r
x = c(1, 2, 3, 4)
x[2]
```

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

```r
# subset multiple entries with more vector notation!
x[c(1,3)]
```

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

```r
# conditional subsetting
x[c(TRUE, FALSE, FALSE, TRUE)] # can use T and F shorthand too
```

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

```
x[x > 2] # only values in x that are greater than 2
```

```
## [1] 3 4
```

```
x[x == 3] # only values in x that are equal to 3
```

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

You can also combine vectors together using the same `c()` command. This is because the elements in a vector are also vectors! They are treated a single entry vectors.

```
x = c(1, 2, 3)
y = c(4, 5, 6)
c(x,y)
```

```
## [1] 1 2 3 4 5 6
```

Other useful functions for vectors include:

```
# Make a sequentially ordered numeric vector from a to b
# You can also adjust the step size or how many elementts you want
a = 1
b = 5
step = 0.5
seq(a, b, step)
```

```
## [1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0
```

```
seq(a, b, length.out = 10)
```

```
##  [1] 1.000000 1.444444 1.888889 2.333333 2.777778 3.222222 3.666667 4.111111
##  [9] 4.555556 5.000000
```

```
# Short hand sequential vector counting by 1
1:5
```

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

```
# repeat values
rep("a", 3)
```

```
## [1] "a" "a" "a"
```

```
rep(1:3, 3)
```

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

```
rep(1:3, each = 3)
```

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

```
# get length of a vector
length(1:10)
```

```
## [1] 10
```

Factor vectors

When you are working with categorical data, R has a special data class called `Factor`. If you have categories that are composed of `character` strings, R will understand and coerce those to `Factor` if necessary. However, if you have numerical catergories, R will not coerce those to `Factor` unless told to do so.

A given factor vector will be composed of unique levels. R will usually report the levels in alphanumeric order, which will affect subsequent commands like plotting.

```
colors = rep(c("red", "blue", "green"), 3)
as.factor(colors)
```

```
## [1] red   blue  green red   blue  green red   blue  green
## Levels: blue green red
```

```
my_categories = rep(0:2, 3)
class(my_categories) # notice it is of class integer
```

```
## [1] "integer"
```

```
# but what if 0, 1, and 2 are just categories
as.factor(my_categories)
```

```
## [1] 0 1 2 0 1 2 0 1 2
## Levels: 0 1 2
```

You can also reorder the levels of factor vectors.

```
colors = factor(colors, levels = c("red", "green", "blue"))
colors
```

```
## [1] red   blue  green red   blue  green red   blue  green
## Levels: red green blue
```

Matrices

Matrices are collections of vectors, so now there are 2 dimensions. Matrices, like vectors, can only be of a single type, which is usually `integer/numeric`. Since R is for use with math, it has easy-to-use matrix functions. Typically matrices are built from smaller vectors.

```
x = 1:5
y = 6:10
z = 11:15

# make a 3x5 matrix where each individual vector is a matrix row
# Note here that the order matters!
matrix(data = c(x, y, z), nrow = 3, ncol = 5, byrow = T)
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    2    3    4    5
## [2,]    6    7    8    9   10
## [3,]   11   12   13   14   15
```

You can also just combine vectors using the functions `cbind` (column bind) and `rbind` (row bind).

```
x = 1:5
y = 6:10
z = 11:15

# Note here that the order matters!
cbind(x,y,z)
```

```
##      x  y  z
## [1,] 1  6 11
## [2,] 2  7 12
## [3,] 3  8 13
## [4,] 4  9 14
## [5,] 5 10 15
```

```
rbind(x,y,z)
```

```
##   [,1] [,2] [,3] [,4] [,5]
## x    1    2    3    4    5
## y    6    7    8    9   10
## z   11   12   13   14   15
```

You can also combine matrices, or vectors and matrices using the same `cbind` and `rbind` commands.

Subsetting matrices

Since matrices are collections of vectors, you can subset them the exact same way. You just take into account that there are two dimensions, you need to subset in both dimensions.

```
x = 1:5
y = 6:10
z = 11:15

m = matrix(data = c(x, y, z), nrow = 3, ncol = 5, byrow = T)

m
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    2    3    4    5
## [2,]    6    7    8    9   10
## [3,]   11   12   13   14   15
```

```
# Cell at row 2, col 3
m[2,3]
```

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

```
# Row 3
m[3, ]
```

```
## [1] 11 12 13 14 15
```

```
# Column 5
m[, 5]
```

```
## [1]  5 10 15
```

If you've done any linear algebra, you know that you can do matrix operations as well. These functions are all present in R, like the determinant `det()` and `%*%` for matrix multiplication.

Data frames

Like matrices, data frames are collections of vectors. Unlike matrices, however, the vectors that make up a data frame do not all have to be of the same type, meaning you can have a data frame consisting of `character`, `logical`, and `numeric` vectors. Data frames can use many of the same operations as matrices.

When you combine vectors into a data frame using the `data.frame()` commmand, the standard convention is that each combined vector will be a column, and they must all be of the same length.

```
x = 1:3
y = c("a", "b", "c")
z = c(T, F, F)

data.frame(x, y, z)
```

x	y	z
1	a	TRUE
2	b	FALSE
3	c	FALSE

Here is a familiar looking data frame you might work with of A550 values from a lysis curve.

Time	Vector	Lyser	Nonlyser
0	0.200	0.200	0.20
5	0.308	0.233	0.25
10	0.417	0.267	0.30
15	0.525	0.300	0.35
20	0.633	0.333	0.40

Time	Vector	Lyser	Nonlyser
25	0.742	0.367	0.45
30	0.850	0.400	0.50
35	0.958	0.400	0.55
40	1.067	0.322	0.60
45	1.175	0.244	0.65
50	1.283	0.166	0.70
55	1.392	0.088	0.75
60	1.500	0.010	0.80

Subsetting data frames

You can subset data frames the exact same way as matries using the `[i,j]` notation. In addition, since the columns (and sometimes rows) are named in data frames, you can also make use of that to subset a data frame.

```
x = 1:3
y = c("a", "b", "c")
z = c(T, F, F)

df = data.frame(x, y, z)

df[1,2]
```

```
## [1] "a"
```

```
df[1,]
```

x	y	z
1	a	TRUE

```
df[1:2,1:2]
```

x	y
1	a
2	b

```
df[,2] # column 2
```

```
## [1] "a" "b" "c"
```

```
df$y   # column 2 is also named "y" so you can grab it this way too
```

```
## [1] "a" "b" "c"
```

```
# set row names of our data frame
rownames(df) = c("a", "b", "c")
df
```

	x	y	z
a	1	a	TRUE
b	2	b	FALSE
c	3	c	FALSE

```
df["a",] # use the row name
```

	x	y	z
a	1	a	TRUE

Other commands to subset data frames and matrices are the `head` and `tail` commands. These will show the top `n` rows or bottom `n` rows, respectively. The default number of rows is 6.

```r
# data frame with 100 rows
x = 1:100
y = rnorm(100)
z = seq(0,500,length.out=100)

df = data.frame(x,y,z)

head(df)
```

x	y	z
1	1.3108163	0.000000
2	-0.9749727	5.050505
3	2.5730180	10.101010
4	-1.0784715	15.151515
5	-0.5649932	20.202020
6	-1.0131581	25.252525

```r
tail(df)
```

	x	y	z
95	95	0.8161620	474.7475
96	96	-1.1303912	479.7980
97	97	0.2703901	484.8485
98	98	0.1804100	489.8990
99	99	-0.2965986	494.9495
100	100	1.0455985	500.0000

```r
# show first 10 rows instead of first 6
head(df,10)
```

x	y	z
1	1.3108163	0.000000
2	-0.9749727	5.050505
3	2.5730180	10.101010
4	-1.0784715	15.151515
5	-0.5649932	20.202020
6	-1.0131581	25.252525
7	0.0198383	30.303030
8	-0.2766765	35.353535
9	-1.0547649	40.404040
10	-1.8538550	45.454546

Renaming rows and columns

Often when you import data into a `data.frame` object, the column names will be messed up since R like them to not have spaces or special characters. It is easy to rename them since the row and column names are also. . . wait for it. . . vectors! You just need to pass a new vector of names to `colnames(data.frame)`.

```r
x = 1:3
y = c("a", "b", "c")
z = c(T, F, F)
df = data.frame(x, y, z)
colnames(df) # notice they are the same as the vector variables
```

```
## [1] "x" "y" "z"
```

```r
colnames(df) = c("red", "yellow", "blue")
df
```

red	yellow	blue
1	a	TRUE
2	b	FALSE
3	c	FALSE

You can also rename rows analogously. A common example would be to change a column of names to be the row names.

```r
Gene = c("gp01", "gp02", "gp03", "gp04")
Log2FC = rnorm(4, mean=0, sd=1)
Fitness = runif(4, min=0, max=1)

gene_data = data.frame(Gene, Log2FC, Fitness)
gene_data
```

Gene	Log2FC	Fitness
gp01	-1.5551809	0.7865708
gp02	0.2922220	0.6885144
gp03	-0.4712069	0.9013048
gp04	0.0427157	0.5812105

```r
# make rownames by the Gene name
rownames(gene_data) = gene_data$Gene

# remove Gene column
gene_data$Gene = NULL
gene_data
```

	Log2FC	Fitness
gp01	-1.5551809	0.7865708
gp02	0.2922220	0.6885144
gp03	-0.4712069	0.9013048
gp04	0.0427157	0.5812105

Other dataframe functions

These functions also apply to matrices as well.

```r
# get dimensions of a dataframe or matrix
dim(df)
```

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

```r
nrow(df)
```

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

```r
ncol(df)
```

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

```r
# apply a function to a column or row
mat = matrix(1:9, nrow=3)
apply(mat, # df or matrix
      1,  # 1 = rows, 2 = columns
      mean # function to apply
      )
```

```
## [1] 4 5 6
```

Lists

I like to think of lists as super vectors. Like vectors, they are composed of different elements. Unlike vectors, those elements can be any data structure. For example, you can have a list of a data frame, matrix, and a vector.

Lists are very important since they keep information for an object combined together in a hierarchical manner. Many packages will return list objects with a ton of information in a single variable.

```r
vec = 1:5
df = data.frame(vec,rep(T,5))
mat = cbind(vec, 6:10)
mylist = list(vec,df,mat)
mylist
```

```
## [[1]]
## [1] 1 2 3 4 5
##
## [[2]]
##   vec rep.T..5.
## 1   1      TRUE
## 2   2      TRUE
## 3   3      TRUE
## 4   4      TRUE
## 5   5      TRUE
##
## [[3]]
##      vec
## [1,]   1  6
## [2,]   2  7
## [3,]   3  8
## [4,]   4  9
## [5,]   5 10
```

You can also name each element in a list.

```r
names(mylist) = c("vector", "dataframe", "matrix")
mylist
```

```
## $vector
## [1] 1 2 3 4 5
##
## $dataframe
##   vec rep.T..5.
## 1   1      TRUE
## 2   2      TRUE
## 3   3      TRUE
## 4   4      TRUE
## 5   5      TRUE
##
## $matrix
##      vec
## [1,]   1  6
## [2,]   2  7
## [3,]    3  8
## [4,]    4  9
## [5,]    5 10
```

Subsetting lists

This is a combination of both vector and dataframe subsetting. You can use `[[]]` to subset a list using the index numbers, or if you have a named list, you can use the `list$name` notation. Each element of the list can then be subset further.

```
# get 1st element
mylist[[1]]
```

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

```
# subset 1st element, a vector, further
mylist[[1]][3]
```

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

```
# grab elements by name
mylist$dataframe
```

vec	rep.T..5.
1	TRUE
2	TRUE
3	TRUE
4	TRUE
5	TRUE

```
# subset dataframe element further
mylist$dataframe[1:3,]
```

vec	rep.T..5.
1	TRUE
2	TRUE
3	TRUE

Other basic functions

Loops

R also has standard loops like `for` and `while` loops. Here is the syntax of an R `for` loop: `for (index in range) {command}`.

```
for (i in 1:5) {
  print(i)
}
```

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

`while` loops have similar syntax, but rely on a condition to continue looping.

Conditional statements

R also has `if`, `else`, and `else if` statements to conditionally execute functions.

```
x = 1:6

for (i in 1:length(x)) {
```

```r
  if (x[i] > 5) {
    print(paste(x[i], "greater than 5"))
  } else if (x[i] > 2) {
    print(paste(x[i],"greater than 2"))
  } else {
    print(paste(x[i],"tiny"))
  }
}
```

```
## [1] "1 tiny"
## [1] "2 tiny"
## [1] "3 greater than 2"
## [1] "4 greater than 2"
## [1] "5 greater than 2"
## [1] "6 greater than 5"
```

Reading in data

R has many functions for reading in data, but the most commonly used ones if you are using locally stored data are:

- `read.csv()`
- `read.delim()`
- `read.table()` These allow you to specify the filepath to your data, your data delimiter, if headers are present, etc. Use `?read.csv`, for example, to see all the arguments that can be passed.

Here is an example reading in data from a lysis curve with OD values per timepoint per sample.

```r
data = read.delim("lysis_curves/test1.txt", sep = "\t")
data
```

Time	A	B	C	D	E	F	G
0	0.223	0.239	0.237	0.231	0.228	0.235	0.231
30	0.523	0.488	0.489	0.505	0.510	0.506	0.504
35	0.620	0.558	0.441	0.505	0.550	0.503	0.510
40	0.685	0.612	0.212	0.354	0.517	0.332	0.379
45	0.772	0.683	0.110	0.222	0.428	0.176	0.244
50	0.844	0.718	0.089	0.137	0.371	0.118	0.174
55	0.888	0.789	0.067	0.111	0.354	0.098	0.136
60	0.968	0.815	0.068	0.105	0.346	0.095	0.126
70	1.390	0.859	0.051	0.091	0.331	0.108	0.111
80	1.544	0.962	0.053	0.068	0.151	0.086	0.074
93	2.010	1.224	0.048	0.070	0.125	0.064	0.084

Here, the `data` object is a `data.frame`. That is the usual object type when you read in tabular data.

R and many packages also have built-in datasets that can be called by using the variable name. If you've already assigned variable of the same name to a different value, you can create the correct variable using `data(dataset)`. One commonly used base R dataset is called `iris`. You can use `data(iris)` to create a variable named `iris` in your environment.

```r
head(iris)
```

Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
5.1	3.5	1.4	0.2	setosa
4.9	3.0	1.4	0.2	setosa
4.7	3.2	1.3	0.2	setosa
4.6	3.1	1.5	0.2	setosa
5.0	3.6	1.4	0.2	setosa
5.4	3.9	1.7	0.4	setosa

The base R cannot interpret MS Excel `.xlsx` documents, so remove that from your working use. Best practice is to use a universal file type like `.txt` `.csv` `.tsv` etc.

Installing and loading packages

R has many useful packages like `ggplot2` for making plots. To install any packages from the publicly available CRAN repositories, just use the command `install.packages("package_name")`. This only needs to be done once. Alternatively, you can use the `Tools` tab in the toolbar.

Once you've install a package, it needs to be loaded every time you open an R session. There are two ways to do this:

1. `library(package)` - This is preferred. Your code will immediately stop if the package hasn't been installed.
2. `require(package)` - Will not stop immediately if the package isn't installed.

You can check what packages you've installed in the bottom right panel when you click the packages tab.

Plotting

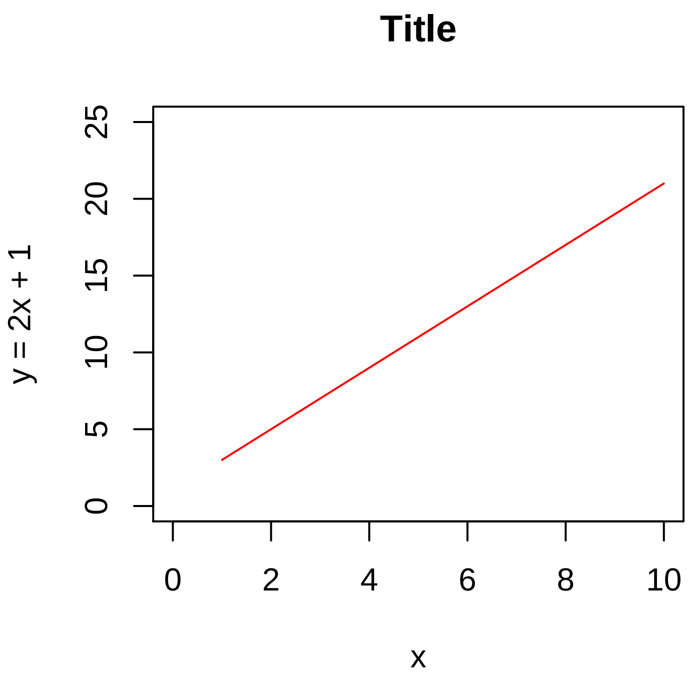
The basic R package has decent plotting functions. If you need a quick, simple plot, it can be useful. However, `ggplot2` is far superior to base R for many reasons incuding aesthetic, manipulation power, and layering details. I will go over both.

Base R plots

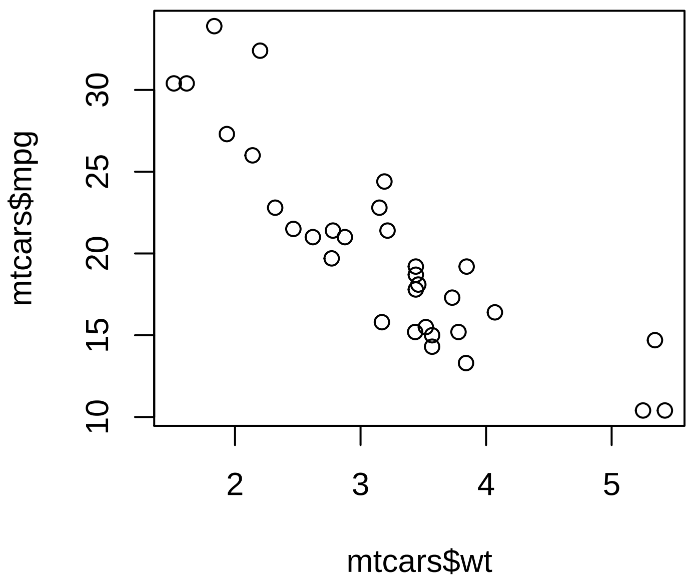
Scatterplots

Use the `plot` function to make a scatterplot.

```
plot(x = 1:10,              # x variable
     y = 2*(1:10) + 1,      # y variable
     type = "l",            # plot type - l=line, p=point, etc
     xlab = "x",            # x axis label
     ylab = "y = 2x + 1",   # y axis label
     main = "Title",        # plot title
     xlim = c(0,10),        # adjust x limits
     ylim = c(0, 25),       # adjust y limits
     col = "red"            # change point color - can take names and hex values
) # There are more parameters that you can adjust
```



```
# more data from base R mtcars dataset
plot(mtcars$wt, mtcars$mpg)
```

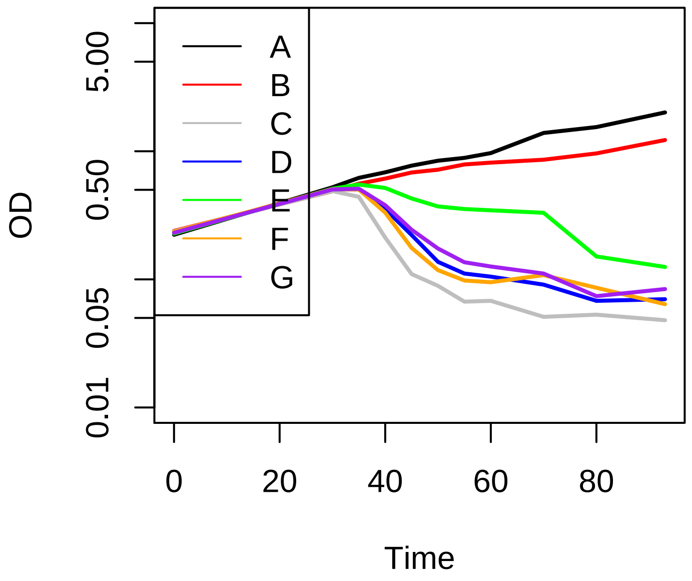


Here is where base R sucks - if you have multiple lines you want to plot, you will essentially need multiple lines of code to keep adding new lines. Let's return to my real lysis curve data.

```
data
```

Time	A	B	C	D	E	F	G
0	0.223	0.239	0.237	0.231	0.228	0.235	0.231
30	0.523	0.488	0.489	0.505	0.510	0.506	0.504
35	0.620	0.558	0.441	0.505	0.550	0.503	0.510
40	0.685	0.612	0.212	0.354	0.517	0.332	0.379
45	0.772	0.683	0.110	0.222	0.428	0.176	0.244
50	0.844	0.718	0.089	0.137	0.371	0.118	0.174
55	0.888	0.789	0.067	0.111	0.354	0.098	0.136
60	0.968	0.815	0.068	0.105	0.346	0.095	0.126
70	1.390	0.859	0.051	0.091	0.331	0.108	0.111
80	1.544	0.962	0.053	0.068	0.151	0.086	0.074
93	2.010	1.224	0.048	0.070	0.125	0.064	0.084

```
plot(data[,1], data[,2], type="l", xlab="Time", ylab="OD",
     xlim=c(0,max(data[,1])), ylim=c(0.01,10), log="y",
     lwd=2)
lines(data[,1], data[,3], col="red", lwd=2)
lines(data[,1], data[,4], col="gray", lwd=2)
lines(data[,1], data[,5], col="blue", lwd=2)
lines(data[,1], data[,6], col="green", lwd=2)
lines(data[,1], data[,7], col="orange", lwd=2)
lines(data[,1], data[,8], col="purple", lwd=2)

# of course we want a legend
legend("topleft", # position, you can also do (x,y) coordinates
       legend = colnames(data[,-1]), # Names in legend
       lty=1,
       col=c("black", "red", "gray", "blue", "green", "orange", "purple")) # feature colors
```

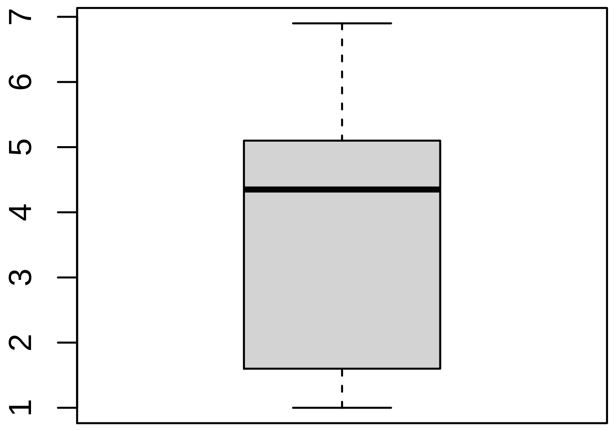


That is a brute force way to add multiple lines. You could be more sophisticated with a `for` loop, but that becomes too much work eventually.

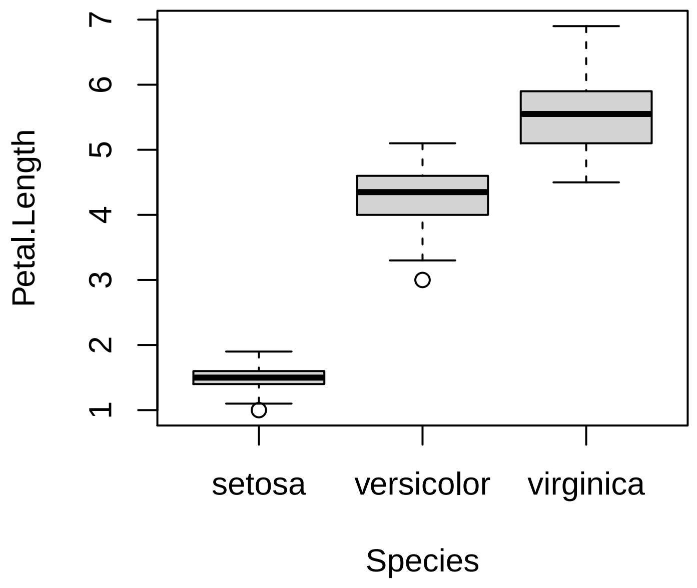
Boxplots

Many of the base plotting commands take similar arguments to change colors and other aesthetics, so now I will just focus on showing examples.

```r
# Simple
boxplot(iris$Petal.Length)
```

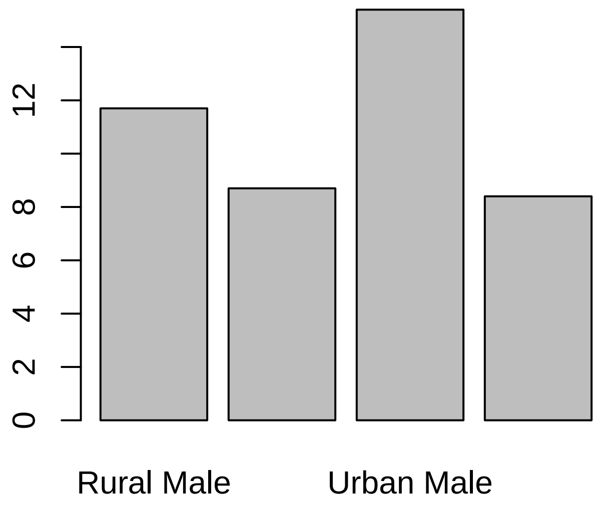


```r
# Also group by another variable
boxplot(Petal.Length ~ Species, data = iris)
```



Bar plots

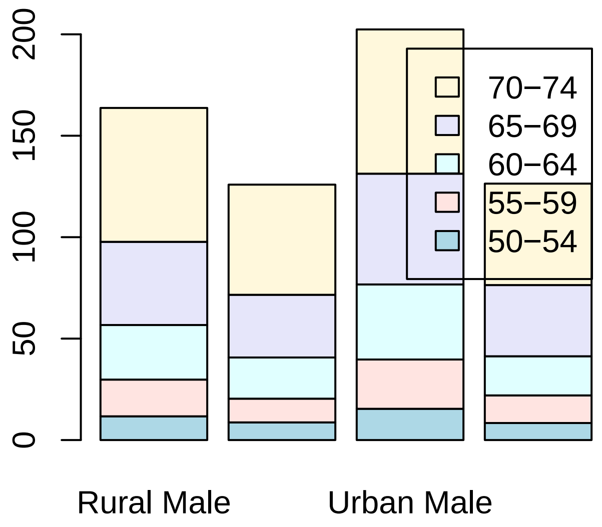
```r
# Plot VADeath rate among people aged 50-54
barplot(VADeaths[1,])
```



Stacked bar plot

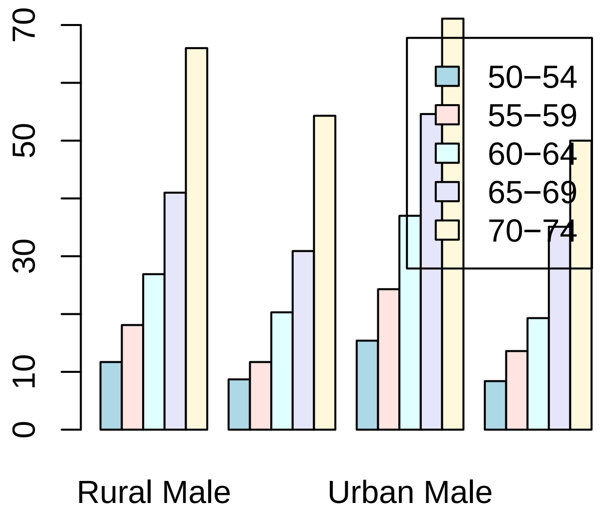
Note: you can also make relative stacked bar charts using proportions or percents, but it requires more effort with base R.

```r
barplot(VADeaths,
         col = c("lightblue", "mistyrose", "lightcyan",
                 "lavender", "cornsilk"),
        legend = rownames(VADeaths))
```



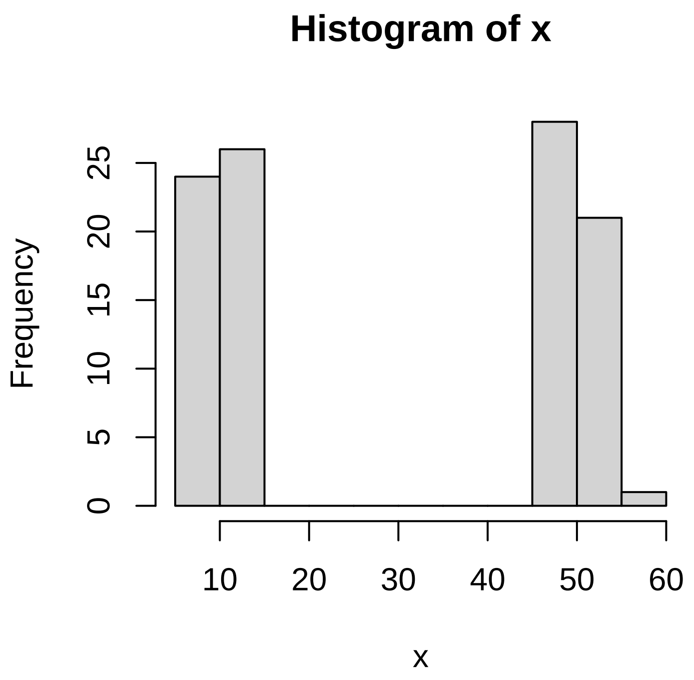
Grouped bar plot

```r
barplot(VADeaths,
         col = c("lightblue", "mistyrose", "lightcyan",
                 "lavender", "cornsilk"),
        legend = rownames(VADeaths), beside = TRUE)
```

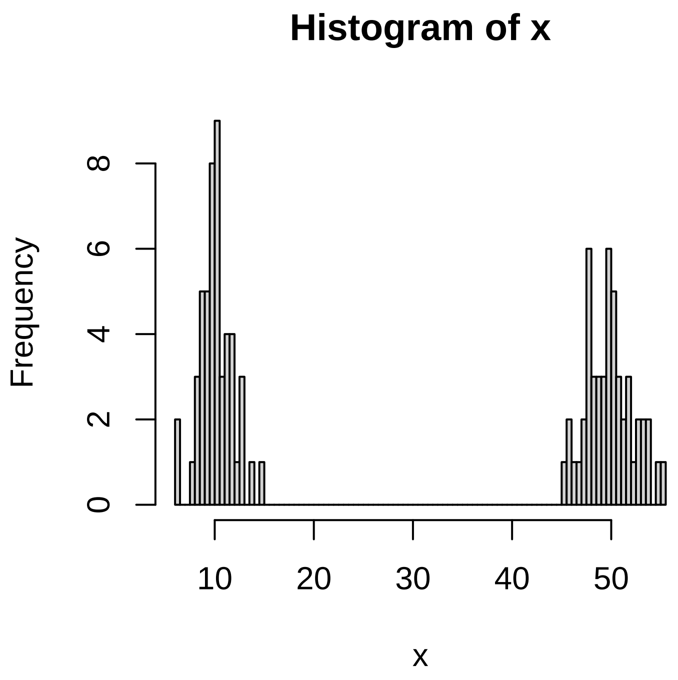


Histograms

```r
set.seed(100) # set random seed to be same every time
x = c(rnorm(50,10,2),
      rnorm(50,50,2))
hist(x)
```



```
# can also change number of bins
hist(x, breaks=100)
```

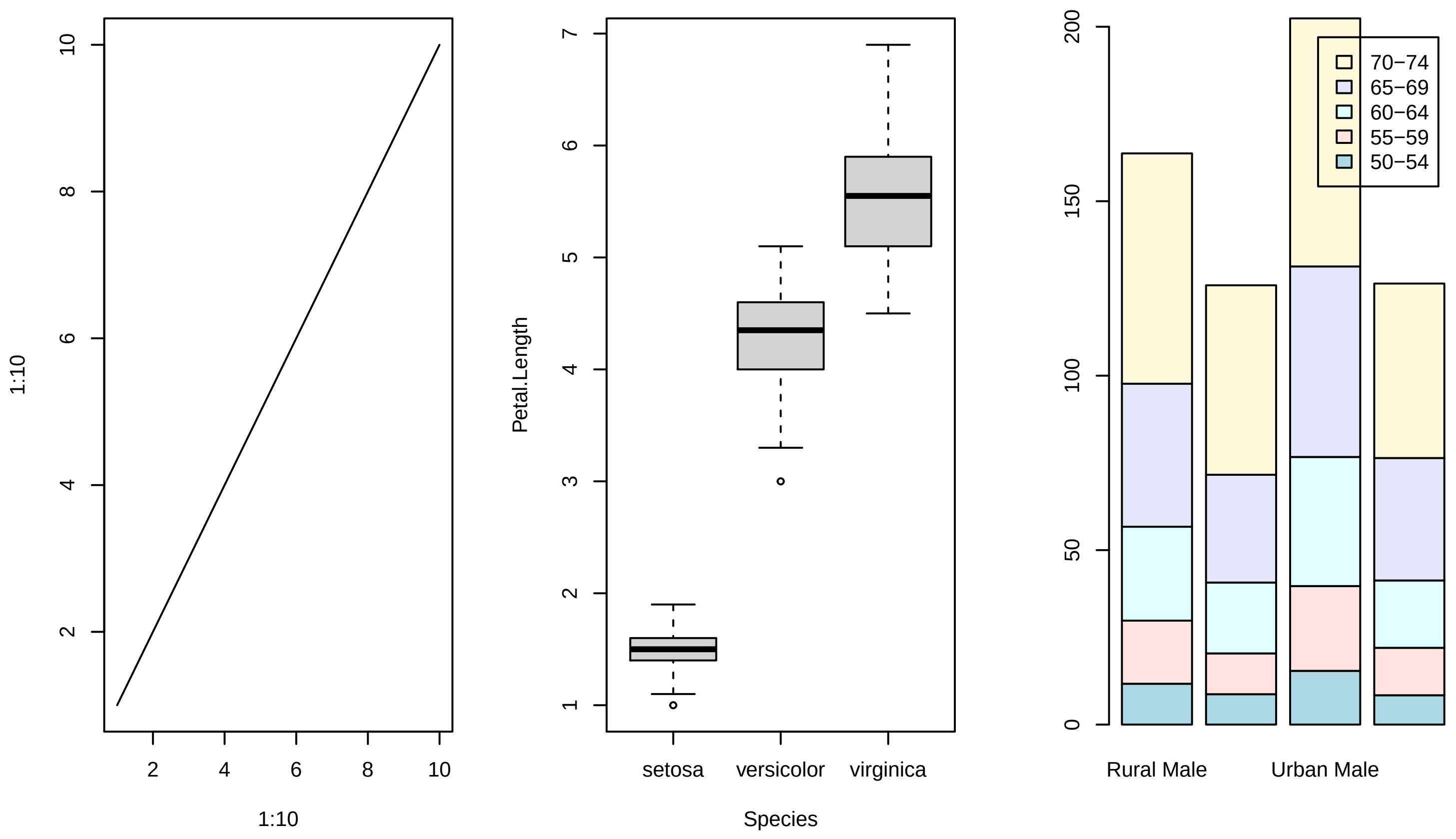


Multipanel plots

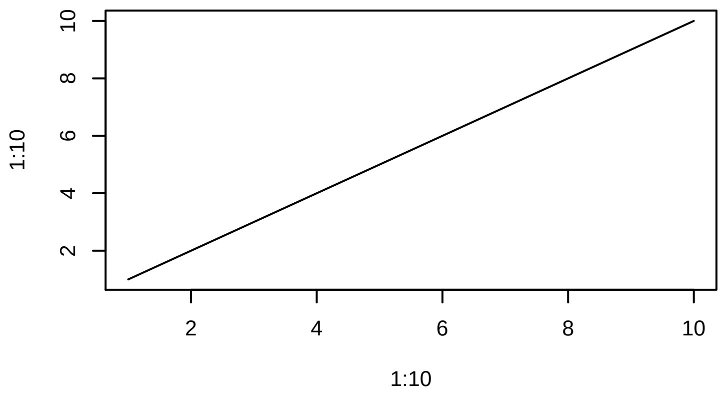
The powerhouse function to control base R plotting parameters is `par`. I chiefly used it to assemble multipanel plots, but it can also adjust margin sizes and many other criteria.

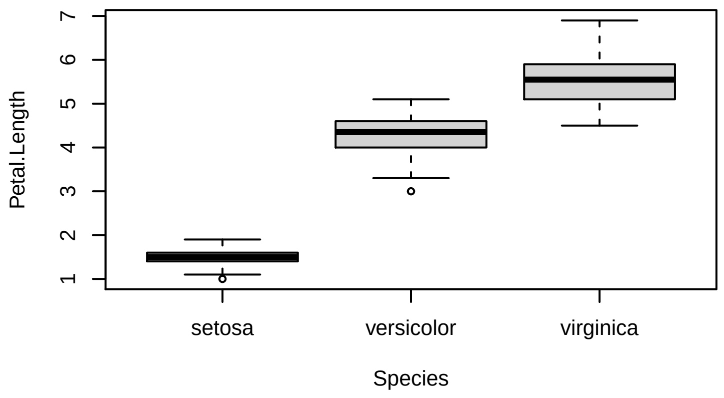
`mfrow` and `mfcol` will adjust the panel layout and plot by rows or columns, respectively.

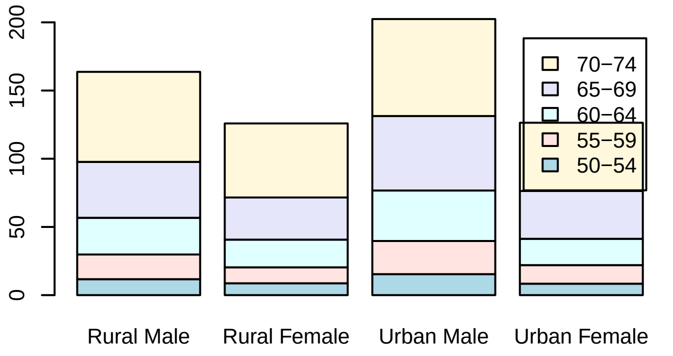
```
par(mfrow=c(1,3))
plot(1:10, 1:10, type="l")
boxplot(Petal.Length ~ Species, data = iris)
barplot(VADeaths,
        col = c("lightblue", "mistyrose", "lightcyan",
                 "lavender", "cornsilk"),
        legend = rownames(VADeaths))
```



```r
par(mfcol=c(3,1))
plot(1:10, 1:10, type="l")
boxplot(Petal.Length ~ Species, data = iris)
barplot(VADeaths,
        col = c("lightblue", "mistyrose", "lightcyan",
                "lavender", "cornsilk"),
        legend = rownames(VADeaths))
```







Other base plots

You can also make a variety of other plots including:

- Pie charts (ew)
- Violin plots / 1D scatterplots by group
- Density plots
- QQ-plots

In addition, you can add information with confidence intervals to your plots. The base R plotting is fairly powerful for getting all the information you want on a plot, but modifying aesthetics becomes very difficult.

ggplot

You will first need to install this using `install.packages("ggplot2")`, then load the package using `library(ggplot2)`. Something to note is that the syntax of making a `ggplot` is different from base R, so you will need to get some practice.

Another **HUGE** benefit of the `ggplot2` package is that it is super popular, which means there are many packages built on top of it to get full customization. Some useful ones include:

1. `ggprism` - Stylize your plots to look like they were made in graphpad prism
2. `ggrepel` - Useful text / label repelling so it doesn't overlap
3. `ggExtra` - Add plots in the margins to show distributions of x and y variables.
4. `gridExtra` - Arrange multiple ggplots together into a single figure.

Structing data for use in ggplot

The big power of using `ggplot` is that it can stylize your plots with minimal code. The catch is that your data needs to be long formatted instead of the more inuitive wide format Here's what I mean. Let's take my lysis data again.

Here is my lysis data in wide format:

Time	A	B	C	D	E	F	G
0	0.223	0.239	0.237	0.231	0.228	0.235	0.231
30	0.523	0.488	0.489	0.505	0.510	0.506	0.504
35	0.620	0.558	0.441	0.505	0.550	0.503	0.510
40	0.685	0.612	0.212	0.354	0.517	0.332	0.379
45	0.772	0.683	0.110	0.222	0.428	0.176	0.244
50	0.844	0.718	0.089	0.137	0.371	0.118	0.174
55	0.888	0.789	0.067	0.111	0.354	0.098	0.136
60	0.968	0.815	0.068	0.105	0.346	0.095	0.126
70	1.390	0.859	0.051	0.091	0.331	0.108	0.111
80	1.544	0.962	0.053	0.068	0.151	0.086	0.074
93	2.010	1.224	0.048	0.070	0.125	0.064	0.084

This is **NOT** suitable for use with `ggplot`. (Actually it can work, but it's more complicated.)

Here is the same lysis data vertically structured (only first 13 rows). In R, you can use the `dplyr` function `gather` to perform this conversion for you. If you load `tidyverse` or `dplyr` in your script you will have access to `gather` and the useful pipe command `%>%`. `%>%` takes the object before it, and uses that as an input to the next command! How nifty.

Here, notice how we also write `-Time`. We want to keep that column static for all the data sets. In other words, cycle through times for all the data values here.

```r
data %>%
  gather(key="Sample", value = "OD", -Time) %>%
  head(13)
```

Time	Sample	OD
0	A	0.223
30	A	0.523
35	A	0.620
40	A	0.685
45	A	0.772
50	A	0.844
55	A	0.888
60	A	0.968
70	A	1.390
80	A	1.544
93	A	2.010
0	B	0.239
30	B	0.488

Notice how every row is a **SINGULAR** observation. It is the OD at one timepoint with the sample. Suppose you had something more complicated like ± DNP addition. You could add a fourth column with that information. You can also name your columns in wide-formatted data to contain the metadata for your experiment. Here's an example:

The data in this table is from a lysis curve measuring optical density over time for strains with 3 different genotypes A, B, and C. In addition, the column names contain metadata for whether or not phage were added (MinusPhage vs PlusPhage) and whether or not a chemical was added.

```r
data = read.delim("lysis_curves/test2.txt")
data[,c(1,2,6,7)] %>% head()
```

Time	A_MinusPhage_MinusChem	B_PlusPhage_MinusChem	C_PlusPhage_MinusChem
0	0.200	0.215	0.168
15	0.305	0.334	0.276
20	0.342	0.352	0.302
25	0.387	0.419	0.332
30	0.440	0.418	0.308
35	0.504	0.452	0.296

We can convert this wide data into long data easily with `gather`, but now we also want to separate the columns further into new columns called "Sample", "Phage_Add", "Chem_Add", to track samples, phage addition, and chemical addition. We can use the aptly named `separate` function from the `dplyr` package in `tidyverse`. `separate` will split a text string at non-alphanumeric characters, but you can also specify a delimiter.

We can also further replace "MinusXX" and "PlusXX" with "-" and "+" now too for easier viewing with the `mutate` command, which adds or changes entire columns.

```r
data_long = data %>%
  gather(key = "Group", value = "OD", -Time) %>%
  separate(Group, remove=F, into=c("Sample", "Phage_Add", "Chem_Add")) %>%
  mutate(
    Phage_Add = ifelse(Phage_Add == "MinusPhage", "-", "+"),
    Chem_Add = ifelse(Chem_Add == "MinusChem", "-", "+")
  )

data_long %>% head()
```

Time	Group	Sample	Phage_Add	Chem_Add	OD
0	A_MinusPhage_MinusChem	A	-	-	0.200
15	A_MinusPhage_MinusChem	A	-	-	0.305
20	A_MinusPhage_MinusChem	A	-	-	0.342
25	A_MinusPhage_MinusChem	A	-	-	0.387
30	A_MinusPhage_MinusChem	A	-	-	0.440
35	A_MinusPhage_MinusChem	A	-	-	0.504

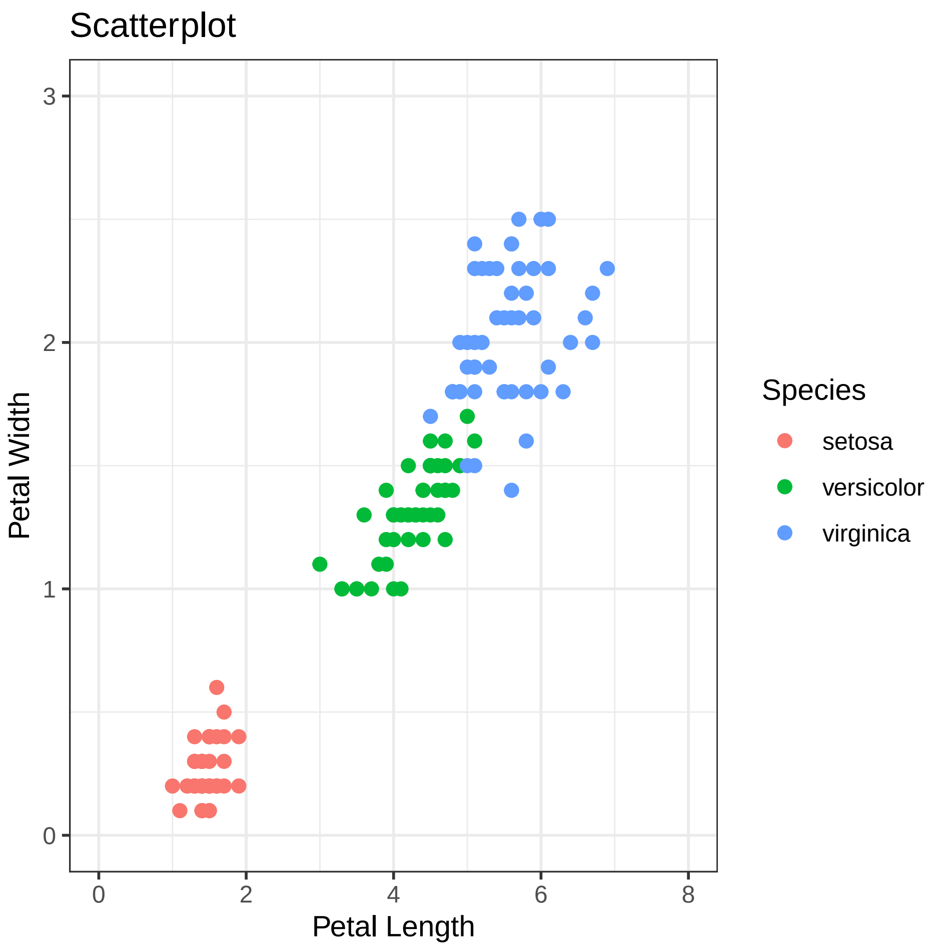
```r
data_long %>% tail()
```

	Time	Group	Sample	Phage_Add	Chem_Add	OD
148	65	C_PlusPhage_PlusChem	C	+	+	0.215
149	70	C_PlusPhage_PlusChem	C	+	+	0.166
150	75	C_PlusPhage_PlusChem	C	+	+	0.163
151	80	C_PlusPhage_PlusChem	C	+	+	0.140
152	85	C_PlusPhage_PlusChem	C	+	+	0.128
153	90	C_PlusPhage_PlusChem	C	+	+	0.123

If you are going to commit to using `ggplot`, you should either stop making wide tables, or (**preferred method**) just write code to switch wide data to long data. As shown above, if you name your columns in a logical and consistent manner, you can use functions like `gather` and `separate` to structure your data in long format easily.

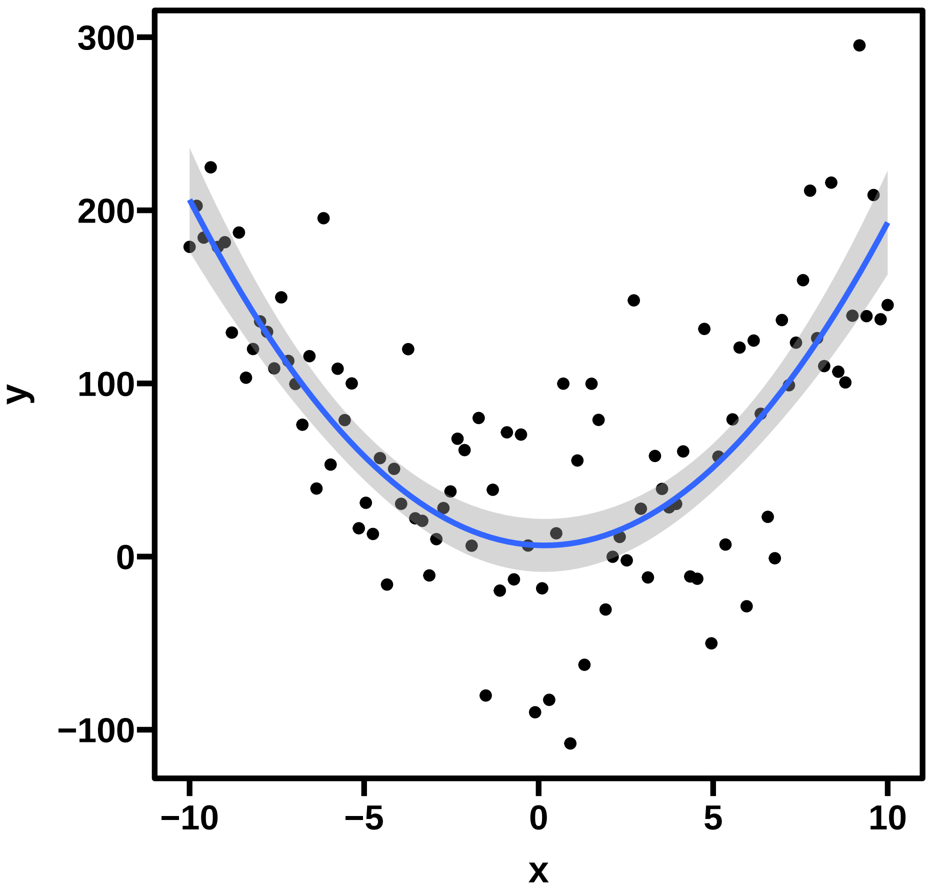
ggplot scatterplots

```
library(ggplot2)
# Note that the variables in the aes() are column names in the iris dataset
ggplot(data = iris, aes(x = Petal.Length, y = Petal.Width)) +
  geom_point(aes(color = Species), size=2) +
  xlim(0,8) +
  ylim(0,3) +
  labs(x="Petal Length",
       y="Petal Width",
       title="Scatterplot",
       color="Species") +
  theme_bw()
```



You can also fit lines to scattersplots and display the confidence intervals of the lines fairly easily using `geom_smooth`. To display your regression equations, you will need more statistics to get the formulas, but you could display those if you wanted.

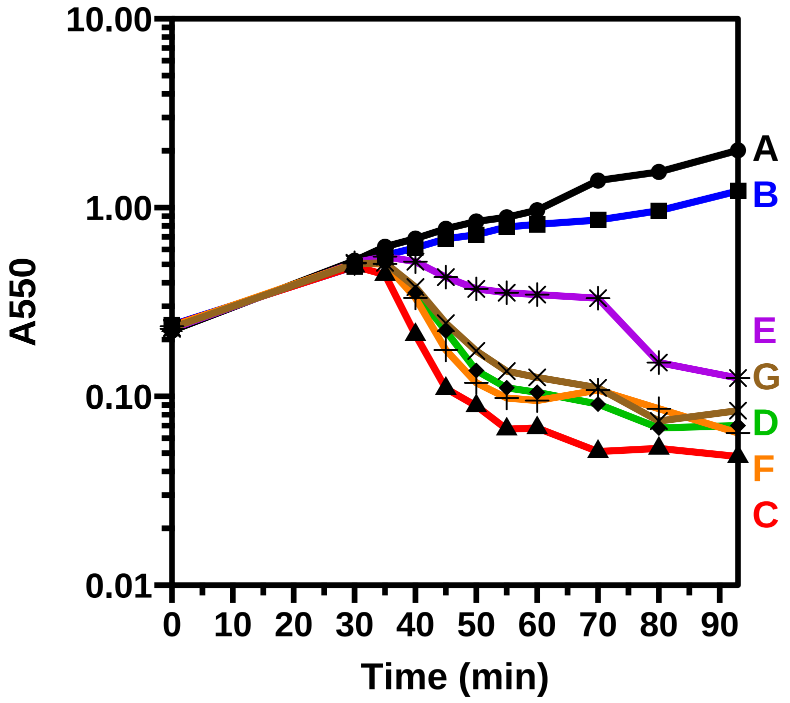
```
library(ggplot2)
library(ggprism)

set.seed(100)
x = seq(-10, 10, length.out = 100)
y = 2 * x^2 + 4 + rnorm(100, 0, sd = 50) # model y = 2x^2 + 4 but with some random noise
df = data.frame(x,y)

ggplot(df, aes(x,y)) +
  geom_point() +
  geom_smooth(formula = y ~ poly(x, 2), method = "lm", se=T) +
  theme_prism(border=T, palette = "colors") + # theme like prism plot
  coord_cartesian(clip = "off") +
  theme(aspect.ratio = 1/1)
```



Lysis curves

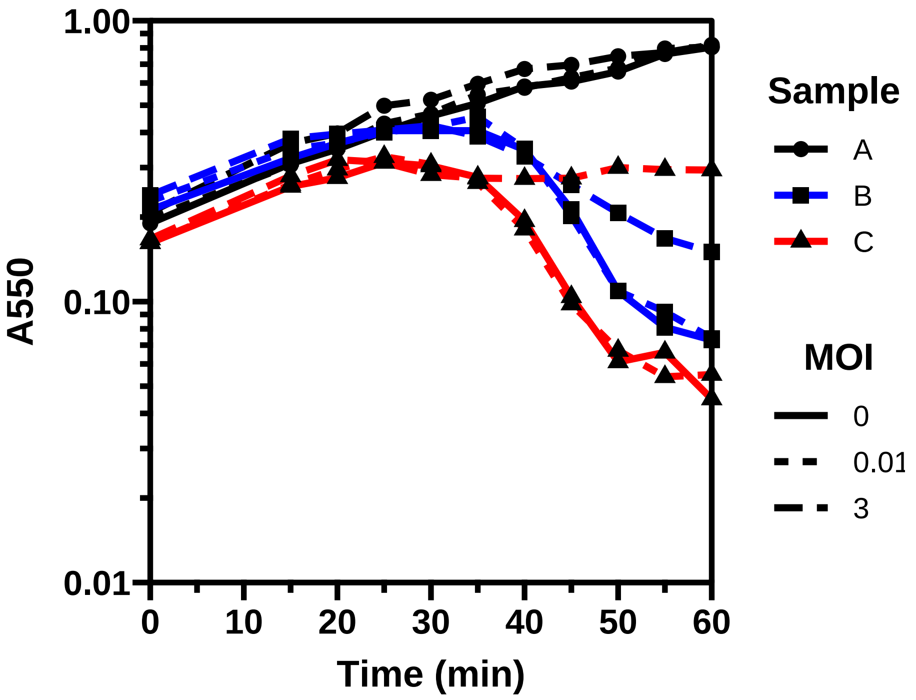
Here is how to plot a beautiful lysis curve.

```r
library(ggplot2)
library(ggprism)
library(ggrepel)

data = read.delim("lysis_curves/test1.txt")
data = data %>%
  gather(key="Sample", value = "OD", -Time)

# define custom offset to move line labels away from axis
offset = max(data$Time)*0.025
# ggprism has default colors to use, but I want to reorder them
cols = ggprism_data$colour_palettes$colors[c(6,1:5,7:20)]

# custom minor ticks on y-axis for log scale
y_minor = rep(1:9, 3)*(10^rep(-2:0, each=9))

ggplot(data = data, aes(x = Time, y = OD)) +
  geom_line(aes(color = Sample), size=1.25) +
  geom_point(aes(shape = Sample), size=2.5, fill="black", na.rm = T) +
  geom_text_repel(data = subset(data, Time == max(data$Time)), # labels next to lines
                  aes(label = Sample,
                      color = Sample,
                      x = Inf, # put label off plot
                      y = OD), # put label at same height as last data point
                  direction = "y",
                  xlim = c(max(data$Time)+offset, Inf), # offset labels
                  min.segment.length = Inf, # won't draw lines
                  hjust=0, # left justify
                  size=5,
                  fontface="bold") + # move it away from the axis
  scale_shape_prism(palette = "default") + # use prism defined shapes
  scale_color_manual(values = cols) + # use my reordered prism colors
  scale_y_log10(limits=c(0.01, 10), # convert y-axis to log10 scale
                guide=guide_prism_minor(),
                minor_breaks=y_minor,
                expand=c(0,0)) +
  scale_x_continuous(breaks=seq(0,max(data$Time),10),
                     guide=guide_prism_minor(),
                     expand=c(0,0)) +
  labs(x="Time (min)",
       y="A550") +
  theme_prism(border=T, palette = "colors") + # theme like prism plot
  coord_cartesian(clip = "off") +
  theme(aspect.ratio = 1/1,
        legend.position = "none",
        plot.margin = unit(c(1,5,1,1), "lines"))
```



Here's a more complicated lysis curve where the data is from a lysis curve with 3 samples, A, B, and C, testing phage addition at 3 different MOIs. (Think of that like phage concentration added)

```
library(ggplot2)
library(ggprism)

data = read.delim("lysis_curves/test3.txt")

# ggprism has default colors to use, but I want to reorder them
cols = ggprism_data$colour_palettes$colors[c(6,1:5,7:20)]

# custom minor ticks on y-axis for log scale
y_minor = c(rep(1:9, 2)*(10^rep(c(-2, -1), each=9)), 1)

ggplot(data = data, aes(x = Time, y = OD)) +
  geom_line(aes(color = Sample, linetype = as.factor(MOI)), size=1.25) +
  geom_point(aes(shape = Sample),
                 size=2.5, fill="black", na.rm = T) +
  scale_shape_prism(palette = "default") + # use prism defined shapes
  scale_color_manual(values = cols) + # use my reordered prism colors
  scale_y_log10(limits=c(0.01, 1), # convert y-axis to log10 scale
                guide=guide_prism_minor(),
                minor_breaks=y_minor,
                expand=c(0,0)) +
  scale_x_continuous(breaks=seq(0,max(data$Time),10),
                     guide=guide_prism_minor(),
                     expand=c(0,0)) +
  labs(x="Time (min)",
       y="A550",
       color="Sample",
       shape="Sample",
       linetype="MOI") + # rename MOI legend title
  theme_prism(border=T, palette = "colors") + # theme like prism plot
  coord_cartesian(clip = "off") +
  theme(aspect.ratio = 1/1,
        legend.title = element_text())
```

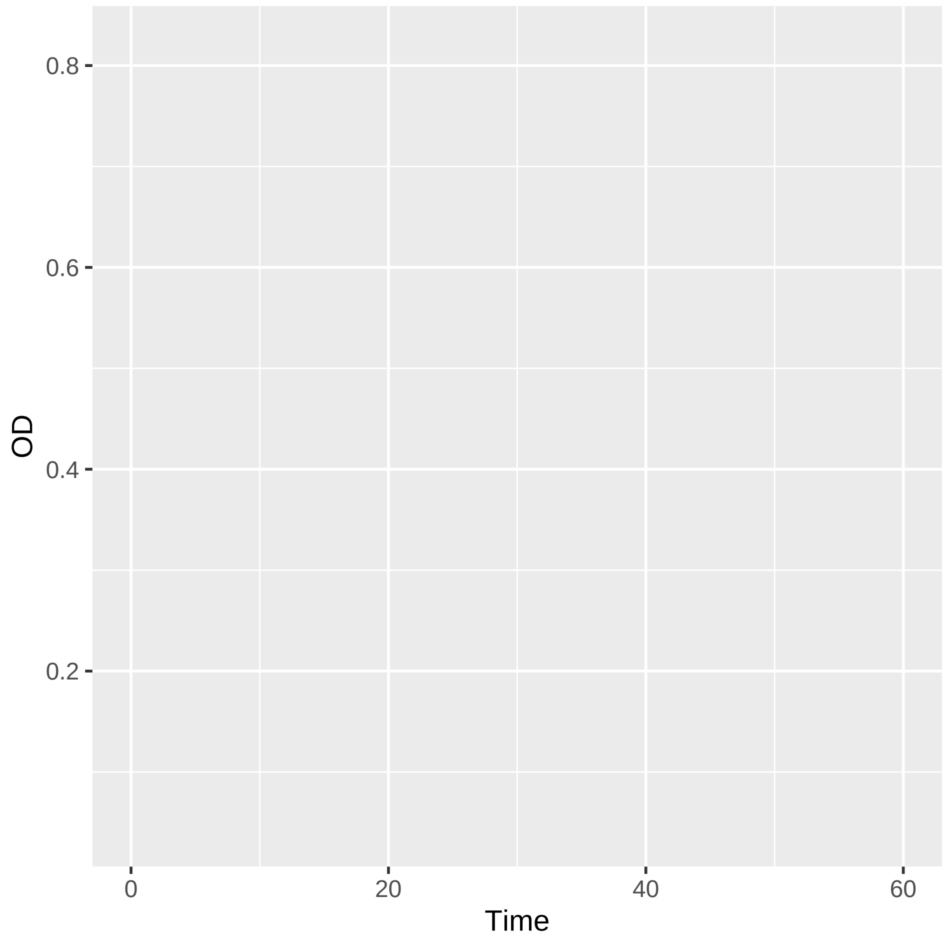


Let's break down all these commands. `ggplot()` just draws a plot background and the frame. Setting the `data = data` indicates that the data being used is the data frame named `data` I made containing my lysis curve data.

The aesthetics of the plot are passed to the `aes()` command. `aes` takes arguments for:

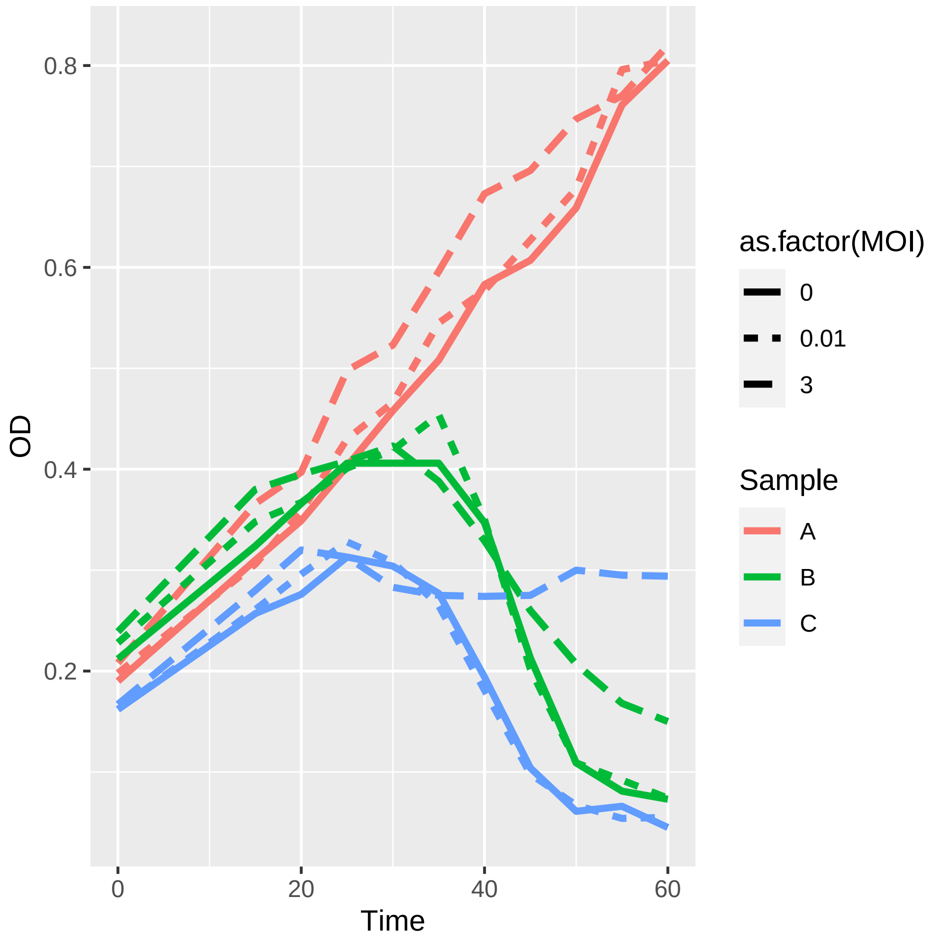
- x
- y
- color
- fill
- shape
- linetype So that you can customize you plot appearance by different groups of data.

```
library(ggplot2)
ggplot(data = data, aes(x = Time, y = OD))
```



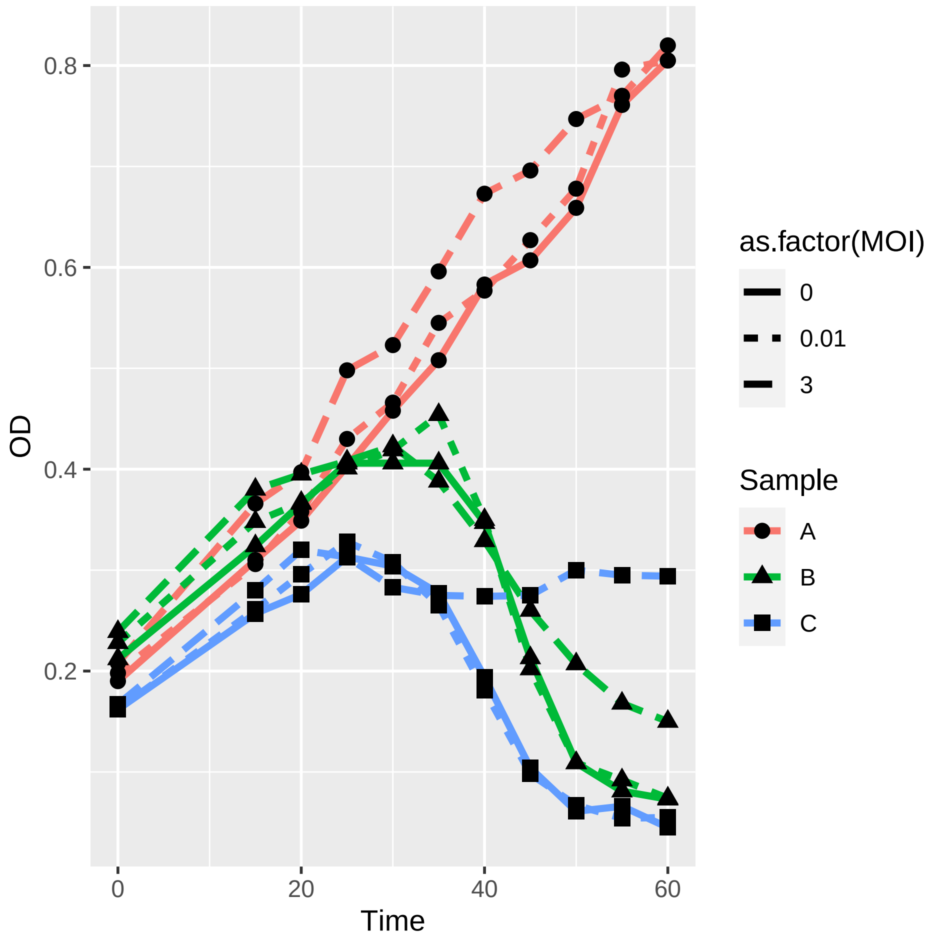
`geom_line()` will add lines to plot `Time` against `OD`. Notice that I added `aes(color = ...)`. This allows us to color the lines by the samples. Without that argument, only one giant mess of lines would be plotted.

```r
library(ggplot2)
ggplot(data = data, aes(x = Time, y = OD)) +
  geom_line(aes(color = Sample, linetype = as.factor(MOI)), size=1.25)
```



`geom_point()` will also add bullet points at each time point. Notice that we can adjust the shape to match the sample type as well.

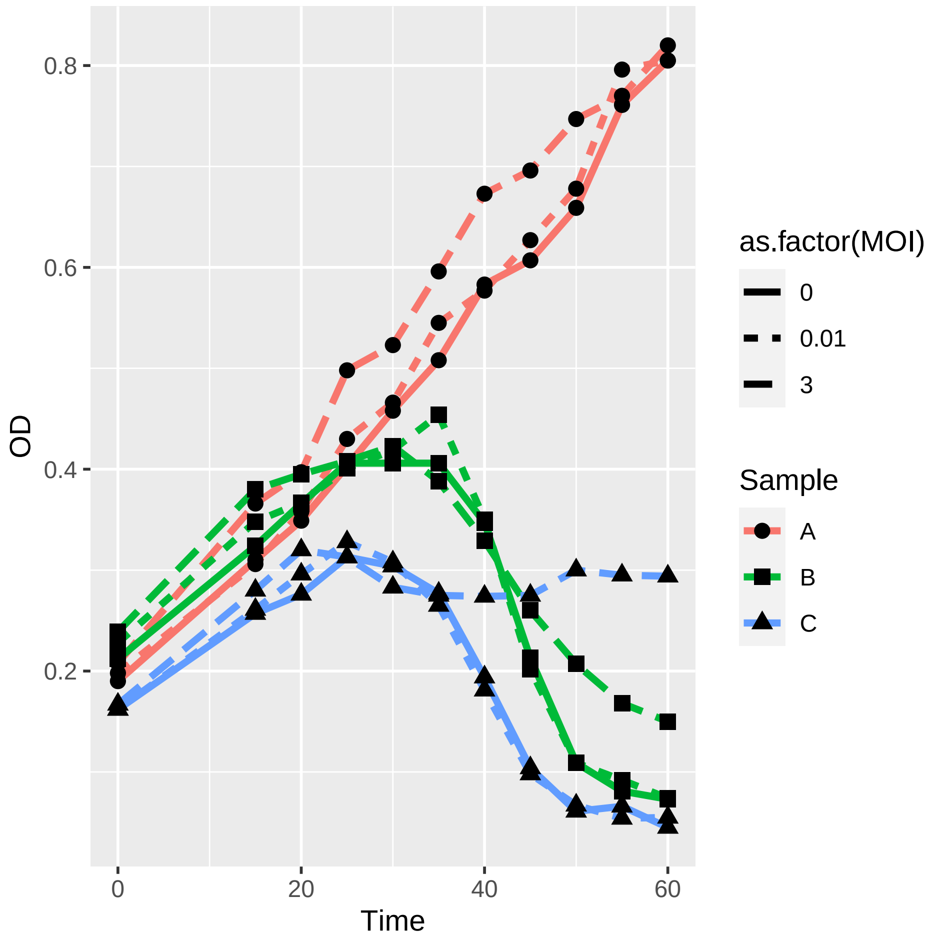
```
library(ggplot2)
ggplot(data = data, aes(x = Time, y = OD)) +
  geom_line(aes(color = Sample, linetype = as.factor(MOI)), size=1.25) +
  geom_point(aes(shape = Sample), size=2.5, fill="black", na.rm = T)
```



There are also many other `geom` arguments you can add depending on your plot type. I will show examples of these later. Notice they take other arguments to adjust sizes and constant color changes.

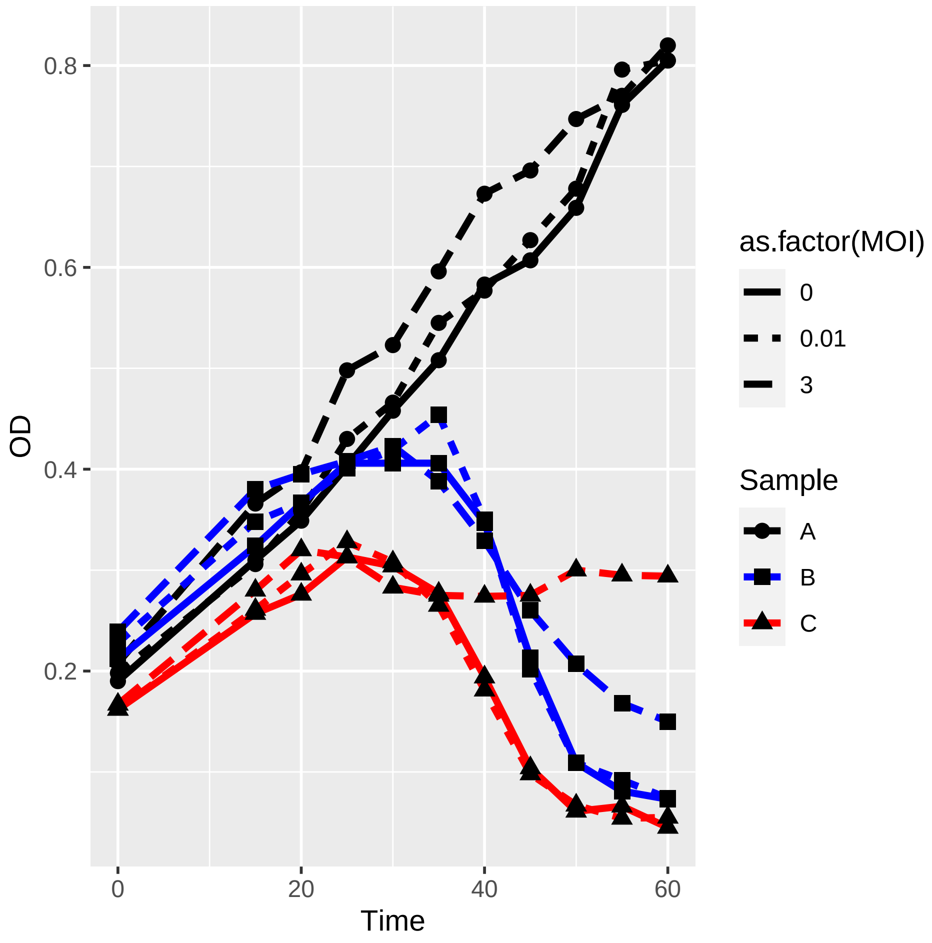
Now let's tell `ggplot` how to work with our `aes` settings by changing `scales`. First, we can change the shapes we use for our bullets to be defined by the `ggprism` package.

```r
library(ggplot2)
library(ggprism)
ggplot(data = data, aes(x = Time, y = OD)) +
  geom_line(aes(color = Sample, linetype = as.factor(MOI)), size=1.25) +
  geom_point(aes(shape = Sample), size=2.5, fill="black", na.rm = T) +
  scale_shape_prism(palette = "default")
```



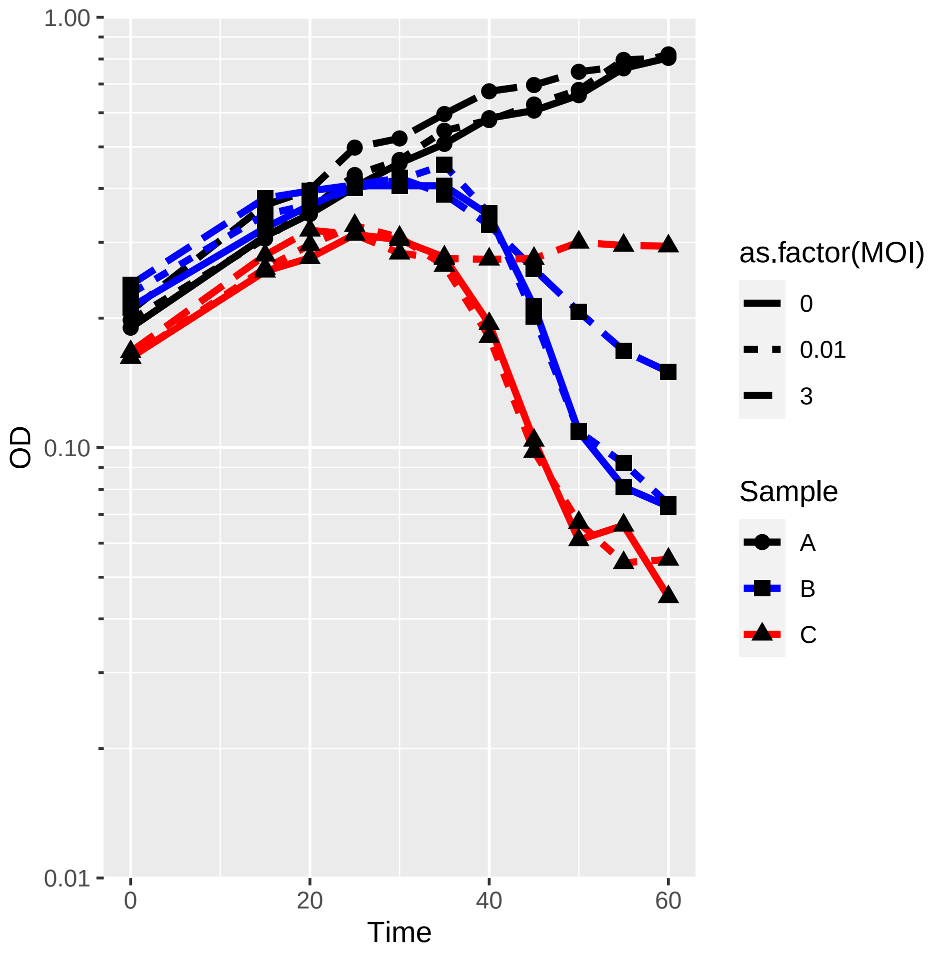
Next, we can change how `ggplot` will color anything passed to a `color` argument. I custom defined a vector of colors called `cols` based on `ggprism` colors.

```r
library(ggplot2)
library(ggprism)
ggplot(data = data, aes(x = Time, y = OD)) +
  geom_line(aes(color = Sample, linetype = as.factor(MOI)), size=1.25) +
  geom_point(aes(shape = Sample), size=2.5, fill="black", na.rm = T) +
  scale_shape_prism(palette = "default") +
  scale_color_manual(values = cols)
```



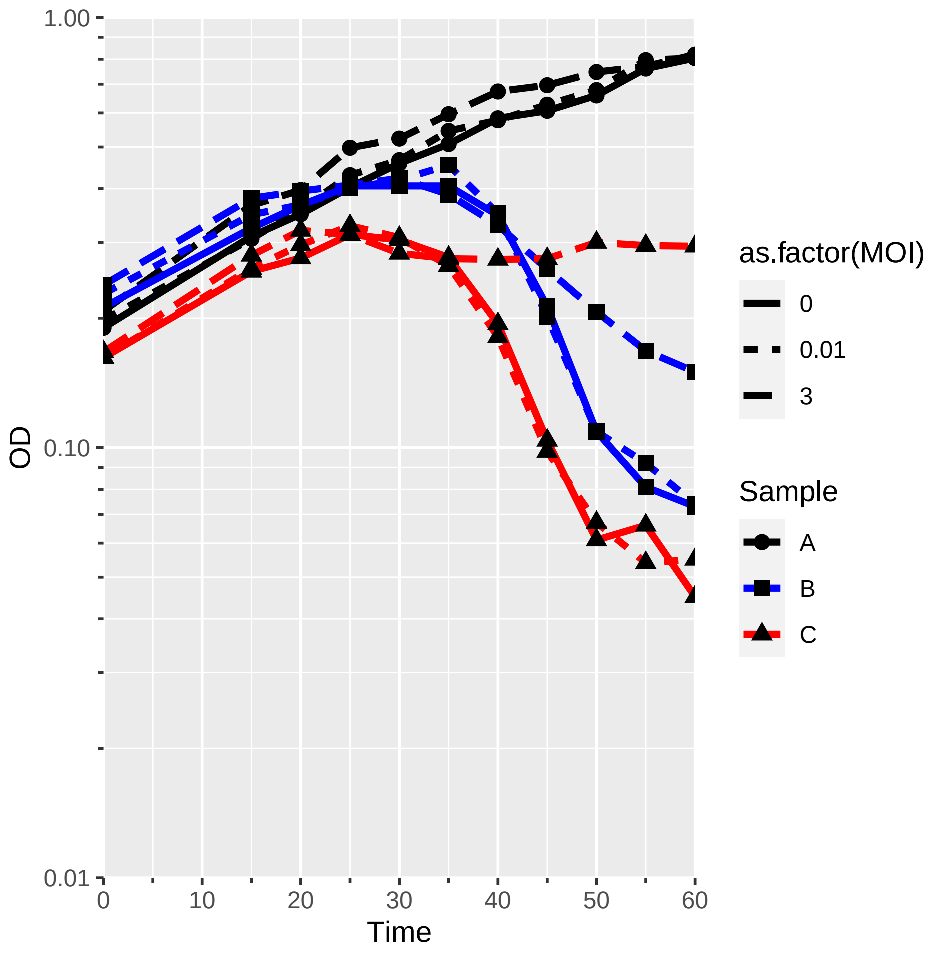
Then we can also change how the axes are structured. First, I want to make the y-axis on log10 scale. I can supply the end points with the `limits` and change the minor ticks using the `guides` and `minor_breaks` arguments. Finally, `ggplot` automatically leaves 5% of space on each end of each axes. Aesthetically, I do not like this usually, so I use the `expand` command to reduce that 0%.

```r
library(ggplot2)
library(ggprism)
ggplot(data = data, aes(x = Time, y = OD)) +
  geom_line(aes(color = Sample, linetype = as.factor(MOI)), size=1.25) +
  geom_point(aes(shape = Sample), size=2.5, fill="black", na.rm = T)+
  scale_shape_prism(palette = "default") +
  scale_color_manual(values = cols) +
  scale_y_log10(limits=c(0.01, 1),
                guide=guide_prism_minor(),
                minor_breaks=y_minor,
                expand=c(0,0))
```



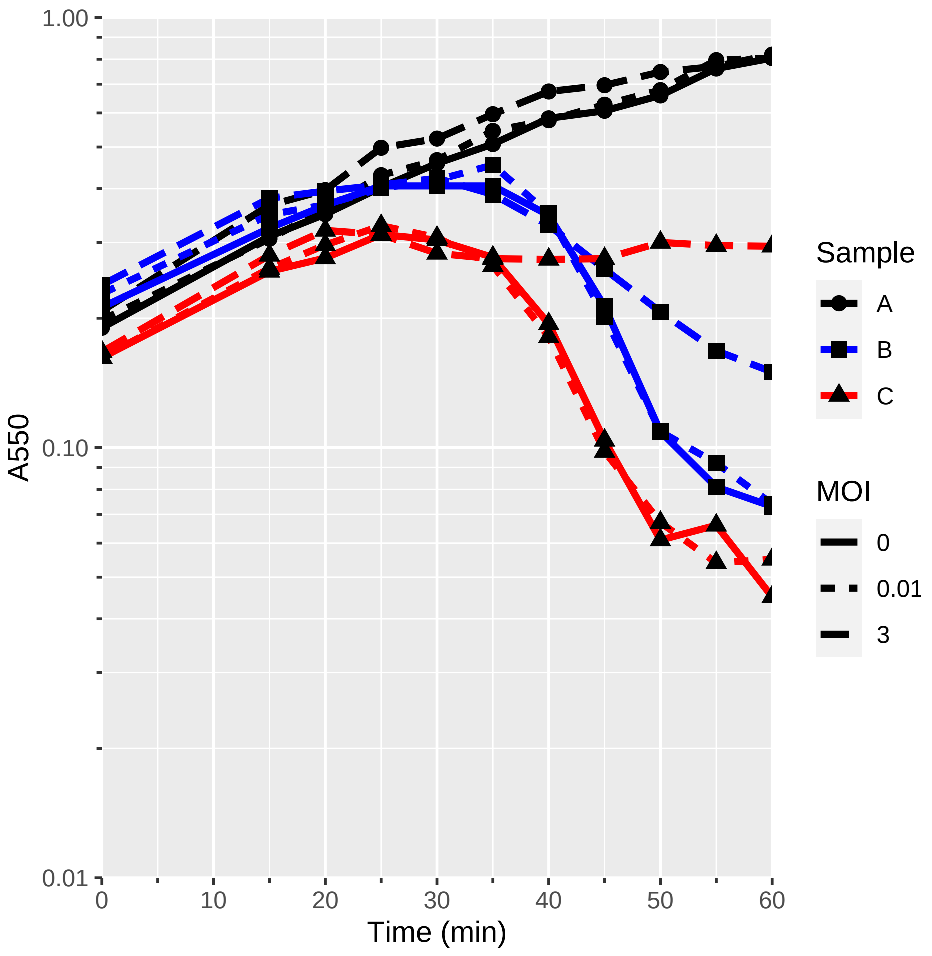
We can also change the scaling of the x axis too. I like to only have x-axis go from the start to end time without any extra space, but this currently looks very strange. It will need more commands to continue looking better.

```r
library(ggplot2)
library(ggprism)
ggplot(data = data, aes(x = Time, y = OD)) +
  geom_line(aes(color = Sample, linetype = as.factor(MOI)), size=1.25) +
  geom_point(aes(shape = Sample), size=2.5, fill="black", na.rm = T) +
  scale_shape_prism(palette = "default") +
  scale_color_manual(values = cols) +
  scale_y_log10(limits=c(0.01, 1),
                guide=guide_prism_minor(),
                minor_breaks=y_minor,
                expand=c(0,0)) +
  scale_x_continuous(breaks=seq(0,max(data$Time),10),
                     guide=guide_prism_minor(),
                     expand=c(0,0))
```



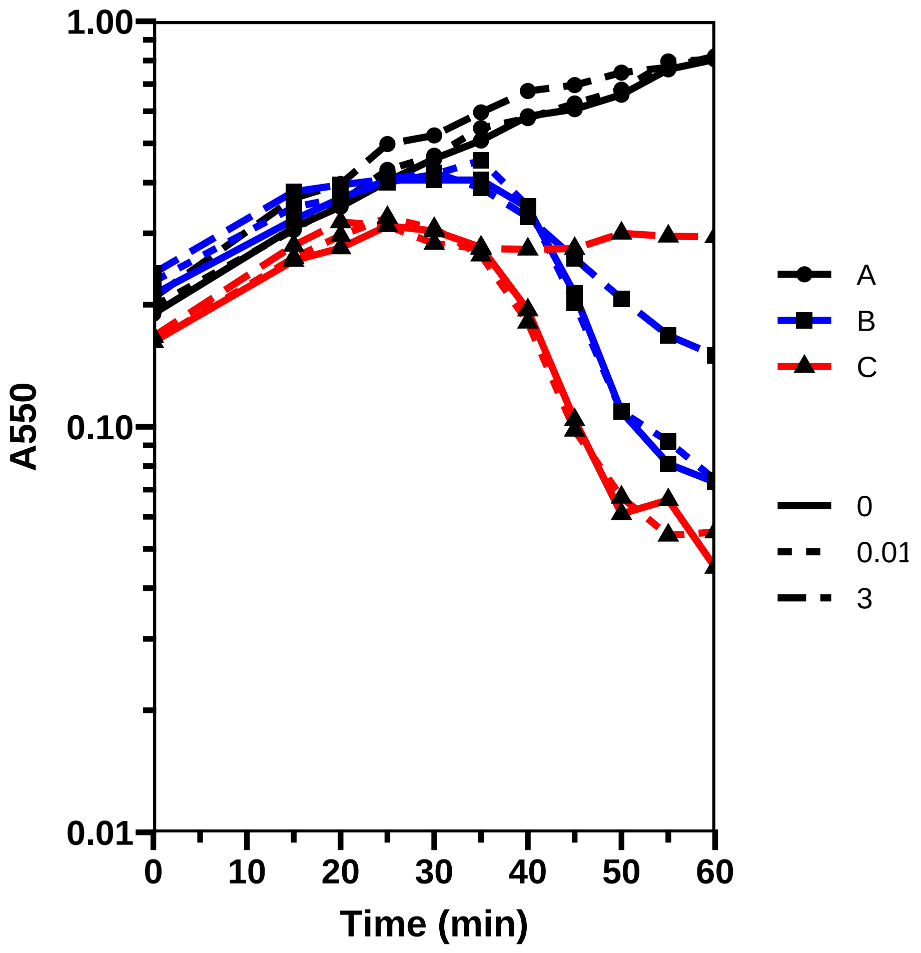
We can add labels to the axes, plot titles, and any legend titles. Note: legend titles are the same variables as variables passed to the `aes` function.

```r
library(ggplot2)
library(ggprism)
ggplot(data = data, aes(x = Time, y = OD)) +
  geom_line(aes(color = Sample, linetype = as.factor(MOI)), size=1.25) +
  geom_point(aes(shape = Sample), size=2.5, fill="black", na.rm = T) +
  scale_shape_prism(palette = "default") +
  scale_color_manual(values = cols) +
  scale_y_log10(limits=c(0.01, 1),
                guide=guide_prism_minor(),
                minor_breaks=y_minor,
                expand=c(0,0)) +
  scale_x_continuous(breaks=seq(0,max(data$Time),10),
                     guide=guide_prism_minor(),
                     expand=c(0,0)) +
  labs(x="Time (min)",
       y="A550",
       color="Sample",
       shape="Sample",
       linetype="MOI")
```



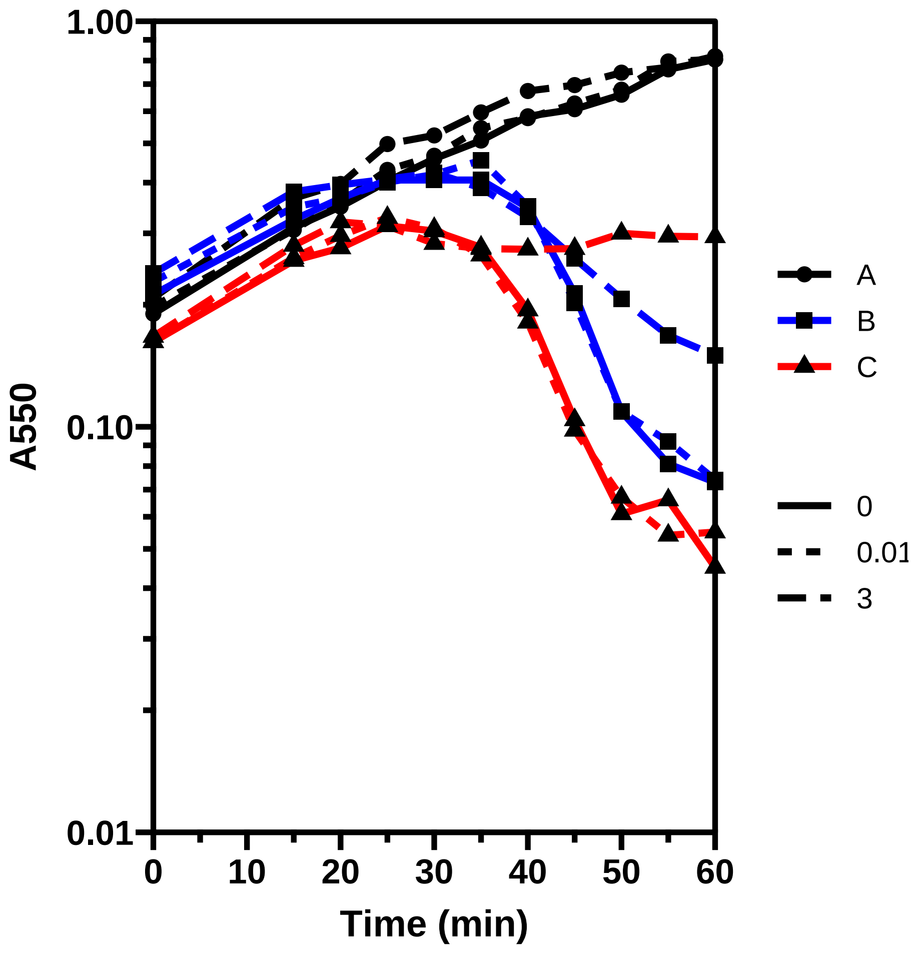
ggplot has many themes to change the way the entire plot looks, especially the plot backgrounds. I like to use theme_prism to stylize my plots like graphpad prism. Additionally, I like to have borders around the entire graph and specifiy the prism colors to use.

```
library(ggplot2)
library(ggprism)
ggplot(data = data, aes(x = Time, y = OD)) +
  geom_line(aes(color = Sample, linetype = as.factor(MOI)), size=1.25) +
  geom_point(aes(shape = Sample), size=2.5, fill="black", na.rm = T) +
  scale_shape_prism(palette = "default") +
  scale_color_manual(values = cols) +
  scale_y_log10(limits=c(0.01, 1),
               guide=guide_prism_minor(),
               minor_breaks=y_minor,
               expand=c(0,0)) +
  scale_x_continuous(breaks=seq(0,max(data$Time),10),
                     guide=guide_prism_minor(),
                     expand=c(0,0)) +
  labs(x="Time (min)",
       y="A550",
       color="Sample",
       shape="Sample",
       linetype="MOI") +
  theme_prism(border=T, palette = "colors")
```



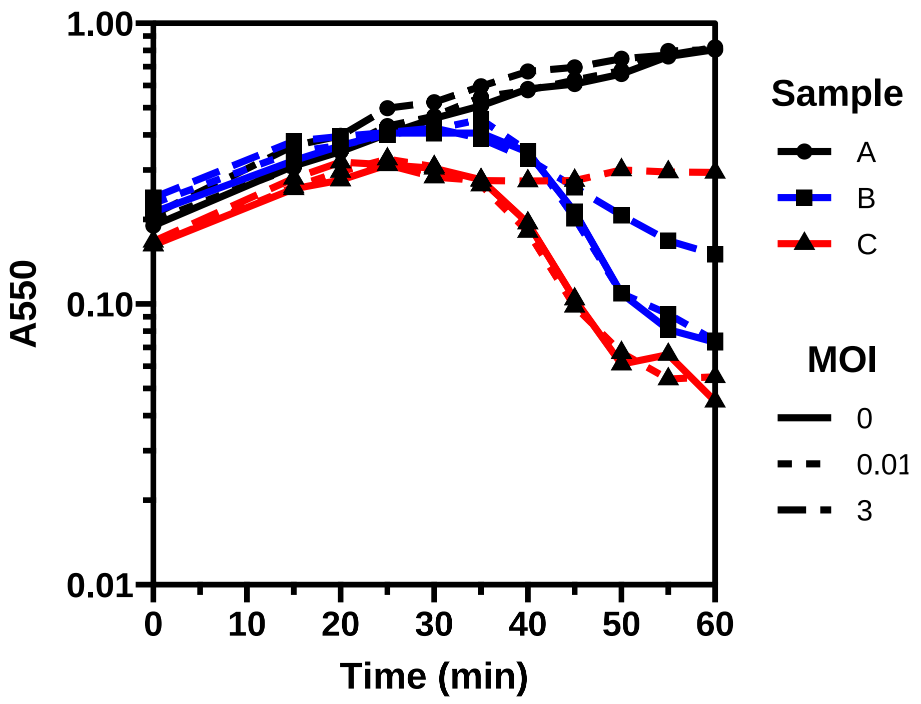
Notice how the points at the min and max values on the x-axis are hidden underneath the plot itself. We can fix this by changing `clip = "off"` in the `coord_cartesian()` function.

```
library(ggplot2)
library(ggprism)
ggplot(data = data, aes(x = Time, y = OD)) +
  geom_line(aes(color = Sample, linetype = as.factor(MOI)), size=1.25) +
  geom_point(aes(shape = Sample), size=2.5, fill="black", na.rm = T) +
  scale_shape_prism(palette = "default") +
  scale_color_manual(values = cols) +
  scale_y_log10(limits=c(0.01, 1),
                guide=guide_prism_minor(),
                minor_breaks=y_minor,
                expand=c(0,0)) +
  scale_x_continuous(breaks=seq(0,max(data$Time),10),
                     guide=guide_prism_minor(),
                     expand=c(0,0)) +
  labs(x="Time (min)",
       y="A550",
       color="Sample",
       shape="Sample",
       linetype="MOI") +
  theme_prism(border=T, palette = "colors") +
  coord_cartesian(clip = "off")
```



For the final aesthetic changes, I like to have square plots, so I change the `aspect.ratio` in the `theme()` function. Additionally, `theme_prism()` automatically hides all legend titles, so we need to unhide those using `legend.title = element_text()` to specify that the title needs to be a text element instead of a blank element (`element_blank()`).

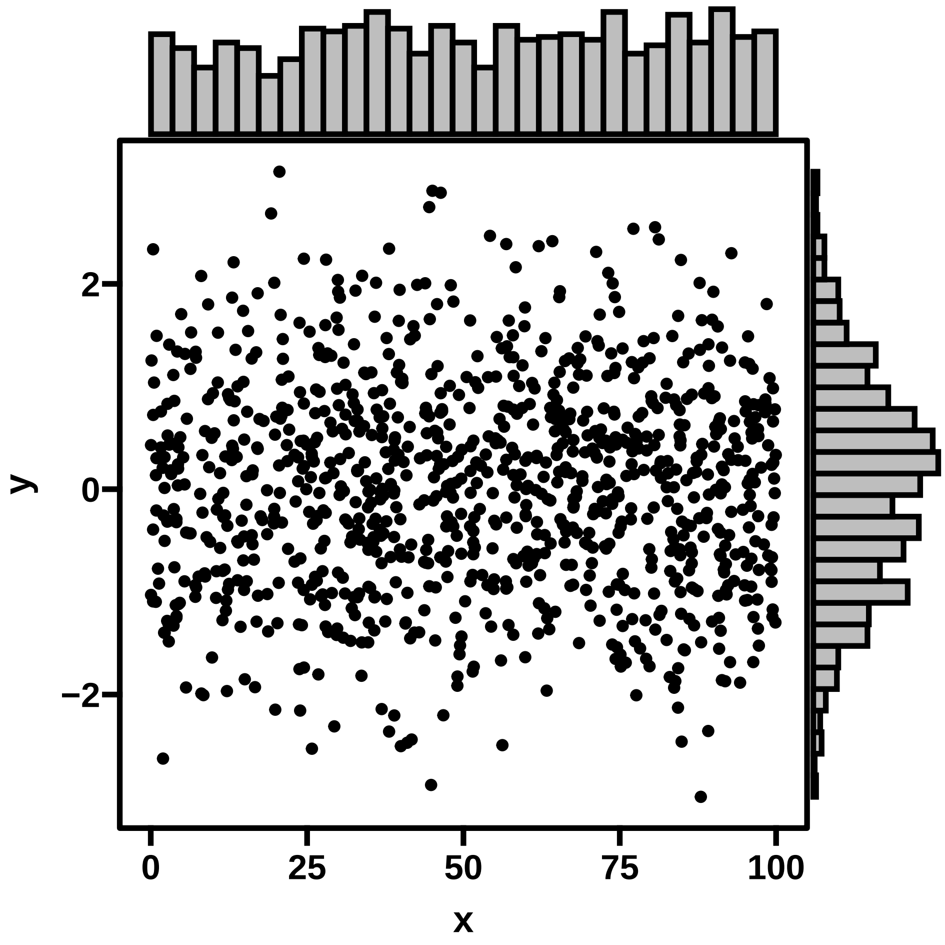
```
library(ggplot2)
library(ggprism)
ggplot(data = data, aes(x = Time, y = OD)) +
  geom_line(aes(color = Sample, linetype = as.factor(MOI)), size=1.25) +
  geom_point(aes(shape = Sample), size=2.5, fill="black", na.rm = T) +
  scale_shape_prism(palette = "default") +
  scale_color_manual(values = cols) +
  scale_y_log10(limits=c(0.01, 1),
                guide=guide_prism_minor(),
                minor_breaks=y_minor,
                expand=c(0,0)) +
  scale_x_continuous(breaks=seq(0,max(data$Time),10),
                     guide=guide_prism_minor(),
                     expand=c(0,0)) +
  labs(x="Time (min)",
       y="A550",
       color="Sample",
       shape="Sample",
       linetype="MOI") +
  theme_prism(border=T, palette = "colors") +
  coord_cartesian(clip = "off") +
  theme(aspect.ratio = 1/1,
        legend.title = element_text())
```



ggplot marginal histograms

Suppose you have a scatterplot and you want to show the distribution of each variable in addition. Use the `ggExtra` package.

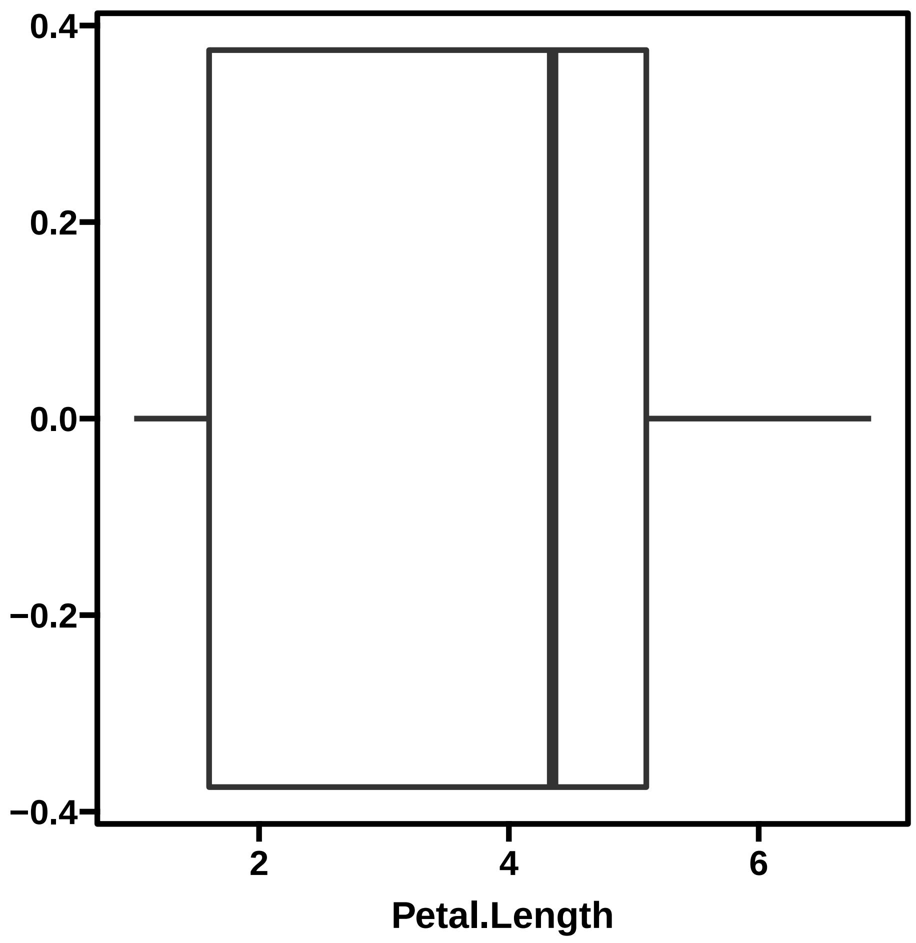
```r
library(ggplot2)
library(ggprism)
library(ggExtra)

set.seed(100) # set a random seed
# x is generated from a uniform distribution from 0 to 100
x = runif(1000, 0, 100)
# y is generated from a normal distribution with mean 0 and sd 1
y = rnorm(1000)
df = data.frame(x,y)

g = ggplot(df, aes(x, y)) +
      geom_point() +
      theme_prism(border=T) +
      coord_cartesian(clip = "off") +
      theme(aspect.ratio = 1/1)
ggMarginal(g, type = "histogram", fill = "gray", lwd=1)
```



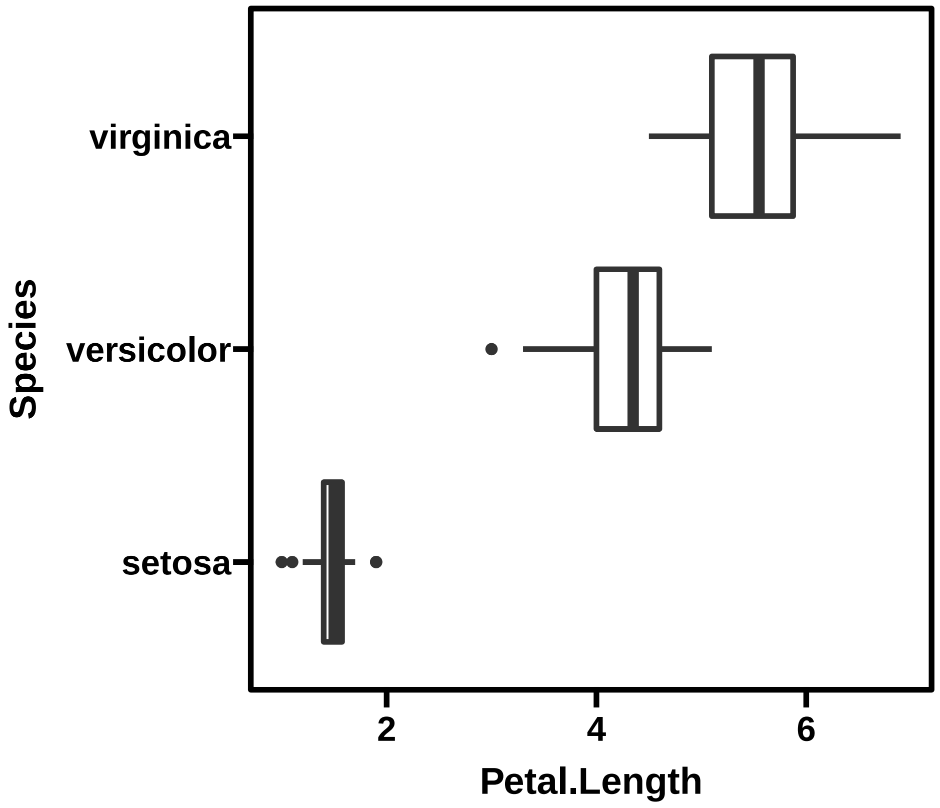
ggplot boxplots

You can easily rotate these by changing `Petal.Length` to be on the y-axis.

```r
library(ggplot2)
library(ggprism)

# simple box plot
ggplot(iris, aes(x = Petal.Length)) +
  geom_boxplot(size=1) +
  theme_prism(border=T) +
  coord_cartesian(clip = "off") +
  theme(aspect.ratio = 1/1)
```



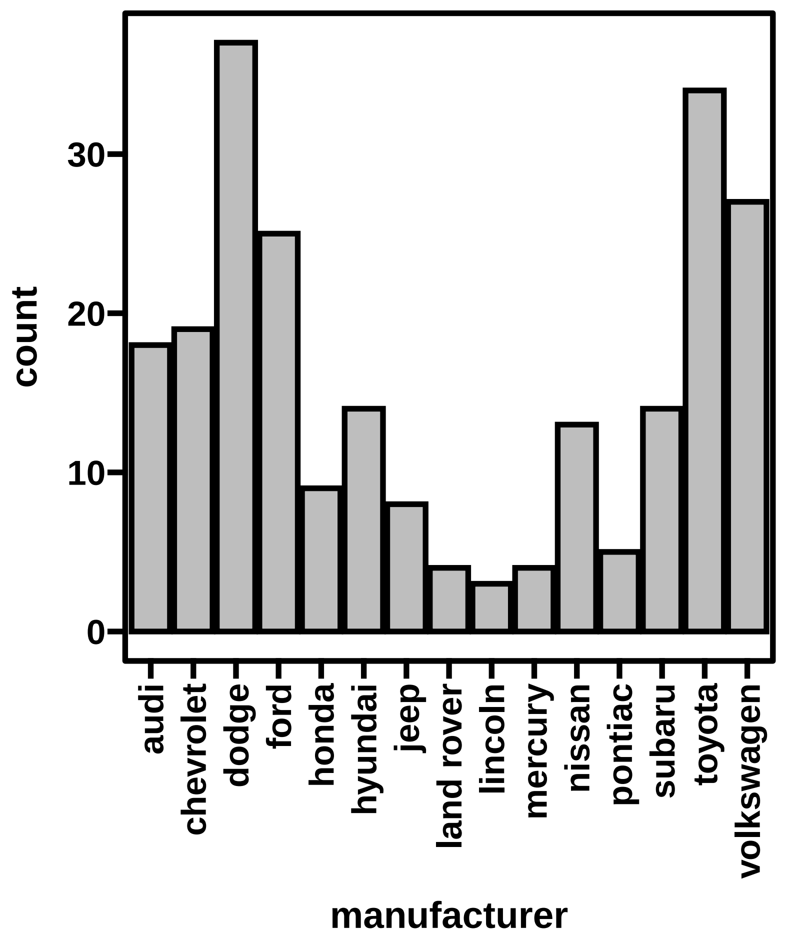
```r
# box plot faceted by another variable
ggplot(iris, aes(x = Petal.Length, y = Species)) +
  geom_boxplot(size=1) +
  theme_prism(border=T) +
  coord_cartesian(clip = "off") +
  theme(aspect.ratio = 1/1)
```



ggplot barplots

simple ggplot barplots

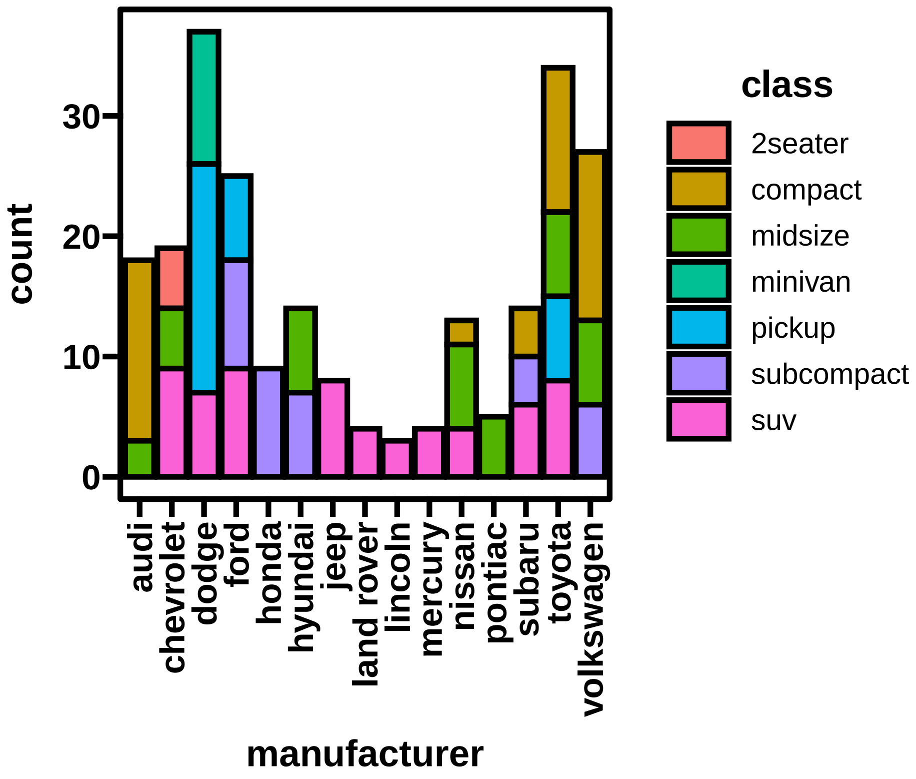
```
library(ggplot2)
library(ggprism)
data(mpg, package = "ggplot2")

ggplot(mpg, aes(manufacturer)) +
  geom_bar(size=1, col="black", fill="gray") +
  theme_prism(border=T) +
  coord_cartesian(clip = "off") +
  theme(aspect.ratio = 1/1,
        axis.text.x = element_text(angle = 90, hjust = 1)) # needed to rotate x labels
```



stacked ggplot barplots

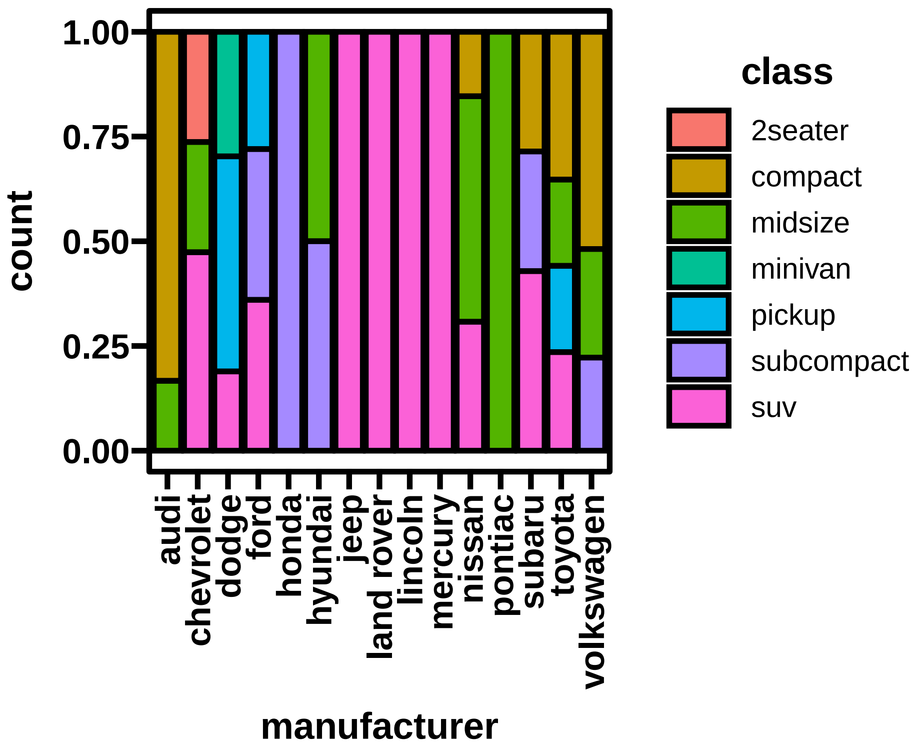
We can do both stacked with the raw counts

```
library(ggplot2)
library(ggprism)
data(mpg, package = "ggplot2")

ggplot(mpg, aes(manufacturer)) +
  geom_bar(aes(fill=class), position = "stack", # these two arguments are required
           size=1, col="black") +
  theme_prism(border=T) +
  coord_cartesian(clip = "off") +
  theme(aspect.ratio = 1/1, legend.title = element_text(),
        axis.text.x = element_text(angle = 90, hjust = 1)) # needed to rotate x labels
```



and percent stacked bar charts very easily with `ggplot`. You don't have to format your data to already contain the percent values either, which you do with base R. Here, `ggplot` will calculate the values for you.

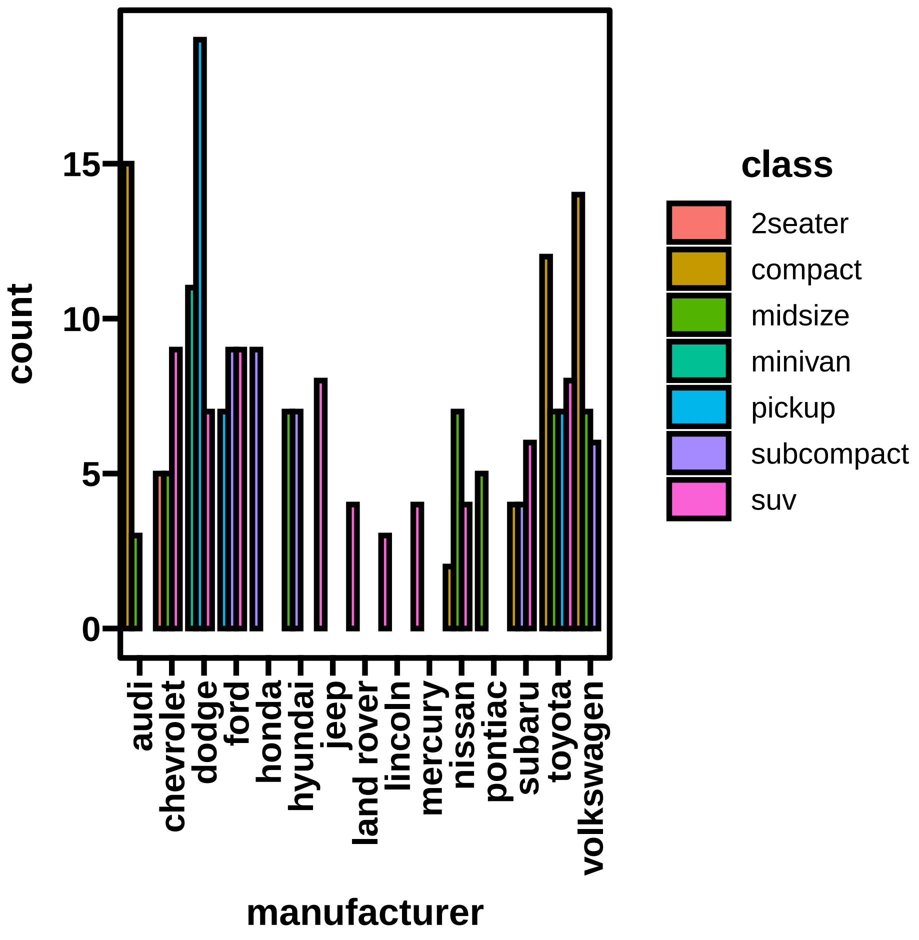
```r
library(ggplot2)
library(ggprism)
data(mpg, package = "ggplot2")

ggplot(mpg, aes(manufacturer)) +
  geom_bar(aes(fill=class), position = "fill", # these two arguments are required
           size=1, col="black") +
  theme_prism(border=T) +
  coord_cartesian(clip = "off") +
  theme(aspect.ratio = 1/1, legend.title = element_text(),
        axis.text.x = element_text(angle = 90, hjust = 1)) # needed to rotate x labels
```



grouped ggplot barplots

Normally you can just set `position = "dodge"`, but if you have groups that don't have all the sub groups, then all the bars will not be constant width. For example, Jeeps only have SUVs in this dataset, so the bar there will be larger. I do not find that aesthetically pleasing, so I use `position = position_dodge(preserve = "single")`.

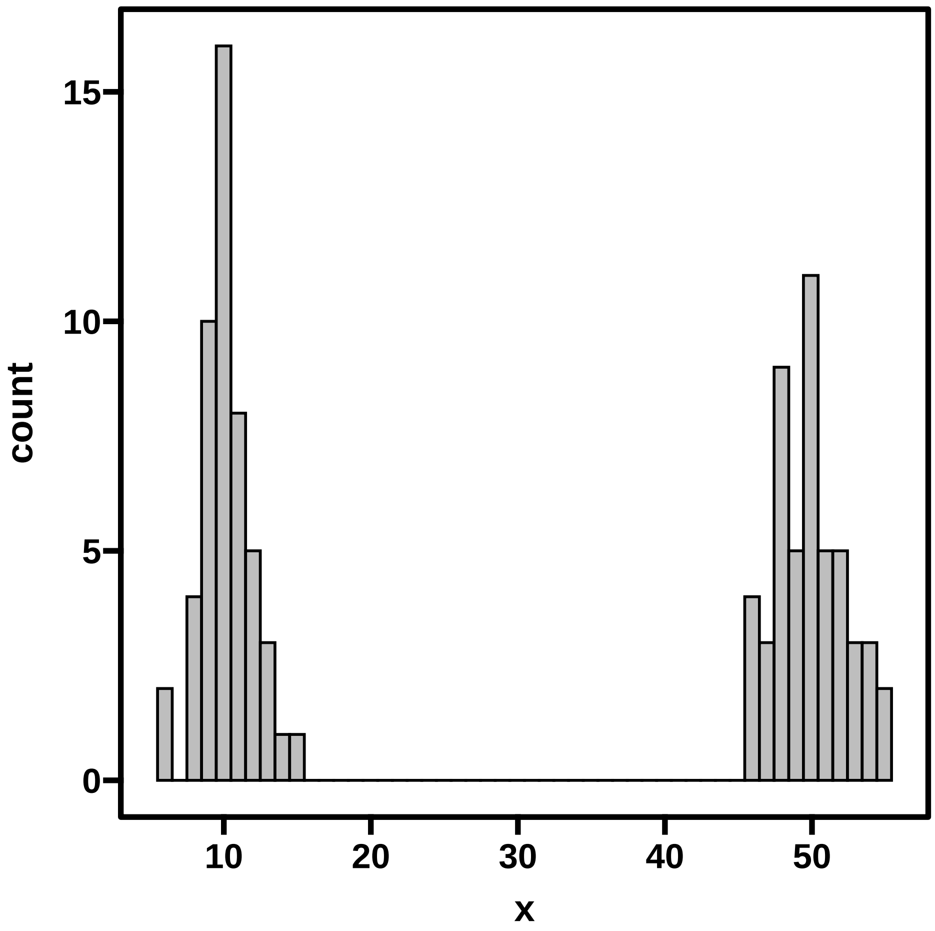
```r
library(ggplot2)
library(ggprism)

data(mpg, package = "ggplot2")

ggplot(mpg, aes(manufacturer)) +
  geom_bar(aes(fill=class), position = position_dodge(preserve = "single"),
           size=1, col="black", width = 1) +
  theme_prism(border=T) +
  coord_cartesian(clip = "off") +
  theme(legend.title = element_text(),
        axis.text.x = element_text(angle = 90, hjust = 1)) # needed to rotate x labels
```



ggplot histograms

You can more easily control the number of bins or the binwidth in a `ggplot` histogram here.

```r
library(ggplot2)
library(ggprism)

set.seed(100) # set random seed to be same every time
x = c(rnorm(50,10,2),
      rnorm(50,50,2))
df = data.frame(x)

ggplot(df, aes(x)) +
  geom_histogram(fill="gray", col="black", bins = 50) +
  theme_prism(border=T) +
  coord_cartesian(clip = "off") +
  theme(aspect.ratio = 1/1)
```



Suppose now that you have information about a circular genome, and a circular histogram would accurately display that information. . . of course, `ggplot` can do this.

```r
library(ggplot2)
genome = 4e6 # 4 Mb genome like E.coli and B. subtilis
set.seed(100) # set random seed for reproducibility
genomic_features = sample(genome, size = 5000, replace = T)
genome_data = data.frame(genomic_features)

myhist = ggplot(genome_data, aes(genomic_features)) +
            geom_histogram(binwidth=50000, fill="gray", col="black")

min_y = -max(ggplot_build(myhist)$data[[1]]$count)/3 # make lower bound 1/3 magnitude of max
max_y = max(ggplot_build(myhist)$data[[1]]$count)

circhist = myhist +
  coord_polar() +
  ylim(min_y, max_y) +
  theme_minimal() +
  theme(
      axis.text = element_blank(),
      axis.title = element_blank(),
      panel.grid = element_blank(),
      plot.margin = unit(rep(-0.7,4), "cm"))

circhist
```