An introduction to visualization in R

Tad Dallas

Contents

Why is data visualization important?

exploratory data analysis

Visualizations help identify interesting things about the data. This includes how the data are distributed, values that are outliers, and potential mistakes in data entry.

clear demonstration of results relative to tables or statistics

Visualizations are often more compelling at showing relationships between variables (and interactions between variables) than tables of statistical tests, and show more information than simple mean and standard deviations which would be reported in a table.

pretty

Visualizations can be striking ways to show relationships. Nearly all academic manuscripts will contain at least one or two figures, and knowing how to make data-rich figures is an important skill.

Visualizations in R all start by identifying the plotting area (the regions of space where the actual figure goes relative to the margins and other bits). This is done using the `par` function before a plot is created. Even if you do not explicitly use `par`, there are default arguments to `par` which are used whenever you call a plotting function (e.g., `plot`).

setting up the plotting window

`par` takes a number of arguments. I will only go over a couple of common ones that are important to making figures which eliminate unnecessary white space. The most important of these is `mar`, which sets up the exterior plotting margins. This is the amount of whitespace on either side of the plotting area.

```
par(mar=c(4,4,0.5,0.5))
```

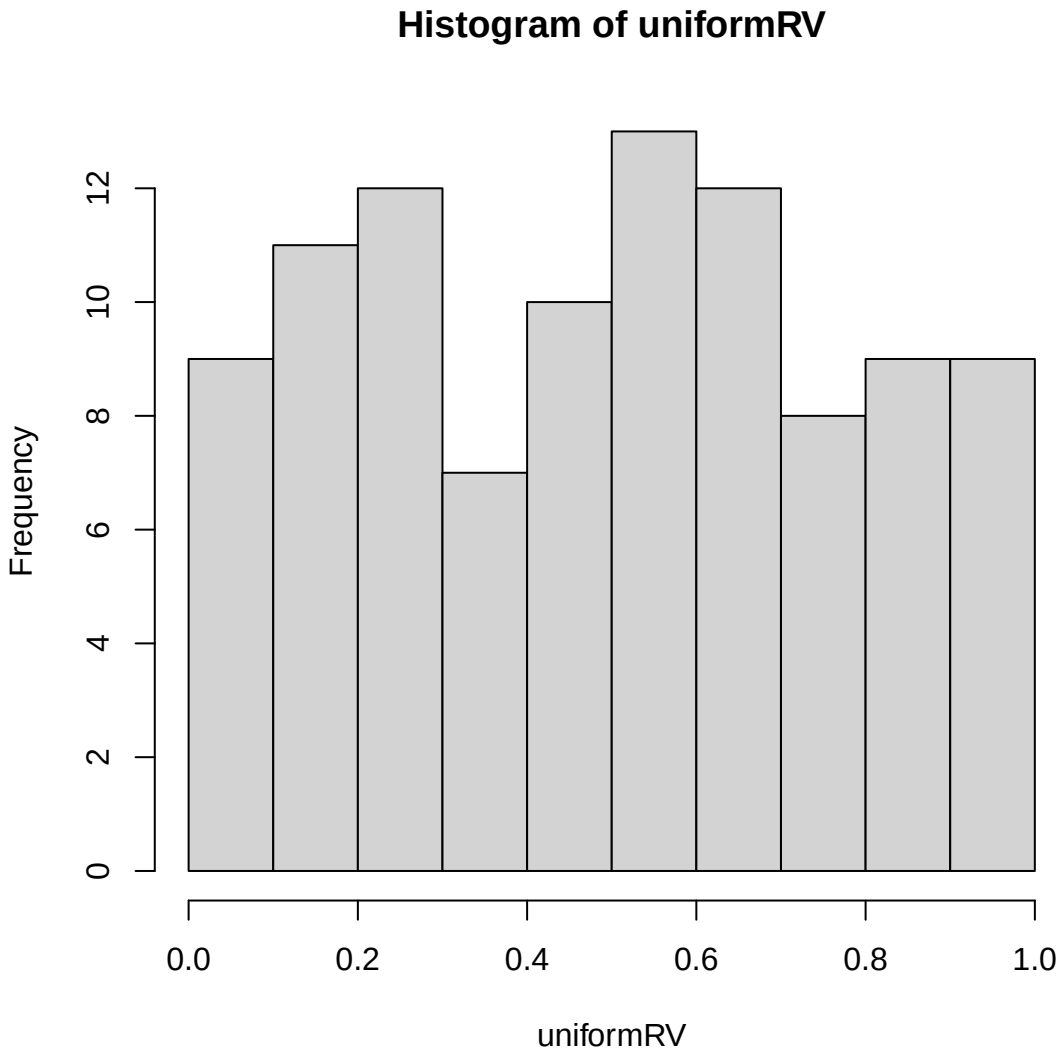
Running this code should open up a blank plotting window, but with margins present. The input to `mar` is a vector of 4 values, identifying the amount of space (in lines, I believe) for the bottom, left, top, and right plotting areas, respectively.

> What does the above code identify the margins as? Why do we opt to have values of 4 for the first 2 and then values of 0.5 for the second 2?

histograms

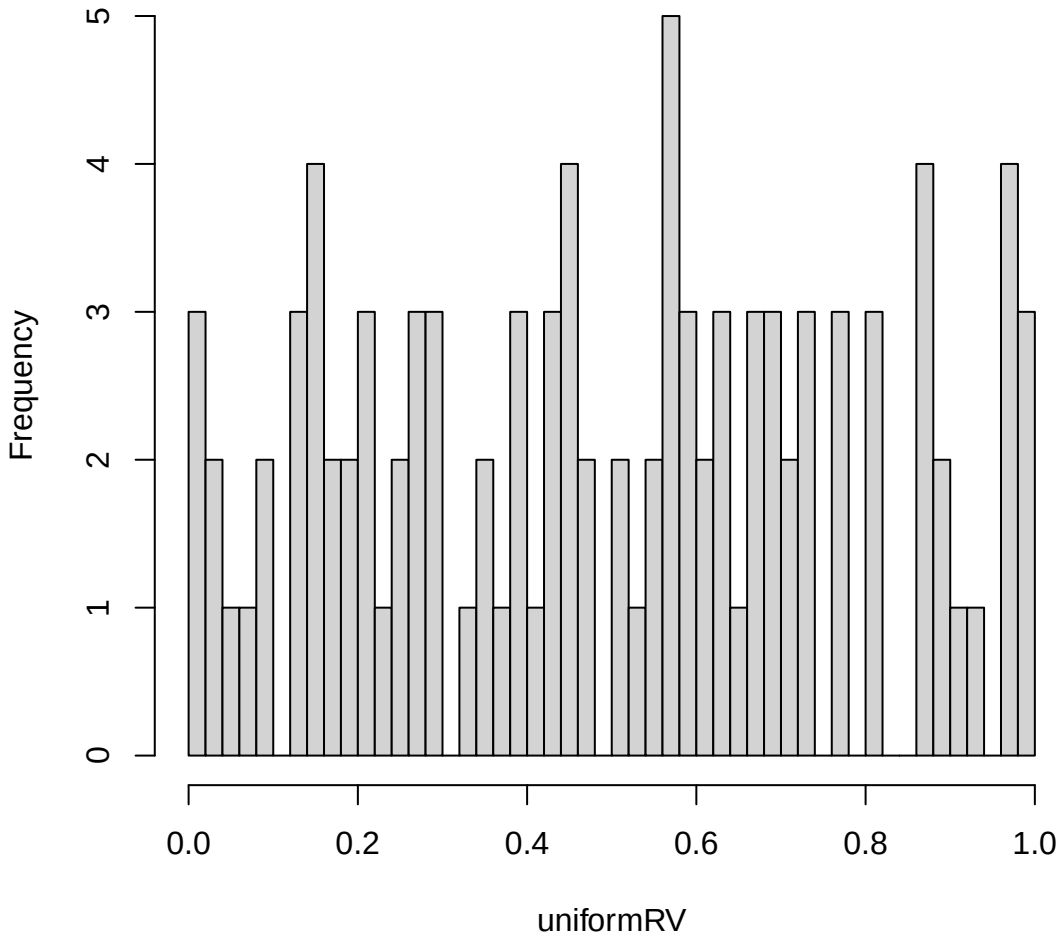
Histograms are useful to explore the distribution of your data. The goal is to show frequencies (the number of times your data falls into a given bin or range of values). This is useful to start to explore the distribution of your data.

```
uniformRV <- runif(100)
hist(uniformRV)
```

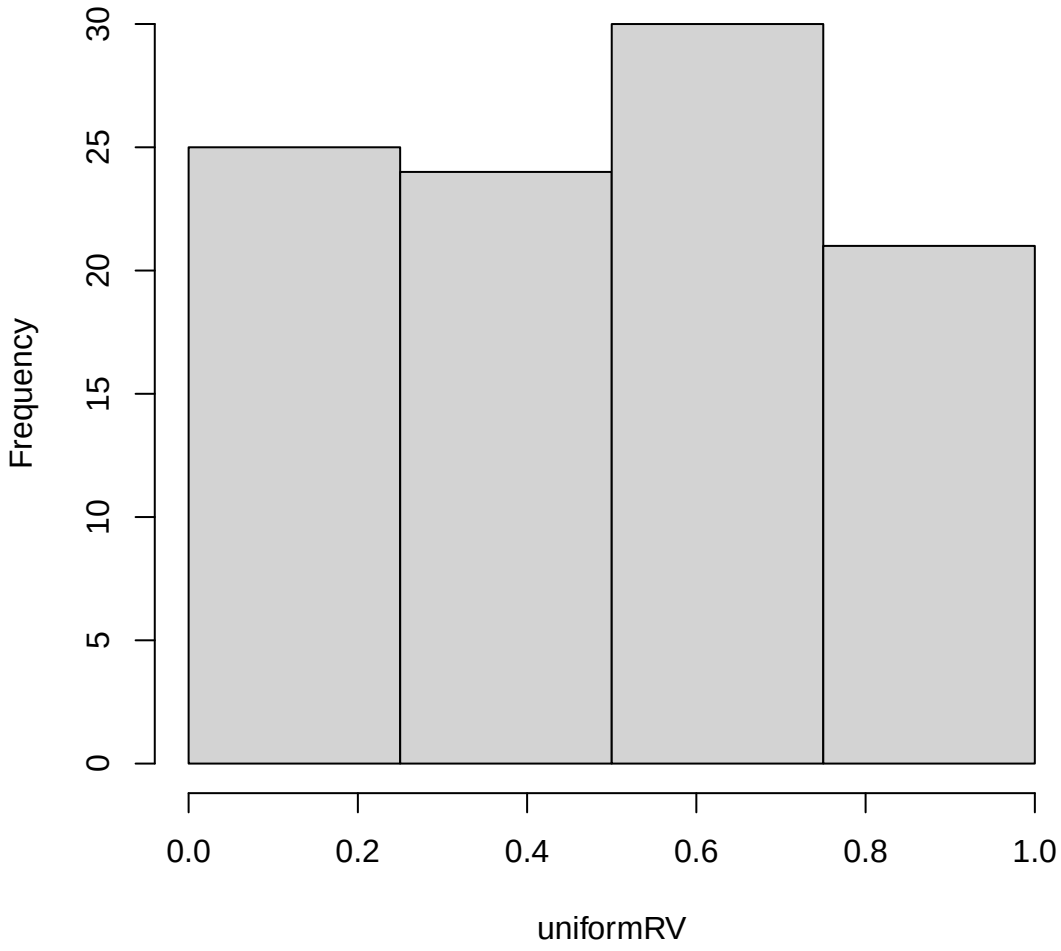


What we have done above is to create a vector of pulls from a uniform distribution (a distribution where every value between two bounds is equally likely of being sampled). The default range is between 0 and 1, so we should expect the histogram to look fairly 'flat', in that we expect that each bin in the histogram should have roughly the same frequency. Here we have 10 bins, so we should see about 10 observations in each bin. We can control the number of bins as well, using the `breaks` argument. This is useful because it also allows us more fine-scale control of how the data are binned.

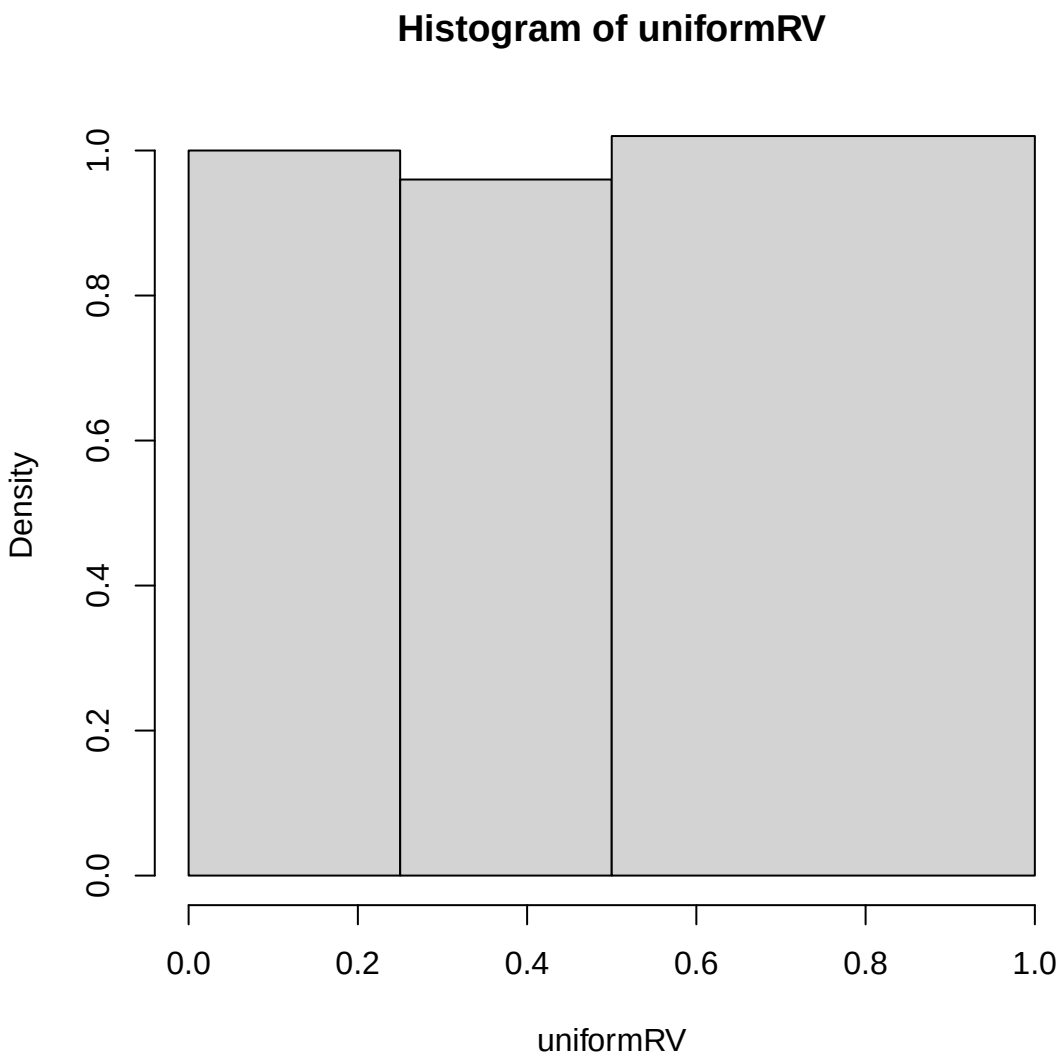
```
hist(uniformRV, breaks=50)
```




```
hist(uniformRV, breaks=c(0,0.25,0.5,0.75,1))
```

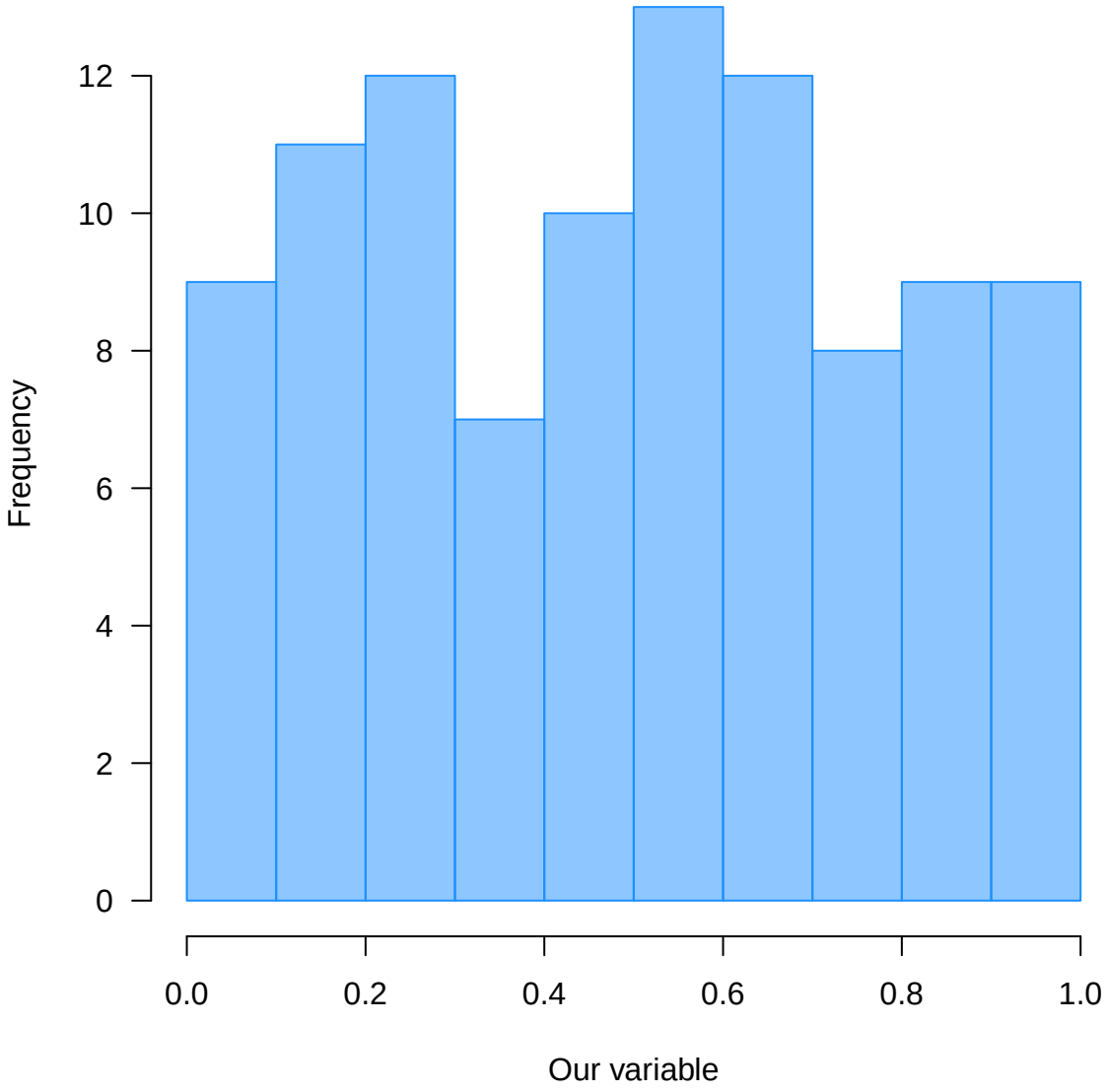



```
hist(uniformRV, breaks=c(0,0.25,0.5,1))
```



Unequal bin sizes are sometimes useful for displaying the data, but notice that there was a switch on the y-axis in terms of what is actually being represented? It went from "frequency" (count of number of observations falling into that bin) to "density". For now, we won't go into this change or why it's important. Instead, let's make this plot a bit less ugly.

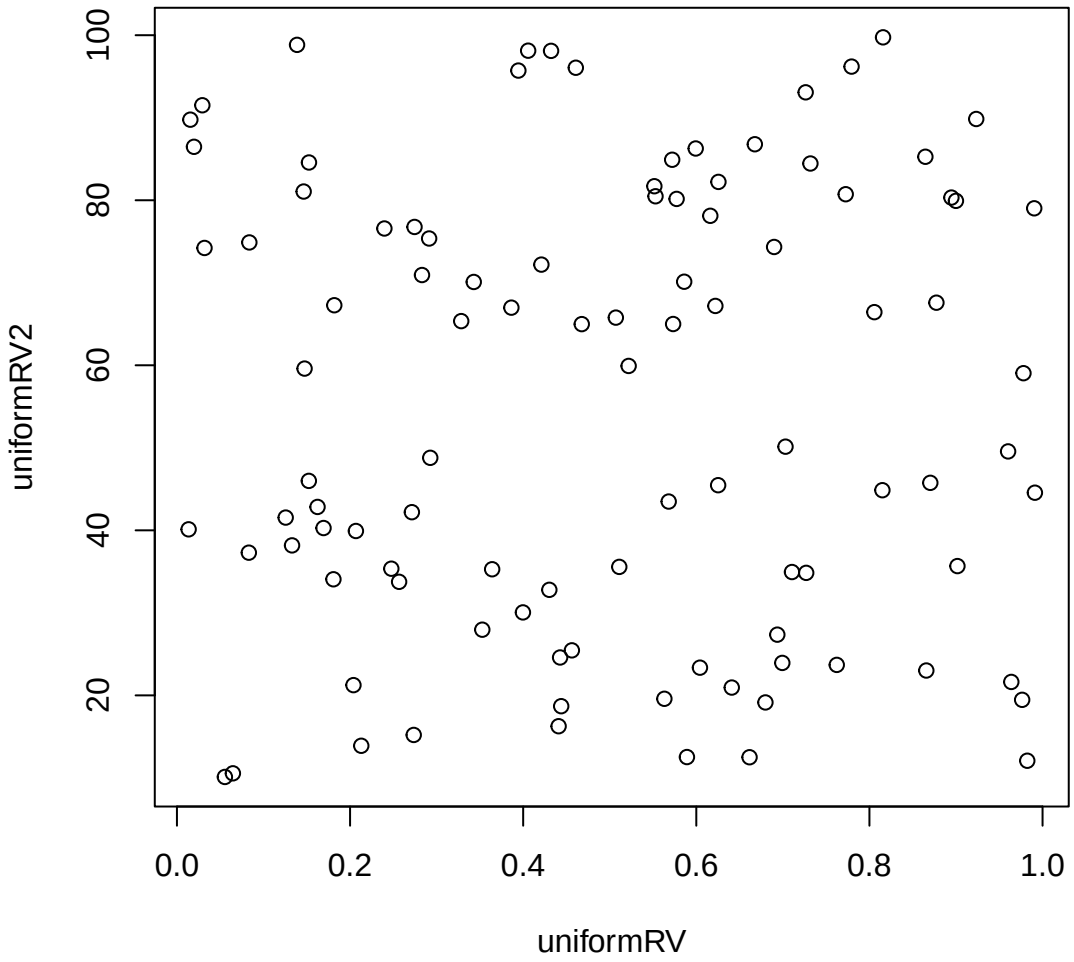
```
par(mar=c(4,4,0.5,0.5))
hist(uniformRV, breaks=10,
  xlab='Our variable', ylab='Frequency',
  main='', las=1,
  col=adjustcolor('dodgerblue', 0.5),
  border='dodgerblue')
```



scatterplots

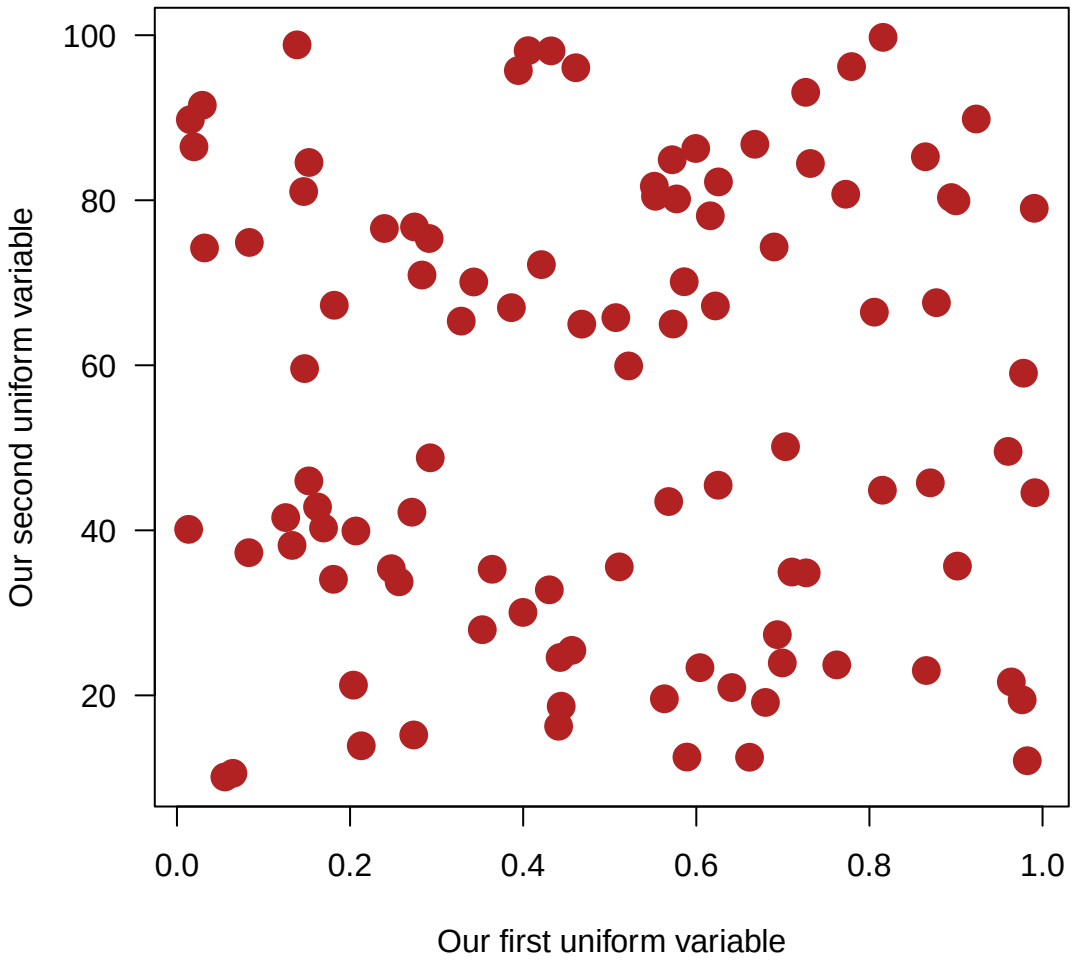
Most of the time, when we are thinking about plotting, we want to show the relationship between two or more variables. To do this, we can use the base `plot` function (which can powerfully handle many data of different types e.g., `numeric`, `factor`, etc.). So let's create a second variable (`uniformRV2`) and explore the relationship between the two continuous variables.

```
uniformRV2 <- runif(100, 10, 100)

plot(uniformRV, uniformRV2)
```



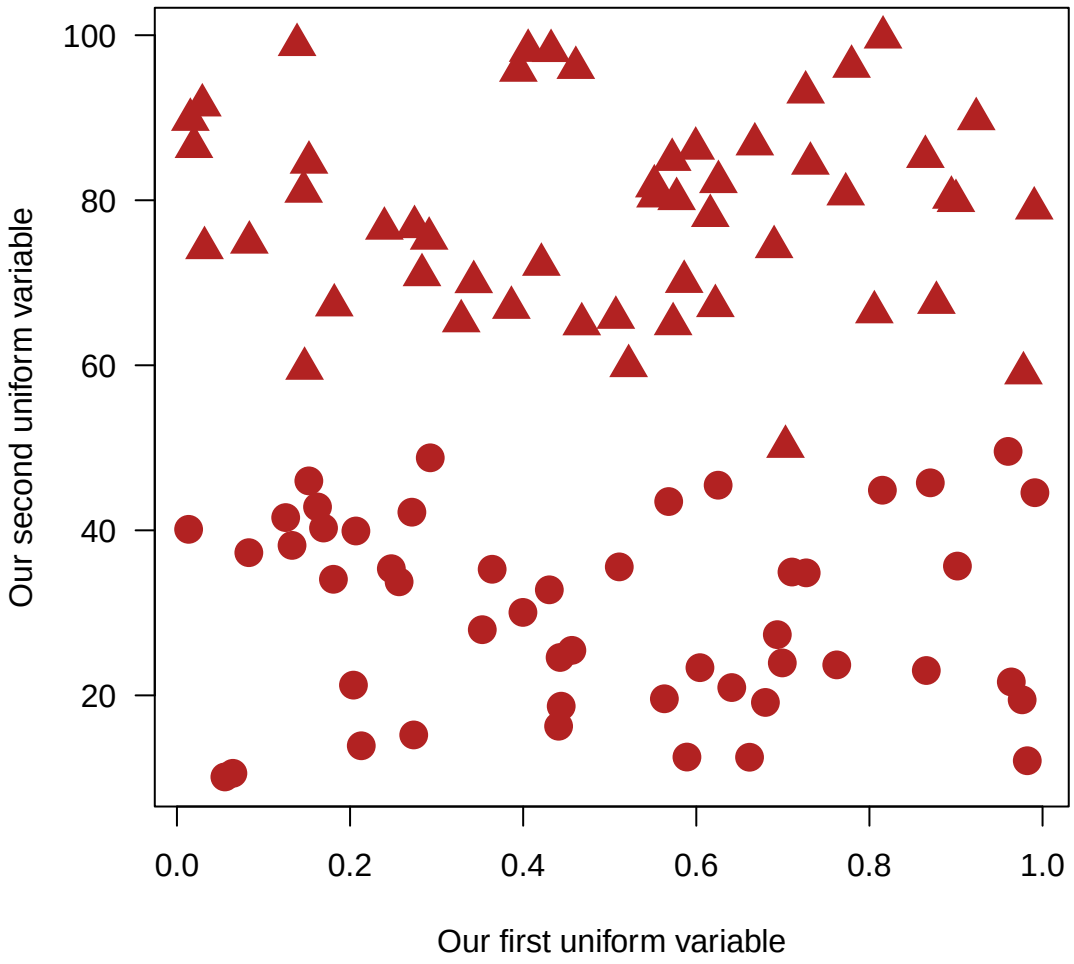
And it looks like there's no relationship between the variables (because there isn't, right?). Another thing to note is that the order matters (`x` is the first argument to the plot function, so we could be more explicit about how we hand variables to the plotting function). And we can also start making this prettier. `hist` had a smaller list of arguments that we could hand to manipulate the display of the data. `plot` has many more arguments, as `plot` is the main function for visualization in R. In fact, packages written in R that deal with different types of data work with the `plot` argument to display data well beyond what the `plot` function was initially designed for, as we will see when we start visualizing maps and other fun stuff. Take some time and explore what arguments `plot` can take by issuing the command `?plot` into the R console.

```
plot(x=uniformRV, y=uniformRV2,
  pch=16, cex=2, las=1,
  xlab = 'Our first uniform variable',
  ylab='Our second uniform variable',
  col='firebrick')
```



It is important to note here that (as in many other areas of programming), the `plot` function will try to work with whatever information you give it, and this can sometimes lead you down weird paths. For instance, the `col`, `cex`, and `pch` arguments can accept vectors, where it expects you to provide a vector of the same length as the number of data points you have. This can be useful, as in the example below where we can color all points above a threshold a different color or point shape (`pch`).

```
plot(x=uniformRV, y=uniformRV2,
  pch=c(16,17)[1+(uniformRV2 > 50)],
  cex=2, las=1,
  xlab = 'Our first uniform variable',
  ylab='Our second uniform variable',
  col='firebrick')
```



This builds on what we previously learned about **conditionals**, as well as what we've learned about indexing vectors. Let's break this down. For the `pch` argument above, we create a vector of two values (`c(16,17)`) and then index these based on the output of a conditional.

```
1+(uniformRV2 > 50)
```

```
##   [1] 2 1 2 2 1 2 1 2 2 1 1 1 1 2 2 1 1 1 1 1 2 2 1 2 2 2 1 2 1 2 2 1 1 2 1 2 1
##  [38] 2 1 1 2 1 2 2 2 2 2 2 1 1 2 1 2 2 1 1 1 1 2 2 2 2 2 2 1 2 1 2 1 2 1 1 2 2
##  [75] 1 1 2 1 2 2 1 1 2 1 2 1 2 1 2 1 1 1 1 2 2 2 1 1 2 1
```

Do the same as above, but indexing different colors for points on the x-axis (`uniformRV`).

Showing mean and standard deviation

A common visualization is to show the mean and standard deviation of some continuous values across treatments. For instance, let's say we have an experiment where we expose plants to different combinations of nitrogen and phosphorous. If we have a single level of each of N and P, then it means we have 4 treatments (control, N, P, N+P), right?

What is the importance of the control in this experiment?

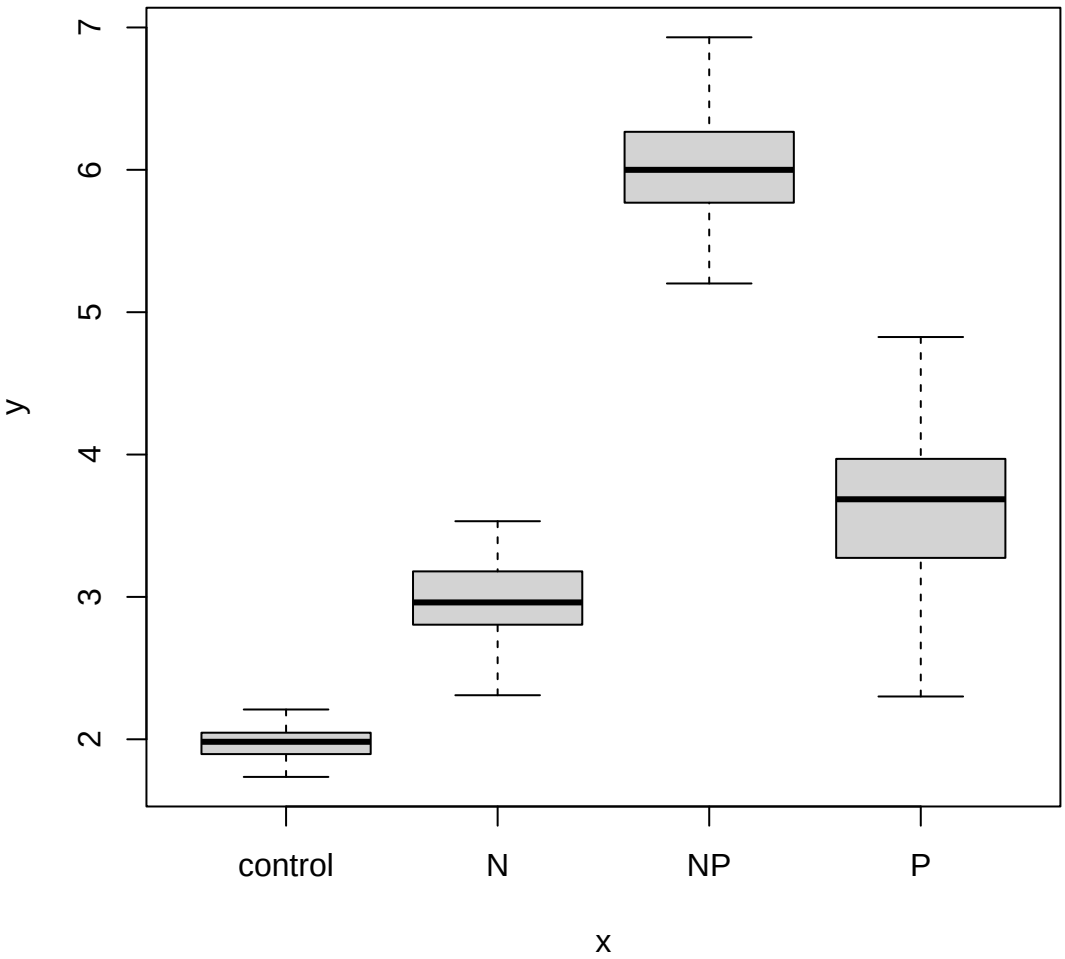
```
control <- rnorm(100, 2, 0.1)
N <- rnorm(100, 3, 0.25)
P <- rnorm(100, 3.5, 0.5)
NP <- rnorm(100, 6, 0.4)
```

How would you first start to visualize the data?

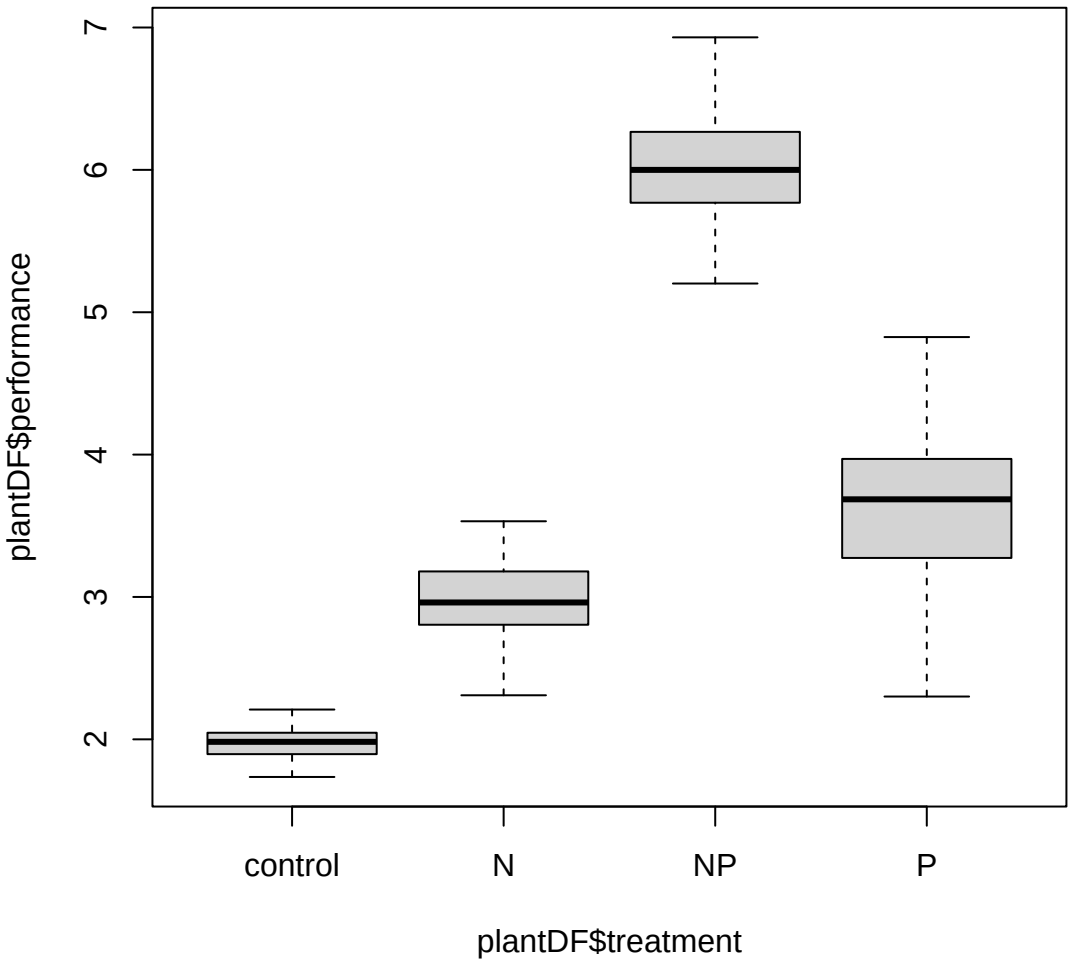
There are many different ways we could visualize the differences among treatments, depending on the amount of data we want to show. I'll start with the simplest, and then we can spend some time thinking about other ways to display the data.

So let's say I only want to see the means and standard deviations of the treatments. This means my x-axis will be treatment, and the y-axis will be the values obtained from the experiment in terms of plant performance.

```
plantDF <- data.frame(performance=c(control, N, P, NP),
  treatment=c(
    rep('control',100), rep('N', 100),
    rep('P', 100), rep('NP',100))
)

plot(as.factor(plantDF$treatment), plantDF$performance)
```



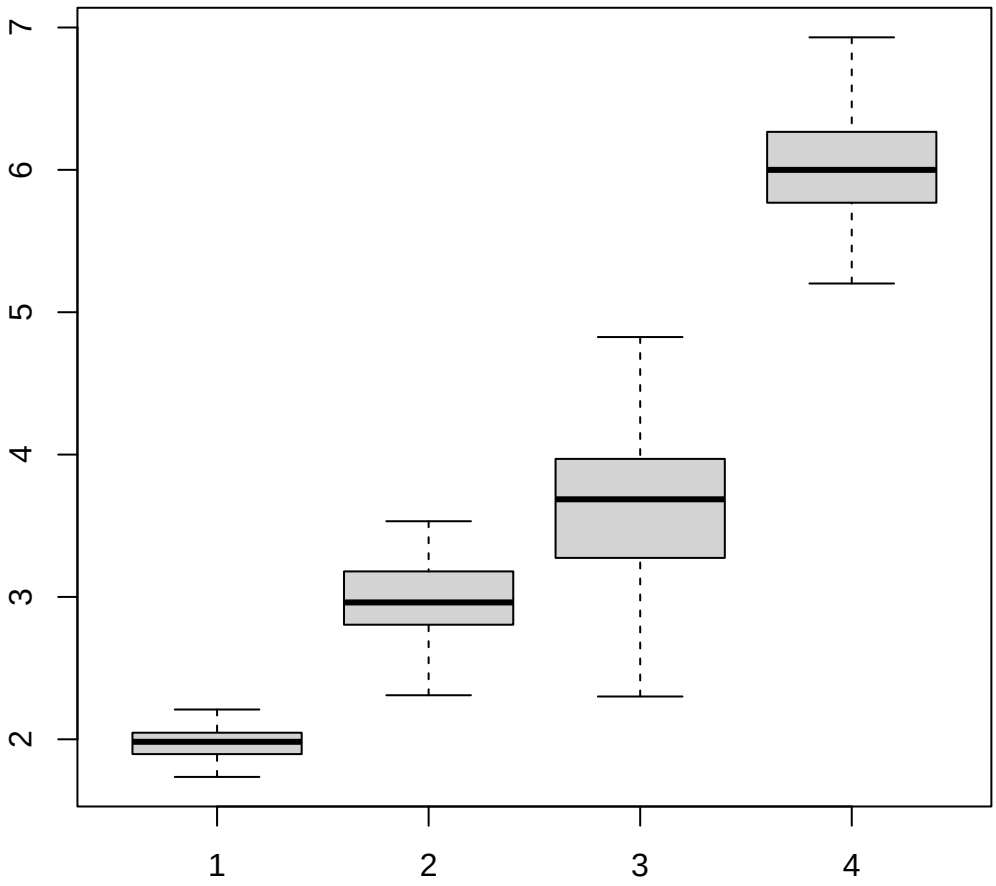

```
boxplot(plantDF$performance ~ plantDF$treatment)
```



Wait, what have we plotted? And why are there two ways to get the exact same plot?

This has to do with the plotting function being the R multi-tool in terms of visualization. I hand it a categorical variable and a continuous variable, and it defaults to visualizing these as a boxplot. There is also the `boxplot` function, which does the same thing, but needs the `formula` interface. The nice part about `boxplot` is that you can hand it as many vectors as you want and it'll create more levels to the plot.

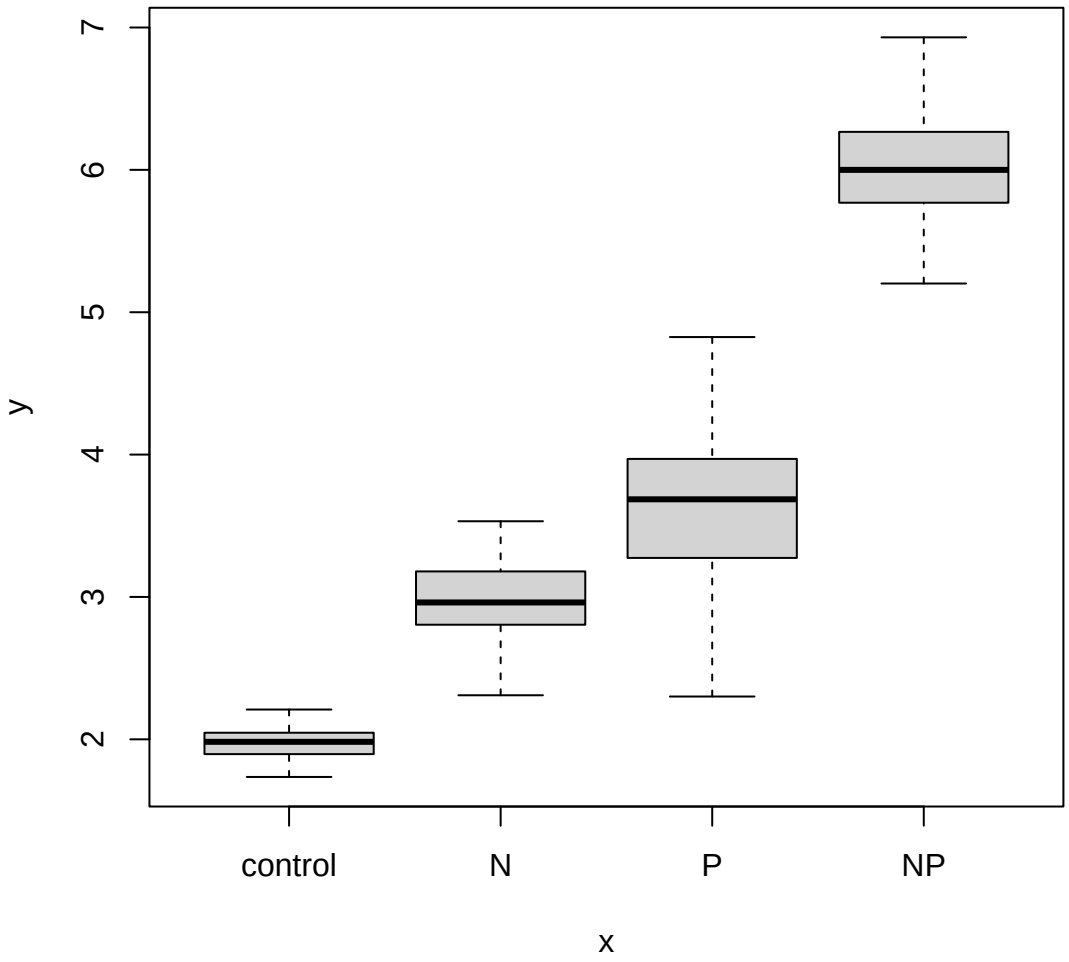
```
boxplot(control, N, P, NP)
```



But wait... something else interesting happened here, right? There was a shift between the variables in the third and fourth positions. What's going on there?

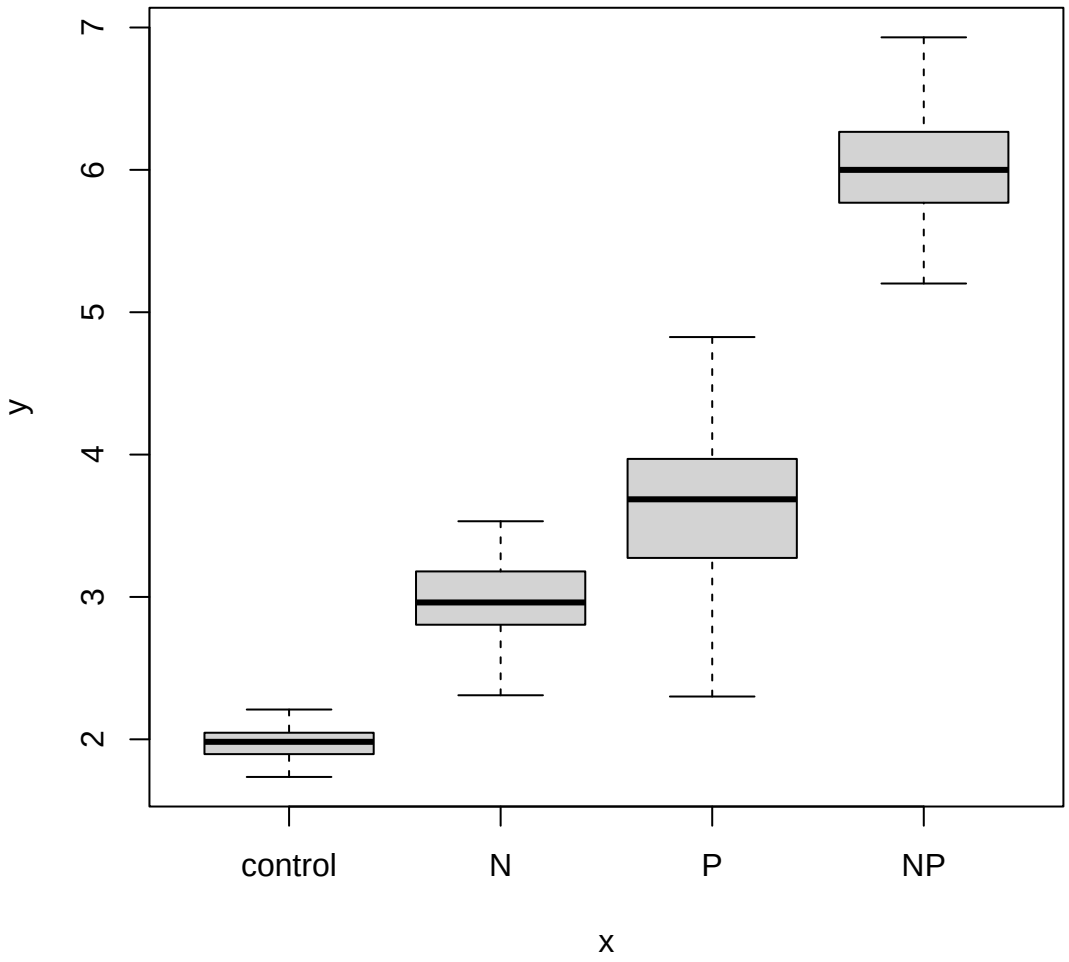
R is ordering the x-axis alphabetically, such that our `NP` treatment is in the third position for the first two plots, but last when we explicitly hand the boxplot function the order we want.

```
plantDF$trtFactor <- factor(plantDF$treatment, levels=unique(plantDF$treatment))

plot(plantDF$trtFactor, plantDF$performance)
```



This was a bit of a hacky solution, as I knew the order of the `levels` when I called `unique`. So `unique` provided a vector of the unique treatment levels, but it did so sequentially (so the way I had the data formatted forced it to be `control, N, P, NP`). We can also specify the order of the levels ourselves.

```
plantDF$trtFactor <- factor(plantDF$treatment, levels=c('control', 'N', 'P', 'NP'))

plot(plantDF$trtFactor, plantDF$performance)
```



So there's one way to visualize these data. Let's spend the next 10 minutes seeing what you can do to improve this visualization, either aesthetically, or by plotting it differently (are we seeing too much detail? not enough detail?)

? maybe go into some of the boxplot specific syntax for making those less ugly?

? Worth going into linear models and anova at this point?

```
summary(lm(plantDF$performance ~ plantDF$trtFactor))
```

```
##
## Call:
## lm(formula = plantDF$performance ~ plantDF$trtFactor)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -1.31403 -0.19207  0.00507  0.18106  1.21082
##
## Coefficients:
##                      Estimate Std. Error t value Pr(>|t|)
## (Intercept)           1.97720    0.03619   54.64   <2e-16 ***
## plantDF$trtFactorN    1.00184    0.05117   19.58   <2e-16 ***
## plantDF$trtFactorP    1.63748    0.05117   32.00   <2e-16 ***
## plantDF$trtFactorNP   4.06310    0.05117   79.40   <2e-16 ***
## ---
```

```
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.3619 on 396 degrees of freedom
## Multiple R-squared:  0.9453, Adjusted R-squared:  0.9449
## F-statistic:  2282 on 3 and 396 DF,  p-value: < 2.2e-16
```

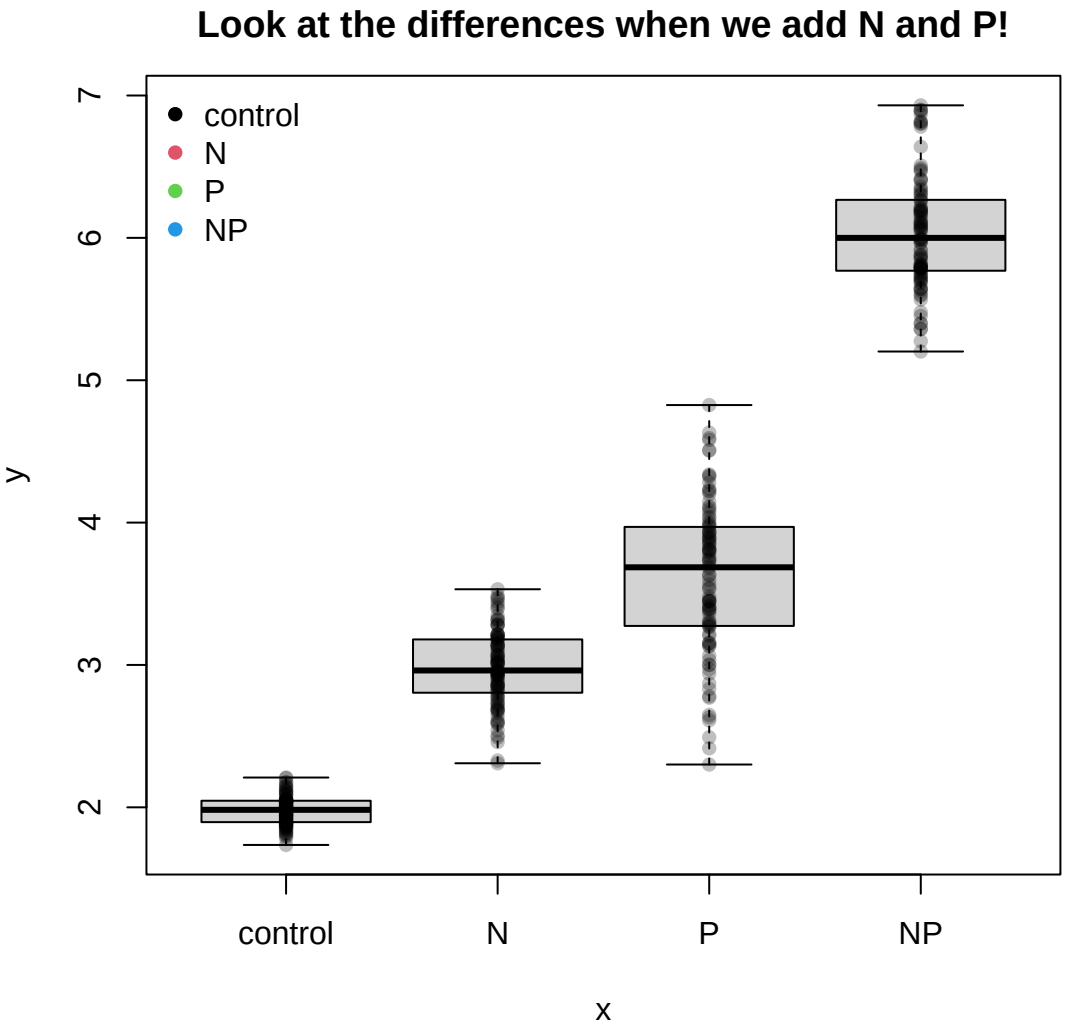
```
t.test(plantDF$performance[which(plantDF$trt=='control')],
  plantDF$performance[which(plantDF$trt=='N')])
```

```
##
##  Welch Two Sample t-test
##
## data:  plantDF$performance[which(plantDF$trt == "control")] and plantDF$performance[which(plantDF$tr
## t = -35.101, df = 126.92, p-value < 2.2e-16
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -1.058324 -0.945365
## sample estimates:
## mean of x mean of y
##  1.977200  2.979045
```

Adding things on top of visualizations

There are often many things that cannot be done with a single plotting call. That is, we need to add something to the plotting area (e.g., a legend). R has numerous functions that add things to existing plots, allowing complete controllability of what gets plotted.

```
plot(plantDF$trtFactor, plantDF$performance)
title('Look at the differences when we add N and P!', line=1)
points(y=plantDF$performance, x=plantDF$trtFactor,
  col=adjustcolor(1,0.25), pch=16)

legend('topleft', legend=c('control', 'N', 'P', 'NP'),
  col=1:4, pch=16, bty='n')
```



Make the colors of the additional points (or of the boxes in the boxplot) match the colors that we have supplied in the legend?

In practice, this is not a great visualization. That is, we don't need the legend, as the x-axis already contains all that of that information. The goal with plotting is to keep the output as simple as possible while layering on the necessary and important information.

sessionInfo

```
sessionInfo()
```

```
## R version 4.3.1 (2023-06-16)
## Platform: x86_64-pc-linux-gnu (64-bit)
## Running under: Ubuntu 22.04.2 LTS
##
## Matrix products: default
## BLAS:   /usr/lib/x86_64-linux-gnu/blas/libblas.so.3.10.0
## LAPACK: /usr/lib/x86_64-linux-gnu/lapack/liblapack.so.3.10.0
##
## locale:
##  [1] LC_CTYPE=en_US.UTF-8       LC_NUMERIC=C
##  [3] LC_TIME=en_US.UTF-8        LC_COLLATE=en_US.UTF-8
```

```
##  [5] LC_MONETARY=en_US.UTF-8    LC_MESSAGES=en_US.UTF-8
##  [7] LC_PAPER=en_US.UTF-8       LC_NAME=C
##  [9] LC_ADDRESS=C               LC_TELEPHONE=C
## [11] LC_MEASUREMENT=en_US.UTF-8 LC_IDENTIFICATION=C
##
## time zone: America/New_York
## tzcode source: system (glibc)
##
## attached base packages:
## [1] stats     graphics  grDevices utils     datasets  methods   base
##
## other attached packages:
## [1] ROCR_1.0-11    geodata_0.5-8  terra_1.7-39   maps_3.4.1     rgbif_3.7.7
## [6] gbm_2.1.8.1    rmarkdown_2.23
##
## loaded via a namespace (and not attached):
##  [1] viridis_0.6.3     utf8_1.2.3        generics_0.1.3    xml2_1.3.5
##  [5] stringi_1.7.12    lattice_0.21-8    httpcode_0.3.0    digest_0.6.33
##  [9] magrittr_2.0.3    evaluate_0.21     grid_4.3.1        fastmap_1.1.1
## [13] plyr_1.8.8        jsonlite_1.8.7    Matrix_1.6-0      whisker_0.4.1
## [17] crul_1.4.0        tinytex_0.45      survival_3.5-5    urltools_1.7.3
## [21] gridExtra_2.3     httr_1.4.6        fansi_1.0.4       viridisLite_0.4.2
## [25] scales_1.2.1      oai_0.4.0         codetools_0.2-19  lazyeval_0.2.2
## [29] cli_3.6.1         rlang_1.1.1       triebeard_0.4.1   munsell_0.5.0
## [33] splines_4.3.1     yaml_2.3.7        parallel_4.3.1    tools_4.3.1
## [37] dplyr_1.1.2       colorspace_2.1-0  ggplot2_3.4.2     curl_5.0.1
## [41] vctrs_0.6.3       R6_2.5.1          lifecycle_1.0.3   stringr_1.5.0
## [45] pkgconfig_2.0.3   pillar_1.9.0      gtable_0.3.3      data.table_1.14.8
## [49] glue_1.6.2        Rcpp_1.0.11       xfun_0.39         tibble_3.2.1
## [53] tidyselect_1.2.0  highr_0.10        knitr_1.43        htmltools_0.5.5
## [57] compiler_4.3.1
```