Part 1 - The basics

R is just a calculator.

In [40]:
```
2+2
```

4

In [41]:
```
5+10
```

15

There is special syntax for multiplication and division.

In [42]:
```
10*3
```

30

In [43]:
```
12/4
```

3

We can also assign numbers to variables and to operations.

In [44]:
```
A = 2
B = 4
A + B
```

6

A function is just something with an input and an output.

In [45]:
```
sum(5,5)
```

10

In [46]:
```
sqrt(B)
```

2

In [47]:
```
sum(A,B)
```

6

Part 2 - A bit more advanced

The nice thing about R is that unlike on a calculator, you can keep track of your operations. This is important for reproducing your work, and doing more complicated stuff. Let's look at some of this more complicated stuff!

We can assign lists of numbers to variables. This is called an array. 'c' is just a function. It means 'concatenate'.

In [48]:
```
A <- c(4,2,8, 10, 6)
B <- c(8,17,2, 8, 6)
```

We can do operations on lists too, the same as for simpler variables. We can make a new variable in the process.

In [49]:
```
C = A + B
```

To see the output, you need to call the variable directly.

In [50]:
```
C
```

$12 \cdot 19 \cdot 10 \cdot 18 \cdot 12$

We can use other functions to make lists. We can make sequences...

In [51]:
```
A = seq(0,8,2)
A
```

$0 \cdot 2 \cdot 4 \cdot 6 \cdot 8$

...or repeat numbers

In [52]:
```
B = rep(10, 4)
B
```

$10 \cdot 10 \cdot 10 \cdot 10$

We can also generate random lists.

In [53]:
```
# runif(iteration, min, max)
weight1 = runif(20, 0, 10)
weight2 = runif(20, 10, 20)

weight1
weight2
```

$4.02282168157399 \cdot 7.36743829445913 \cdot 3.86971024796367 \cdot 3.85985085973516 \cdot 0.317090165335685 \cdot 5.66608191700652 \cdot 8.70465866755694 \cdot 7.275123486761 \cdot 8.87709266971797 \cdot 6.94583089090884 \cdot 3.80971017992124 \cdot 5.10769734857604 \cdot 8.24764204677194 \cdot 5.32008058391511 \cdot 5.67995429970324 \cdot 2.88394647883251 \cdot 9.7403873433359 \cdot 1.63132434245199 \cdot 4.29301923373714 \cdot 8.31335004651919$

$18.2124416064471 \cdot 18.7904014764354 \cdot 12.6591618428938 \cdot 14.866896814201 \cdot 14.928466216661 \cdot 16.7292733583599 \cdot 12.6962541276589 \cdot 15.885639579501 \cdot 14.407747972291 \cdot 14.8633555416018 \cdot 12.1070770965889 \cdot 13.9465122297406 \cdot 13.806713167578 \cdot 17.7244890667498 \cdot 13.2509942725301 \cdot 17.4240779154934 \cdot 13.2308574137278 \cdot 10.809252646286 \cdot 17.4764641164802 \cdot 17.524515518453$

A uniform distribution is a probability distribution in which all outcomes are equally likely, resulting in a constant probability density across a specified interval.

If you are unsure about how a function works, there's a built-in way to get help:

In [54]:
```
?runif
```

The Uniform Distribution

Description

These functions provide information about the uniform distribution on the interval from `min` to `max`. `dunif` gives the density, `punif` gives the distribution function `qunif` gives the quantile function and `runif` generates random deviates.

Usage

```
dunif(x, min = 0, max = 1, log = FALSE)
punif(q, min = 0, max = 1, lower.tail = TRUE, log.p = FALSE)
qunif(p, min = 0, max = 1, lower.tail = TRUE, log.p = FALSE)
runif(n, min = 0, max = 1)
```

Arguments

`x, q`	vector of quantiles.
`p`	vector of probabilities.
`n`	number of observations. If `length(n) > 1`, the length is taken to be the number required.
`min, max`	lower and upper limits of the distribution. Must be finite.
`log, log.p`	logical; if TRUE, probabilities p are given as log(p).
`lower.tail`	logical; if TRUE (default), probabilities are `P[X \le x]`, otherwise, `P[X > x]`.

Details

If `min` or `max` are not specified they assume the default values of `0` and `1` respectively.

The uniform distribution has density

$$f(x) = \frac{1}{max-min}$$

for `min \le x \le max`.

For the case of `u := min == max`, the limit case of `X \equiv u` is assumed, although there is no density in that case and `dunif` will return `NaN` (the error condition).

`runif` will not generate either of the extreme values unless `max = min` or `max-min` is small compared to `min`, and in particular not for the default arguments.

Value

`dunif` gives the density, `punif` gives the distribution function, `qunif` gives the quantile function, and `runif` generates random deviates.

The length of the result is determined by `n` for `runif`, and is the maximum of the lengths of the numerical arguments for the other functions.

The numerical arguments other than `n` are recycled to the length of the result. Only the first elements of the logical arguments are used.

Note

The characteristics of output from pseudo-random number generators (such as precision and periodicity) vary widely. See `.Random.seed` for more information on **R**'s random number generation algorithms.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

`RNG` about random number generation in **R**.

Distributions for other standard distributions.

Examples

```
u <- runif(20)

## The following relations always hold :
punif(u) == u
dunif(u) == 1

var(runif(10000))  #- ~ = 1/12 = .08333
```

[Package *stats* version 4.4.2]

We don't have to use a uniform distribution. Actually, a normal distribution will be better. Let's do that instead.

In [55]:
```
weight1 = rnorm(20, 10, 20)
weight2 = rnorm(20, 10, 20)

weight1
weight2
```

-5.75765520022185 · 3.95047002401602 · 39.8082185969646 · -26.4170833123525 · -11.9567039799787 · -21.9113630814711 · -11.7038460414286 · 23.747068667741 · -7.149846410816 · 27.0337650644571 · 1.92267995415769 · -16.6714767887723 · -8.22552128524041 · -1.50573210122105 · 30.1059095592605 · 28.4818512992909 · 31.7578406773292 · -21.060304217081 · 36.7476675730236 · 19.9254249090842

22.8614889925309 · 17.3048572575358 · 14.4568770044311 · -1.49286106090126 · -23.8546530482248 · 1.10400390040186 · -3.5971941064263 · 5.67452586854355 · -29.6741829141018 · 13.4634199724689 · 35.6594832782067 · -3.69022970742043 · 56.0340866470725 · 33.6371888175838 · 34.3768423234046 · 15.9775311516616 · -7.5425969313492 · -19.8875267410882 · 49.519185025254 · 31.2715440726483

A normal distribution is a symmetric, bell-shaped probability distribution characterized by its mean and standard deviation, where approximately 68% of the data falls within one standard deviation of the mean.

We can also make lists of words. This can be useful if our variables are categorical.

In [56]:
```
species1 = rep("dog", 20)
species2 = rep("cat", 20)
species1
species2
```

'dog' · 'dog' · 'dog' · 'dog' · 'dog' · 'dog' · 'dog' · 'dog' · 'dog' · 'dog' · 'dog' · 'dog' · 'dog' · 'dog' · 'dog' · 'dog' · 'dog' · 'dog' · 'dog' · 'dog'

'cat' · 'cat' · 'cat' · 'cat' · 'cat' · 'cat' · 'cat' · 'cat' · 'cat' · 'cat' · 'cat' · 'cat' · 'cat' · 'cat' · 'cat' · 'cat' · 'cat' · 'cat' · 'cat' · 'cat'

We can also manipulate arrays. This can be necessary for tidying data, and getting it into the right format. We can for example append one array to another. We just use the c() function for this.

In [57]:
```
Weight = c(weight1, weight2)
Species = c(species1, species2)
Weight
```

-5.75765520022185 · 3.95047002401602 · 39.8082185969646 · -26.4170833123525 · -11.9567039799787 · -21.9113630814711 · -11.7038460414286 · 23.747068667741 · -7.149846410816 · 27.0337650644571 · 1.92267995415769 · -16.6714767887723 · -8.22552128524041 · -1.50573210122105 · 30.1059095592605 · 28.4818512992909 · 31.7578406773292 · -21.060304217081 · 36.7476675730236 · 19.9254249090842 · 22.8614889925309 · 17.3048572575358 · 14.4568770044311 · -1.49286106090126 · -23.8546530482248 · 1.10400390040186 · -3.5971941064263 · 5.67452586854355 · -29.6741829141018 · 13.4634199724689 · 35.6594832782067 · -3.69022970742043 · 56.0340866470725 · 33.6371888175838 · 34.3768423234046 · 15.9775311516616 · -7.5425969313492 · -19.8875267410882 · 49.519185025254 · 31.2715440726483

This gives us our weights and species as nice tidy data arrays. We can also organise our data into spreadsheets like excel. These are called data frames in R.

In [58]:
```
df = data.frame(Weight, Species)
```

You can always take a look at the dataframe by calling it.

In [59]:
```
df
```

A data.frame: 40 × 2

Weight	Species
<dbl>	<chr>
-5.757655	dog
3.950470	dog
39.808219	dog
-26.417083	dog
-11.956704	dog
-21.911363	dog
-11.703846	dog
23.747069	dog
-7.149846	dog
27.033765	dog
1.922680	dog
-16.671477	dog
-8.225521	dog
-1.505732	dog
30.105910	dog
28.481851	dog
31.757841	dog
-21.060304	dog
36.747668	dog
19.925425	dog
22.861489	cat
17.304857	cat
14.456877	cat
-1.492861	cat
-23.854653	cat
1.104004	cat
-3.597194	cat
5.674526	cat
-29.674183	cat
13.463420	cat
35.659483	cat

Weight	Species
<dbl>	<chr>
-3.690230	cat
56.034087	cat
33.637189	cat
34.376842	cat
15.977531	cat
-7.542597	cat
-19.887527	cat
49.519185	cat
31.271544	cat

We can look at just the top of this spreadsheet.

In [63]:
```
head(df)
```

A data.frame: 6 × 2

	Weight	Species
	<dbl>	<chr>
1	-5.757655	dog
2	3.950470	dog
3	39.808219	dog
4	-26.417083	dog
5	-11.956704	dog
6	-21.911363	dog

We can also pick out specific variables to do stuff to. We use the '$' for this.

In [66]:
```
df$Species
```

'dog' · 'cat' · 'cat'

We can also save our spreadsheet.

In [67]:
```
save(df, file = "SpeciesNWeight.RData")
```

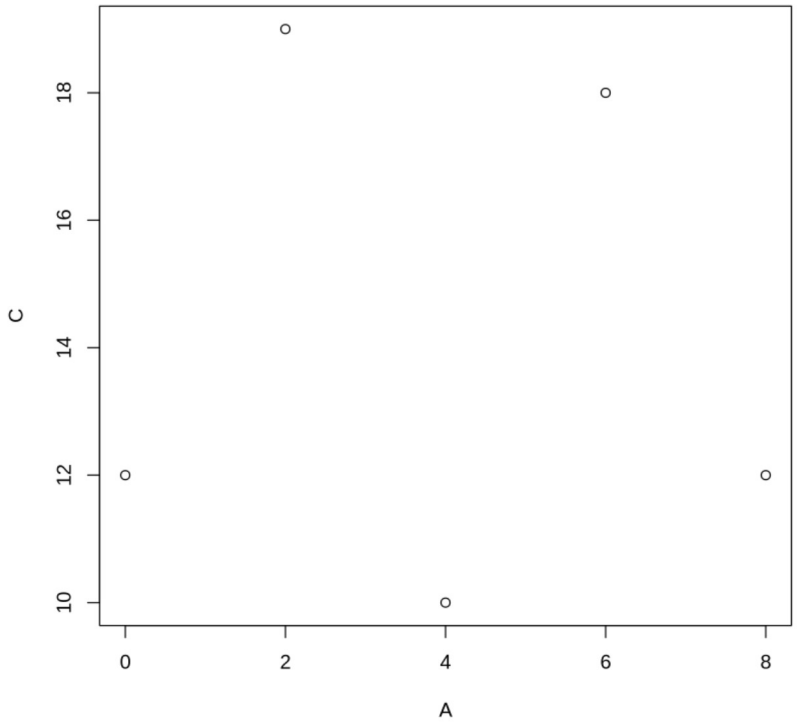
Part 3 - Basic plotting

Plotting is very important. It allows you to see relationships between variables. There are different kinds of plots that are useful for different kinds of relationships.

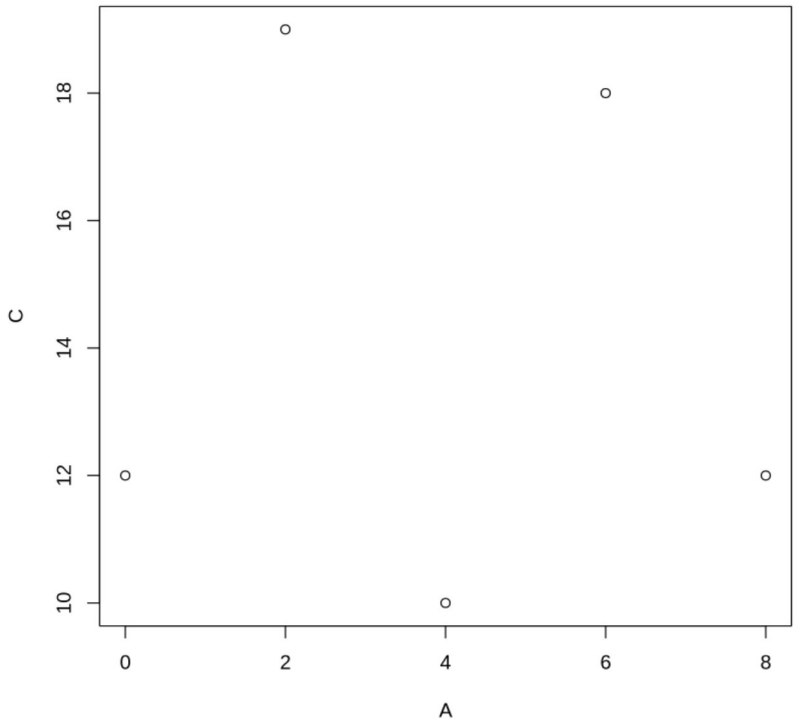
If you want to plot something very quickly, base R (when you don't download any extra packages) has some great options.

To visualize the relationship between two continuous variables, you can use a scatterplot...

In [68]:
```
plot(A, C)
```

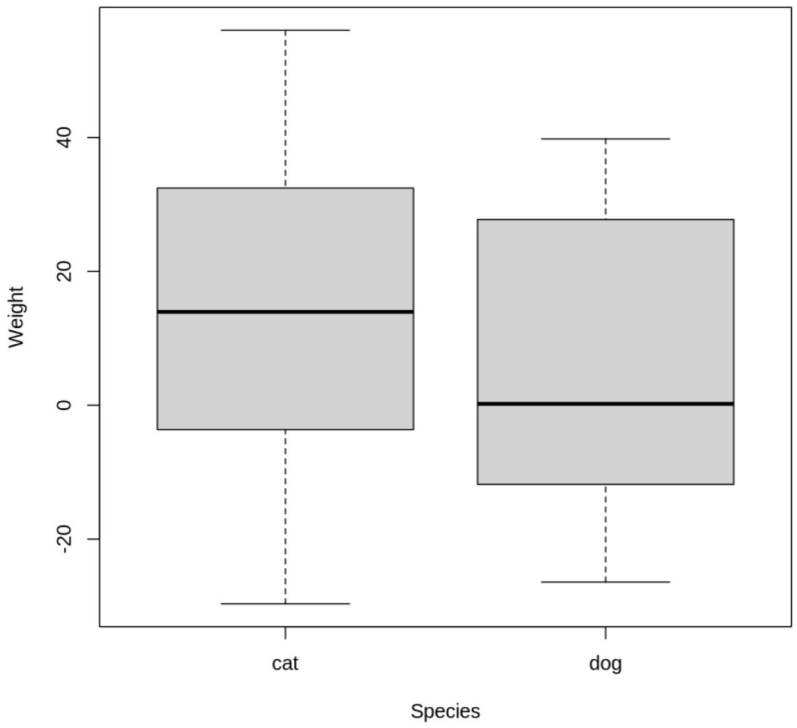


In [69]:
```
plot(C~A)
```



... while boxplots tell you the relationship between categorical and continuous variables.

In [70]:
```
boxplot(Weight~Species)
```



The tilde "~" is important in R. It can be read to mean "as a function of", or "distributed as". You will see it again, and you will get used to it. This little syntactic difference is important, because it can allow us to make the same plot using different kinds of data

structures.

These plots are a bit ugly. Even base R can make beautiful plots, you can look at options to fancy plots up using the ?plot command.

A lot of the R community uses a package called ggplot2 to plot. It is very well-documented online and has a lot of options for customization. Packages are like modules in Python. We'll talk a bit more about packages next class. For now, let's just install and load:

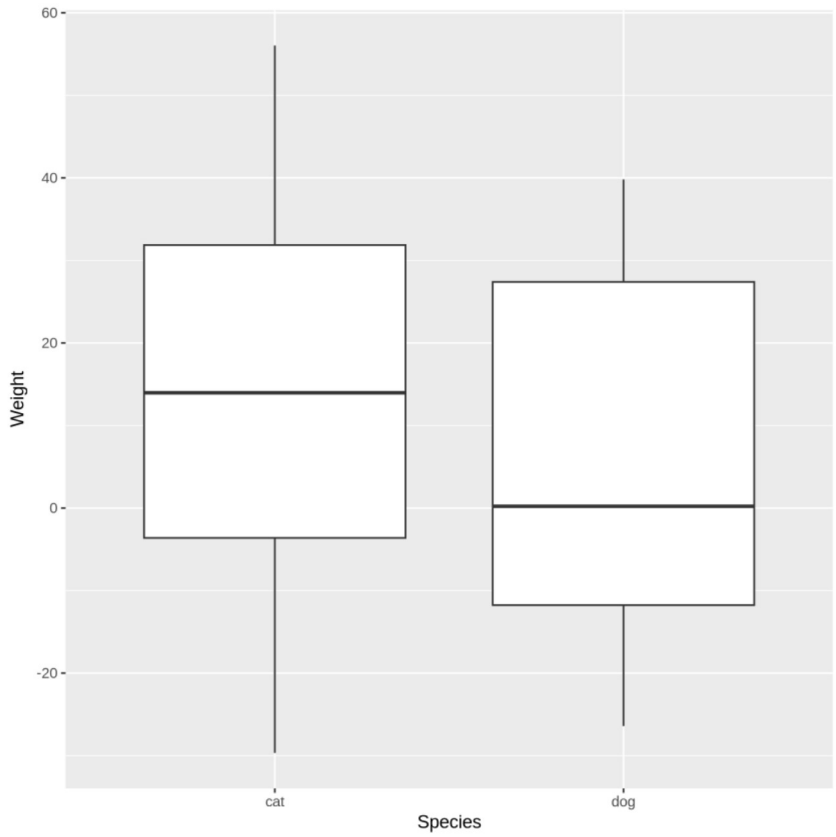
In [71]:
```
install.packages("dplyr")
install.packages("ggplot2")
```

```
Installing package into ‘/usr/local/spark-3.5.4-bin-hadoop3/R/lib’
(as ‘lib’ is unspecified)

Installing package into ‘/usr/local/spark-3.5.4-bin-hadoop3/R/lib’
(as ‘lib’ is unspecified)
```

In [72]:
```
library(ggplot2)
library(dplyr)
```

In [73]:
```
ggplot(df, aes(x = Species, y = Weight)) +
    geom_boxplot()
```

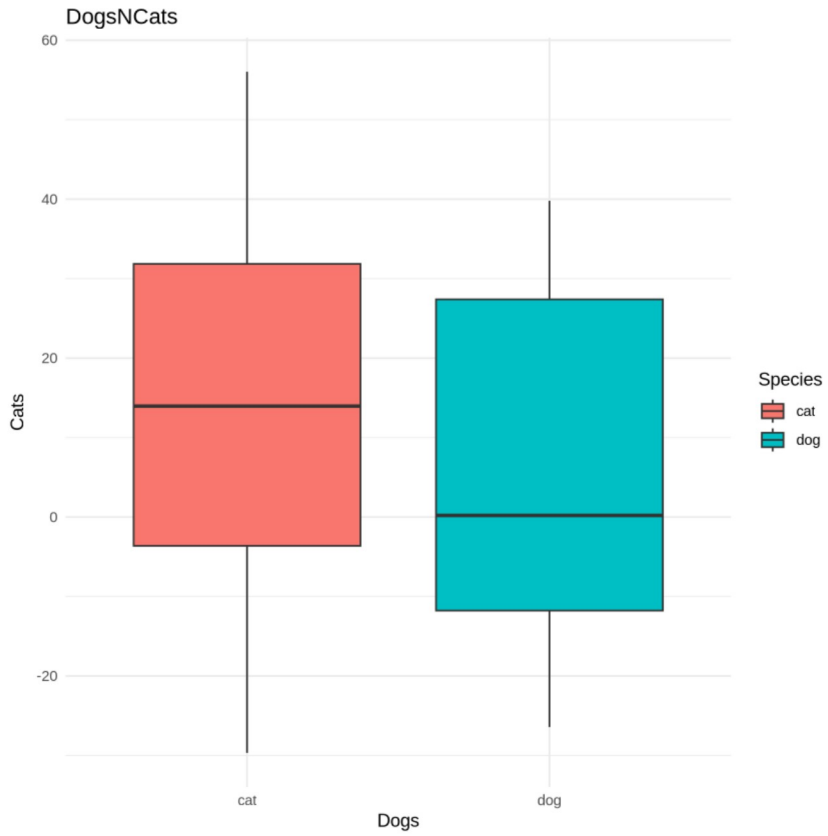


ggplot2 is a very modular way of plotting. You give commands in a sequential order, and the output of one command is passed to the command with a '+' operator.

The first block (before the + sign) specifies the data we want to use and how we want to use it. Using + signs, we can add extra options, for example the style of plot. You can

really go crazy with the customizations!

In [74]:
```
ggplot(df,
       aes(x = Species,
           y = Weight,
           fill = Species)) +
    geom_boxplot() +
    labs(title = "DogsNCats",
         x = "Dogs",
         y = "Cats") +
    theme_minimal()
```



There are several excellent resources for learning about ggplot2. Try to play around with them. For example: https://rstudio.github.io/cheatsheets/html/data-visualization.html

You have encountered some basic plotting functions in R. You have learned the tilde operator. You have learned how to search the help documentation.

Part 4 - Indexing and basic statistics

R is a statistical language, and so it has functions for running statistical operations and models.

We can calculate the average a of variable:

In [75]:
```
mean(Weight)
```

8.81807884272432

Sometimes we are interested in sorting or carving our data up, and doing stats on a part

of it. We can use indexing for this. For example, we might just want the average of each species in our study

Let's look at how to extract weights for either just the Deers or the Reindeers in our study

In [80]:
```
Weight[Species=="dog"]
```

-5.75765520022185 · 3.95047002401602 · 39.8082185969646 · -26.4170833123525 · -11.9567039799787 · -21.9113630814711 · -11.7038460414286 · 23.747068667741 · -7.149846410816 · 27.0337650644571 · 1.92267995415769 · -16.6714767887723 · -8.22552128524041 · -1.50573210122105 · 30.1059095592605 · 28.4818512992909 · 31.7578406773292 · -21.060304217081 · 36.7476675730236 · 19.9254249090842

In [81]:
```
Weight[Species=="cat"]
```

22.8614889925309 · 17.3048572575358 · 14.4568770044311 · -1.49286106090126 · -23.8546530482248 · 1.10400390040186 · -3.5971941064263 · 5.67452586854355 · -29.6741829141018 · 13.4634199724689 · 35.6594832782067 · -3.69022970742043 · 56.0340866470725 · 33.6371888175838 · 34.3768423234046 · 15.9775311516616 · -7.5425969313492 · -19.8875267410882 · 49.519185025254 · 31.2715440726483

...so that we can compare the average weight

In [82]:
```
mean(Weight[Species=="dog"])
```

5.55606819533706

In [83]:
```
mean(Weight[Species=="cat"])
```

12.0800894901116

Indexing is incredibly useful. You need to know it. But it is awkward for data frames. We can make a table instead using some basic data manipulation functions. This new table is a new data frame

In [85]:
```
meanTable <- df %>%
    group_by(Species) %>%
    summarize(MeanWeight = mean(Weight))
```

In [86]:
```
meanTable
```

A tibble: 2 × 2

Species	MeanWeight
<chr>	<dbl>
cat	12.080089
dog	5.556068

This is the exact same logic as with the plotting. First you define the data you are working on, then you sequentially operate on it. This is called "piping".

The mean is only one statistic. We can also look at other statistics. How about the median?

In [87]:
```
medianTable <- df %>%
    group_by(Species) %>%
    summarize(MedianWeight = median(Weight))
```

In [88]:
```
medianTable
```

A tibble: 2 × 2

Species	MedianWeight
<chr>	<dbl>
cat	13.9601485
dog	0.2084739

We can also look at the standard deviation.

In [89]:
```
sdTable <- df %>%
    group_by(Species) %>%
    summarize(SdWeight = sd(Weight))
```

In [90]:
```
sdTable
```

A tibble: 2 × 2

Species	SdWeight
<chr>	<dbl>
cat	23.69543
dog	21.87102

Now we are almost ready to do some statistics. We will cover statistical concepts in the next lecture. BUt as a teaser, let's look at a quick comparison, and see if these means are really different.

In [91]:
```
t.test(weight1, weight2)
```

```
        Welch Two Sample t-test

data:  weight1 and weight2
t = -0.9048, df = 37.759, p-value = 0.3713
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -21.123901   8.075859
sample estimates:
mean of x mean of y
 5.556068 12.080089
```

You have learned about indexing. You have learned about basic statistical functions. You have learned how to generate summary tables. You have run your first statistical simulation, and your first hypothesis test.