

R tutorial - I

Isheng Jason Tsai

Introduction to NGS Data and Analysis
Week 16; v2020



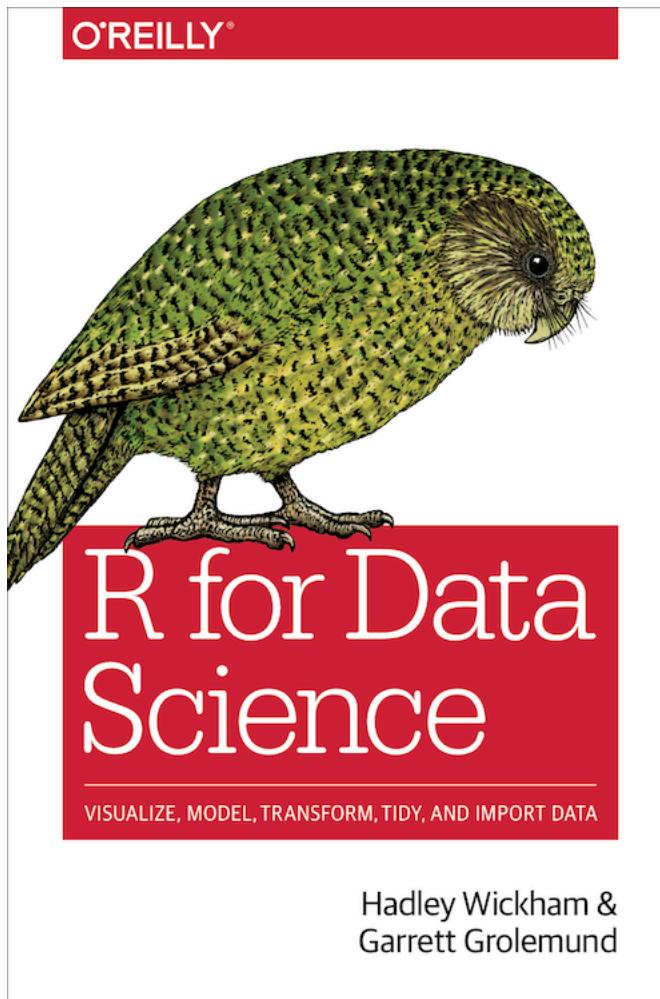
R is a programming environment



- It's **free**
 - Hence R is supported by a large user network
 - R is open source
- Can be run on Windows, Linux and Mac
- Provides an unparalleled platform for programming new statistical methods in an easy and straightforward manner.
- **Excellent graphics capabilities**
- **Lots and lots of analysis packages**
- It is also **old**, hence you need to know new functions which do things much faster

Recommended textbook

<http://r4ds.had.co.nz/>



R for Data Science

Garrett Grolemund

Hadley Wickham

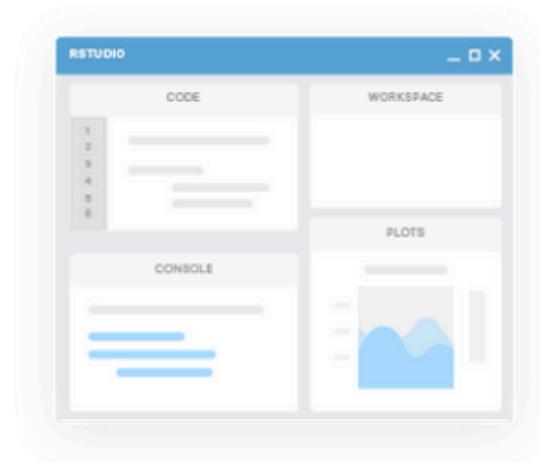
Welcome

This is the website for “**R for Data Science**”. This book will teach you how to do data science with R: You’ll learn how to get your data into R, get it into the most useful structure, transform it, visualise it and model it. In this book, you will find a practicum of skills for data science. Just as a chemist learns how to clean test tubes and stock a lab, you’ll learn how to clean data and draw plots—and many other things besides. These are the skills that allow data science to happen, and here you will find the best practices for doing each of these things with R. You’ll learn how to use the grammar of graphics, literate programming, and reproducible research to save time. You’ll also learn how to manage cognitive resources to facilitate discoveries when wrangling, visualising, and exploring data.

Download R (ver4.0.1) and Rstudio



<http://www.r-project.org>
<https://www.rstudio.com/>

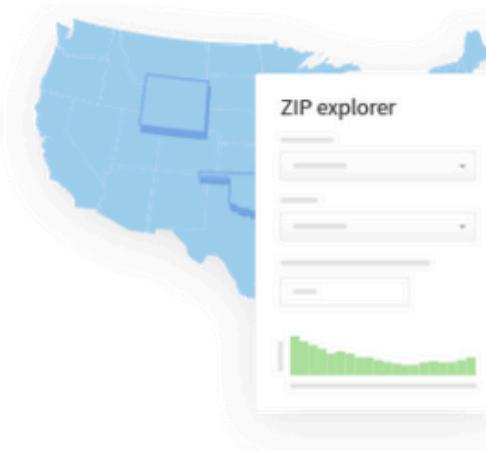


RStudio

RStudio makes R easier to use. It includes a code editor, debugging & visualization tools.

 Download

 Learn More



Shiny

Shiny helps you make interactive web applications for visualizing data. Bring R data analysis to life.

 Learn More

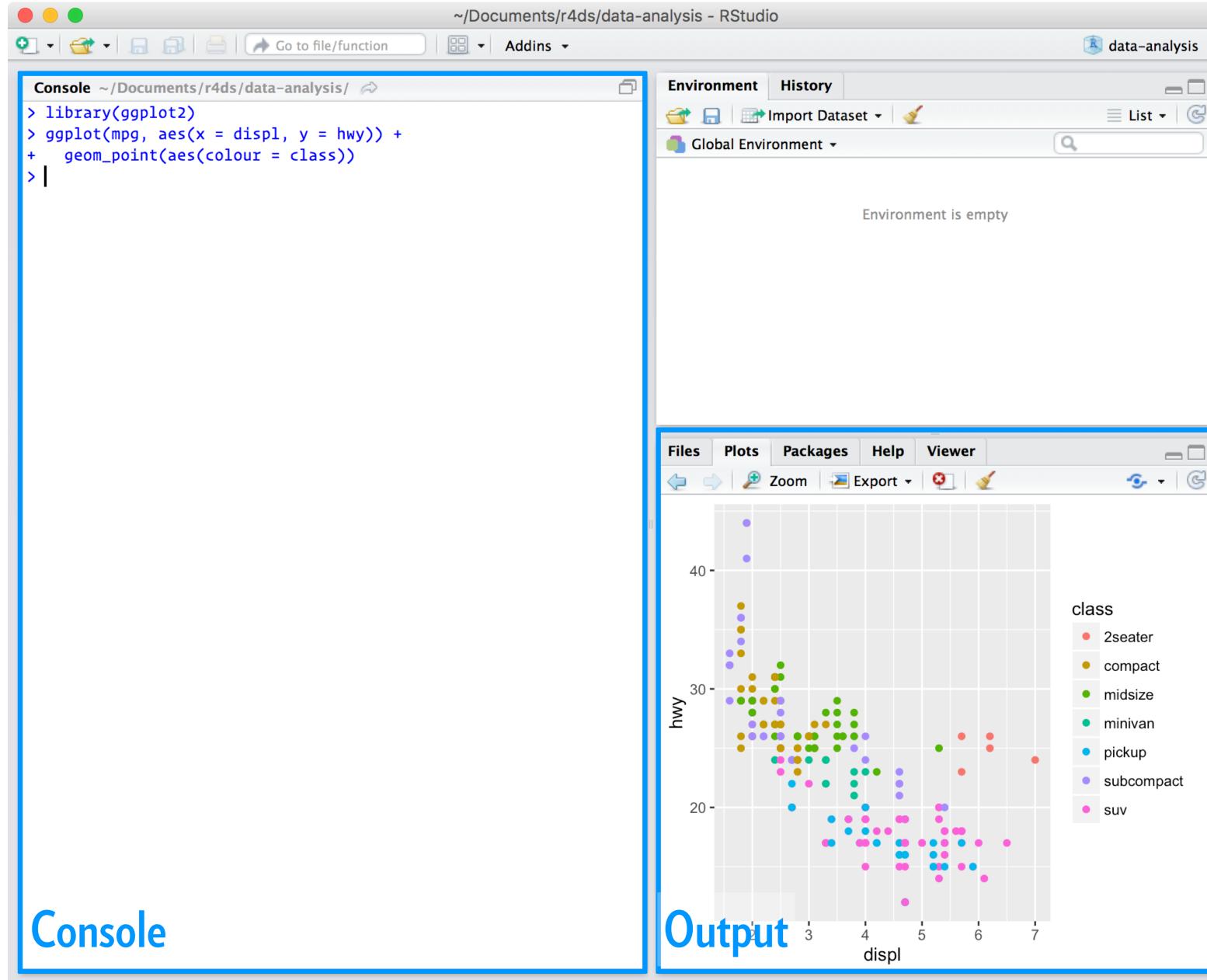


R Packages

Our developers create popular packages to expand the features of R. Includes ggplot2, dplyr, R Markdown & more.

 Learn More

Rstudio interface



Slide annotation

```
#code box  
# Comment
```

```
function(x)
```

R as a calculator

```
> 2+3  
[1] 5  
> 2*3  
[1] 6  
> 1  
[1] 1  
> 1 + 3  
[1] 4  
> 3 +  
+ 1111 -  
+ 1000  
[1] 114  
>
```

Press enter to complete the expression

Completed expression



Incomplete expression will result in continuation prompt +

Assignment

```
> x <- 5  
> x  
[1] 5  
> y <- 10  
> y  
[1] 10  
> x+y  
[1] 15  
> X <- 10  
> X  
[1] 10  
> x  
[1] 5  
> x <- 100  
> x  
[1] 100  
> z <- x + y + X  
> z  
[1] 120
```

<- is the assignment operation

R is case sensitive ; x does not equal to X

Original value is replaced

New value can be assigned as the result
of calculation

Boolean assignment

```
student <- 30000  
phd <- 56000
```

```
student > phd
```

[1] FALSE

```
student < phd
```

[1] TRUE

```
student != phd
```

[1] FALSE

```
student + student > phd
```



#Two heads are better than one

[1] TRUE

Vector is the simplest data structure in R

```
x<- c(1,2,3,4,5,6,7,8,9,10)
```

c = combine or concatenate

In this case, we assign a **vector** of 10 numbers into x

```
x * 2  
x /10 + 1
```

Different types of vectors

```
x<- c(1,2,3,4,5,6,7,8,9,10)  
strings <- c("AS","BRC")
```

```
typeof(x)  
typeof(strings)
```

This matters when one data type is numbers, and you want to sort them categorically (factors)

```
> typeof(x)  
[1] "double"  
> typeof(char)  
[1] "character"  
> typeof(strings)  
[1] "character"
```

Data frames

- Think of these like Excel spreadsheets
- **All the values of the same variable must go in the same column**
 - E.g., age, sex, RPKM, numbers
- **Rows represent samples**
 - E.g., sample A collected in Taiwan, sample B collected in Japan
- Like matrices but different types of data are allowed
- Tibble from the **dplyr** package ; basically like data frame but much easier to manipulate

R has some pre-installed data frames

The data set consists of **50 samples from each of three species of Iris (*Iris setosa*, *Iris virginica* and *Iris versicolor*)**. Four features were measured from each sample: **the length and the width of the sepals and petals, in centimetres**. Based on the combination of these four features, Fisher developed a linear discriminant model to distinguish the species from each other.

This data set became a typical test case for many statistical classification techniques in machine learning such as support vector machines

```
iris  
head(iris)
```



THE USE OF MULTIPLE MEASUREMENTS IN TAXONOMIC PROBLEMS

By R. A. FISHER, Sc.D., F.R.S.

I. DISCRIMINANT FUNCTIONS

WHEN two or more populations have been measured in several characters, x_1, \dots, x_s , special interest attaches to certain linear functions of the measurements by which the populations are best discriminated. At the author's suggestion use has already been made of this fact in craniometry (a) by Mr E. S. Martin, who has applied the principle to the sex differences in measurements of the mandible, and (b) by Miss Mildred Barnard, who showed how to obtain from a series of dated series the particular compound of cranial measurements showing most distinctly a progressive or secular trend. In the present paper the application of the same principle will be illustrated on a taxonomic problem; some questions connected with the precision of the processes employed will also be discussed.

II. ARITHMETICAL PROCEDURE

Table I shows measurements of the flowers of fifty plants each of the two species *Iris setosa* and *I. versicolor*, found growing together in the same colony and measured by Dr E. Anderson, to whom I am indebted for the use of the data. Four flower measurements are given. We shall first consider the question: What linear function of the four measurements

$$X = \lambda_1 x_1 + \lambda_2 x_2 + \lambda_3 x_3 + \lambda_4 x_4$$

will maximize the ratio of the difference between the specific means to the standard deviations within species? The observed means and their differences are shown in Table II. We may represent the differences by d_p , where $p = 1, 2, 3$ or 4 for the four measurements.

The sums of squares and products of deviations from the specific means are shown in Table III. Since fifty plants of each species were used these sums contain 98 degrees of freedom. We may represent these sums of squares or products by S_{pq} , where p and q take independently the values 1, 2, 3 and 4.

Then for any linear function, X , of the measurements, as defined above, the difference between the means of X in the two species is

$$D = \lambda_1 d_1 + \lambda_2 d_2 + \lambda_3 d_3 + \lambda_4 d_4,$$

while the variance of X within species is proportional to

$$S = \sum_{p=1}^4 \sum_{q=1}^4 \lambda_p \lambda_q S_{pq}.$$

The particular linear function which best discriminates the two species will be one for

Selection in data frames

Square brackets

- `dat[i ,]` would select the i -th row (which is a **vector**)
- `dat[, j]` would select the j -th column (which is a **vector**)
- `dat[i, j]` would select the value from the i -th row and j -th column

```
iris[,1]  
iris[1,]  
iris[1,1]
```

dollar (\$) operation (for columns only)

```
iris$Petal.Width
```

Function

```
function (arg1, arg2, arg3... , option1=,option2=...)
```

```
x<- c(1,2,3,4,5,6,7,8,9,10)  
y<- c(3,6,9,10,13,30,20,100)
```

```
mean(x)  
mean(y)  
median(x)  
max(x)
```

```
> x<- c(1,2,2,3,5,6,7,10)  
> y<- c(3,6,9,10,13,30,20,100)  
> mean(x)  
[1] 4.5  
> mean(y)  
[1] 23.875  
> median(x)  
[1] 4  
> median(y)  
[1] 11.5  
> max(x)  
[1] 10  
> min(y)  
[1] 3  
... .
```

- Must have **assigned names**
- Applies using **round brackets**
- Takes **argument** and options

Useful base functions here

```
y<- abs(-20)
x<- Sum(y+5)
Z<- Log(x)
round(x,1)
summary(iris)
head(iris)
tail(iris)
ncol(iris)
nrow(iris)
```

R base function **plot**

```
x<- c(1,2,3,4,5,6,7,8,9,10)  
y<- c(3,6,9,10,13,30,20,100,220,100)
```

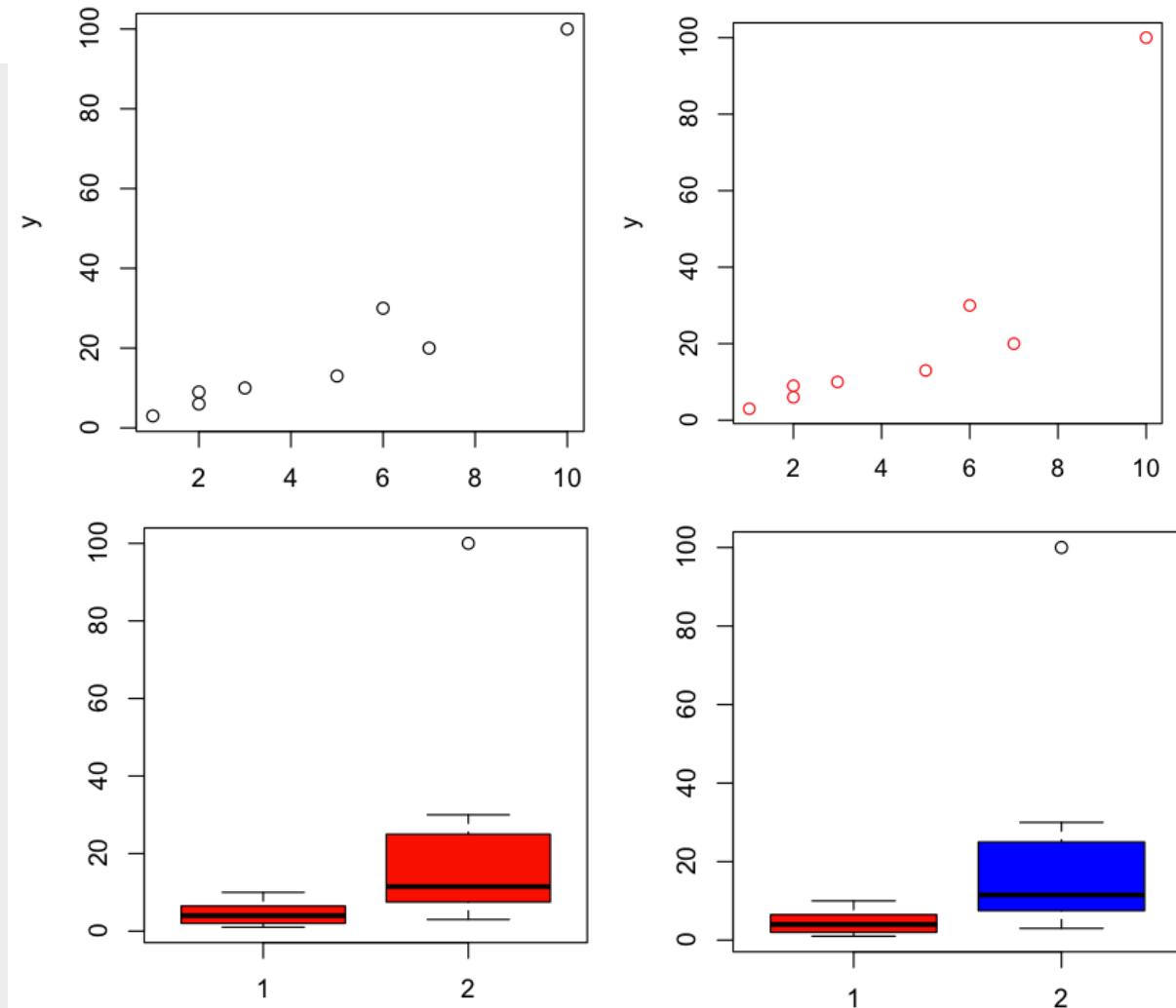
```
plot(x,y)  
plot(x,y,col="red")
```

```
# many functions have additional parameters
```

```
boxplot(x,y,col="red")  
boxplot(x,y,col=c("hotpink", "yellow"))
```

```
boxplot(x,y,col=c("hotpink", "yellow"),main="Lec2")
```

```
# can use the ? sign to find out more about function  
?boxplot
```



Running out of functions to use?

Use Packages

- R consists of a **core** and **additional packages**.
- Collections of R functions, data, and compiled code
- Well-defined format that ensures easy installation, a basic standard of documentation, and enhances portability and reliability

Install R packages

- An **R package** is a collection of functions, data, and documentation that extends the capabilities of base R.
- Using packages is key to the successful use of R.
- Understanding **tidyverse** is highly recommended.

```
# install  
install.packages("tidyverse")  
  
#after it's installed, you can initiate by using library  
library(tidyverse)
```

Tidyverse package (much better/easier functions)

Tidyverse

Packages Articles Learn Help Contribute

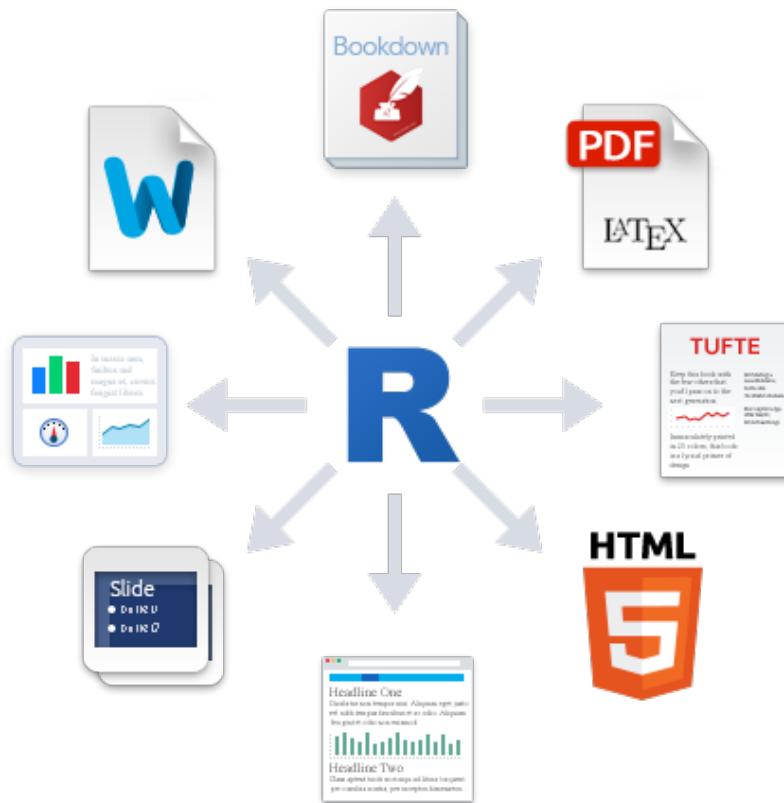
R packages for data science

The tidyverse is an opinionated [collection of R packages](#) designed for data science. All packages share an underlying design philosophy, grammar, and data structures.

Install the complete tidyverse with:

```
install.packages("tidyverse")
```

Communicate: R markdown in Rstudio



The screenshot shows the RStudio interface with the following components:

- Left Panel (Code Editor):** The file is named "chunks.Rmd". The code editor contains R code chunks:

```
1 R Code Chunks
2 =====
3
4 With R Markdown, you can insert R code
5 chunks including plots:
6
7 ```{r qplot, fig.width=4, fig.height=3,
8 message=FALSE}
9 # quick summary and plot
10 library(ggplot2)
11 summary(cars)
12 qplot(speed, dist, data=cars) +
13   geom_smooth()

```

- Right Panel (Preview):** The preview window shows the rendered content:

R Code Chunks

With R Markdown, you can insert R code chunks including plots:

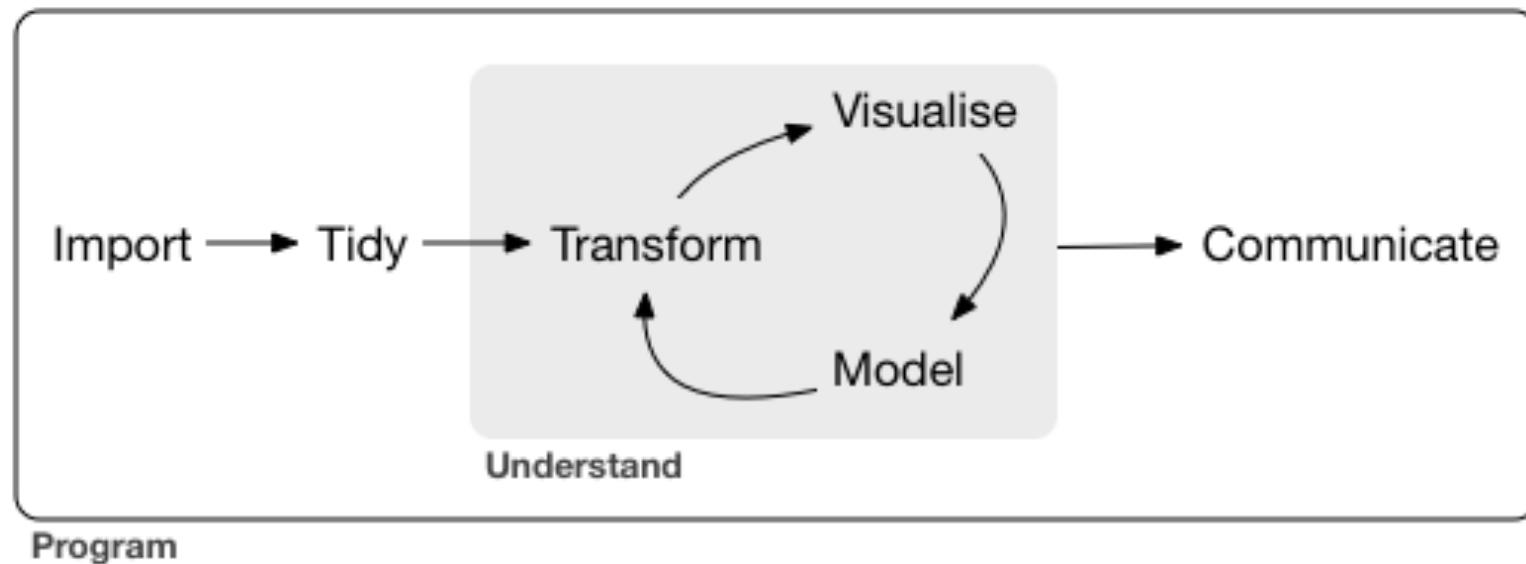
```
# quick summary and plot
library(ggplot2)
summary(cars)
```

```
##      speed         dist
## Min.   : 4.0   Min.   : 2
## 1st Qu.:12.0   1st Qu.: 26
## Median :15.0   Median : 36
## Mean   :15.4   Mean   : 43
## 3rd Qu.:19.0   3rd Qu.: 56
## Max.   :25.0   Max.   :120
```

```
qplot(speed, dist, data = cars) + geom_smooth()
```

A scatter plot with a smoothing line is displayed below the code preview.

Tidyverse package (much better/easier functions)



dplyr package

- Pick observations by their values (`filter()`).
- Reorder the rows (`arrange()`).
- Pick variables by their names (`select()`).
- Create new variables with functions of existing variables (`mutate()`).
- Collapse many values down to a single summary (`summarise()`).

filter

`filter()` allows you to subset observations based on their values. The first argument is the name of the data frame. The second and subsequent arguments are the expressions that filter the data frame. For example, we can select all flights on January 1st with:

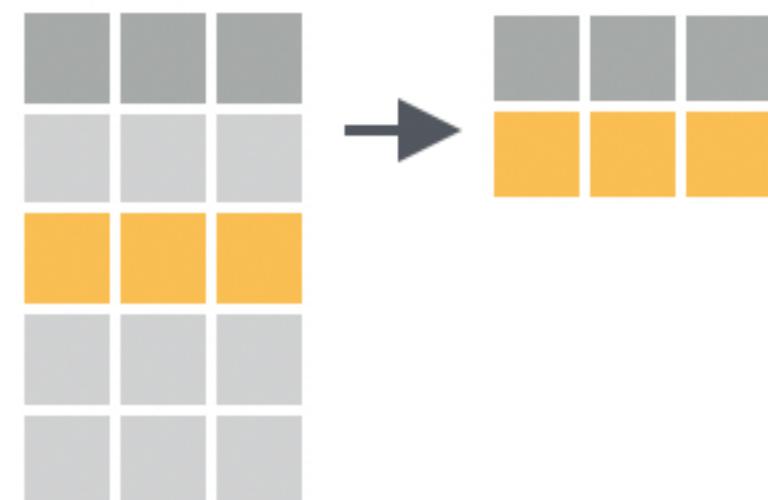
```
filter(flights, month == 1, day == 1)
```

```
jan1 <- filter(flights, month == 1, day == 1)
```

```
filter(flights, month = 1)
```

```
filter(flights, month == 11 | month == 12)
```

```
nov_dec <- filter(flights, month %in% c(11, 12))
```

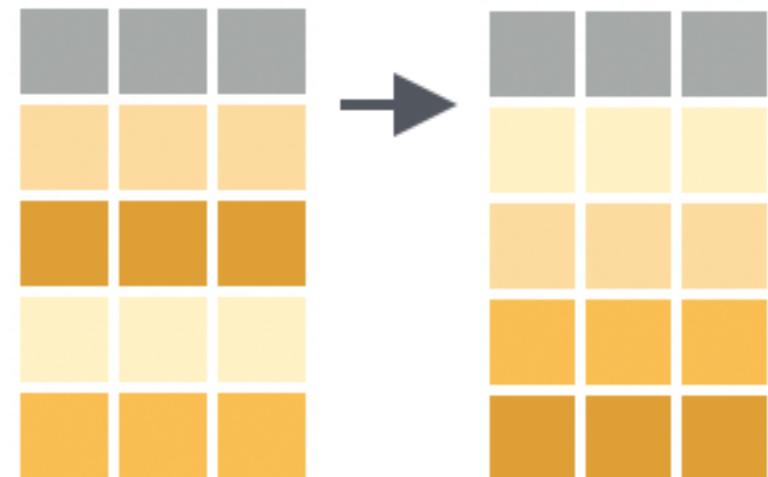


arrange

`arrange()` works similarly to `filter()` except that instead of selecting rows, it changes their order. It takes a data frame and a set of column names (or more complicated expressions) to order by. If you provide more than one column name, each additional column will be used to break ties in the values of preceding columns:

```
arrange(flights, desc(dep_delay))
```

```
arrange(flights, dep_delay)
```



select

Besides selecting sets of existing columns, it's often useful to add new columns that are functions of existing columns. That's the job of `mutate()`.

```
select(flights, year, month, day)  
# Select all columns between year and day (inclusive)  
select(flights, year:day)  
  
select(flights, -(year:day))
```



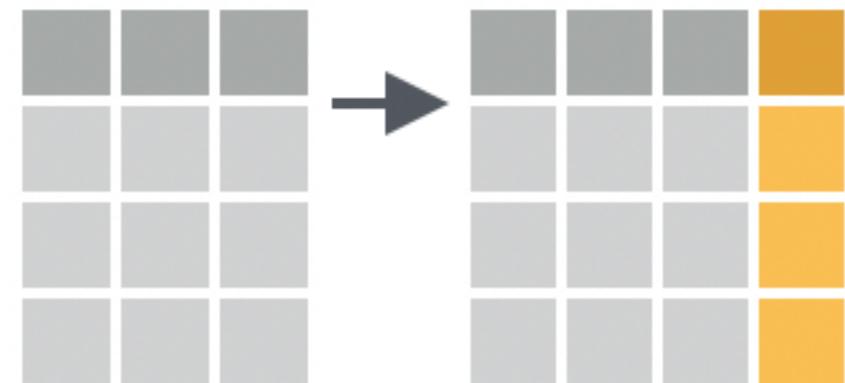
mutate

```
select(flights, year, month, day)
```

```
# Select all columns between year and day (inclusive)
```

```
select(flights, year:day)
```

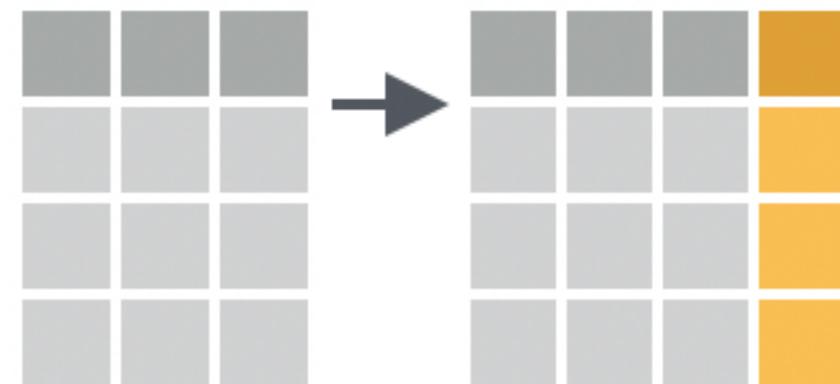
```
select(flights, -(year:day))
```



mutate()

Besides selecting sets of existing columns, it's often useful to add new columns that are functions of existing columns. That's the job of `mutate()`.

```
flights_sml <- select(flights,  
  year:day,  
  ends_with("delay"),  
  distance,  
  air_time  
)  
  
mutate(flights_sml,  
  gain = dep_delay - arr_delay,  
  speed = distance / air_time * 60  
)
```



transmute()

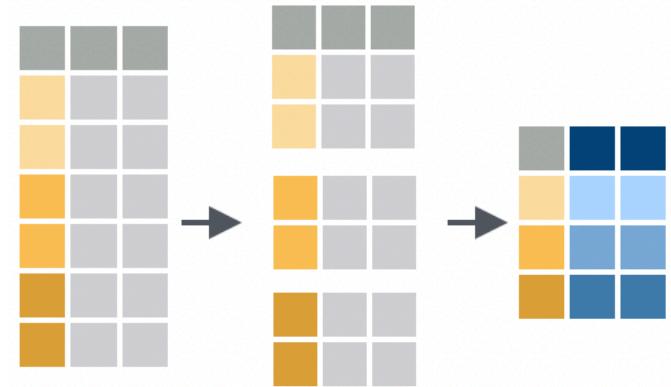
If you only want to keep the new variables, use `transmute()` :

```
transmute(flights,  
          gain = dep_delay - arr_delay,  
          hours = air_time / 60,  
          gain_per_hour = gain / hours )
```

summarise() and group_by

```
by_day <- group_by(flights, year, month, day)
summarise(by_day, delay = mean(dep_delay, na.rm = TRUE))
```

```
by_dest <- group_by(flights, dest)
delay <- summarise(by_dest,
  count = n(),
  dist = mean(distance, na.rm = TRUE),
  delay = mean(arr_delay, na.rm = TRUE) )
delay <- filter(delay, count > 20, dest != "HNL")
```

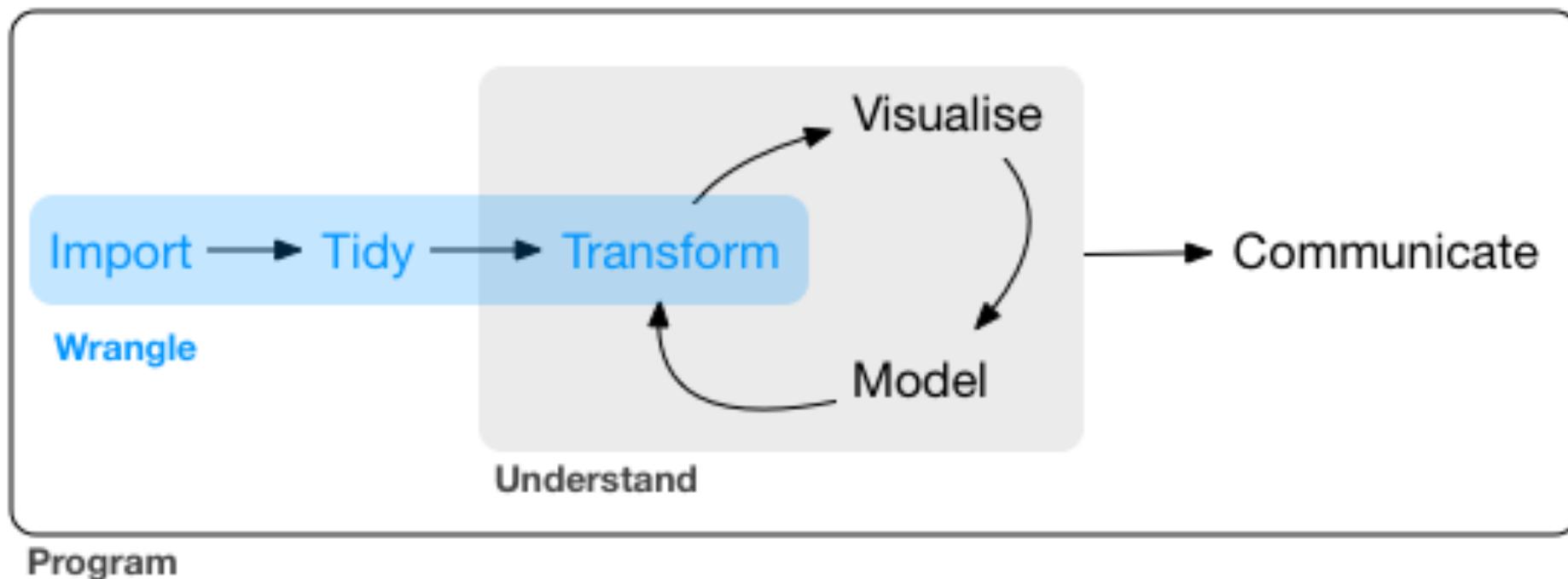


Combine functions into a pipe

```
delays <- flights %>%
  group_by(dest) %>%
  summarise( count = n(),
             dist = mean(distance, na.rm = TRUE),
             delay = mean(arr_delay, na.rm = TRUE) ) %>%
  filter(count > 20, dest != "HNL")
```

Data wrangling

Data wrangling is very important: without it you can't work with your own data!



Tibble

- work with “**tibbles**” instead of R’s traditional `data.frame`. Tibbles are data frames, but they tweak some older behaviours to make life a little easier.
- R is an old language, and some things that were useful 10 or 20 years ago now get in your way. It’s difficult to change base R without breaking existing code, so most innovation occurs in packages.

```
as_tibble(iris)
```

```
#> # A tibble: 150 x 5
#>   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
#>       <dbl>     <dbl>      <dbl>      <dbl> <fct>
#> 1     5.1       3.5        1.4       0.2  setosa
#> 2     4.9       3          1.4       0.2  setosa
#> 3     4.7       3.2        1.3       0.2  setosa
#> 4     4.6       3.1        1.5       0.2  setosa
#> 5     5          3.6        1.4       0.2  setosa
#> 6     5.4       3.9        1.7       0.4  setosa
#> # ... with 144 more rows
```

Tibble vs. data.frame

- Tibbles have a refined print method that shows only the first 10 rows, and all the columns that fit on screen. This makes it much easier to work with large data. In addition to its name, each column reports its type
- So far all the tools you've learned have worked with complete data frames. If you want to pull out a single variable, you need some new tools, `$` and `[[`. `[[` can extract by name or position; `$` only extracts by name but is a little less typing.

Tidy data (perhaps the most important part of tutorial)

tidyverse package

“Happy families are all alike; every unhappy family is unhappy in its own way.” — Leo Tolstoy

“Tidy datasets are all alike, but every messy dataset is messy in its own way.” — Hadley Wickham

Dataset example; which is preferred?

```
#> # A tibble: 6 x 4
#>   country     year  cases population
#>   <chr>      <int> <int>      <int>
#> 1 Afghanistan 1999    745 19987071
#> 2 Afghanistan 2000   2666 20595360
#> 3 Brazil       1999  37737 172006362
#> 4 Brazil       2000  80488 174504898
#> 5 China        1999 212258 1272915272
#> 6 China        2000 213766 1280428583
```

```
#> # A tibble: 12 x 4
#>   country     year type      count
#>   <chr>      <int> <chr>      <int>
#> 1 Afghanistan 1999 cases      745
#> 2 Afghanistan 1999 population 19987071
#> 3 Afghanistan 2000 cases     2666
#> 4 Afghanistan 2000 population 20595360
#> 5 Brazil       1999 cases     37737
#> 6 Brazil       1999 population 172006362
#> # ... with 6 more rows
```

table1
table2
table3

#spread across two tables
table4a
table4b

Definition of tidy dataset

1. Each variable must have its own column.
2. Each observation must have its own row.
3. Each value must have its own cell.

country	year	cases	population
Afghanistan	1990	745	18657071
Afghanistan	2000	2666	20595360
Brazil	1999	37737	172006362
Brazil	2000	80488	174504898
China	1999	212258	1272915272
China	2000	21366	1280428583

variables

country	year	cases	population
Afghanistan	1990	745	18657071
Afghanistan	2000	2666	20595360
Brazil	1999	37737	172006362
Brazil	2000	80488	174504898
China	1999	212258	1272915272
China	2000	21366	1280428583

observations

country	year	cases	population
Afghanistan	1990	745	18657071
Afghanistan	2000	2666	20595360
Brazil	1999	37737	172006362
Brazil	2000	80488	174504898
China	1999	212258	1272915272
China	2000	21366	1280428583

values

table1 is tidy. It's the only representation where each column is a variable.

Advantage of tidy dataset

1. There's a general advantage to picking one consistent way of storing data. If you have a consistent data structure, it's easier to learn the tools that work with it because they have an underlying uniformity.
2. There's a specific advantage to placing variables in columns because it allows R's vectorised nature to shine. As you learned in [mutate](#) and [summary functions](#), most built-in R functions work with vectors of values. That makes transforming tidy data feel particularly natural.

Most data that you will encounter will be untidy.

There are two main reasons:

- Most people aren't familiar with the principles of tidy data, and it's hard to derive them yourself unless you spend a *lot* of time working with data.
- Data is often organised to facilitate some use other than analysis. For example, data is often organised to make entry as easy as possible.

This means for most real analyses, you'll need to do some tidying.

- first step is always to figure out what the variables and observations are. Sometimes this is easy; other times you'll need to consult with the people who originally generated the data.
- second step is to resolve one of two common problems:
 - One variable might be spread across multiple columns.
 - One observation might be scattered across multiple rows.

`pivot_longer()` and `pivot_wider()`

A common problem is a dataset where some of the column names are not names of variables, but *values* of a variable. Take `table4a`: the column names `1999` and `2000` represent values of the `year` variable, the values in the `1999` and `2000` columns represent values of the `cases` variable, and each row represents two observations, not one.

```
table4a  
#> # A tibble: 3 x 3  
#>   country     `1999` `2000`  
#> * <chr>       <int>  <int>  
#> 1 Afghanistan    745   2666  
#> 2 Brazil        37737  80488  
#> 3 China         212258 213766
```

table4a

pivot_longer() and pivot_wider()

To tidy a dataset like this, we need to **pivot** the offending columns into a new pair of variables. To describe that operation we need three parameters:

- The set of columns whose names are values, not variables. In this example, those are the columns `1999` and `2000`.
- The name of the variable to move the column names to. Here it is `year`.
- The name of the variable to move the column values to. Here it's `cases`.

Pivoting `table4` into a longer, tidy form.

country	year	cases	country	1999	2000
Afghanistan	1999	745	Afghanistan	745	2666
Afghanistan	2000	2666	Brazil	37737	80488
Brazil	1999	37737	China	212258	213766
Brazil	2000	80488			
China	1999	212258			
China	2000	213766			

table4

The diagram illustrates the transformation of the wide-form `table4` into a long-form dataset. It shows two tables side-by-side. The left table, `table4`, has columns `country`, `year`, and `cases`. The right table, resulting from the pivot, has columns `country`, `1999`, and `2000`. Arrows indicate the mapping of rows: each row in `table4` corresponds to a row in the pivot table. The `cases` value in `table4` maps to the `cases` column in the pivot table. The `year` value in `table4` maps to the `year` column in the pivot table. The country names in `table4` map to the `country` column in the pivot table.

```
table4a %>%
  pivot_longer(c(`1999`, `2000`), names_to = "year", values_to = "cases")
```

pivot_longer() and pivot_wider()

`pivot_wider()` is the opposite of `pivot_longer()`. You use it when an observation is scattered across multiple rows. For example, take `table2`: an observation is a country in a year, but each observation is spread across two rows.

```
#> # A tibble: 12 x 4
#>   country     year type     count
#>   <chr>       <int> <chr>    <int>
#> 1 Afghanistan 1999 cases     745
#> 2 Afghanistan 1999 population 19987071
#> 3 Afghanistan 2000 cases     2666
#> 4 Afghanistan 2000 population 20595360
#> 5 Brazil       1999 cases     37737
#> 6 Brazil       1999 population 172006362
#> # ... with 6 more rows
```

table2

pivot_longer() and pivot_wider()

To tidy this up, we first analyse the representation in similar way to `pivot_longer()`. This time, however, we only need two parameters:

- The column to take variable names from. Here, it's `type`.
- The column to take values from. Here it's `count`.

```
table2 %>%  
  pivot_wider(names_from = type, values_from = count)
```

The diagram illustrates the transformation of a long-format data frame into a wide-format data frame using the `pivot_wider` function. On the left, a long-format data frame `table2` is shown with columns: `country`, `year`, `key`, and `value`. The data consists of rows for Afghanistan, Brazil, and China across the years 1999 and 2000, with values for both `cases` and `population`. On the right, a wide-format data frame is shown with columns: `country`, `year`, `cases`, and `population`. The data is grouped by country and year, with specific values for `cases` and `population` for each group. Arrows point from the `value` column of the long frame to the corresponding `cases` and `population` columns of the wide frame, indicating the mapping of the variable names and their values.

country	year	key	value
Afghanistan	1999	cases	745
Afghanistan	1999	population	19987071
Afghanistan	2000	cases	2666
Afghanistan	2000	population	20595360
Brazil	1999	cases	37737
Brazil	1999	population	172006362
Brazil	2000	cases	80488
Brazil	2000	population	174504898
China	1999	cases	212258
China	1999	population	1272915272
China	2000	cases	213766
China	2000	population	1280428583

country	year	cases	population
Afghanistan	1999	745	19987071
Afghanistan	2000	2666	20595360
Brazil	1999	37737	172006362
Brazil	2000	80488	174504898
China	1999	212258	1272915272
China	2000	213766	1280428583

table2

More functions in `tidyverse`

table3 %>%

```
separate(rate, into = c("cases", "population"), sep = "/")
```

country	year	rate
Afghanistan	1999	745 / 19987071
Afghanistan	2000	2666 / 20595360
Brazil	1999	37737 / 172006362
Brazil	2000	80488 / 174504898
China	1999	212258 / 1272915272
China	2000	213766 / 1280428583

table3

country	year	cases	population
Afghanistan	1999	745	19987071
Afghanistan	2000	2666	20595360
Brazil	1999	37737	172006362
Brazil	2000	80488	174504898
China	1999	212258	1272915272
China	2000	213766	1280428583

table5 %>%

```
unite(new, century, year, sep = "")
```

country	year	rate
Afghanistan	1999	745 / 19987071
Afghanistan	2000	2666 / 20595360
Brazil	1999	37737 / 172006362
Brazil	2000	80488 / 174504898
China	1999	212258 / 1272915272
China	2000	213766 / 1280428583

table5

country	century	year	rate
Afghanistan	19	99	745 / 19987071
Afghanistan	20	0	2666 / 20595360
Brazil	19	99	37737 / 172006362
Brazil	20	0	80488 / 174504898
China	19	99	212258 / 1272915272
China	20	0	213766 / 1280428583

table6

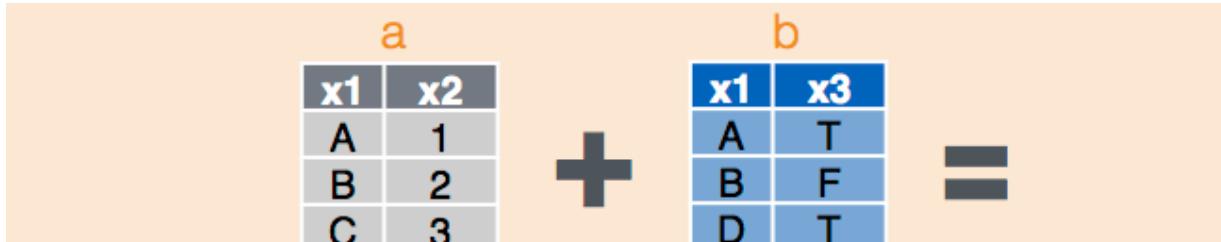
Relational data

- It's rare that a data analysis involves only a single table of data. Typically you have many tables of data, and you must combine them to answer the questions that you're interested in. Collectively, multiple tables of data are called **relational data** because it is the relations, not just the individual datasets, that are important.

To work with relational data you need verbs that work with pairs of tables. There are three families of verbs designed to work with relational data:

- **Mutating joins**, which add new variables to one data frame from matching observations in another.
- **Filtering joins**, which filter observations from one data frame based on whether or not they match an observation in the other table.
- **Set operations**, which treat observations as if they were set elements.

Mutating joins



The diagram illustrates a mutating join operation. It shows two tables, **a** and **b**, represented as grids of values. A plus sign (+) between them indicates the operation, and an equals sign (=) to the right indicates the result.

Mutating Joins

x1	x2	x3
A	1	T
B	2	F
C	3	NA

x1	x3
A	T
B	F
D	T

dplyr::left_join(a, b, by = "x1")
Join matching rows from b to a.

x1	x3	x2
A	T	1
B	F	2
D	T	NA

dplyr::right_join(a, b, by = "x1")
Join matching rows from a to b.

x1	x2	x3
A	1	T
B	2	F

dplyr::inner_join(a, b, by = "x1")
Join data. Retain only rows in both sets.

x1	x2	x3
A	1	T
B	2	F
C	3	NA
D	NA	T

dplyr::full_join(a, b, by = "x1")
Join data. Retain all values, all rows.

<https://rstudio.com/wp-content/uploads/2015/02/data-wrangling-cheatsheet.pdf>

Filtering joins; semi_join and anti_join

Filtering joins keep cases from the left data table (i.e. the X-data) and use the right data (i.e. the Y-data) as filter.

semi_join

ID	X1
1	a1
2	a2

ID	X2
2	b1
3	b2

ID	X1
2	a2

anti_join

ID	X1
1	a1
2	a2

ID	X2
2	b1
3	b2

ID	X1
1	a1

```
semi_join(data1, data2, by = "ID")
anti_join(data1, data2, by = "ID")
```

<https://statisticsglobe.com/r-dplyr-join-inner-left-right-full-semi-anti>

Visualisation

“The simple graph has brought more information to the data analyst’s mind than any other device.”
— John Tukey

Examples I - amplicon

Q: Is the relative abundance of OTU_n associated with abiotic factors?

OTU	Site1	Site2	Site3
OTU2	4	10	3
OTU4	8	15	1

Site	pH	metadata2	metadata3
Site1	3	1	4
Site2	5	2	5
Site3	7	2	6

Q: Is the relative abundance of OTU_n associated with particular trait?

OTU	Species	Trait2	Trait3
OTU2	Bacteria	Pathogen	44
OTU4	Fungus	free-living	32

Examples II – genome features

Program1 output

Chromosome	Region start	Region end	Gene density
Chr1	1	1000	30
Chr1	1001	2000	40
Chr1	2001	3000	3

#Ideal table?

Program2 output

Chromosome	Region start	Region end	TE density
Chr1	1	1000	3
Chr1	1001	2000	1
Chr1	2001	3000	60

Chromosome	Midpoint	Gene density	TE density	GC content
Chr1	500	30	3	30
Chr1	1500	40	1	35
Chr1	2500	3	60	40

Program3 output

Chromosome	Region start	Region end	GC content
Chr1	1	1000	30
Chr1	1001	2000	35
Chr1	2001	3000	40

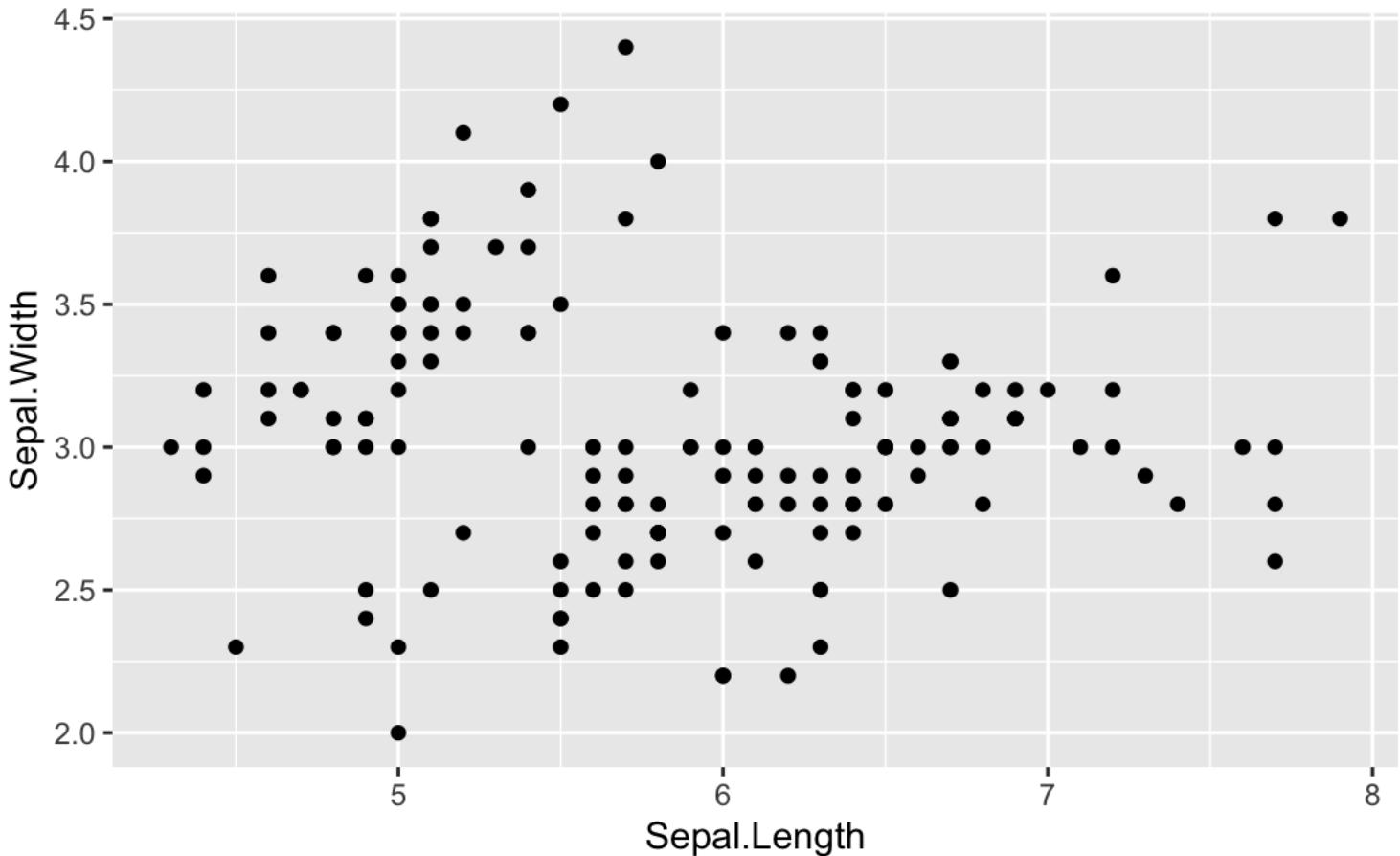
```
# A tibble: 150 x 5
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
          <dbl>       <dbl>        <dbl>       <dbl>   <fct>
1          5.1         3.5         1.4        0.2 setosa
2          4.9         3.0         1.4        0.2 setosa
3          4.7         3.2         1.3        0.2 setosa
4          4.6         3.1         1.5        0.2 setosa
5          5.0         3.6         1.4        0.2 setosa
6          5.4         3.9         1.7        0.4 setosa
7          4.6         3.4         1.4        0.3 setosa
8          5.0         3.4         1.5        0.2 setosa
9          4.4         2.9         1.4        0.2 setosa
10         4.9         3.1         1.5        0.1 setosa
# ... with 140 more rows
```

ggplot

```
ggplot(tb, aes(x = Sepal.Length, y = Sepal.Width)) +  
  geom_point()
```

aes : Aesthetic mappings describe how variables in the data are mapped to visual properties (aesthetics) of geoms

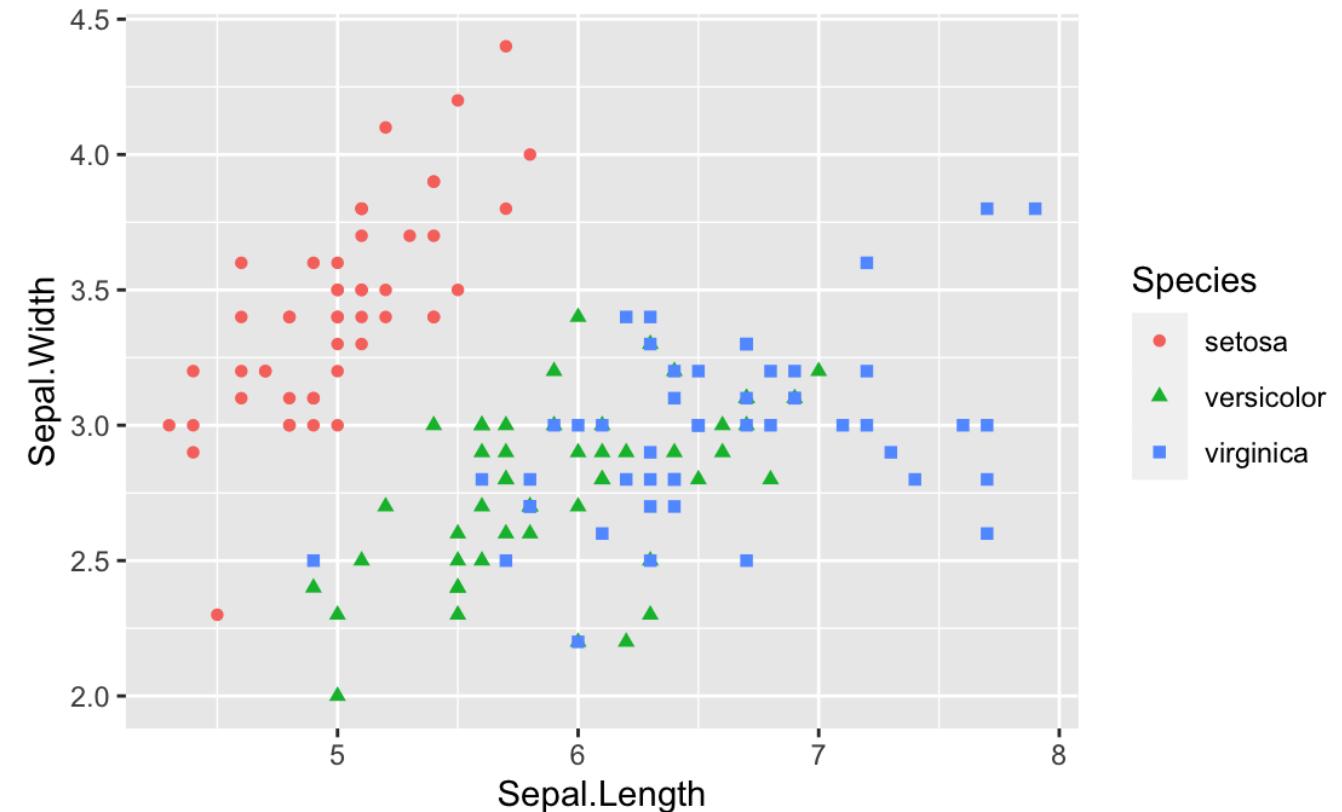
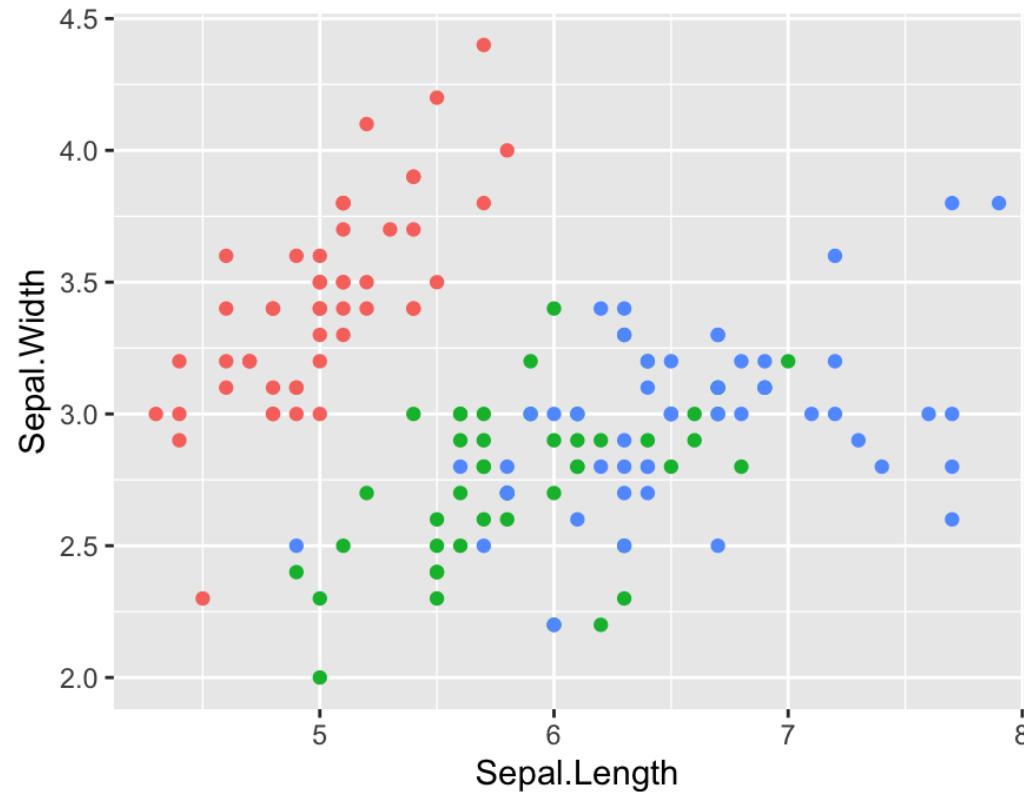
```
ggplot(tb) +  
  geom_point(aes(x = Sepal.Length, y = Sepal.Width))
```



ggplot

```
ggplot(tb, aes(x = Sepal.Length, y = Sepal.Width)) +  
  geom_point( aes(color=Species))
```

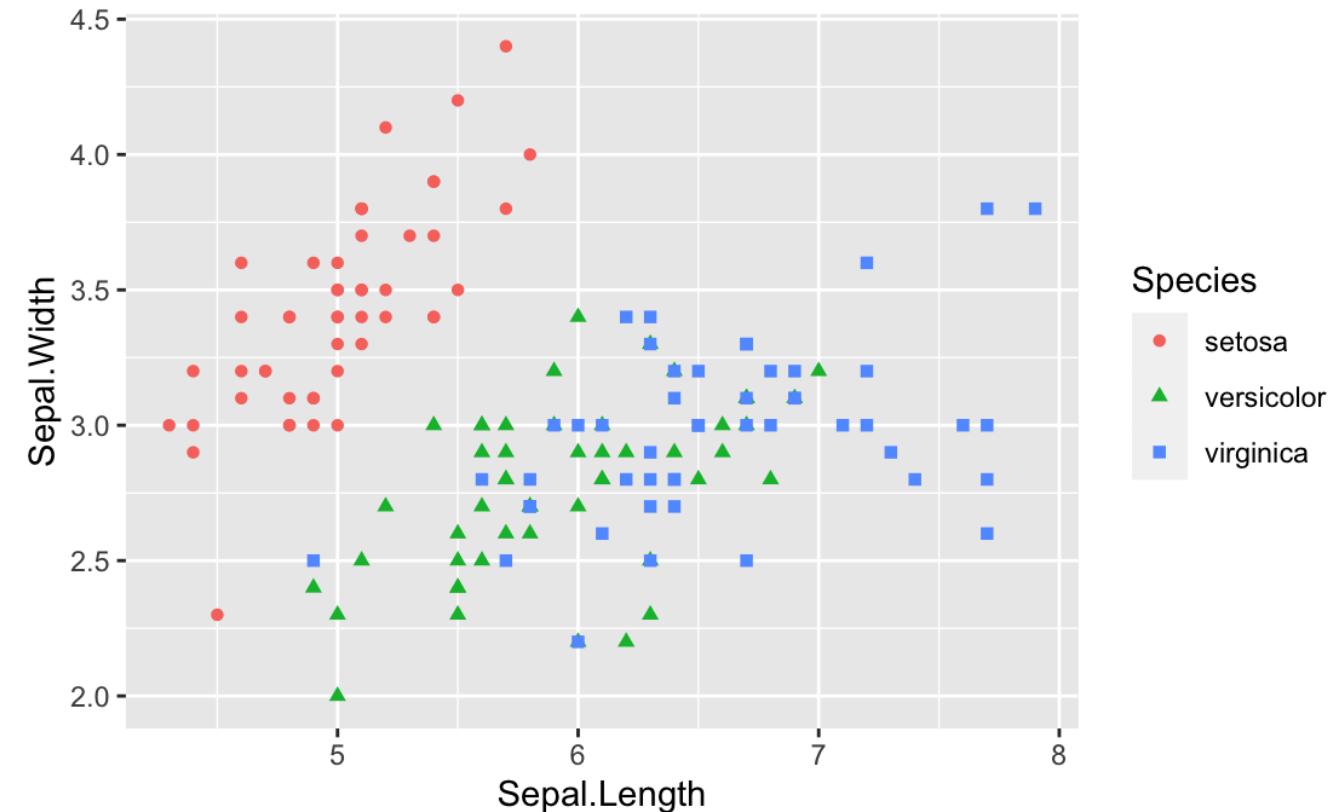
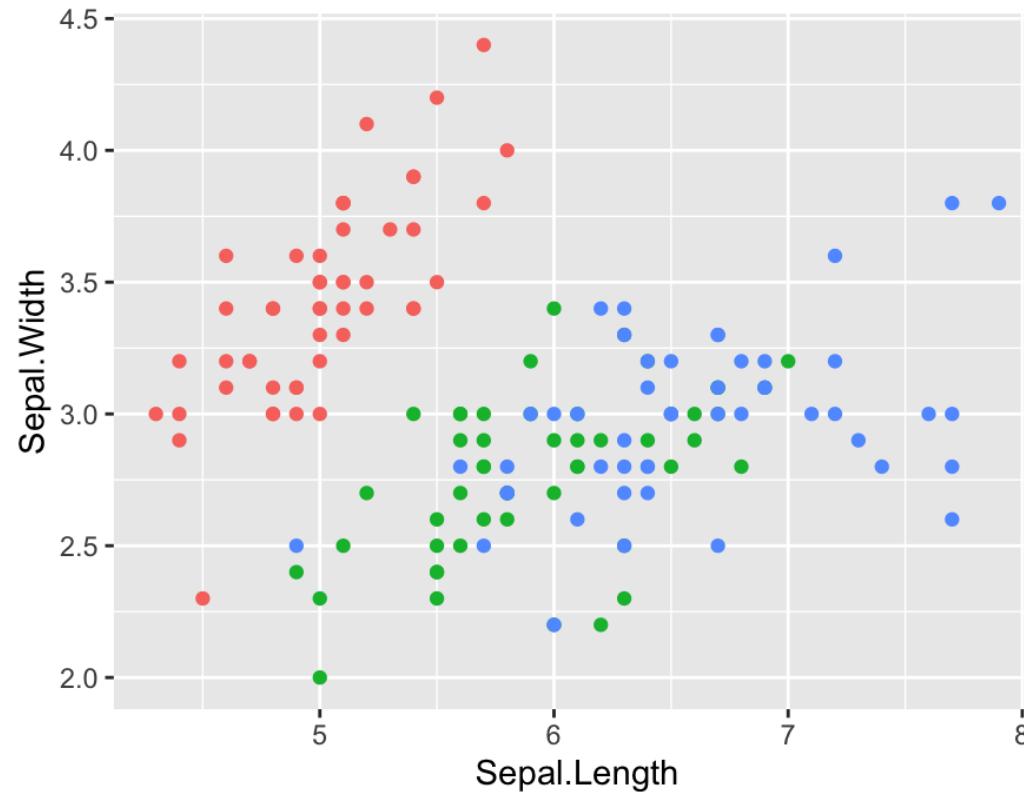
```
ggplot(tb, aes(x = Sepal.Length, y = Sepal.Width)) +  
  geom_point( aes(color=Species, shape=Species))
```



ggplot

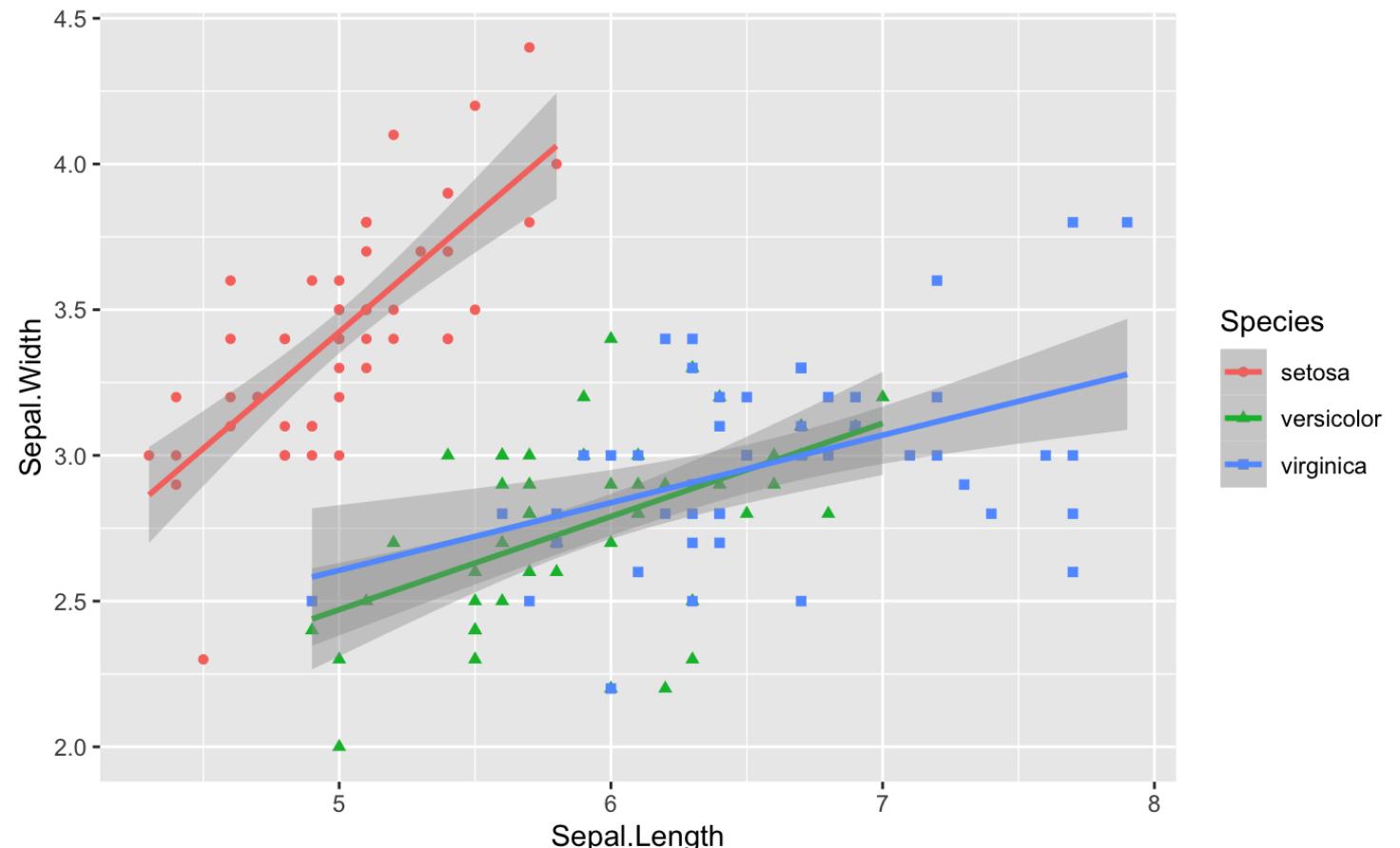
```
ggplot(tb, aes(x = Sepal.Length, y = Sepal.Width)) +  
  geom_point( aes(color=Species))
```

```
ggplot(tb, aes(x = Sepal.Length, y = Sepal.Width)) +  
  geom_point( aes(color=Species, shape=Species))
```



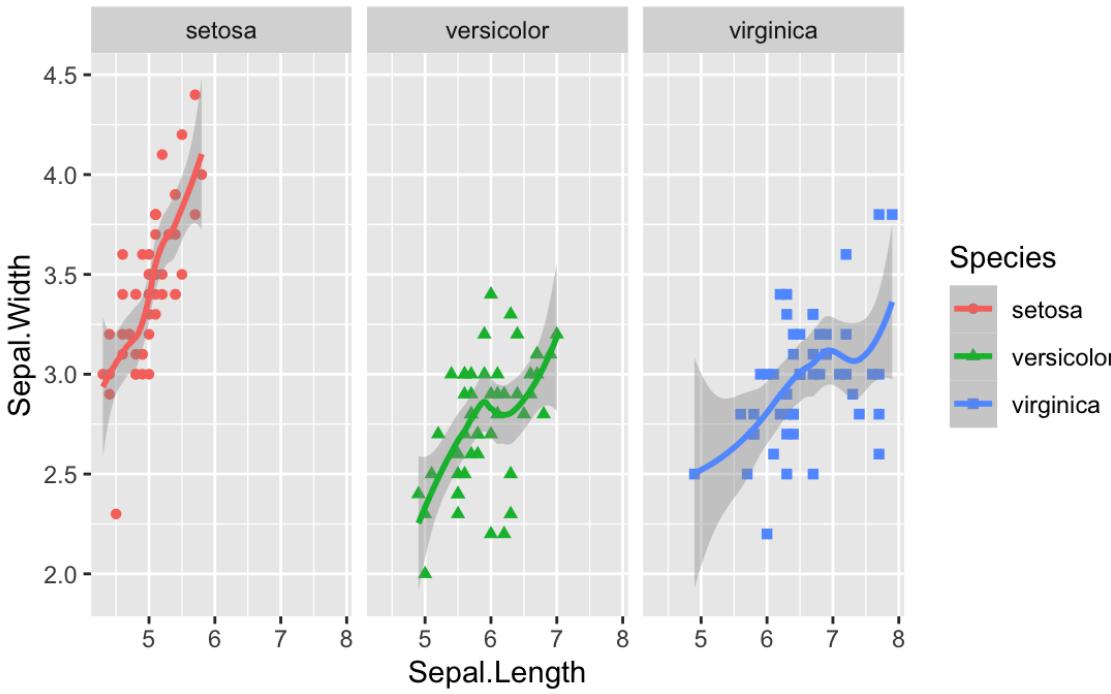
ggplot

```
p <- ggplot(tb, aes(x = Sepal.Length, y = Sepal.Width, color=Species)) +  
  geom_point( aes(shape=Species))  
  
p + geom_smooth(method="lm")
```

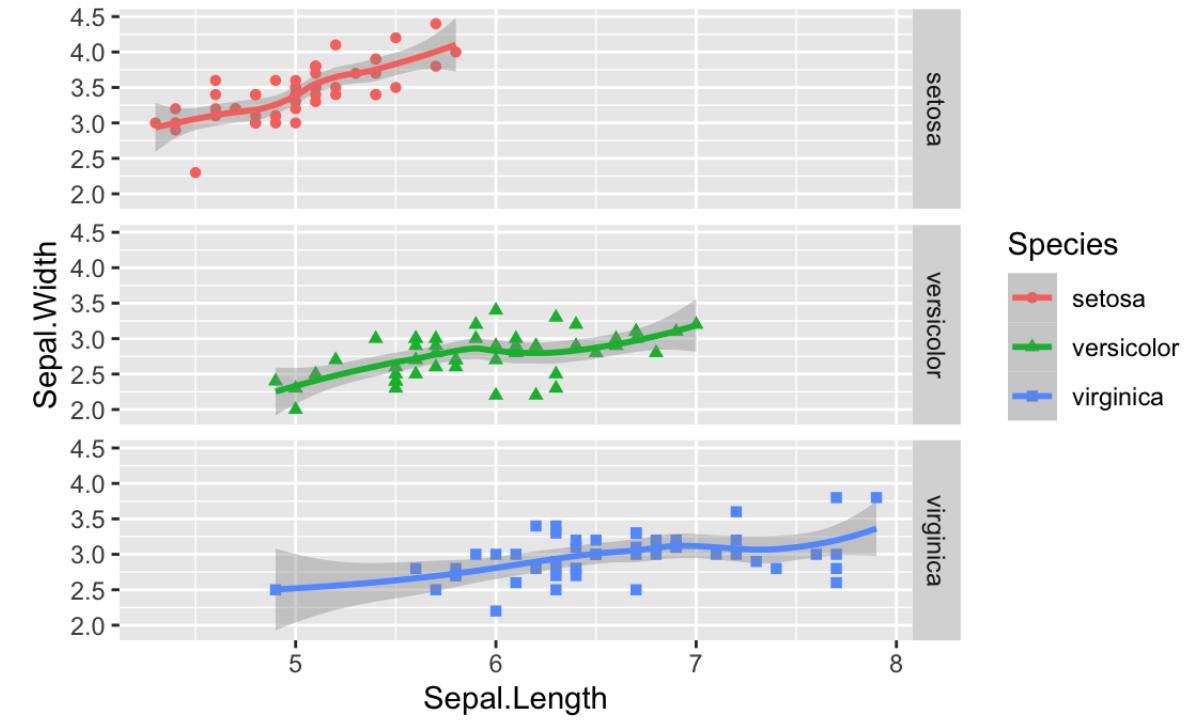


facet

```
# rows  
p + geom_smooth(method="loess") +  
  facet_grid(. ~ Species)
```

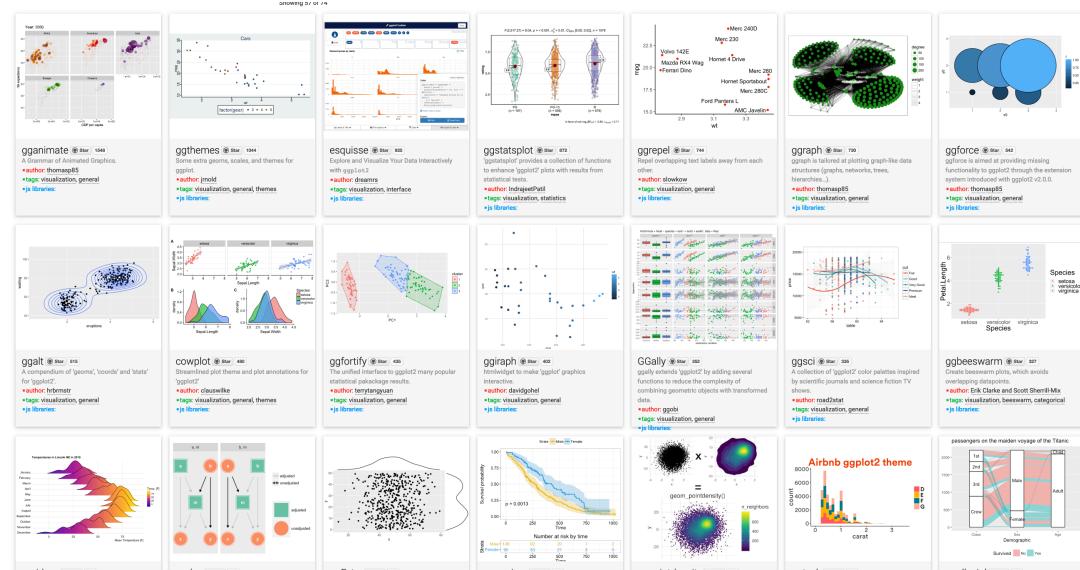


```
# along columns  
p + geom_smooth(method="loess") +  
  facet_grid(Species ~ .)
```



More ggplot functions and extensions

<https://exts.ggplot2.tidyverse.org/gallery/>



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

Top 50 ggplot2 Visualizations - The Master List (With Full R Code)

What type of visualization to use for what sort of problem? This tutorial helps you choose the right type of chart for your specific objectives and how to implement it in R using ggplot2.

Exercise – iris

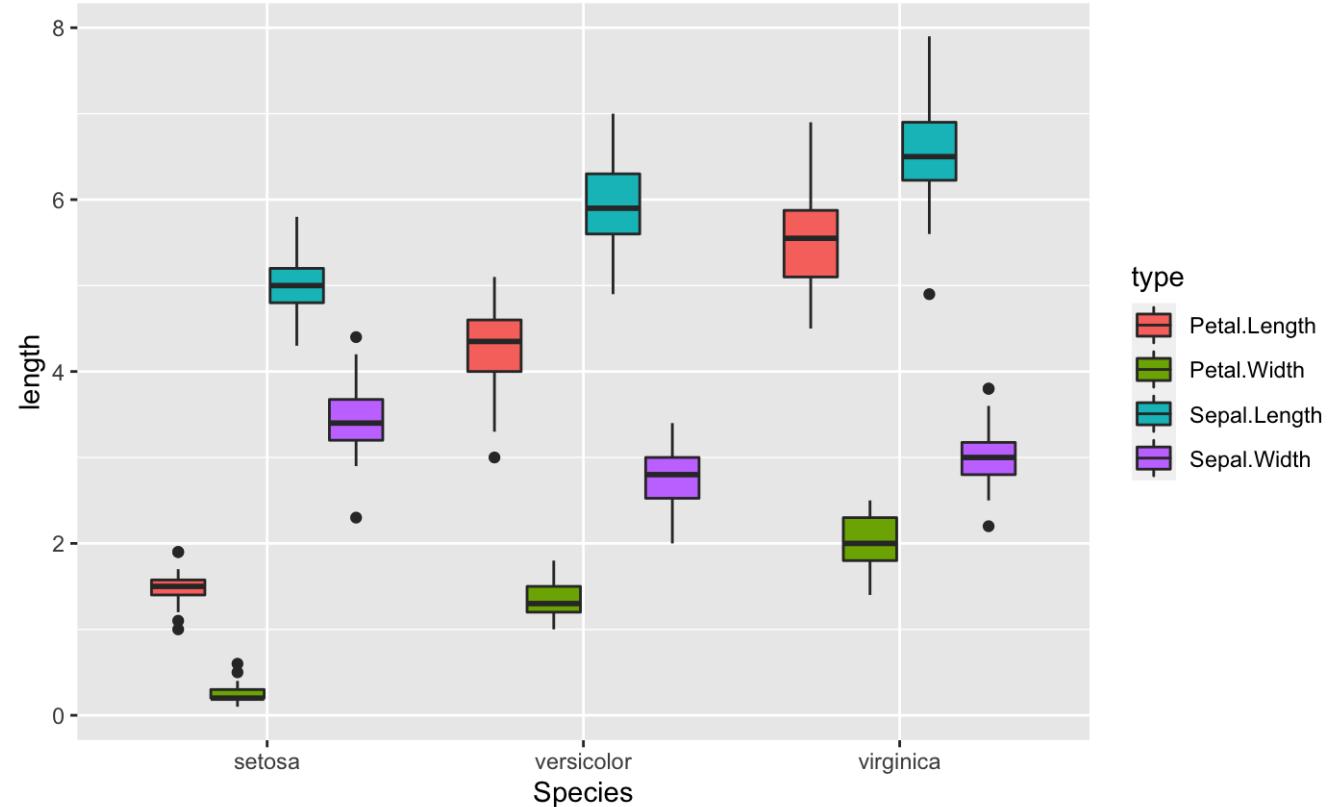
```
library(tidyverse)
tb <- as_tibble(iris)

# Try to plot boxplot of Sepal.Width categorized by species
# hint: use geom_boxplot and ?geom_boxplot to find usage

# add colour to each boxplot
# hint: use fill

# plot all four lengths in multiple boxes categorized by species
# hint: think about tidy data (long and wide?)
# final plot

# Can you make it better?
# title?
# a different theme?
```



Final plot

Exercise – starwar ☺

```
library("tidyverse")
starwars
glimpse(starwars)

# some data exploration
# plot height vs mass using ggplot and geom_point
# who's the heaviest? ; add color by sex to categorise this character quickly
# alternatively, facet_grid?

# average height by species?
# hint: group_by and summarise
# hint2: use mean(height, na.rm = TRUE) to deal with na figures

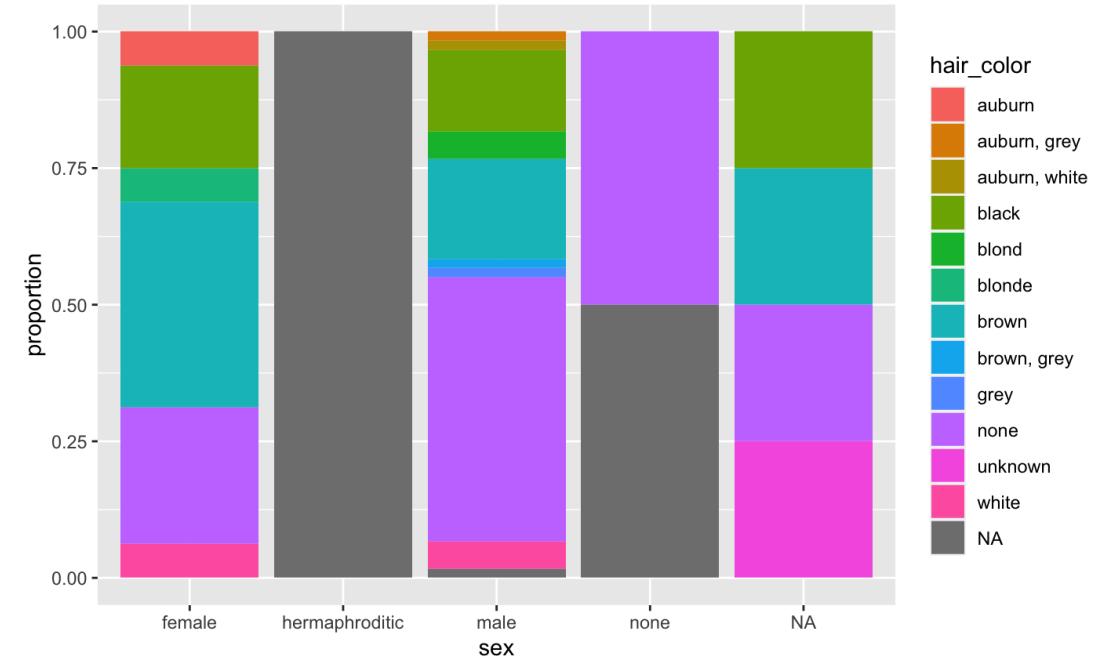
# which species is the shortest?
# hint: arrange

# visualize numbers of characters based on sex
# this is categorical data ; use geom_bar

# for each bar, can you further split them into hair_color?

# what about proportions? (final plot)

# why NA in hair_color in the hermaphroditic category?
# Note: this is purely for star wars fans
```



Final plot

Good References

OpenIntro Statistics

*** great book about statistics

- <https://www.openintro.org/>

R statistics

- <http://r-statistics.co/>

If you want to practice more:

- <https://www.datacamp.com/onboarding/learn?technology=r>
- <https://www.coursera.org/specializations/data-science-foundations-r>