From Excel to R
(Session 1-1 - Welcome to R)

Agenda

1) Introduction
2) About this workshop
3) RStudio and Tidyverse
4) Introduction to R
5) Data manipulation
6) Loops

Introduction

Hanjo Odendaal

LEAD DATA SCIENTIST (71POINT4)

ABOUT ME

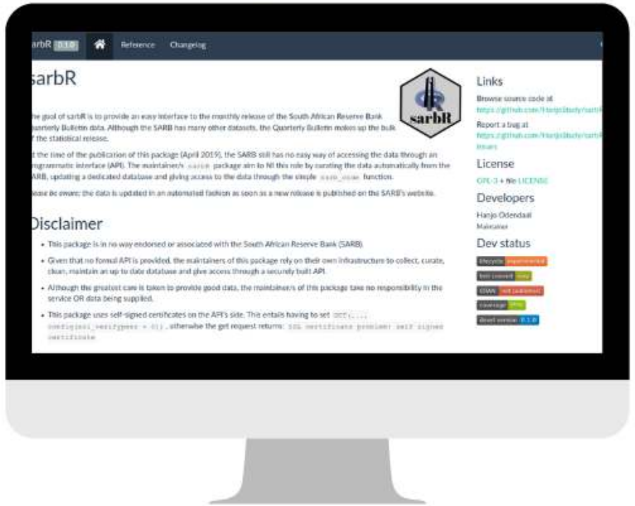
I lead the advanced data analytics and statistical modelling aspects of the work at 71point4. I am passionate about exploring different methodologies to collect and analyse new and alternative data sets.

I hold a PhD in Economics from the University of Stellenbosch: News, Sentiment and the Real Economy.

Hanjo Odendaal

LEAD DATA SCIENTIST (71POINT4)

Software Engineering



Production Machine Learning

High Performance Cloud Computing

R + amazon webservices

Web Scraping

useR!

About this workshop

What this workshop aims to achieve

- Upskill all participants to understand code used in the data pipeline

> The training only aims to serve as a foundation for participants' R coding journey

- Familiarisation with data pipeline

> Although not all of the participants will be working on data pipelines on a day to day basis, but understanding how to conduct basic analysis in R its a very powerful skill to have

Key outcomes

Following the workshop, we want the participants team to:

- Have a basic understanding of the **R code** used in a data pipeline
- Understand the **flow** of data analysis pipeline
- Being able to do a basic exploratory analysis

PLEASE:

- Ask questions, we've been down this road before! 🤓

Asking for assistance

- Please feel free to stop me and ask a question
- If you feel more comfortable asking questions in writing feel free to email them to **hanjo@71point4.com**
- Help each other out! Some might be further along their data journeys than others

Why open source

- Open source software such as R has a very large and active communtity
    - This means that the velocity of new package being made available is growing and an almost exponential rate
- This also means that the access to the latest statistical techniques is available in R with extensive documentation
- Specialized procedures is where the R community's strength lies
- Besides the direct community of R developers, there are online forums which play a significant role in the development of the software as well as you as a user

Why open source (Cont.)

- These forums give insight into practical solutions to problems and are easily accesible through the use of google:
  - Go and explore Stackoverflow
  - Dont be afraid to ask
- Open source also allows for the construction of bespoke software to use in-house
- How to receive the latest information on what people are doing
  - R-bloggers

Why move away from Excel

- Excel is a general point and click camera setup
- Reliability is not its main focus; same can be said for reproducibility
- Platform is unfortunately slow as it contains a lot of overhead

- Excel has a very limited capacity and is memory intensive as it is reactive

- Fundamental flaws:
  - Solver gives the wrong result about 40% of the time
  - Random number generation is not always random
  - Documentation is sparse

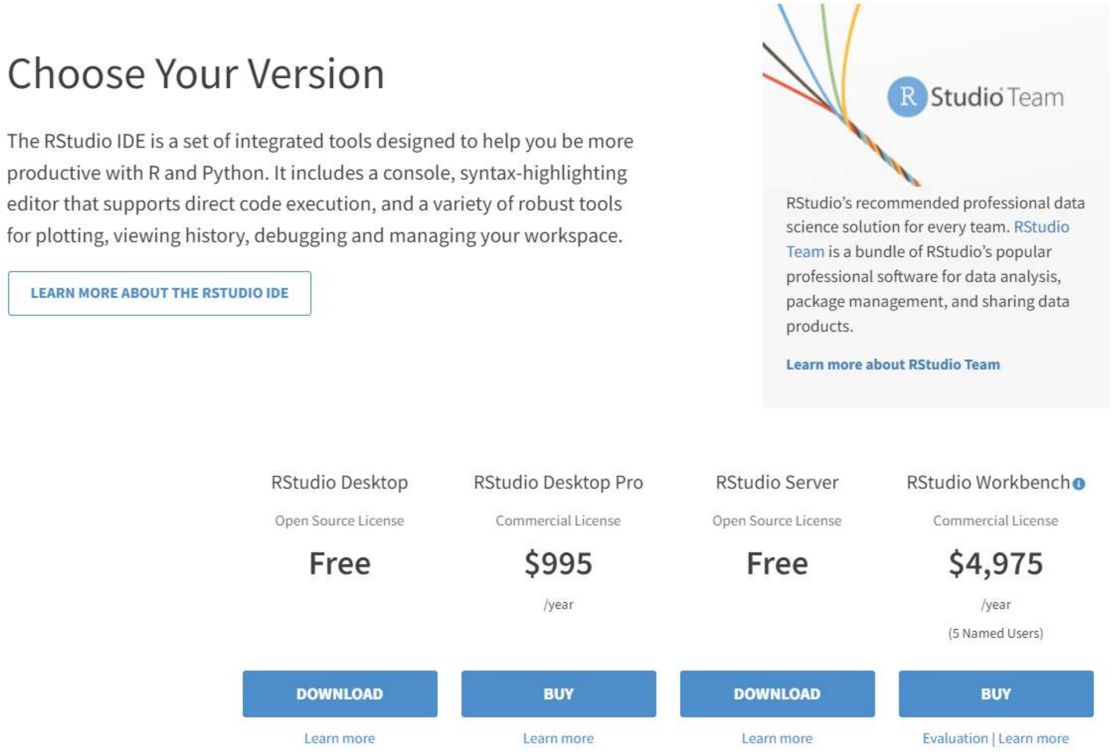
Why move away from Excel (Cont.)

- R is free
- The capabilities of the program provides the necessary toolset for data analysis. The most obvious is the plotting features
  - Histogram
  - boxplot
  - LOESS smoothing of data

RStudio setup

Installing RStudio

- RStudio on your computer - installation instructions

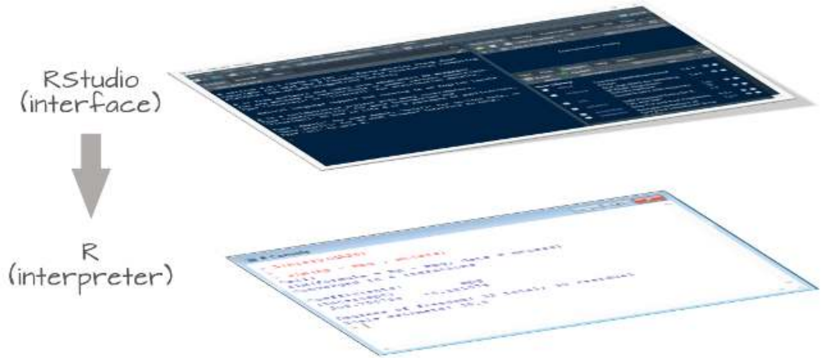


Introduction to R

Understanding the terminology

- R vs RStudio

> R and RStudio are two distinctly different applications that serve different purposes. R is the software that performs the actual instructions. It's the workhorse. Without R installed on your computer or server, you would not be able to run any commands. RStudio is a software that provides a nifty interface to R. It's sometimes referred to as an Integrated Development Environment (IDE). Its purpose is to provide bells and whistles that can improve your experience with the R software.



Understanding the teminology

- RStudio Desktop vs RStudio Server

> RStudio Desktop is an R IDE that works with the version of R you have installed on your local Windows, Mac OS X, or Linux workstation. RStudio Workbench and RStudio Server are Linux server applications that provide a web-browser-based interface to the version of R running on the server.

What is the tidyverse?

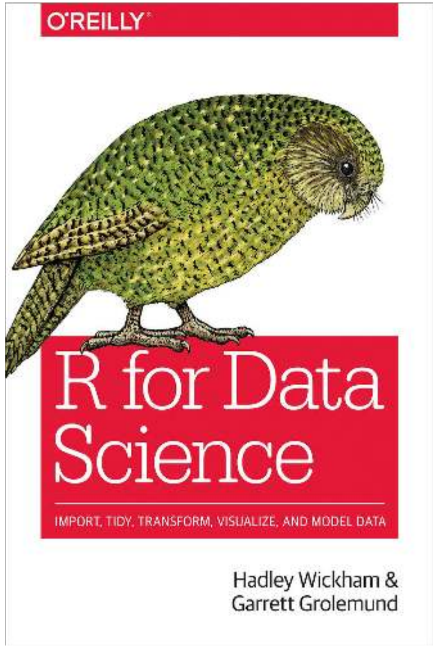
Art by Allison Horst

What is the tidyverse?

> The tidyverse is an opinionated collection of R packages designed for data science. All packages share an underlying design philosophy, grammar, and data structures.

This collection contains some of the most used libaries that an `R` data scientist will use on a daily basis. The most used packages are probably `dplyr` and `ggplot`. Today we gonna explore the *basics* of the `dplyr` package.

- `dplyr` is the grammar of data manipulation (`select`, `filter`, `group_by`, `mutate`)
- `ggplot` is the grammar of graphics - beyond the scope of this workshop (useful resources: R for Data Science and ggplot2)



What is the tidyverse?

Although we only going to be learning the basics of the tidyverse universe, there is A LOT more to explore in terms of the power of programming languages like `R` (and `Python`).

We recommend R for data analyses due to its firm pedigree in statistical analysis. `Python` is getting better at manipulating data with packages like `pandas` and alike, while `R` has become a more general language over the last few years.

Even though `python` does offer some nice integration features, `R` offers a much better ecosystem that supports reproducible research and data analysis (`Rmarkdown`, `blogdown`, `targets` etc.).

Also, once you grasp the fundamentals of programming, it is very easy to learn another language if it is better suited towards what you want to achieve.

Useful terminlogy for workshop

- Package
  - In R, the fundamental unit of shareable code is the package. A package bundles together code, data, documentation, and tests, and is easy to share with others.
  - Comprehensive R Archive Network, or **CRAN**, is the public clearing house for R packages.

- Pipe operator
  - The pipe operator is a special operational function available under the magrittr and dplyr package (basically developed under magrittr), which allows us to pass the result of one function/argument to the other one in sequence. It is generally denoted by symbol %>% in R Programming.
  - Keyboard shortcut: `ctrl` + `shift` + `m`
  - Note you can view all keyboard shortcuts with: `alt` + `shift` + `k`
  - Shortcuts can be modified through the **Tools** menu at the top of your RStudio IDE

Useful terminlogy for workshop

- Function
  - In R, a function is an object so the R interpreter is able to pass control to the function, along with arguments that may be necessary for the function to accomplish the actions

- Assign
  - To do useful and interesting things in R, we need to assign values to objects. To create an object, we need to give it a name followed by the assignment operator <-, and the value we want to give it.
  - Keyboard shortcut: `alt` + `-` (dash)

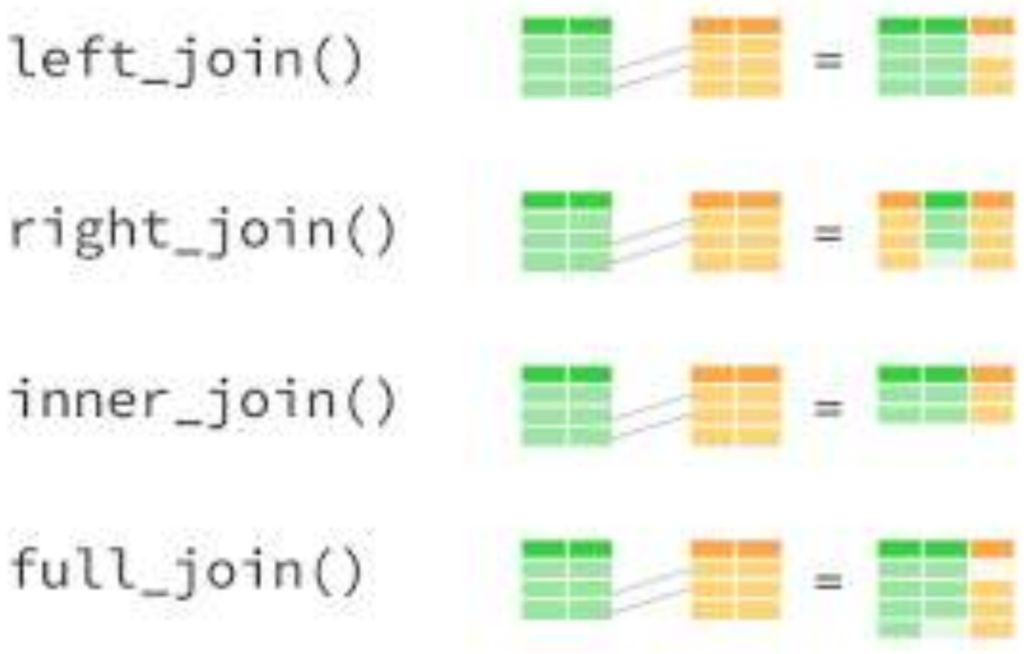
Useful terminlogy for workshop

- Loop
  - A `loop` is a control statement that allows multiple executions of a statement or a set of statements. The word 'looping' means cycling or iterating.

- Apply
  - `apply` functions are a family of functions in base R which allow you to repetitively perform an action on multiple chunks of data. An apply function is essentially a loop, but run faster than loops and often require less code.

Useful terminlogy for workshop

- Map
  - The `map` functions transform their input by applying a function to each element of a list or atomic vector and returning an object of the same length as the input. A map function is a more elegant version of a loop (requires less code therefore less room for error)



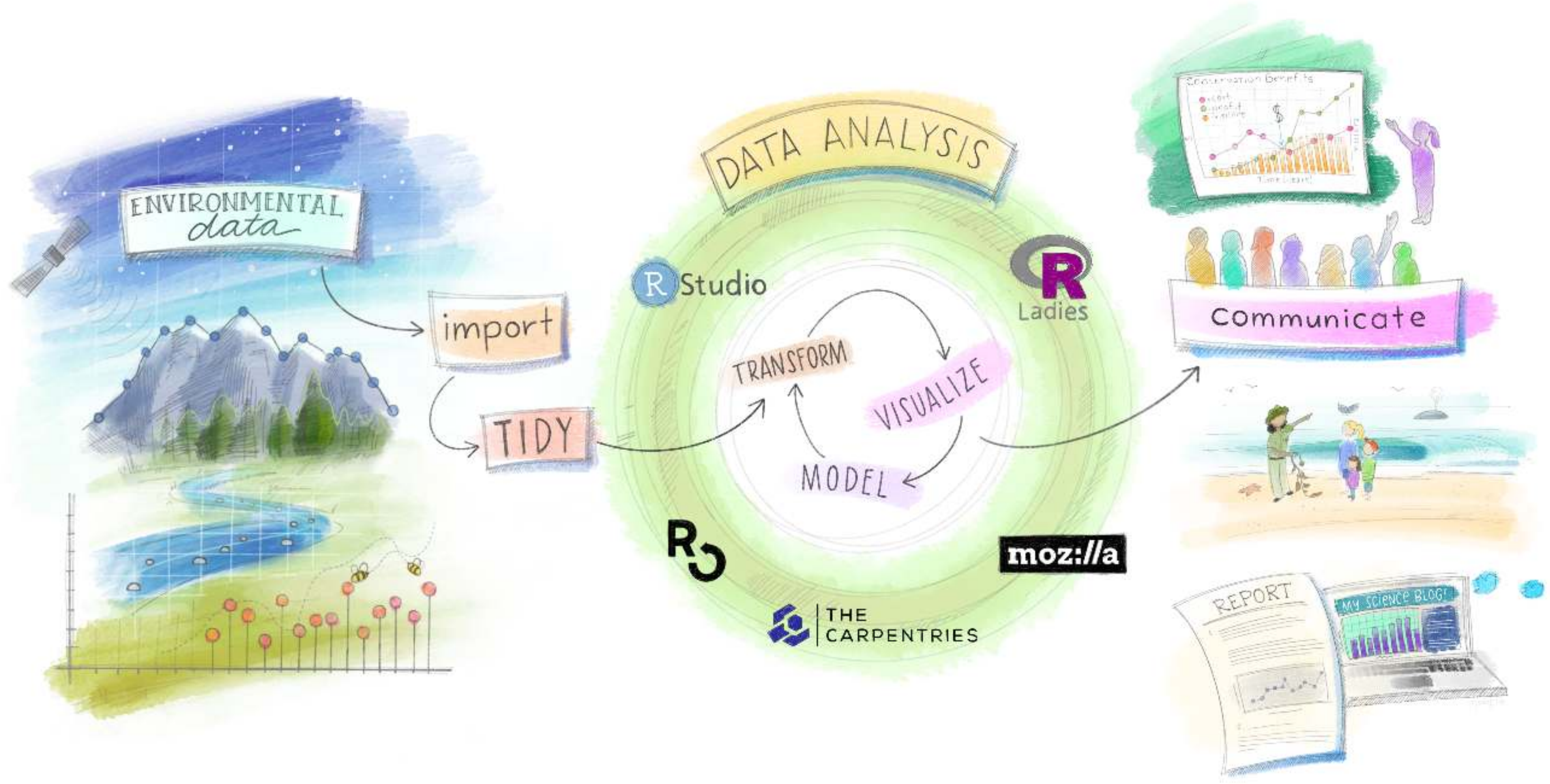
- Joins
  - We can merge two data frames in R by using the join of functions.

Useful terminlogy for workshop

- If statements
  - It is one of the easiest decision-making statements. It is used to decide whether a certain statement or block of statements will be executed or not i.e if a certain condition is true then a block of statement is executed otherwise not.

- Scripts
  - A script is simply a text file containing a set of commands and comments. The script can be saved and used later to re-execute the saved commands. The script can also be edited so you can execute a modified version of the commands.
  - Keyboard shortcut: `ctrl` + `shift` + `n`
  - Comments in R scripts are preceded by the `#` (pound/hastag) symbol

Useful terminlogy for workshop



The best way is the see for yourself

Get to know some keyboard shortcuts:

- How do I search for a word in my code?
- Where do I find that Rstudio IDE cheatsheet again?
- What version of RStudio are you running?
- What is the shortcut for the assignment "<-" operator?
- What is the shortcut for the pipe "%>%" operator?
- Using only the keyboard, how do I move to console?
- How do I set my layout so that my background is black and easier on the eyes?

Installing Solar-Putty

What is Solar-Putty?

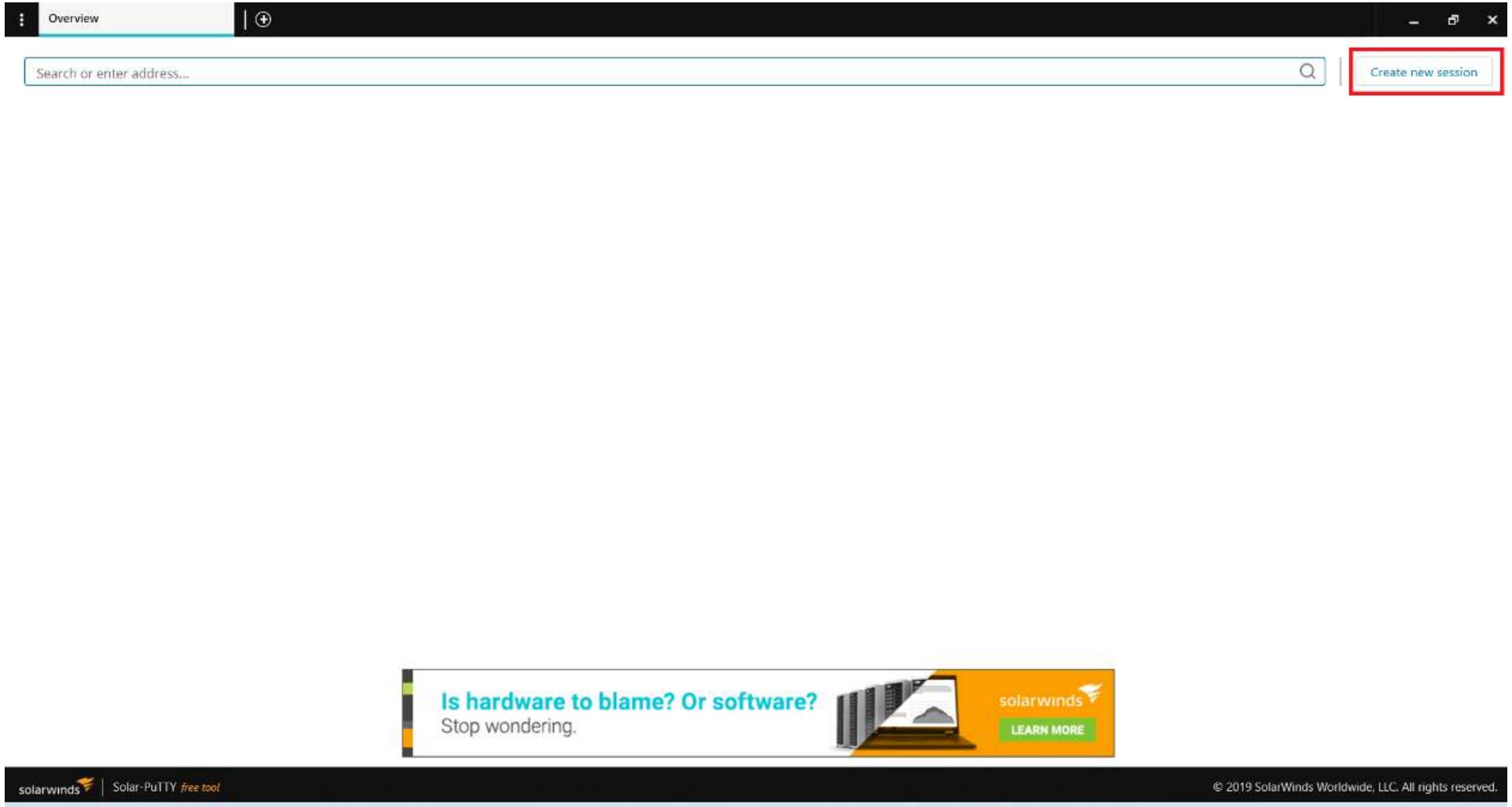
A tool to help us manage remote sessions, which is us logging onto remote servers

Managing remote sessions have never been so easy and comfortable!

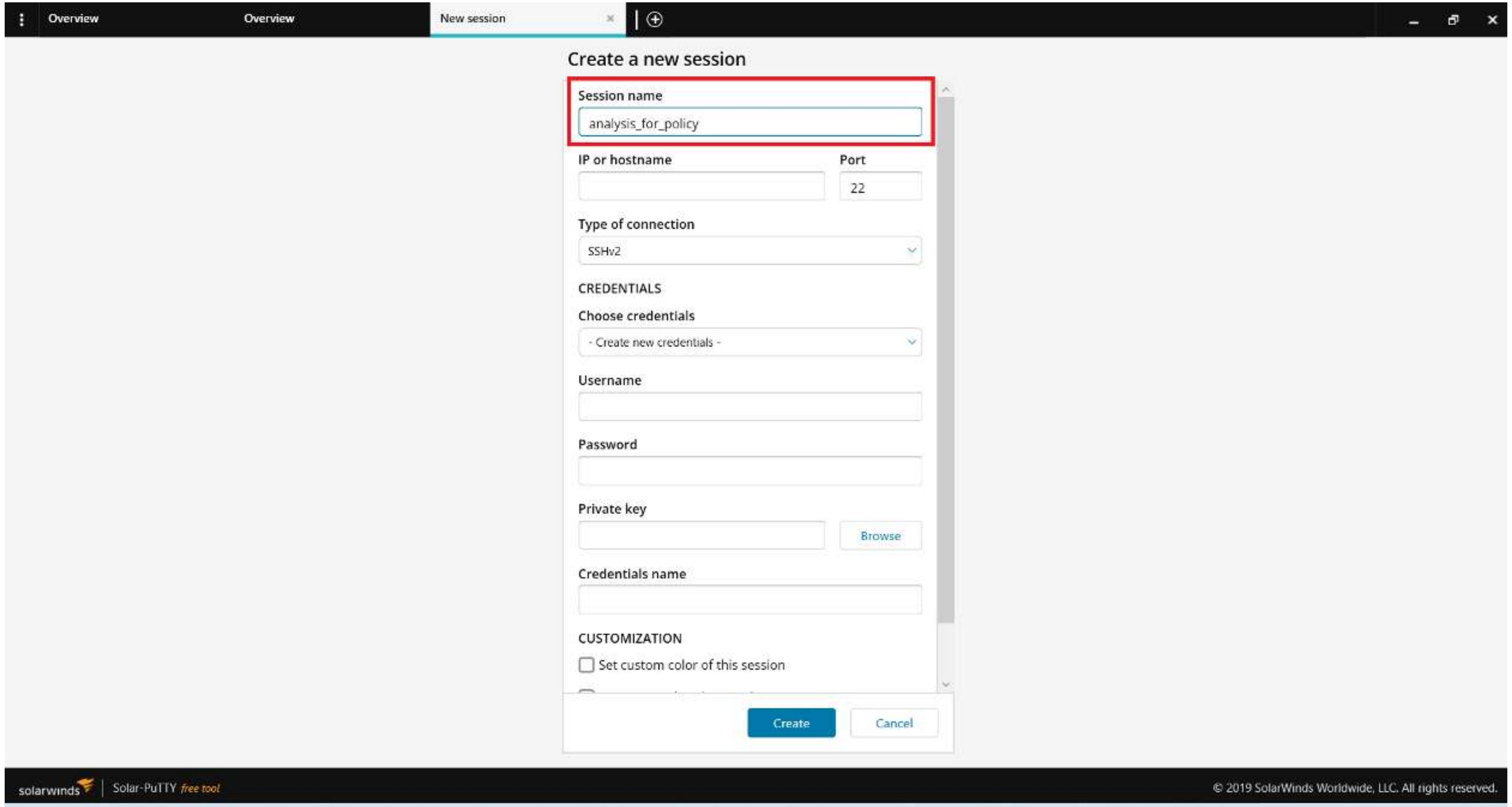
Experience Solar-PuTTY, the SSH client you always wanted

	Solar-PuTTY 100% Free	PuTTY
Support of SCP, SSH, Telnet, SFTP	✔	✔
Saving credentials (including private key) for auto-login	✔	–
Support of multiple sessions in tabbed interface	✔	–
Quick access to the most used sessions	✔	–
Auto-reconnecting capability	✔	–
Graphical SFTP file transfer	✔	–
Support of post-connections scripts	✔	–
Integration of Windows Search	✔	–
	DOWNLOAD FREE TOOL	

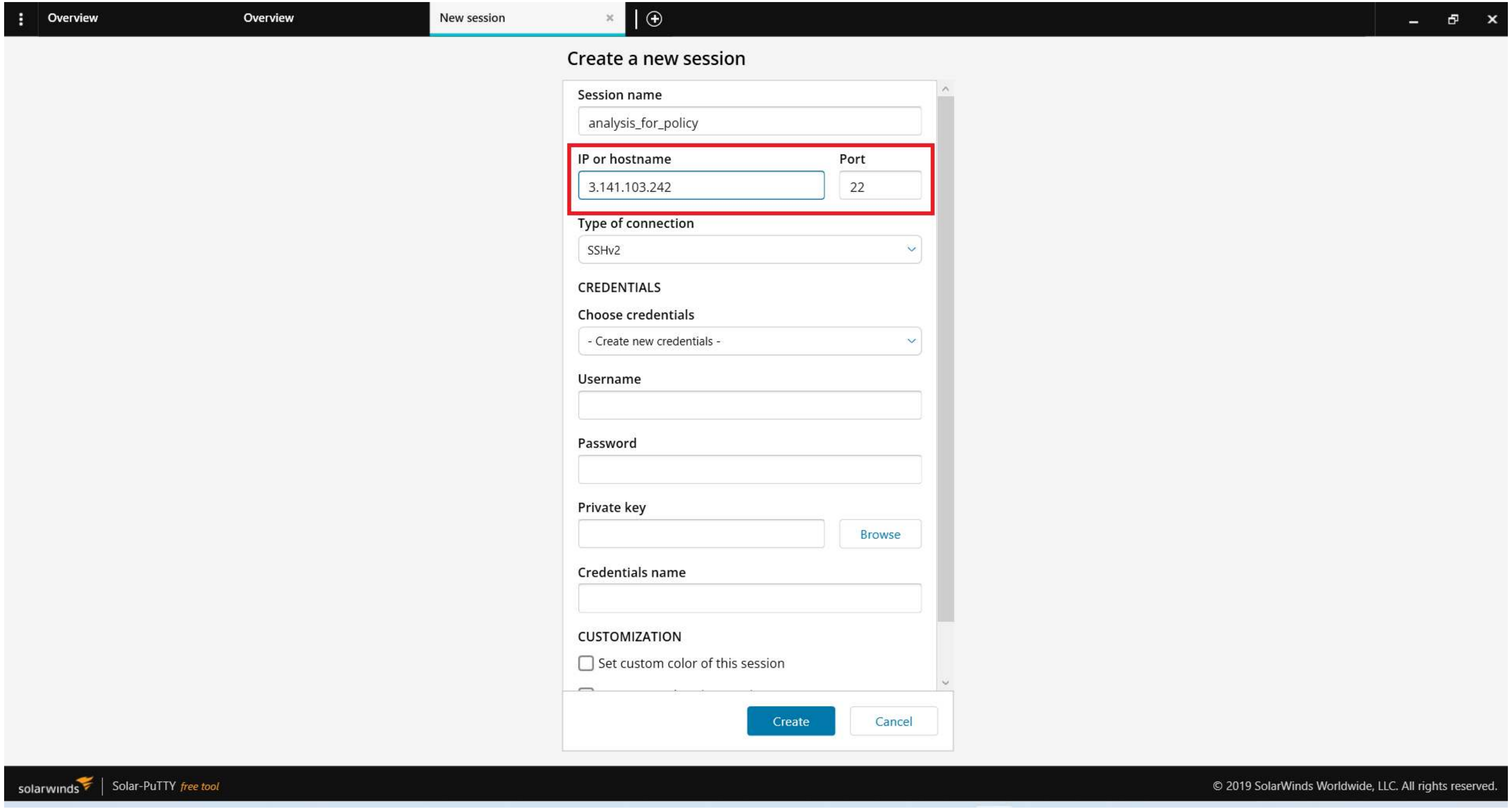
Steps to installation



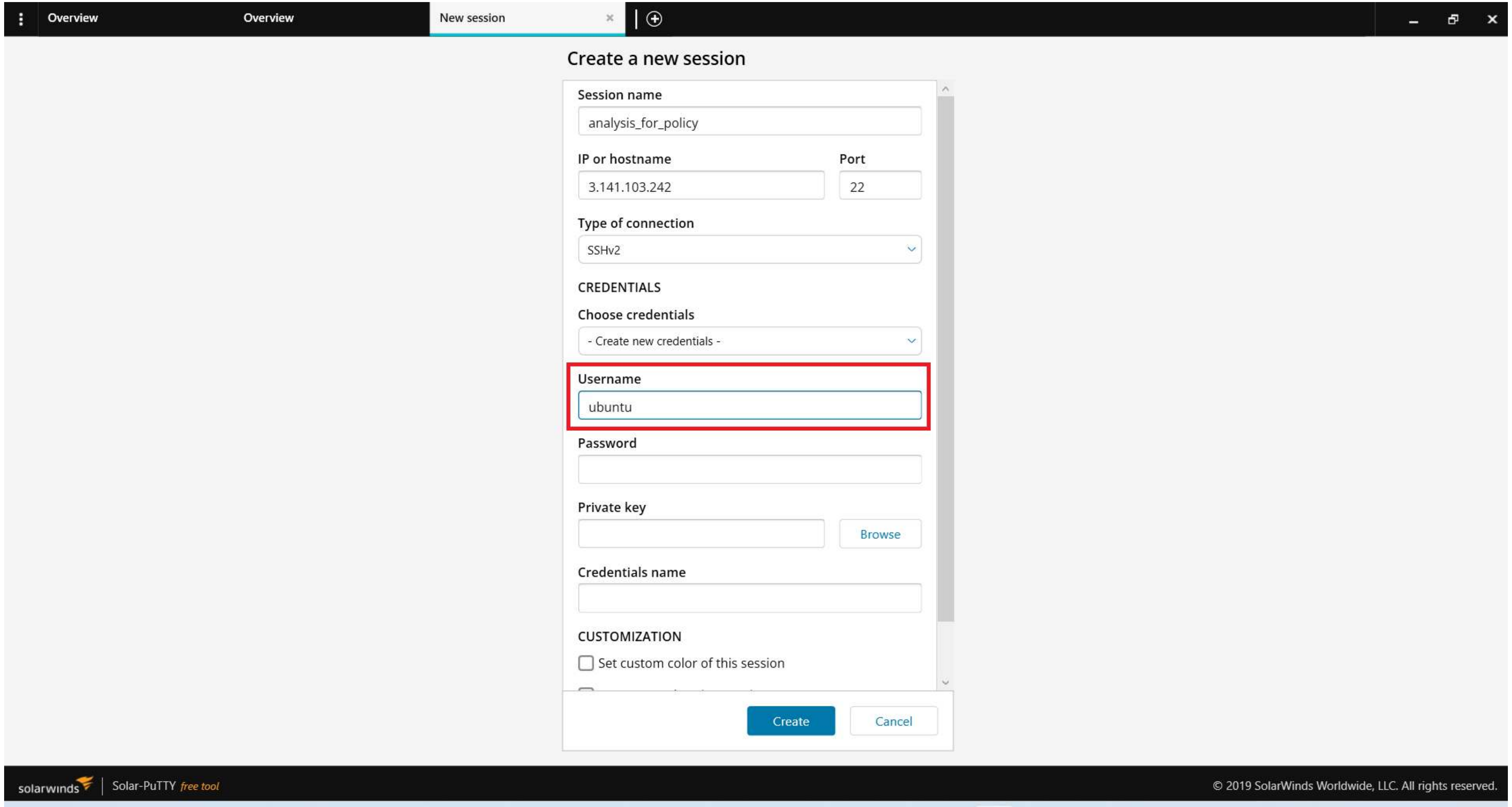
Steps to installation



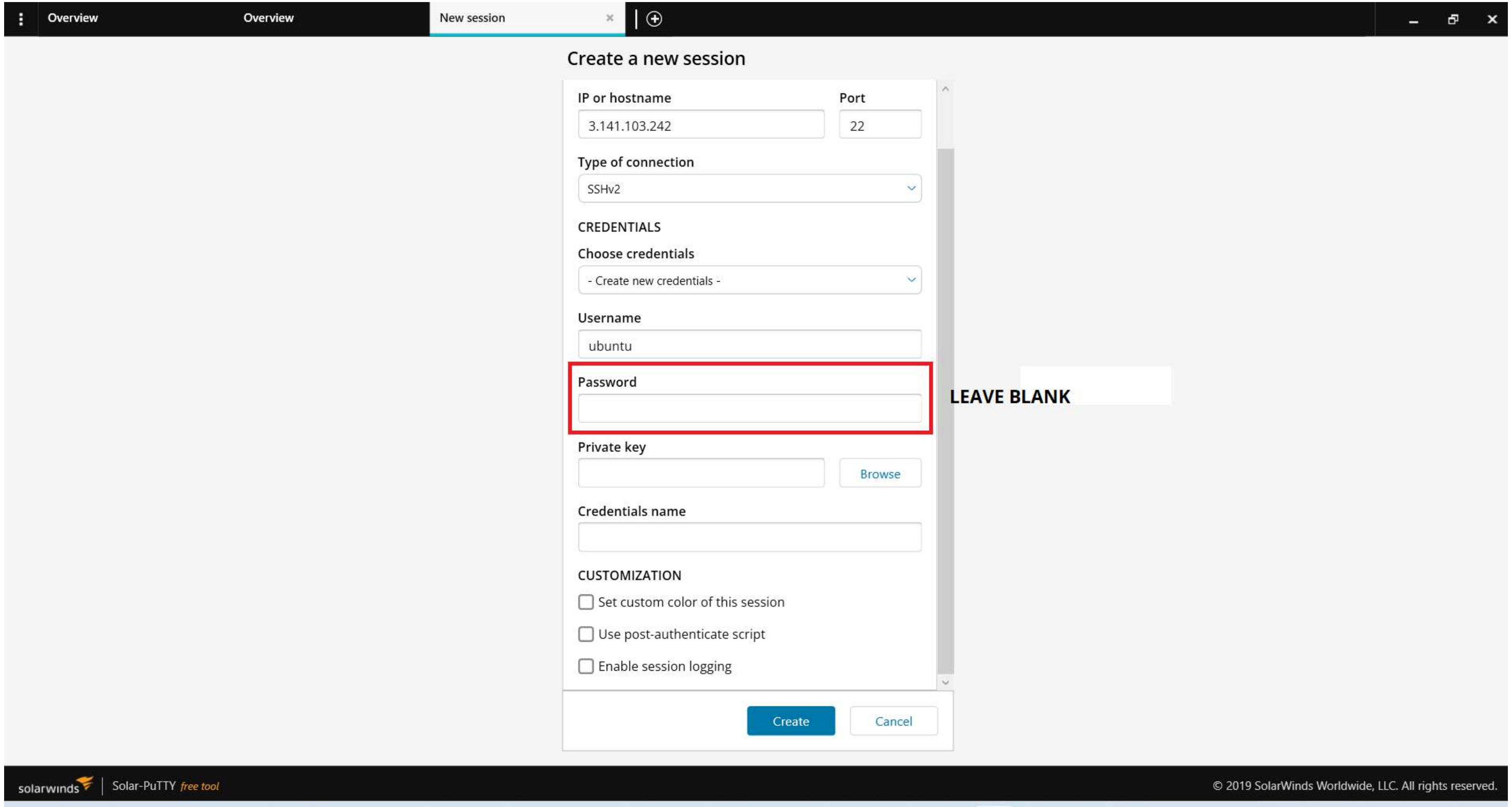
Steps to installation



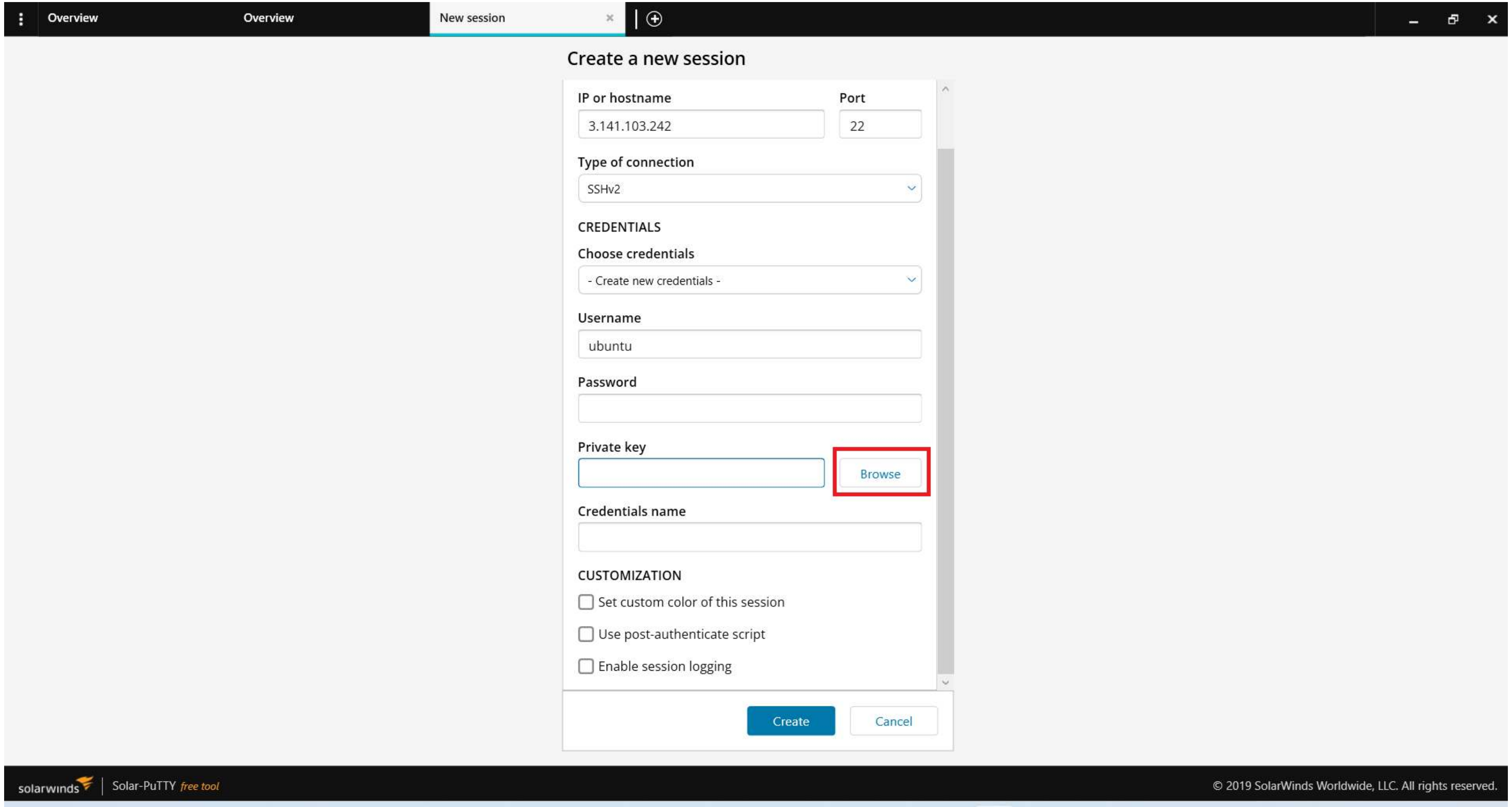
Steps to installation



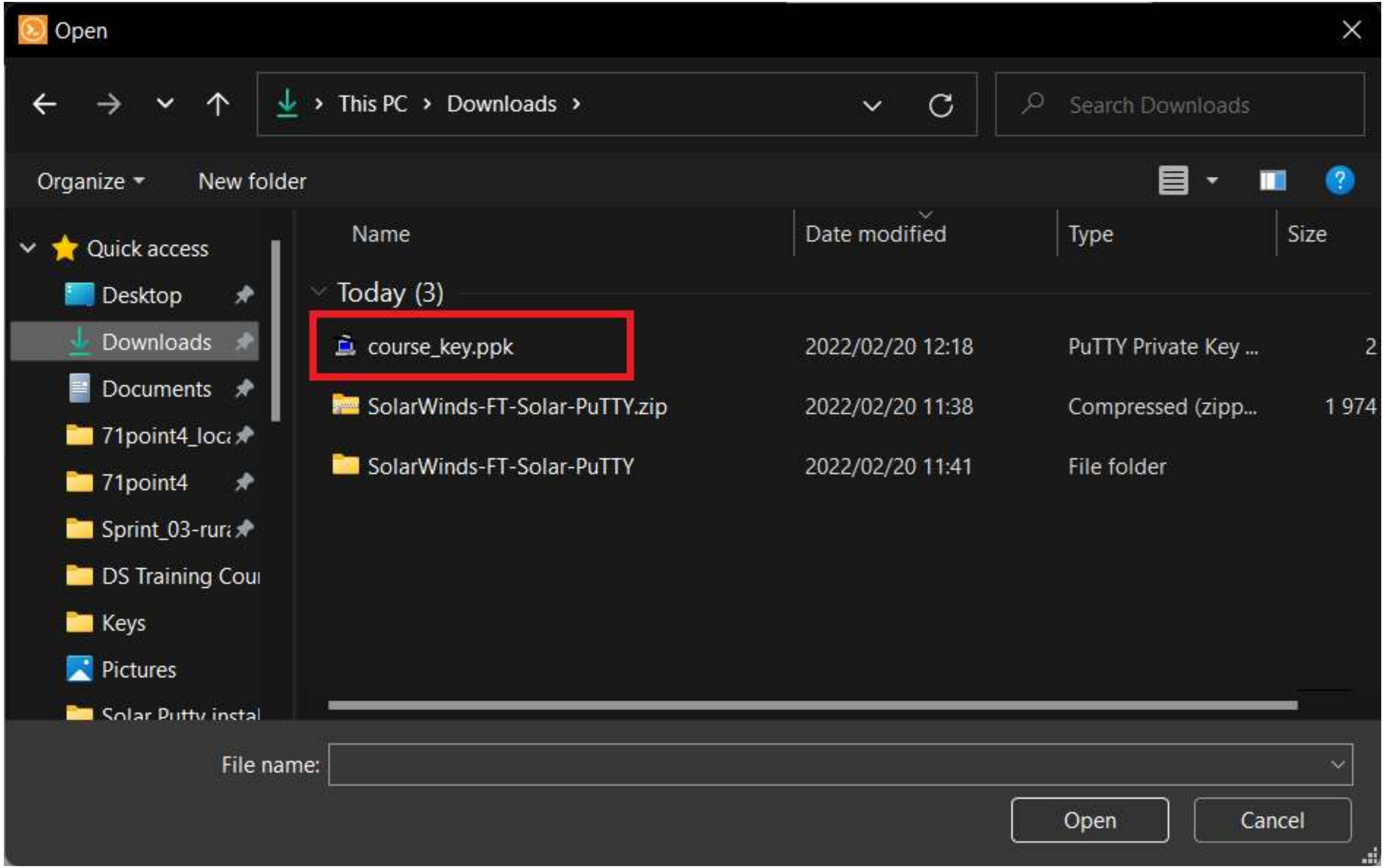
Steps to installation



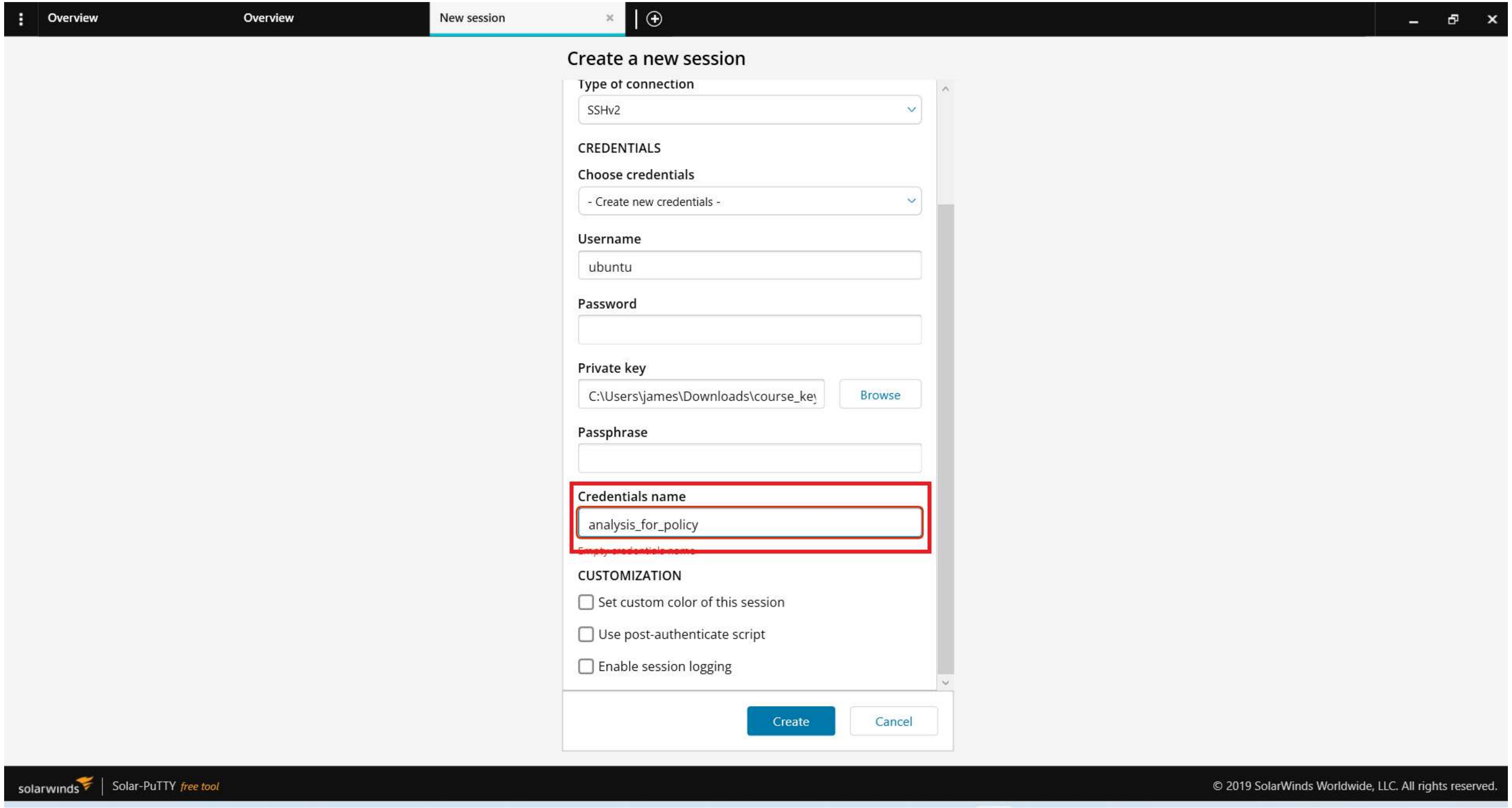
Steps to installation



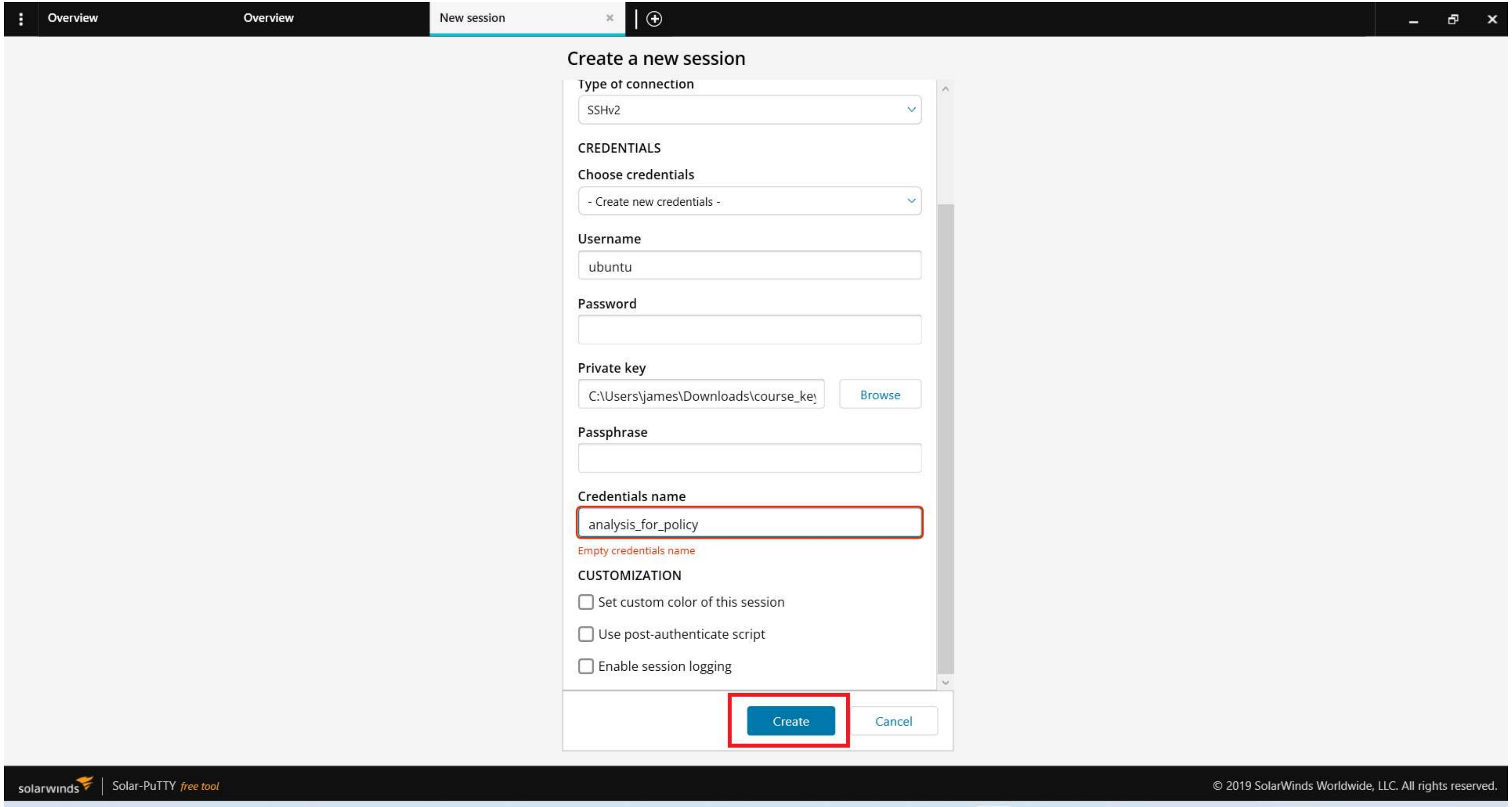
Steps to installation



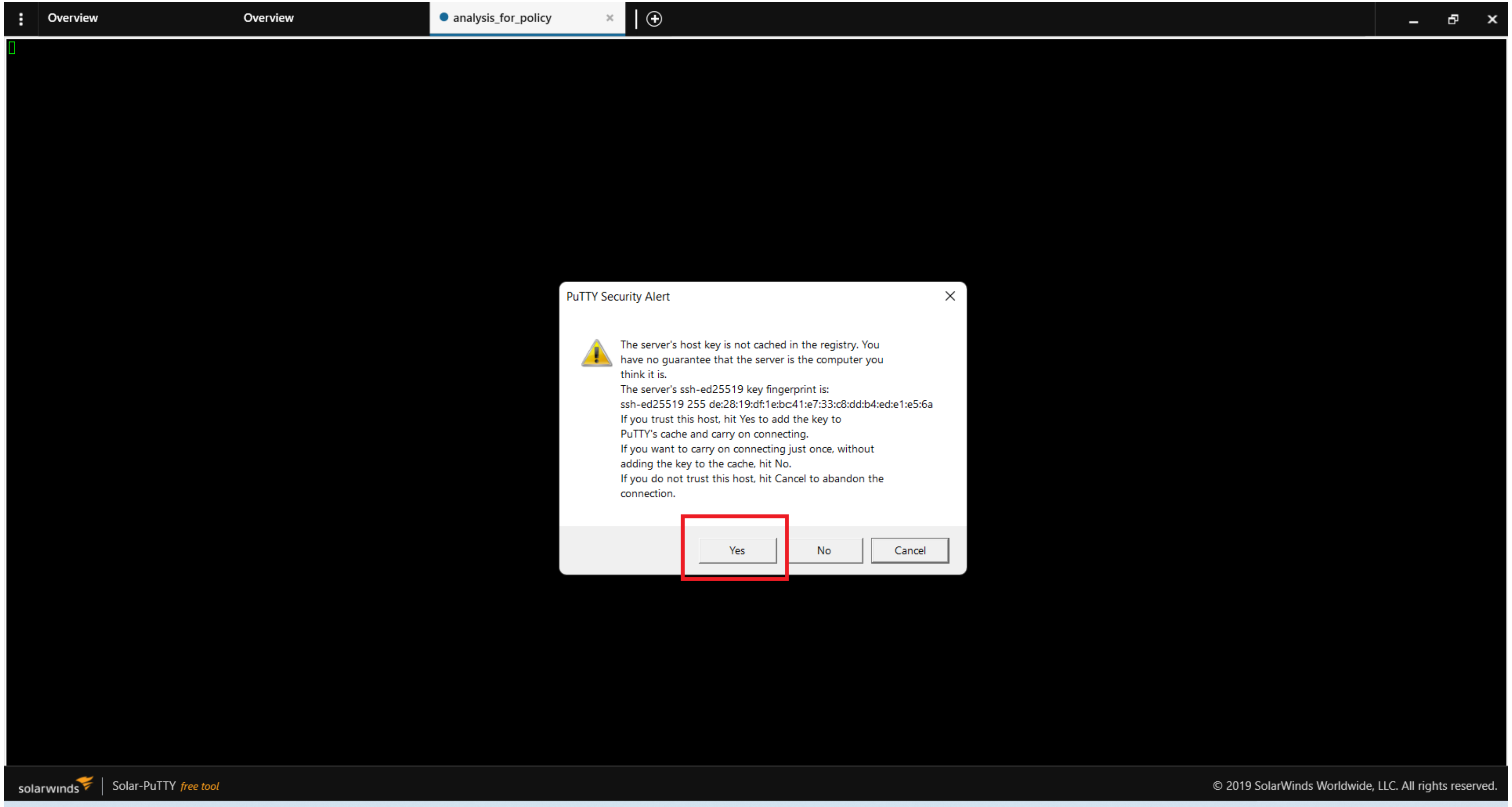
Steps to installation



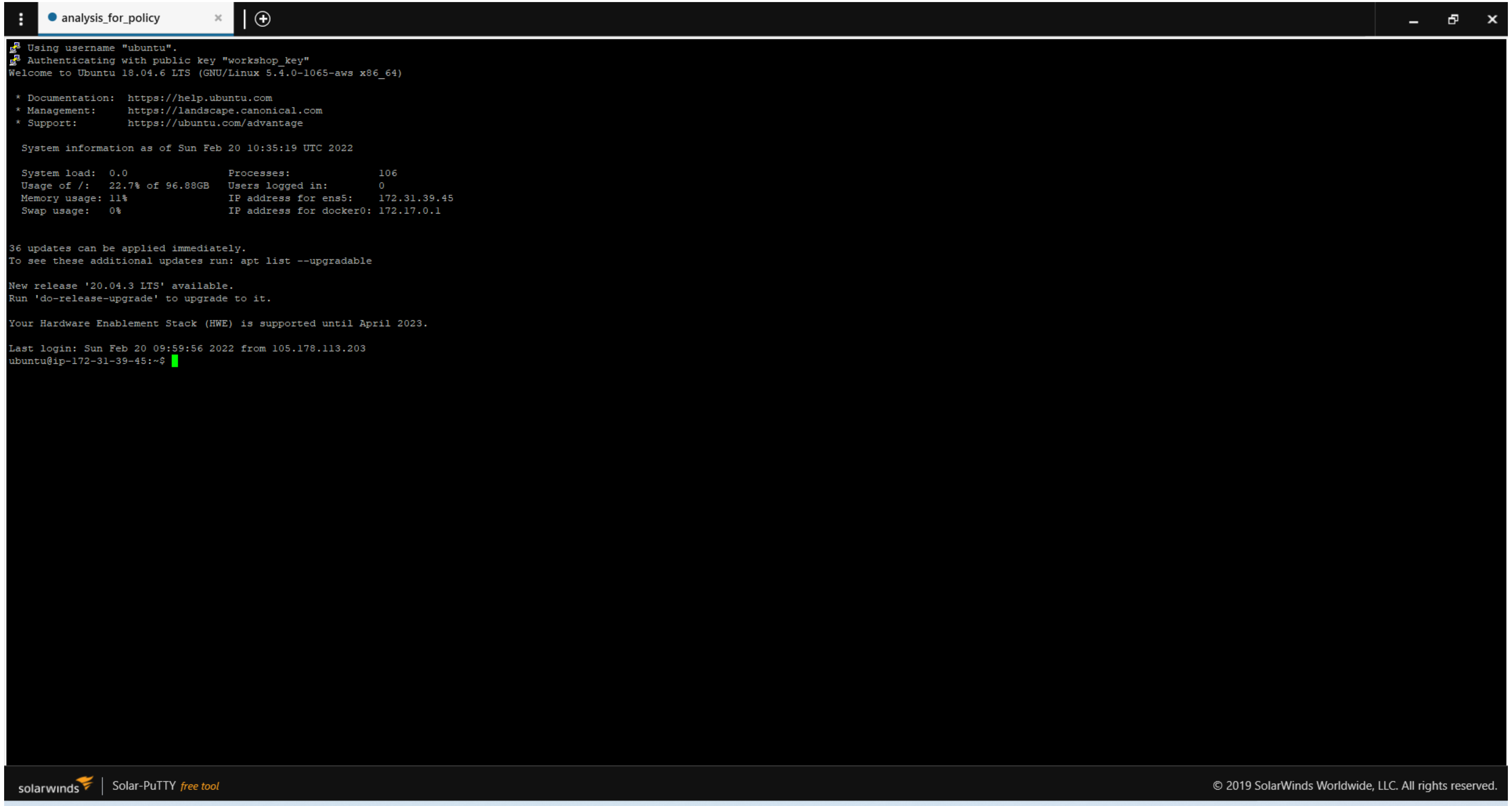
Steps to installation



Steps to installation



Steps to installation



From Excel to R

(Session 1-2 - Documenting Recap)

Agenda

1) Homework
2) Documenting with Rstudio
3) Introduction to databases

Documenting with Rstudio 🧪

Why do we document our code?

When working in a lab, it is important to always take notes on the steps taken in the experiment - why?

- Ensure robustness of results.
- Reliability of reproducibility.
- Ensures that decision can be made using the notes.
- Future you will hate you if you didn't write good documentation and need to redo the experiment or analysis.

But we do not just write down irrelevant comments, we need to make sure our documentation FAIR:

- findable
- accessible
- interoperable
- reusable

> i.e. they must adequately describe procedure, archive changes, and make the results accessible in an easy manner.

⚠️ As programmers, we need to ensure that we document both the code that produced the results as well as the procedures used to conduct the analysis (data cleaning, sampling, source of information etc.).

Reproducible research as a philosophy

A data analysis is reproducible if all the information (data, files, etc.) required to reproduce the analysis is available to someone else (or future you). These include (but is not limited to):

- Data repository.
- All code files for cleaning raw data.
- All code files and software (specific versions, packages) used in the analysis.

Some advantages of making your research reproducible are [1]:

- You can (easily) figure out what you did six months from now.
  - If your documentation was well done.
- You can (easily) make adjustments to code or data, even early in the process, and re-run all analysis.
- When you're ready to publish, you can (easily) do a last double-check of your full analysis, from cleaning the raw data through generating figures and tables for the paper.
- You can pass along or share a project with others.
  - Especially true once you learn `git`
- You can give useful code examples to people who want to extend your research.

[1] *Gandrud, C., 2013. Reproducible research with R and Rstudio. CRC Press.*

Installing your first piece of software

> Experience the ease of software installation in Linux!

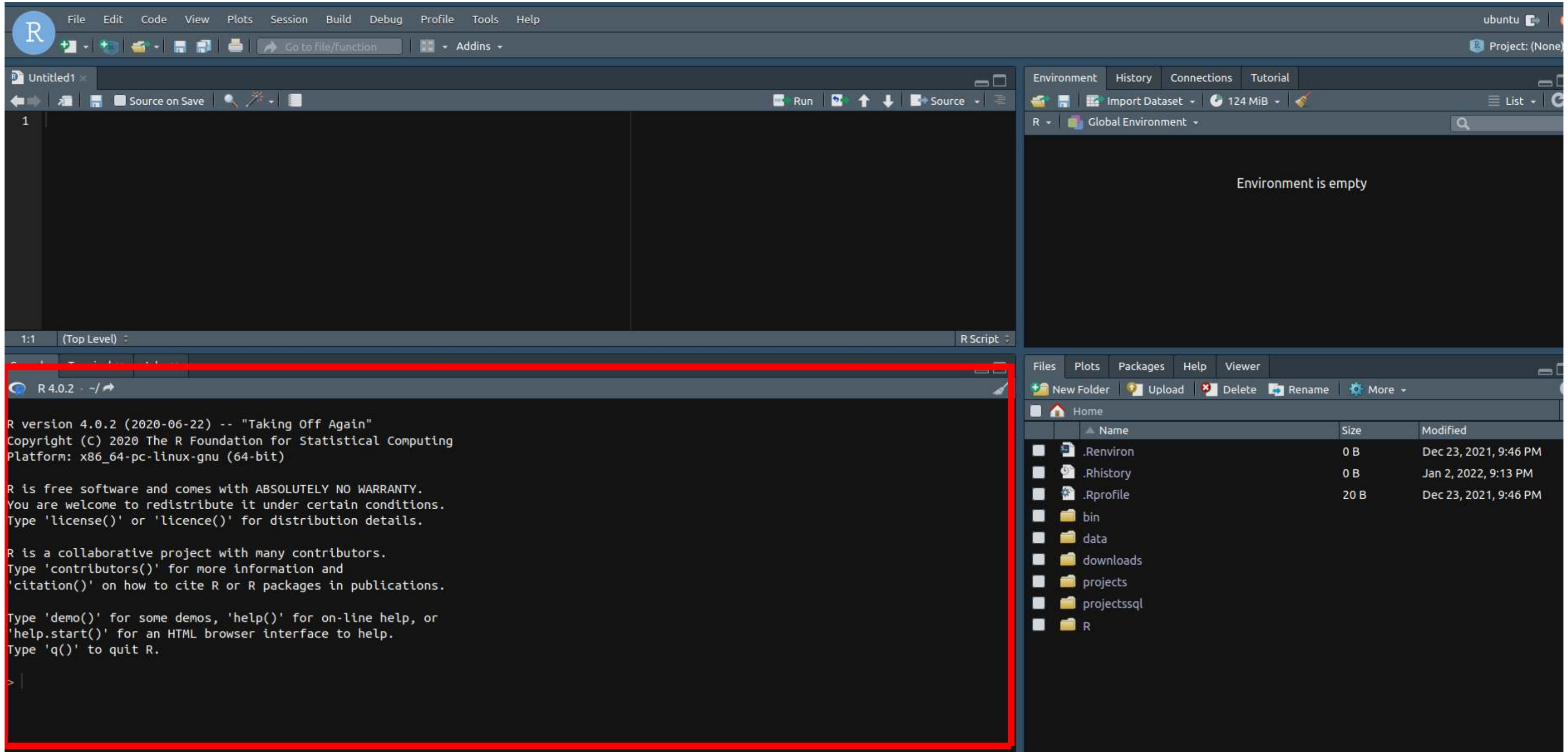
Go to rstudio server website.

```
hanjo@optimus:~$ cd Downloads
hanjo@optimus:~$ wget {installation file path}
hanjo@optimus:~$ dpkg -i {installation file}
```

Verify your install, go to the ip of your machine in the web-browser

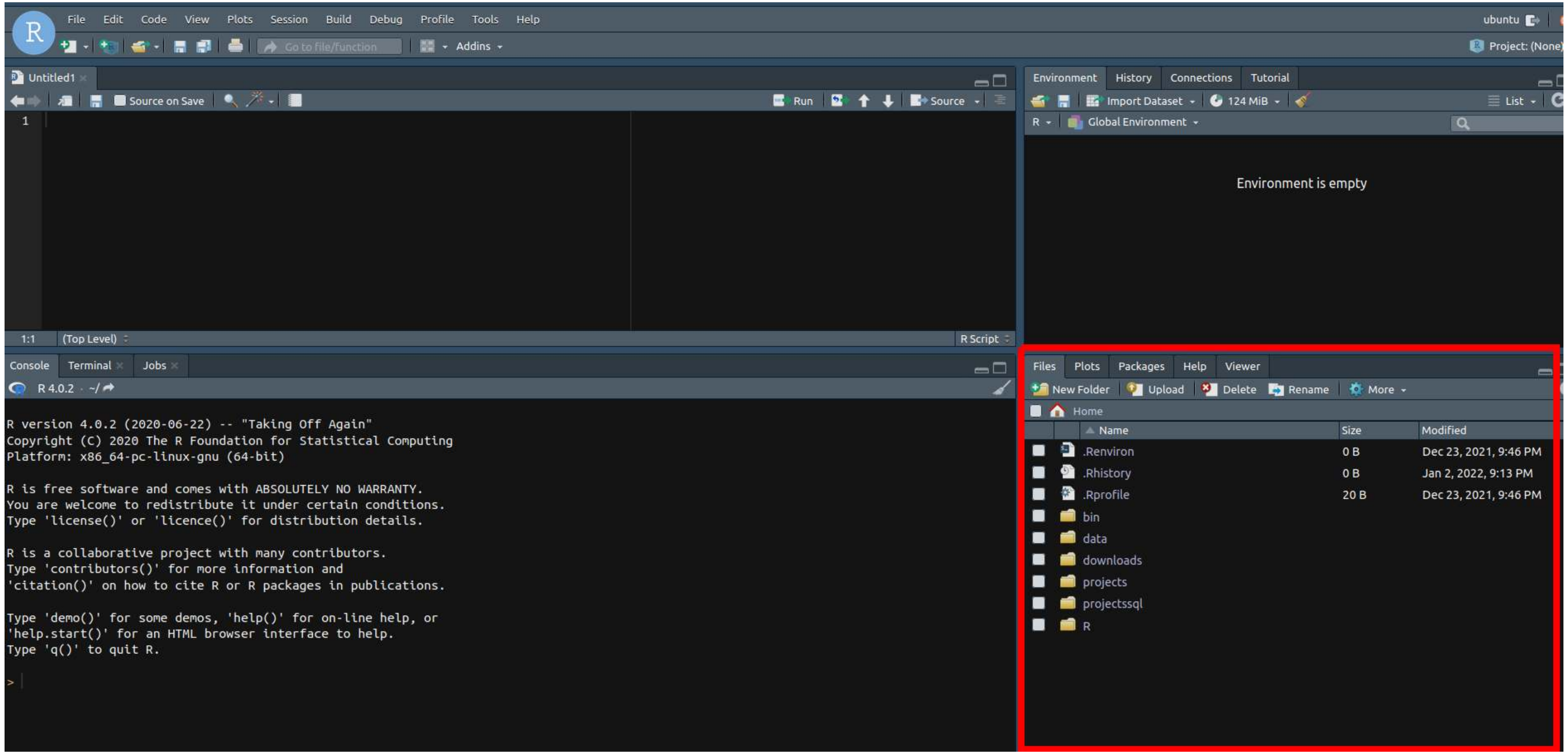
Rstudio recap

The console give you a place to execute commands written in R.



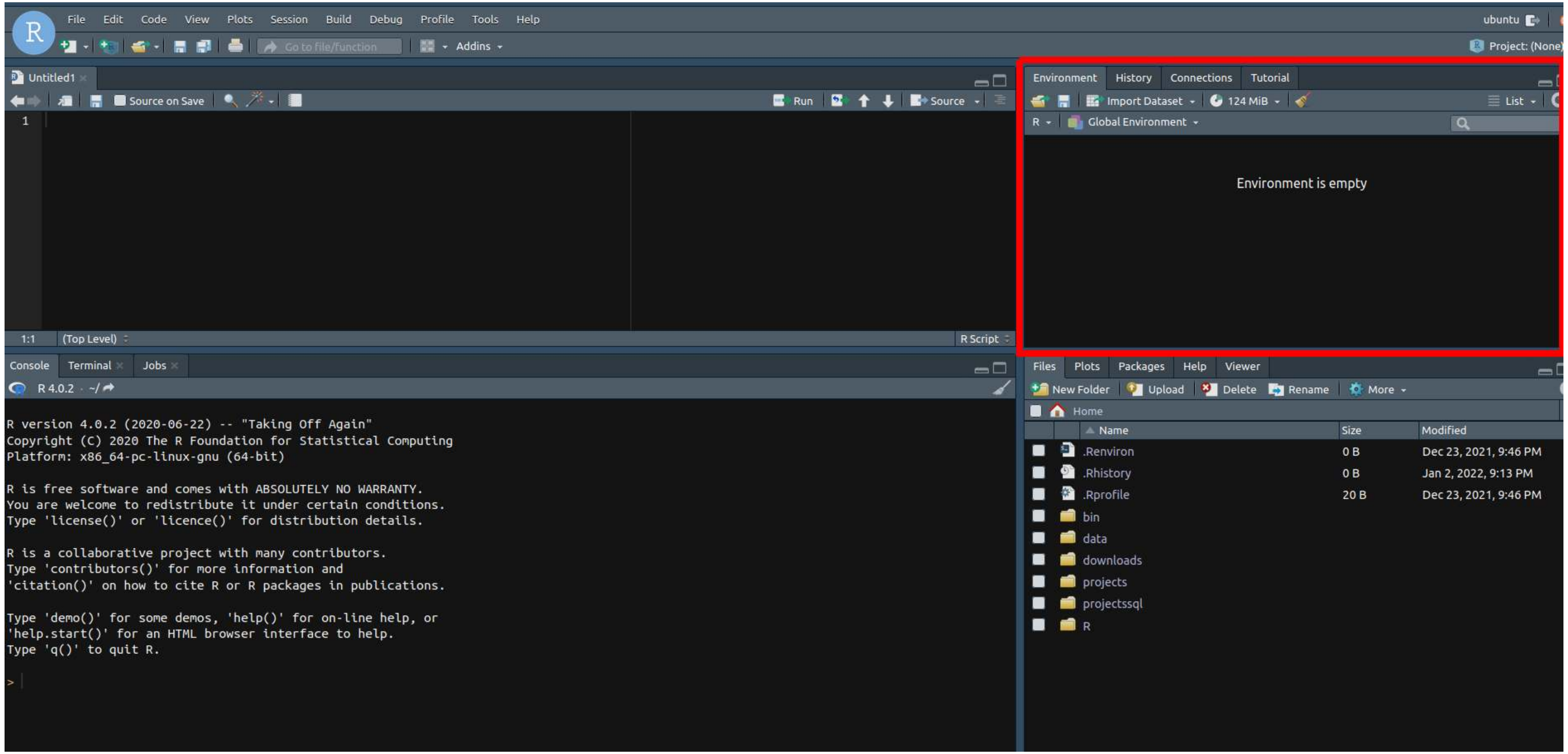
Rstudio recap

Rstudio also provides a *file explorer* which allows users to navigate the folders easily.



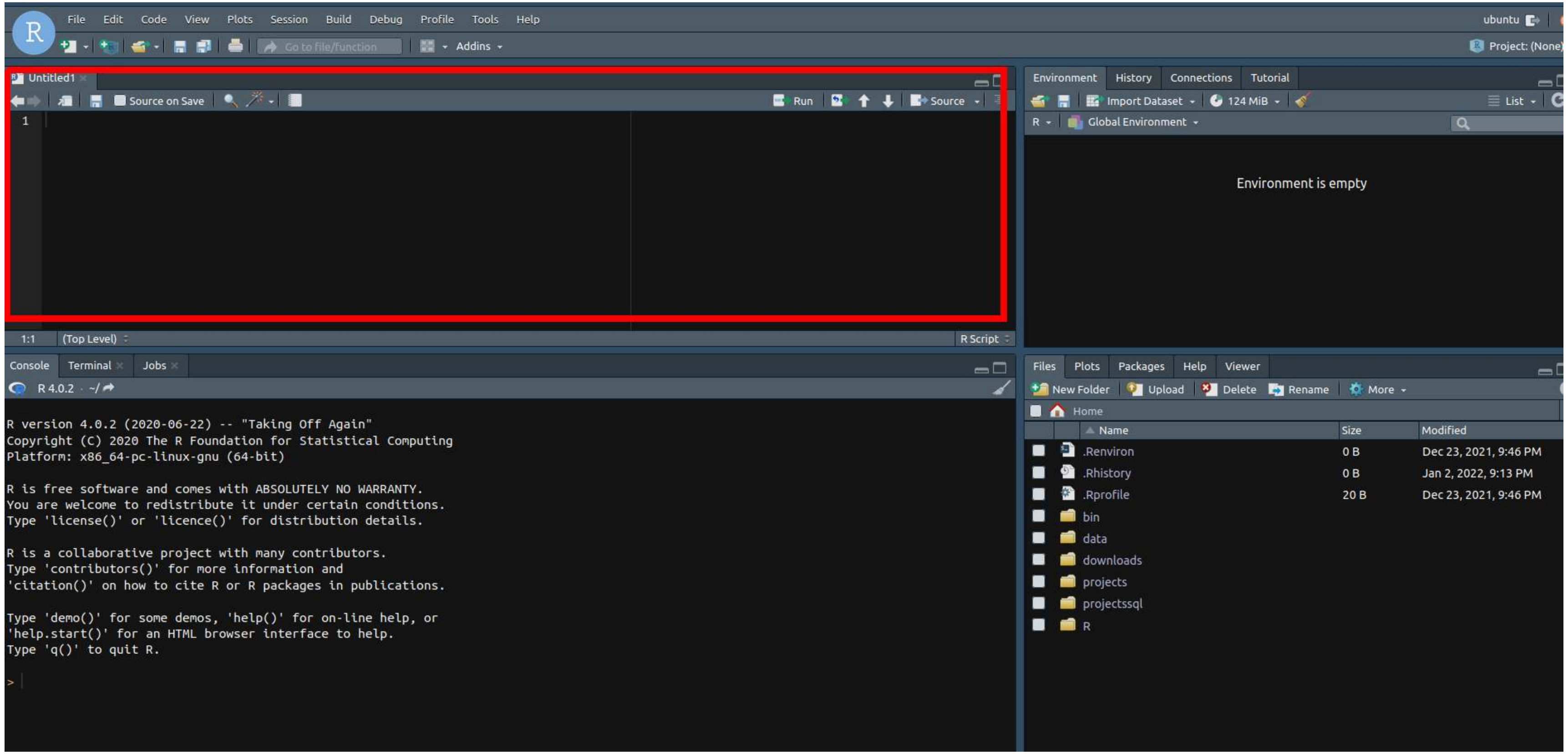
Rstudio recap

Once we start *assiging* outputs to objects, they will appear in the environment window.



Rstudio recap

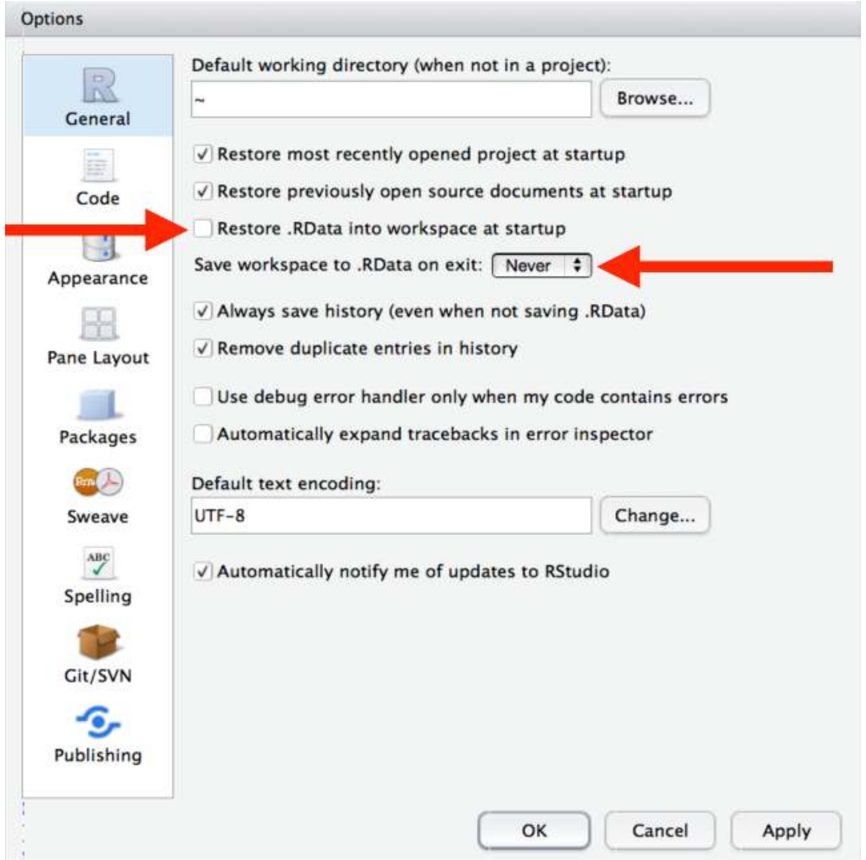
Lastly, and most importantly, we want to write scripts that we can rerun at a later time.



Setting up Rstudio for analysis

> To ensure reproducibility, we want to ensure that our scripts are always able to run without needing some hidden data.

- Make sure that the Rstudio never restores `.RData` at startup.
- This ensures that no hidden *objects* are still in your *environment* when you start Rstudio.
    - We will talk a little bit more about these concepts later in the course.



Using Projects

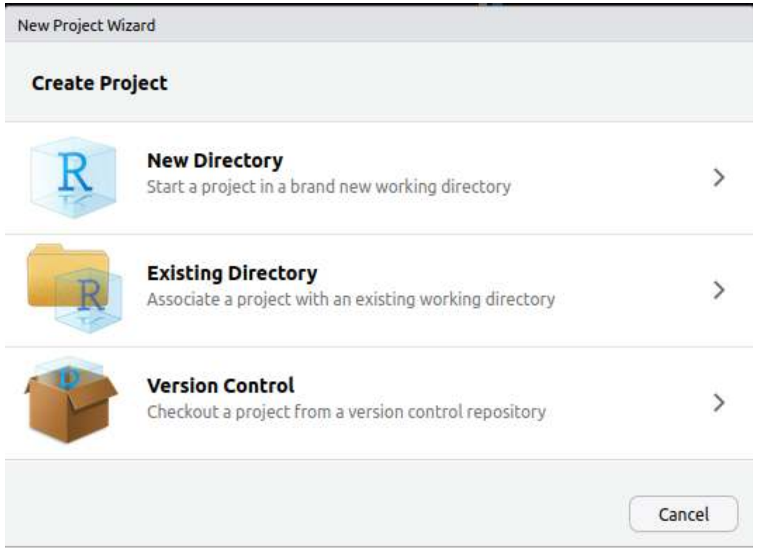
Ever had the following expression when people ask you 8 months later "*Where is that bit of analysis you did for me*": 🤔.

We want to avoid feeling like that by keeping all our *notes*, *scripts*, *data* and *output* in one single place. This is where Rstudio makes it easy by creating a project.

Start by creating a folder in your `home` directory called `projects` and starting a project called `markdown`:

```
hanjo@optimus:~$ mkdir -p projects/data_analysis
```

- Next click on the menu: `File > New Project`



Using Projects

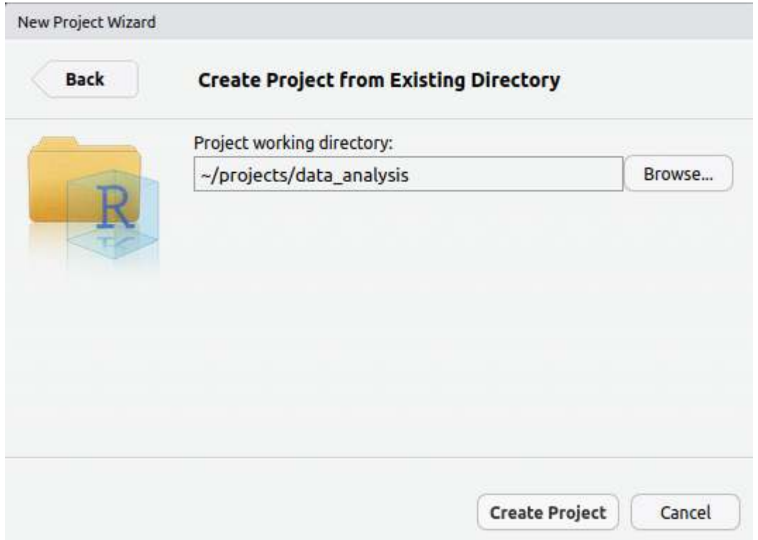
Ever had the following expression when people ask you 8 months later "*Where is that bit of analysis you did for me*": 🤔.

We want to avoid feeling like that by keeping all our *notes*, *scripts*, *data* and *output* in one single place. This is where Rstudio makes it easy by creating a project.

Start by creating a folder in your `home` directory called `projects` and starting a project called `markdown`:

```
hanjo@optimus:~$ mkdir -p projects/data_analysis
```

- Next click on the menu: `File > New Project`
- Then select the path `projects/data_analysis` as your project folder and click `Create Project`



Using Projects

Beyond having a dedicated work environment for you project, projects also have other advantages.

The biggest one of them all is *relative paths*. Ever get a document from someone and they have a link in their document, but it says something like `/Documents/Hanjo/my_work/data/data.csv` and now the link no longer works on your computer.

What Rstudio does is anchor the link from the project directory. So if I ever send Chris my `markdown` project, and the data is stored in `data/data.csv`, it will work on both myself and Chris' computer.

Create the following folder structure in your new project.

```
.
└── data_analysis
    ├── scripts
    ├── output
    └── data
        ├── raw
        └── processed
```

05:00

Software for analysis

We are also going to install some R packages to ensure that Rstudio can render our lab-books to both PDF and HTML.

write each of these lines in the command-line console of Rstudio and press `enter`. We will be diving deeper in the `R` universe later in this course. For now, just follow along with how I do it.

```
install.packages("rmarkdown")
install.packages("knitr")
install.packages(c("tinytex", "usethis", "rmdformats", "prettydoc"))
```

10:00

Last, but not least...

> Making your Rstudio look cool for you!

Take some time and go into preferences to choose your default color scheme that suites you. OR

Customize your own theme:

```
https://tmtheme-editor.herokuapp.com/#!/editor/theme/Monokai
```

Markdown 🧙

What is markdown?

R Markdown wizard monsters creating a R Markdown document from a recipe. Art by Allison Horst

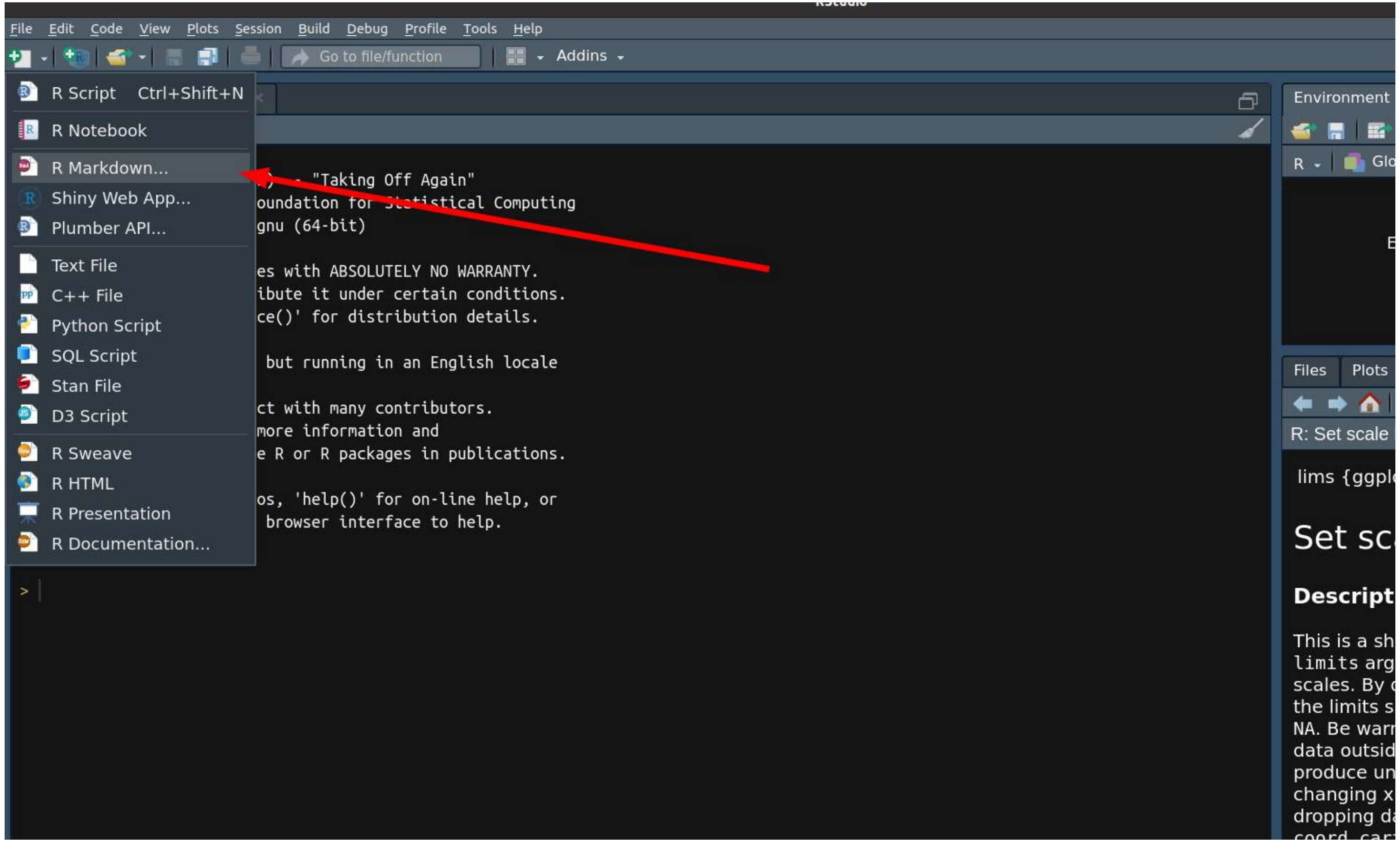
What is markdown?

> Markdown is a lightweight markup language for creating formatted text using a plain-text editor. *John Gruber* and *Aaron Swartz* created Markdown in 2004 as a markup language that is appealing to human readers in its source code form. Markdown is widely used in blogging, instant messaging, online forums, collaborative software, documentation pages, and readme files.
>
> — *Wikipedia*

- Abstraction layer *above* certain compiling formats such as PDF, HTML, Word (XML).
  - This is pretty cool as you only have to learn the very basic syntax of markdown to be able to convert your document to any of the formats.
- `Rstudio` uses a productive notebook interface (called *Rmarkdown*) to weave together narrative text and code to produce elegantly formatted output.
  - Great thing is it supports over 51 languages. Main ones are `R`, `python`, `shell` and `SQL`.

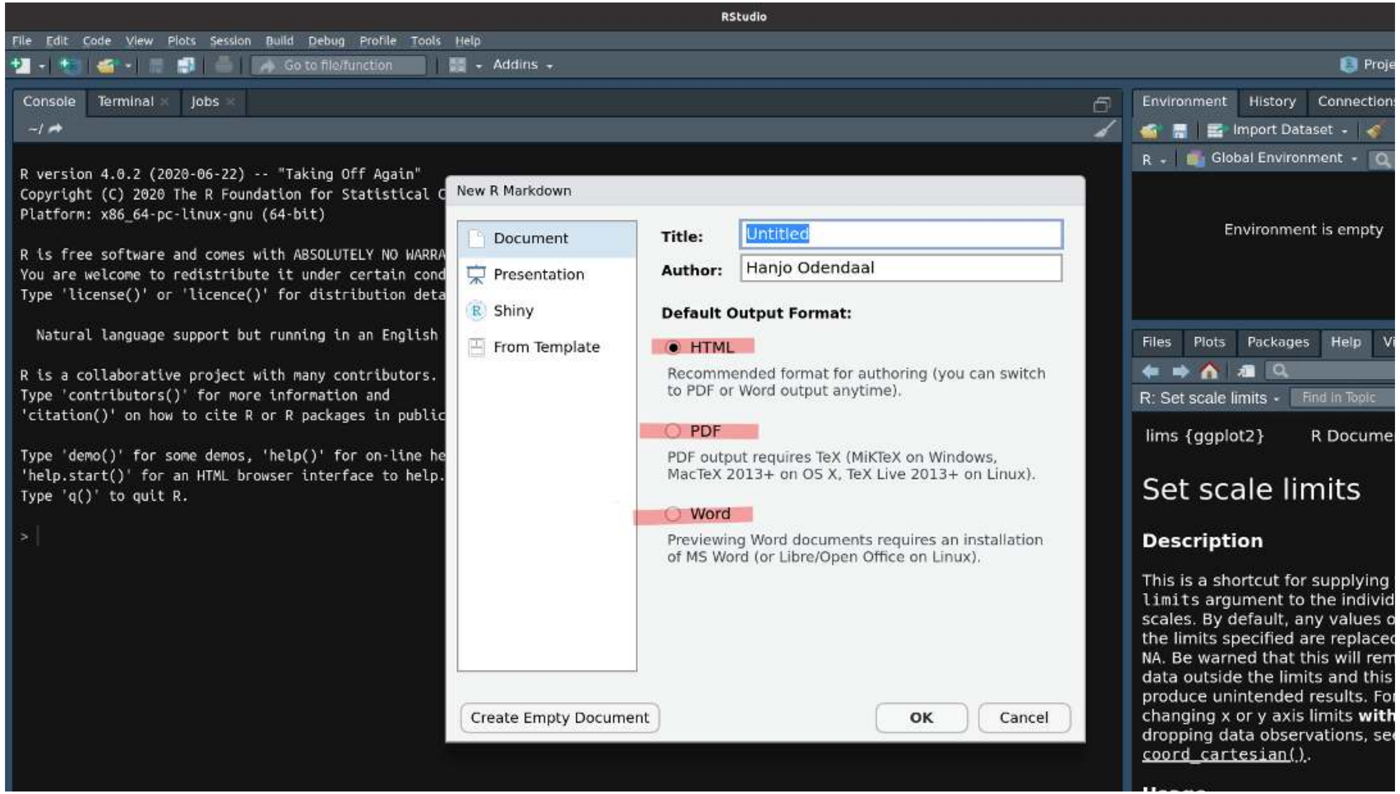
Understanding markdown in Rstudio

- Start by opening a new *Rmarkdown* file (`.rmd`) in your `data_analysis project`.



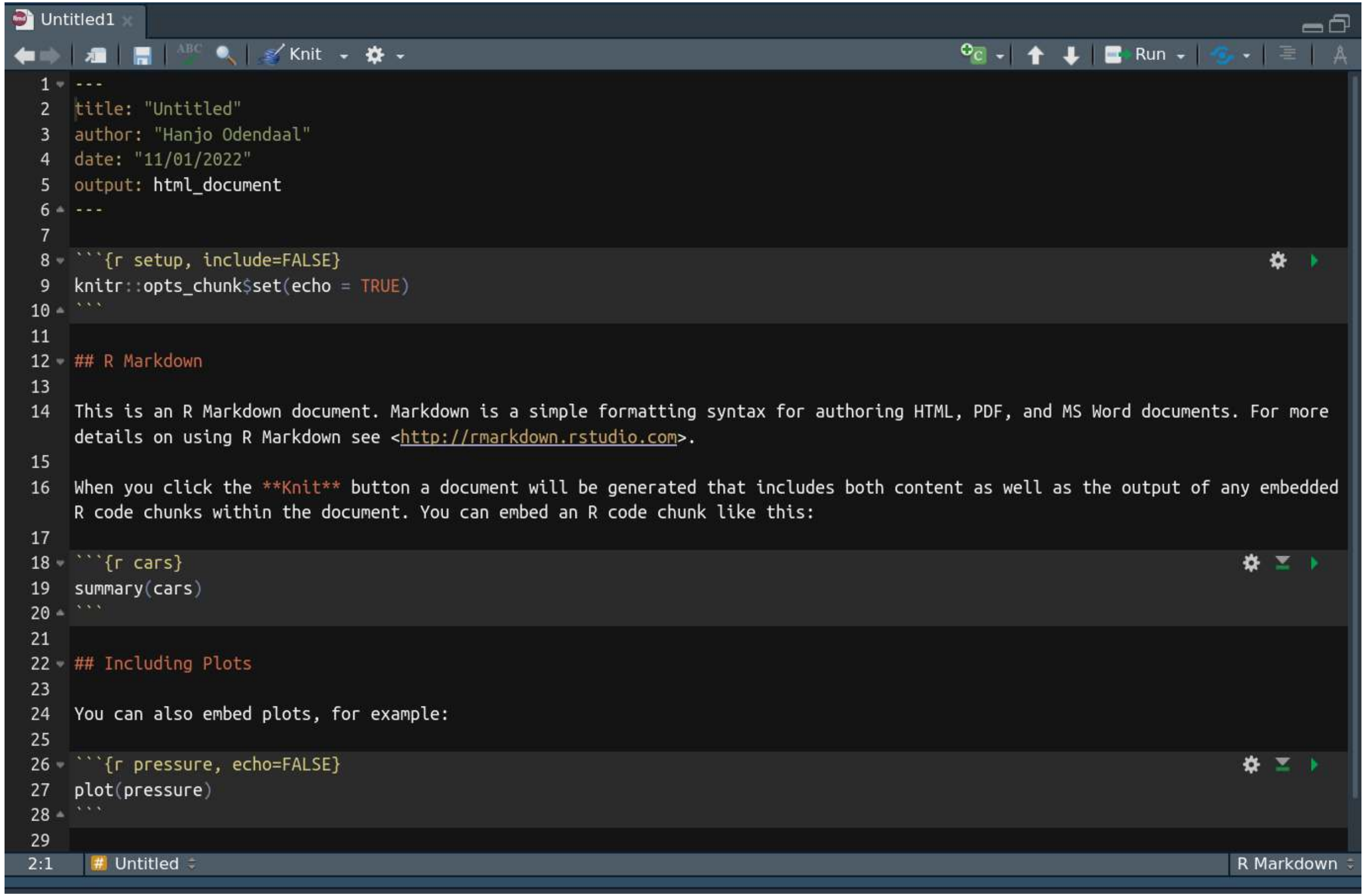
Understanding markdown in Rstudio

- Start by opening a new *Rmarkdown* file (`.rmd`) in your `markdown project`.

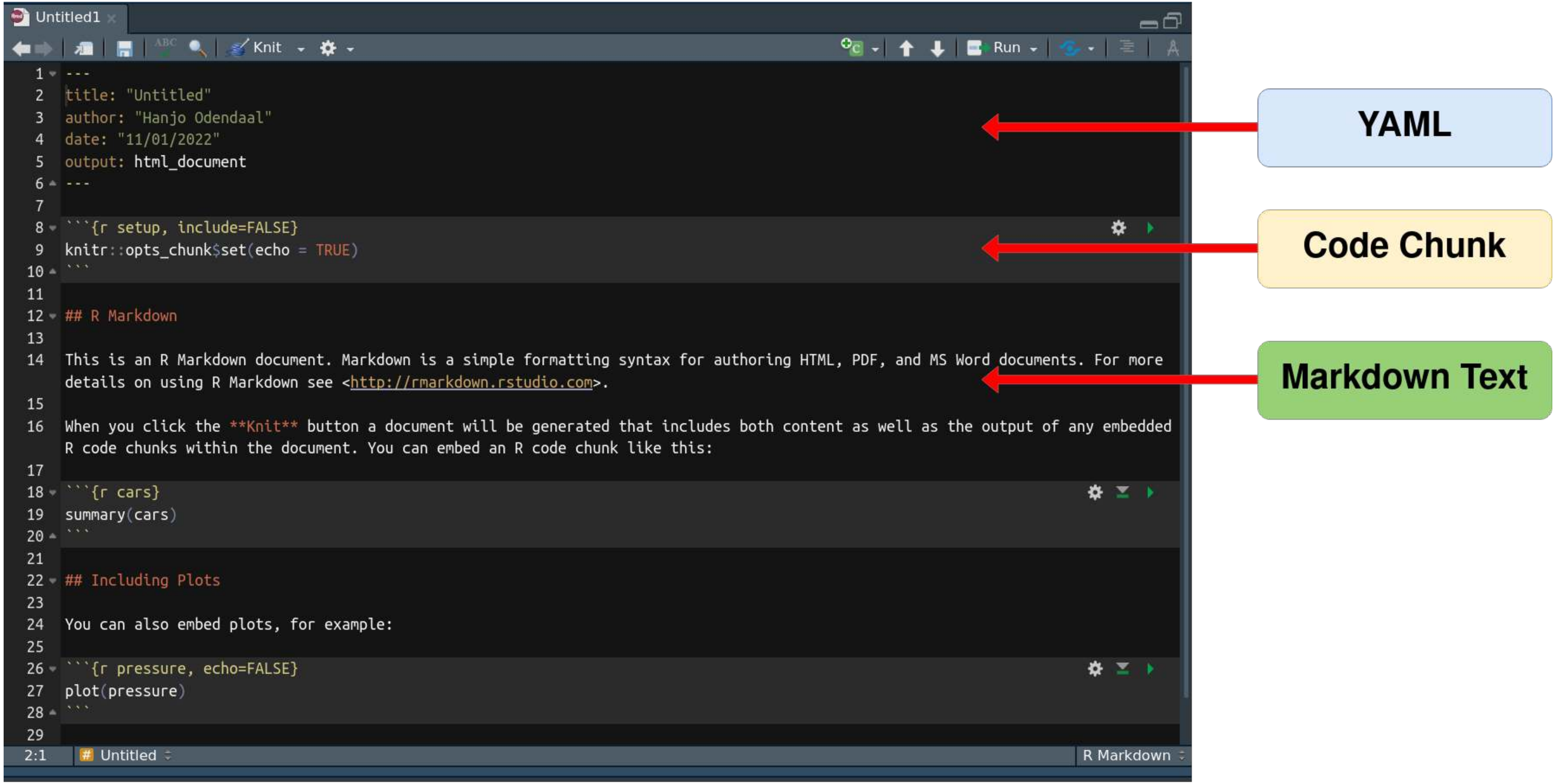


Understanding markdown in Rstudio

- Start by opening a new *Rmarkdown* file (`.rmd`) in your `markdown` project.



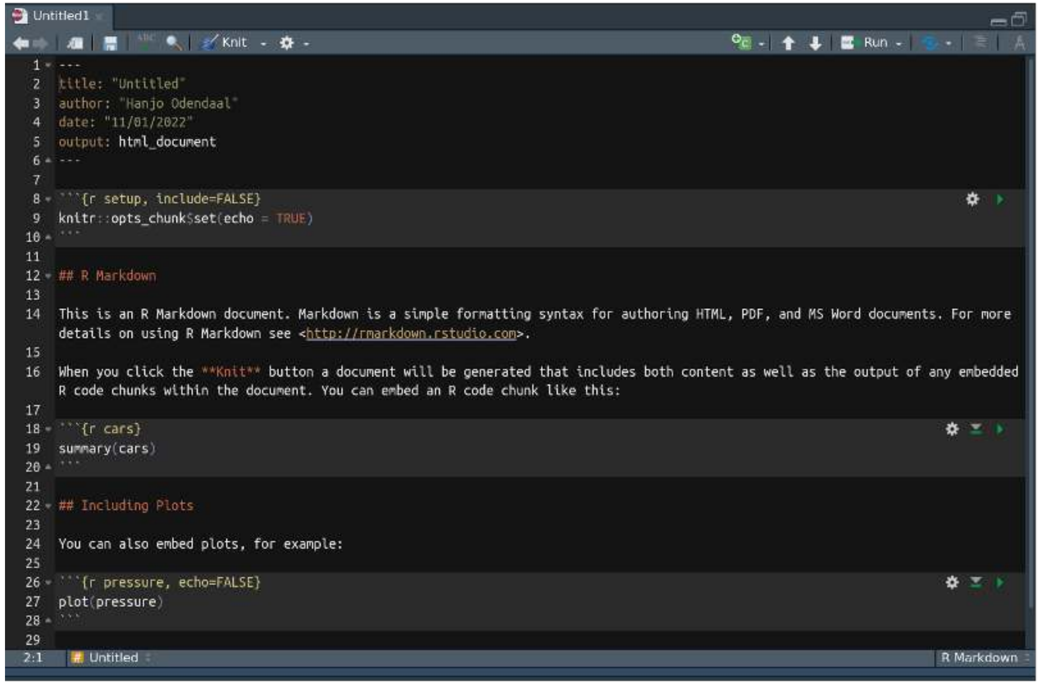
Components of markdown

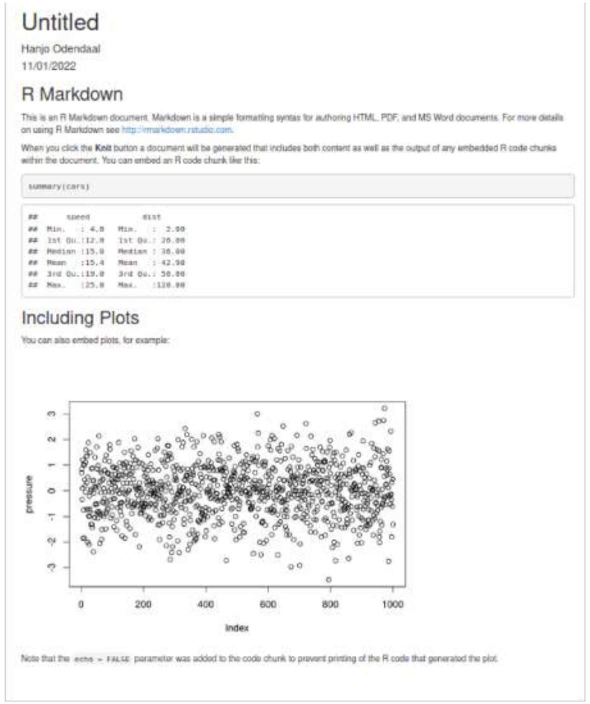


Understanding markdown in Rstudio

We need to `knit` our documents in order to produce the output.

- Save your `.rmd` document in your folder as `README.rmd`.
- Next, press the `knit` button at the top OR (be cool) and use `CTRL + SHIFT + k`!





Components of markdown: YAML

> YAML: YAML Ain't Markup Language

The *YAML* component specifies the metadata of the file:

- Type of output to produce
- Formatting preferences of things like tables
- Other metadata such as document title, author, and date.

YAML is dependent on indentation so be careful:

```
---
title: "My cool document"
author: "Hanjo Odendaal"
date: "11/01/2022"
output: html_document
---
```

Components of markdown: Code Chunks

Code Chunks are the sections of the document where you will write your code that you wish to include into your document.

For now, we will only use the code chunks as a documentation tool for any code that we write. Later on in the course we will actually be executing the code to produce tables and plots in a document!

```
```{r pressure, echo=FALSE}
pressure <- rnorm(1000)
plot(pressure)
```
```

Each chunk is opened with a line that starts with three back-ticks, and curly brackets that contain parameters for the chunk (`{ }`). The chunk ends with three more back-ticks.

😉 use shortcut (`CTRL + ALT + i`) to open chunk

Components of markdown: Code Chunks

What do we mean by parameters in the `{}` brackets? Lets start with the programming language specification.

- They start with `r` to indicate that the language name within the chunk is `R` (we can also do `python` or `sql` etc.)
- After the `r` you can optionally write a chunk "name" - good practice for debugging later on

The curly brackets can include other options too, written as `tag`=`value`, such as:

- `eval = FALSE` to not run the R code.
- `echo = FALSE` to not print the chunk's `R` source code in the output document.
- `warning = FALSE` to not print warnings produced by code.
- `message = FALSE` to not print any messages produced by code.
- `include = TRUE/FALSE` whether to include chunk outputs (e.g. plots) in the document.
- `out.width` and `out.height` provide in style `out.width = "75%"`.
- `fig.align = "center"` adjust how a figure is aligned across the page.
- `fig.show='hold'` if your chunk prints multiple figures and you want them printed next to each other (pair with `out.width = c("33%", "67%")`. Can also set `animate` to concatenate multiple into an animation.

Components of markdown: Markdown Text

> Markdown Text is what makes using it as a lab-book (and writing journal articles) so versatile.

📔 Would you believe that these slides were all made in using `Rmarkdown`?

So lets start with some basics: *Headings* and *Formatting*

```
# Header 1

## Header 2

### Header 3
```

So *how* would **this** text `look`?

```
So _how_ would **this** text `look`?
```

Components of markdown: Markdown Text

Unordered list items start with `*`, `-`, or `+`, and you can nest one list within another list by indenting the sub-list:

```
- Fruits
- Vegtables
  * Carrot
  * Spinach
```

- Fruits
- Vegtables
    - Carrot
    - Spinach

```
1. Dog
     - German Shepherd #(two spaces)
     - Belgian Shepherd #(two spaces)
2. Cat
     - Siberian  #(two spaces)
     - Siamese  #(two spaces)
```

1. Dog
    - German Shepherd #(two spaces)
    - Belgian Shepherd #(two spaces)
2. Cat
    - Siberian #(two spaces)
    - Siamese #(two spaces)

Your turn!

> Can you produce the following document?

My first Markdown

Hanjo Odendaal

11/01/2022

About me

My name is *Hanjo Odendaal* and I am a **Principal** data scientist at 71point4.

My favourite food is:

- Steak & Salad

Coding languages

I code in

- `R`, `SQL` and `python`

10:00

Changing formats

- How does the following code affect your output?

```
---
title: "My first Markdown"
author: "Hanjo Odendaal"
date: "11/01/2022"
output: pdf_document
---
```

- Lets change the output to a `Word` document:

```
---
title: "My first Markdown"
author: "Hanjo Odendaal"
date: "11/01/2022"
output: word_document
---
```

Using advanced YAML

If we are *knitting* a document to `html` there are a couple of really cool things we can do in terms of formatting.

- How does the following code affect your output?
- All available themes: "cerulean", "cosmo", "flatly", "journal", "lumen", "paper", "readable", "sandstone", "simplex", "spacelab", "united", and "yeti".

```
---
title: "Your title here"
date: "Todays date"
output:
  html_document:
    theme: journal
    highlight: espresso
    toc: true
    toc_depth: 4
    toc_float: true
    code_folding: show
---
```

Using advanced YAML

We can also combine our outputs:

- Some of the `YAML` options might not be available for when you want to switch between formats. (example `PDF` does not take `theme` as a parameter)
- To account for those differences, we split up the yaml parameters between the different formats.

```
---
title: "Your title here"
date: "Todays date"
output:
  pdf_document:
    highlight: espresso
    toc: true
    toc_depth: 4
  html_document:
    theme: journal
    toc: false
    highlight: haddock
---
```

Your turn!

> Create the following output with a theme and format of your choice.

⚠️ Remember to use chunk option `eval = FALSE` & `echo = TRUE` to ensure code **doesn't** run, but is displayed.

SQL

Example

Time to start to Code!

Code ▾

2022-01-11

SQL

`SQL` stands for *structured query language*.

It is one of the most widely used `coding` languages in the world and most people using data on a day-to-day basis should use it.

Example

The following is an example of a `SQL` statement:

Hide

```
SELECT * FROM transactions LIMIT 10;
```

20:00

Quarto is the new thing ;-)

Quarto is a multi-language, next generation version of R Markdown from RStudio, with many new features and capabilities.

```
---
title: "ggplot2 demo"
author: "Norah Jones"
date: "5/22/2021"
format:
  html:
    code-fold: true
---

## Air Quality

@fig-airquality further explores the impact of temperature on ozone level.

'```{r}
#| label: fig-airquality
#| fig-cap: Temperature and ozone level.
#| warning: false

library(ggplot2)

ggplot(airquality, aes(Temp, Ozone)) +
```

From Excel to R
(Session 1-3 - R Basics)

Getting Started With R

Basics

I will illustrate how R basically thinks and how you should *think* in R with an example.

Note this is what we call *base* R coding. Other more optimized packages like `dplyr`, have different notations. You should, however, be able to understand base R and at a later stage the more advanced libraries will be explored.

Basics

R uses columns and arrays in order to define data frames. These can be adjusted (as will be seen) to tell R whether your data is a time-series, panel, or whichever format intended.

Type the following code to create a set in R: **(Remember: R is case sensitive!)**

```
R ← c("Very Happy", "Happy", "Not Happy")
# Let's now create responses:
W ← c(15,5,3)
M ← c(35,15,14)
C ← c(23,35,32)
```

- The function `c` is just concatenate the vector.
- Here we are creating: `character` and `numeric` vector.

So we *assign* a vector to a *variable*. Do you think my variable names are good?

Basics

Now we have many variables assigned names, but we now want to concatenate it all... i.e. let's merge the columns together in a single `data.frame` (as a single unit) - Excel Spreadsheet.

To change the column names, simply type the name first:

```
HappySurvey <- data.frame(Responses = R, Women = W, Men = M, Children = C)
```

Now to isolate a column, say Men, and count the responses, use the $ sign. Note the following syntax to access a column:

```
sum(HappySurvey$Men)

x <- HappySurvey$Men
x <- HappySurvey[,3]   # calling all rows of column 3
```

Basics

Other useful base R commands include:

```
mean(x)
min(x)
median(x)
summary(x)
```

> Congratulations, that's your first successful command in R...

If you are using a package or base R functions, and you do not know what the inputs are: do the following:

- Type `?` before the command to get info in the Help page in Rstudio (or ??xxx for internet help)
- Type the command, e.g.: `chisq.test` ; add brackets `chisq.test()` ; within the brackets type CTRL + SPACE.
    - You now see the possible inputs to the function (some are required, others may have defaults).

Exercises

- Create this string

```
example_string <- "This string is 33 characters long"
```

- What is the length of the string?

```
length(example_string)
nchar(example_string)
```

- We can use length() to see the number of elements in a vector

```
length(W)
```

Data Structures

Vectors

As we have been exploring, `R` contains several object structures: we can store and operate on a bit of data by placing it in a particular structure called a `vector`. Vectors are collections of one or more elements of data of the **same** type.

```
my_numbers <- c(1, 3, 4, 5:10)
sqrt(my_numbers)
```

```
## [1] 1.000000 1.732051 2.000000 2.236068 2.449490 2.645751 2.828427 3.000000
## [9] 3.162278
```

```
is.vector(my_numbers)
```

```
## [1] TRUE
```

```
length(my_numbers)
```

```
## [1] 9
```

Vectors

Just as the same as a *numeric* vector, we can create a *character* vectors and apply functions to it:

```
my_names <- c("jill", "jack", "chris", "hanjo", "tivan", 400)
toupper(my_names)
```

```
## [1] "JILL"  "JACK"  "CHRIS" "HANJO" "TIVAN" "400"
```

```
nchar(my_names)
```

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

> Can you notice what happens when we mix characters and numerics in a vector?

Vectors

Accessing the *elements* in a *vector* we use `[`. Unlike `Python`, `R` uses 1 base, not zero.

```
my_numbers ← c(1, 3, 4, 5:10)
my_numbers[0]
```

```
## numeric(0)
```

```
my_numbers[1]
```

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

```
my_names ← c("jill", "jack", "chris", "hanjo", "tivan", 400)
my_names[c(1, 5)]
```

```
## [1] "jill"  "tivan"
```

Lists

Can be considered to be an bit more of an advanced storing method as it has to do with how the list object stores data. The best thing about list though is that you can store different kinds of data in a list object. They do not have to conform to a structure as with `arrays` and `data.frames`. Thus you will encounter them a lot of the times when you are working with R functions from packages

```
Odendaal ← list(name = "Hanjo",
                 title = "R master Joda",
                 subject = "R training",
                 university = "Stellenbosch"
                 salary = "$10 million Zim")
```

```
Odendaal
```

```
## $name
## [1] "Hanjo"
##
## $title
## [1] "R master Joda"
##
## $subject
## [1] "R training"
##
## $university
## [1] "Stellenbosch"
##
## $salary
## [1] "$10 million Zim"
```

Lists

The Important thing about accessing lists, is that the syntax uses `[[]]` types of brackets:

```r
Odendaal[['name']]
```

```
## [1] "Hanjo"
```

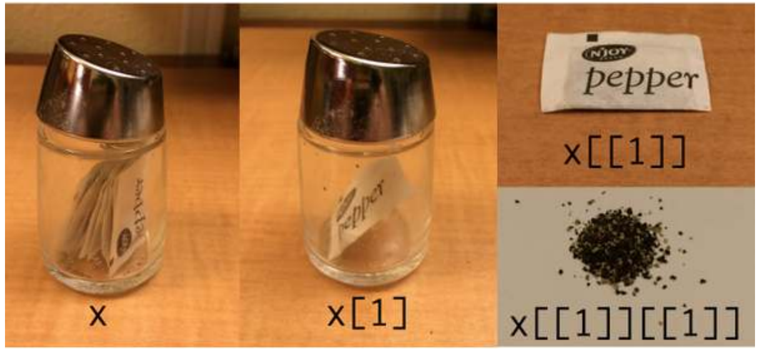
```r
Odendaal[[1]]
```

```
## [1] "Hanjo"
```

```r
Odendaal[c(2,3)]
```

```
## $title
## [1] "R master Joda"
##
## $subject
## [1] "R training"
```

One of the best explanations of what a list is and how to access them was tweeted once by Hadley Wickham:



Data Frames

So `R`'s vectors are one-dimensional with *n* length, BUT `data frames` are the bread and butter of `R` and allows for storing data in both rows and columns. This makes the `data frame` the R equivalent of an Excel spreadsheet.

As opposed to a vector, a data frame is a two-dimensional (and even `n`-dimensional) data structure where records in each column are of the same `class` and all columns are of the same length.

```
data.frame(
  class_marks = c(1:5),
  people = letters[1:5],
  attended = c(TRUE, FALSE, TRUE, TRUE, TRUE)
)
```

```
##   class_marks people attended
## 1           1      a     TRUE
## 2           2      b    FALSE
## 3           3      c     TRUE
## 4           4      d     TRUE
## 5           5      e     TRUE
```

Data frames are cool... but `tibbles` are *next* level

Functions

The more and more advanced you get, you will not only be using functions, but start to write your own.

Lets think of a basic function... perhaps we want to multiply a number by itself for some reason and round the number to two decimals. How would you do that?

```
normal_dist <- rnorm(100)

multiply <- function(x){
  res <- round(x*x, 2)

  return(res)
}

head(normal_dist)
```

```
## [1] 0.2918109 1.4738248 1.9730454 0.4633440 0.0411252 0.9137999
```

```
head(multiply(normal_dist))
```

```
## [1] 0.09 2.17 3.89 0.21 0.00 0.84
```

Exercises

- Create the following data frames & lists

```r
stock_data ← data.frame(
  crops = c("maize", "soya", "rice", "potato
  quantity_ordered = c(100, 200, 38, 1050),
  price_per_kg = c(1000, 1855.99, 99.50, 500
  in_stock = c(TRUE, TRUE, FALSE, TRUE)
)
```

```r
staff_data ←  data.frame(
  names = c("Jean de Dieu", "Martha"),
  monday = c(TRUE, TRUE),
  tuesday = c(TRUE, TRUE),
  wednesday = c(FALSE, TRUE),
  thursday = c(FALSE, TRUE),
  friday = c(TRUE, TRUE)
)
```

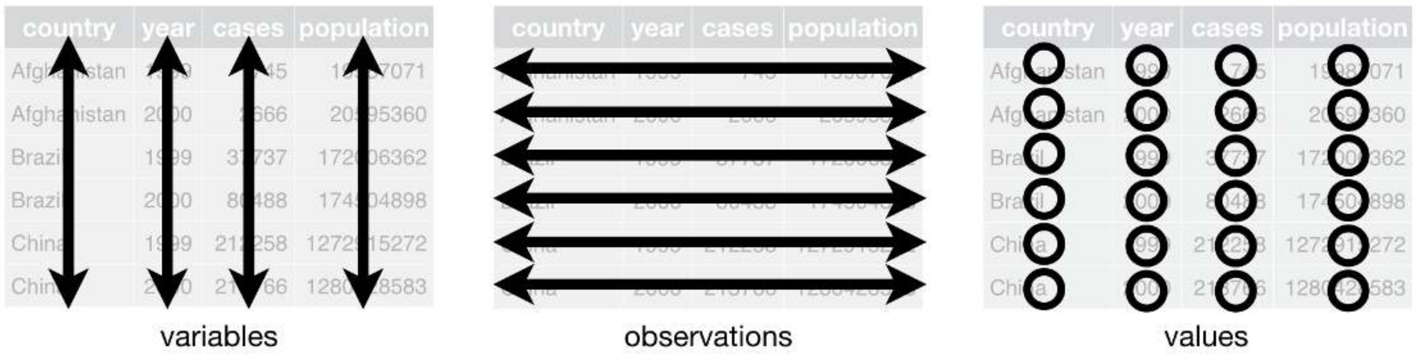
```r
shop_1_list ← list(
  active = TRUE,
  stock = stock_data,
  staff = staff_data
)

shop_2_list ← list(
  active = FALSE
)

shops_list ← list(
  shop_1 = shop_1_list,
  shop_2 = shop_2_list
)
```

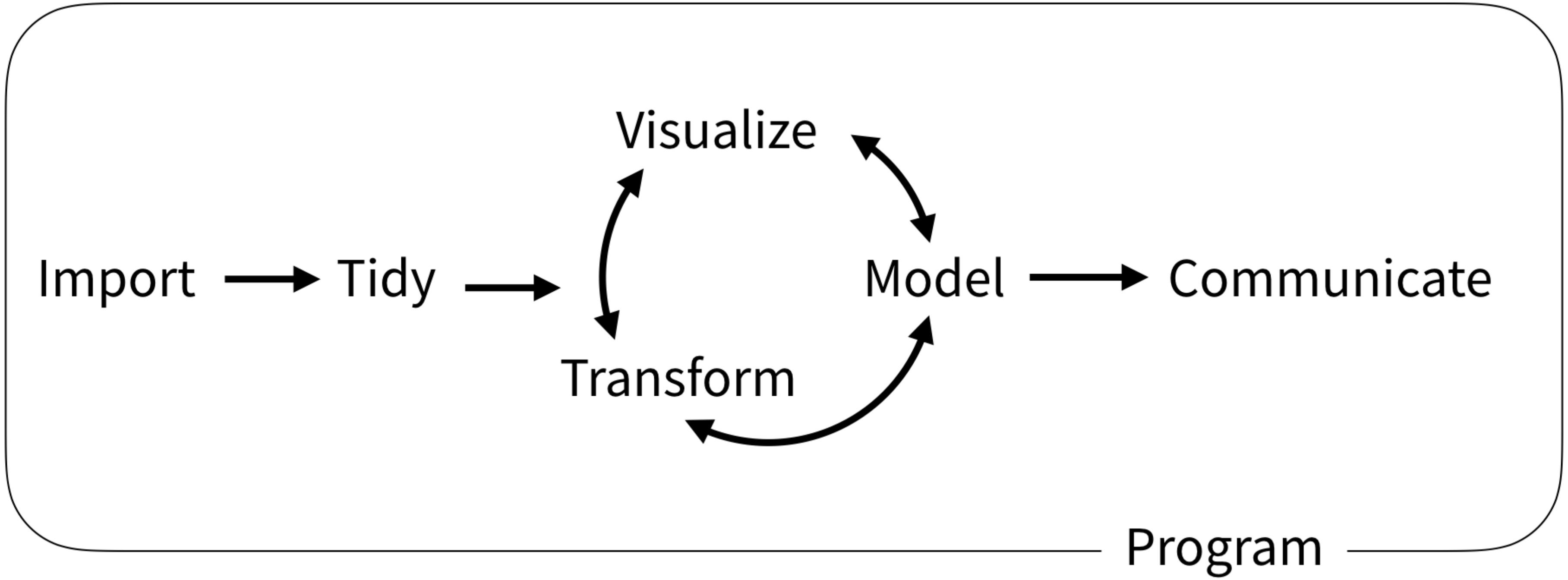
Tidy Data

Tidy data is a standard way of mapping the meaning of a dataset to its structure. A dataset is messy or tidy depending on how rows, columns and tables are matched up with observations, variables and types. In tidy data: Every column is a variable. Every row is an observation.

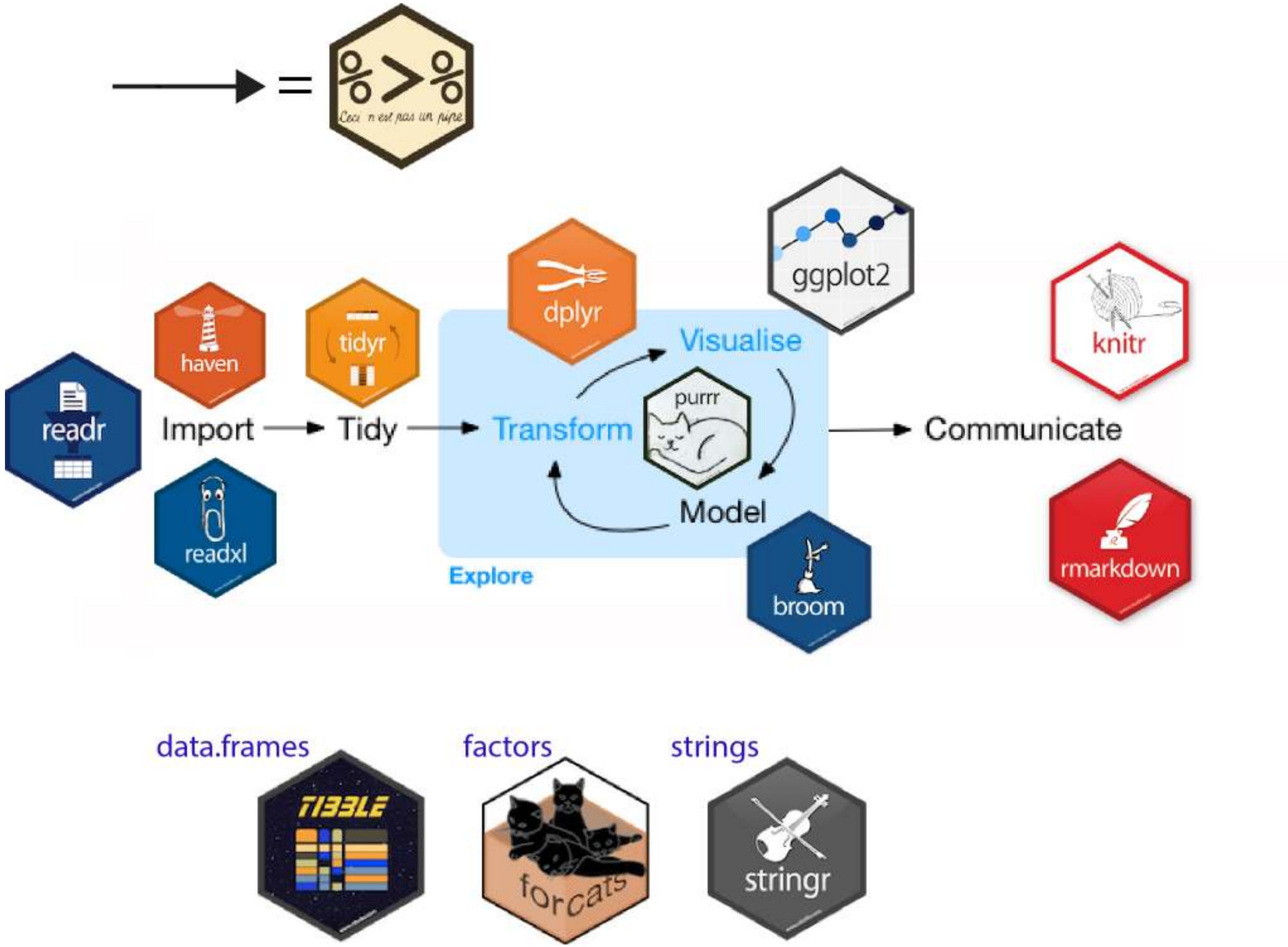


Unleash Tidyverse 🧙

Data Analysis Workflow



Tidyverse Packages



Packages

Lets load the packages!

```
library(tidyverse)

# ── Attaching packages ─────────────── tidyverse 1.3.0 ──
# ✔ ggplot2 3.3.5     ✔ purrr   0.3.4
# ✔ tibble  3.1.6     ✔ dplyr   1.0.8
# ✔ tidyr   1.1.3     ✔ stringr 1.4.0
# ✔ readr   1.3.1     ✔ forcats 0.5.0
# ── Conflicts ────────────────── tidyverse_conflicts() ──
# ✖ dplyr::filter() masks stats::filter()
# ✖ dplyr::lag()    masks stats::lag()
```

Now, lets create the same `data frame` of earlier, but using a tibble!

Tibble

`Tibbles` have amazing properties such as pretty print, showing you the *class* of the column and not creating `factors` for characters.

```
df ← tibble(
  class_marks = c(1:5),
  people = letters[1:5],
  attended = c(TRUE, FALSE, TRUE, TRUE, TRUE)
)

df
```

```
## # A tibble: 5 × 3
##   class_marks people attended
##         <int> <chr>  <lgl>
## 1           1 a      TRUE
## 2           2 b      FALSE
## 3           3 c      TRUE
## 4           4 d      TRUE
## 5           5 e      TRUE
```

Exploring data frames

To start, lets load a dataset using the `read_csv` function. One of the cool things about reading data is that we can either read local data OR we can load data straight from the internet!

```
worldcup ← read_csv("data/worldcup.csv")
```

```
## Rows: 21 Columns: 10
## ── Column specification ────────────────────────────────────────────────────────
## Delimiter: ","
## chr (5): host, winner, second, third, fourth
## dbl (5): year, goals_scored, teams, games, attendance
##
## ℹ Use `spec()` to retrieve the full column specification for this data.
## ℹ Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

Exploring data frames

The `glimpse()` function is a way to print several records of the data frame, along with its column names and class:

```
glimpse(worldcup)
```

```
## Rows: 21
## Columns: 10
## $ year         <dbl> 1930, 1934, 1938, 1950, 1954, 1958, 1962, 1966, 1970, 197…
## $ host         <chr> "Uruguay", "Italy", "France", "Brazil", "Switzerland", "S…
## $ winner       <chr> "Uruguay", "Italy", "Italy", "Uruguay", "West Germany", "…
## $ second       <chr> "Argentina", "Czechoslovakia", "Hungary", "Brazil", "Hung…
## $ third        <chr> "USA", "Germany", "Brazil", "Sweden", "Austria", "France"…
## $ fourth       <chr> "Yugoslavia", "Austria", "Sweden", "Spain", "Uruguay", "W…
## $ goals_scored <dbl> 70, 70, 84, 88, 140, 126, 89, 89, 95, 97, 102, 146, 132, …
## $ teams        <dbl> 13, 16, 15, 13, 16, 16, 16, 16, 16, 16, 16, 24, 24, 24, 2…
## $ games        <dbl> 18, 17, 18, 22, 26, 35, 32, 32, 32, 38, 38, 52, 52, 52, 5…
## $ attendance   <dbl> 434000, 395000, 483000, 1337000, 943000, 868000, 776000, …
```

Exploring data frames

There is also a really nice function from the `skimr` package. We *don't* need to load a package to use one if its functions.

```
skimr::skim(worldcup)
```

```
── Data Summary ────────────────────────
                           Values
Name                       adult_data
Number of rows             21
Number of columns          10
_______________________
Column type frequency:
  character                5
  numeric                  5
________________________
Group variables            None

── Variable type: character ──────────────────────────────────────────────────────
  skim_variable n_missing complete_rate min max empty n_unique whitespace
1 host                  0             1   3  18     0       16          0
2 winner                0             1   5  12     0        9          0
3 second                0             1   5  14     0       11          0
4 third                 0             1   3  12     0       15          0
5 fourth                0             1   5  12     0       16          0

── Variable type: numeric ────────────────────────────────────────────────────────
  skim_variable n_missing complete_rate     mean       sd     p0    p25     p50     p75    p100 hist
1 year                  0             1    1977.     26.7   1930   1958    1978    1998    2018
2 goals_scored          0             1     121.     33.9     70     89     126     146     171
3 teams                 0             1     21.8     7.46     13     16      16      32      32
4 games                 0             1     42.9     17.5     17     32      38      64      64
5 attendance            0             1 1898122. 1027950. 395000 943000 1774022 2724604 3568567
```

Writing data frames

Given that we used `read_csv` to read in the data, I think it speaks for itself that we will use `write_csv` to write the data to a csv. We can also use `write_delim` if you want to change the delimiter.

```
write_csv(worldcup, "output/worldcup.csv")
write_delim(worldcup, "output/worldcup.csv", delim = "|")
```

Exercises

- Write out a csv for me of the first 3 columns into the folder: `output/world_cup_winners.csv`

Data manipulation

Manipulating objects with dplyr

Art by Allison Horst

Manipulating objects with dplyr

We can use the dplyr library to manipulate the data using very basic functions:

- `select`: Selects specific columns by name.
- `filter`: Filter data based on certain criteria.
- `mutate`: Create a new column.
- `group_by`: Column to aggregate on.
- `summarise`: How do you want to summarise the data?

In `R` we gonna *chain* these commands using whats called the `pipe` operator: `%>%`. The shortcut to print this symbol is: `Ctrl + Shift + m`.

> We read this `%>%` symbol as: *and then*

- So for instance `worldcup %>% select(winner)` reads in english as: Take object `worldcup` *and then* select column `winner`.
- Another would be `worldcup %>% filter(games < 22)`: Take object `worldcup` *and then* filter out rows where `games` is more than 22.

select()

Extract columns by name: `select(.data, ... )`

```
worldcup %>% select(winner, goals_scored, attendance)
```

```
## # A tibble: 21 × 3
##    winner       goals_scored attendance
##    <chr>               <dbl>      <dbl>
##  1 Uruguay                70     434000
##  2 Italy                  70     395000
##  3 Italy                  84     483000
##  4 Uruguay                88    1337000
##  5 West Germany          140     943000
##  6 Brazil                126     868000
##  7 Brazil                 89     776000
##  8 England                89    1614677
##  9 Brazil                 95    1673975
## 10 West Germany           97    1774022
## # ℹ 11 more rows
```

select()

These helpers select variables by matching patterns in their names:

- `:` for selecting a range of consecutive variables.
- `!` for taking the complement of a set of variables.

- `c()` for combining selections.

- `starts_with()`: Starts with a prefix.

- `ends_with()`: Ends with a suffix.
- `contains()`: Contains a literal string.
- `matches()`: Matches a regular expression.
- `num_range()`: Matches a numerical range like x01, x02, x03.

- `where()`: Applies a function to all variables and selects those for which the function returns `TRUE`.

- `&` and `|` for selecting the intersection or the union of two sets of variables.

select()

- `:` for selecting a range of consecutive variables.

```
worldcup %>% select(winner:goals_scored)
```

```
## # A tibble: 21 × 5
##    winner       second         third        fourth        goals_scored
##    <chr>        <chr>          <chr>        <chr>                <dbl>
##  1 Uruguay      Argentina      USA          Yugoslavia              70
##  2 Italy        Czechoslovakia Germany      Austria                 70
##  3 Italy        Hungary        Brazil       Sweden                  84
##  4 Uruguay      Brazil         Sweden       Spain                   88
##  5 West Germany Hungary        Austria      Uruguay                140
##  6 Brazil       Sweden         France       West Germany           126
##  7 Brazil       Czechoslovakia Chile        Yugoslavia              89
##  8 England      West Germany   Portugal     Soviet Union            89
##  9 Brazil       Italy          West Germany Uruguay                 95
## 10 West Germany Netherlands    Poland       Brazil                  97
## # ℹ 11 more rows
```

select()

- `!` for taking the complement of a set of variables.

```
worldcup %>% select(!(winner:goals_scored))
```

```
## # A tibble: 21 × 5
##     year host        teams games attendance
##    <dbl> <chr>       <dbl> <dbl>      <dbl>
##  1  1930 Uruguay        13    18     434000
##  2  1934 Italy          16    17     395000
##  3  1938 France         15    18     483000
##  4  1950 Brazil         13    22    1337000
##  5  1954 Switzerland    16    26     943000
##  6  1958 Sweden         16    35     868000
##  7  1962 Chile          16    32     776000
##  8  1966 England        16    32    1614677
##  9  1970 Mexico         16    32    1673975
## 10  1974 Germany        16    38    1774022
## # ℹ 11 more rows
```

select()

- `starts_with()`: Starts with a prefix.

```
worldcup %>% select(starts_with("goals_"))
```

```
## # A tibble: 21 × 1
##    goals_scored
##           <dbl>
##  1           70
##  2           70
##  3           84
##  4           88
##  5          140
##  6          126
##  7           89
##  8           89
##  9           95
## 10           97
## # ℹ 11 more rows
```

select()

- `ends_with()`: Ends with a suffix.

```
worldcup %>% select(ends_with("_scored"))
```

```
## # A tibble: 21 × 1
##    goals_scored
##           <dbl>
##  1           70
##  2           70
##  3           84
##  4           88
##  5          140
##  6          126
##  7           89
##  8           89
##  9           95
## 10           97
## # ℹ 11 more rows
```

select()

- `contains()`: Contains a literal string.

```
worldcup %>% select(contains("s"))
```

```
## # A tibble: 21 × 5
##    host        second         goals_scored teams games
##    <chr>       <chr>                 <dbl> <dbl> <dbl>
##  1 Uruguay     Argentina                70    13    18
##  2 Italy       Czechoslovakia           70    16    17
##  3 France      Hungary                  84    15    18
##  4 Brazil      Brazil                   88    13    22
##  5 Switzerland Hungary                 140    16    26
##  6 Sweden      Sweden                  126    16    35
##  7 Chile       Czechoslovakia           89    16    32
##  8 England     West Germany             89    16    32
##  9 Mexico      Italy                    95    16    32
## 10 Germany     Netherlands              97    16    38
## # ℹ 11 more rows
```

select()

- `matches()`: Matches a regular expression.

```
worldcup %>% select(matches(".*s$"))
```

```
## # A tibble: 21 × 2
##     teams games
##     <dbl> <dbl>
##  1     13    18
##  2     16    17
##  3     15    18
##  4     13    22
##  5     16    26
##  6     16    35
##  7     16    32
##  8     16    32
##  9     16    32
## 10     16    38
## # ℹ 11 more rows
```

select()

- `where()`: Applies a function to all variables and selects those for which the function returns `TRUE`.

```
worldcup %>% select(where(is.numeric))
```

```
## # A tibble: 21 × 5
##     year goals_scored teams games attendance
##    <dbl>        <dbl> <dbl> <dbl>      <dbl>
##  1  1930           70    13    18     434000
##  2  1934           70    16    17     395000
##  3  1938           84    15    18     483000
##  4  1950           88    13    22    1337000
##  5  1954          140    16    26     943000
##  6  1958          126    16    35     868000
##  7  1962           89    16    32     776000
##  8  1966           89    16    32    1614677
##  9  1970           95    16    32    1673975
## 10  1974           97    16    38    1774022
## # ℹ 11 more rows
```

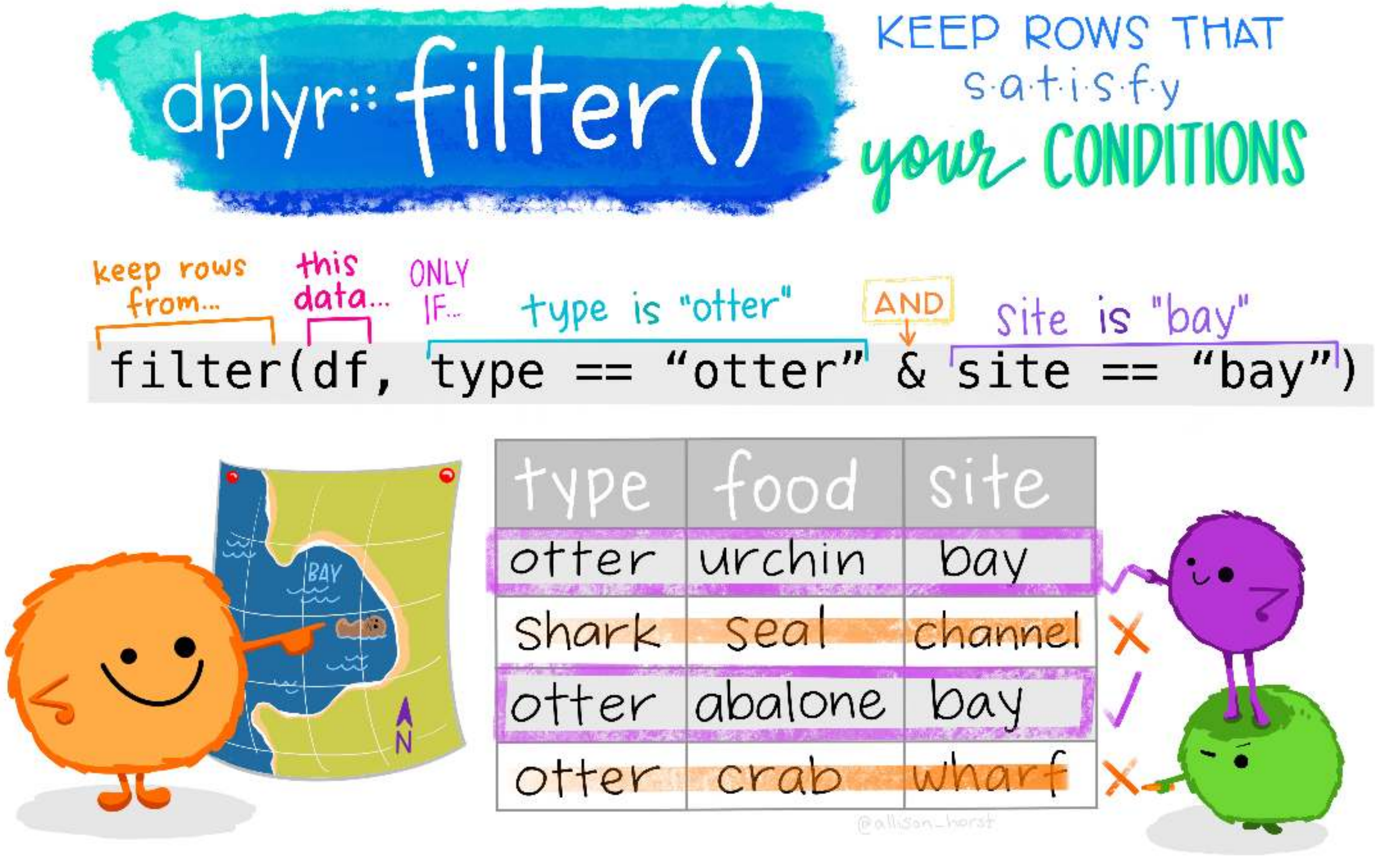
select()

- `&` and `|` for selecting the intersection or the union of two sets of variables.

```
worldcup %>% select(where(is.numeric) & ends_with("s"))
```

```
## # A tibble: 21 × 2
##    teams games
##    <dbl> <dbl>
##  1    13    18
##  2    16    17
##  3    15    18
##  4    13    22
##  5    16    26
##  6    16    35
##  7    16    32
##  8    16    32
##  9    16    32
## 10    16    38
## # ℹ 11 more rows
```

filter()



Art by Allison Horst

filter()

What if we want to only analyze certain rows? In `dplyr` we use the `filter()` function:

```
worldcup %>% filter(goals_scored > 100)
```

```
## # A tibble: 13 × 10
##     year host     winner second third fourth goals_scored teams games attendance
##    <dbl> <chr>    <chr>  <chr>  <chr> <chr>         <dbl> <dbl> <dbl>      <dbl>
##  1  1954 Switzer… West … Hunga… Aust… Urugu…          140    16    26     943000
##  2  1958 Sweden   Brazil Sweden Fran… West …          126    16    35     868000
##  3  1978 Argenti… Argen… Nethe… Braz… Italy           102    16    38    1610215
##  4  1982 Spain    Italy  West … Pola… France          146    24    52    1856277
##  5  1986 Mexico   Argen… West … Fran… Belgi…          132    24    52    2407431
##  6  1990 Italy    West … Argen… Italy Engla…          115    24    52    2527348
##  7  1994 USA      Brazil Italy  Swed… Bulga…          141    24    52    3568567
##  8  1998 France   France Brazil Croa… Nethe…          171    32    64    2859234
##  9  2002 Japan, … Brazil Germa… Turk… South…          161    32    64    2724604
## 10  2006 Germany  Italy  France Germ… Portu…          147    32    64    3367000
## 11  2010 South A… Spain  Nethe… Germ… Urugu…          145    32    64    2167984
## 12  2014 Brazil   Germa… Argen… Neth… Brazil          171    32    64    3441450
## 13  2018 Russia   France Croat… Belg… Engla…          169    32    64    3031768
```

filter()

We can use multiple conditions to filter (this represents an `AND`):

```
worldcup %>% filter(goals_scored > 100, year > 1975)
```

```
## # A tibble: 11 × 10
##     year host     winner second third fourth goals_scored teams games attendance
##    <dbl> <chr>    <chr>  <chr>  <chr> <chr>         <dbl> <dbl> <dbl>      <dbl>
##  1  1978 Argenti… Argen… Nethe… Braz… Italy           102    16    38    1610215
##  2  1982 Spain    Italy  West … Pola… France          146    24    52    1856277
##  3  1986 Mexico   Argen… West … Fran… Belgi…          132    24    52    2407431
##  4  1990 Italy    West … Argen… Italy Engla…          115    24    52    2527348
##  5  1994 USA      Brazil Italy  Swed… Bulga…          141    24    52    3568567
##  6  1998 France   France Brazil Croa… Nethe…          171    32    64    2859234
##  7  2002 Japan, … Brazil Germa… Turk… South…          161    32    64    2724604
##  8  2006 Germany  Italy  France Germ… Portu…          147    32    64    3367000
##  9  2010 South A… Spain  Nethe… Germ… Urugu…          145    32    64    2167984
## 10  2014 Brazil   Germa… Argen… Neth… Brazil          171    32    64    3441450
## 11  2018 Russia   France Croat… Belg… Engla…          169    32    64    3031768
```

filter()

The special function `%in%` also gets used often to specify multiple conditions:

```r
worldcup %>% filter(winner %in% c("Italy", "Spain"))
```

```
## # A tibble: 5 × 10
##    year host      winner second third fourth goals_scored teams games attendance
##   <dbl> <chr>     <chr>  <chr>  <chr> <chr>         <dbl> <dbl> <dbl>      <dbl>
## 1  1934 Italy     Italy  Czech… Germ… Austr…           70    16    17     395000
## 2  1938 France    Italy  Hunga… Braz… Sweden           84    15    18     483000
## 3  1982 Spain     Italy  West … Pola… France          146    24    52    1856277
## 4  2006 Germany   Italy  France Germ… Portu…          147    32    64    3367000
## 5  2010 South Af… Spain  Nethe… Germ… Urugu…          145    32    64    2167984
```

filter()

Its also possible to create `OR` filters using the `pipe` delimiter ("|"):

```
worldcup %>% filter(winner %in% c("Italy", "Spain") | goals_scored < 100)
```

```
## # A tibble: 11 × 10
##     year host     winner second third fourth goals_scored teams games attendance
##    <dbl> <chr>    <chr>  <chr>  <chr> <chr>         <dbl> <dbl> <dbl>      <dbl>
##  1  1930 Uruguay  Urugu… Argen… USA   Yugos…           70    13    18     434000
##  2  1934 Italy    Italy  Czech… Germ… Austr…           70    16    17     395000
##  3  1938 France   Italy  Hunga… Braz… Sweden           84    15    18     483000
##  4  1950 Brazil   Urugu… Brazil Swed… Spain            88    13    22    1337000
##  5  1962 Chile    Brazil Czech… Chile Yugos…           89    16    32     776000
##  6  1966 England  Engla… West … Port… Sovie…           89    16    32    1614677
##  7  1970 Mexico   Brazil Italy  West… Urugu…           95    16    32    1673975
##  8  1974 Germany  West … Nethe… Pola… Brazil           97    16    38    1774022
##  9  1982 Spain    Italy  West … Pola… France          146    24    52    1856277
## 10  2006 Germany  Italy  France Germ… Portu…          147    32    64    3367000
## 11  2010 South A… Spain  Nethe… Germ… Urugu…          145    32    64    2167984
```

filter()

Lastly, we can also use a function on a column (as a vector) and then filter on the outcome:

```
worldcup %>% filter(goals_scored > mean(goals_scored, na.rm = TRUE))
```

```
## # A tibble: 11 × 10
##     year host     winner second third fourth goals_scored teams games attendance
##    <dbl> <chr>    <chr>  <chr>  <chr> <chr>         <dbl> <dbl> <dbl>      <dbl>
##  1  1954 Switzer… West … Hunga… Aust… Urugu…          140    16    26     943000
##  2  1958 Sweden   Brazil Sweden Fran… West …          126    16    35     868000
##  3  1982 Spain    Italy  West … Pola… France          146    24    52    1856277
##  4  1986 Mexico   Argen… West … Fran… Belgi…          132    24    52    2407431
##  5  1994 USA      Brazil Italy  Swed… Bulga…          141    24    52    3568567
##  6  1998 France   France Brazil Croa… Nethe…          171    32    64    2859234
##  7  2002 Japan, … Brazil Germa… Turk… South…          161    32    64    2724604
##  8  2006 Germany  Italy  France Germ… Portu…          147    32    64    3367000
##  9  2010 South A… Spain  Nethe… Germ… Urugu…          145    32    64    2167984
## 10  2014 Brazil   Germa… Argen… Neth… Brazil          171    32    64    3441450
## 11  2018 Russia   France Croat… Belg… Engla…          169    32    64    3031768
```

Exercises 💻

Exercises

- Select all columns from year to winner.
- Select all the columns that is of the class character.
- Where the host was in the top three?
- Filter the World Cup had the most attendance and select the goals scored, the year and the winner.
    - Write out results to csv.

20:00

mutate()

Art by Allison Horst

mutate()

Often you will need to add a new column that you derive. To accomplish this using `dplyr` we use `mutate`. Lets calculate average goals per game:

```
worldcup %>% mutate(avg_goals = goals_scored/games)
```

```
## # A tibble: 21 × 11
##     year host     winner second third fourth goals_scored teams games attendance
##    <dbl> <chr>    <chr>  <chr>  <chr> <chr>         <dbl> <dbl> <dbl>      <dbl>
##  1  1930 Uruguay  Urugu… Argen… USA   Yugos…           70    13    18     434000
##  2  1934 Italy    Italy  Czech… Germ… Austr…           70    16    17     395000
##  3  1938 France   Italy  Hunga… Braz… Sweden           84    15    18     483000
##  4  1950 Brazil   Urugu… Brazil Swed… Spain            88    13    22    1337000
##  5  1954 Switzer… West … Hunga… Aust… Urugu…          140    16    26     943000
##  6  1958 Sweden   Brazil Sweden Fran… West …          126    16    35     868000
##  7  1962 Chile    Brazil Czech… Chile Yugos…           89    16    32     776000
##  8  1966 England  Engla… West … Port… Sovie…           89    16    32    1614677
##  9  1970 Mexico   Brazil Italy  West… Urugu…           95    16    32    1673975
## 10  1974 Germany  West … Nethe… Pola… Brazil           97    16    38    1774022
## # ℹ 11 more rows
## # ℹ 1 more variable: avg_goals <dbl>
```

mutate()

I do not enjoy having to code with capitals in character columns, so lets use `tolower` and `across` to fix this problem over all the character columns.

```
worldcup %>% mutate(across(where(is.character), tolower))
```

```
## # A tibble: 21 × 10
##     year host     winner second third fourth goals_scored teams games attendance
##    <dbl> <chr>    <chr>  <chr>  <chr> <chr>         <dbl> <dbl> <dbl>      <dbl>
##  1  1930 uruguay  urugu… argen… usa   yugos…           70    13    18     434000
##  2  1934 italy    italy  czech… germ… austr…           70    16    17     395000
##  3  1938 france   italy  hunga… braz… sweden           84    15    18     483000
##  4  1950 brazil   urugu… brazil swed… spain            88    13    22    1337000
##  5  1954 switzer… west … hunga… aust… urugu…          140    16    26     943000
##  6  1958 sweden   brazil sweden fran… west …          126    16    35     868000
##  7  1962 chile    brazil czech… chile yugos…           89    16    32     776000
##  8  1966 england  engla… west … port… sovie…           89    16    32    1614677
##  9  1970 mexico   brazil italy  west… urugu…           95    16    32    1673975
## 10  1974 germany  west … nethe… pola… brazil           97    16    38    1774022
## # ℹ 11 more rows
```

mutate()

Another nice feature we can use is the `case_when` function inside the mutate:

```
worldcup %>%
  mutate(year_groups = case_when(
    year < 1950 ~ "before 1950",
    between(year, 1950, 1970) ~ "1970s",
    year > 2000 ~ "2000s",
    TRUE ~ "other"
  ), .after = year) %>%
  head()
```

```
## # A tibble: 6 × 11
##    year year_groups host      winner second third fourth goals_scored teams games
##   <dbl> <chr>       <chr>     <chr>  <chr>  <chr> <chr>         <dbl> <dbl> <dbl>
## 1  1930 before 1950 Uruguay   Urugu… Argen… USA   Yugos…           70    13    18
## 2  1934 before 1950 Italy     Italy  Czech… Germ… Austr…           70    16    17
## 3  1938 before 1950 France    Italy  Hunga… Braz… Sweden           84    15    18
## 4  1950 1970s       Brazil    Urugu… Brazil Swed… Spain            88    13    22
## 5  1954 1970s       Switzer… West … Hunga… Aust… Urugu…          140    16    26
## 6  1958 1970s       Sweden    Brazil Sweden Fran… West …          126    16    35
## # ℹ 1 more variable: attendance <dbl>
```

group_by() & summarise()

We might want to run an aggregation over a certain variables to calculate means, medians, etc. This can be done in `R` using dplyr `group_by` and `summarise`.

```
worldcup %>%
  group_by(winner) %>%
  summarise(ave_attendance = mean(attendance, na.rm = T))
```

```
## # A tibble: 9 × 2
##   winner       ave_attendance
##   <chr>                 <dbl>
## 1 Argentina           2008823
## 2 Brazil              1922229.
## 3 England             1614677
## 4 France              2945501
## 5 Germany             3441450
## 6 Italy               1525319.
## 7 Spain               2167984
## 8 Uruguay              885500
## 9 West Germany        1748123.
```

group_by() & summarise()

We can also extend the calculations to multiple outputs:

```
worldcup %>%
  group_by(winner) %>%
  summarise(
    ave_attendance = mean(attendance, na.rm = T),
    min_attendance = min(attendance, na.rm = T),
    max_attendance = max(attendance, na.rm = T),
    number_wins = n()
    ) %>%
  arrange(desc(number_wins)) %>%
  filter(number_wins > 3)
```

```
## # A tibble: 2 × 5
##   winner ave_attendance min_attendance max_attendance number_wins
##   <chr>           <dbl>          <dbl>          <dbl>       <int>
## 1 Brazil       1922229.         776000        3568567           5
## 2 Italy        1525319.         395000        3367000           4
```

Exercise

> Exercises for mutate and summarise

- Calculate the average number of games per team for each world cup.
- Create a new column in the data that shows how many times the specific host country has hosted the world cup.
- Summarise the data to show the countries that have hosted the world cup, what the first year and last year was that they hosted it and what the total attendance for all the years they hosted it was. (Bonus, arrange the rows from the country with the highest all time attendance to the lowest)

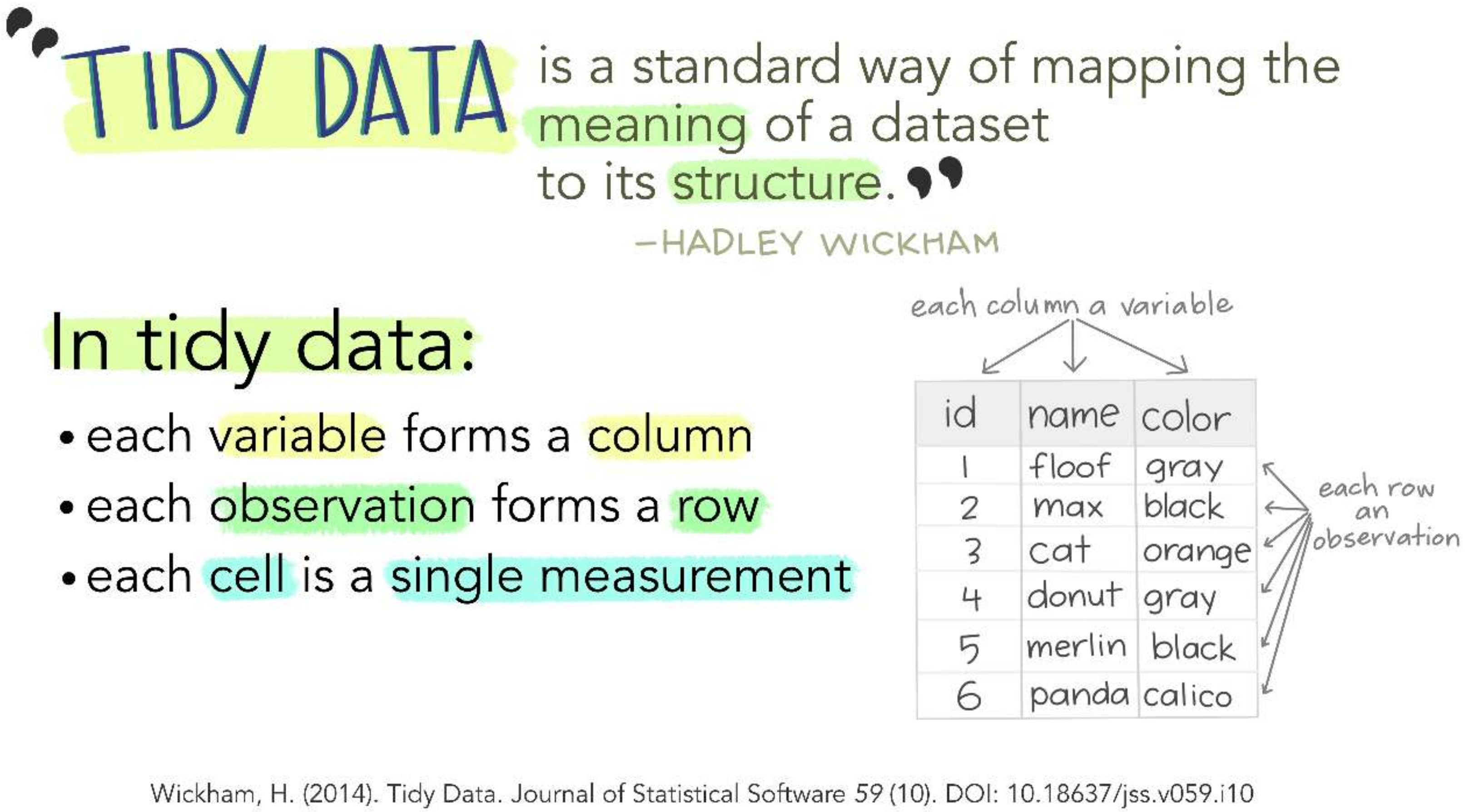
30:00

mastercard foundation

71point4
NUMBERS TO LIFE

From Excel to R
(Session 2-1 - Tidyr and Database connections)

Manipulating objects with tidyr



Art by Allison Horst

Data Manipulation Tidyr

```r
breed_traits <- readr::read_csv('data/breed_traits.csv') %>%
  janitor::clean_names() %>%
  mutate(breed = gsub("\u00A0", " ", breed, fixed =TRUE))
```

```r
breed_traits %>% head
```

```
## # A tibble: 6 × 17
##   breed        affectionate_with_fa…¹ good_with_young_chil…² good_with_other_dogs
##   <chr>                         <dbl>                  <dbl>                <dbl>
## 1 Retrievers…                       5                      5                    5
## 2 French Bul…                       5                      5                    4
## 3 German She…                       5                      5                    3
## 4 Retrievers…                       5                      5                    5
## 5 Bulldogs                          4                      3                    3
## 6 Poodles                           5                      5                    3
## # ℹ abbreviated names: ¹affectionate_with_family, ²good_with_young_children
## # ℹ 13 more variables: shedding_level <dbl>, coat_grooming_frequency <dbl>,
## #   drooling_level <dbl>, coat_type <chr>, coat_length <chr>,
## #   openness_to_strangers <dbl>, playfulness_level <dbl>,
## #   watchdog_protective_nature <dbl>, adaptability_level <dbl>,
## #   trainability_level <dbl>, energy_level <dbl>, barking_level <dbl>,
## #   mental_stimulation_needs <dbl>
```

https://twitter.com/WeAreRLadies/status/1494728669864112130

pivot_longer()

`pivot_longer()` is probably one of the most used functions when doing any analysis as most data come in 'human' readable format, while we want 'computer' readable data for data analysis. The arguments for the function is: `pivot_longer(names_to = ..., values_to = ... )`

- How does the data need to look if we want to get the breed with the best average score?

```
breed_traits %>%
  select(breed, where(is.numeric)) %>%
  pivot_longer(names_to = "attribute",
               values_to = "values",
               -breed)
```

```
## # A tibble: 2,730 × 3
##    breed                  attribute                  values
##    <chr>                  <chr>                       <dbl>
##  1 Retrievers (Labrador) affectionate_with_family         5
##  2 Retrievers (Labrador) good_with_young_children         5
##  3 Retrievers (Labrador) good_with_other_dogs             5
##  4 Retrievers (Labrador) shedding_level                   4
##  5 Retrievers (Labrador) coat_grooming_frequency          2
##  6 Retrievers (Labrador) drooling_level                   2
##  7 Retrievers (Labrador) openness_to_strangers            5
##  8 Retrievers (Labrador) playfulness_level                5
##  9 Retrievers (Labrador) watchdog_protective_nature       3
## 10 Retrievers (Labrador) adaptability_level               5
## # ℹ 2,720 more rows
```

pivot_longer()

`pivot_longer()` is probably one of the most used functions when doing any analysis as most data come in 'human' readable format, while we want 'computer' readable data for data analysis. The arguments for the function is: `pivot_longer(names_to = ..., values_to = ...)`

- How does the data need to look if we want to get the breed with the best average score?

```
breed_traits %>%
  select(breed, where(is.numeric)) %>%
  pivot_longer(names_to = "attribute",
               values_to = "values",
               -breed) %>%
  group_by(breed) %>%
  summarise(avg_values = mean(values),
            .groups = "drop") %>%
  arrange(desc(avg_values))
```

```
## # A tibble: 195 × 2
##    breed                         avg_values
##    <chr>                              <dbl>
##  1 Keeshonden                          4.29
##  2 Portuguese Water Dogs               4.14
##  3 Retrievers (Labrador)               4.14
##  4 Papillons                           4.07
##  5 Retrievers (Flat-Coated)            4.07
##  6 Shetland Sheepdogs                  4.07
##  7 German Shepherd Dogs                4
##  8 Poodles                             4
##  9 Setters (Irish)                     4
## 10 Vizslas                             4
## # ℹ 185 more rows
```

pivot_longer()

I am in need of a dog that has *short* hair, *highly trainable* and *good with young children*. Can you identify the breed that will best suite my needs? Use `filter`, `pivot_longer`, `group_by`, `summarise` and `arrange` to solve the problem…

30:00

pivot_wider()

`pivot_wider()` works just as `pivot_longer` did, but now it *spreads* the columns out in a wide format. I find I mostly use this when I am doing modeling exercises or outputting the values for team members in Excel to work with.

`pivot_wider(names_from = ... , values_from = ... )`

```
breed_traits %>%
  select(shedding_level, coat_type, coat_length) %>%
  group_by(coat_length, coat_type) %>%
  summarise(avg_shedding = mean(shedding_level),
            .groups = "drop")
```

```
## # A tibble: 19 × 3
##    coat_length  coat_type    avg_shedding
##    <chr>        <chr>               <dbl>
##  1 Long         Corded               1
##  2 Long         Curly                1.5
##  3 Long         Double               2.44
##  4 Long         Rough                2
##  5 Long         Silky                1.5
##  6 Long         Wavy                 2
##  7 Medium       Corded               1
##  8 Medium       Curly                1.4
##  9 Medium       Double               3.03
## 10 Medium       Rough                2.5
## 11 Medium       Silky                3
## 12 Medium       Smooth               3
## 13 Medium       Wavy                 1.8
## 14 Medium       Wiry                 2.53
## 15 Plott Hounds Plott Hounds          0
## 16 Short        Double               3.18
## 17 Short        Hairless             1
## 18 Short        Smooth               2.80
## 19 Short        Wiry                 2.36
```

pivot_wider()

`pivot_wider()` works just as `pivot_longer` did, but now it *spreads* the columns out in a wide format. I find I mostly use this when I am doing modeling exercises or outputting the values for team members in Excel to work with.

`pivot_wider(names_from = ... , values_from = ... )`

```
breed_traits %>%
  select(shedding_level, coat_type, coat_length) %>%
  group_by(coat_length, coat_type) %>%
  summarise(avg_shedding = mean(shedding_level),
            .groups = "drop") %>%
  pivot_wider(names_from = "coat_length",
              values_from = "avg_shedding")
```

```
## # A tibble: 10 × 5
##    coat_type      Long Medium `Plott Hounds` Short
##    <chr>         <dbl>  <dbl>               <dbl> <dbl>
##  1 Corded         1      1                     NA NA
##  2 Curly          1.5    1.4                   NA NA
##  3 Double         2.44   3.03                  NA  3.18
##  4 Rough          2      2.5                   NA NA
##  5 Silky          1.5    3                     NA NA
##  6 Wavy           2      1.8                   NA NA
##  7 Smooth        NA      3                     NA  2.80
##  8 Wiry          NA      2.53                  NA  2.36
##  9 Plott Hounds NA     NA                     0 NA
## 10 Hairless      NA     NA                     NA  1
```

Excercise

Use the `starwars` data set which has been loaded along with the tidyverse. Use tidyverse functions to (1) select all the columns from the first up to species; (2) use `pivot_longer()` and create a column that contains the attributes (hair, skin and eye) and a column that contains the color of the corresponding attribute and save it as `starwars_longer`; (3) use `pivot_wider()` to get the data frame back into its original wider format and save it as `starwars_wider`.

Answer

```
# pivot longer
starwars_long ← starwars %>%
  select(name:species) %>%
  pivot_longer(
    contains("_color"),
    names_to = "attribute",
    values_to = "color"
  )

starwars_long
```

After looking at the data in this format, it is perhaps not as sensible to have colors related to different attributes in a singloe column. We can use `pivot_wider` to return the data frame to its original form.

```
starwars_wide ← starwars_long %>%
  pivot_wider(
    names_from = attribute,
    values_from = color
  )

starwars_wide
```

unite() & separate()

`unite()` & `separate()` are two very useful functions when you want to construct a new variable by combining multiple columns into one (or *separating* columns that were previously joined).

- `unite(data, col, ..., sep = "_", remove = TRUE, na.rm = FALSE)`

```
breed_traits %>%
  select(coat_length, coat_type) %>%
  mutate(across(c(coat_type, coat_length),tolower)) %>%
  unite(new_coat_type, c(coat_type, coat_length),
        sep = "_")
```

```
## # A tibble: 195 × 1
##    new_coat_type
##    <chr>
##  1 double_short
##  2 smooth_short
##  3 double_medium
##  4 double_medium
##  5 smooth_short
##  6 curly_long
##  7 smooth_short
##  8 smooth_short
##  9 smooth_short
## 10 smooth_short
## # ℹ 185 more rows
```

unite() & separate()

`unite()` & `separate()` are two very useful functions when you want to construct a new variable by combining multiple columns into one (or *separating* columns that were previously joined).

- `separate(data, col, into, sep = "[^[:alnum:]]+", remove = TRUE, convert = FALSE, extra = "warn", fill = "warn", ... )`

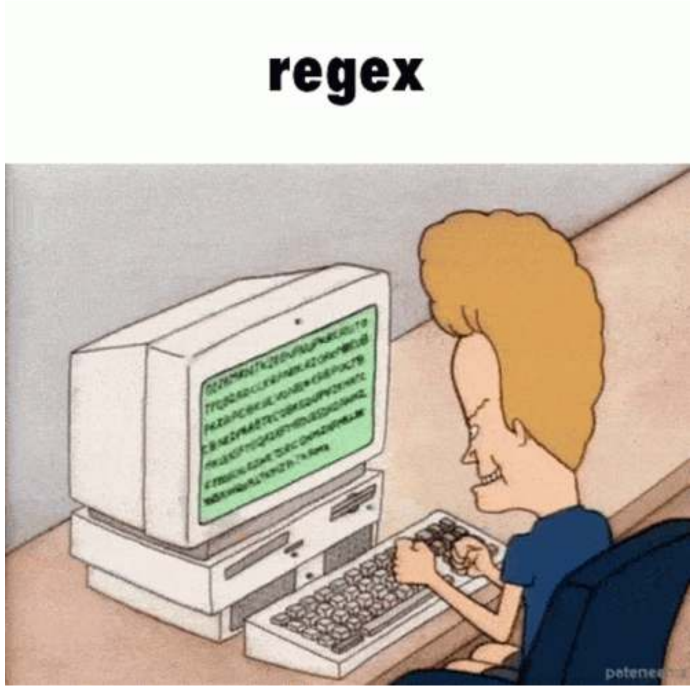
```r
breed_traits %>%
  select(coat_length, coat_type) %>%
  mutate(across(c(coat_type, coat_length),tolower)) %>%
  unite(new_coat_type, c(coat_type, coat_length),
        sep = "_") %>%
  separate(new_coat_type, into = c("coat_type", "coat_length"))
```

```
## # A tibble: 195 × 2
##    coat_type coat_length
##    <chr>     <chr>
##  1 double    short
##  2 smooth    short
##  3 double    medium
##  4 double    medium
##  5 smooth    short
##  6 curly     long
##  7 smooth    short
##  8 smooth    short
##  9 smooth    short
## 10 smooth    short
## # ℹ 185 more rows
```

extract()

`extract()` uses powerful regular expressions to split out a column into multiple columns. In regexp we use `()` to capture groups.

- `extract(data, col, into, regex = "([[:alnum:]]+)", remove = TRUE, convert = FALSE, ...)`




```
breed_traits %>%
  select(breed) %>%
  extract(breed, into = c('first_name', 'second_name'),
          '(.*)\\((.*)\\)',
          remove = FALSE) %>%
  head
```

```
## # A tibble: 6 × 3
##   breed                 first_name    second_name
##   <chr>                 <chr>         <chr>
## 1 Retrievers (Labrador) "Retrievers " Labrador
## 2 French Bulldogs        <NA>         <NA>
## 3 German Shepherd Dogs   <NA>         <NA>
## 4 Retrievers (Golden)   "Retrievers " Golden
## 5 Bulldogs               <NA>         <NA>
## 6 Poodles                <NA>         <NA>
```

complete()

Sometimes we want to have our groupings `complete` and so we can turn our *implicit* missing values into *explicit* missing values.

```
df <- tibble(
  group = c(1:2, 1),
  item_id = c(1:2, 2),
  item_name = c("a", "b", "b"),
  value1 = 1:3,
  value2 = 4:6
)
df %>% head
```

```
## # A tibble: 3 × 5
##   group item_id item_name value1 value2
##   <dbl>   <dbl> <chr>      <int>  <int>
## 1     1       1 a              1      4
## 2     2       2 b              2      5
## 3     1       2 b              3      6
```

complete()

Sometimes we want to have our groupings `complete` and so we can turn our *implicit* missing values into *explicit* missing values.

```
df <- tibble(
  group = c(1:2, 1),
  item_id = c(1:2, 2),
  item_name = c("a", "b", "b"),
  value1 = 1:3,
  value2 = 4:6
)
df %>%
  complete(group,
           nesting(item_id, item_name))
```

```
## # A tibble: 4 × 5
##   group item_id item_name value1 value2
##   <dbl>   <dbl> <chr>      <int>  <int>
## 1     1       1 a              1      4
## 2     1       2 b              3      6
## 3     2       1 a             NA     NA
## 4     2       2 b              2      5
```

Working with NAs in dataframe

- `drop_na()` drops all rows where there is a missing value
- Replace missing values with next/previous value with `fill()`
- Or a known value with `replace_na()`.

```
breed_traits %>%
  filter(!grepl("Hounds",coat_length)) %>%
  group_by(coat_length, coat_type) %>%
  summarise(avg_playfulness_level = mean(playfulness_level),
            .groups = "drop") %>%
  pivot_wider(names_from = "coat_type", values_from = "avg_playfulness_level")
```

```
## # A tibble: 3 × 10
##   coat_length Corded Curly Double Rough Silky  Wavy Smooth  Wiry Hairless
##   <chr>        <dbl> <dbl>  <dbl> <dbl> <dbl> <dbl>  <dbl> <dbl>    <dbl>
## 1 Long             3   5     3.81     4  3.5      4  NA    NA        NA
## 2 Medium           4   3.6   3.56     3  4.33     3   3.8   3.58     NA
## 3 Short           NA  NA     3.82    NA NA       NA   3.62  4         3.33
```

Working with NAs in dataframe

- `drop_na()` drops all rows where there is a missing value
- Replace missing values with next/previous value with `fill()`
- Or a known value with `replace_na()`.

- Use `replace_na()` to fill in the missing values of the *Curly* column with the mean (tip, use within `mutate`)
- Then use `fill()` to fill the values of *Hairless* upwards
- Lastly, drop all the rows that contain an NA with `drop_na()`

30:00

From Excel to R

(Session 2-2 - Database connections)

Connection with `dbplyr`

Power of dplyr and DBI

`dbplyr` is the database backend for dplyr. It allows you to use remote database tables as if they are in-memory data frames by automatically converting dplyr code into SQL.

The two main libraries we are going to use is: `dbplyr` and `DBI`

```
library(dbplyr)
library(DBI)
con <- dbConnect(RSQLite::SQLite(), ":memory:")
copy_to(con, breed_traits)
dbDisconnect(con)
```

- ⚠️ `dbplyr` is very cool, but will limit your functionality when working in larger teams. Raw SQL is more powerful and easier to maintain.
- 💀 Be careful to *only* use dbplyr for your data pipelines. The package is meant for Data Analysts and Data Scientist who don't do *anything* with the backend databases. So best used in large teams where roles are clearly defined and you only want to pull data from a database not interact with it in complex ways.
- ⚠️ Call `dbDisconnect()` when finished working with a connection!

Power of dplyr and DBI

Lets see if our connection worked:

```
con <- dbConnect(RSQLite::SQLite(), ":memory:")
copy_to(con, breed_traits)
breeds_db <- tbl(con, "breed_traits")
breeds_db
```

```
## # Source:   table<breed_traits> [?? x 17]
## # Database: sqlite 3.42.0 [:memory:]
##    breed      affectionate_with_fa…¹ good_with_young_chil…² good_with_other_dogs
##    <chr>                       <dbl>                  <dbl>                <dbl>
##  1 Retriever…                      5                      5                    5
##  2 French Bu…                      5                      5                    4
##  3 German Sh…                      5                      5                    3
##  4 Retriever…                      5                      5                    5
##  5 Bulldogs                        4                      3                    3
##  6 Poodles                         5                      5                    3
##  7 Beagles                         3                      5                    5
##  8 Rottweile…                      5                      3                    3
##  9 Pointers …                      5                      5                    4
## 10 Dachshunds                      5                      3                    4
## # ℹ more rows
## # ℹ abbreviated names: ¹affectionate_with_family, ²good_with_young_children
## # ℹ 13 more variables: shedding_level <dbl>, coat_grooming_frequency <dbl>,
## #   drooling_level <dbl>, coat_type <chr>, coat_length <chr>,
## #   openness_to_strangers <dbl>, playfulness_level <dbl>,
## #   watchdog_protective_nature <dbl>, adaptability_level <dbl>,
## #   trainability_level <dbl>, energy_level <dbl>, barking_level <dbl>, …
```

But what if we want to see the SQL?

All dplyr calls are evaluated lazily, generating SQL that is only sent to the database when you request the data!

```
coat_summary <- breeds_db %>%
  group_by(coat_length) %>%
  summarise(total_shedding = sum(shedding_level))

coat_summary %>%
  show_query()
```

```
## <SQL>
## SELECT `coat_length`, SUM(`shedding_level`) AS `total_shedding`
## FROM `breed_traits`
## GROUP BY `coat_length`
```

```
coat_summary %>%
  collect()
```

```
## # A tibble: 4 × 2
##   coat_length  total_shedding
##   <chr>                 <dbl>
## 1 Long                     58
## 2 Medium                  212
## 3 Plott Hounds               0
## 4 Short                   235
```

What are the most common connectors?

MySQL

```
RMySQL::MySQL()
```

PostgreSQL

```
RPostgreSQL::PostgreSQL()
```

Oracle

- ⚠️ Here be 🐉s!

```
ROracle::Oracle()
```

Lets write a basic connector function

💀 Never have plain text passwords in your code! Use the `usethis` package to edit your *environmental* variables which can then be called.

```
usethis::edit_r_environ()
```

```
mysql_user=ubuntu
mysql_passwd=my_long_password_2022
mysql_port=3310
mysql_hostname='localhost'
```

```
conn <- dbConnect(
  RMySQL::MySQL(),
  host = Sys.getenv("mysql_hostname"),
  port = Sys.getenv("mysql_port"),
  user = Sys.getenv("mysql_user"),
  password = Sys.getenv("mysql_passwd"),
  dbname = "warehouse",
  timeout = 10
)

DBI::dbGetQuery(conn, "SELECT * FROM test LIMIT 10")

DBI::dbDisconnect(conn)
```

Obviously also beware not to push your `.Renviron` *file to Github or similar websites.*

Write your own connector function!

Using the information below, write your own `db_query` function that takes a `SQL` query as input, runs the query and disconnects. Saving you a lot of time in the future!

```
conn <- dbConnect(
  RMySQL::MySQL(),
  host = Sys.getenv("mysql_hostname"),
  port = Sys.getenv("mysql_port"),
  user = Sys.getenv("mysql_user"),
  password = Sys.getenv("mysql_passwd"),
  dbname = "warehouse",
  timeout = 10
)

DBI::dbGetQuery(conn, "SELECT * FROM test LIMIT 10")

DBI::dbDisconnect(conn)
```

15:00

Loading data into DB from commandline

Before we load a dataset into a DB, we have to create the correct table format! Remember from the *foundations* how to do this:

```sql
CREATE database somedb;
CREATE TABLE sometable(
  id  VARCHAR(32),
  date_creation DATE,
  derived_lcy DOUBLE,
  PRIMARY KEY(id, date_creation)
)
;
```

```sql
LOAD DATA LOCAL INFILE '/home/ubuntu/data/{filename}.cs
INTO TABLE somedb.{tablename}
FIELDS TERMINATED BY ','
IGNORE 1 LINES
;
```

Loading data into DB from R

Before we load a dataset into a DB, we have to create the correct table format! Remember from the *foundations* how to do this:

```sql
CREATE database somedb;
CREATE TABLE sometable(
  id  VARCHAR(32),
  date_creation DATE,
  derived_lcy DOUBLE,
  PRIMARY KEY(id, date_creation)
)
;
```

```r
conn <- dbConnect(
  RMySQL::MySQL(),
  host = Sys.getenv("mysql_hostname"),
  port = Sys.getenv("mysql_port"),
  user = Sys.getenv("mysql_user"),
  password = Sys.getenv("mysql_passwd"),
  dbname = "warehouse",
  timeout = 10
)

DBI::dbWriteTable(conn, name = "sometable", value = df,
                  append = TRUE, overwrite = FALSE)
```

Penguins database exercise

- Connect to a database.
- Load in Palmer penguins into the database (`palmerpenguins::penguins`).
- Use the `tbl()` function to make a reference to penguins to ease access to the database.
- Get the average body mass of penguins by sex and species.
- What is the most observed species on every island?
- Disconnect when finished using the connection.

40:00

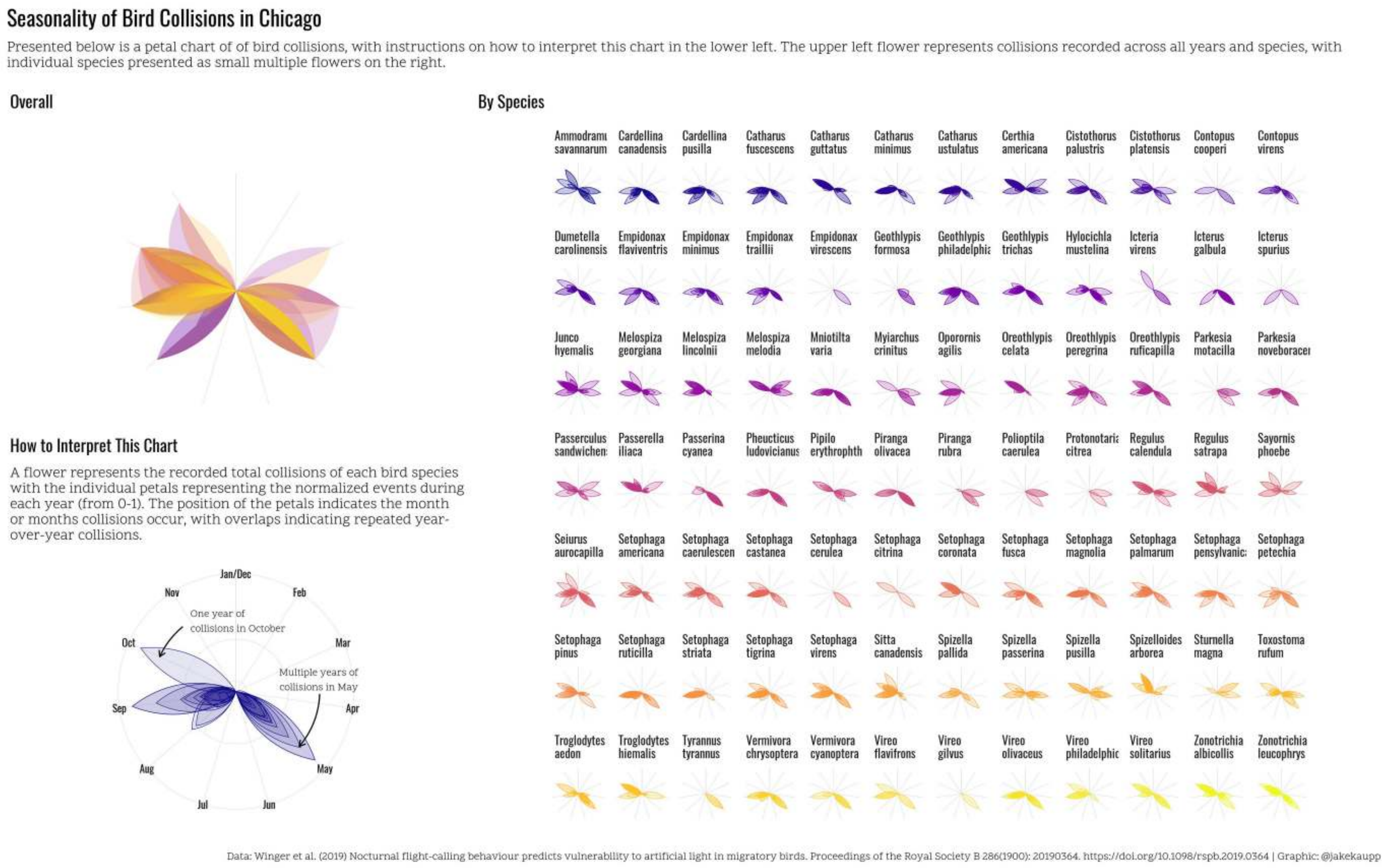
From Excel to R

(Session 3-1 - ggplot2 Grammer of Graphics)

Plotting with

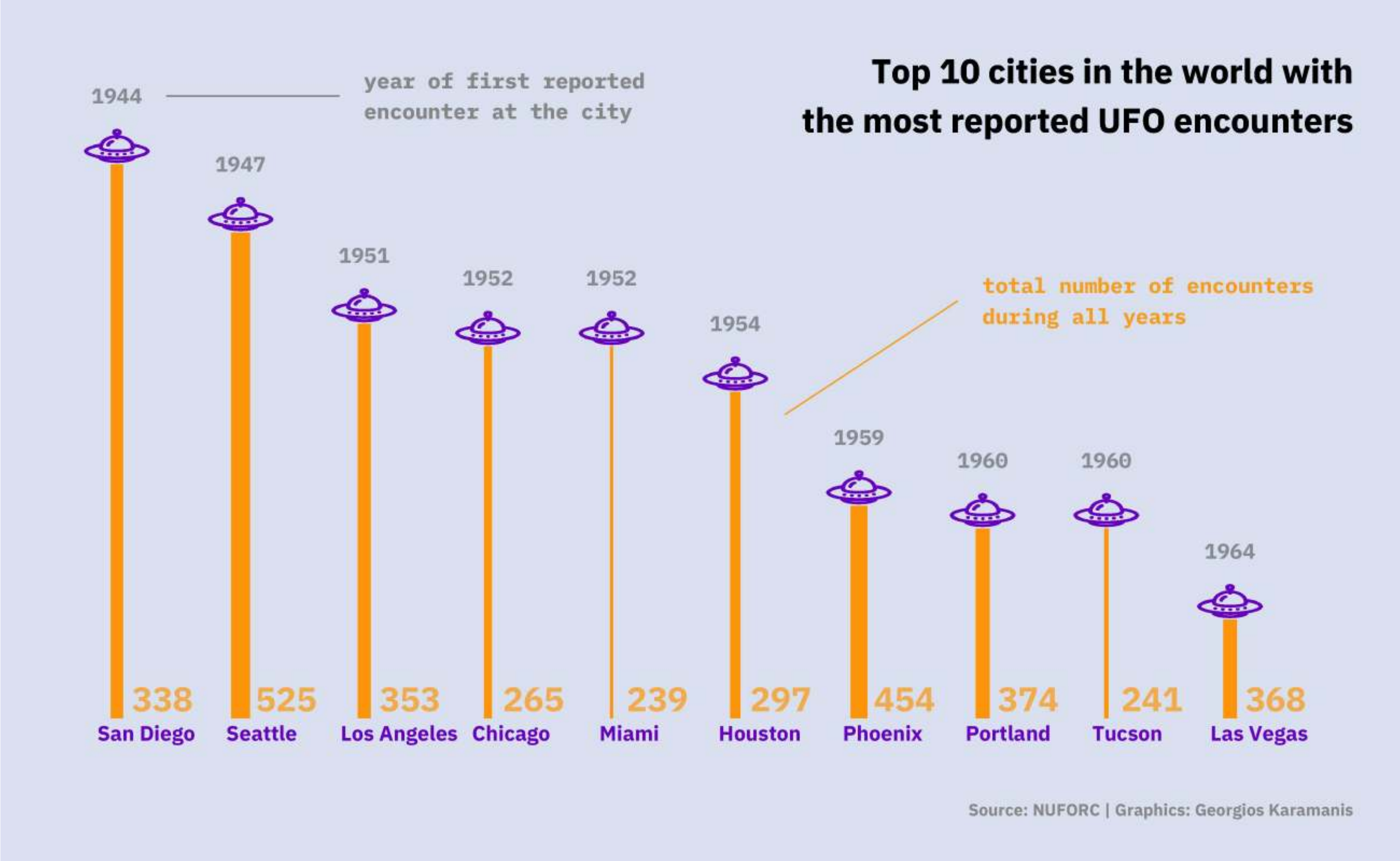
`ggplot2` 🎨

Examples

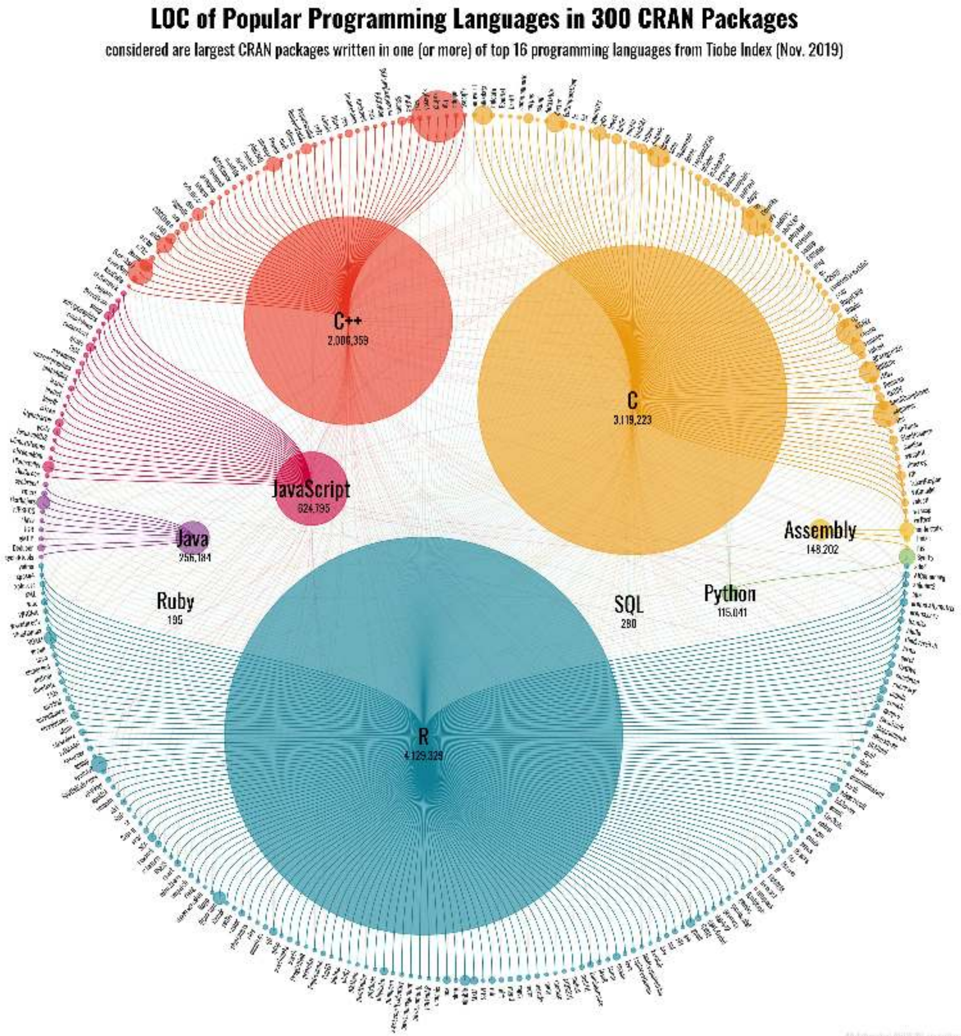


Seasonality of Bird Collisions in Chicago
Presented below is a petal chart of of bird collisions, with instructions on how to interpret this chart in the lower left. The upper left flower represents collisions recorded across all years and species, with individual species presented as small multiple flowers on the right.
Overall
By Species
Ammodramu savannarum
Cardellina canadensis
Cardellina pusilla
Catharus fuscescens
Catharus guttatus
Catharus minimus
Catharus ustulatus
Certhia americana
Cistothorus palustris
Cistothorus platensis
Contopus cooperi
Contopus virens
Dumetella carolinensis
Empidonax flaviventris
Empidonax minimus
Empidonax traillii
Empidonax virescens
Geothlypis formosa
Geothlypis philadelphia
Geothlypis trichas
Hylocichla mustelina
Icteria virens
Icterus galbula
Icterus spurius
Junco hyemalis
Melospiza georgiana
Melospiza lincolnii
Melospiza melodia
Mniotilta varia
Myiarchus crinitus
Oporornis agilis
Oreothlypis celata
Oreothlypis peregrina
Oreothlypis ruficapilla
Parkesia motacilla
Parkesia noveboracen
Passerculus sandwichen
Passerella iliaca
Passerina cyanea
Pheucticus ludovicianus
Pipilo erythrophth
Piranga olivacea
Piranga rubra
Polioptila caerulea
Protonotaria citrea
Regulus calendula
Regulus satrapa
Sayornis phoebe
Seiurus aurocapilla
Setophaga americana
Setophaga caerulescen
Setophaga castanea
Setophaga cerulea
Setophaga citrina
Setophaga coronata
Setophaga fusca
Setophaga magnolia
Setophaga palmarum
Setophaga pensylvanic
Setophaga petechia
Setophaga pinus
Setophaga ruticilla
Setophaga striata
Setophaga tigrina
Setophaga virens
Sitta canadensis
Spizella pallida
Spizella passerina
Spizella pusilla
Spizelloides arborea
Sturnella magna
Toxostoma rufum
Troglodytes aedon
Troglodytes hiemalis
Tyrannus tyrannus
Vermivora chrysoptera
Vermivora cyanoptera
Vireo flavifrons
Vireo gilvus
Vireo olivaceus
Vireo philadelphic
Vireo solitarius
Zonotrichia albicollis
Zonotrichia leucophrys
How to Interpret This Chart
A flower represents the recorded total collisions of each bird species with the individual petals representing the normalized events during each year (from 0-1). The position of the petals indicates the month or months collisions occur, with overlaps indicating repeated year-over-year collisions.
Jan/Dec
Feb
Mar
Apr
May
Jun
Jul
Aug
Sep
Oct
Nov
One year of collisions in October
Multiple years of collisions in May
Data: Winger et al. (2019) Nocturnal flight-calling behaviour predicts vulnerability to artificial light in migratory birds. Proceedings of the Royal Society B 286(1900): 20190364. https://doi.org/10.1098/rspb.2019.0364 | Graphic: @jakekaupp

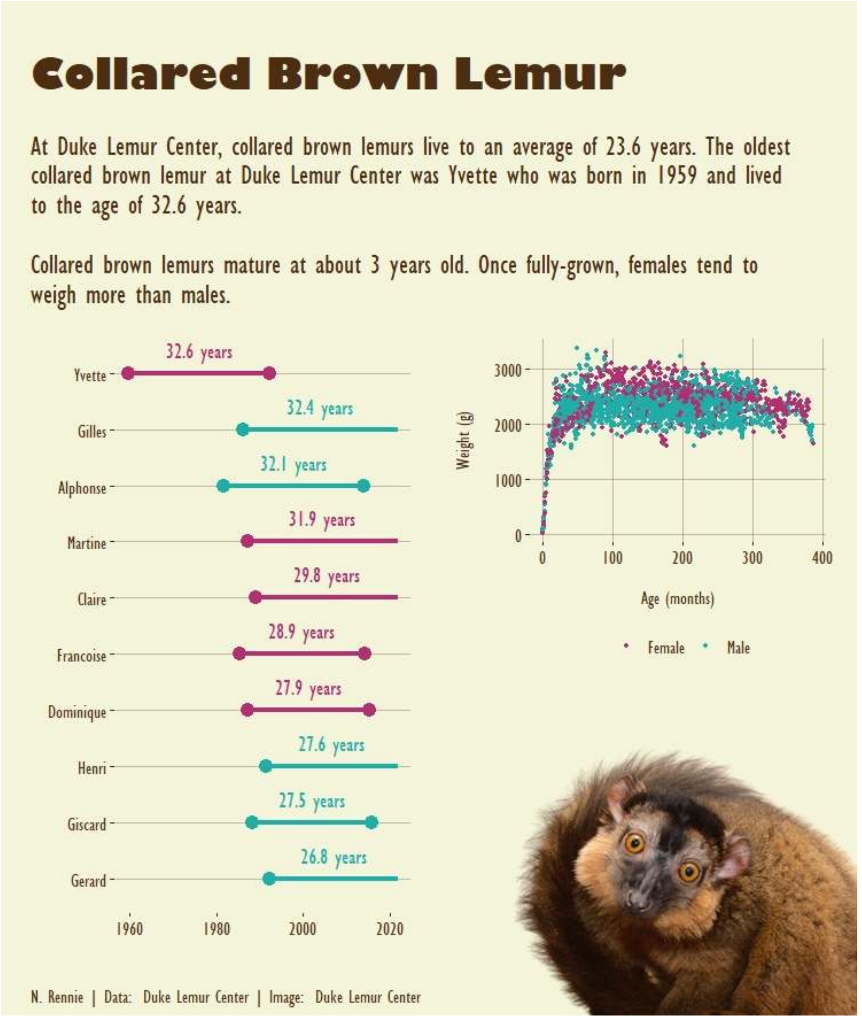
Examples


Top 10 cities in the world with the most reported UFO encounters
year of first reported encounter at the city
total number of encounters during all years
1944
338
San Diego
1947
525
Seattle
1951
353
Los Angeles
1952
265
Chicago
1952
239
Miami
1954
297
Houston
1959
454
Phoenix
1960
374
Portland
1960
241
Tucson
1964
368
Las Vegas
Source: NUFORC | Graphics: Georgios Karamanis

Examples

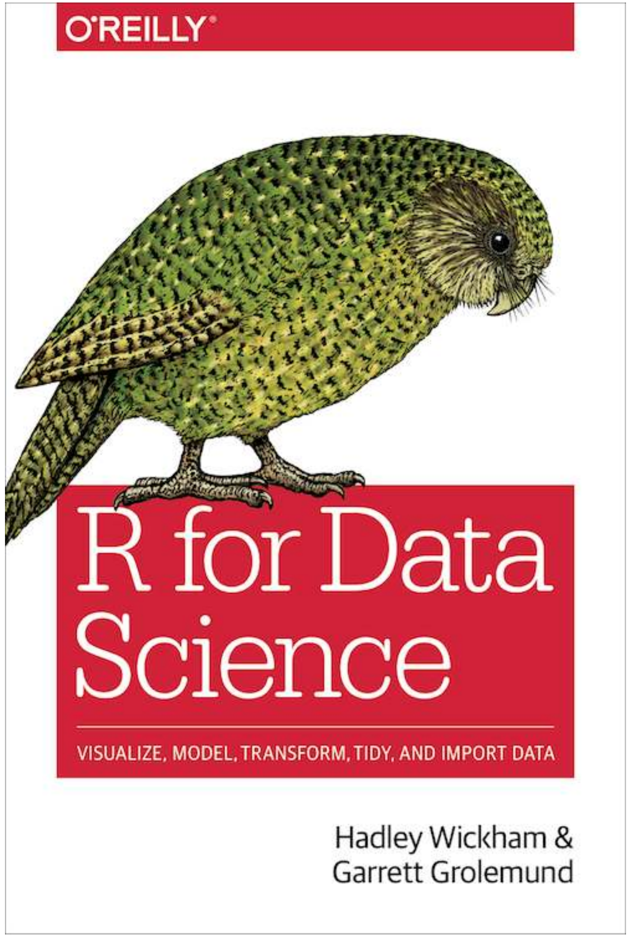

LOC of Popular Programming Languages in 300 CRAN Packages
considered are largest CRAN packages written in one (or more) of top 16 programming languages from Tiobe Index (Nov. 2019)
C++
2,000,359
C
3,119,223
JavaScript
624,795
Java
256,184
Assembly
148,202
Ruby
195
SQL
280
Python
115,041
R
4,129,328

Examples

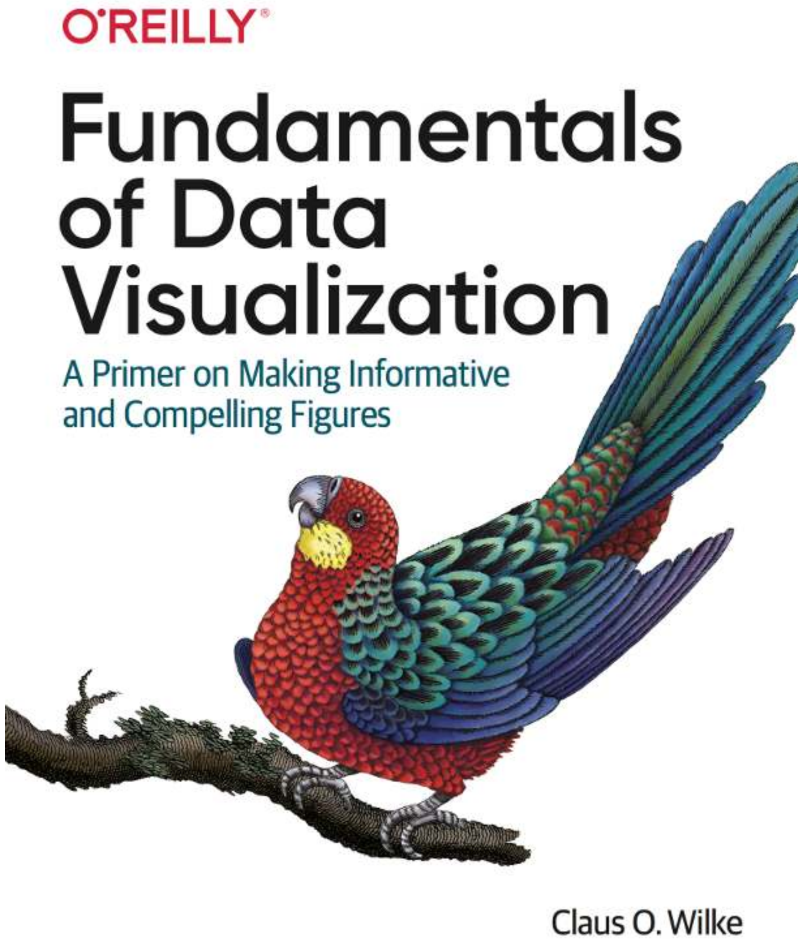

Collared Brown Lemur
At Duke Lemur Center, collared brown lemurs live to an average of 23.6 years. The oldest collared brown lemur at Duke Lemur Center was Yvette who was born in 1959 and lived to the age of 32.6 years.
Collared brown lemurs mature at about 3 years old. Once fully-grown, females tend to weigh more than males.
32.6 years
Yvette
32.4 years
Gilles
32.1 years
Alphonse
31.9 years
Martine
29.8 years
Claire
28.9 years
Francoise
27.9 years
Dominique
27.6 years
Henri
27.5 years
Giscard
26.8 years
Gerard
1960
1980
2000
2020
Weight (g)
3000
2000
1000
0
0
100
200
300
400
Age (months)
Female
Male
N. Rennie | Data: Duke Lemur Center | Image: Duke Lemur Center

Best books to consult

- General



- Advanced



Understanding ggplot2

The ggplot2 package is widely used and valued for its simple, consistent approach to making plots.

The most common aspects you will be engaging with in terms of creating a plot will be:

- aesthetics
- geometric representations
- facets
- coordinate space
- coordinate labels
- plot theme

What is important to understand is that `ggplot2` is a layered interface.

Read in our dataset

```r
scoobydoo <- read_csv("data/scoobydoo.csv")
head(scoobydoo)
```

```
## # A tibble: 6 × 75
##   index series_name    network season title imdb  engagement date_aired run_time
##   <dbl> <chr>          <chr>   <chr>  <chr> <chr> <chr>      <date>        <dbl>
## 1     1 Scooby Doo, W… CBS     1      What… 8.1   556        1969-09-13       21
## 2     2 Scooby Doo, W… CBS     1      A Cl… 8.1   479        1969-09-20       22
## 3     3 Scooby Doo, W… CBS     1      Hass… 8     455        1969-09-27       21
## 4     4 Scooby Doo, W… CBS     1      Mine… 7.8   426        1969-10-04       21
## 5     5 Scooby Doo, W… CBS     1      Deco… 7.5   391        1969-10-11       21
## 6     6 Scooby Doo, W… CBS     1      What… 8.4   384        1969-10-18       21
## # ℹ 66 more variables: format <chr>, monster_name <chr>, monster_gender <chr>,
## #   monster_type <chr>, monster_subtype <chr>, monster_species <chr>,
## #   monster_real <chr>, monster_amount <dbl>, caught_fred <chr>,
## #   caught_daphnie <chr>, caught_velma <chr>, caught_shaggy <chr>,
## #   caught_scooby <chr>, captured_fred <chr>, captured_daphnie <chr>,
## #   captured_velma <chr>, captured_shaggy <chr>, captured_scooby <chr>,
## #   unmask_fred <chr>, unmask_daphnie <chr>, unmask_velma <chr>, …
```

Tell me something interesting about Scooby Doo

Tell me something interessing about this dataset...

30:00

Creating a base plot

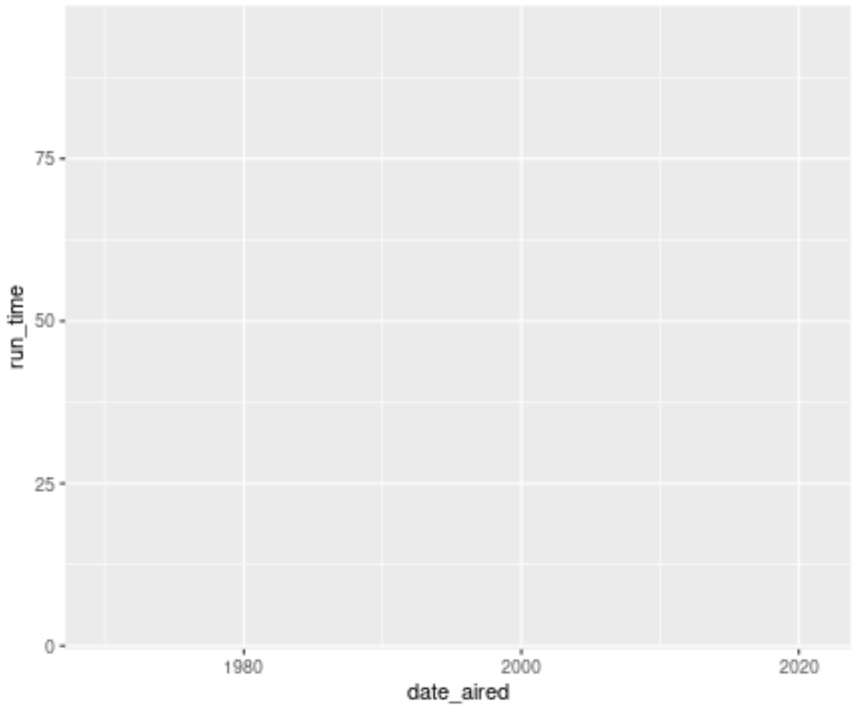
Running this command will produce an empty grey panel which serves as your *canvas*. We need to specify how different columns of the data frame should be represented in the plot.

```
scoobydoo %>% ggplot()
```

Creating a base plot

Column names are given as `aesthetic` elements to the ggplot function, and are wrapped in the `aes()` function. Note that we can use "lazy notation" (i.e, don't need to have columns in quotes). But we still don't have anything on the graph...

```
scoobydoo %>% ggplot(., aes(x = date_aired, y = run_time))
```

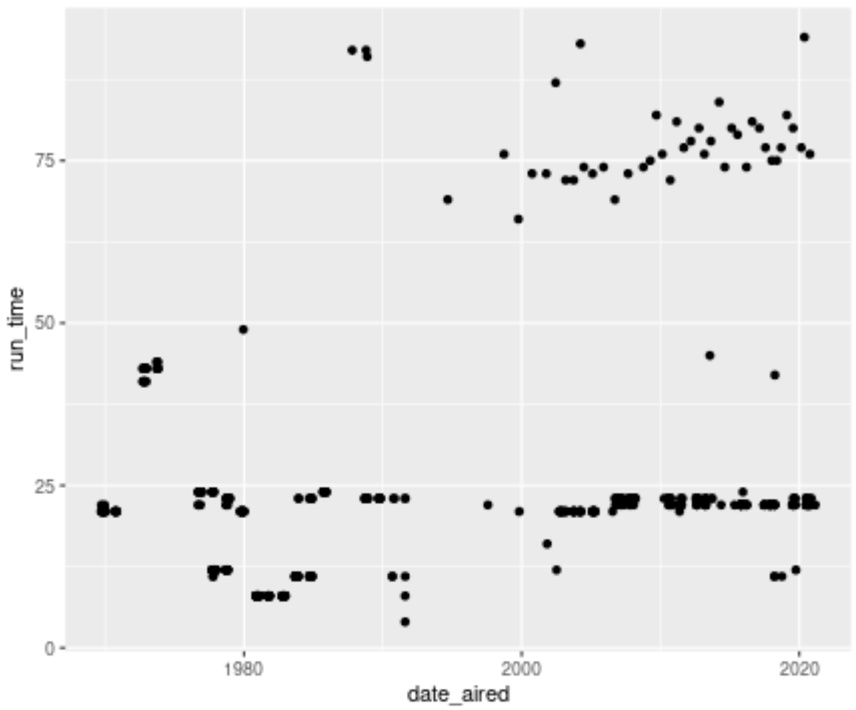


Geometric representations geom()

Now that we have a base layer, we need to add a `geom` layer (hence the + sign) of points to the plot.

- ⚠️ Its not the usual `%>%` but a `+` for `ggplot2`.

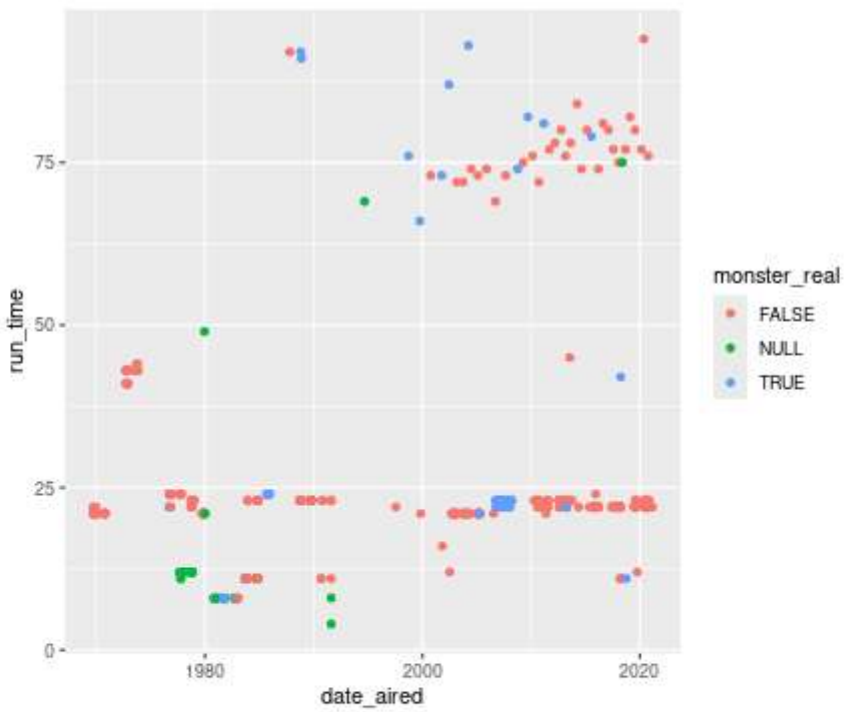
```
scoobydoo %>%
  ggplot(., aes(x = date_aired, y = run_time)) +
  geom_point()
```



Adding color to geom()

Because its an aesthetic, the color has to go in the `aes` component:

```
scoobydoo %>%
  ggplot(., aes(x = date_aired, y = run_time, color = monster_real)) +
  geom_point()
```



What other geom_point() plots can you make?

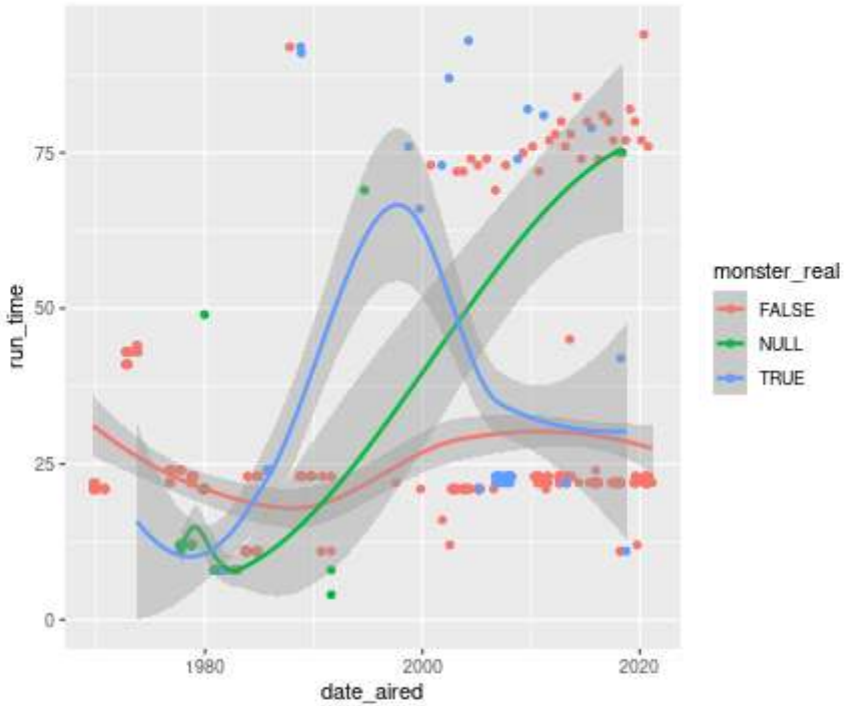
What other interesting relationships exist within the data?

20:00

Learning about adding layers

In the beginning I told you ggplot allows you to add layers. What does mean? Well, lets add a `geom_smooth` line to our plot:

```
scoobydoo %>%
  ggplot(., aes(x = date_aired, y = run_time, color = monster_real)) +
  geom_point() +
  geom_smooth()
```

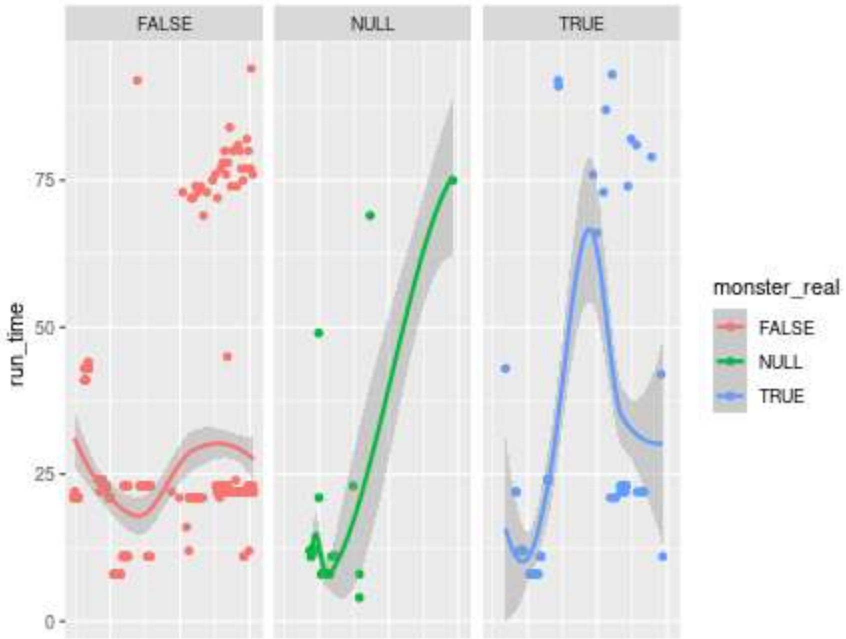


What if I want to split my plot?

Well, then you are in luck, we can use a `facet_wrap` command to split the plots!

- `facet_wrap(facets, nrow = NULL, ncol = NULL, scales = "fixed", ... )`

```
scoobydoo %>%
  ggplot(., aes(x = date_aired, y = run_time, color = monster_real)) +
  geom_point() +
  geom_smooth() +
  facet_wrap(~monster_real)
```

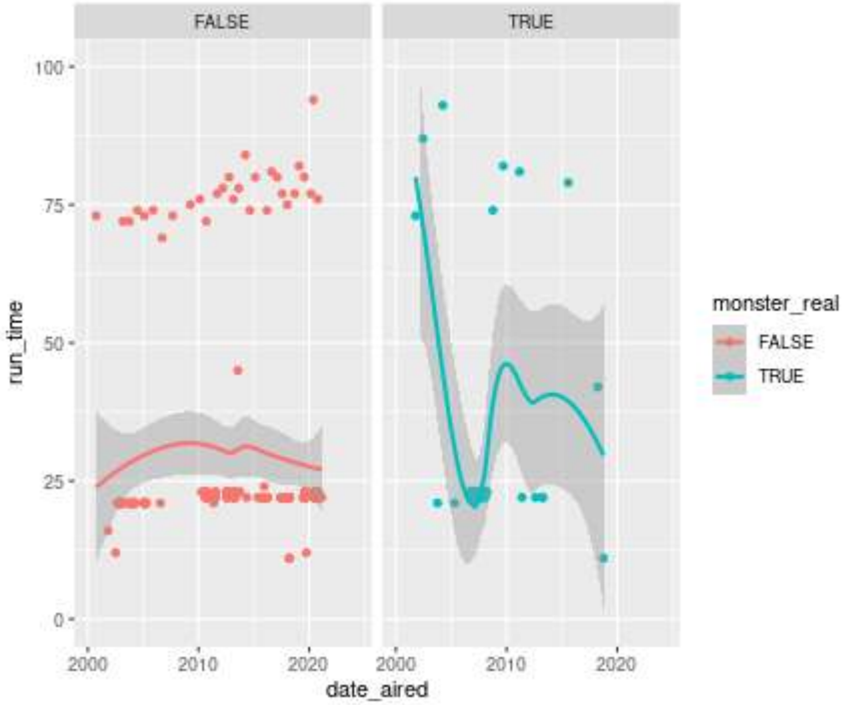


Coordinate space & labels

We might want to adjust to coordinates that are represented on the graph. For this we can use `xlim` and `ylim`.

- ⚠️ Remember, for most graphs we want to start the graph at 0, otherwise the insight might be misleading…

```
scoobydoo %>%
  filter(monster_real ≠ "NULL") %>%
  ggplot(., aes(x = date_aired, y = run_time
  geom_point() +
  geom_smooth() +
  facet_wrap(~monster_real) +
  xlim(as.Date("2000-01-01"), Sys.Date()) +
  ylim(0, 100)
```



Coordinate space & labels

Currently our plot has some ugly names… lets change that and make it 🥳

R Code   ggplot

```
scoobydoo %>%
  filter(monster_real != "NULL") %>%
  ggplot(., aes(x = date_aired, y = run_time, color = monster_real)) +
  geom_point() +
  geom_smooth() +
  facet_wrap(~monster_real) +
  xlim(as.Date("2000-01-01"), Sys.Date()) +
  ylim(0, 100) +
  labs(
    y = "Run Time (Mins)",
    x = "Date Show Aired",
    title = "Scooby Doo in the 2000s",
    subtitle = "Are monsters more real in longer shows?",
    caption = "Scooby Doo DB from TidyTuesday"
  )
```

What more can we do you ask?

ggplot2 has hundreds of themes to choose from

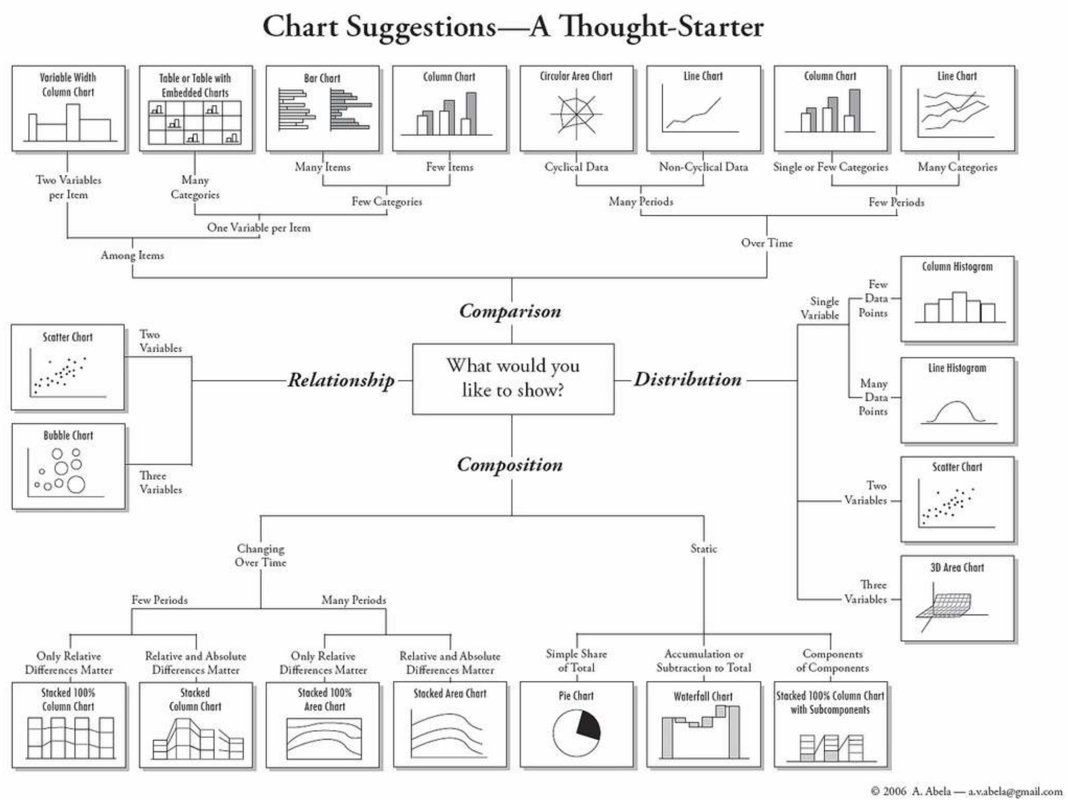
Lets choose a `ggthemes::theme_economist()` theme and push the legend to the bottom

R Code   ggplot

```
scoobydoo %>%
  filter(monster_real ≠ "NULL") %>%
  ggplot(., aes(x = date_aired, y = run_time, color = monster_real)) +
  geom_point() +
  geom_smooth() +
  facet_wrap(~monster_real) +
  xlim(as.Date("2000-01-01"), Sys.Date()) +
  ylim(0, 100) +
  labs(
    y = "Run Time (Mins)",
    x = "Date Show Aired",
    title = "Scooby Doo in the 2000s",
    subtitle = "Are monsters more real in longer shows?",
    caption = "Scooby Doo DB from TidyTuesday",
    color = "Monter Real?"
  ) + ggthemes::theme_economist() +
  theme(legend.position = "bottom")
```

What about going beyond basic points?

Although the `geom_point` was a place to start in answering our question. Perhaps another type of plot will be better suited?



https://flowingdata.com/2009/01/15/flow-chart-shows-you-what-chart-to-use/

Plotting density distributions

Our `geom_point` geometry didn't really provide use with the answer we needed. I think a `geom_density()` would be better suited.

> Using `geom_density()`, answer our question of: Are monsters more real (*monster_real*) in longer shows? 😉 you dont need a x-axis for density estimation. What happens when you use `geom_density(alpha = 0.3)`?

30:00

Plotting density distributions

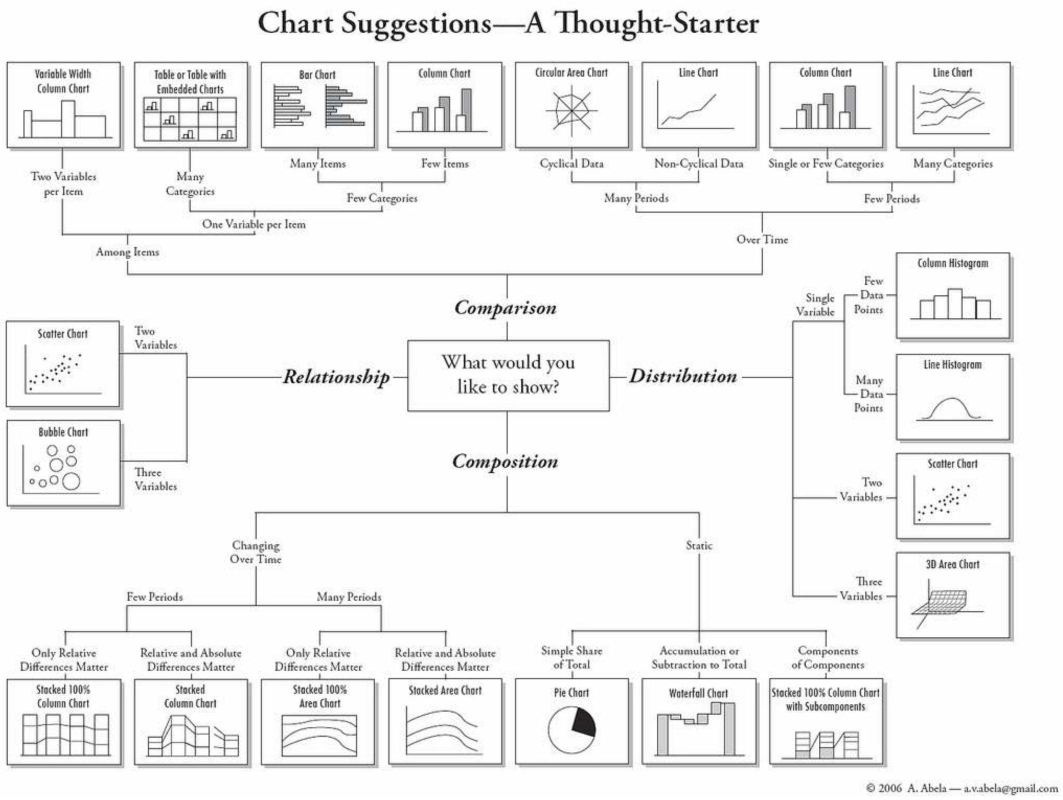
Another way of showing densities is to use boxplots!

What happens when you try `geom_boxplot` or `geom_violin()`? Try adding a `geom_jitter(position = position_jitter(0.2))` layer and see how your chart changes

30:00

Exploring other type charts

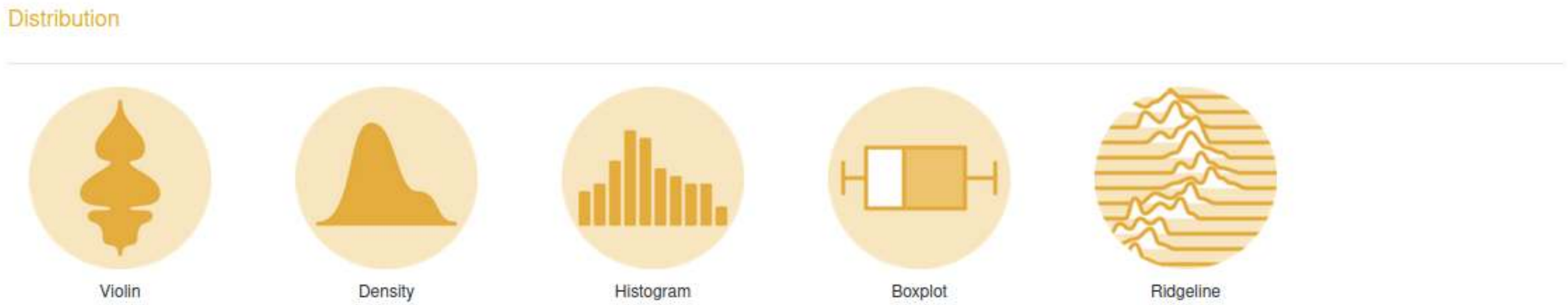
Best place to look: https://r-graph-gallery.com/



Practice Practice Practice

Using your newly found skills and the `Scooby Doo` dataset. Investigate your data by plotting one plot each from the following categories:

- Distribution



30:00

Practice Practice Practice

Using your newly found skills and the `Scooby Doo` dataset. Investigate your data by plotting one plot each from the following categories:

- Correlation

30:00

Practice Practice Practice

Using your newly found skills and the `Scooby Doo` dataset. Investigate your data by plotting one plot each from the following categories:

- Ranking

30:00

Practice Practice Practice

Using your newly found skills and the `Scooby Doo` dataset. Investigate your data by plotting one plot each from the following categories:

- Evolution

30:00

71point4
NUMBERS TO LIFE

cenfri

mastercard
foundation

TIVAN

> Investigate … read some data from the internet, ask question about data in viz

From Excel to R

(Session 3-2 - Advanced R Functional)

Making your code purrr with `purrr` 😸

Making your code purrr

Someone has to write the loop, that doesnt mean that it has to be you...

```r
x <- c(1:10)

empty <- vector()

for(i in 1:length(x)){
  if(x[i] %% 2)  {
    empty[i] <- x[i]*2
  } else {
    empty[i] <- x[i]*3
  }
}

print(empty)
```

```
##  [1]  2  6  6 12 10 18 14 24 18 30
```

I coded like 3 bugs in the loop above before getting it right...

```r
x <- c(1:10)

addition <- function(i){
    if(i %% 2)  {
    out <- i*2
  } else {
    out <- i*3
  }
  return(out)
}

purrr::map_dbl(x, addition)
```

```
##  [1]  2  6  6 12 10 18 14 24 18 30
```

Basics of purrr: map

So what is happening when we *apply (or map)* a function across an *object*.

Whats happening on the right?

- Take sequence that goes from 1 to 5.
- For each of the elements in the vector apply some function.
- Return the correct element type.

What else?

- Also, it doesnt need to be a sequence... we can also map a `tibble`. 🔥
- Besides `map`, we can `map2` or even `pmap`

```r
x <- c(1:10)

addition <- function(i){
    if(i %% 2)  {
    out <- i*2
  } else {
    out <- i*3
  }
  return(out)
}

purrr::map_dbl(x, addition)
```

```
##  [1]  2  6  6 12 10 18 14 24 18 30
```

Time to get practical

Write your own `map` function to `paste0` a sequence of numbers with the words "the number is: {number}".

15:00

Basics of purrr: map2

In most cases you are not just going to give it a single vector, but perhaps two vectors. Lets extend our example from above to use a map2

```
x <- c(1:10)
y <- rnorm(10)

addition_two <- function(i, j){
    if(i %% 2)  {
    out <- i*j
  } else {
    out <- i*j
  }
  return(out)
}

purrr::map2(x, y, addition_two)
```

- Do you notice something about the type of output?

Basics of purrr: map and tibble

Playing around with vectors are important to understand the function of what `map` does under the hood. And as we just saw, `map` on default delivers an `object` that is of class `list`… and a tibble is a `list` of `lists`.

```
tibble(x = c(1:10))
```

```
## # A tibble: 10 × 1
##        x
##    <int>
##  1     1
##  2     2
##  3     3
##  4     4
##  5     5
##  6     6
##  7     7
##  8     8
##  9     9
## 10    10
```

Basics of purrr: map and tibble

Because a tibble is a `list` of `lists`, we can just use `mutate` + `map` in very efficient ways. In addition, you can use the `map_*` suffice to define a specific output type.

```
addition <- function(i){
    if(i %% 2)  {
    out <- i*2
  } else {
    out <- i*3
  }
  return(out)
}
```

```
tibble(x = c(1:10)) %>%
  mutate(res = map(x, addition)) %>%
  head
```

```
## # A tibble: 6 × 2
##         x res
##     <int> <list>
## 1       1 <dbl [1]>
## 2       2 <dbl [1]>
## 3       3 <dbl [1]>
## 4       4 <dbl [1]>
## 5       5 <dbl [1]>
## 6       6 <dbl [1]>
```

Basics of purrr: map and tibble

Because a tibble is a `list` of `lists`, we can just use `mutate` + `map` in very efficient ways. In addition, you can use the `map_*` suffice to define a specific output type.

```
addition <- function(i){
    if(i %% 2)  {
    out <- i*2
  } else {
    out <- i*3
  }
  return(out)
}
```

```
tibble(x = c(1:10)) %>%
  mutate(res = map_dbl(x, addition)) %>%
  head
```

```
## # A tibble: 6 × 2
##         x   res
##     <int> <dbl>
## 1       1     2
## 2       2     6
## 3       3     6
## 4       4    12
## 5       5    10
## 6       6    18
```

Basics of purrr: map and tibble

What is going to happen when we start doing more complex things? Like perhaps not output an `integer` but we output a `tibble`?

```r
addition_tbl <- function(i){
    if(i %% 2)  {
    out <- i*2
  } else {
    out <- i*3
  }

  res <- tibble(was_div_two = i %% 2, out)

  return(res)
}
```

```r
tibble(x = c(1:10)) %>%
  mutate(res = map_dbl(x, addition_tbl)) %>%
  head
```

```
# Error in `mutate()`:
# ! Problem while computing `res = map_dbl(x, addition_
# Caused by error in `stop_bad_type()`:
# ! Result 1 must be a single double, not a vector of c
# Run `rlang::last_error()` to see where the error occu
```

Basics of purrr: map and tibble

What is going to happen when we start doing more complex things? Like perhaps not output an `integer` but we output a `tibble`?

- We can fix this by using the standard `map` function and then have the the output be a column of `lists`.

```
addition_tbl <- function(i){
    if(i %% 2)  {
    out <- i*2
  } else {
    out <- i*3
  }

  res <- tibble(was_div_two = i %% 2, out)

  return(res)
}
```

```
tibble(x = c(1:10)) %>%
  mutate(res = map(x, addition_tbl)) %>%
  head
```

```
## # A tibble: 6 × 2
##         x res
##     <int> <list>
## 1       1 <tibble [1 × 2]>
## 2       2 <tibble [1 × 2]>
## 3       3 <tibble [1 × 2]>
## 4       4 <tibble [1 × 2]>
## 5       5 <tibble [1 × 2]>
## 6       6 <tibble [1 × 2]>
```

Basics of purrr: map and tibble

What is going to happen when we start doing more complex things? Like perhaps not output an `integer` but we output a `tibble`?

- We can fix this by using the standard `map` function and then have the the output be a column of `lists`. Then using `unnest` to unnest our *nested* `tibble`.

```
addition_tbl <- function(i){
    if(i %% 2)  {
    out <- i*2
  } else {
    out <- i*3
  }

  res <- tibble(was_div_two = i %% 2, out)

  return(res)
}
```

```
tibble(x = c(1:10)) %>%
  mutate(res = map(x, addition_tbl)) %>%
  head %>%
  unnest(cols = c(res))
```

```
## # A tibble: 6 × 3
##         x was_div_two   out
##     <int>       <dbl> <dbl>
## 1       1           1     2
## 2       2           0     6
## 3       3           1     6
## 4       4           0    12
## 5       5           1    10
## 6       6           0    18
```

TIVAN

> Do something with map_* chr and lgl

Basics of purrr: pmap

Last of the powerful `map` functions is `pmap`, which means *parallel* mapping, not to be confused with executing in parallel.

```
df <- tibble(x = rnorm(5),
             y = rnorm(5),
             z = rnorm(5))

pmap(df, sum)
```

So, we can see that `pmap` executes a *function* across a row.

Basics of purrr: pmap

Executing across a row can be very useful if you have functions with multiple inputs and dont want to specify them all.

```
df <- tibble(x = rnorm(5),
             y = rnorm(5),
             z = rnorm(5))

df %>%
  mutate(output = pmap_dbl(., sum))
```

```
df <- tibble(x = rnorm(5),
             y = rnorm(5),
             z = rnorm(5),
             a = rnorm(5))

product_all <- function(x, y, z){
  x * y * z
}

df %>%
  mutate(output = pmap_dbl(list(x, y, z), product_all))
```

TIVAN

pmap function

Other functions in purrr

The `purrr` library is not just about using `map` function on lists. It also has a whole range of amazing functions to help filter and manipulate lists. Although you might not use all of these all the time, they are good to know.

- Filter
    - `pluck` & `chuck`
    - `keep`
    - `discard`
    - `compact`
- Reshaping
    - `flatten`
    - `nest`
    - `group_nest`
- Manipulate
    - `every`
    - `some`
    - `has_element`
    - `detect`
    - `detect_index`
- Combine
    - `append`
    - `prepend`
    - `cross_df`
    - `reduce`
    - `accumulate`

Back to scooby 🐶

Back to scooby 🐶

```r
scoobydoo ← read_csv("data/scoobydoo.csv")
```

Show 3 entries Search:

index	series_name	network	season	title	imdb	engagement	date_aired	run_time
1	Scooby Doo, Where Are You!	CBS	1	What a Night for a Knight	8.1	556	1969-09-13	2
2	Scooby Doo, Where Are You!	CBS	1	A Clue for Scooby Doo	8.1	479	1969-09-20	22
3	Scooby Doo, Where Are You!	CBS	1	Hassle in the Castle	8	455	1969-09-27	2

Showing 1 to 3 of 603 entries

Previous 1 2 3 4 5 … 201 Next

What if we want to investigate seasons?

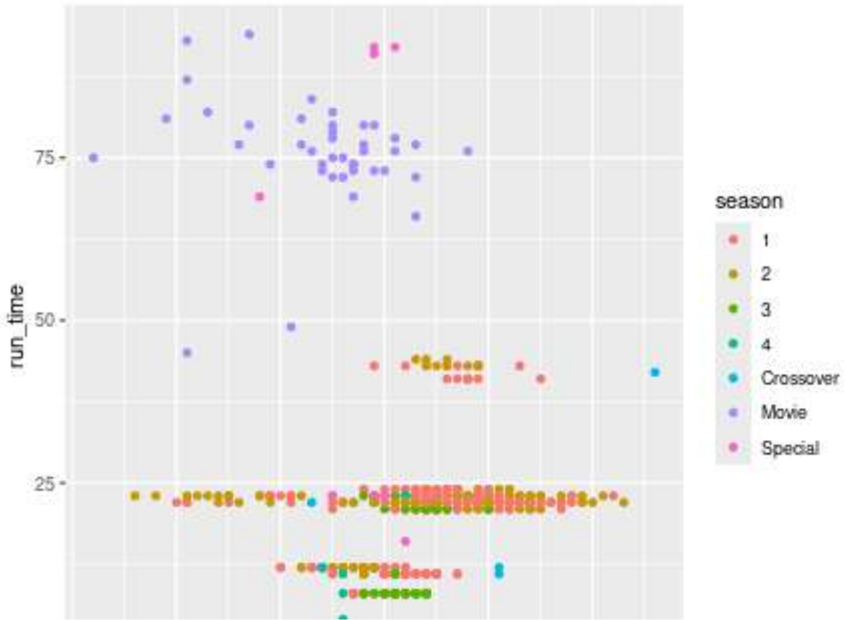
Lets start by investigating the relationship between run time and imdb ratings?

```
df ← scoobydoo %>% select(season, imdb, run_time) %>%
  mutate(imdb = as.numeric(imdb)) %>%
  drop_na()
```

```
lm(imdb ~ run_time, data = df)
```

```
##
## Call:
## lm(formula = imdb ~ run_time, data = df)
##
## Coefficients:
## (Intercept)     run_time
##    7.468733    -0.008102
```

```
df %>%
  ggplot(., aes(imdb, run_time, color = season)) +
  geom_point()
```



What if we want to investigate seasons?

Lets start by investigating the relationship between run time and imdb ratings?

```r
lm_seasons<- function(season_information){
  lm(imdb ~ run_time, data = season_informat
}

lm_tidy<- function(lm_model){
  lm_model %>% broom::tidy() %>%
    filter(term == "run_time")
}
```

```r
scoobydoo %>% select(season, imdb, run_time) %>%
  mutate(imdb = as.numeric(imdb)) %>%
  drop_na() %>%
  group_nest(season) %>%
  filter(map_lgl(data, ~nrow(.x)> 20)) %>%
  mutate(
    lm_res = map(data, lm_seasons),
    lm_beta = map(lm_res, lm_tidy)
    )
```

```
## # A tibble: 4 × 4
##   season                data lm_res lm_beta
##   <chr>  <list<tibble[,2]>> <list> <list>
## 1 1                [311 × 2] <lm>   <tibble [1 × 5]>
## 2 2                [149 × 2] <lm>   <tibble [1 × 5]>
## 3 3                 [60 × 2] <lm>   <tibble [1 × 5]>
## 4 Movie             [42 × 2] <lm>   <tibble [1 × 5]>
```

What if we want to investigate seasons?

We can now `unnest` the output of our functions to see what the relationship is:

```
scoobydoo %>% select(season, imdb, run_time) %>%
  mutate(imdb = as.numeric(imdb)) %>%
  drop_na() %>%
  group_nest(season) %>%
  filter(map_lgl(data, ~nrow(.x)> 20)) %>%
  mutate(
    lm_res = map(data, lm_seasons),
    lm_beta = map(lm_res, lm_tidy)
    ) %>%
  unnest(lm_beta) %>%
  janitor::clean_names()
```

```
## # A tibble: 4 × 8
##   season                data lm_res term     estimate std_error statistic  p_value
##   <chr>  <list<tibble[,2]>> <list> <chr>       <dbl>     <dbl>     <dbl>    <dbl>
## 1 1               [311 × 2] <lm>   run_ti…   0.0297   0.00421     7.04  1.22e-11
## 2 2               [149 × 2] <lm>   run_ti…   0.0230   0.00881     2.61  1.00e- 2
## 3 3                [60 × 2] <lm>   run_ti…   0.0171   0.00459     3.72  4.45e- 4
## 4 Movie            [42 × 2] <lm>   run_ti…  -0.0108   0.0138     -0.784 4.38e- 1
```

TIVAN

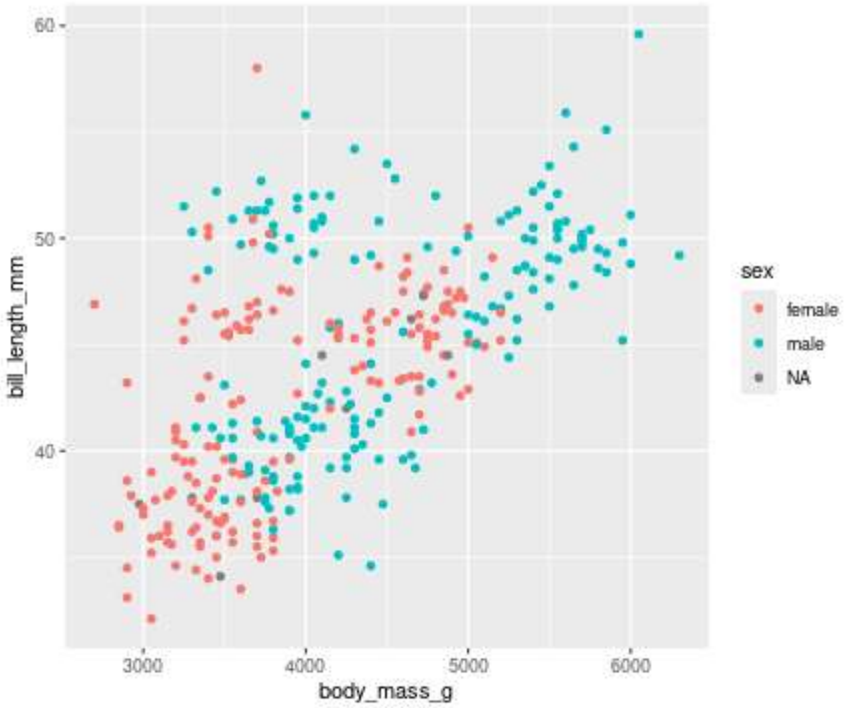
- Execute a pmap idea

From Excel to R
(Session 4-1 - Xaringan)

Making slides like a ninja 🤺

Mixing R and output

```
library(palmerpenguins)
penguins %>%
  ggplot(., aes(body_mass_g, bill_length_mm, color = sex)) +
  geom_point()
```



This is a transition slide

Nothing wrong with adding a GIF


Nice background

Or perhaps two pictures side by side

Lets start

What is {xaringan}?

- With xaringan you can easily generate HTML5 presentations.
- The `xaringan` package is an R Markdown extension based on the JavaScript library remark.js.
- To learn more about `xaringan`, review the excellent `xaringan` introduction from the package's author Yihui Xi.

> The name "xaringan" came from Sharingan (http://naruto.wikia.com/wiki/Sharingan) in the Japanese manga and anime "Naruto." The word was deliberately chosen to be difficult to pronounce for most people (unless you have watched the anime), because its author (me) loved the style very much, and was concerned that it would become too popular.
>
> — Yihui

Installing {xaringan} and creating slide

- As you should know by now, its not very difficult to install packages in `R`

```
install.packages("xaringan")
```

- Defining a new slide:

```
---
class: .large

# Installing {xaringan}
```

- Defining a specific type of slide:

```
---
class: clear, no_number, transition

# Lets start
```

Changing the CSS

We define the CSS output in a `.css` (*cascading style sheet*) file. In your projects, you can change the color of your top bar by changing the following line in the `gen_theme.css` file:

```
.remark-slide-content {
  background-color: #FFFFFF;
  border-top: 80px solid #01524a;
  font-size: 20px;
  font-weight: 300;
  line-height: 1.5;
  padding: 1em 2em 1em 2em
}
```

- There is a lot more to explore! So play around in the `gen_theme.css` file to find out how I customized the slides!

Rate the course!

https://forms.gle/ZEE5xVk7HYnphZQx5