

Resources

Data analyst

A data analyst collects, transforms, and organizes data to draw conclusions, make predictions, and drive informed decision-making. The most popular programming languages used by data analysts are R and Python.

R offers convenient statistical features for data analysis and is useful for creating advanced data visualizations. Check out these resources to learn more about R:

- [The R Project for Statistical Computing](#): a website for downloading R, documentation, and help
- [R Manuals](#): links to manuals from the R core team, including introduction, administration, and help
- [Coding Club R Tutorials](#): a collection of coding tutorials for R
- [R for Beginners](#): a starting guide to help you work with data, graphics, and statistics in R
- **Posit (RStudio)**: The best place to find help with R is in R itself! You can input '?' or the help() command to search in R. You can also open the Help pane to find more R resources.
- **Posit Blog**: Posit's blog is a great place to find information about RStudio, including company news. You can read the most recent **featured posts** or use the search bar and the list of categories on the left side of the page to explore specific topics you might find interesting or to search for a specific post.
- **Stack Overflow**: The Stack Overflow blog posts opinions and advice from other coders. This is a great place to stay in touch with conversations happening in the community.
- **R-Bloggers**: The R-Bloggers blog has useful tutorials and news articles posted by other R users in the community.
- **R-Bloggers' tutorials for learning R**: This blog post from R-Bloggers compiles some basic R tutorials and also links to more advanced guides.

Note: In any program: readability is important as you would be working within a team

-> looking through your code to see if you have coded sth multiple times, or use the same logic to consolidate the code

-> Use comment

-> Use documentation to explain in depth what your code is doing, why it was built, what is the purpose for it, and any limitations.

-> make your code available to be scalable and dynamic (able to work with large and small dataset and can be changed base on your purpose)

Python is a general-purpose language that you can use to create what you need for data analysis. Here are a few resources to begin learning Python:

- [The Python Software Foundation \(PSF\)](#): a website with guides to help you get started as a beginner
- [Python Tutorial](#): a Python 3 tutorial from the PSF site
- [Coding Club Python Tutorials](#): a collection of coding tutorials for Python

Kaggle is an online repository of various datasets that can be used in both R and Python. It's a robust platform that regularly hosts solution-based competitions using data sets in high-interest industries. Learners may also explore a vast trove of data modeling discussions, trending plug-in models, and useful code snippets. Here are some great resources to get started in Kaggle:

- [Datasets](#): explore and download a vast collection of data sets while up-voting your favorite collection.
- [Competitions](#): commit individually or collaborate in a team towards data competitions for the possibility of financial rewards. Even without winning the competitions, this is a great way to network with other analysts.
- [Learn](#): use this resource for an additional perspective on data visualization, linear regression techniques, or time series charting code.

Web designer

A web designer is responsible for the styling and layout of web pages containing text, graphics, and video. Web designers generally use Hypertext Markup Language v5 (HTML5) and Cascading Style Sheets (CSS) to create web pages.

HTML5 provides structure for web pages and is used to connect to hosting platforms. Learn more about HTML5 and CSS using these resources:

- [HTML Tutorial](#): an introduction to HTML with links to HTML5 features, examples, and references
- [HTML5 Cheat Sheet](#): a handy summary of HTML5 tags, attributes, and compatibility with HTML4
- [HTML5 and CSS Fundamentals course](#): a free W3C course on edX; a verified course certificate can be issued for \$199

CSS is used for web page design and controls graphic elements (color, layout, and font) and page presentation on multiple devices (large screens, mobile screens, and printers). Check out these cheat sheets for CSS:

- [Interactive CSS Cheat Sheet](#): includes the most common CSS snippets for gradient, background, font-family, border, and much more
- [50 Best HTML & CSS Cheat Sheets](#): a list of 50 cheat sheets—choose a few that are useful to you

Mobile application developer

A mobile application developer uses programming to create applications used on laptops, mobile phones, and tablets. The most popular programming languages for mobile application developers are Swift, Java, and C#.

Swift (for Apple platforms) is an open source scripting language for macOS, iOS, watchOS, and tvOS. Its main goal is to make applications run faster. Browse these resources for more information about Swift:

- [Swift.org](#): an open source community with resources to learn how to use Swift, including videos and sample code
- [Swift developer site](#): an Apple developer website with information for developers who want to use Swift
- [Swift development resources](#): Apple's collection of documentation, sample code, videos, and recommended books

Java (for Android devices) is the official language for Android development. The article [I want to develop Android apps - which languages should I learn?](#) explores some other languages used for Android development. Check out these resources for Java:

- [Android Studio](#): a downloadable integrated development environment (IDE) with tools to build apps for Android devices
- [Build your first Android app in Java](#): instructions for installing Android Studio and creating your first app
- [Java tutorial for beginners: write a simple app with no previous experience](#): an overview of how to learn Java, with examples

C# (pronounced C-sharp) is an object-oriented programming language that is widely used to create mobile apps in the .NET open source developer platform. Xamarin extends the .NET platform with a framework for developers to create cross-platform mobile apps for both iOS and Android. Here are a few resources to help you learn C#:

- [Microsoft .NET learning materials for C#](#): includes free courses, tutorials, and videos to learn the programming language C#
- [Microsoft Xamarin learning materials](#): includes free courses, tutorials, and videos to learn about mobile development with Xamarin

- [Xamarin Tutorial - build your first iOS or Android app in C#](#): instructions for building a mobile app that displays the text “Hello World”
- [Learn C# from Codecademy](#): a website with free basic interactive lessons, and additional activities that can be accessed with a monthly subscription

Web application developer

A web application developer designs and develops network applications used across the web. The most popular programming languages used by web application developers are Java, Python, Ruby, and PHP.

Java is widely used to create enterprise web applications that can run on multiple clients. Java’s main strength is its “Write Once, Run Anywhere” (WORA) approach. Browse these resources to learn more about Java:

- [Oracle Java Tutorials](#): Java tutorials from Oracle documentation
- [Java for Beginners](#): a free Java course for beginners from the website “Home and Learn”

Python is a general-purpose programming language. Check out the Python resources listed in the data analyst section.

Ruby is a general-purpose, object-oriented programming language used for web application development. Ruby isn't the same as Ruby on Rails, which is an open source web application framework that runs using Ruby. Browse these resources to learn more about Ruby:

- [Ruby news](#): information about the latest Ruby releases and links to other resources
- [Ruby documentation](#): includes guides, tutorials, and reference material to help you learn more about Ruby
- [Ruby programmer’s guide](#): a tutorial and reference guide for Ruby
- [Learn Ruby from Codecademy](#): a website with free basic interactive lessons, and additional activities that can be accessed with a monthly subscription

PHP is a scripting language particularly suited for web application development. It was based on Perl, another programming language. PHP is simple, flexible, and relatively easy to learn. Check out these resources to learn more about PHP:

- [PHP downloads and documentation](#): information about the latest PHP releases and links to other resources
- [PHP the Right Way](#): a quick reference for popular PHP coding standards
- [Interactive PHP tutorial](#): a free tutorial that runs PHP code in exercises

Game developer

A game developer is an application developer who specializes in video game creation. Game developers most commonly use the programming languages C# and C++.

C# is an object-oriented programming language that is widely used to create games. Check out the C# resources listed in the mobile application developer section.

C++ is an extension of the C programming language that is also used to create console games, like those for Xbox. Browse more information about C++:

- [Microsoft resources for C++](#): learn how to install the Visual Studio IDE and write C++ code
- [Microsoft C++ and C# code samples for gaming](#): a resource with over 40 C++ and C# code samples for gaming
- [Interactive C++ tutorial](#): a free tutorial that runs C++ code in exercises

Reason to learn R

R is designed for data analysis. Reproducing code is easy (code can be modified and shared by anyone who uses it)

When the data is spread across multiple categories or groups, it is best to use R.

For example, imagine you are analyzing sales data for every city across an entire country, each city has its own group of data.

=> Using RStudio makes it easy to take a specific analysis step and perform it for each group using basic code. In this example, you could calculate the yearly average sales data for every city.

RStudio also allows for flexible data visualization. You can visualize differences across the cities effectively using plotting features like facets—which you'll learn more about later on.

You can also use RStudio to automatically create an output of summary stats—or even your visualized plots—for each group.

Click Help>Shortcut help to learn about shortcut in R

Note: R is case-sensitive.

There are two main ways of writing code in RStudio:

- Using the console to type commands directly into the console, but they'll be forgotten when you close your current session.
- Using **the source editor**. If you save your script in the editor, you can reaccess your work and show others how you did it.

RStudio vs RCloud:

- RStudio Desktop allows you to use RStudio locally, even if you aren't connected to the internet.
- RStudio Cloud gives you the flexibility of accessing your account from any computer.

Where to ask: R community

- [RStudio Community](#): The RStudio Community forum is a great place to get help and find solutions to challenges you have with R—and maybe help someone else out, too!
- [r/RLanguage](#): The R language subreddit is an active online community on the social media platform Reddit, where R users go to discuss R, ask questions, and share tips.
- [rOpenSci](#): rOpenSci has a community forum where R users can ask questions and search for solutions. It also includes links to their Best Practices guide and support pages.
- [R4DS Online Learning Community and Slack channel](#): This is a community with another Slack channel where R learners and mentors can gather and connect. This is a great place to chat about using R for data science.
- [Twitter #rstats](#): If you use Twitter, you can connect with other R users using the hashtag #rstats; a lot of R developers and analysts are active on Twitter.

Programming fundamentals

Basic concepts:

- Functions: like `print()`, `paste`
- Argument: Information needed by a function in R in order to run
- Comments: to describe what's going on your code. Eg. `#abc`
- Variables: eg. `x <- abcxyz`
- Data types
- Vectors: `c(x,y,z)`. (x,y,z: same type)
To create vector:

```
vec_1 <- c(13, 48, 71)
```
- Pipes: expressing a sequence of multiple operations, represented with “`%>%`”
Eg.

```
ToolGrowth %>%  
Filter(dose== 0.5) %>%  
Arrange(len)
```

Note: get to know more about each functions by type “`?[function name]`” eg. `?print()`

Vector and list in R

There are two types of vectors: **atomic vectors** and **lists**.

Atomic vector

Types of atomic vector:

Type	Description	Example
Logical	True/False	TRUE
Integer	Positive and negative whole values	3
Double	Decimal values	101.175
Character	String/character values	"Coding"

Creating vector

Use combine function:

- Numeric data: `c(2.5, 48.5, 101.5)`
For integer: `c(1L, 5L, 15L)`
- characters or logicals:
`c("Sara", "Lisa", "Anna")`
`c(TRUE, FALSE, TRUE)`

Determining the properties of vectors

Eg1. <code>typeof(c("a", "b"))</code> <code>#> [1] "character"</code>	Eg2. <code>x <- c(33.5, 57.75, 120.05)</code> <code>length(x)</code> <code>#> [1] 3</code>	<i>Check if a vector is a specific type</i> <code>x <- c(2L, 5L, 11L)</code> <code>is.integer(x)</code> <code>#> [1] TRUE</code>
---	--	---

Naming vector by using name() function

```
x <- c(1, 3, 5)
names(x) <- c("a", "b", "c")
```

Run the code

x

```
#> a b c
```

```
#> 1 3 5
```

List

If you want to store elements of different types in the same data structure, you can use a list.

List function: `list()`

```
list("a", 1L, 1.5, TRUE)
```

```
list(list(list(1, 3, 5)))
```

Determining the structure of lists

```
str(list("a", 1L, 1.5, TRUE))
```

Run the code

```
#> List of 4
```

```
#> $ : chr "a"
```

Commented [NL1]: character

```
#> $ : int 1
```

```
#> $ : num 1.5
```

```
#> $ : logi TRUE
```

Naming list

Use `list()` function

```
list('Chicago' = 1, 'New York' = 2, 'Los Angeles' = 3)
```

```
$`Chicago`
```

```
[1] 1
```

```
$`New York`
```

```
[1] 2
```

```
$`Los Angeles`
```


[1] 3

To learn more about vectors and lists, check out [R for Data Science, Chapter 20: Vectors](#)

Date and times in R

Open Rstudio.

(If you haven't already installed tidyverse, you can use the `install.packages()` function to do so:

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

Next, load the tidyverse and lubridate packages using the `library()` function.

First, load the core tidyverse to make it available in your current R session:

```
library(tidyverse)
library(lubridate)
```

Working with dates and times

Types

- A date ("2016-08-16")
- A time within a day ("20:11:59 UTC")
- And a date-time. This is a date plus a time ("2018-03-31 18:15:48 UTC")

`today()`

`now()`

```
#> [1] "2021-01-20"
```

```
#> [1] "2021-01-20 16:25:05 UTC"
```

When working with R, there are three ways you are likely to create date-time formats.

From strings:

```
ymd("2021-01-20")
#> [1] "2021-01-20"
```

```
mdy("January 20th,
2021")
#> [1] "2021-01-20"
```

```
dmy("20-Jan-2021")
#> [1] "2021-01-20"
```

```
ymd(20210120)
#> [1] "2021-01-20"
```

Create a date-time from a date

Add an underscore and one or more of the letters *h*, *m*, and *s* (hours, minutes, seconds)

```
ymd_hms("2021-01-20 20:11:59")
```

Commented [NL2]: Universal Time Coordinated (world regulates clocks and time.)

```
#> [1] "2021-01-20 20:11:59 UTC"
```

```
mdy_hm("01/20/2021 08:01")
```

```
#> [1] "2021-01-20 08:01:00 UTC"
```

Switch between a date-time and a date.

```
as_date(now())
```

```
#> [1] "2021-01-20"
```

Additional sources

- [lubridate.tidyverse](#): This is the “lubridate” entry from the official tidyverse documentation, which offers a comprehensive reference guide to the various tidyverse packages. Check out this link for an overview of key concepts and functions.
- [Dates and times with lubridate: Cheat Sheet](#)

Other common data structures

Data frame

	carat	cut	color	clarity	depth	table	price	x	y	z
1	0.23	Ideal	E	SI2	61.5	55.0	326	3.95	3.98	2.43
2	0.21	Premium	E	SI1	59.8	61.0	326	3.89	3.84	2.31
3	0.23	Good	E	VS1	56.9	65.0	327	4.05	4.07	2.31
4	0.29	Premium	I	VS2	62.4	58.0	334	4.20	4.23	2.63
5	0.31	Good	J	SI2	63.3	58.0	335	4.34	4.35	2.75
6	0.24	Very Good	J	VVS2	62.8	57.0	336	3.94	3.96	2.48
7	0.24	Very Good	I	VVS1	62.3	57.0	336	3.95	3.98	2.47
8	0.26	Very Good	H	SI1	61.9	55.0	337	4.07	4.11	2.53
9	0.22	Fair	E	VS2	65.1	61.0	337	3.87	3.78	2.49
10	0.23	Very Good	H	VS1	59.4	61.0	338	4.00	4.05	2.39
11	0.30	Good	J	SI1	64.0	55.0	339	4.25	4.28	2.73
12	0.23	Ideal	J	VS1	62.8	56.0	340	3.93	3.90	2.46
13	0.22	Premium	F	SI1	60.4	61.0	342	3.88	3.84	2.33
14	0.31	Ideal	J	SI2	62.2	54.0	344	4.35	4.37	2.71
15	0.20	Premium	E	SI2	60.2	62.0	345	3.79	3.75	2.27
16	0.32	Premium	E	I1	60.9	58.0	345	4.38	4.42	2.68
17	0.30	Ideal	I	SI2	62.0	54.0	348	4.31	4.34	2.68
18	0.30	Good	J	SI1	63.4	54.0	351	4.23	4.29	2.70
19	0.30	Good	J	SI1	63.8	56.0	351	4.23	4.26	2.71
20	0.30	Very Good	J	SI1	62.7	59.0	351	4.21	4.27	2.66
21	0.30	Good	I	SI2	63.3	56.0	351	4.26	4.30	2.71

Showing 1 to 22 of 53,940 entries, 10 total columns

- First, columns should be named.
- Second, data frames can include many different types of data, like numeric, logical, or character.
- Finally, elements in the same column should be of the same type.

To create data frame: use `data.frame()`

```
data.frame(x = c(1, 2, 3), y = c(1.5, 5.5, 7.5))
```

in which: x,y: name of the columns

`c(1,2,3)`: vector containing the information of that column

x y

1 1 1.5

2 2 5.5

3 3 7.5

In most cases, you won't need to manually create a data frame yourself, as you will typically import data from another source, such as a .csv file, a relational database, or a software program.

Files

For more information on working with files in R, check out [R documentation: files](#)

1. Use the **dir.create** function to create a new folder, or directory, to hold your files

dir.create ("destination_folder")

2. Use the **file.create()** function to create a blank file

file.create ("new_text_file.txt")

file.create ("new_word_file.docx")

file.create ("new_csv_file.csv")

If the file is successfully created when you run the function, R will return a value of **TRUE** (if not, R will return **FALSE**).

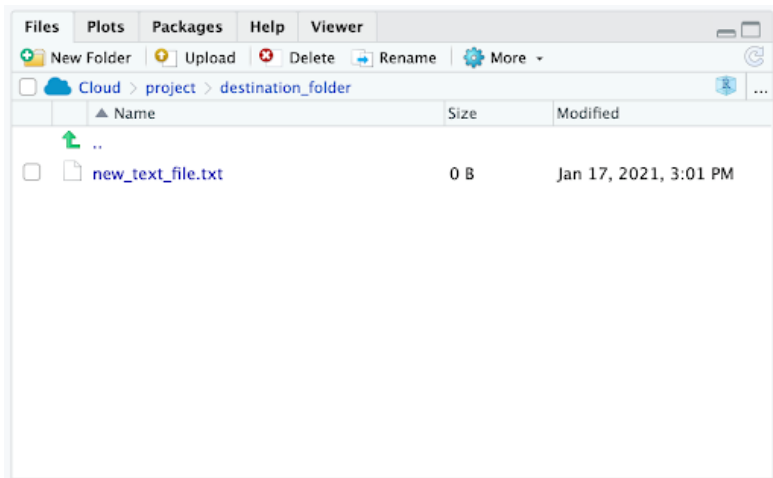
file.create ("new_csv_file.csv")

[1] TRUE

3. Copying a file can be done using the **file.copy()** function.

file.copy ("new_text_file.txt", "destination_folder")

If you check the Files pane in RStudio, a copy of the file appears in the relevant folder:



You can delete R files using the **unlink()** function.

```
unlink ("some_file.csv")
```

If you want to learn more about working with data frames, matrices, and arrays in R, check out the [Data Wrangling](#)

Matrices

To create a matrix in R, you can use the **matrix()** function. First, add a vector. Next, add at least one matrix dimension. You can choose to specify the number of rows or the number of columns by using the code **nrow =** or **ncol =**.

Example:

```
matrix(c(3:8), nrow = 2)
```

Run the R code, we get:

```
[,1] [,2] [,3]
```

```
[1,] 3 5 7
```

```
[2,] 4 6 8
```

```
matrix(c(3:8), ncol = 2)
```

Run the R code, we get:

```
[,1] [,2]
```

```
[1,] 3 6
```

```
[2,] 4 7
```

```
[3,] 5 8
```

Operator and calculations

Assignment operator: <-

Operator	Description	Example Code (after the sample code below, typing x will generate the output in the next column)	Result/ Output
<-	Leftwards assignment	x <- 2	[1] 2
<<-	Leftwards assignment	x <<- 7	[1] 7
=	Leftwards assignment	x = 9	[1] 9
->	Rightwards assignment	11 -> x	[1] 11
->>	Rightwards assignment	21 ->> x	[1] 21

Arithmetic operator: +, -, *, /

x <- 2

y <- 5

Operator	Description	Example Code	Result/ Output
+	Addition	x + y	[1] 7
-	Subtraction	x - y	[1] -3
*	Multiplication	x * y	[1] 10
/	Division	x / y	[1] 0.4
%%	Modulus (returns the remainder after division)	y %% x	[1] 1
%/%	Integer division (returns an integer value after division)	y %/% x	[1] 2
^	Exponent	y ^ x	[1] 25

Relational operators, also known as comparators, allow you to compare values.

Operator	Description	Example Code	Result/Output
<	Less than	x < y	[1] TRUE
>	Greater than	x > y	[1] FALSE
<=	Less than or equal to	x <= 2	[1] TRUE
>=	Greater than or equal to	y >= 10	[1] FALSE

Operator	Description	Example Code	Result/Output
==	Equal to	y == 5	[1] TRUE
!=	Not equal to	x != 2	[1] FALSE

Logical operators

Logical operators return a logical data type such as TRUE or FALSE.

- AND (sometimes represented as & or && in R)
- OR (sometimes represented as | or || in R)
- NOT (!)

The NOT operator (!) simply negates the logical value it applies to. In other words, !TRUE evaluates to **FALSE**, and !FALSE evaluates to **TRUE**.

Zero is considered FALSE and non-zero numbers are taken as TRUE.

Eg: x<-2

```
> !x [1] FALSE #as x=TRUE => !x=FALSE
```

Mini case:

Given a data frame:

	Ozone	Solar.R	Wind	Temp	Month	Day
1	41	190	7.4	67	5	1
2	36	118	8.0	72	5	2
3	12	149	12.6	74	5	3
4	18	313	11.5	62	5	4

AND example: Solar.R > 150 & Wind > 10.

	Ozone	Solar.R	Wind	Temp	Month	Day
1	18	313	11.5	62	5	4

OR example: Solar.R > 150 | Wind > 10.

	▲ Ozone ↕	Solar.R ↕	Wind ↕	Temp ↕	Month ↕	Day ↕
1	41	190	7.4	67	5	1
2	12	149	12.6	74	5	3
3	18	313	11.5	62	5	4

NOT example: **Day != 1**. (days are not 1)

	▲ Ozone ↕	Solar.R ↕	Wind ↕	Temp ↕	Month ↕	Day ↕
1	36	118	8.0	72	5	2
2	12	149	12.6	74	5	3
3	18	313	11.5	62	5	4

!(Solar.R > 150 | Wind > 10) (either a *Solar* measurement greater than 150 or a *Wind* measurement greater than 10 = not true)

Commented [NL3]: Or we can say: both

	▲ Ozone ↕	Solar.R ↕	Wind ↕	Temp ↕	Month ↕	Day ↕
1	36	118	8.0	72	5	2

Conditional statements

if() and else() statement:

```
x <- 7
```

```
if (x > 0) {
```

```
  print ("x is a positive number")
```

```
} else {
```

```
  print ("x is either a negative number or zero")
```

```
}
```

else if statement

```
if (condition1) {
```

```
  expr1
```

```

} else if (condition2) {

  expr2

} else {

  expr3

}

```

To learn more about logical operators and conditional statements, check out DataCamp's tutorial [Conditionals and Control Flow in R](#)

Keeping Your Code Readable

Style

Using a clear and consistent coding style generally makes your code easier for others to read

	Guidance	Examples of best practice	Examples to avoid
Files	File names should be meaningful and end in .R. Avoid using special characters in file	# Good explore_penguins.R annual_sales.R	# Bad Untitled.r stuff.r

	names—stick with numbers, letters, dashes, and underscores.		
Object names	Variable and function names should be lowercase. Use an underscore _ to separate words within a name. Try to create names that are clear, concise, and meaningful. Generally, variable names should be nouns.	# Good day_one	# Bad DayOne
	Function names should be verbs.	# Good add ()	# Bad addition ()

Syntax

	Guidance	Examples of best practice	Examples to avoid
Spacing	Most operators (==, +, -, <-, etc.) should be surrounded by spaces.	# Good x == y a <- 3 * 2	# Bad x==y a<-3*2
	Always put a space <i>after</i> a comma (never before).	# Good y[, 2]	# Bad y[,2] y[,2]

	Do not place spaces around code in parentheses or square brackets (unless there's a comma, in which case see above).	# Good if (debug) do(x) species["dolphin",]	# Bad if (debug) do(x) species["dolphin" ,]
	Place a space before left parentheses, except in a function call.	# Good sum(1:5) plot(x, y)	# Bad sum (1:5) plot (x, y)
Curly braces	An opening curly brace should never go on its own line and should always be followed by a new line. A closing curly brace should always go on its own line (unless it's followed by an else statement). Always indent the code inside curly braces.	# Good x <- 7 if (x > 0) { print("x is a positive number") } else { print ("x is either a negative number or zero") }	# Bad x <- 7 if (x > 0) { print("x is a positive number") } else { print ("x is either a negative number or zero") }
Indentation	When indenting your code, use two spaces. Do not use tabs or mix tabs and spaces.	-	-
Line length	Try to limit your code to 80 characters per line. This fits nicely on a printed page with a reasonably sized font.	-	-

Embracing

The embracing operator, { { } }, should always have inner spaces to help emphasise its special behaviour:

```
# Good
max_by <- function(data, var, by) {
  data %>%
    group_by({{ by }}) %>%
    summarise(maximum = max({{ var }}, na.rm = TRUE))
}

# Bad
max_by <- function(data, var, by) {
  data %>%
    group_by({{by}}) %>%
    summarise(maximum = max({{var}}, na.rm = TRUE))
}
```

Infix operators

Most infix operators (`=`, `+`, `-`, `<=`, etc.) should always be surrounded by spaces:

```
# Good
height <- (feet * 12) + inches
mean(x, na.rm = TRUE)

# Bad
height<-(feet*12)+inches
mean(x, na.rm=TRUE)
```

except for: `::`, `:::`, `$`, `@`, `[`, `[[`, `^`, unary `-`, unary `+`, and `:`.

```
# Good
sqrt(x^2 + y^2)
df$z
x <- 1:10

# Bad
sqrt(x ^ 2 + y ^ 2)
df $ z
x <- 1 : 10
```

Resources

- Check out this tidyverse style guide to get a more comprehensive breakdown of the most important stylistic conventions for writing R code (and working with the tidyverse).
- The [styler package](#) is an automatic styling tool that follows the tidyverse formatting rules.

Install packages

Packages in R include reusable R functions, documentation about how to use the functions, sample datasets, and tests for checking your code.

In Rstudio:

`installed.packages()` → shows a list of packages currently installed in an RStudio session

“based”: have loaded

“recommended”: have installed, but not yet loaded

How to load: `library([name of the pack])`

Get to know more about packages: click the name of the packages in the Tab session

Or type: `browseVignettes("packagename")`

How to choose the right packages

- **Tidyverse**: the tidyverse is a collection of R packages specifically designed for working with data. It's a standard library for most data analysts, but you can also download the packages individually.

- **Quick list of useful R packages:** this is RStudio Support's list of useful packages with installation instructions and functionality descriptions.
- **CRAN Task Views:** this is an index of CRAN packages sorted by task. You can search for the type of task you need to perform and it will pull up a page with packages related to that task for you to explore.

The tidyverse

tidyverse packages help you read, manipulate, visualize, and do many other important things with data.

To update tidyverse: `tidyverse_update()`

Update 1 package: `install.packages("package name")`

4 main packages

Ggplot2

Tidyr: use to clean data

Readr: use for importing data

Dplyr: offers consistent sets of function for data manipulation

Pipes

A tool for expressing a sequence of multiple operations, represented with “`%>%`”

Shortcut:

For PC: `ctrl + shift + m`

Mac: `cmd + shift + m`

```
1 data("ToothGrowth")
2 install.packages('dplyr')
3 install.packages("dplyr")
4 library("dplyr")
5
6 filtered_TG <- ToothGrowth %>%
7   filter(dose==0.5) %>%
8   arrange(len)
9
```

Instead of call out a new df, and then filter, and then arrange

Pipes allow a more readable and avoid starting over dataframe which you can work as a new df without changing the original df

When using pipes:

- Add the pipe operator at the end of each line of the piped operation except the last one
- Check your code after you've programmed your pipe
- Revisit piped operations to check for parts of your code to fix

Model 3: Data in R

R data frames

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

data("diamonds")
#view head of the data frame only
head(diamonds)
#see structure
str(diamonds)
#make change to the df
library(tidyverse) #load the library to load mutate
mutate(diamonds, carat_2 = carat * 100) # add column carat_2, which = carat*2]
```

Creating df in R by using vector:

Note: When creating data frames, columns should be named and each column should contain the same number of data items.

```
install.packages("tidyverse")

library(tidyverse) #load the packages

# create vector

names <- c("A", "B", "C", "D")
age <- c(1,2,3,4)

#create dataframe

people <- data.frame(names, age)
```

View df

```
str(people), head(people), glimpse(people)
```

Add a column by using mutate: `mutate(people, age_in_20 = age + 20)`

Tibbles

Tibbles

- Never change the data types of the inputs
- Never change the names of your variables
- Never create row names
- Make printing easier

Tibbles are like streamlined data frames that are automatically set to pull up only the first 10 rows of a dataset, and only as many columns as can fit on the screen.

```
as_tibble(diamonds)
```

```
# A tibble: 53,940 x 10
  carat cut    color clarity depth table price     x     y     z
<dbl> <ord> <ord> <ord>    <dbl> <dbl> <int> <dbl> <dbl> <dbl>
1  0.23 Ideal  E     SI2      61.5    55   326   3.95   3.98   2.43
2  0.21 Prem... E     SI1      59.8    61   326   3.89   3.84   2.31
3  0.23 Good   E     VS1      56.9    65   327   4.05   4.07   2.31
4  0.290 Prem... I     VS2      62.4    58   334   4.2    4.23   2.63
5  0.31 Good   J     SI2      63.3    58   335   4.34   4.35   2.75
6  0.24 Very... J     VVS2     62.8    57   336   3.94   3.96   2.48
7  0.24 Very... I     VVS1     62.3    57   336   3.95   3.98   2.47
8  0.26 Very... H     SI1      61.9    55   337   4.07   4.11   2.53
9  0.22 Fair   E     VS2      65.1    61   337   3.87   3.78   2.49
10 0.23 Very... H     VS1      59.4    61   338   4      4.05   2.39
# ... with 53,930 more rows
```

While RStudio's built-in data frame tool returns thousands of rows in the *diamonds* dataset, the tibble only returns the first 10 rows in a neatly organized table.

The [Tidy chapter](#) in "A Tidyverse Cookbook" is a great resource if you want to learn more about how to work with tibbles using R code. The chapter explores a variety of R functions that can help you create and transform tibbles to organize and tidy your data.

Import data to R

The data() function

`data()` to load these datasets in R

If you run the data function without an argument, R will display a list of the available datasets.

If you want to load a specific dataset, just enter its name in the parentheses of the data() function.

`data(mtcars)`

The readr package

The readr package is part of the core tidyverse. So, if you've already installed the tidyverse, you have what you need to start working with readr.

- **read_csv()**: comma-separated values (.csv) files
- **read_tsv()**: tab-separated values files
- **read_delim()**: general delimited files
- **read_fwf()**: fixed-width files
- **read_table()**: tabular files where columns are separated by white-space
- **read_log()**: web log files

import spreadsheet by readxl package

```
library(readxl)
```

```
read_excel()
```

You can use the [excel_sheets\(\)](#) function to list the names of the individual sheets.

```
excel_sheets(readxl_example("type-me.xlsx"))
```

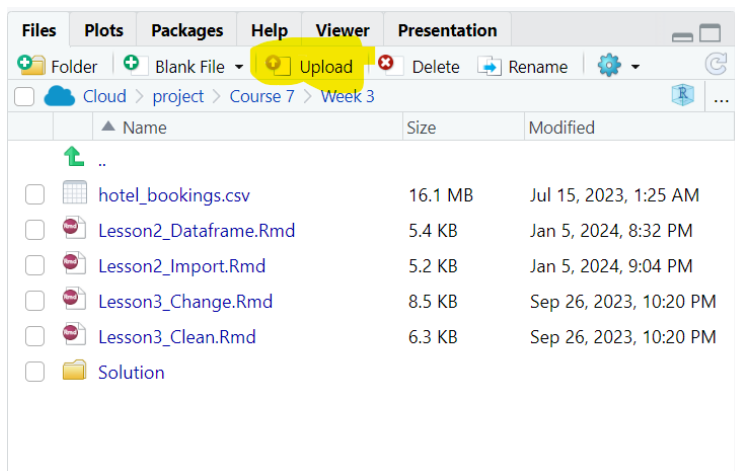
You can also specify a sheet by name or number.

```
read_excel(readxl_example("type-me.xlsx"), sheet = "numeric_coercion")
```

If you want to learn how to use readr functions to work with more complex files, check out the [Data Import chapter](#) of the R for Data Science book. It explores some of the common issues you might encounter when reading files, and how to use readr to manage those issues.

Hands on activity

Upload data into R



Create df:

```
bookings_df <- read_csv("hotel_bookings.csv")
```

Create another data frame using `bookings_df` that focuses on the average daily rate, which is referred to as `adr` in the data frame, and `adults`

```
new_df <- select(bookings_df, 'adr', 'adults')
```

	adr	adults
1	0.00	2
2	0.00	2
3	75.00	1
4	75.00	1
5	98.00	2
6	98.00	2
7	107.00	2
8	103.00	2

Clean data: preview and rename data so that it's easier to work with

Keywords: select, rename, clean_names, skim_without_chart

#package makes referencing files easier

```
install.packages("here")
```

```
library("here")
```

packages simplify data cleaning tasks

```
install.packages("skimr")
```

```
library("skimr")
```

```
install.packages("janitor")
```



```
library("janitor")

# pipes
install.packages("dplyr")
library("dplyr")

# install penguins data
install.packages("palmerpenguins")
library("palmerpenguins")

#read data
skim_without_charts(penguins)
glimpse(penguins)
head(penguins)

#only need to check species column
penguins %>%
  select(-species) #check everything without species

penguins %>%
  rename(island_new =island)

# change columns name to be more consistent
rename_with(penguins, tolower) #change all to lower case
clean_names(penguins) #automatically make sure that the column names are unique and
consistent
#eg. makes all names to lower case and lower letter
```

Organize data: sort, filter, and summarize your data

```
library(tidyverse)
penguins %>% arrange(bill_depth_mm) #arrange with ascending order
penguins %>% arrange(-bill_depth_mm) #descending

# as this is just in our console, create a df
penguins2 <- penguins %>% arrange(-bill_depth_mm)

# groupby, careful to use the drop_na when applicable since it will skip rows
penguins %>% group_by(island) %>% drop_na() %>% summarize(mean_bill_length_mm =
  mean(bill_length_mm))

#group by island and species and then summarize to calculate both the mean
# and max
penguins %>% group_by(species, island) %>% drop_na() %>%
  summarize(max_bl = max(bill_length_mm), mean = mean(bill_length_mm))

# filter to appear Adelie only
penguins %>% filter(species == "Adelie") |
```

Make a tibble (for easier reading) with arrange in desc: `arrange([df_names], desc([col_names]))`

Find relevant information of each column: `max([df_names]$(col_names))` #apply to min and mean

Keywords: `arrange`, `group_by`, `filter`, `summarize` (max, mean, etc.)

Transforming data: separate, combine data, and create new variables.

Keywords: `unite`, `mutate`, `separate`

unite: Calculate the total number of canceled bookings and the average lead time for booking

```
calculation1_df <- bookings_df %>%
```

```
  summarize(number_canceled = sum(is_canceled), average_lead_time = mean(lead_time))
```

```
combine_df <- bookings_df %>%
```

```
  select(arrival_date_year, arrival_date_month) %>%
```

```
  unite(arrival_month_year, c("arrival_date_year", "arrival_date_month"), sep = " ")
```

mutate: create a new column (without delete original column) that calculate based on the original column

summed up all the adults, children, and babies on a reservation for the total number of people.

```
mutate_df <- bookings_df %>%
```

```
  mutate(guests = adults, children, babies, abc = total_of_special_requests/2, ....)
```

bookings × Lesson3_Clean.Rmd × mutate_df × combine_df × bookings_ >> Filter

id	total_of_special_requests	reservation_status	reservation_status_date	guests
0	2	Canceled	2015-06-29	2
1	0	Check-Out	2015-07-05	2
0	0	Check-Out	2015-07-05	2
0	0	Check-Out	2015-07-05	2
0	2	Canceled	2015-06-16	3
0	2	Canceled	2015-06-18	2
0	1	Canceled	2015-07-03	2
0	1	Canceled	2015-06-12	2

Showing 65 to 72 of 119,390 entries, 33 total columns

separate:

```
#use the separate function to split first and last name into separate columns
# df name col to be separate
separate(employee, name, into=c('first name', 'last name'), sep= ' ')
```

Convert long data to wide data and vice versa

Review: Long data has many rows, wide data has many columns

- `pivot_longer()`: increasing the number of rows and decreasing the number of columns
- `pivot_wider()`: vice versa

To learn more about these two functions and how to apply them in your R programming, check out these resources:

- **Pivoting:** It explores the components of the `pivot_longer` and `pivot_wider` functions using specific details, examples, and definitions.
- **CleanItUp 5: R-Ladies Sydney: Wide to Long to Wide to...PIVOT:** This resource gives you additional details about the `pivot_longer` and `pivot_wider` functions. The examples provided use interesting datasets to illustrate how to convert data from wide to long and back to wide.
- **Plotting multiple variables:** This resource explains how to visualize wide and long data, with `ggplot2` to help tidy it. The focus is on using `pivot_longer` to restructure data and make similar plots of a number of variables at once. You can apply what you learn from the other resources here for a broader understanding of the pivot functions.

File naming conventions

- Keep your filenames to a reasonable length
- Use underscores and hyphens for readability
- Start or end your filename with a letter or number
- Use a standard date format when applicable; example: YYYY-MM-DD
- Use filenames for related files that work well with default ordering; example: in chronological order, or logical order using numbers first

Examples of good filenames

2020-04-10_march-attendance.R

2021_03_20_new_customer_ids.csv

01_data-sales.html

02_data-sales.html

Don't

- Use unnecessary additional characters in filenames
- Use spaces or “illegal” characters; examples: &, %, #, <, or >
- Start or end your filename with a symbol
- Use incomplete or inconsistent date formats; example: M-D-YY
- Use filenames for related files that do not work well with default ordering; examples: a random system of numbers or date formats, or using letters first

Biased data with R

```
install.packages("SimDesign")
library(SimDesign)

actual_temp <- c(68.3, 70, 72.4, 71, 67, 70)
predict_temp <- c(67.9, 69, 71.5, 70, 67, 69)
bias(actual_temp, predict_temp) #avg. (actual - predicted)

#While the outcome is pretty close to 0, prediction seemed biased towards lower temperature
```

sample() function: mix the data set to make a random sample

- **Bias function:** This web page is a good starting point to learn about how the bias function in R can help you identify and manage bias in your analysis.

- **Data Science Ethics:** This online course provides slides, videos, and exercises to help you learn more about ethics in the world of data analytics. It includes information about data privacy, misrepresentation in data, and applying ethics to your visualizations.

Visualization with R

ggplot2

Packages:

- Plotly for general purpose
- RGL: 3D visuals
- Ggplot2 is the fundamental code -> can be reuse with other plotting code

Building blocks of ggplot2:

- A dataset
- A set of geoms: geometric object used to represent your data.
Eg. points to create a scatter plot, bars to create a bar chart, or lines to create a line diagram.
- A set of aesthetic attributes: An aesthetic is a visual property of an object in your plot.
Eg. in a scatterplot, aesthetics include things like the size, shape, color, or location (x-axis, y-axis) of your data points.

Write a ggplot code:

```
ggplot(data=<DATA>)+  
<GEOM_FUNCTION>(mapping=aes(<AESTHETIC MAPPINGS>))
```

```
ggplot(data = penguins) + geom_point(mapping = aes(x = flipper_length_mm, y =  
body_mass_g))
```

or:

```
ggplot(data = penguins, mapping = aes(x = flipper_length_mm, y = body_mass_g))  
+ geom_point()
```

In which:

`ggplot(data = [dataset_names])`

`geom_function():`

`geom_point():` to create a scatter plot.

For a bar chart: use `geom_bar`

For a line chart: `geom_line` and so on

`aes(x = flipper_length_mm, y = body_mass_g)`: how variables in your dataset are mapped to visual properties.

`aes(x = [x-axis], y = [y-axis],`

Labels and annotations: customize your plot

[View ggplot cheatsheet](#) which has everything about ggplot

Facet() function: display smaller groups, or subsets, of your data

Facet by 01 variable:

`Facet_wrap(~species)`

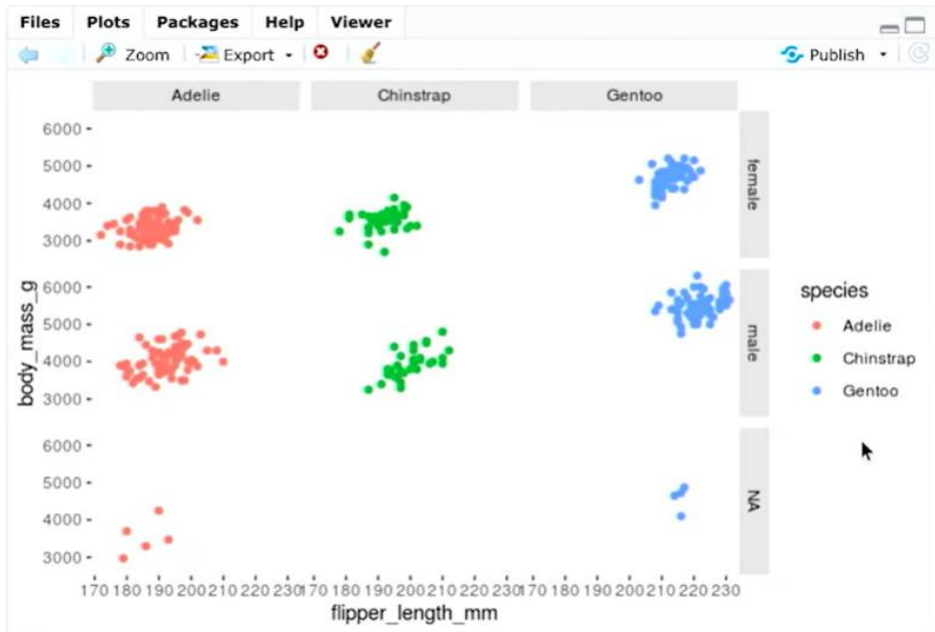
```
ggplot(data=penguins)+  
  geom_point(mapping=aes(x=flipper_length_mm,y=body_mass_g,color=species))+  
  facet_wrap(~species)
```

`Facet_grid()`: split the plot into facets vertically by the values of the first variable and horizontally by the values of the second variable

```
ggplot(data=penguins)+  
  geom_point(mapping=aes(x=flipper_length_mm,y=body_mass_g,color=species))+  
  facet_grid(sex~species)
```

Commented [NL4]: 'tilde' (~): is used to define the relationship between dependent variable and independent variables in a statistical model formula.

- The variable on the left-hand side of tilde operator is the dependent variable
- The variable(s) on the right-hand side of tilde operator is/are called the independent variable(s)



There are nine separate plots, each based on a combination of the three species of penguin and three categories of sex.

In case that there is too much on the x-axis to read →

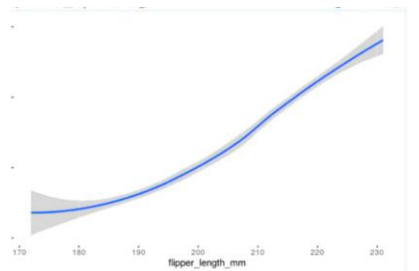
```
ggplot(data = hotel_bookings) +  
  geom_bar(mapping = aes(x = distribution_channel)) +  
  facet_wrap(~deposit_type) +  
  theme(axis.text.x = element_text(angle = 45))
```

Use different geom functions to create different types of plots

geom_smooth(): general trends in the data

Make the data readable

2 types of smoothing:



Type of smoothing	Description	Example code
Loess smoothing	The loess smoothing process is best for smoothing plots with less than 1000 points.	<pre>ggplot(data, aes(x=, y=)) + geom_point() + geom_smooth(method="loess")</pre>
Gam smoothing	Gam smoothing, or generalized additive model smoothing, is useful for smoothing plots with a large number of points.	<pre>ggplot(data, aes(x=, y=)) + geom_point() + geom_smooth(method="gam", formula = y ~s(x))</pre>

For more information about smoothing, refer to the Smoothing section in the [Stats Education's Introduction to R](#)

geom_jitter() function

Creates a scatter plot and then adds a small amount of random noise to each point in the plot

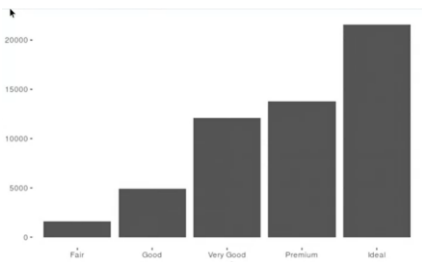
When: over-plotting -> jitter makes it easier to find the points in a scatter plot



geom_bar()

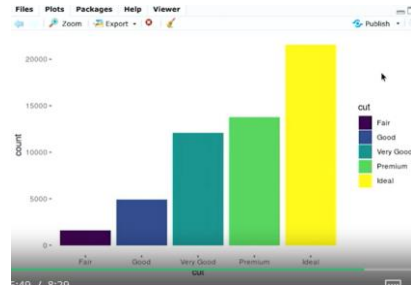
R auto counts how many times each x-value appears in the data, and then shows the counts on the y-axis

```
ggplot(data=diamonds)+  
  geom_bar(mapping=aes(x=cut))
```

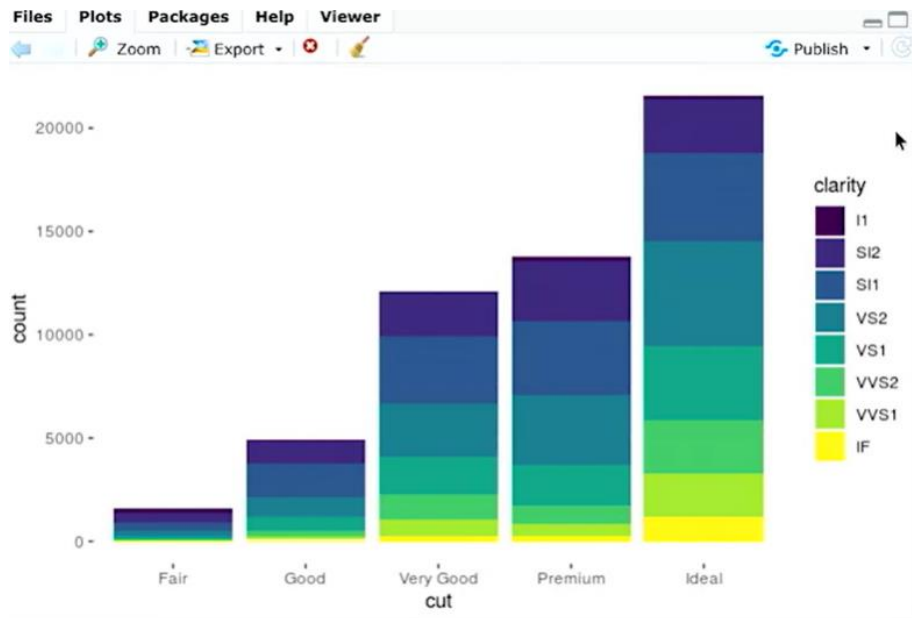


```
ggplot(data=diamonds)+  
  geom_bar(mapping=aes(x=cut,fill=cut))
```

Add fill to the variables

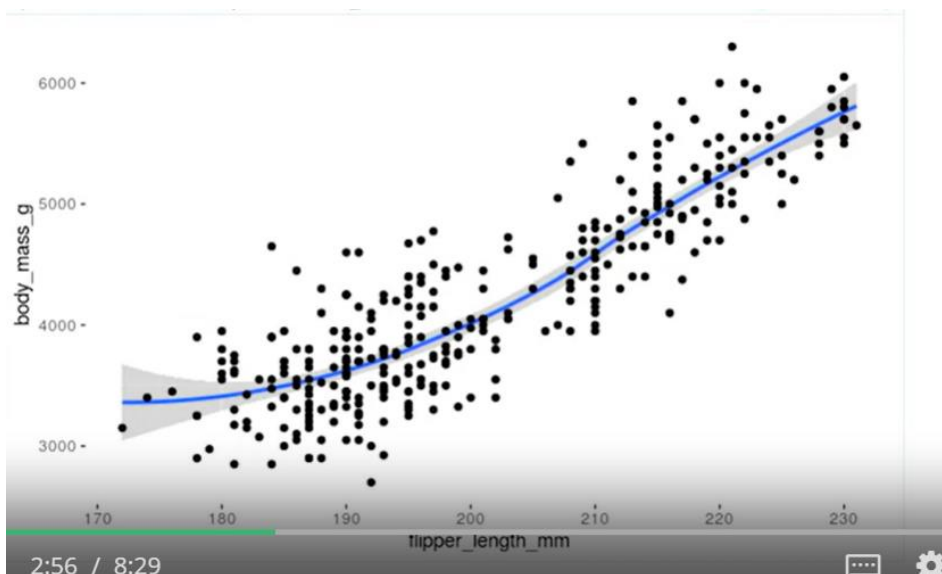


Use fill = [other variables] → display a stacked bar chart



Combine 2 (or more) types of plots:

```
ggplot(data=penguins)+  
  geom_smooth(mapping=aes(x=flipper_length_mm,y=body_mass_g)) +  
  geom_point(mapping=aes(x=flipper_length_mm,y=body_mass_g))
```

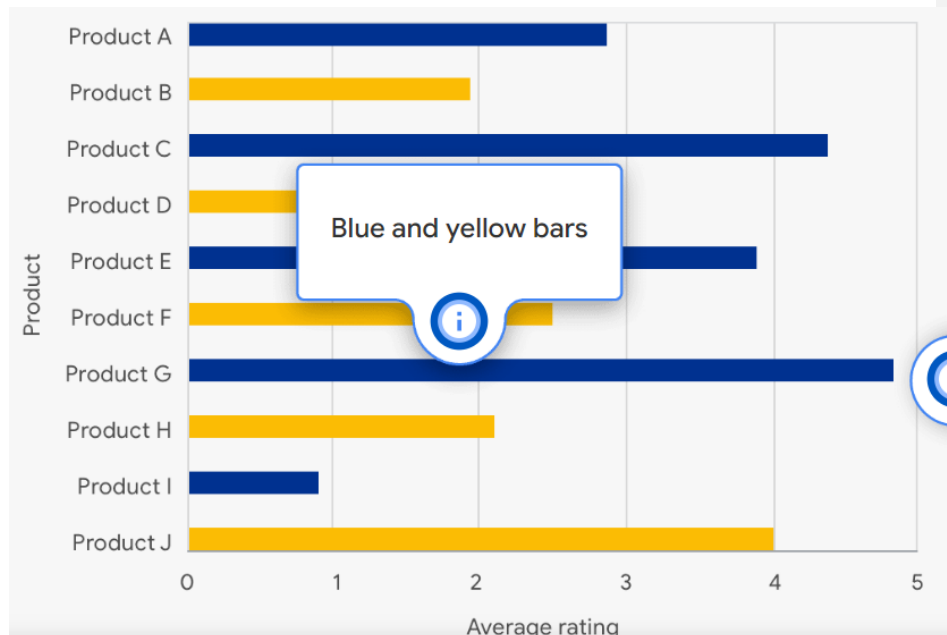


Enhancing visualization in R

In `aes()`: you can add

- `color` = [col_names whose specified categories]
- `shape` = [col_names whose specified categories] (assign the shape into the geom point)
- `size` = [col_names whose specified categories] (make each categories is into different size)
- `alpha` = [col_names whose specified categories] (add transparenttness into each categories)
- `linetype` = [col_names whose specified categories] (each variables get a specific line types illustration)

- `col = ifelse (x<2, 'blue', 'yellow')`. (To highlight underperforming products, use an aesthetics function)



Change the appearance of our overall plot without regard to specific variables

→ write code outside of the aes function. Outside the `aes()`, add `color = "[color_names]"`

```
ggplot(data=penguins)+
  geom_point(mapping=aes(x=flipper_length_mm,y=body_mass_g),color="purple")
```

Add chart title: add `title = [chart_title]`

Additional sources:

- **Data visualization with ggplot2 cheat sheet:** RStudio's cheat sheet is a great reference to use while working with ggplot2. It has tons of helpful information, including explanations of how to use geoms and examples of the different visualizations that you can create.

- **Stats Education's Introduction to R:** This resource is a great way to learn the basics of ggplot2 and how to apply aesthetic attributes to your plots. You can return to this tutorial as you work more with ggplot2 and your own data.
- **RDocumentation aes function:** This guide describes the syntax of the aes function and explains what each argument does.

Filtering data for plotting

Why: To focus on specific subsets of your data and gain more targeted insights

How: include the dplyr filter() function in your ggplot syntax.

Example code

```
data %>%
  filter(variable1 == "DS") %>%
  ggplot(aes(x = weight, y = variable2, colour = variable1)) + geom_point(alpha = 0.3, position = position_jitter()) +
  stat_smooth(method = "lm")
```

Hands on activity

Create a plot that shows the relationship between lead time and guests traveling with children for online bookings at city hotels

```
hotel_bookings %>%
```

```
  filter(hotel == "City Hotel" & hotel_bookings$market_segment == "Online TA") %>%
```

```
  ggplot(aes(x = lead_time, y = children)) + geom_point()
```

Or:

```
hotel_bookings %>%
```

```
  filter(hotel=="City Hotel") %>%
```

```
  filter(market_segment=="Online TA")
```

Additional sources:

- **Putting it all together: (dplyr+ggplot):** The RLadies of Sydney's course on R uses real data to demonstrate R functions. This lesson focuses specifically on combining dplyr and ggplot to filter data before plotting it. The instructional video will guide you through every step in the process while you follow along with the data they have provided.

- **Data transformation:** This resource focuses on how to use the `filter()` function in R, and demonstrates how to combine `filter()` with `ggplot()`. This is a useful resource if you are interested in learning more about how `filter()` can be used before plotting.
- **Visualizing data with ggplot2:** This comprehensive guide includes everything from the most basic uses for `ggplot2` to creating complicated visualizations. It includes the `filter()` function in most of the examples so you can learn how to implement it in R to create data visualizations.

Common problems when visualizing in R

Case sensitivity (viet hoa)

Balancing parentheses () and quotation marks ""

Incorrect use of the plus sign to add layers: The plus sign should always be placed at the end of a line of code, and not at the beginning of a line.

- **Help resources:** ?function_name, google, R community ([R for Data Science Online Learning Community](#), [RStudio Community](#), [Stackoverflow](#), [Twitter \(#rstats\)](#))

Annotation layer

1. Add title

```
ggplot(data=penguins)+
  geom_point(mapping=aes(x=flipper_length_mm,y=body_mass_g,color=species))+
  labs(title="Palmer Penguins: Body Mass vs. Flipper Length")
```

2. Add subtitle (below title) and caption (below graph) inside the parenthesis of labs

`Labs(title = "ABC", subtitle = "XUZ", caption = "xxx")`

3. Add more information

For example, add description about what time period this data covers

```
mindate <- min(hotel_bookings$arrival_date_year)
```

```
maxdate <- max(hotel_bookings$arrival_date_year)
```

draw the plot

```
ggplot(data = hotel_bookings) +
```

```
  geom_bar(mapping = aes(x = market_segment)) +
```

```

facet_wrap(~hotel) +
theme(axis.text.x = element_text(angle = 45)) +
# add information
labs(title="Comparison of market segments by hotel type for hotel bookings",
      subtitle/caption=paste0("Data from: ", mindate, " to ", maxdate)),

```

Change the name of x-axis and y-axis

```

ggplot(data = hotel_bookings) +
  geom_bar(mapping = aes(x = market_segment)) +
  facet_wrap(~hotel) +
  theme(axis.text.x = element_text(angle = 45)) +
  labs(title="Comparison of market segments by hotel type for
hotel bookings",
        caption=paste0("Data from: ", mindate, " to ",
maxdate),
        x="Market Segment",
        y="Number of Bookings")
...

```

Commented [NL5]: if the data gets updated and there is more recent data added, you don't have to change the code below because the variables are dynamic

Annotate (text label):

```

+ annotate("text", x = [# of point of x-axis], y = [# of point of y-axis], label = "[text caption]",
color = "[color_names]", fontface = "[bold/ etc.]", size = [size], angle = [angle degreee] )

```

Commented [NL6]: Optional

```

ggplot(data=penguins)+
  geom_point(mapping=aes(x=flipper_length_mm,y=body_mass_g,color=species)) +
  labs(title="Palmer Penguins: Body Mass vs. Flipper Length", subtitle="Data collected by Dr. Kristen Gorman")+
  annotate("text", x=220,y=3500,label="The Gentoos are the largest")

```

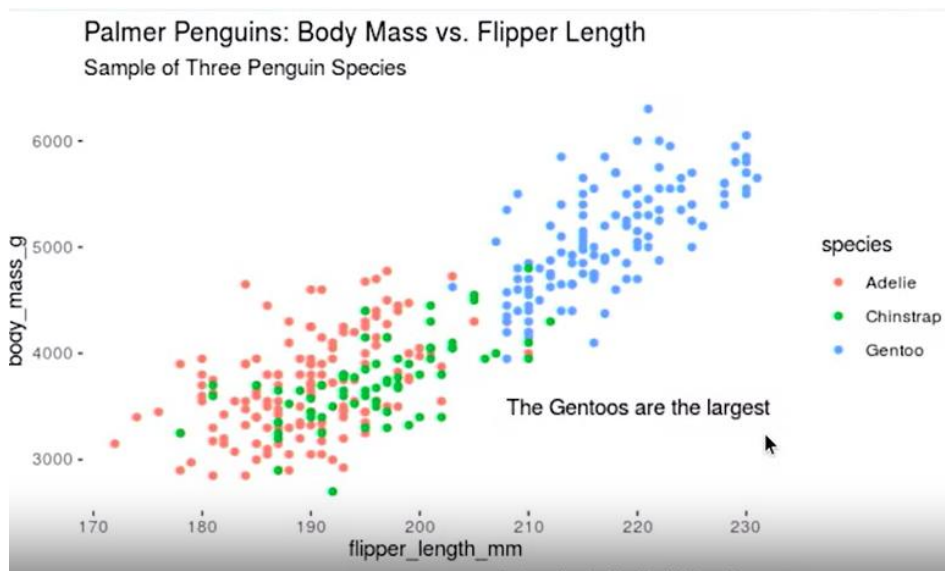
Tips: since the code is too long now, we can assign a variable to the plot

```

p <- ggplot(data=penguins)+
  geom_point(mapping=aes(x=flipper_length_mm,y=body_mass_g,color=species)) +
  labs(title="Palmer Penguins: Body Mass vs. Flipper Length", subtitle="Data collected by Dr. Kristen Gorman")

p+annotate("text",x=220,y=3500,label="The Gentoos are the largest")

```



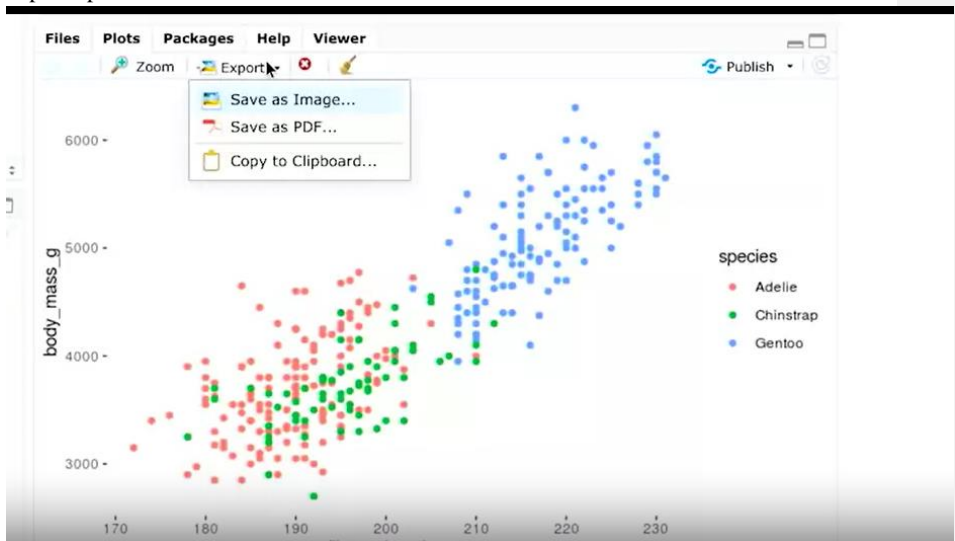
You can also add lines, arrows, and shapes to your plots using **ggplot2**.

- **Create an annotation layer:** This guide explains how to add an annotation layer with ggplot2. It includes sample code and data visualizations with annotations created in ggplot2.
- **How to annotate a plot in ggplot2:** This resource includes explanations about how to add different kinds of annotations to your ggplot2 plots, and is a great reference if you need to quickly look up a specific kind of annotation.
- **Annotations:** Chapter eight of the online ggplot2 textbook is focused entirely on annotations. It provides in-depth explanations of the different types of annotations, how they are used, and detailed examples.
- **How to annotate a plot:** This R-Bloggers article includes explanations about how to annotate plots in ggplot2. It starts with basic concepts and covers more complicated information the further on you read.
- **Text Annotations:** This resource focuses specifically on adding text annotations and labels to ggplot2 visualizations.

Saving your viz

Use:

- export option



The picture would be in the Files tab.

- ggsave() function

`ggsave("[file name].[type of files name you want to save]", width = [width number], height = [height of pic])` -> auto save the latest plot

eg. `Ggsave("Three Penguins Species.png")`

- Not use ggsave

Use when: it might be best to save your plot by writing it directly to a graphics device
 -> Save your plot by `png()` or `pdf()`, or print the plot and then close the device using `dev.off()`.

Commented [NL7]: A graphics device allows a plot to appear on your computer. Examples include:

- A window on your computer (screen device)
- A PDF, PNG, or JPEG file (file device)
- An SVG, or scalable vector graphics file (file device)

Example of using <code>png()</code>	Example of using <code>pdf()</code>
<pre>png(file = "exampleplot.png", bg = "transparent") plot(1:10) rect(1, 5, 3, 7, col = "white") dev.off()</pre>	<pre>pdf(file = "/Users/username/Desktop/example.pdf", width = 4, height = 4) plot(x = 1:10, y = 1:10) abline(v = 0) text(x = 0, y = 1, labels = "Random text") dev.off()</pre>

To learn more about the different processes for saving images, check out these resources:

- **Saving images without ggsave():** This resource is pulled directly from the ggplot2 documentation at tidyverse.org. It explores the tools you can use to save images in R, and includes several examples to follow along with and learn how to save images in your own R workspace.
- **How to save a ggplot:** This resource covers multiple different methods for saving ggplots. It also includes copyable code with explanations about how each function is being used so that you can better understand each step in the process.
- **Saving a plot in R:** This guide covers multiple file formats that you can use to save your plots in R. Each section includes an example with an actual plot that you can copy and use for practice in your own R workspace.

Module 5: Documentation and Reports in Rstudio

R Markdown

A file format for making dynamic documents with R

You can use an R Markdown file as a code notebook (like Jupyter notebook) to save, organize, and document your analysis using code chunks, comments, and other features. When you finish your data cleaning and exploration, you can create a report in R Markdown to summarize your findings for stakeholders

Markdown: syntax for formatting plain text files

Resources

RStudio's [R Markdown documentation](#): a series of tutorials that will help you learn about the main features of R Markdown

The [R Markdown Reference Guide](#)

The [R Markdown Cheat Sheet](#)

If you want to really explore the capabilities of R Markdown in a systematic way

→ [R Markdown: The Definitive Guide](#)

Alternatives of Markdown: Jupyter notebook

Jupyter notebooks include basic formatting tools and rules that will help you keep your work organized and user-friendly. In fact, Jupyter uses R Markdown as its language for writing and formatting text in a notebook.

To learn about basic formatting in Jupyter notebooks, check out these resources:

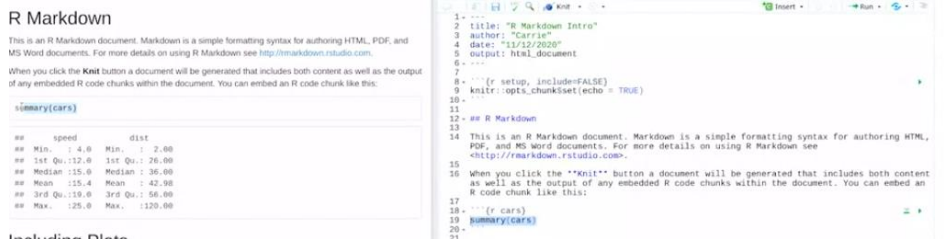
- **The Jupyter Notebook:** This resource provides an overview of Jupyter notebooks, including information about the structure of the user interface and notebook document. You'll also learn about the basic workflow for using a notebook document, along with information about keyboard shortcuts and other features that will help you format your work.
- **Using Jupyter Notebook for Writing:** This resource focuses on how to use Markdown to format your writing in a Jupyter notebook. Use this as a guide to manage the syntax of your writing, including making titles and subtitles and adding links.
- **The Jupyter Notebook Formatting Guide:** This resource includes a wide variety of formatting options for Jupyter notebooks. You'll learn about the basics as well as some more advanced options, like embedding PDF documents and videos.

Install R Markdown

Install.packages("rmarkdown")

Go to Files → Save a new R Markdown document

After editing your R Markdown files → Knit button → go to your HTML Report. At this report, the code chunk has all been run and now we have their output



The screenshot shows the R Markdown editor interface. On the left, a preview of the HTML report is visible, showing the output of the `summary(cars)` function. On the right, the source R Markdown file is shown with a code chunk containing `summary(cars)`. The code chunk is highlighted in blue.

R Markdown

This is an R Markdown document. Markdown is a simple formatting syntax for authoring HTML, PDF, and MS Word documents. For more details on using R Markdown see <http://rmarkdown.rstudio.com>.

When you click the **Knit** button a document will be generated that includes both content as well as the output of any embedded R code chunks within the document. You can embed an R code chunk like this:

```
>summary(cars)
```

	speed	dist
## Min.:	4.0	Min.: 2.00
## 1st Qu.:	12.0	1st Qu.: 26.00
## Median:	15.0	Median: 36.00
## Mean:	15.4	Mean: 42.98
## 3rd Qu.:	19.0	3rd Qu.: 56.00
## Max.:	25.0	Max.: 120.00

including Date

```
1. ---
2. title: "R Markdown Intro"
3. author: "Carrie"
4. date: "11/12/2020"
5. output: HTML_document
6. ---
7.
8. [r setup, include=FALSE]
9. knitr::opts_chunk$set(echo = TRUE)
10.
11.
12. ## R Markdown
13.
14. This is an R Markdown document. Markdown is a simple formatting syntax for authoring HTML,
15. PDF, and MS Word documents. For more details on using R Markdown see
16. http://rmarkdown.rstudio.com.
17.
18. When you click the "Knit" button a document will be generated that includes both content
19. as well as the output of any embedded R code chunks within the document. You can embed an
20. R code chunk like this:
21.
22. [r cars]
23. summary(cars)
24.
```

Create R Markdown documents

YAML header section.

You can change the information in this section at any time by adding text or by typing over the current text

```
1 ---
2 title: "Penguins Plots"
3 author: "DA Cert"
4 date: "2/26/2021"
5 output: html_document
6 ---
```

Code chunk.

This chunk basically means that your code will be shown in your final report when you're ready to render it.

To add a code chunk in your file

1. To start a code chunk:

```
``{r}
```

Commented [NL8]: This is delimiter

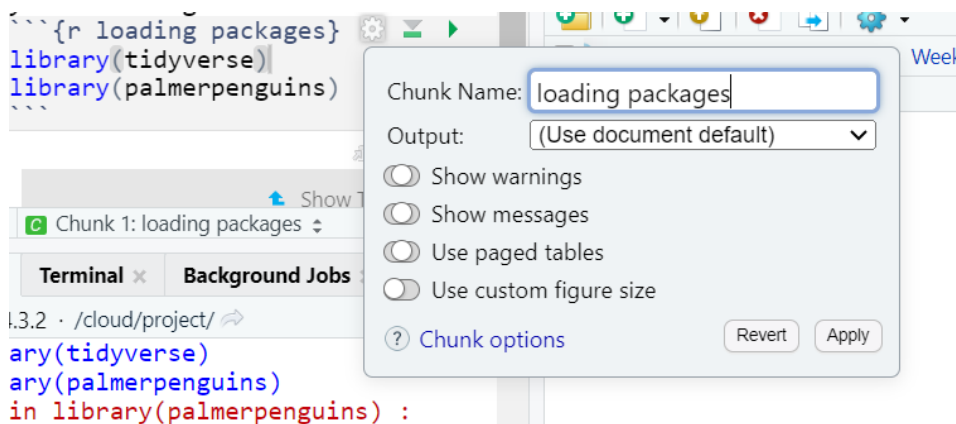
To end it, type just the three tick marks: ``

1.1. Add header to the code chunk: by adding the header next to r, you can find the code chunk section easily in the menu below

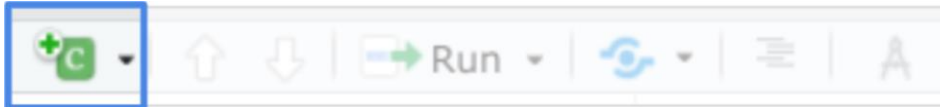
Eg.

```
``{r loading packages}
```

1.2. Change the setting of code chunk:



Short cut to adding code: press **Ctrl + Alt + I** (PC) or **Cmd + Option + I** (Mac). Or you can click the **Add Chunk** command in the editor toolbar:



2. Press Enter (Windows) or Return (Mac) two or three times after the default code chunk to create space between the existing code chunk and the next code chunk you will add.
3. Copy the code from the analysis file you opened earlier and paste it in the gray area between the beginning and ending delimiters.
4. Select the rest of the template content in the file and delete it. This gives you a blank space to work in to help avoid potential errors from mixing your own comments and code with the pre-existing ones in the template.

The white background is where you will type plain text with markdown syntax. As you learned earlier in this course, markdown is a syntax for formatting plain text files. Using markdown makes it easier to write and format text in your notebook.

```
1 ---
2 title: "Penguin Plots"
3 author: "DA Cert"
4 date: "3/1/2021"
5 output: html_document
6 ---
7
8 ```{r setup, include=FALSE}
9 knitr::opts_chunk$set(echo = TRUE)
10 ```
11
12 ## R Markdown
13
14 This is an R Markdown document. Markdown is a simple formatting syntax for authoring
15 HTML, PDF, and MS Word documents. For more details on using R Markdown see
16 <http://rmarkdown.rstudio.com>.
17
18 When you click the Knit button a document will be generated that includes both
19 content as well as the output of any embedded R code chunks within the document. You can
20 embed an R code chunk like this:
21
22 ```{r cars}
23
24 ```
```

Some basic formatting options:

- To start a new paragraph, end a line with two spaces
- To apply italics to a word or phrase, place an asterisk at the beginning and at the end of the word or phrase, for example, **italics works** / *_italics works_*
- To apply bold to a word or phrase, place two asterisks at the beginning and at the end of the word or phrase, for example, ****bold is useful****
- To create a header, type a hashtag (#) followed by a space and your text for example: # Getting Started with R Markdown

- Create a bullets: add * before the statements bullet
- Include link:
 - `<link>` appears as text link
 - Embedding link to text: click here `[link](text link)`
 `click here[link](http://rmarkdown.rstudio.com)`.
 click here[link](http://rmarkdown.rstudio.com).
 →
- Embedd images

`! [Plot this way](https://cdn.pixabay.com/photo/2013/07/12/13/52/arrow-147464__340.png)`



You can also embed plots, for example:



Plot this way

Header

When creating headers keep the following in mind:

- Headers will appear in blue
- A single hashtag is the largest header
- The more hashtags you add (up to six), the smaller the header

#Heading 1

Heading 2

To format comments in your notebook, follow these steps:

1. Click in a line above the code chunk you added but below the YAML section.
2. Type a main header for your report using a single hashtag. You might want to restate the title in the YAML in a different way or add to it with a short description.
3. Add a smaller header below that to label the first part of your programming. Follow that with a description of the code chunk that you added.

```
8 - # Penguin Plots: Practice analysis
9
10 - ## Setting up my environment
11 Notes: setting up my R environment by loading the `tidyverse` and
    `palmerpenguins` packages:
12 - ```{r }
13   library(tidyverse)
14   library(palmerpenguins)
15 - ```
```

Tick marks format the text to appear as code even though the text is not in a code chunk. The tick marks in the code above create a gray background behind “tidyverse” and “palmerpenguins.”

Continue formatting

To check how your file is going, click “Knit”

The structure and components of the document

YAML

Heading

Inline codes

Code chunks, including code for table of plots

Note that the ``echo = FALSE`` parameter was added to the code chunk to prevent printing of the R code that generated the plot.

Process of exporting your documentation

Use knit to export rmd file. Stick to export to HTML as HTML doesn't have page breaks, so you can focus on generating content for your report and not its appearance.

Available document outputs

In addition to the default HTML output (**html_document**), you can create other types of documents in R Markdown using the following output settings:

- **pdf_document** – This creates a PDF file with LaTeX (an open source document layout system). If you don't already have LaTeX, RStudio will automatically prompt you to install it.
- **word_document** – This creates a Microsoft Word document (.docx).
- **odt_document** – This creates an OpenDocument Text document (.odt).
- **rtf_document** – This creates a Rich Text Format document (.rtf).
- **md_document** – This creates a Markdown document (which strictly conforms to the original Markdown specification)
- **github_document** – This creates a GitHub document which is a customized version of a Markdown document designed for sharing on GitHub.

Notebooks

A **notebook** (html_notebook) is a variation on an HTML document (html_document). Overall, the output formats are similar; the main difference between them is that the rendered output of a notebook always includes an embedded copy of the source code.

HTML documents are good for communicating with stakeholders. Notebooks are better for collaborating with other data analysts or data scientists.

Presentations

R Markdown renders files to specific presentation formats when you use the following output settings:

- **beamer_presentation** – for PDF presentations with beamer
- **ioslides_presentation** – for HTML presentations with ioslides
- **slidy_presentation** – for HTML presentations with Slidy
- **powerpoint_presentation** – for PowerPoint presentations
- **revealjs :: revealjs_presentation** – for HTML presentations with reveal.js (a framework for creating HTML presentations that requires the reveal.js package)

To learn more, check out the section on [Slide Presentations](#)

Dashboards

Dashboards are a useful way to quickly communicate a lot of information. Tea [flexdashboard](#) package lets you publish a group of related data visualizations as a dashboard. Flexdashboard also provides tools for creating sidebars, tabsets, value boxes, and gauges.

To learn more, visit the [flexdashboard for R](#) page and the [Dashboards](#) section in the R Markdown documentation.

Shiny

Shiny is an R package that lets you build interactive web apps using R code. You can embed your apps in R Markdown documents or host them on a webpage.

To call Shiny code from an R Markdown document, add runtime: shiny to the YAML header:

```
---  
  
title: "Shiny Web App"  
  
output: html_document  
  
runtime: shiny  
  
---
```

To learn more about Shiny and how to use R code to add interactive components to an R Markdown document, check out the [Shiny](#) tutorial from RStudio.

Other packages provide even more output formats:

- Tea [bookdown](#) package is helpful for writing books and long-form articles.
- Tea [prettydoc](#) package provides a range of attractive themes for R Markdown documents.
- Tea [articles](#) package provides templates for various journals and publishers.

Visit the [RStudio Formats](#) page in the R Markdown documentation for a more comprehensive list of output formats and packages.

Additional resources

For more information, check out these additional resources:

- Tea [R Markdown gallery](#) from RStudio has tons of examples of the outputs you can create with R Markdown.

- Tea [R Markdown Formats](#) chapter in the *R for Data Science* book provides more details about the output formats introduced in this reading. This reading was compiled from information in this book.

Create a template

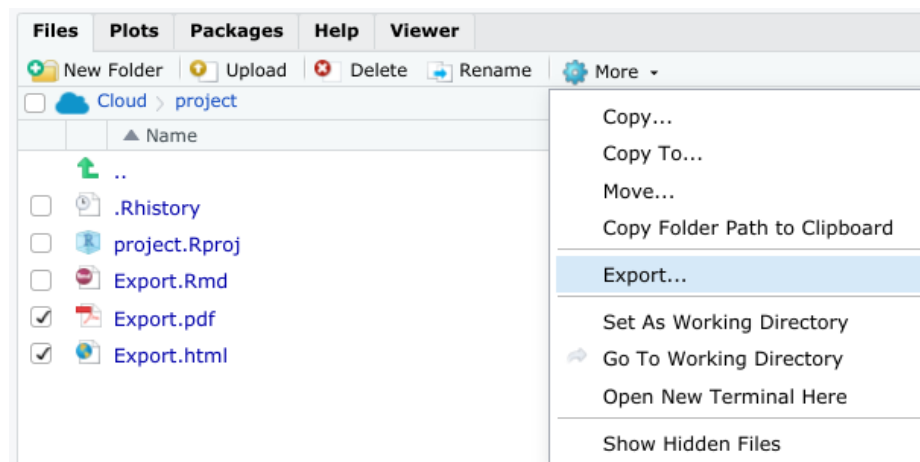
When: need to create a certain type of document over and over or if you want to customize the appearance of your final report (eg. Monthly or annual report, etc.)

Some popular packages with templates for R Markdown include the following:

- The **vita** package contains templates for creating and maintaining a résumé or curriculum vitae (CV)
- The **rticles** package provides templates for various journals and publishers
- The **learnr** package makes it easy to turn any R Markdown document into an interactive tutorial
- The **bookdown** package facilitates writing books and long-form articles
- The **flexdashboard** package lets you publish a group of related data visualizations as a dashboard

Download the files you exported

1. find them in the Files explorer in the bottom-right of the screen.
2. Check the boxes of the file(s) you want to download and click the **More** dropdown.



3. Click **Export...** and name the file by a title that will help you find it later. Click **Download**. Now your exported file will be downloaded to your computer.

Export Files

The selected file(s) will be downloaded to your computer. Please specify a name for the downloaded file:

Download

Cancel