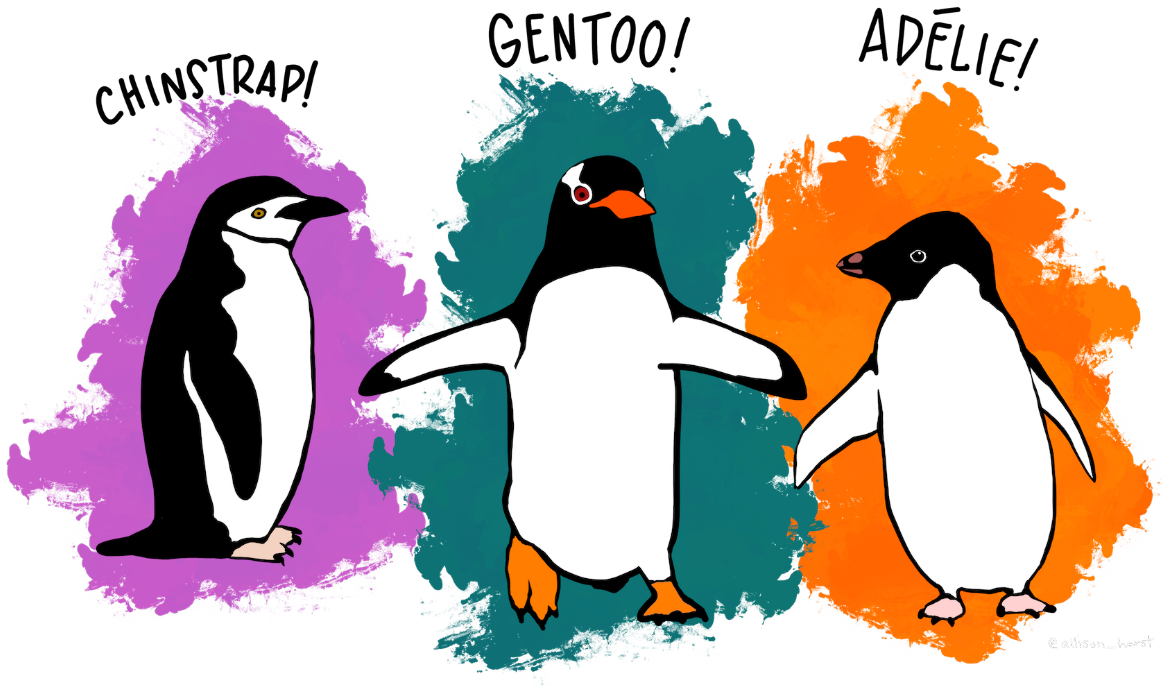
# R Programming: Data Analysis & Visualization

Stacey Borrego (edited by Kristina Riemer)

## Helpful Links

- Data Carpentry: Data Analysis and Visualization in R for Ecologists
- Tidyverse: R packages for Data Science
- R for Data Science 2nd Edition by Hadley Wickham and Garret Grolemund
- Advanced R by Hadley Wickham
- ggplot2: Elegant graphics for data analysis by Hadley Wickham
- Posit Cheatsheets by Posit
- The R Gallery by Kyle W. Brown
- Introduction to R by Douglas, Roos, Mancini, Cuoto, & Lusseau



Artwork by Allison Horst

## Starting with Data

This is what the data will look like:

| Column | Description |
|---|---|
| species | Name of species |
| island | Island where species was recorded |
| bill_length_mm | Measured length of individual penguin's bill |
| bill_depth_mm | Measured depth of individual penguin's bill |
| flipper_length_mm | Measured length of individual penguin's flipper |
| body_mass_g | Measured mass of individual penguin |
| sex | Penguin's sex |
| year | Year that penguin was measured |

```r
# Install R package
install.packages("palmerpenguins")
```

```r
# Read in palmerpenguins
library(palmerpenguins)

# To see data sets available
data(package = "palmerpenguins")

# Load the data set of choice
# use the data set name as the argument to `data()`
data(penguins)
```

Open the dataset in RStudio's Data Viewer

```r
# Check the data type of the data just loaded
class(penguins)

# View the whole data set
View(penguins)
```

Inspect the data

```r
# See a few rows of the data
head(penguins)
tail(penguins)

# See a specific number of rows of the data
head(penguins, n = 10)
tail(penguins, n = 10)
```

- Size:

    - `dim(penguins)` - returns a vector with the number of rows in the first element, and the number of columns as the second element (the dimensions of the object)
    - `nrow(penguins)` - returns the number of rows
    - `ncol(penguins)` - returns the number of columns

- Content:
    - `head(penguins)` - shows the first 6 rows
    - `tail(penguins)` - shows the last 6 rows

- Names:
    - `names(penguins)` - returns the column names (synonym of colnames() for data.frame objects)
    - `rownames(penguins)` - returns the row names

- Summary:
    - `str(penguins)` - structure of the object and information about the class, length and content of each column
    - `summary(penguins)` - summary statistics for each column

**Exercises**

1. How many rows and columns are there in this dataframe?
2. What is the variable name of the third to last column
3. How many islands were the penguins measured on, and what are their names?
4. Which columns have missing data?

```
nrow(penguins)
ncol(penguins)
colnames(penguins)
str(penguins) # see island column
summary(penguins) # bill length, bill depth, flipper length, body mass, sex
```

### Indexing and subsetting data frames

A data frame has rows and columns in 2 dimensions. To extract specific data, specify the "coordinates" in `[ ]` indicating row numbers first followed by column numbers.

```
# dim() gives the output of the rows and columns of the object
dim(penguins)

# Extract specific values by specifying row and column indices
# in the format:
# data_frame[row_index, column_index]
```

```r
# Extract the first row and first column from penguins
penguins[1, 1]

# Extract the first row and sixth column
penguins[1, 6]

# To select all columns, leave the column index blank
# Extract the first row and all columns
penguins[1, ]

# Extract all rows and the first column
penguins[, 1]

# An even shorter way to select first column across all rows:
penguins[1] # No comma!

# Select multiple rows or columns with vectors
# Extract the first three rows of the 1st and 2nd columns
penguins[c(1, 2, 3), c(1, 2)]

# We can use the : operator to create a range and select all listed vectors
penguins[1:3, 1:2]

# This is equivalent to head(penguins)
penguins[1:6, ]

# Subsetting with single square brackets ("[]") always returns a data frame.
# If you want a vector, use double square brackets ("[[]]")

# For instance, to get a column as a vector:
penguins[[1]]
penguins[["species"]]

# To get the first value in our data frame:
penguins[[1, 1]]
penguins[1, "species"]

# Data frames can be subset by calling their column names directly
# Use the $ operator with column names to return a vector
penguins$species
```

```r
# Exceptions: subsetting the whole data frame, except the first
# and second columns
```

```r
penguins[, -c(1, 2)]

# Exceptions: subsetting all columns of the data frame
# except for the first six rows
penguins[-(7:nrow(penguins)), ]
```

## Plotting

Helpful links:

- Installation
- ggplot2: Elegant Graphics for Data Analysis

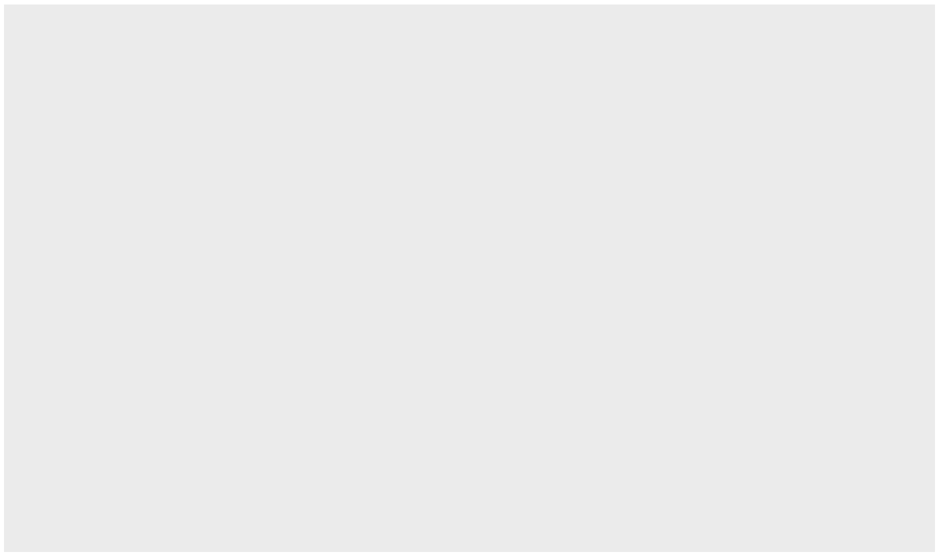
Plotting can be done in many different ways in R. Base R has several basic plotting functions which can be valuable for a quick peek at your data. However, a common and customizable option is to use the package `ggplot2`.

Every ggplot2 plot has three key components:

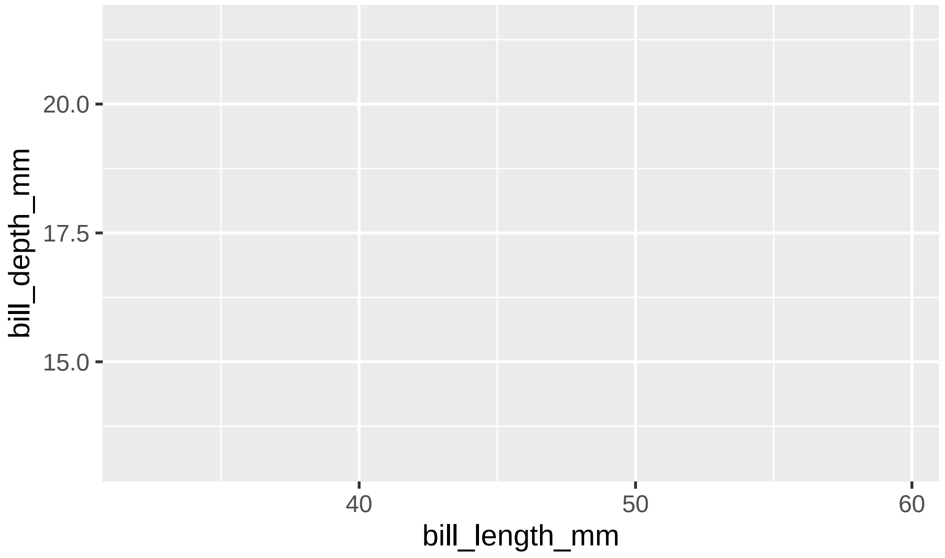
- **Data** (`data`)
- A set of **aesthetic mappings** between variables in the data and visual properties (`aes(x, y)`)
- At least one **layer** which describes how to render each observation. Layers are usually created with a geom function. It is important to note that layers are added with a `+`.

Demonstrate this one step at a time

```r
# Install the following library
# install.packages("ggplot2")

# Load the ggplot2 package to access its unique functions
library(ggplot2)

# Step 0: Specify dataset to plot with `ggplot` function
ggplot(data = penguins)
```

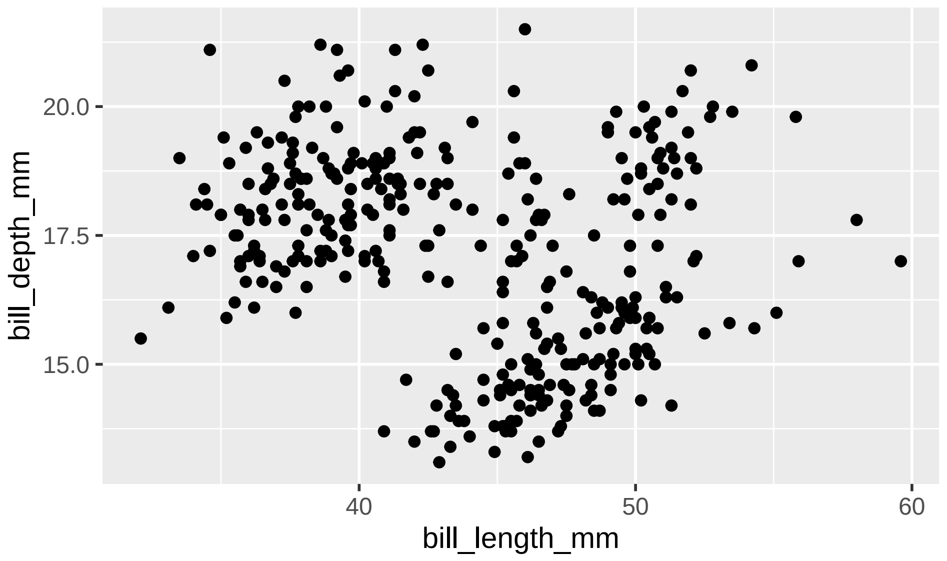


```r
# Step 1: Specify variables for each axis
ggplot(data = penguins, aes(x = bill_length_mm, y = bill_depth_mm))
```



```r
# Step 2: Add geom to specify type of plot with `+`
ggplot(data = penguins, aes(x = bill_length_mm, y = bill_depth_mm)) +
  geom_point()
```

```
## Warning: Removed 2 rows containing missing values or values outside the scale range
## (‘geom_point()‘).
```



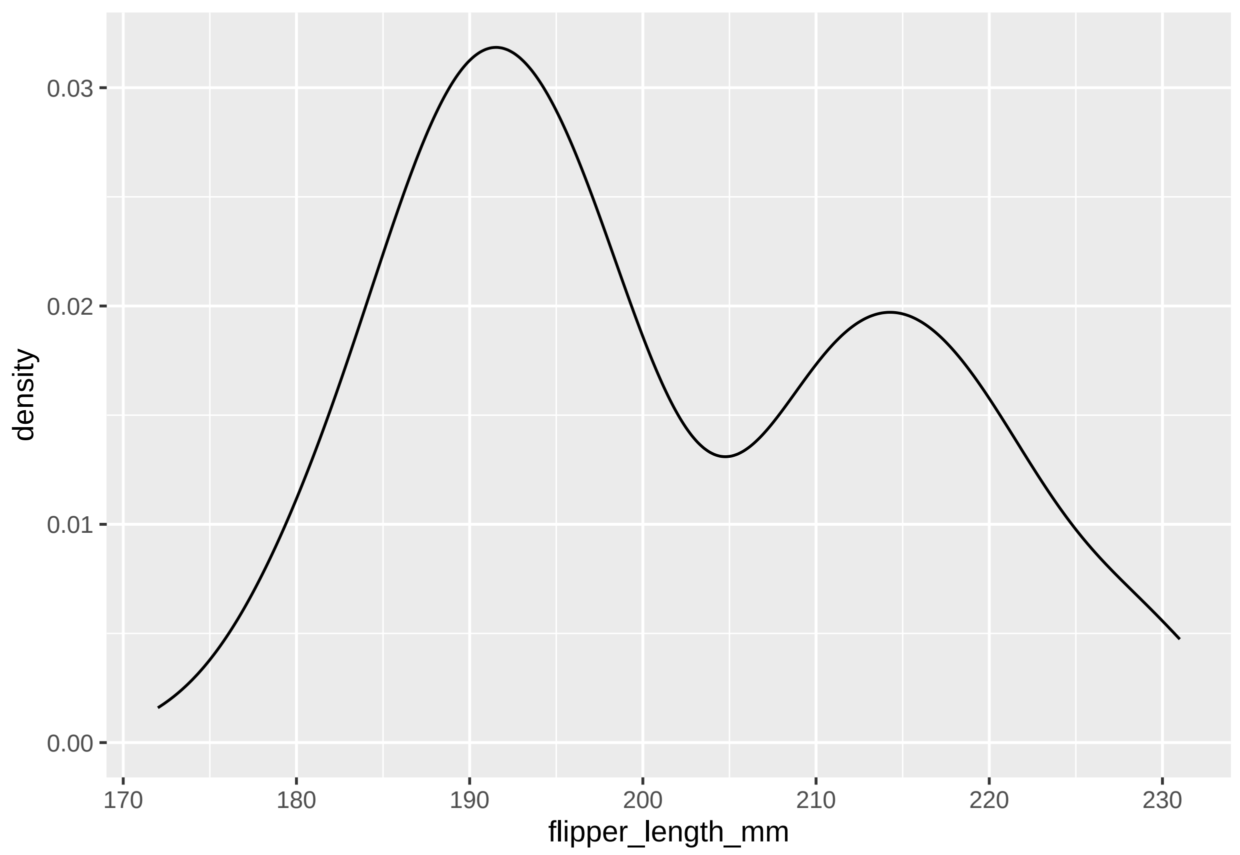
```
# Check out help page for `ggplot()`
?ggplot
```

**Exercises**

1. Create a distribution of flipper length using `geom_density()`

```
ggplot(data = penguins, aes(x = flipper_length_mm)) +
  geom_density()
```

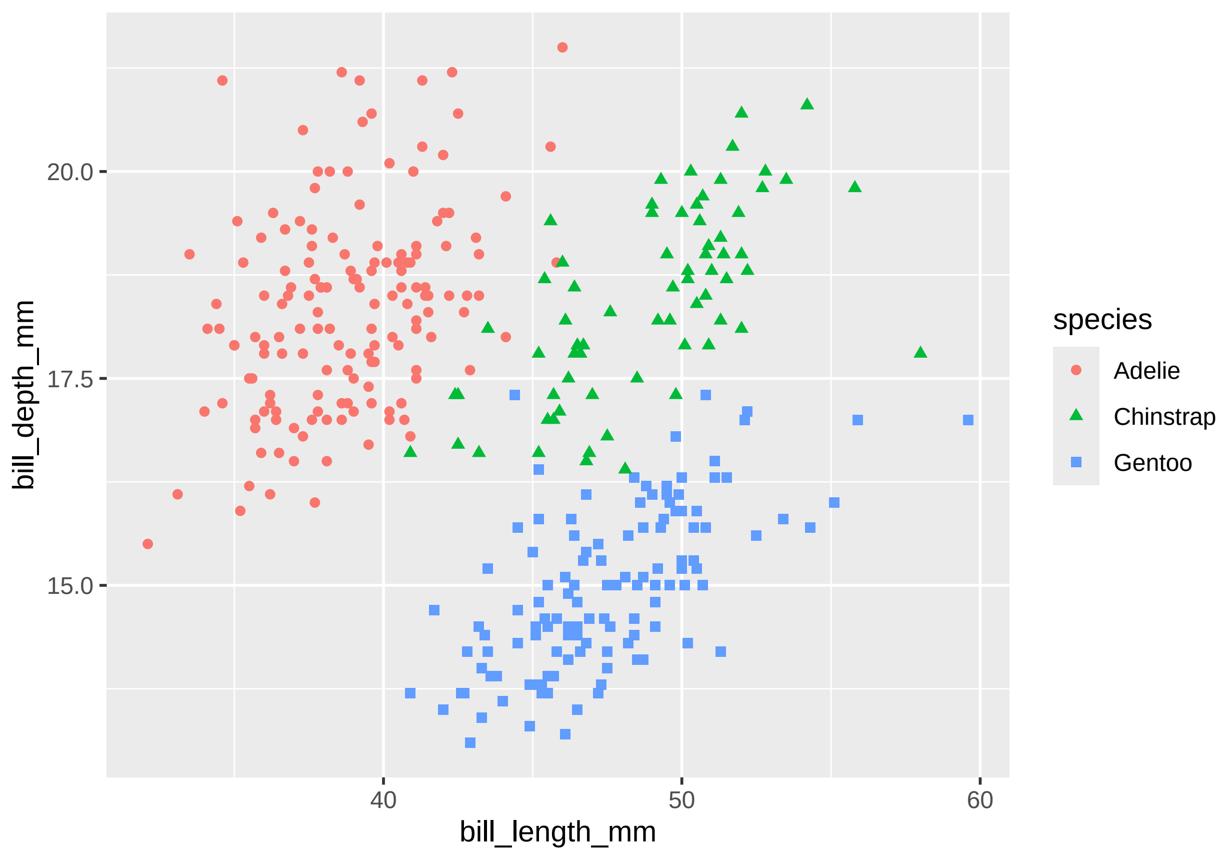
```
## Warning: Removed 2 rows containing non-finite outside the scale range
## (‘stat_density()‘).
```



Modifying plots

```
# Plot points and colors by categorical variable
# Legend shows up automatically
ggplot(data = penguins, aes(x = bill_length_mm, y = bill_depth_mm)) +
  geom_point(aes(color = species, shape = species))
```

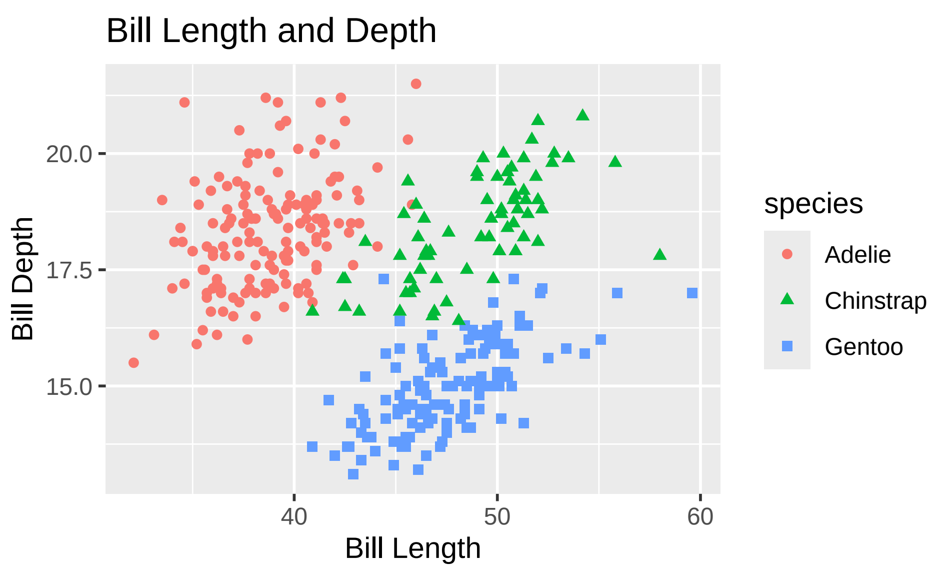
```
## Warning: Removed 2 rows containing missing values or values outside the scale range
## (‘geom_point()‘).
```



```r
# How to know what can be modified? See Aesthetics section
?geom_point
```

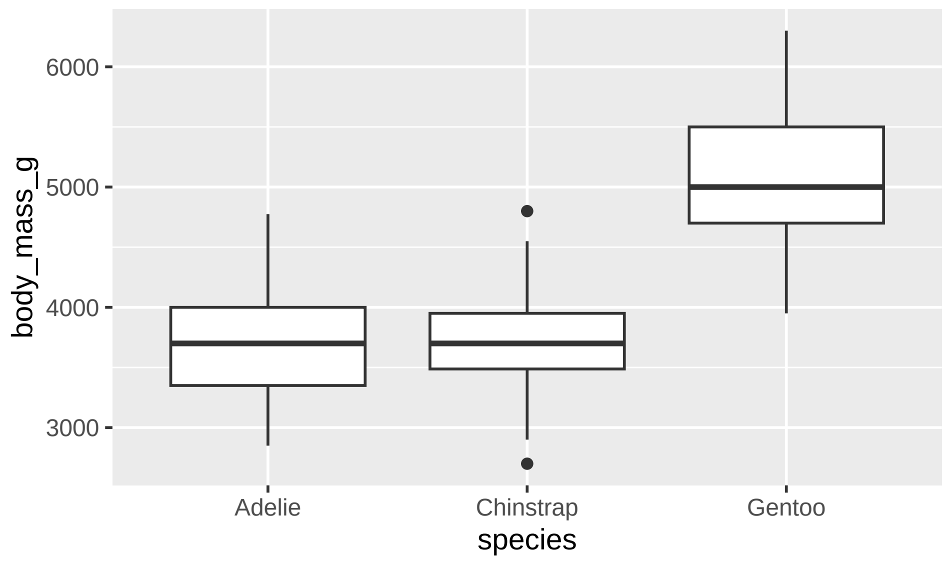
```r
# Add labels using `labs()` and `+`
ggplot(data = penguins, aes(x = bill_length_mm, y = bill_depth_mm)) +
  geom_point(aes(color = species, shape = species)) +
  labs(x = "Bill Length", y = "Bill Depth", title = "Bill Length and Depth")
```

```
## Warning: Removed 2 rows containing missing values or values outside the scale range
## (‘geom_point()‘).
```



```r
# Boxplot
ggplot(data = penguins, aes(x = species, y = body_mass_g)) +
  geom_boxplot()
```

```
## Warning: Removed 2 rows containing non-finite outside the scale range
## (‘stat_boxplot()‘).
```



**Exercises** Modify the box plot

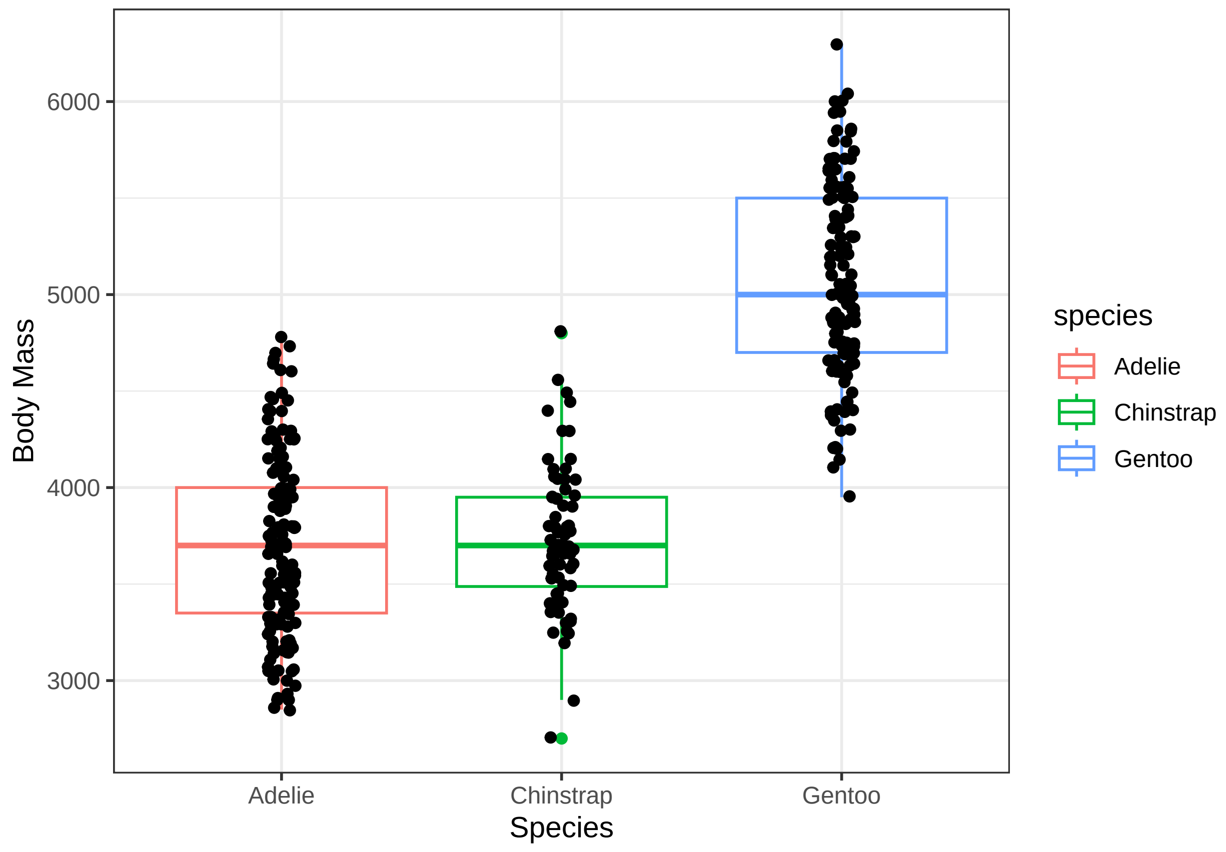
1. Color the boxplot by species
2. Change x and y axis labels to look nicer

3. Simplify the theme using `theme_bw` (and check out the help page for more options)
4. Bonus: Plot the points and spread them out with `geom_jitter`

```r
ggplot(data = penguins, aes(x = species, y = body_mass_g)) +
  geom_boxplot(aes(color = species)) +
  labs(x = "Species", y = "Body Mass") +
  theme_bw() +
  geom_jitter(position = position_jitter(width = 0.05))
```

```
## Warning: Removed 2 rows containing non-finite outside the scale range
## (‘stat_boxplot()‘).
```

```
## Warning: Removed 2 rows containing missing values or values outside the scale range
## (‘geom_point()‘).
```



R comes with default colors and so does ggplot2. These are nice enough but sometimes it is nice to customize our plots. There are few ways we can do this.

- RColorBrewer: Such a helpful tool - contains color palettes and can make a range of colors of your choice. Must be installed and loaded prior to use:

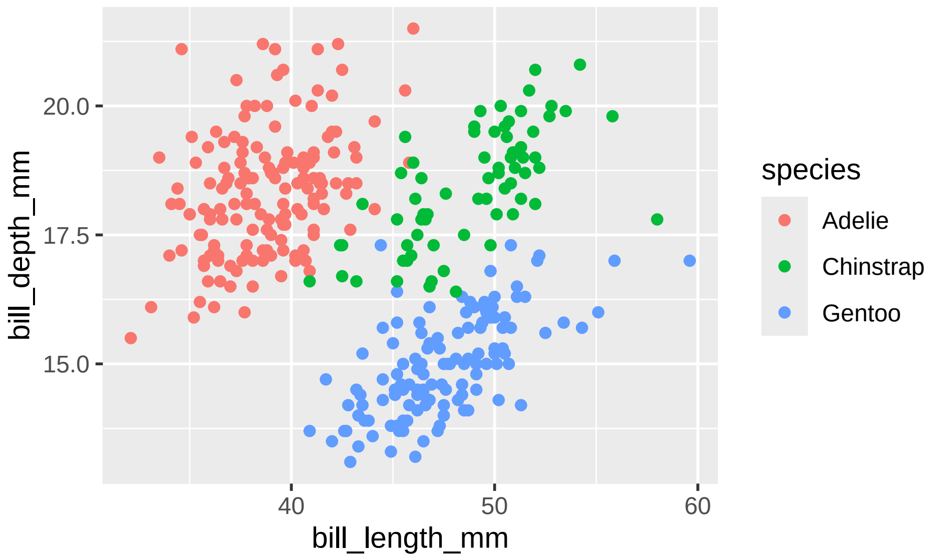
  - install.packages("Rcolorbrewer")
  - library(Rcolorbrewer)

- viridisLite: This is part of ggplot2 that provides color palettes. This does not need to be installed and loaded as it comes with ggplot2.

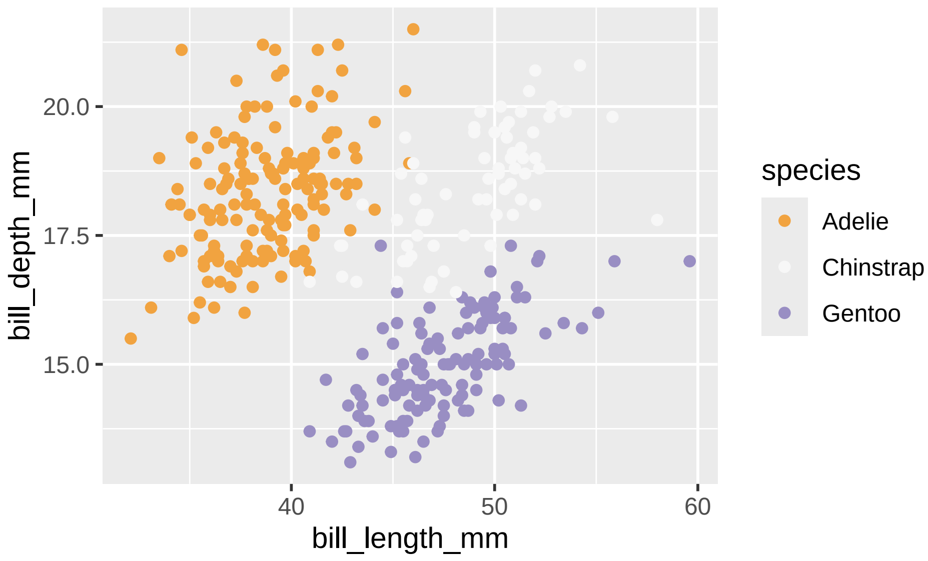
  - Another resource

- Colors in R: A list of all the color names available in R
- You can also provide hexidecimal values. HERE is a helpful tool.

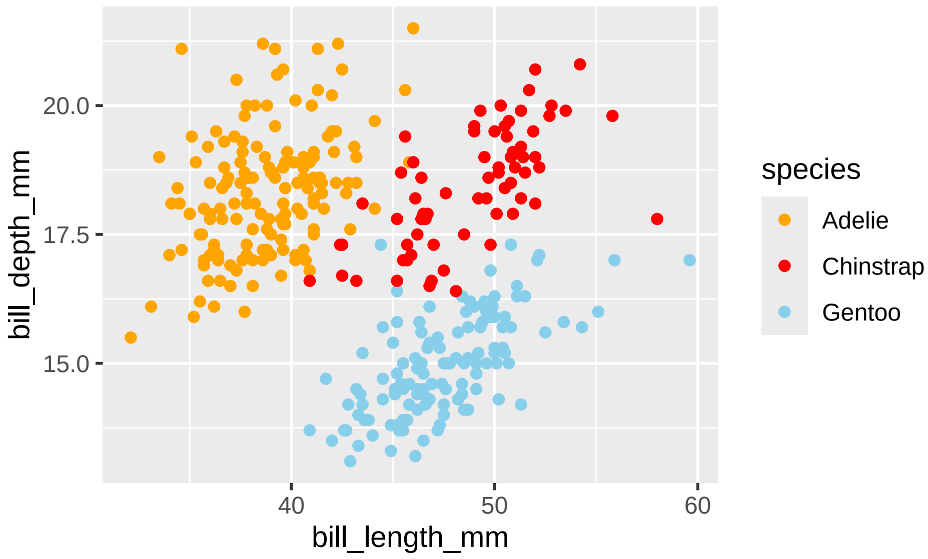
```
# Creating the base plot. Here we have indicated the data set and its
# mappings.We have also added which column should be used for color
# and fill mappings.
ggplot(data = penguins, aes(x = bill_length_mm, y = bill_depth_mm, color = species, fill
  geom_point()
```



```
# RColorBrewer Examples
# install.packages("RColorBrewer")
library(RColorBrewer)

# RColorBrewer Example 1
ggplot(data = penguins, aes(x = bill_length_mm, y = bill_depth_mm, color = species, fill
  geom_point() +
  scale_color_brewer(palette = "PuOr")
```



```r
# Using Your Own Colors, Example 2
ggplot(data = penguins, aes(x = bill_length_mm, y = bill_depth_mm, color = species, fill
  geom_point() +
  scale_color_manual(values = c("orange", "red", "skyblue"))
```



## Factors

This can be the cause of many issues when plotting data. Always check if the data is factored and determined whether is should or should not be to get the correct plot.

R has a special class for working with categorical data, called factors. Once created, factors can only contain a pre-defined set of values, known as levels. Factors are stored as integers associated with labels and they can be ordered or unordered. While factors look (and often behave) like character vectors, they are actually treated as integer vectors by R. So you need to be very careful when treating them as strings.

By default, R always sorts levels in alphabetical order. In the example below, R will assign 1 to the level "experimental" and 2 to the level "wild type" (because "e" comes before "w", even though the first element in this vector is "wild type").

```
sample <- factor(c("wild type", "experimental", "wild type", "experimental"))

# Different ways to see the levels of factored data
sample
levels(sample)
nlevels(sample)
```

### Reorder factors

```
sample <- factor(sample,
                 levels = c("wild type", "experimental"))

sample
levels(sample)
nlevels(sample)
```

### Converting Factors

When working with data in a data frame, the columns that contain text are not automatically coerced (= converted) into the factor data type, but once we have loaded the data we can do the conversion using the factor() function

```
head(penguins)
penguins$species <- as.character(penguins$species)
levels(penguins$species)

penguins$species <- factor(penguins$species)
head(penguins)
levels(penguins$species)
```

### Renaming Factors

```
# Inspect the contents of the column
levels(penguins$species)

# Replace level names with a new vector of names
levels(penguins$species) <- c("P. adeliae", "P. antarcticus", "P. papua")
head(penguins)

ggplot(data = penguins, aes(x = bill_length_mm, y = bill_depth_mm, color = species, fill
  geom_point()

#Reorder new names
penguins$species <- factor(penguins$species, levels = c("P. papua",  "P. adeliae", "P. a

ggplot(data = penguins, aes(x = bill_length_mm, y = bill_depth_mm, color = species, fill
  geom_point()
```

# Manipulating, analyzing and exporting data with tidyverse

## Data manipulation using dplyr and tidyr

dplyr is a package for helping with tabular data manipulation. It pairs nicely with tidyr which enables you to convert between different data formats for plotting and analysis.

The tidyverse package is an "umbrella-package" that installs tidyr, dplyr, and several other useful packages for data analysis, such as ggplot2, tibble, etc.

dplyr functions

- `select()` subset columns
- `filter()` subset rows on conditions
- `mutate()` create new columns by using information from other columns
- `group_by()` and `summarize()` create summary statistics on grouped data
- `arrange()` sort results
- `count()` count discrete values

### Selecting columns and filtering rows

- `select()` subset columns
  - The first argument is the data frame (penguins), and the subsequent arguments are the columns to keep.

- `filter()` subset rows on conditions

```r
# Install Tidyverse
# install.packages("tidyverse")

# Load the umbrella package Tidyverse
library(tidyverse)
```

```
## -- Attaching core tidyverse packages ------------------------ tidyverse 2.0.0 --
## v dplyr     1.1.4     v readr     2.1.5
## v forcats   1.0.0     v stringr   1.5.1
## v lubridate 1.9.3     v tibble    3.2.1
## v purrr     1.0.2     v tidyr     1.3.1
## -- Conflicts ------------------------------------------ tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
## i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts
```

```r
# Reloading data to ensure nothing has been manipulated
data(penguins)
```

```r
# Select specific columns
select(penguins, species, bill_length_mm, bill_depth_mm)
```

Select all columns except certain ones by using a "-" in front of the variable to exclude it

```r
select(penguins, -bill_length_mm, -bill_depth_mm)
```

Choose rows based on specific criterion with `filter()`

```r
filter(penguins, species == "Gentoo")
```

### Pipes

What if you want to select and filter at the same time? There are three ways to do this: use intermediate steps, nested functions, or pipes.

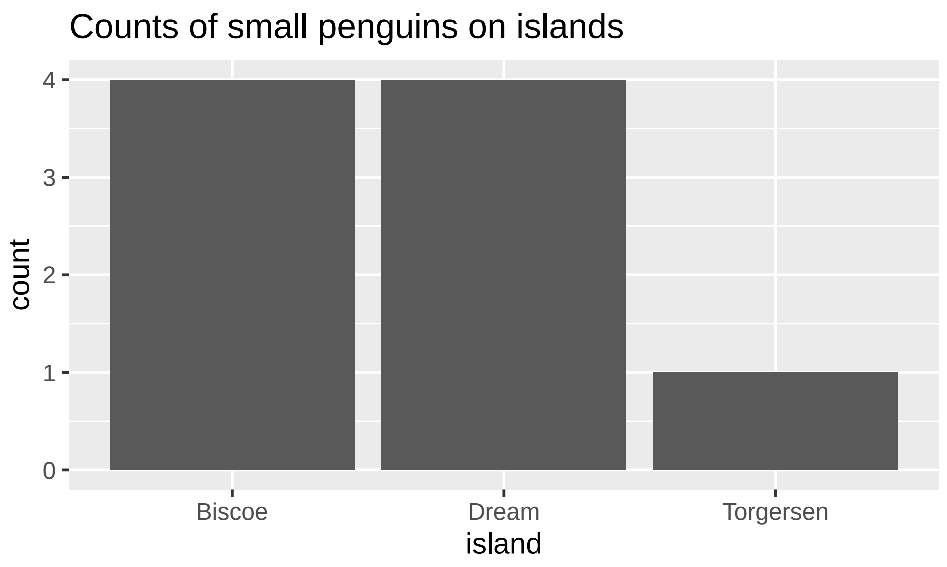
Pipes let you take the output of one function and send it directly to the next, which is useful when you need to do many things to the same data set. Pipes in R look like `%>%` and are made available via the magrittr package, installed automatically with dplyr. If you use RStudio, you can type the pipe with `Ctrl` + `Shift` + `M` if you have a PC or `Cmd` + `Shift` + `M` if you have a Mac.

```r
# Filter rows and save as an object
small <- filter(penguins, body_mass_g < 3000)
small

# Select columns and save as another object
small_year_only <- select(small, year)
small_year_only

# Combine the two operations in one piece of code using pipes
penguins %>%
  filter(body_mass_g < 3000) %>%
  select(island)
```

Create a plot using the subset data

```r
# Create a new object with the subset data
small_islands <- penguins %>%
  filter(body_mass_g < 3000) %>%
  select(island)

# Use new data to create a plot object
ggplot(small_islands, aes(x = island)) +
  geom_histogram(stat = "count") +
  labs(title = "Counts of small penguins on islands")
```

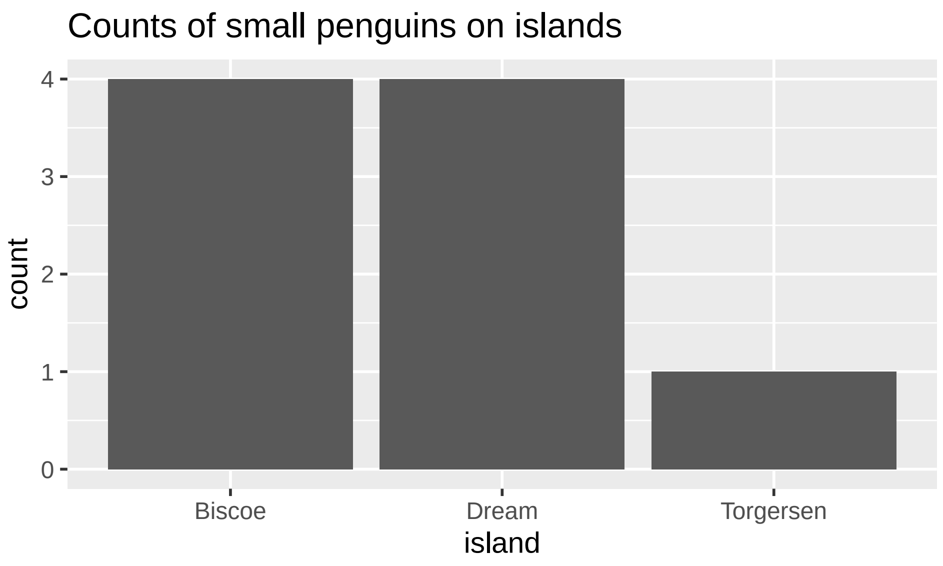
```
## Warning in geom_histogram(stat = "count"): Ignoring unknown parameters:
## ‘binwidth‘, ‘bins‘, and ‘pad‘
```



Write the subset data and plot in one piece of code

```
penguins %>%
  filter(body_mass_g < 3000) %>%
  select(island) %>%
  ggplot(aes(x = island)) +
  geom_histogram(stat = "count") +
  labs(title = "Counts of small penguins on islands")
```

```
## Warning in geom_histogram(stat = "count"): Ignoring unknown parameters:
## ‘binwidth‘, ‘bins‘, and ‘pad‘
```



**Mutate**

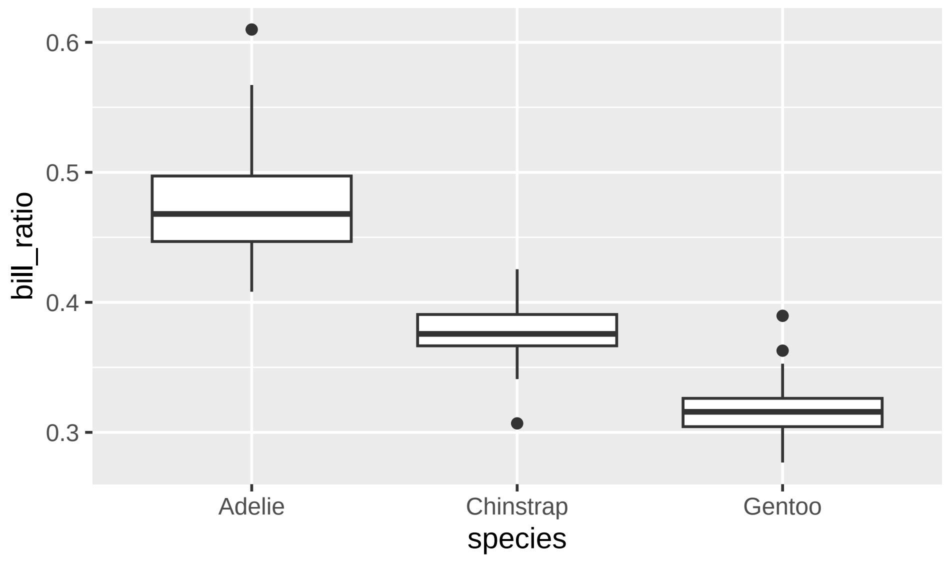
Create new columns based on the values in existing columns using `mutate()`

```
penguins %>%
  mutate(bill_ratio = bill_depth_mm / bill_length_mm,
         bill_percent = bill_ratio * 100)
```

Add a plot in one piece of code

```
penguins %>%
  mutate(bill_ratio = bill_depth_mm / bill_length_mm,
         bill_percent = bill_ratio * 100)  %>%
  ggplot(aes(x = species, y = bill_ratio)) +
  geom_boxplot()
```

```
## Warning: Removed 2 rows containing non-finite outside the scale range
## (‘stat_boxplot()‘).
```



**Split-apply-combine data analysis and the `summarize()` function**

Many data analysis tasks can be approached using the **split-apply-combine paradigm**

- Split the data into groups
- Apply some analysis to each group
- Combine the results

Key functions of dplyr for this workflow are `group_by()` and `summarize()`. `group_by()` is often used together with `summarize()`, which collapses each group into a single-row summary of that group.

```
penguins %>%
  group_by(species) %>%
  summarize(bill_length_mean = mean(bill_length_mm)) #need to add na.rm = TRUE
```

Once the data are grouped, you can also summarize multiple variables at the same time. Add a column indicating the minimum and maximum width for each species.

```
penguins %>%
  filter(!is.na(bill_length_mm)) %>%
  group_by(species) %>%
  summarize(bill_length_mean = mean(bill_length_mm),
            bill_depth_mean = mean(bill_depth_mm))
```

### Exporting plots

`ggsave()` allows you easily change the dimension and resolution of your plot by adjusting the appropriate arguments (width, height, and dpi)

```r
my_plot <- ggplot(data = penguins, aes(x = bill_length_mm, y = bill_depth_mm)) +
  geom_point()

ggsave("penguin_bills.png", my_plot, width = 8, height = 5)
```