A Brief Introduce to R and Useful Packages

Sheida Majouni

2023-05-04

1. Objectives

- Become familiar with basic concept in R
- Work with different Packages with a comparison with Python

2. Basic concept in R

2.1. Vector == Array

- In R, a vector is a basic data structure that represents a collection of elements of the same data type. It is one-dimensional and can hold data of any type, such as numeric, character, or logical values.
- vector start from 1 not 0
- various functions for vector are such as length(), sum(), mean(), and max()

```
myVector <- c(1,4,3,2)
myVector
```

```
## [1] 1 4 3 2
```

```
print(myVector[2]) # print the second element from beginning of vector
```

```
## [1] 4
```

```
print(myVector[-2]) # print the second element from the end of vector
```

```
## [1] 1 3 2
```

```
print(myVector[2:3]) # print the element from second to third of the vector
```

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

```
print(myVector[myVector > 2]) # print all element bigger that 2
```

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

```
cat("The second element of x is ", myVector[2], "\n") # print with format
```

```
## The second element of x is  4
```

```
print(length(myVector)) # print the length of the vector
```

```
## [1] 4
```

```
myVector <- c(myVector, 90) # adding at the end
myVector <- c(30, myVector) # adding at the beginning
```

2.2. Factor

In R, a factor is a categorical variable that represents a discrete number of possible values, each of which belongs to a specific category. Factors are often used to represent qualitative or nominal data. The levels of a factor are the unique values that it can take. by default based on alphabet

```
expression <- c("low", "high", "medium", "high", "low", "medium", "high")
expression <- factor(expression)
expression
```

```
## [1] low    high   medium high   low    medium high
## Levels: high low medium
```

```
# to change the order
levels(expression) <- c("low", "medium", "high")
expression
```

```
## [1] medium low    high   low    medium high   low
## Levels: low medium high
```

```
# Create a factor with custom levels
education <- c("high school", "bachelor's", "master's", "bachelor's", "high schoo
l")
education_factor <- factor(education, levels = c("high school", "bachelor's", "mast
er's"))
print(education_factor)
```

```
## [1] high school bachelor's  master's    bachelor's  high school
## Levels: high school bachelor's master's
```

2.3. Matrix = 2D array

In R, a matrix is a two-dimensional data structure that contains elements of the same data type, arranged in rows and columns. It is a useful data structure for storing and manipulating numeric data.

```
m <- matrix(c(1, 2, 3, 4, 5, 6), nrow = 3, ncol = 2)
print(m)
```

```
##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6
```

```
# get the shape of matrix
shape <- dim(m)
print(shape)
```

```
## [1] 3 2
```

```
# re-shape of the matrix
dim(m) <- c(2, 3)
print(m)
```

```
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
```

```
# to print the first row
print(m[1, ])
```

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

```
# to print the second column
print(m[, 2])
```

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

```
# to create a new matrix of the column 2 to 3
m2 <- cbind(m[, 2:3])
print(m2)
```

```
##      [,1] [,2]
## [1,]    3    5
## [2,]    4    6
```

2.4. Data Frame

In R, a dataframe is a two-dimensional rectangular table-like structure that consists of rows and columns. The rows correspond to cases (observations), while the columns correspond to variables (attributes). Each column in a dataframe can have a different data type (e.g., character, numeric, factor, etc.), but all columns must have the same length. Dataframes are the most commonly used data structure in R and are often used for data manipulation, cleaning, and analysis.

```
df <- data.frame(
  name = c("Alice", "Bob", "Charlie"),
  age = c(25, 30, 35),
  gender = c("female", "male", "male")
)

print(df)
```

```
##      name age gender
## 1   Alice  25 female
## 2     Bob  30   male
## 3 Charlie  35   male
```

```
print(df$age) # return an array
```

```
## [1] 25 30 35
```

```
print(df["age"]) # return an df
```

```
##   age
## 1  25
## 2  30
## 3  35
```

2.5. Functions

x = c(1,2,3)

y = c(2,5,3)

Some statistics functions:

- mean(x) #mean of a numeric vector x.
- sd(x) #standard deviation of a numeric vector x.
- var(x) #variance of a numeric vector x.
- median(x) #median of a numeric vector x.
- min(x) #returns the minimum value of a numeric vector x.
- max(x) #returns the maximum value of a numeric vector x.
- cor(x, y) #calculates the correlation coefficient between two numeric vectors x and y.
- cov(x, y) #calculates the covariance between two numeric vectors x and y.
- ?round # for help and understand the function
- args(round) # returns the names of the arguments
- example("round") # gives you some examples of the function

** we can also have a chain of functions

```
library(dplyr)
```

```
##
## Attaching package: 'dplyr'
```

```
## The following objects are masked from 'package:stats':
##
##     filter, lag
```

```
## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

```
x = c(1,2,3)
y = c(2,5,3)

#  to chain together multiple functions in a single line of code
x <- sqrt(16) %>% exp() # pass the output of sqrt to exp input and ...
x
```

```
## [1] 54.59815
```

3. Packages

R packages are collections of functions, data, and documentation that extend the capabilities of R. They are a powerful tool that makes it easy to perform complex tasks in R. Once installed, R packages can be loaded into R sessions using the library() function. It is important to note that even if you have previously installed a package, you will need to load it again in each new R session to be able to use it.

You can also use search() function to check which library are loaded so far

3.1. Install packages and load them

```
# install packages only once
# install.packages(c("datasauRus", "tidyverse", "dplyr", "readr", "tibble", "ggplot
2"))

# load only needed packages in each project
library(datasauRus)
library(tidyverse)
library(dplyr)
library(readr)
library(tibble)
library(ggplot2)

search()
```

```
##  [1] ".GlobalEnv"          "package:lubridate"  "package:forcats"
##  [4] "package:stringr"    "package:purrr"      "package:readr"
##  [7] "package:tidyr"      "package:tibble"     "package:ggplot2"
## [10] "package:tidyverse"  "package:datasauRus" "package:dplyr"
## [13] "package:stats"      "package:graphics"   "package:grDevices"
## [16] "package:utils"      "package:datasets"   "package:methods"
## [19] "Autoloads"          "package:base"
```

3.2. Core tidyverse

The core tidyverse includes the packages that you're likely to use in everyday data analyses. As of tidyverse 1.3.0, the following packages are included in the core tidyverse: ggplot2, dplyr, tidyr, readr, purrr, tibble, stringr, forcats

The tidyverse also includes many other packages with more specialized usage. They are not loaded automatically with library(tidyverse), so you'll need to load each one with its own call to library().

3.2.1 dplyr

dplyr provides a grammar of data manipulation, providing a consistent set of verbs that solve the most common data manipulation challenges.

dplyr::mutate()

Used for adding new variables to a dataset. Here's an example of adding a new variable that calculates the total price of a product based on its unit price and quantity:

Python equivalent: pandas.DataFrame.assign().

```
# Create sample data
df <- data.frame(product = c("A", "B", "C"),
                 unit_price = c(10, 20, 30),
                 quantity = c(5, 2, 3))

# Add new variable for total price
df <- df %>%
  mutate(total_price = unit_price * quantity)

# View updated data
df
```

```
##   product unit_price quantity total_price
## 1       A         10        5          50
## 2       B         20        2          40
## 3       C         30        3          90
```

dplyr::select()

Used for selecting specific variables from a dataset. Here’s an example of selecting only the product and quantity variables from the df

Python equivalent: pandas.DataFrame.loc[:, column_names] or pandas.DataFrame[column_names].

```
df_selected <- df %>%
  select(product, quantity)

# Output the resulting dataframe
df_selected
```

```
##   product quantity
## 1       A        5
## 2       B        2
## 3       C        3
```

dplyr::filter()

Used for filtering rows based on a condition. Here’s an example of filtering a dataset to only include rows where the unit_price is greater than or equal to 20

Python equivalent: pandas.DataFrame.query().

```
# filter rows where the unit_price is greater than or equal to 20
filtered_df <- df %>%
  filter(unit_price >= 20)

# print the filtered dataframe
print(filtered_df)
```

```
##   product unit_price quantity total_price
## 1       B         20        2          40
## 2       C         30        3          90
```

dplyr::arrange()

Used for sorting a dataset by one or more variables. Here's an example of sorting the df by the quantity variable in descending order:

Python equivalent: pandas.DataFrame.sort_values()

```
# Sort data by cylinders in descending order
df %>%
  arrange(desc(quantity))
```

```
##   product unit_price quantity total_price
## 1       A         10        5          50
## 2       C         30        3          90
## 3       B         20        2          40
```

dplyr::summarise()

Used to summarize data into a single value or a small set of values. It is often used in combination with other functions such as group_by()

Python equivalent: pandas.DataFrame.groupby().agg() or pandas.DataFrame.groupby().apply()

```
df %>%
  summarise(total_sales = sum(unit_price * quantity))
```

```
##   total_sales
## 1         180
```

dplyr::group_by()

a function for grouping a data frame by one or more variables. In this example, we first use the group_by() function to group the data frame by the 'product' column. This creates a new object called grouped_df, which contains the original data frame but with rows grouped by the unique values in the 'product' column.

Next, we use the summarize() function to calculate summary statistics for each group. This creates a new data frame called summarized_df, which contains one row for each unique value in the ‘product’ column, along with the mean ‘unit_price’ and ‘quantity’ for each group.

Python equivalent: pandas.DataFrame.groupby().

```
df2 <- data.frame(product = c("A", "B", "C", "A", "B"),
                  unit_price = c(10, 20, 30, 15, 25),
                  quantity = c(5, 2, 3, 6, 1))

# Group the data frame by the 'product' column
grouped_df <- dplyr::group_by(df2, product)

# Calculate the mean 'unit_price' and 'quantity' for each group
summarized_df <- dplyr::summarize(grouped_df, avg_unit_price = mean(unit_price), av
g_quantity = mean(quantity))
summarized_df
```

```
## # A tibble: 3 × 3
##   product avg_unit_price avg_quantity
##   <chr>            <dbl>        <dbl>
## 1 A                 12.5          5.5
## 2 B                 22.5          1.5
## 3 C                 30            3
```

3.2.2. tidyr

The goal of tidyr is to help you create tidy data. Tidy data is data where:

- Every column is variable.
- Every row is an observation.
- Every cell is a single value.

Tidy data describes a standard way of storing data that is used wherever possible throughout the tidyverse.

tidyr::gather()

This function converts “wide” data to “long” data by gathering columns into key-value pairs. In this example, the gather function is used to convert the data from wide format to long format, with the country column as the id column, the year column as the key column, and the expectancy column as the value column. The resulting dataframe

Python equivalent: pd.melt()

```r
# create a small life expectancy dataset
life_expectancy <- data.frame(
  country = c("Australia", "Canada", "France"),
  `2010` = c(82.0, 80.7, 81.8),
  `2015` = c(82.4, 81.5, 82.3),
  `2020` = c(83.0, 81.9, 83.0)
)
life_expectancy
```

```
##     country X2010 X2015 X2020
## 1 Australia  82.0  82.4  83.0
## 2    Canada  80.7  81.5  81.9
## 3    France  81.8  82.3  83.0
```

```r
# use gather to convert the data from wide to long format
life_expectancy_long <- gather(life_expectancy, year, expectancy, -country)

# print the long format data
print(life_expectancy_long)
```

```
##     country  year expectancy
## 1 Australia X2010       82.0
## 2    Canada X2010       80.7
## 3    France X2010       81.8
## 4 Australia X2015       82.4
## 5    Canada X2015       81.5
## 6    France X2015       82.3
## 7 Australia X2020       83.0
## 8    Canada X2020       81.9
## 9    France X2020       83.0
```

tidyr::spread()

This function is the opposite of gather(). It takes key-value pairs and spreads them out into separate columns.

Python equivalent: pd.pivot()

```r
# use spread to convert the data from long to wide format
life_expectancy_wide <- spread(life_expectancy_long, key = year, value = expectanc
y)

# print the wide format data
print(life_expectancy_wide)
```

```
##     country X2010 X2015 X2020
## 1 Australia  82.0  82.4  83.0
## 2    Canada  80.7  81.5  81.9
## 3    France  81.8  82.3  83.0
```

tidyr::separate()

This function separates one column into multiple columns based on a separator character.

Python equivalent: str.split()

```
df <- data.frame(x = c("a_1", "b_2"))
tidyr::separate(df, x, c("letter", "number"), sep = "_")
```

```
##   letter number
## 1      a      1
## 2      b      2
```

tidyr::fill()

This function replaces missing values with the previous non-missing value.

Python equivalent: df.fillna(method='ffill')

```
df <- data.frame(x = c(1, NA, NA, 4))
tidyr::fill(df, x)
```

```
##   x
## 1 1
## 2 1
## 3 1
## 4 4
```

tidyr::drop_na()

This function removes rows with missing values.

Python equivalent: df.dropna()

```
df <- data.frame(x = c(1, 2, NA), y = c(3, NA, 5))
tidyr::drop_na(df)
```

```
##   x y
## 1 1 3
```

tidyr::replace_na()

This function replaces missing values with a specified value.

Python equivalent: df.fillna(value)

```
df <- data.frame(x = c(1, 2, NA), y = c(3, NA, 5))
tidyr::replace_na(df, list(x = 0, y = 99))
```

```
##   x  y
## 1 1  3
## 2 2 99
## 3 0  5
```

3.2.2. readr

The goal of readr is to provide a fast and friendly way to read rectangular data from delimited files, such as comma-separated values (CSV) and tab-separated values (TSV).

- read_csv(): comma-separated values (CSV)
- read_tsv(): tab-separated values (TSV)
- read_csv2(): semicolon-separated values with , as the decimal mark
- read_delim(): delimited files (CSV and TSV are important special cases)
- read_fwf(): fixed-width files
- read_table(): whitespace-separated files
- read_log(): web log files

3.2.3. tibble

Tibble is an R package that provides a modern reimagining of the data.frame. It offers a suite of functions that make it easy to create, manipulate, and analyze data in R.

tibble::tibble()

Creates a tibble from input vectors or data frames.

Python equivalent: pandas.DataFrame: Creates a DataFrame from input arrays or lists.

```
# create a tibble with three columns
library(tibble)
tb <- tibble(a = 1:3, b = c("one", "two", "three"), c = TRUE)

# print the tibble
print(tb)
```

```
## # A tibble: 3 × 3
##       a b     c
##   <int> <chr> <lgl>
## 1     1 one   TRUE
## 2     2 two   TRUE
## 3     3 three TRUE
```

More functions of tibble:

- as_tibble(): Converts a data frame to a tibble.
- add_column(): Adds new columns to a tibble.
- select(): Selects specific columns of a tibble.
- filter(): Filters rows of a tibble based on a condition.
- mutate(): Adds new columns or modifies existing columns in a tibble.
- group_by(): Groups a tibble by one or more columns.
- summarize(): Aggregates data within groups defined by group_by().
- slice(): Selects specific rows of a tibble.
- arrange(): Sorts a tibble by one or more columns.

3.2.4. ggplot2

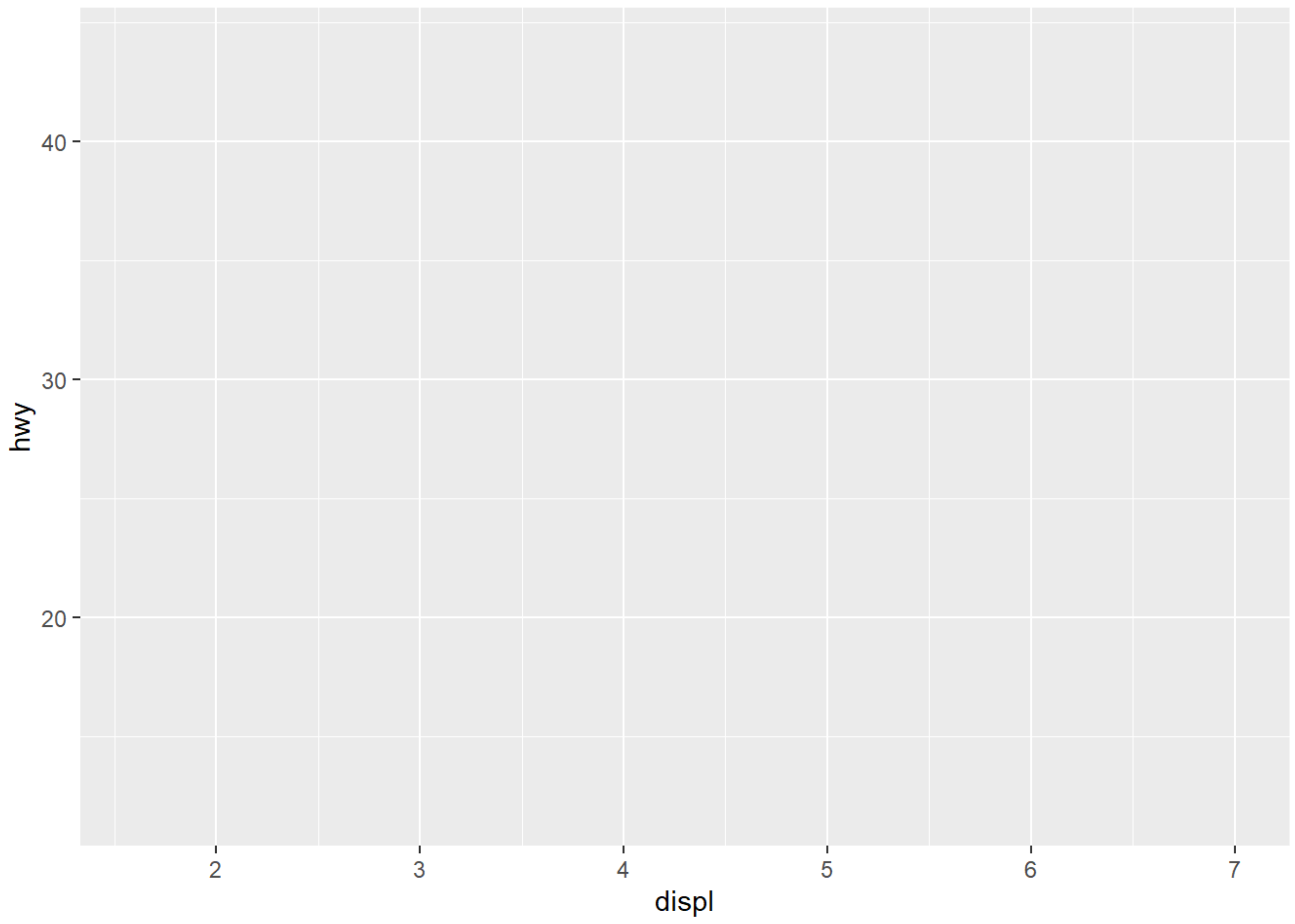
ggplot2 is a popular package in R used for data visualization.

ggplot2::ggplot()

Initializes a plot and sets its default data and aesthetics.

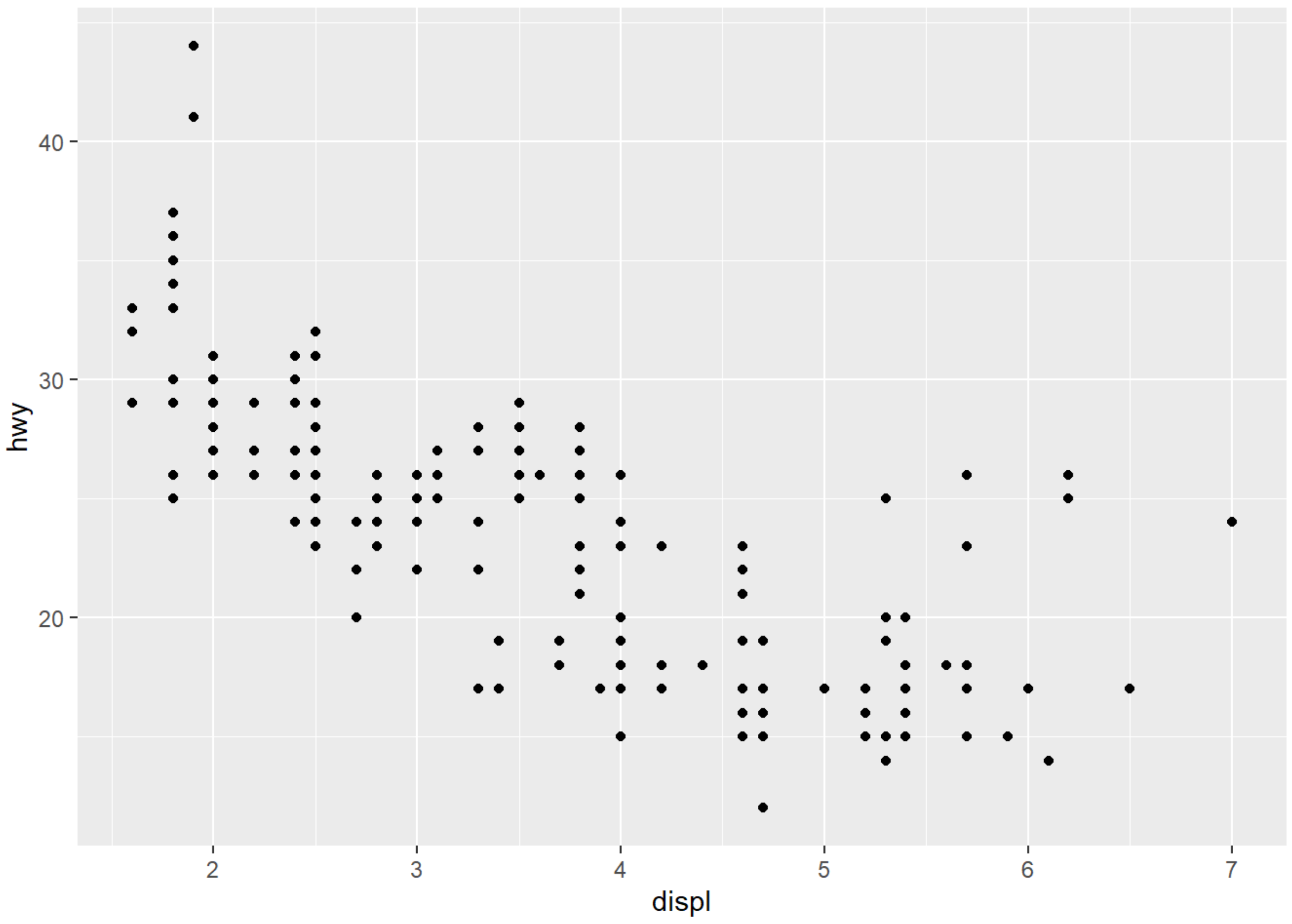
Equivalent in matplotlib: plt.subplots().

```
# created a blank canvas for the plot. any visual elements haven't added to it yet.
data(mpg)
p <- ggplot(data = mpg, aes(x = displ, y = hwy))
p
```



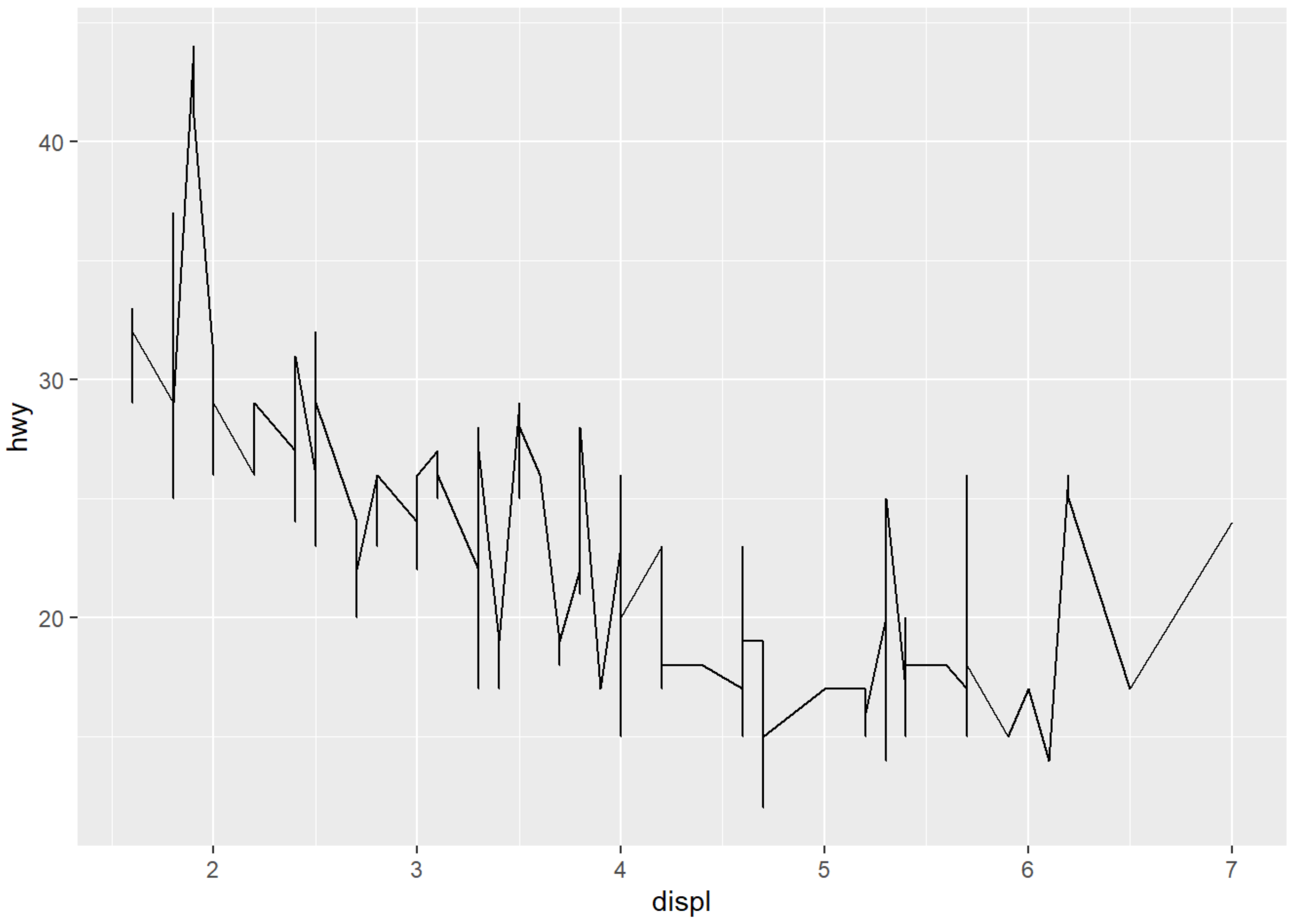
geom_point(): This is used to add points to the plot. Here's an example:

```
p + geom_point()
```



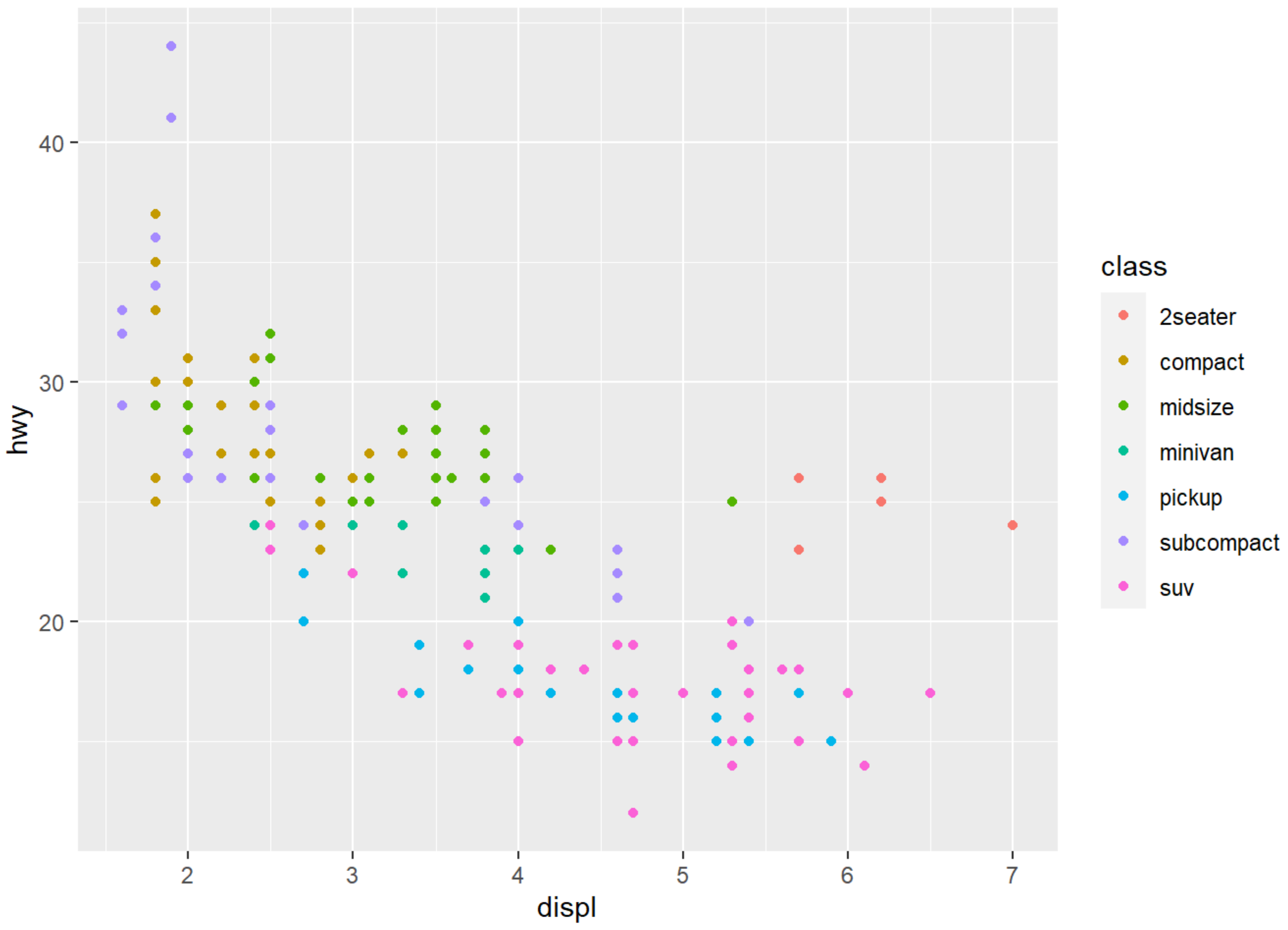
geom_line(): This is used to add lines to the plot. Here’s an example:

```
p + geom_line()
```



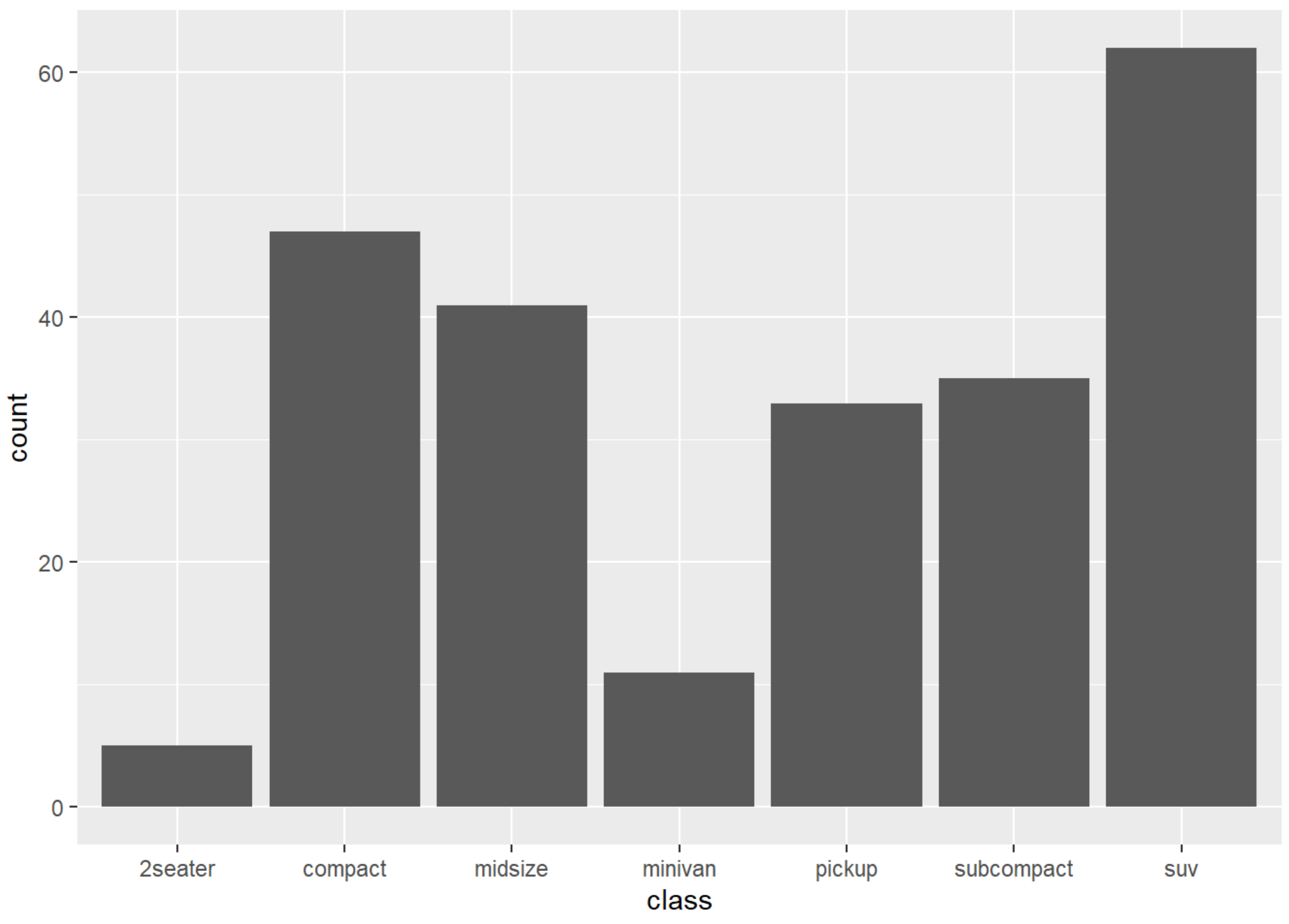
aes(): This is used to specify the aesthetic mappings for the plot. Here's an example:

```
p <- ggplot(data = mpg, aes(x = displ, y = hwy, colour = class)) +
  geom_point()
p
```



geom_bar(): This is used to add bar plots to the plot. Here’s an example:

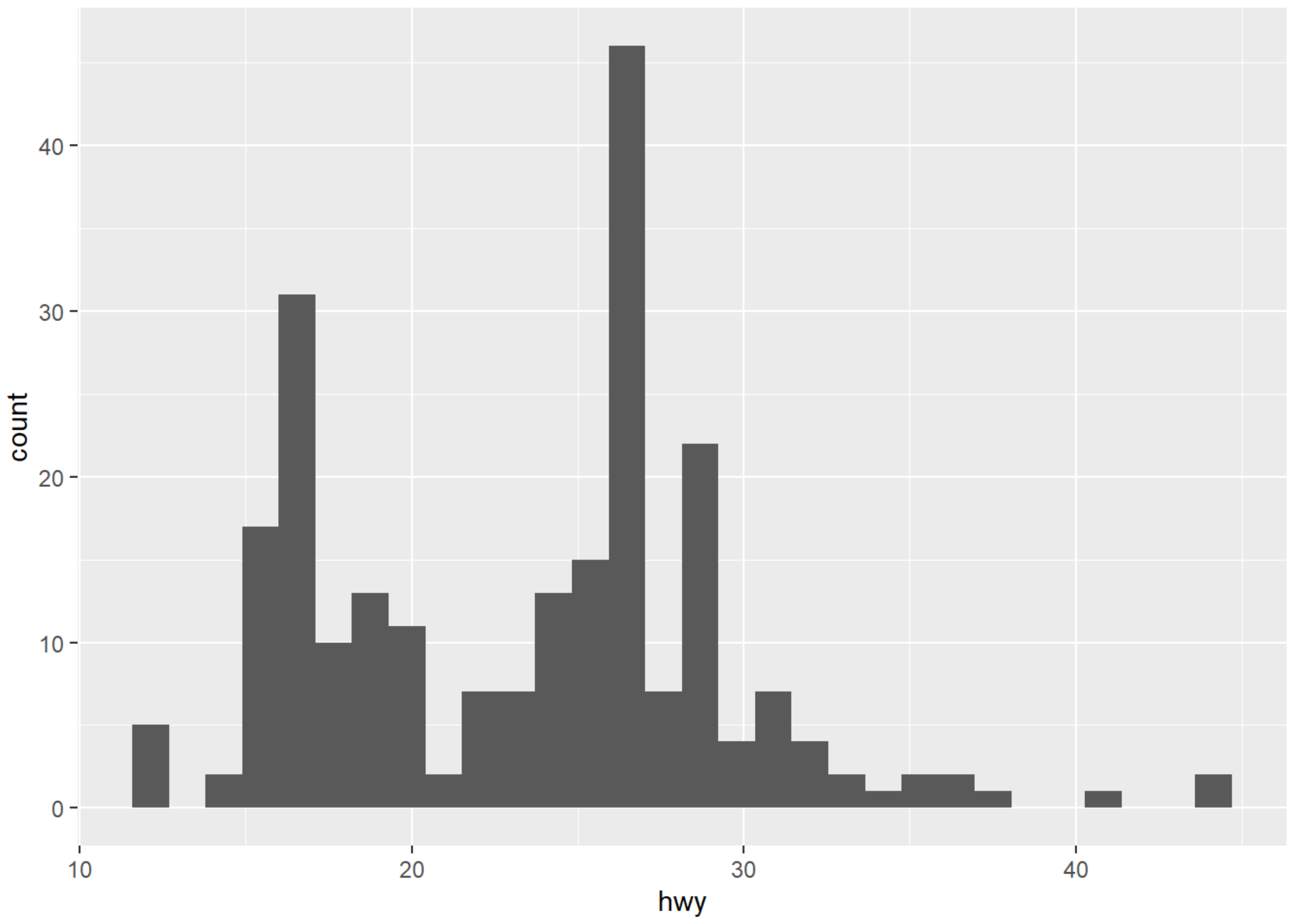
```
ggplot(data = mpg, aes(x = class)) + geom_bar()
```



geom_histogram(): This is used to add histograms to the plot. Here's an example:

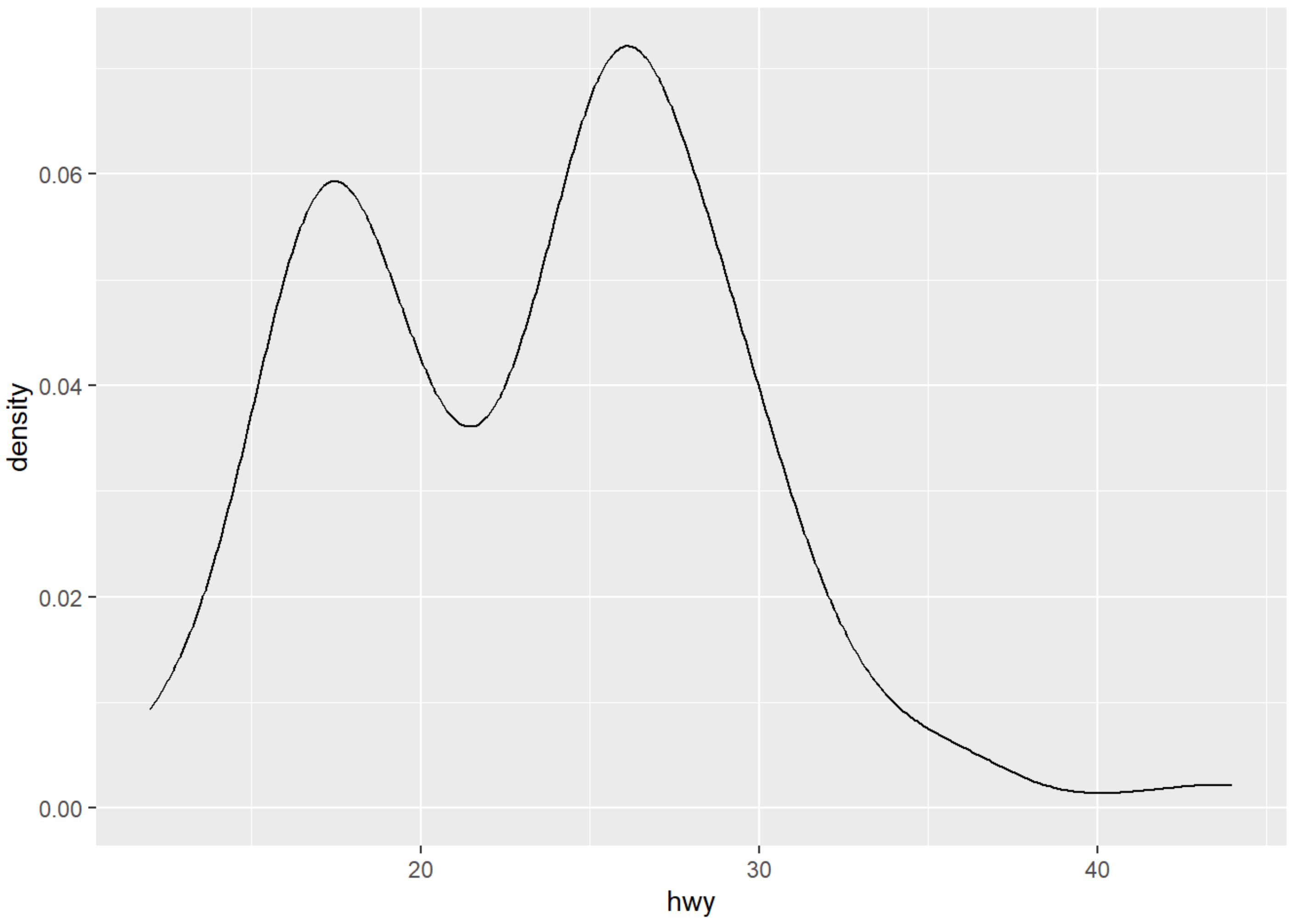
```
ggplot(data = mpg, aes(x = hwy)) + geom_histogram()
```

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



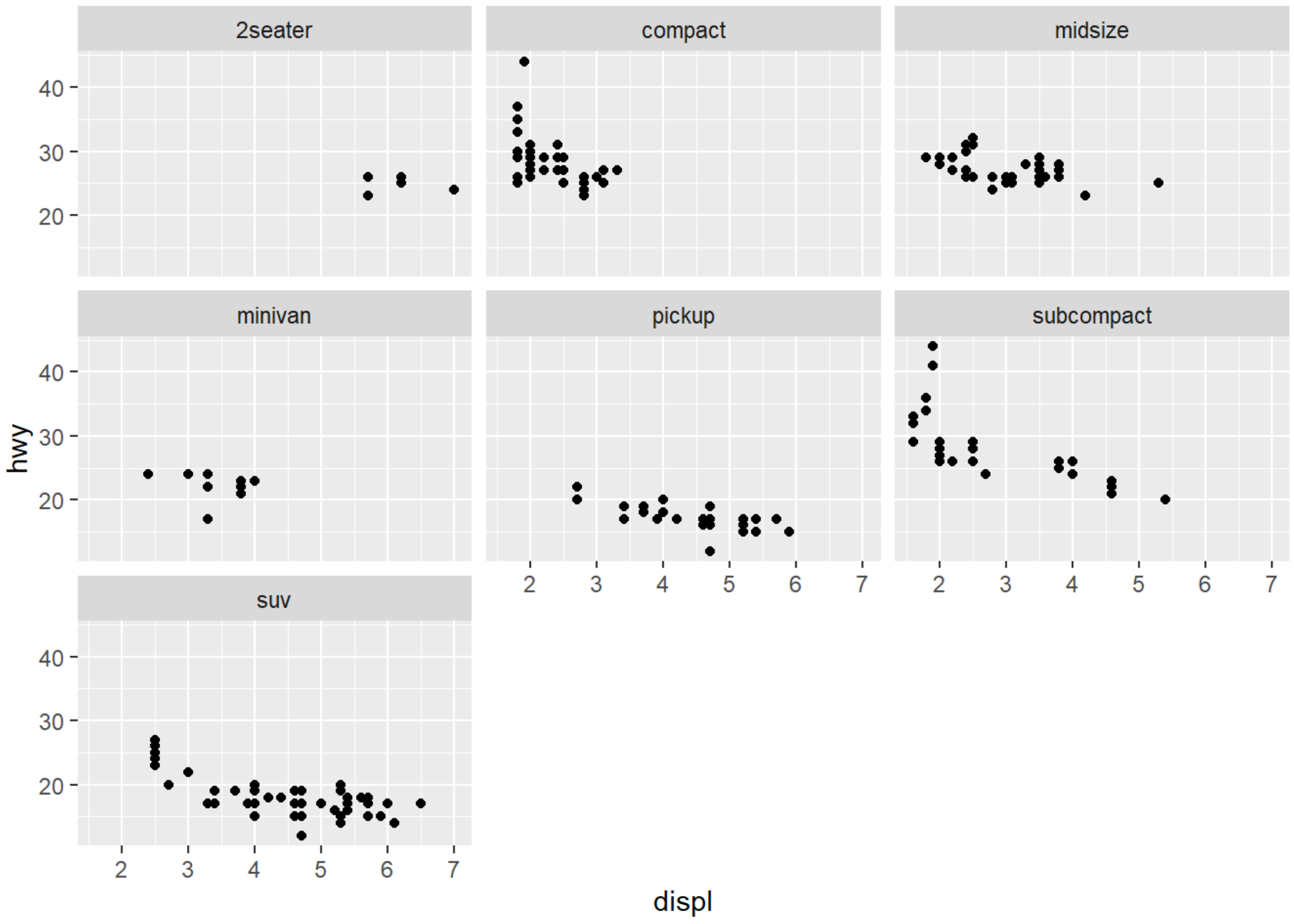
geom_density(): This is used to add density plots to the plot. Here’s an example:

```
ggplot(data = mpg, aes(x = hwy)) + geom_density()
```



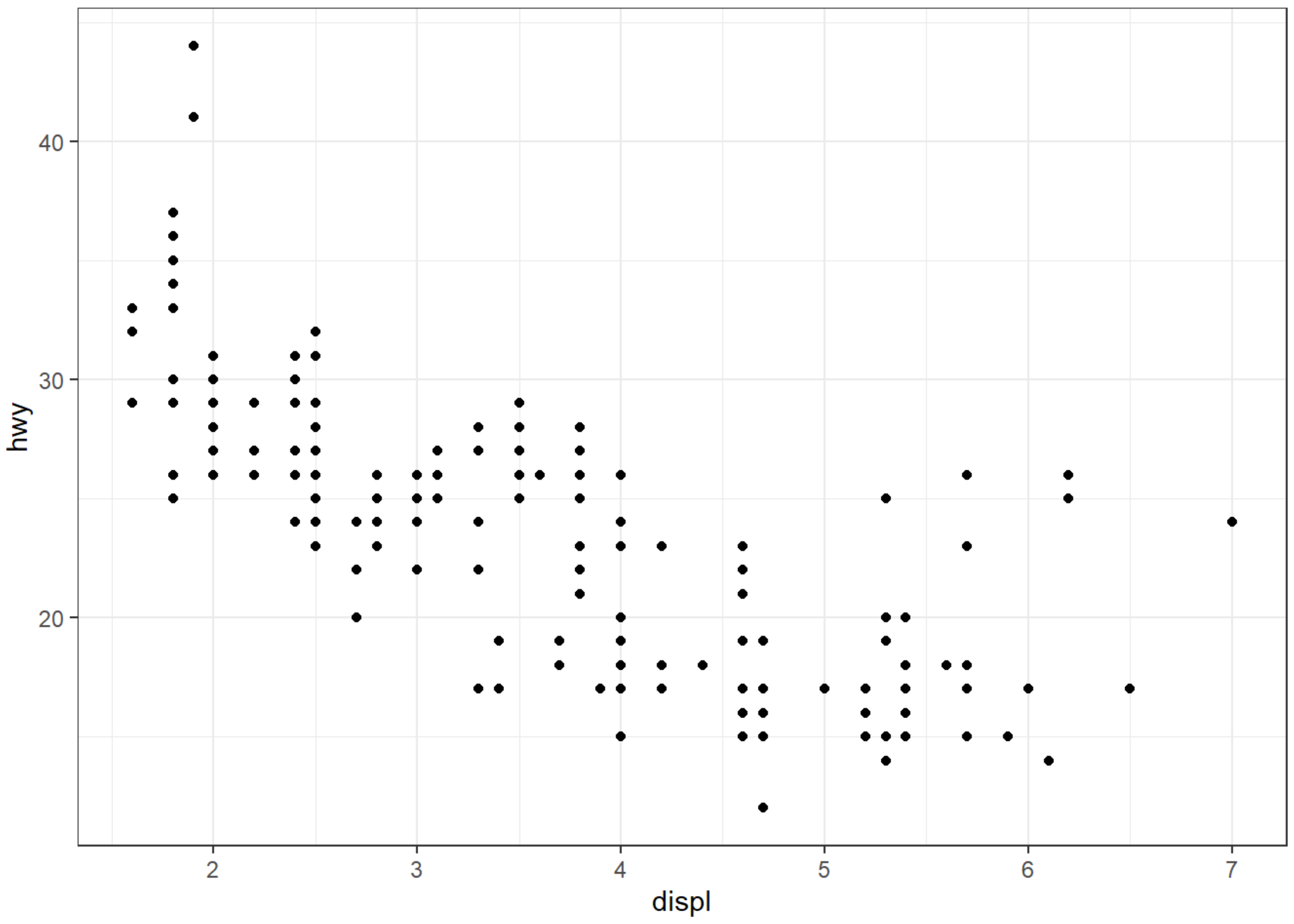
facet_wrap(): This is used to create a set of plots with one plot for each level of a categorical variable. Here's an example:

```
ggplot(data = mpg, aes(x = displ, y = hwy)) + geom_point() + facet_wrap(~class)
```



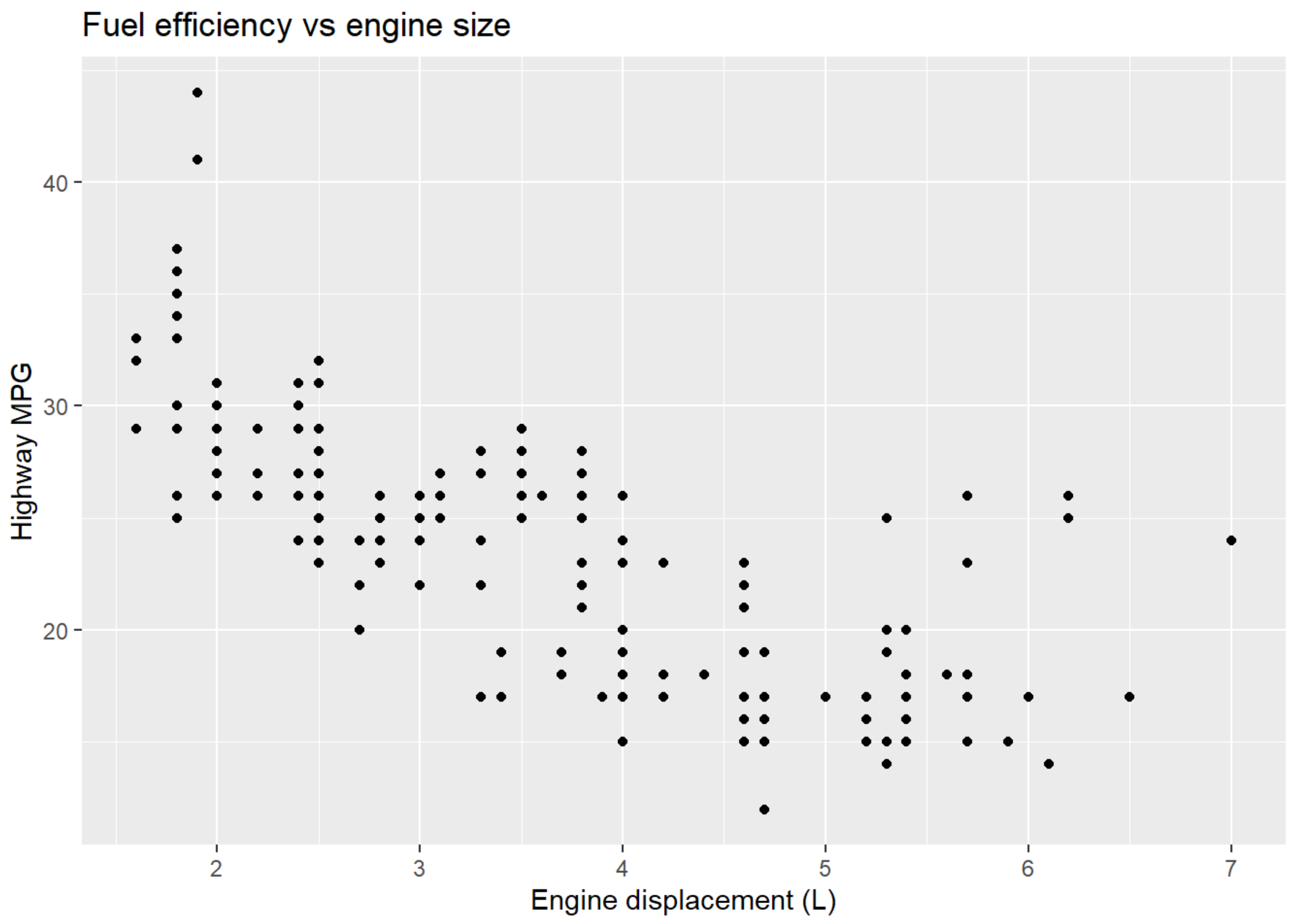
theme(): This is used to modify the overall appearance of the plot. Here's an example:

```
ggplot(data = mpg, aes(x = displ, y = hwy)) + geom_point() + theme_bw()
```



labs(): This is used to modify the axis labels and plot title. Here's an example:

```
ggplot(data = mpg, aes(x = displ, y = hwy)) + geom_point() + labs(x = "Engine displ
acement (L)", y = "Highway MPG", title = "Fuel efficiency vs engine size")
```



3.3. Some other useful packages for the rest of the semester:

- **reshape2** package is used for data reshaping and transformation in R. The Python equivalents mentioned here are based on the pandas library.
- **rsample** is an R package for data splitting and model evaluation. Some of the most commonly used functions in the package are:
  - initial_split(): splits data into training and testing datasets
  - vfold_cv(): creates cross-validation folds for model evaluation
  - group_vfold_cv(): creates grouped cross-validation folds for model evaluation
  - rolling_origin(): creates time-based rolling windows for model evaluation
  - assessment(): calculates evaluation metrics for models on a test set
  - summary(): provides a summary of the results of a model evaluation
  - confusion_matrix(): creates a confusion matrix for model evaluation
  - roc_curve(): creates a receiver operating characteristic (ROC) curve for model evaluation
  - calibration_curve(): creates a calibration curve for model evaluation
  - yardstick package: extends the functionality of rsample for a wider range of evaluation metrics
- **Caret** is an R package that stands for “Classification and Regression Training”. Some of the most commonly used functions in the package are:

  - train(): trains a predictive model using a specified algorithm
  - predict(): makes predictions using a trained model
  - createDataPartition(): creates partitions of the data for training and testing purposes
  - resamples(): creates multiple random splits of the data for cross-validation
  - trainControl(): specifies the type of cross-validation to use during model training
  - tuneGrid(): specifies the grid of hyperparameters to search over during model tuning
  - varImp(): computes variable importance measures for a trained model
  - confusionMatrix(): computes a confusion matrix for a set of predictions
  - plot(): creates diagnostic plots for a trained model
  - saveRDS(): saves a trained model object to a file for later use.
- **stringr** provides a cohesive set of functions designed to make working with strings as easy as possible.
- **scales** provides tools for scaling and formatting plots and axes in R.
- **topicmodels** package in R is used for text mining and modeling
- **tidytext** package provides a set of tools for text mining and natural language processing (NLP) in R.
- **textstem** package provides functions for stemming English language text data.
- **clinspacy** is an R package that provides a high-level interface to the Python-based natural language processing library spaCy. It is designed specifically for clinical text processing tasks, such as named entity recognition, concept extraction, and relationship extraction.