Functions and iteration

Week 14

AEM 2850 / 5850 : R for Business Analytics
Cornell Dyson
Spring 2024

Acknowledgements: Claus Wilke

Announcements

Plan for this week:

- Today: slides and a short example
- Thursday: course evaluations, prelim 2 Q&A, **lab-14**
    - If you come to class and work through the lab, you will get full credit
    - Due Thursday at 11:59pm so you can focus on studying for prelim 2

Plan for next week:

- **Prelim 2** in class on Tuesday, May 7
- **Lab 15 - Course Survey** due Monday, May 13 at 11:59pm

Questions before we get started?

Plan for today

Course evaluations

Functions and iteration

example-14

Course evaluations

Course objectives reminder

1. Develop basic proficiency in R programming
2. Understand data structures and manipulation
3. Describe effective techniques for data visualization and communication
4. Construct effective data visualizations
5. Utilize course concepts and tools for business applications

With these objectives in mind…

Please complete course evaluations

I take feedback seriously and will use it to improve this course!

Concrete suggestions are most helpful

I would appreciate your feedback through two channels:

1. Lab 15 - Course Feedback Survey (on canvas)
2. University course evaluations

Both will be anonymous

I will give you time to complete these in class on Thursday

Lab 15 - course feedback survey

Anonymous: canvas reports whether you submit but does not link responses to individuals

Survey is very short, should only take 5 minutes!

I will give you time in class Thursday to complete it

University course evaluations

Anonymous: we just get summary reports, after grades are submitted

I will award a bonus point on Lab 15 for completing evaluations

I will give you time in class Thursday to complete it

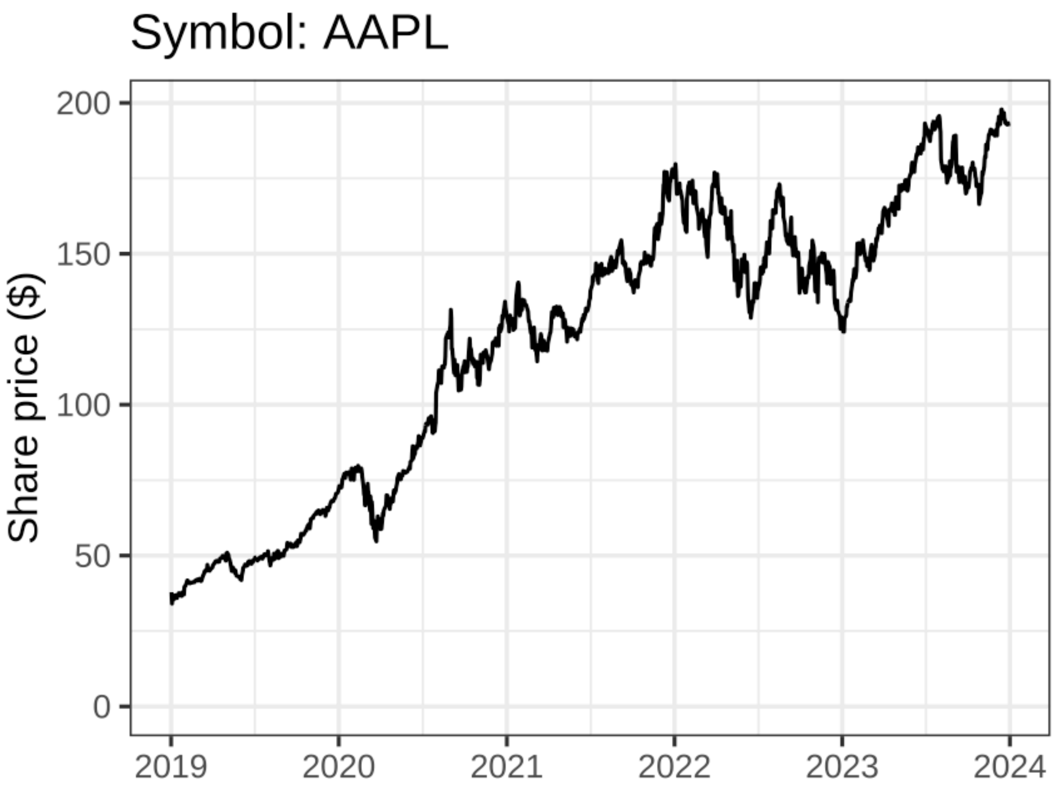
Thank you in advance for your feedback!

Functions and iteration

We often run similar code multiple times

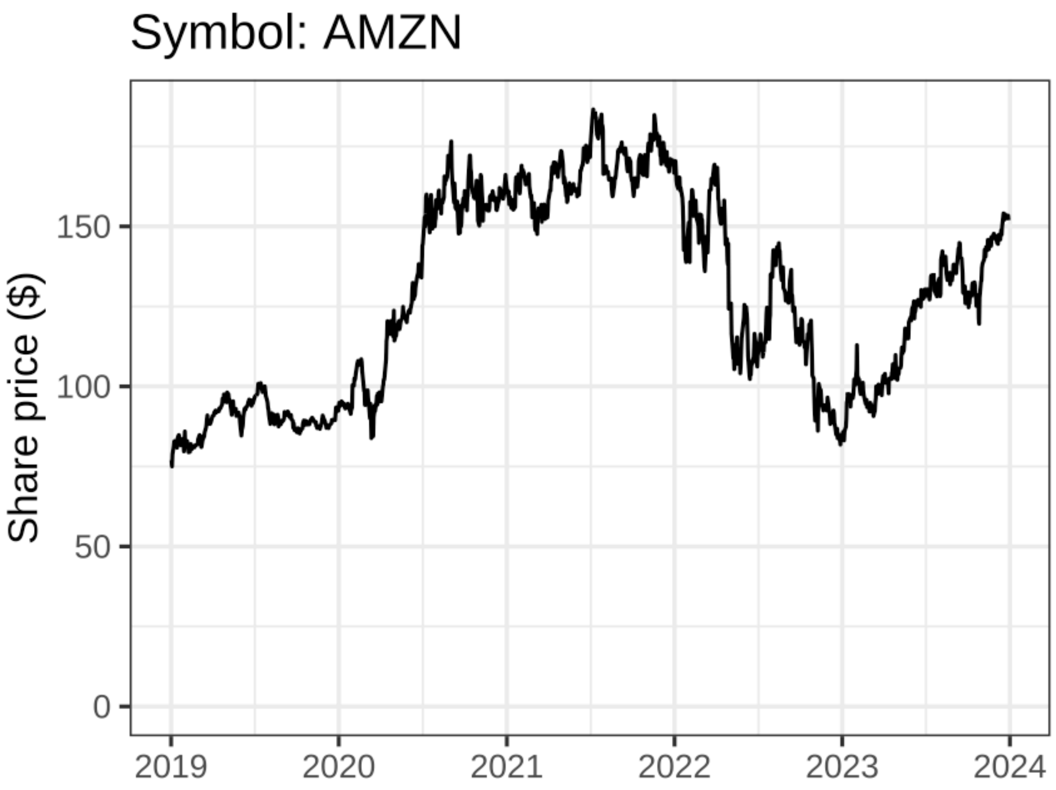
```r
sp500_prices |>
  filter(symbol == "AAPL") |>
  ggplot(aes(x = date, y = adjusted)) +
  geom_line() +
  labs(x = NULL,
       y = "Share price ($)",
       title = "Symbol: AAPL") +
  scale_x_date(date_breaks = "1 year",
               date_labels = "%Y") +
  scale_y_continuous(limits = c(0, NA)) +
  theme_bw()
```

What needs to change if we want to look at AMZN share prices instead?



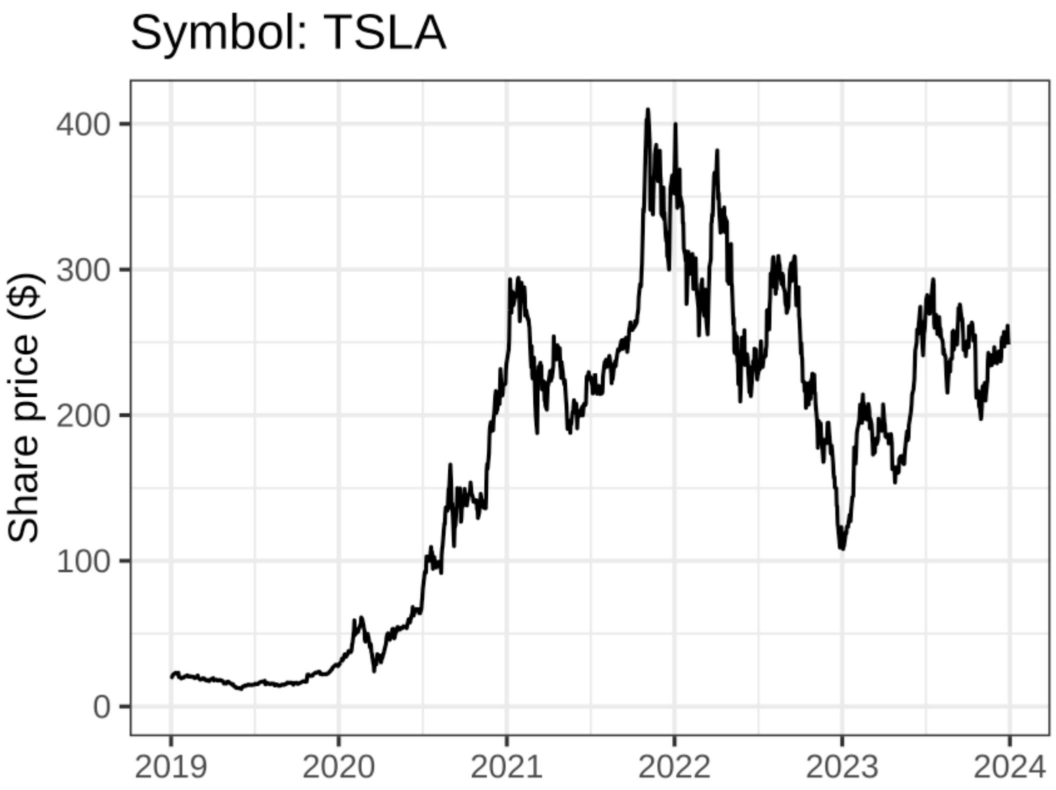
We often run similar code multiple times

```r
sp500_prices |>
  filter(symbol == "AMZN") |>
  ggplot(aes(x = date, y = adjusted)) +
  geom_line() +
  labs(x = NULL,
       y = "Share price ($)",
       title = "Symbol: AMZN") +
  scale_x_date(date_breaks = "1 year",
               date_labels = "%Y") +
  scale_y_continuous(limits = c(0, NA)) +
  theme_bw()
```



We often run similar code multiple times

```r
sp500_prices |>
  filter(symbol == "TSLA") |>
  ggplot(aes(x = date, y = adjusted)) +
  geom_line() +
  labs(x = NULL,
       y = "Share price ($)",
       title = "Symbol: TSLA") +
  scale_x_date(date_breaks = "1 year",
               date_labels = "%Y") +
  scale_y_continuous(limits = c(0, NA)) +
  theme_bw()
```



How can we avoid duplication and mistakes?

1. **Avoid hard-coding specific values**

2. **Define a function**

3. **Automate calling the function**

4. Write a more general function

5. Use these concepts in a tidy pipeline

We will focus on steps 1-3 due to time constraints

Step 1: Avoid hard-coding specific values

```r
sp500_prices |>
  filter(symbol == "AAPL") |>
  ggplot(aes(x = date, y = adjusted)) +
  geom_line() +
  labs(x = NULL,
       y = "Share price ($)",
       title = "Symbol: AAPL") +
  scale_x_date(date_breaks = "1 year",
               date_labels = "%Y") +
  scale_y_continuous(limits = c(0, NA)) +
  theme_bw()
```

What is "hard-coded" here?

Step 1: Avoid hard-coding specific values

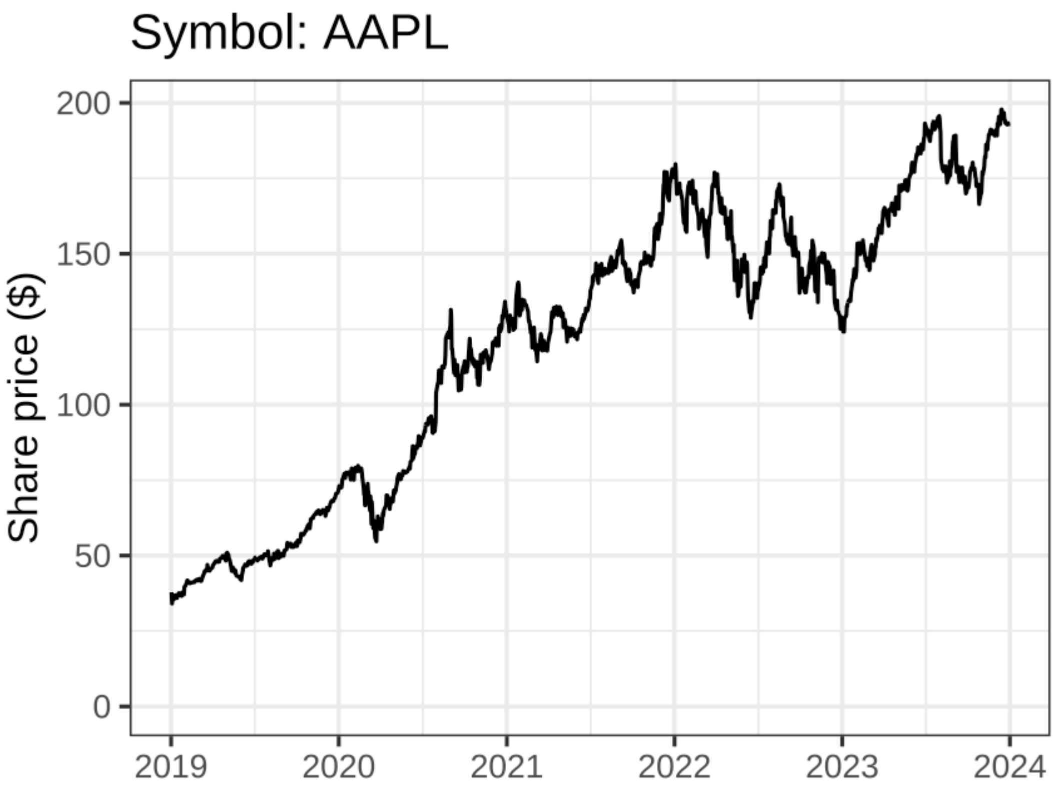
```r
sp500_prices |>
  filter(symbol == "AAPL") |>
  ggplot(aes(x = date, y = adjusted)) +
  geom_line() +
  labs(x = NULL,
       y = "Share price ($)",
       title = "Symbol: AAPL") +
  scale_x_date(date_breaks = "1 year",
               date_labels = "%Y") +
  scale_y_continuous(limits = c(0, NA)) +
  theme_bw()
```

How can we avoid this hard-coding?

Step 1: Avoid hard-coding specific values

```r
ticker <- "AAPL"

sp500_prices |>
  filter(symbol == ticker) |>
  ggplot(aes(x = date, y = adjusted)) +
  geom_line() +
  labs(x = NULL,
       y = "Share price ($)",
       title = str_glue("Symbol: {ticker}")) +
  scale_x_date(date_breaks = "1 year",
               date_labels = "%Y") +
  scale_y_continuous(limits = c(0, NA)) +
  theme_bw()
```



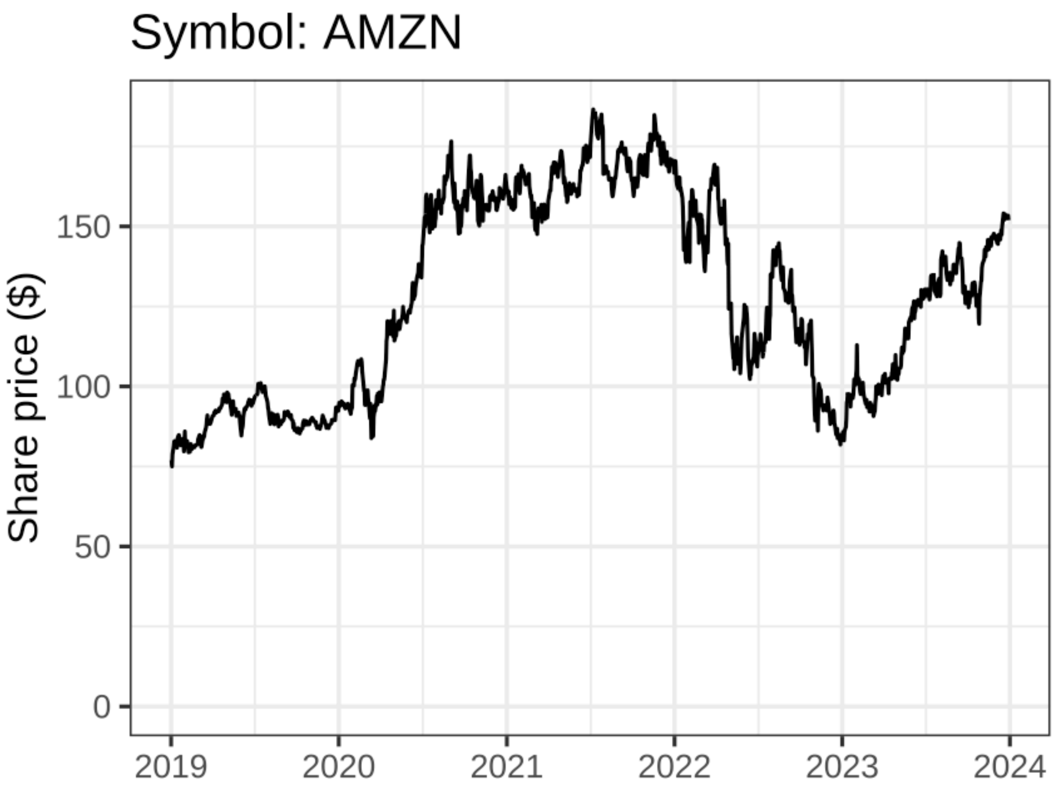
str_glue() allows us to put the contents of **ticker** in the plot's title

Step 1: Avoid hard-coding specific values

```
ticker <- "AMZN"

sp500_prices |>
  filter(symbol == ticker) |>
  ggplot(aes(x = date, y = adjusted)) +
  geom_line() +
  labs(x = NULL,
       y = "Share price ($)",
       title = str_glue("Symbol: {ticker}")) +
  scale_x_date(date_breaks = "1 year",
               date_labels = "%Y") +
  scale_y_continuous(limits = c(0, NA)) +
  theme_bw()
```

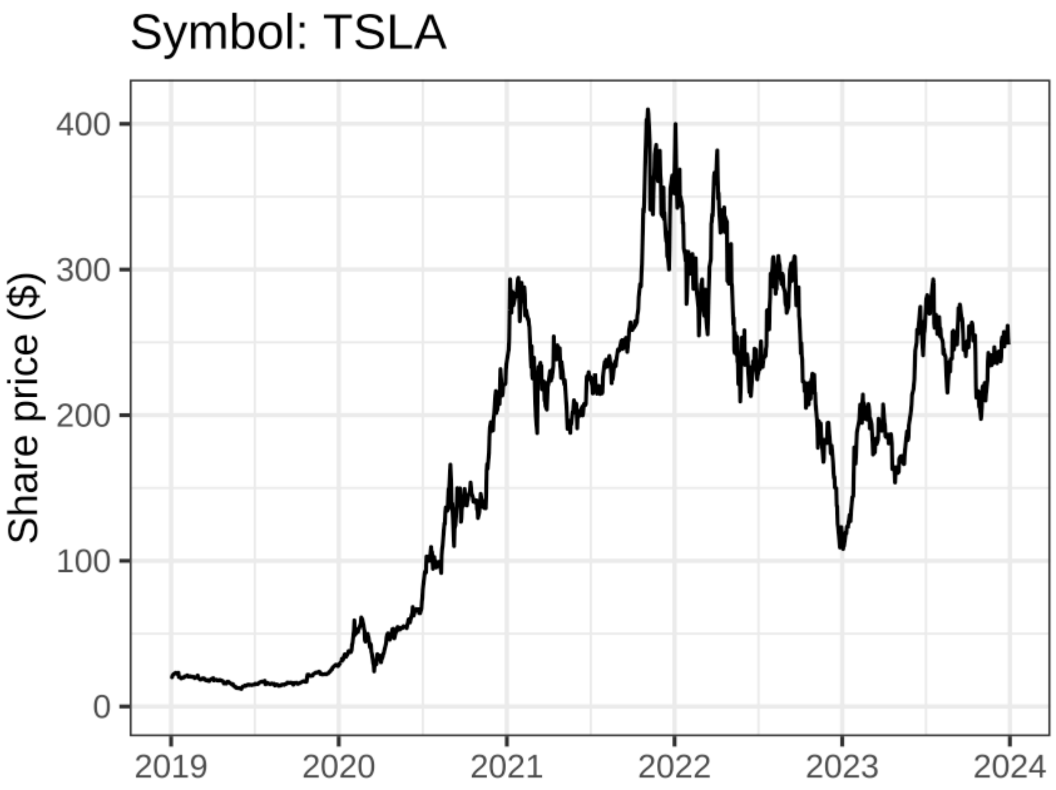
Now **ticker** is the only thing that changes



Step 1: Avoid hard-coding specific values

```r
ticker <- "TSLA"

sp500_prices |>
  filter(symbol == ticker) |>
  ggplot(aes(x = date, y = adjusted)) +
  geom_line() +
  labs(x = NULL,
       y = "Share price ($)",
       title = str_glue("Symbol: {ticker}")) +
  scale_x_date(date_breaks = "1 year",
               date_labels = "%Y") +
  scale_y_continuous(limits = c(0, NA)) +
  theme_bw()
```



Now **ticker** is the only thing that changes

Step 2: Define a function

```r
make_plot <- function(ticker) {
  sp500_prices |>
    filter(symbol == ticker) |>
    ggplot(aes(x = date, y = adjusted)) +
    geom_line() +
    labs(x = NULL,
         y = "Share price ($)",
         title = str_glue("Symbol: {ticker}"))
    scale_x_date(date_breaks = "1 year",
                 date_labels = "%Y") +
    scale_y_continuous(limits = c(0, NA)) +
    theme_bw()
}
```

Three key steps:

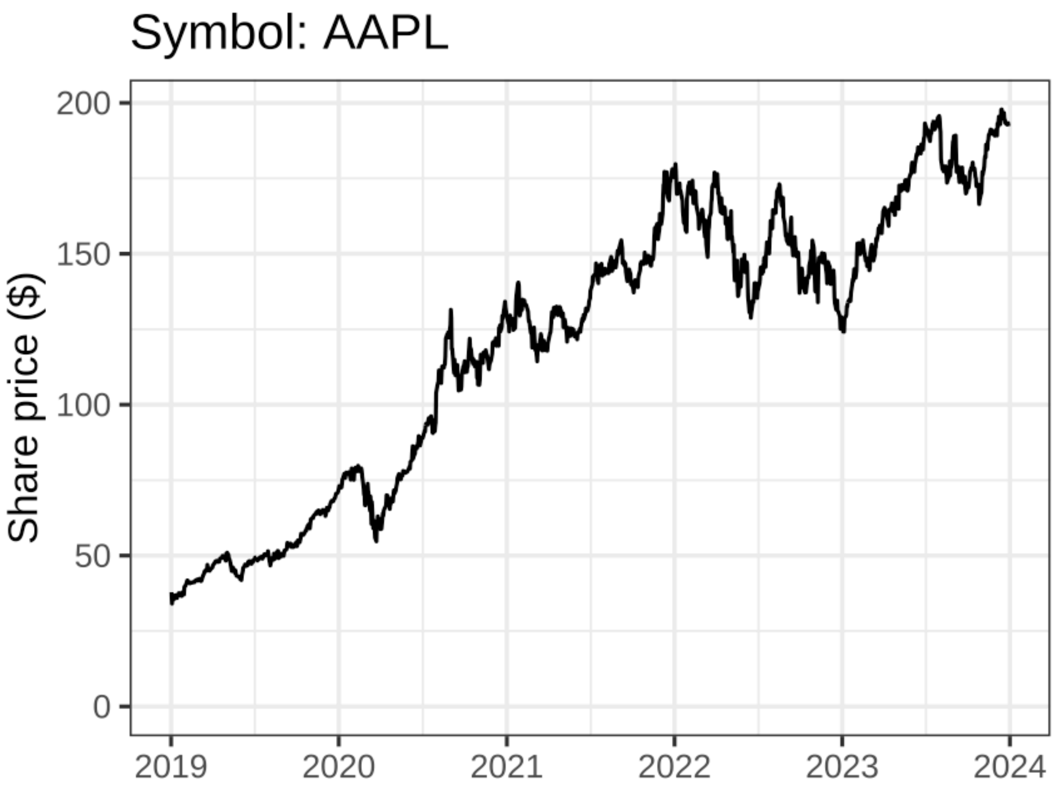
1. Pick a **name**
2. List **arguments** inside `function()`
3. Put code in the **body** of the function, delimited by `{...}`

Easiest to write the body on a test case, *then* convert it into a function

Step 2: Define a function

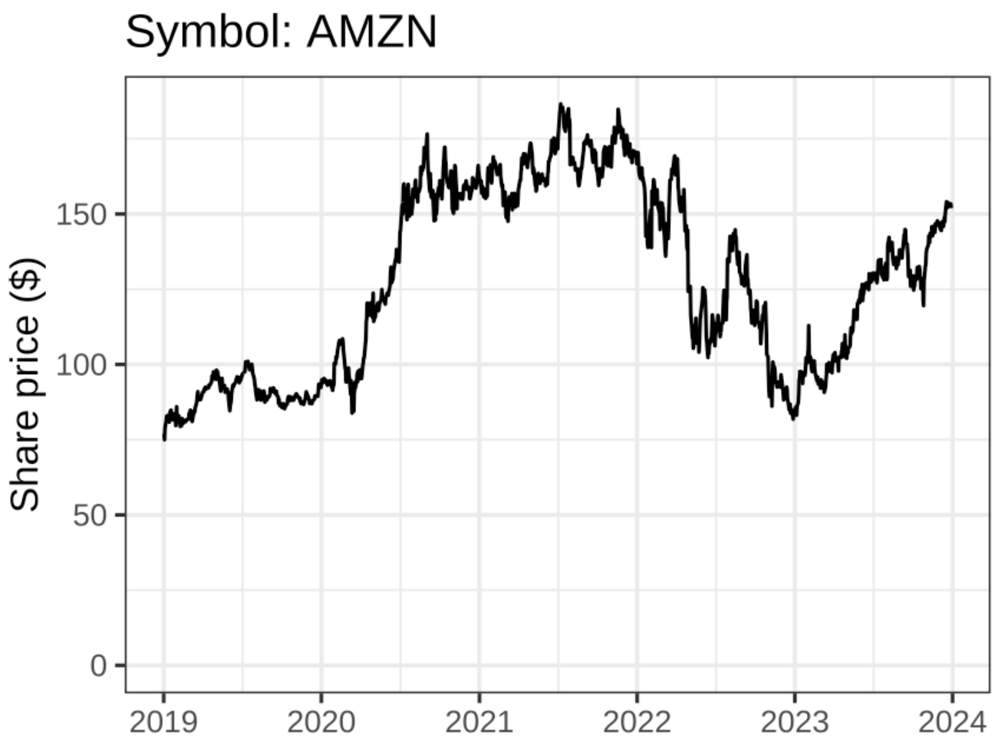
```r
make_plot <- function(ticker) {
  sp500_prices |>
    filter(symbol == ticker) |>
    ggplot(aes(x = date, y = adjusted)) +
    geom_line() +
    labs(x = NULL,
         y = "Share price ($)",
         title = str_glue("Symbol: {ticker}")) +
    scale_x_date(date_breaks = "1 year",
                 date_labels = "%Y") +
    scale_y_continuous(limits = c(0, NA)) +
    theme_bw()
}

make_plot("AAPL")
```

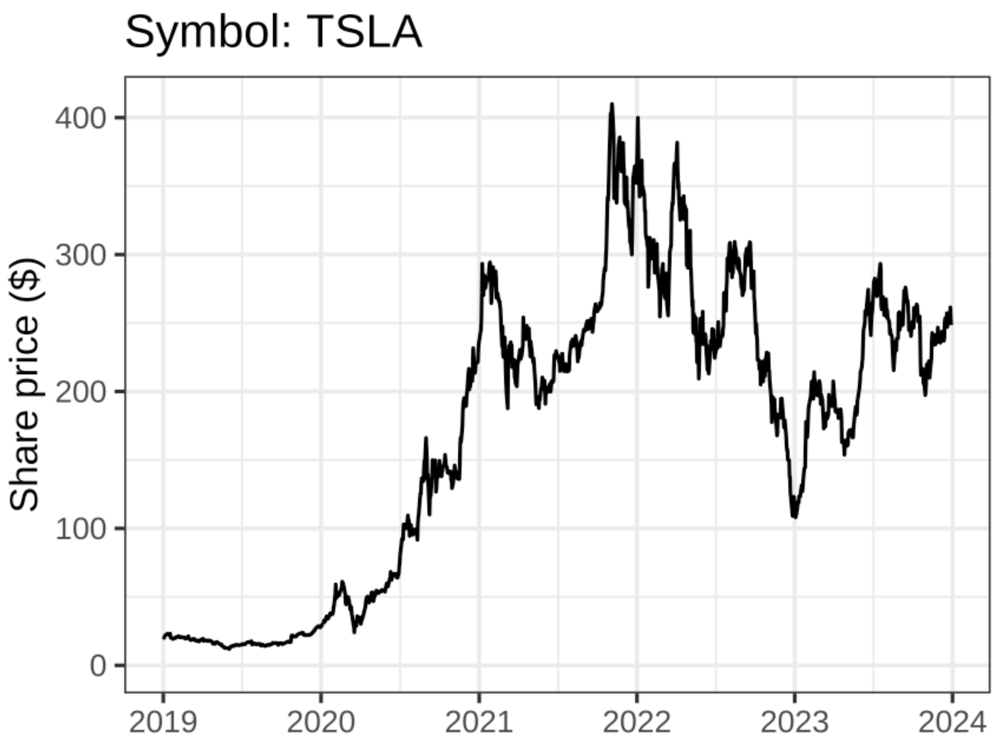


Step 2: Define a function

```r
make_plot("AMZN")
```



```r
make_plot("TSLA")
```



Rules of thumb about functions

- You can (almost) never write too many functions
- When you find yourself writing the same code 2-3 times, put it into a function
- A function should be no longer than 20-40 lines
- If a function is getting too long, break it into smaller functions

Step 3: Automate calling the function

Individual function calls are hard to scale

```
make_plot("AAPL")
make_plot("AMZN")
make_plot("TSLA")
```

What if we wanted to make this plot for every company in the S&P 500?

How could you automate these function calls?

1. Imperative programming (for loops)

2. Functional programming (map functions)

Step 3: Automate calling the function

The `purrr` packages provides `map` functions that take a vector as input, apply a **function** to each element of the vector, and return the results in a new vector:

```
map(some_vector, some_function)
```

- `map` functions are basically identical to base R's `apply` functions

How can we use map to make plots for AAPL, AMZN, and TSLA?

```
symbols <- c("AAPL", "AMZN", "TSLA")
plots <- map(symbols, make_plot)
```

Here `map` takes each element of the vector `symbols` and uses it as input for our function `make_plot()`

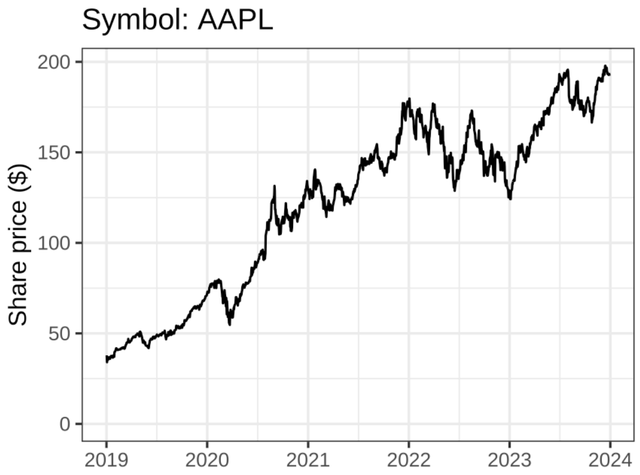
Step 3: Automate calling the function

`map` returns a **list**. In this example, it's a list of plots that we assigned to `plots`:

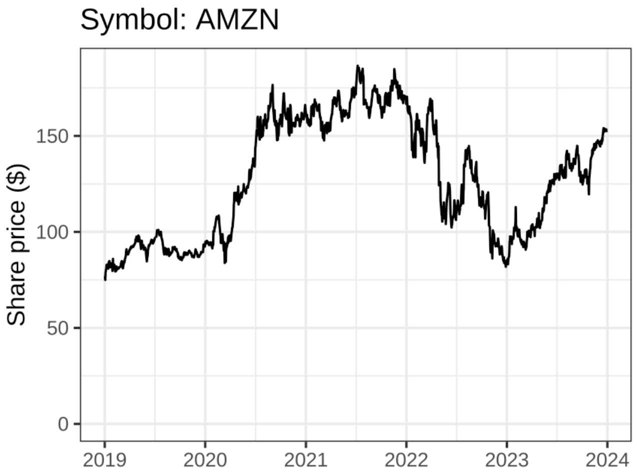
```
class(plots)
```

```
## [1] "list"
```

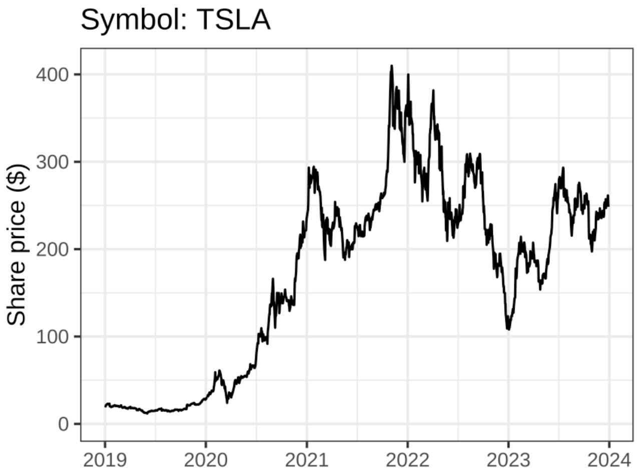
```
plots[[1]]
```



```
plots[[2]]
```



```
plots[[3]]
```



Step 3: Automate calling the function

This scales really easily!

```
all_symbols <- sp500_prices |> distinct(symbol) |> pull() # get all the symbols in the S&P 500
all_plots <- map(all_symbols, make_plot) # make a plot for each of the symbols
```

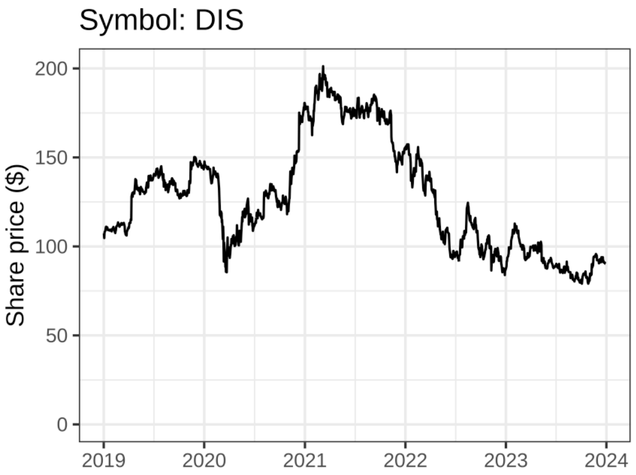
```
length(all_symbols)
```

```
## [1] 504
```

```
length(all_plots)
```

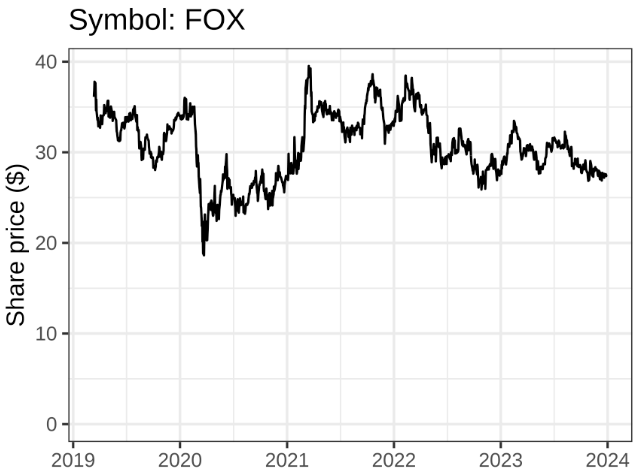
```
## [1] 504
```

```
all_plots[[35]]
```



Symbol: DIS

```
all_plots[[500]]
```



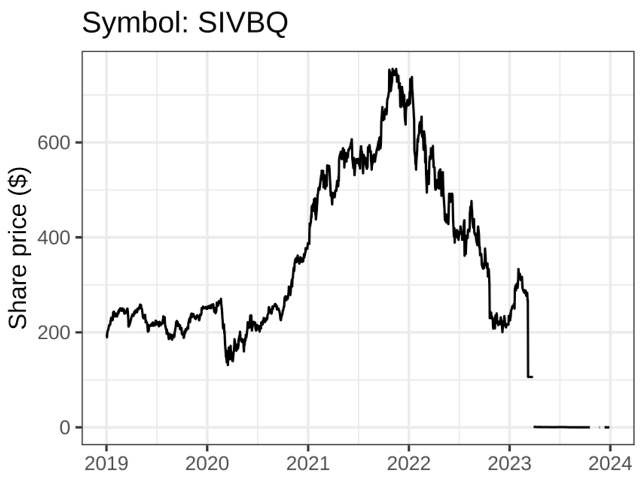
Symbol: FOX

Step 3: Automate calling the function

We can also extract results using logical expressions:

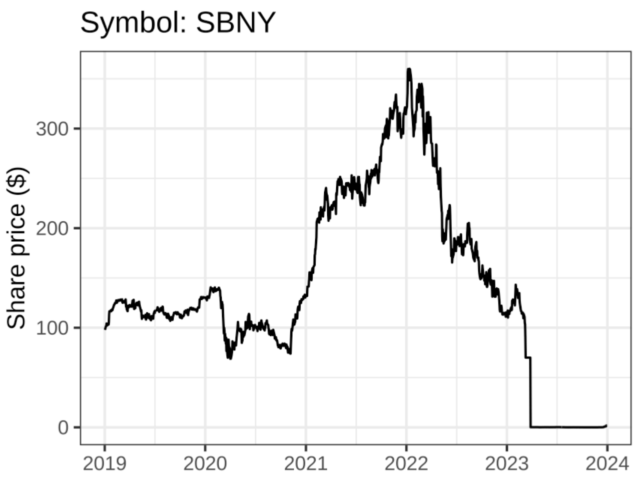
```
all_plots[all_symbols=="SIVBQ"
```

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



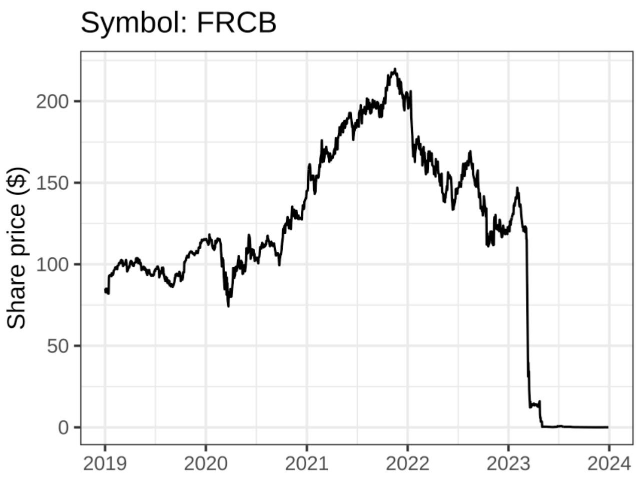
```
all_plots[all_symbols=="SBNY"
```

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



```
all_plots[all_symbols=="FRCB"
```

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



The map functions

The `purrr` package provides a family of `map` functions that return different types of output:

- `map()` makes a list
- `map_lgl()` makes a logical vector
- `map_int()` makes an integer vector
- `map_dbl()` makes a double vector
- `map_chr()` makes a character vector

What about `for` loops?

For loops work too!

```r
symbols <- c("AAPL", "AMZN", "TSLA")
plots <- vector("list", length(symbols)) # 1. allocate space for output
for (i in seq_along(symbols)) {          # 2. specify the sequence to loop over
  plots[[i]] <- make_plot(symbols[i])    # 3. specify what to do in each iteration
}
```

But functional programming is more concise:

```r
symbols <- c("AAPL", "AMZN", "TSLA")
plots <- map(symbols, make_plot)
```

Why not use `for` loops?

- They often require us to think about data logistics (indexing)
- They encourage iterative thinking over conceptual thinking
- Typically require more code, which often means more errors
- Can be harder to parallelize or otherwise optimize

But there is nothing wrong with using them!

example-14

Plan for today

Course evaluations

Prelim 2 info

lab-14

Please take 10 mins to complete course evals

I take feedback seriously and will use it to improve this course!

Concrete suggestions are most helpful

I would appreciate your feedback through two channels:

1. Lab 15 - Course Feedback Survey (on canvas through May 19)
2. University course evaluations (open through May 12)

Both will be anonymous

Reminder: **bonus point on Lab 15 for completing university evaluations**

Note: if you prefer to wait to complete the evaluations until after prelim 2, please feel free to do so. But please don't forget!

Prelim 2

Same format as prelim 1 (write answers and code in `.qmd`, submit `.qmd` + `.pdf`)

- **Covers everything since prelim 1** (weeks 7-14)
- *Tentative* plan is to have ~8 questions:
    - ~2 require no coding
        - "Describe X ways to improve this data visualization"
        - "Name Y ways one graphic is better/worse than another"
        - Tip: study the prologue sections of slides
    - ~1 is a mix of how to improve a dataviz, and then actually doing it
    - ~5 are coding questions where we specify the goal, and you get there
        - Some may have a sub-part where you describe an approach w/o code
- I will list the number of points per question to help you allocate effort

Questions?

lab-14