Functions and iteration

Week 12

AEM 2850: R for Business Analytics

Cornell Dyson

Spring 2022

Acknowledgements: Claus Wilke

Announcements

Mini project 1 graded, will discuss at end of class

Mini project 2 canceled

Final project details to come (due May 19 at 4:30pm)

Questions before we get started?

Plan for today

Prologue

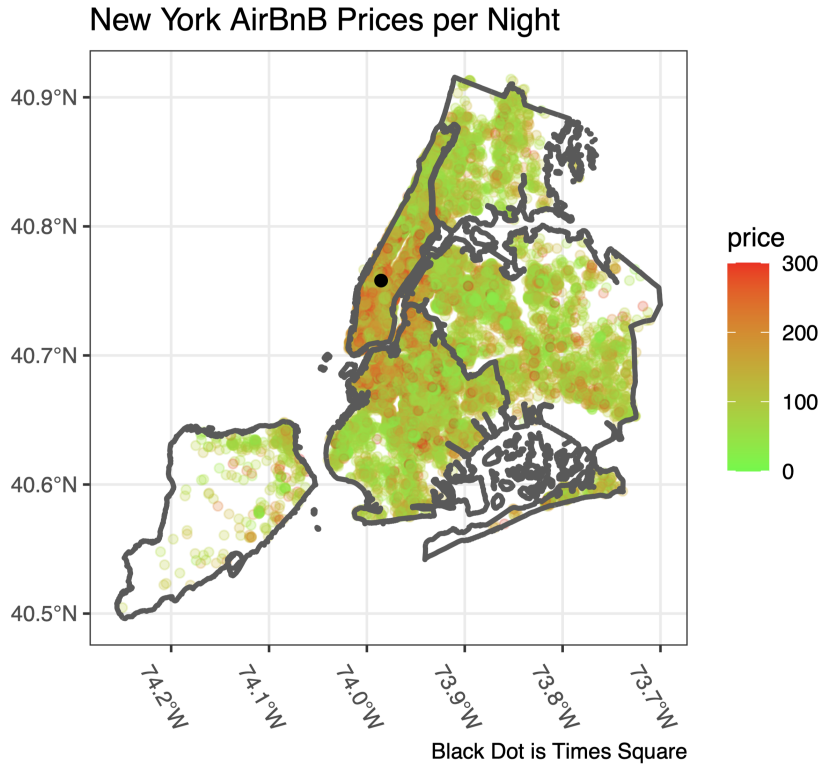
Functions and iteration

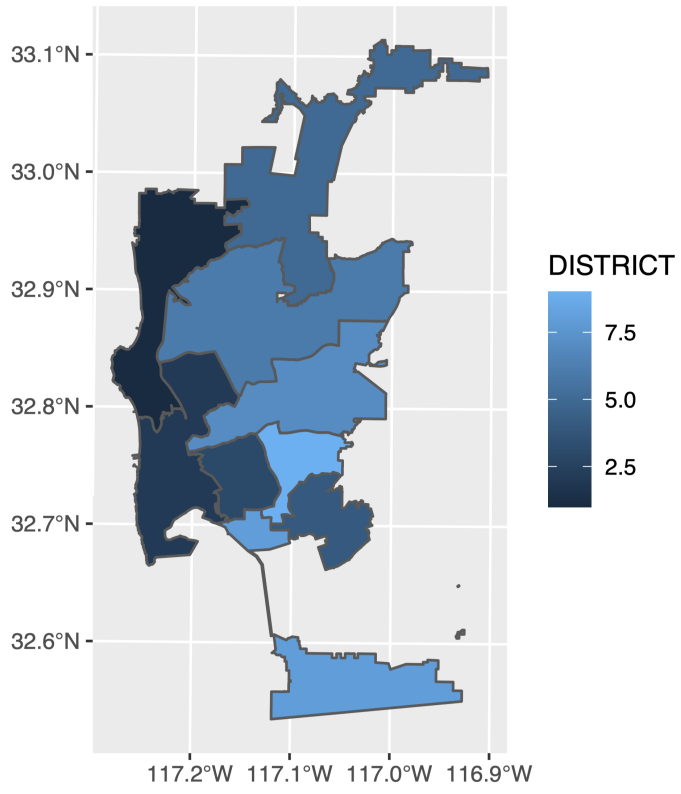
Example 12

Mini project 1

Prologue

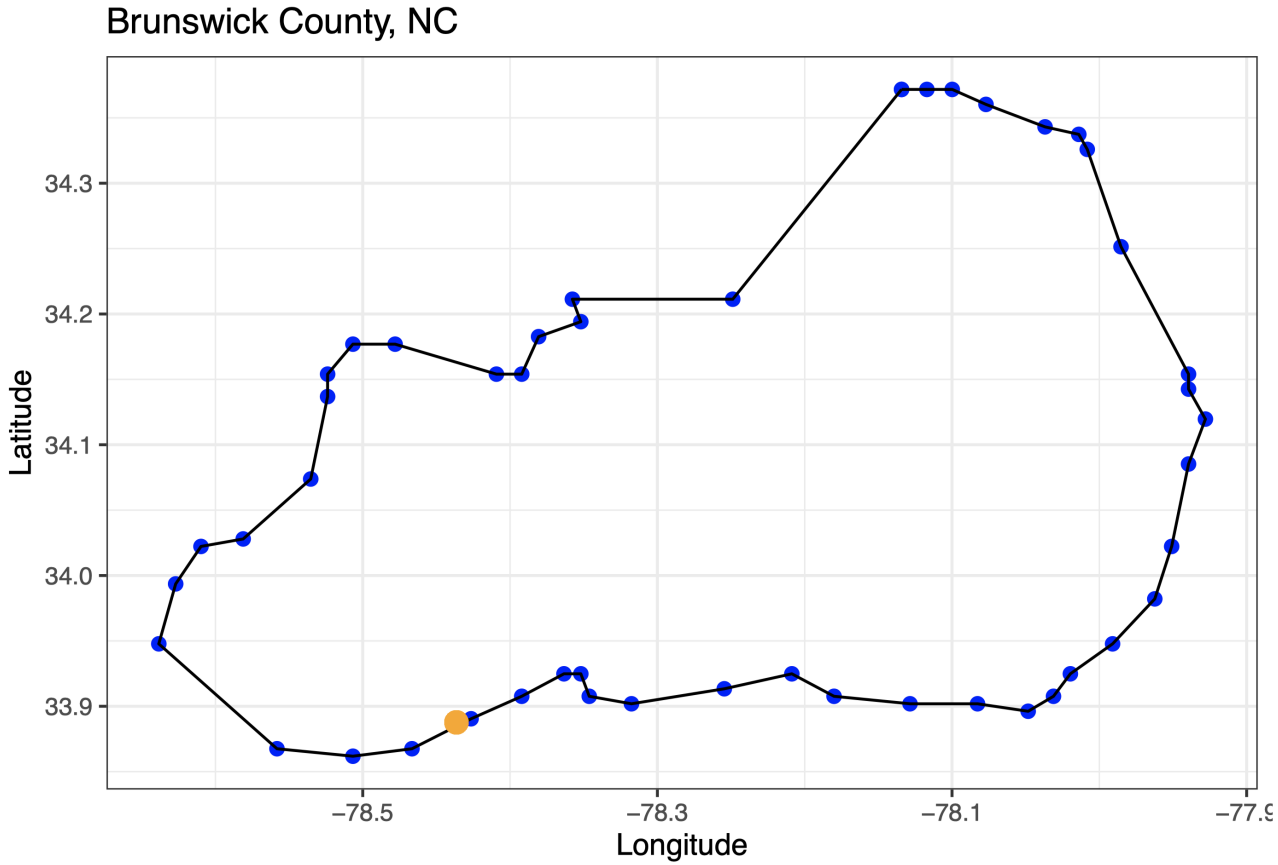
Creative answers to Lab 11, Question 8





"I went to San Diego this spring break"

Creative answers to Lab 11, Question 8

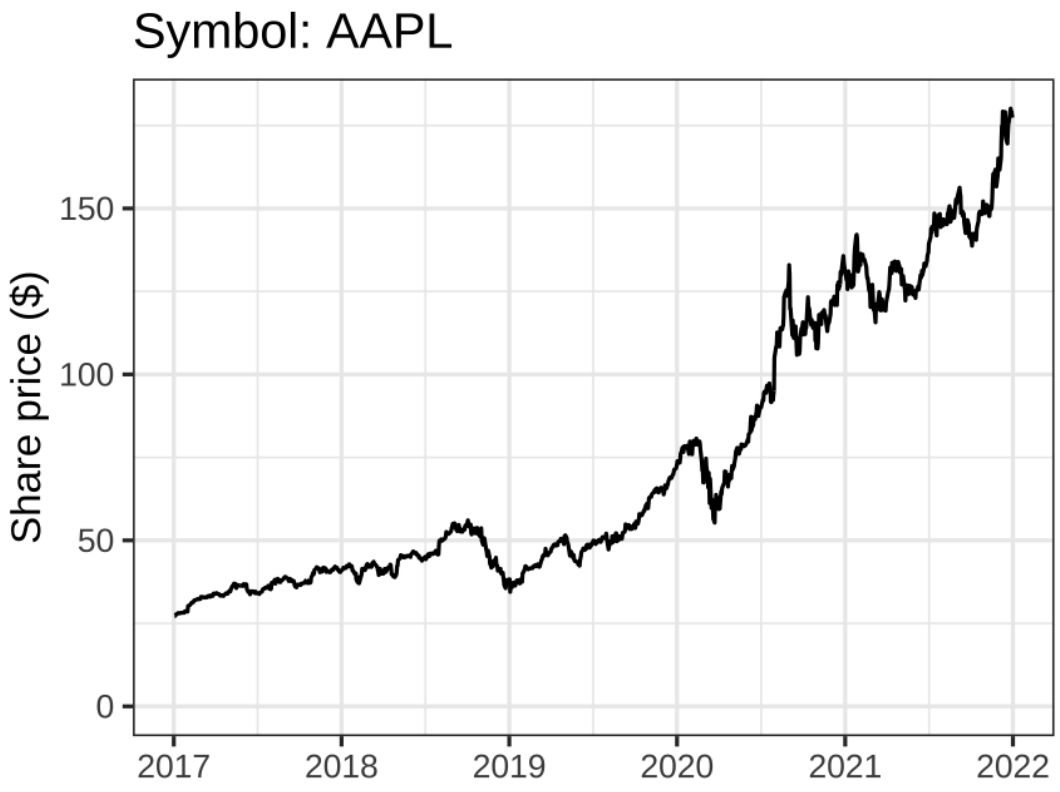


This is a approximate county map of Brunswick County, the southernmost county of North Carolina. The orange dot represents the coordinates of Ocean Isle Beach [33.8877° N, 78.4364° W], the destination of my team spring break.

Functions and iteration

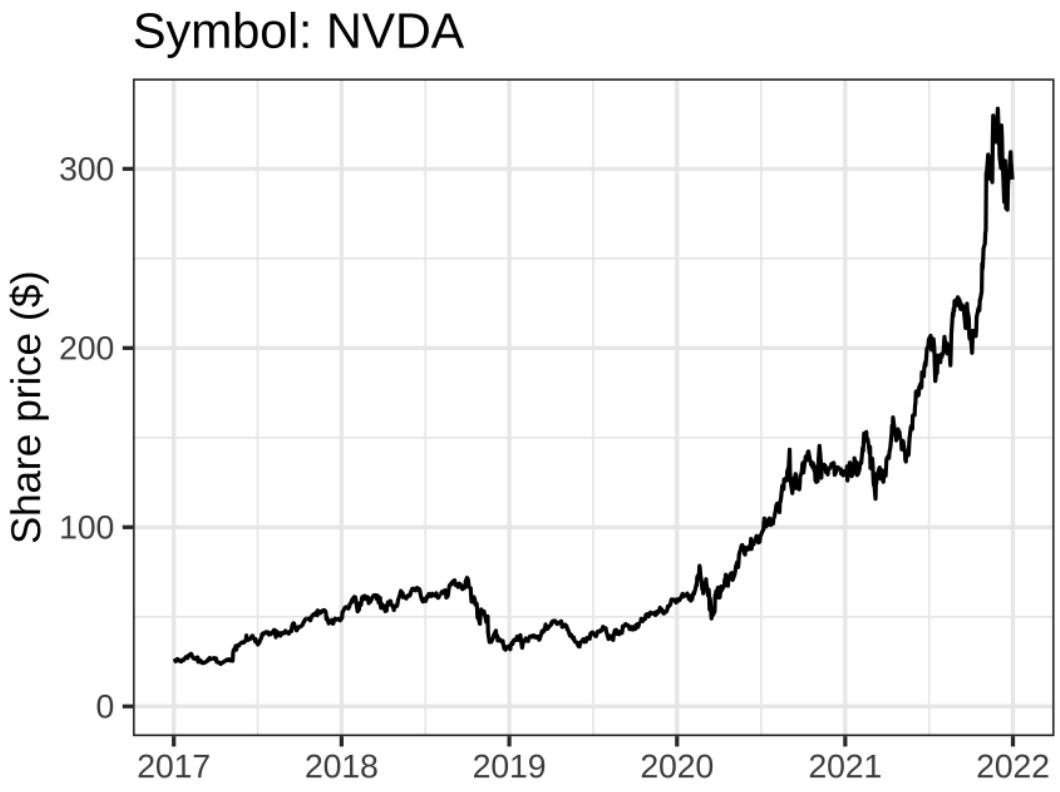
We often run similar code multiple times

```
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()
```



We often run similar code multiple times

```
sp500_prices %>%
  filter(symbol == "NVDA") %>%
  ggplot(aes(x = date, y = adjusted)) +
  geom_line() +
  labs(x = NULL,
       y = "Share price ($)",
       title = "Symbol: NVDA") +
  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

```
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

What is "hard-coded" here?

```
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()
```

Step 1: Avoid hard-coding specific values

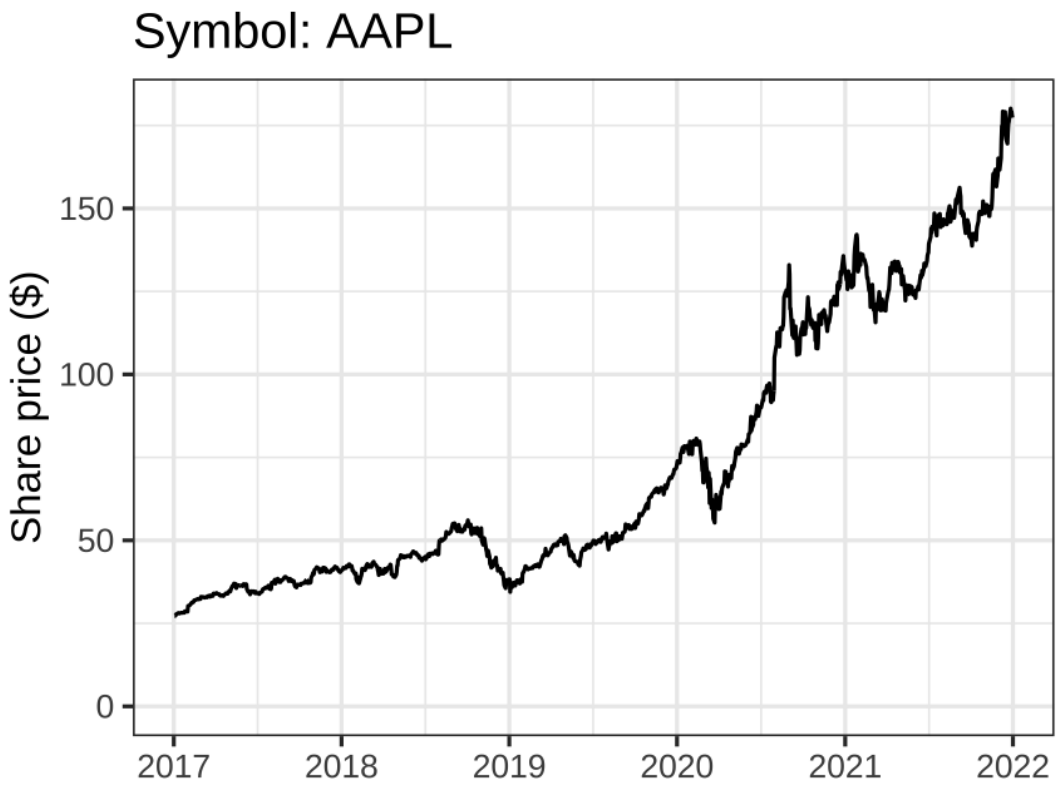
How can we avoid this hard-coding?

```
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()
```

Step 1: Avoid hard-coding specific values

```r
library(glue) # evaluate R code inside strings
symbol_choice <- "AAPL"

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

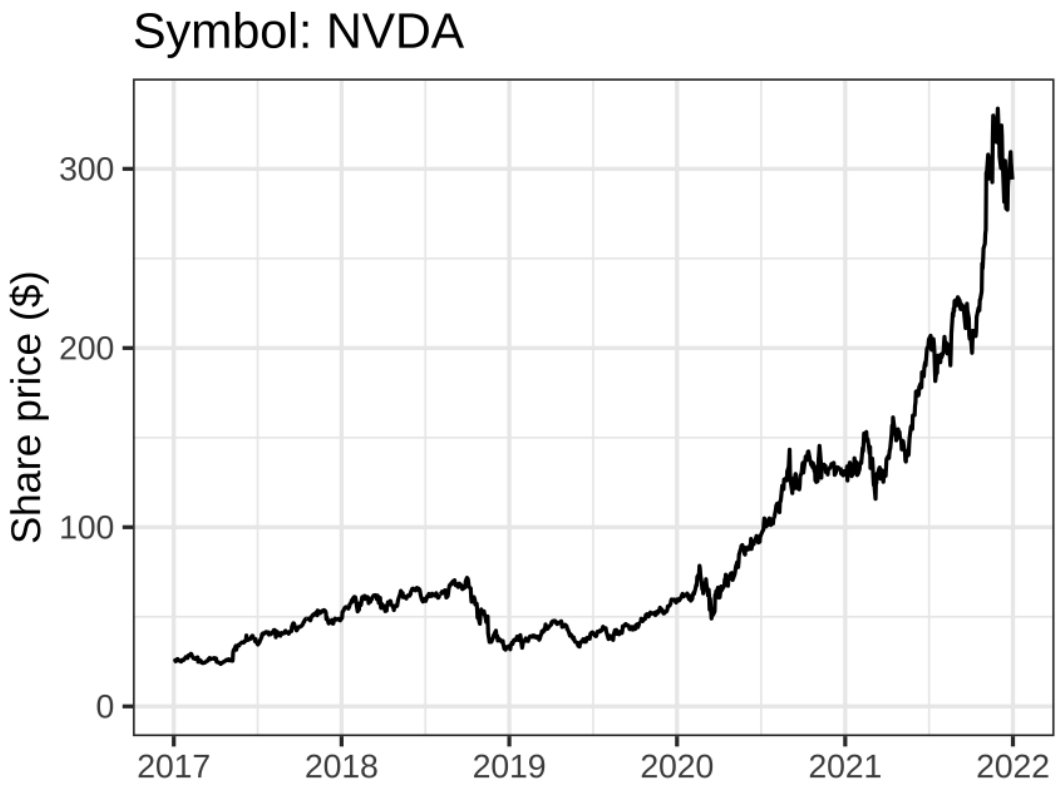


glue() allows us to put the contents of **symbol_choice** in the plot's title

Step 1: Avoid hard-coding specific values

```r
library(glue) # evaluate R code inside strings
symbol_choice <- "NVDA"

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



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

Step 1: Avoid hard-coding specific values

```r
library(glue) # evaluate R code inside strings
symbol_choice <- "TSLA"

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

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

Step 2: Define a function

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

Three key steps:

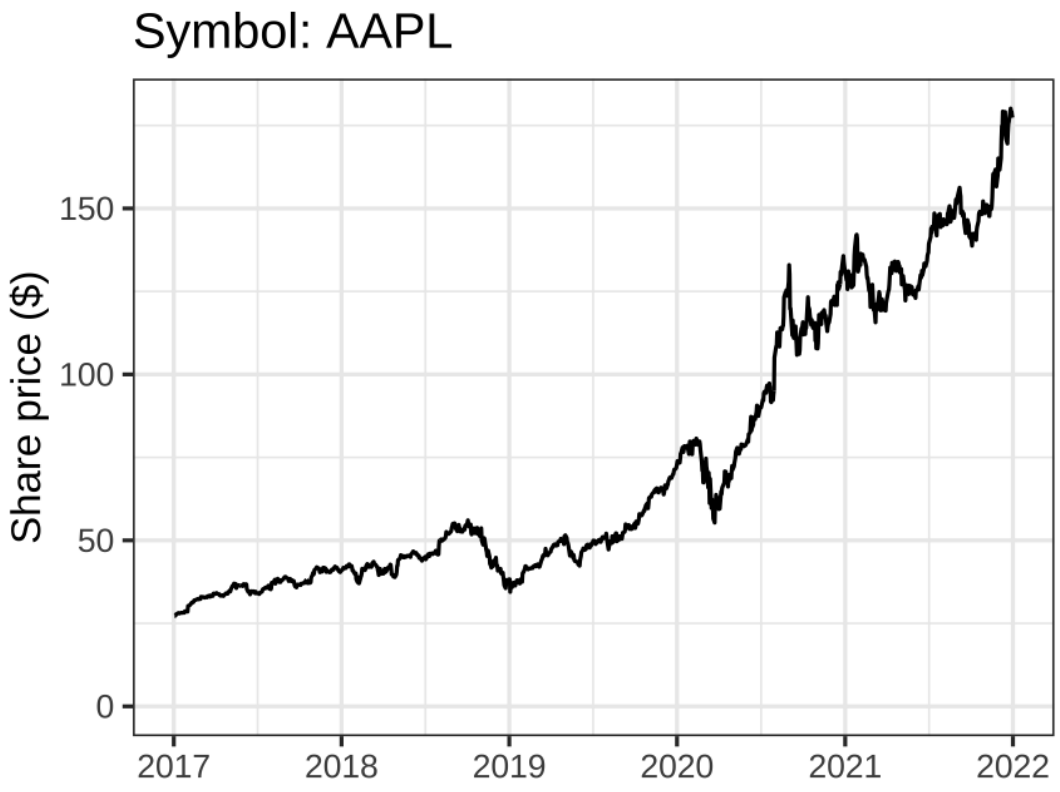
1. Pick a **name**
2. List **arguments** inside `function()`
3. Put code in the **body** of the function, delimited by `{...}`

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

Step 2: Define a function

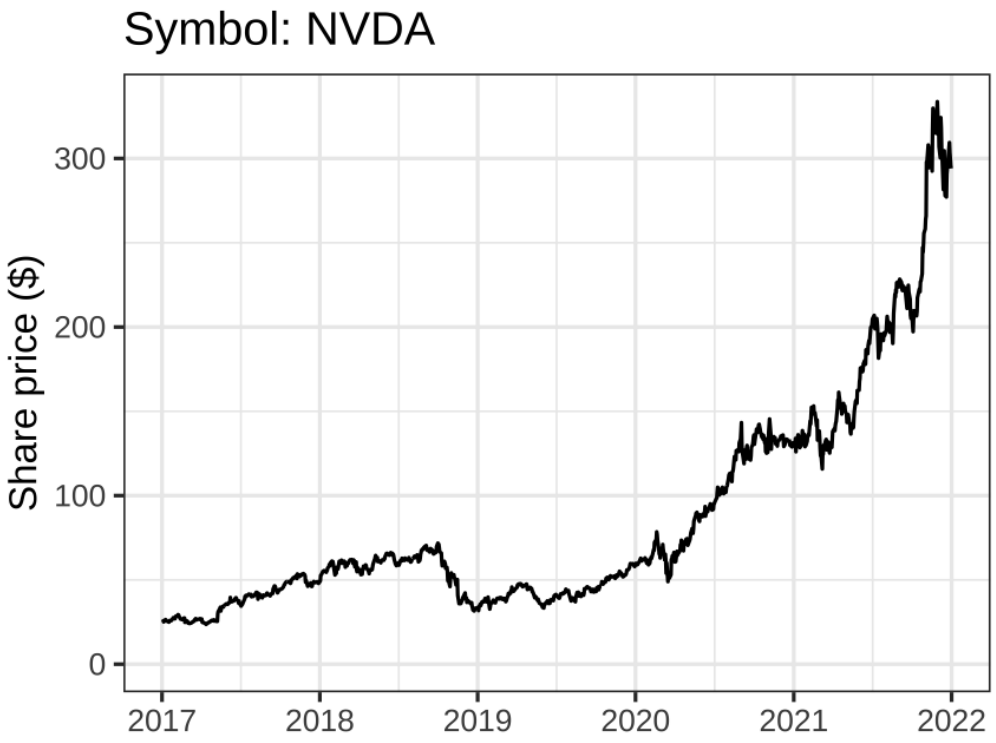
```
make_plot <- function(symbol_choice) {
  sp500_prices %>%
    filter(symbol == symbol_choice) %>%
    ggplot(aes(x = date, y = adjusted)) +
    geom_line() +
    labs(x = NULL,
         y = "Share price ($)",
         title = glue("Symbol: {symbol_choice}")) +
    scale_x_date(date_breaks = "1 year",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

make_plot("AAPL")
```



Step 2: Define a function

```
make_plot("NVDA")
```




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

Rules of thumb about functions

- You can 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("NVDA")
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, NVDA, and TSLA?

```
symbols <- c("AAPL", "NVDA", "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[[279]]
```


```
all_plots[[504]]
```

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!

```
symbols <- c("AAPL", "NVDA", "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:

```
symbols <- c("AAPL", "NVDA", "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
- Harder to parallelize or otherwise optimize

But there is nothing wrong with using them!

Example 12

Mini project 1

Overall feedback

Overall I was happy with everyone's work

This assignment was meant to push you, and it was cool to see everyone take different approaches to solve new problems

Grade summary

50 points total

- Median: 41
- Average: 39

These are raw group scores, don't reflect any adjustments for peer feedback yet

I will post these scores and annotated PDFs on canvas

Please contact me if you have questions, see errors, etc.

Specific feedback

Common challenges:

- Graph labels that were unclear, unreadable, or absent
- Incorrect calculations
  - Portfolio returns that exceed returns for every stock in the portfolio
- Not submitting code
- Overall presentation
  - No names on reports (!)
  - Including the Logistics and Expectations for my perusal

Keep these in mind for the final project!