Homework - Week 12

your name here

2025-04-16

Preface

The goal of this assignment is to help you gain familiarity with writing functions. As always, please come to office hours and reach out to your teaching staff if you have any questions.

Do not get put off by the number of pages in this homework! Roughly half of it is background material that we will go over in class.

Data

We will start by wrapping up the example from Tuesday using data from the S&P 500.

Then we will work with the monthly Zillow data from Zillow Research that we have seen before. As a reminder, `Metro_median_sale_price_uc_sfrcondo_month.csv` contains raw monthly median sale price data. Smoothed and seasonally adjusted price data is in `Metro_median_sale_price_uc_sfrcondo_sm_sa_month.csv`.

Background on Control Flow and Conditional Execution

We can use special statements to control the flow of a program based on whether a condition is met. We will focus on what is known as conditional execution. As a starting point, let's refresh our memory on what happens if we test a condition:

```
3 > 2
```

```
[1] TRUE
```

We can use `if` statements to use logical comparisons to control what our code does:

```
if (TRUE) {
  "You are seeing this message because the condition TRUE was met."
}
```

```
[1] "You are seeing this message because the condition TRUE was met."
```

We can extend the power of this conditional execution by combining `if` with `else if` and `else`. Here is one simple example:

```
my_condition <- TRUE
if (my_condition) {
  "The condition was met :)"
} else {
  "The condition was not met :("
}
```

```
[1] "The condition was met :)"
```

```
my_condition <- FALSE
if (my_condition) {
  "The condition was met :)"
} else {
  "The condition was not met :("
}
```

```
[1] "The condition was not met :("
```

Conditional Execution within Functions

Let's use a slightly more complicated example to demonstrate how conditional execution can be used to control what functions do:

```r
get_letter_grade <- function(x) {
  if (x >= 93) {
    "A"
  } else if (x >= 90) {
    "A-"
  } else if (x >= 87) {
    "B+"
  } else if (x >= 83) {
    "B"
  } else {
    "B- or lower"
  }
}
```

Here is what the function returns for several different inputs:

```r
get_letter_grade(93.5)
```

```
[1] "A"
```

```r
get_letter_grade(92.99)
```

```
[1] "A-"
```

```r
get_letter_grade(88.3)
```

```
[1] "B+"
```

```r
get_letter_grade(79)
```

```
[1] "B- or lower"
```

Part 1: S&P 500 Returns

During class on Tuesday, we had created a function that took a stock's ticker symbol as its argument. For valid tickers, it computed and returned annualized returns over the period April 2020 through April 2025. For invalid tickers, the code would fail. Here is that function:

```r
get_annual_return <- function(ticker){
  prices |>
    filter(symbol == ticker) |>
    filter(date==min(date) | date==max(date)) |>
    mutate(cumul_return = (adjusted - lag(adjusted)) / lag(adjusted) ) |>
    filter(!is.na(cumul_return)) |>
    mutate(annualized_return = ((1 + cumul_return)^(1/5) - 1) * 100) |>
    pull(annualized_return)
}
```

Here are some examples of what we got from calling that function:

```r
get_annual_return("AAPL")
```

```
[1] 23.25395
```

```r
map_dbl(
  c("AAPL", "AMZN", "TSLA"),
  get_annual_return
)
```

```
[1] 23.25395 10.11993 39.75486
```

```r
get_annual_return("DYSON")
```

```
Warning: There were 2 warnings in `filter()`.
The first warning was:
i In argument: `date == min(date) | date == max(date)`.
Caused by warning in `min.default()`:
! no non-missing arguments to min; returning Inf
i Run `dplyr::last_dplyr_warnings()` to see the 1 remaining warning.
```

```
numeric(0)
```

1. Revise our function using conditional execution to provide an informative message to users when they supply an invalid ticker. First, `pull` all the tickers in the `price` data frame and assign them to the name `symbols`. Then use that object in an `if` statement to either output the number, or to `print` the message "The ticker XXXX is not available." Finally, after printing the message, add `NA` so the function returns a missing value.

If you use `map_dbl` to apply `get_annual_return` to AAPL, AMZN, TSLA, and DYSON, will the code fail as it did in example-12-1? Why or why not?

The code does/does not fail because...

2. Modify your code from 1. to depend on two arguments, `df` and `ticker`, where `df` is a data frame to be used for computing annualized returns. The function should no longer depend directly on the `prices` data frame. Call the function with the ticker "XOM" to test it.

3. Modify your code from 2. to automatically detect the number of years rather than hard-coding the number of years as 5. There are several ways to do this. To keep things simple and focus on functions, just create a variable `years = time_length(date - lag(date), "years")` and use that variable in computing annualized returns.

Call the function with the data frame `prices_1y` and ticker "XOM" to compute annualized returns for Exxon Mobil over the course of 2024.

Call the function with the data frame `prices_ytd` and ticker "XOM" to compute annualized returns for Exxon Mobil over the course of 2025 so far (year to date).

Iteration using `for` Loops

As we saw in class, we can use functional programming methods like `map` to apply functions to a number of arguments. Another option is to use `for` loops to control the iteration process. See here for an example from the slides.

Here is a simple reminder of the basic mechanics of `for` loops:

```r
items <- c("item a", "item b", "item c", "item d")

# seq_along(items) generates the sequence 1, 2, ..., length(items)
seq_along(items)
```

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

```r
# loop through items, printing the index
for (i in seq_along(items)) {
  print(i)
}
```

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

```r
# loop through items, printing the contents of items
for (i in seq_along(items)) {
  print(items[i])
}
```

```
[1] "item a"
[1] "item b"
[1] "item c"
[1] "item d"
```

```r
# we used print because output within a loop will not print by default
for (i in seq_along(items)) {
  items[i]
}
```

4. Go back to our original function, copied below for convenience. Write a simple `for` loop to that calls `get_annual_return` and prints the annualized return for all the tickers in `biotech_symbols`. You do not need to populate an object as we did in the slides and with the `map` functions.

```r
# do not edit this code chunk
get_annual_return <- function(ticker){
  prices |>
    filter(symbol == ticker) |>
    filter(date==min(date) | date==max(date)) |>
    mutate(cumul_return = (adjusted - lag(adjusted)) / lag(adjusted) ) |>
    filter(!is.na(cumul_return)) |>
    mutate(annualized_return = ((1 + cumul_return)^(1/5) - 1) * 100) |>
    pull(annualized_return)
}

biotech_symbols
```

```
[1] "ABBV" "AMGN" "BIIB" "GILD" "INCY" "MRNA" "REGN" "VRTX"
```

```r
# write your for loop in this code chunk
```

Part 2: Zillow Data

Here is some code that imports and tidies Zillow sales price data in a few steps: (1) importing a file based on the string it starts with, (2) dropping the observations for the region of "United States", (3) separating RegionName into two variables City and State, (4) pivoting the data longer, and (5) converting date into date format and generating the variables year and month.

```r
raw_tidy_example <- "Metro_median_sale_price_uc_sfrcondo_month.csv" |>
  read_csv() |>
  filter(RegionName != "United States") |>
  separate(RegionName, c("City","State"), sep =",") |>
  pivot_longer(-c(RegionID, SizeRank, City, State, RegionType, StateName),
               names_to = "date", values_to = "price") |>
  mutate(date = ymd(date), year = year(date), month = month(date))
```

5. Use the code on the previous page to write a function `tidy_data` that takes the argument `filename` and returns the tidied data. Call this function once to tidy the raw price data and once to tidy the adjusted price data, assigning the output to `raw_tidy` and `adj_tidy`. What years do `raw_tidy` and `adj_tidy` cover, respectively?

6. Can you modify the function in question 1 so that it can be used to tidy either price data or sales data, based on a second argument that specifies which one the user is asking for? What years do the sales data cover?

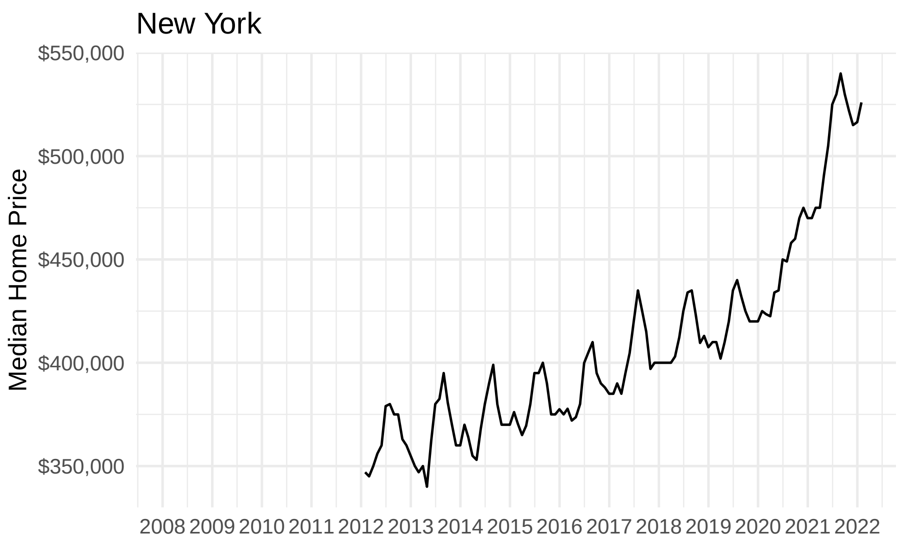
Note: `Metro_sales_count_now_uc_sfrcondo_month.csv` *contains monthly sales counts.*

Note: an alternative way to approach this would be to detect whether the data are price or sales based on the filenames. We could also start by checking to make sure the file exists, and returning an error if not. See the solutions for an example.

Plotting Zillow Data

Here is some code that uses the data we just tidied to plot the time series of raw prices over time for one city:

```r
raw_tidy_example |>
  filter(City == "New York") |>
  ggplot(aes(x = date, y = price)) +
  geom_line() +
  scale_x_date(
    breaks = scales::breaks_width("1 year"),
    labels = scales::label_date("%Y")
  )  +
  scale_y_continuous(labels=scales::dollar_format()) +
  labs(
    x = "",
    y = "Median Home Price",
    title = "New York"
  ) +
  theme_minimal()
```



7. Adapt the code above to create a function `plot_city_price` that plots the time series of raw prices over time for whatever city is provided as its argument. Call this function to make a plot for `"Los Angeles-Long Beach-Anaheim"`.

8. Uncomment the code below to see how we can use `map()` in conjunction with `walk(print)` to print the output of map without returning the objects themselves and their pesky indices.

```r
# c("New York", "Los Angeles-Long Beach-Anaheim", "Chicago") |>
#   map(plot_city_price) |>
#   walk(print)
```