```
library(tidyverse)
library(riingo)
library(almanac)
library(lubridate)
library(roll)
library(plotly)
```

Next, we'll create a tibble of ETF tickers and sectors. I'll stick with my usual iShares funds and SPY for the overall market.

```
etf_ticker_sector <- tibble(
  ticker = c("XLY", "XLP", "XLE",
           "XLF", "XLV", "XLI", "XLB",
           "XLK", "XLU", "XLRE",
           "SPY"),
  sector = c("Consumer Discretionary", "Consumer Staples", "Energy",
           "Financials", "Health Care", "Industrials", "Materials",
           "Information Technology", "Utilities", "Real Estate",
           "Market")
)

etf_ticker_sector
```

```
# A tibble: 11 x 2
   ticker sector

 1 XLY    Consumer Discretionary
 2 XLP    Consumer Staples
 3 XLE    Energy
 4 XLF    Financials
 5 XLV    Health Care
 6 XLI    Industrials
 7 XLB    Materials
 8 XLK    Information Technology
 9 XLU    Utilities
10 XLRE   Real Estate
11 SPY    Market
```

We are going to use `tiingo` via the `riingo` package to import daily prices for these tickers. That means we need an API key and then we need to pass our tickers to `riingo_prices()`.

```
# Need an API key for tiingo

riingo_set_token("your API key here")

etf_prices <-
  etf_ticker_sector %>%
  pull(ticker) %>%
  riingo_prices(.,
                start_date = "2010-01-01",
                end_date = Sys.Date()) %>%
  mutate(date = ymd(date))

etf_prices %>%
  group_by(ticker) %>%
  slice(1:3)
```

```
# A tibble: 33 x 14
```

```
# Groups:   ticker [11]
   ticker date       close  high   low  open volume adjClose adjHigh adjLow

 1 SPY    2010-01-04 113.  113.  112.  112.  1.19e8     92.8    92.8  91.3
 2 SPY    2010-01-05 114.  114.  113.  113.  1.12e8     93.0    93.1  92.4
 3 SPY    2010-01-06 114.  114.  113.  114.  1.16e8     93.1    93.3  92.9
 4 XLB    2010-01-04  34.0  34.0  33.4  33.6 7.57e6     27.2    27.2  26.8
 5 XLB    2010-01-05  34.1  34.2  33.8  34.1 8.84e6     27.3    27.4  27.1
 6 XLB    2010-01-06  34.7  34.8  34.1  34.1 8.09e6     27.8    27.9  27.3
 7 XLE    2010-01-04  58.8  58.8  57.8  57.9 1.69e7     44.8    44.8  44.0
 8 XLE    2010-01-05  59.3  59.4  58.6  58.9 1.74e7     45.2    45.2  44.6
 9 XLE    2010-01-06  60    60.2  59.2  59.3 2.43e7     45.7    45.8  45.1
10 XLF    2010-01-04  14.7  14.7  14.5  14.5 7.55e7     10.0    10.1   9.91
# … with 23 more rows, and 4 more variables: adjOpen ,
#   adjVolume , divCash , splitFactor
```

Let's calculate rolling 50-day and 200-day average for SPY using the blazing fast `roll` package for rolling calculations. It has a built-in rolling mean function called `roll_mean` and is running C under the hood. Did I mention it's fast? It's super fast.

```
  etf_prices %>%
  group_by(ticker) %>%
  filter(ticker == "SPY") %>%
  mutate(sma_50 = roll_mean(as.matrix(close), width = 50, complete_obs = T),
         sma_200 = roll_mean(as.matrix(close), width = 200, complete_obs = T))
%>%
  select(date, sma_50, sma_200) %>%
  filter(!is.na(sma_200)) %>%
  head()
```

```
# A tibble: 6 x 4
# Groups:   ticker [1]
  ticker date       sma_50 sma_200

1 SPY    2010-10-18   112.    112.
2 SPY    2010-10-19   112.    112.
3 SPY    2010-10-20   112.    112.
4 SPY    2010-10-21   112.    112.
5 SPY    2010-10-22   112.    112.
6 SPY    2010-10-25   113.    112.
```

Notice that we are filtering to show just "SPY" for now. When we get to shiny we'll replace `filter(ticker == "SPY")` with `filter(ticker == input$ticker)` – that will allow an end user run these moving average calculations on any ticker we have in our data set.

Let's plot the moving averages and use different colors and line types for the price, 50-day moving average and 200-day moving average.

```
rolling_average_plot <-
etf_prices %>%
  group_by(ticker) %>%
  filter(ticker == "SPY") %>%
  mutate(sma_50 = roll_mean(as.matrix(close), width = 50, complete_obs = T),
         sma_200 = roll_mean(as.matrix(close), width = 200, complete_obs = T))
%>%
  select(date, close, sma_50, sma_200) %>%
  ggplot(aes(x = date)) +
  geom_line(aes(y = close), color = "purple", linetype = "dotted") +
  geom_line(aes(y = sma_50), color = "cornflowerblue", linetype = "solid") +
```

```
  geom_line(aes(y = sma_200), color = "green", linetype = "solid") +
  scale_x_date(breaks = scales::pretty_breaks(n = 10)) +
  scale_y_continuous(label = scales::dollar) +
  labs(title = "Price, SMA 50 and SMA 200", y = "", x = "") +
  theme_minimal() +
  theme(plot.title  = element_text(hjust = .5))

ggplotly(rolling_average_plot)
```

2010201120122013201420152016201720182019202020$100$150$200$250$300

Price, SMA 50 and SMA 200

How have volumes behaved throughout this time period?

```
volume_plot <-
ggplotly(
etf_prices %>%
  group_by(ticker) %>%
  filter(ticker == "SPY") %>%
  ggplot(aes(x = date)) +
  geom_col(aes(y = volume), color = "pink", alpha = .5) +
  scale_x_date(breaks = scales::pretty_breaks(n = 10)) +
  scale_y_continuous(labels = scales::number_format(accuracy = 1,
                                    scale = 1/1000000,
                                    suffix = "M")) +
  theme_minimal() +
  theme(plot.title = element_text(hjust = .5)) +
  labs(y = "", title = "Daily Volume")
)

ggplotly(volume_plot)
```

2010201120122013201420152016201720182019202000M200M400M600M

Daily Volumedate

Nothing crazy so far, how can the `almanac` package help us get creative?

I'm curious if BLS reports or holidays affect volumes and want to add a flag to the charts on the day of BLS reports and the day *after* any market holidays. That would have been very painful before the arrival of `almanac`!

Let's start with the BLS.

The BLS reports on the employment situation on the first Friday of each month *unless* that first Friday is a national holiday.

Here's how to quickly print the first Friday of each month since January 2010, using `recur_on_wday("Friday", nth = 1)`.

```
monthly(since = "2010-01-01") %>%
  recur_on_interval(1) %>%
  recur_on_wday("Friday", nth = 1) %>%
  alma_seq("2010-01-01", "2020-03-01", .)
```

```
  [1] "2010-01-01" "2010-02-05" "2010-03-05" "2010-04-02" "2010-05-07"
  [6] "2010-06-04" "2010-07-02" "2010-08-06" "2010-09-03" "2010-10-01"
 [11] "2010-11-05" "2010-12-03" "2011-01-07" "2011-02-04" "2011-03-04"
 [16] "2011-04-01" "2011-05-06" "2011-06-03" "2011-07-01" "2011-08-05"
 [21] "2011-09-02" "2011-10-07" "2011-11-04" "2011-12-02" "2012-01-06"
```

```
 [26] "2012-02-03" "2012-03-02" "2012-04-06" "2012-05-04" "2012-06-01"
 [31] "2012-07-06" "2012-08-03" "2012-09-07" "2012-10-05" "2012-11-02"
 [36] "2012-12-07" "2013-01-04" "2013-02-01" "2013-03-01" "2013-04-05"
 [41] "2013-05-03" "2013-06-07" "2013-07-05" "2013-08-02" "2013-09-06"
 [46] "2013-10-04" "2013-11-01" "2013-12-06" "2014-01-03" "2014-02-07"
 [51] "2014-03-07" "2014-04-04" "2014-05-02" "2014-06-06" "2014-07-04"
 [56] "2014-08-01" "2014-09-05" "2014-10-03" "2014-11-07" "2014-12-05"
 [61] "2015-01-02" "2015-02-06" "2015-03-06" "2015-04-03" "2015-05-01"
 [66] "2015-06-05" "2015-07-03" "2015-08-07" "2015-09-04" "2015-10-02"
 [71] "2015-11-06" "2015-12-04" "2016-01-01" "2016-02-05" "2016-03-04"
 [76] "2016-04-01" "2016-05-06" "2016-06-03" "2016-07-01" "2016-08-05"
 [81] "2016-09-02" "2016-10-07" "2016-11-04" "2016-12-02" "2017-01-06"
 [86] "2017-02-03" "2017-03-03" "2017-04-07" "2017-05-05" "2017-06-02"
 [91] "2017-07-07" "2017-08-04" "2017-09-01" "2017-10-06" "2017-11-03"
 [96] "2017-12-01" "2018-01-05" "2018-02-02" "2018-03-02" "2018-04-06"
[101] "2018-05-04" "2018-06-01" "2018-07-06" "2018-08-03" "2018-09-07"
[106] "2018-10-05" "2018-11-02" "2018-12-07" "2019-01-04" "2019-02-01"
[111] "2019-03-01" "2019-04-05" "2019-05-03" "2019-06-07" "2019-07-05"
[116] "2019-08-02" "2019-09-06" "2019-10-04" "2019-11-01" "2019-12-06"
[121] "2020-01-03" "2020-02-07"
```

Are any of these national holidays?

Well, the first Friday at some point probably fell on January 1st or July 4th. We can hunt for those dates using `str_subset()` to keep only the subset of our date strings containing either “01-01” (for January 1) or “07-04” (for July 4).

```
monthly(since = "2010-01-01") %>%
  recur_on_interval(1) %>%
  recur_on_wday("Friday", nth = 1) %>%
  alma_seq("2010-01-01", "2020-03-01", .) %>%
  str_subset(paste(c("01-01", "07-04"), collapse = '|'))
```

```
[1] "2010-01-01" "2014-07-04" "2016-01-01"
```

It looks like we have three problematic dates to handle. Now we must make a provision for these dates, but the provision is different for each scenario. For July 4th, the BLS reports the day before, July 3rd, a Thursday. For January 1st, the BLS doesn’t shift to December 31st because that would be shifting into the previous year. Instead it shifts to the following Friday, January 8th! Encoding that logic would be quite cumbersome, indeed, and here the magic of `almanac` saves us some time.

The package contains a prebuilt object called `hldy_independence_day()` and another called `hldy_new_years_day()`. We can create a calendar that skips ahead seven days whenever it sees January 1st, and steps back 1 day whenever it sees July 4th in our schedule.

We do that with `alma_adjust(hldy_new_years_day(), adjustment = 7)`, to skip 7 days from New Year’s Day and then `alma_adjust(hldy_independence_day(), adjustment = -1)` to move back from July 4th.

Here’s what it does to just the Independence Day and New Year’s Days we wish to adjust.

```
monthly(since = "2010-01-01") %>%
  recur_on_interval(1) %>%
  recur_on_wday("Friday", nth = 1) %>%
  alma_seq("2010-01-01", "2020-03-01", .) %>%
  str_subset(paste(c("01-01", "07-04"),collapse = '|')) %>%
  alma_adjust(hldy_new_years_day(), adjustment = 7) %>%
  alma_adjust(hldy_independence_day(), adjustment = -1)
```

```
[1] "2010-01-08" "2014-07-03" "2016-01-08"
```

Let's add our adjusters to the original schedule creation flow and save as an object called `bls_reports`. I eventually want to access this as a data frame so will convert to a tibble as the final step.

```
bls_reports <-
monthly(since = "2010-01-01") %>%
  recur_on_interval(1) %>%
  recur_on_wday("Friday", nth = 1) %>%
  alma_seq("2010-01-01", "2020-03-01", .) %>%
  alma_adjust(hldy_new_years_day(), adjustment = 7) %>%
  alma_adjust(hldy_independence_day(), adjustment = -1) %>%
  tibble(nfp_release_dates = .,
         bls_flag = 1)
```

We also want to investigate volume on the day after a market holiday. To create a list of the days after market holidays, we'll start with `calendar_usa_federal()` and then use `alma_jump(days(1))` to move to the following day. But, we need to make sure we don't jump off of a holiday and onto a weekend. Let's create a list of weekends with `on_weekends <- weekly(since = "2010-01-01") %>% recur_on_weekends()`, and then add it to `sq_jump` so that we move to the day following the holiday and then jump over any weekends. The full call below will look for holidays, then skip one day, then check to make sure we haven't landed on a weekend and if so, move to Monday.

```
on_weekends <- weekly(since = "2010-01-01") %>%
  recur_on_weekends()

post_holiday <-
alma_seq("2010-01-01", "2019-10-01", calendar_usa_federal())  %>%
alma_jump(days(1), on_weekends)

day_after <-
tibble(
  post_holiday = post_holiday,
  holiday_flag = 1
)
```

Now we can add those date flags to our etf data with calls to `left_join(bls_reports)` and `left_join(day_after, by = c("date" = "post_holiday"))`

```
etf_prices_date_flags <-
etf_prices %>%
  group_by(ticker) %>%
  mutate(sma_50 = roll_mean(as.matrix(close), width = 50, complete_obs = T),
         sma_200 = roll_mean(as.matrix(close), width = 200, complete_obs = T))
%>%
  filter(!is.na(sma_200)) %>%
  select(ticker, date, close, volume, sma_50, sma_200) %>%
  left_join(bls_reports, by = c("date" = "nfp_release_dates")) %>%
  left_join(day_after, by = c("date" = "post_holiday"))
```

Let's save that as an RDS object for use in a Shiny app later.

```
write_rds(etf_prices_date_flags, "etf_prices_date_flags.RDS")
```

Before building the app, let's plot the daily volumes of `SPY` as pink bars, adding an orange dot for BLS release dates and a black dot for the day after market holidays.

```
volume_plot_flags <-
(
etf_prices_date_flags %>%
  filter(ticker ==  "SPY") %>%
  ggplot(aes(x = date)) +
```

```
  geom_col(aes(y = volume), color = "pink", alpha = .5) +
  geom_point(data = etf_prices_date_flags %>% filter(ticker == "SPY" & bls_flag
== 1), aes(date, volume, text = paste("Jobs Report", "
date:", date, "
 volume:", volume)), color = "orange") +
  geom_point(data = etf_prices_date_flags %>% filter(ticker == "SPY" &
holiday_flag == 1), aes(date, volume, text = paste("Post Holiday", "
date:", date, "
 volume:", volume)), color = "black") +
  scale_x_date(breaks = scales::pretty_breaks(n = 10)) +
  scale_y_continuous(labels = scales::number_format(accuracy = 1,
                                    scale = 1/1000000,
                                    suffix = "M")) +
  theme_minimal() +
  theme(plot.title = element_text(hjust = .5)) +
  labs(y = "")
) %>%
  ggplotly(tooltip = "text") %>%
  layout(title = "Daily Volume")

volume_plot_flags
```

20112012201320142015201620172018201920200M200M400M600M

Daily Volumedate

Here's a quick look at the price and moving daily averages.

```
rolling_average_plot <-
(
  etf_prices_date_flags %>%
  filter(ticker ==  "SPY") %>%
  ggplot(aes(x = date)) +
  geom_line(aes(y = close), color = "purple", linetype = "dotted") +
  geom_line(aes(y = sma_50), color = "cornflowerblue", linetype = "solid") +
  geom_line(aes(y = sma_200), color = "green", linetype = "solid") +
  scale_y_continuous(labels = scales::dollar) +
  scale_x_date(breaks = scales::pretty_breaks(n = 10)) +
  theme_minimal() +
  theme(plot.title = element_text(hjust = .5)) +
  labs(x = "date", title = "SMA 50 v. SMA 200", y = "", x= "")
) %>%
  ggplotly()

rolling_average_plot
```

20112012201320142015201620172018201920202$100$150$200$250$300

SMA 50 v. SMA 200date

Those two charts look close to what we want, but for fun let's give our end user the ability to toggle different aesthetics onto and off of the charts. `plotly` has a nifty `updatemenus` argument for it's `layout()` function that allows us to customize buttons.

We first create a list of buttons with the following code.

```
price_update_menus <- list(

  list(
    active = -1,
```

```
    type= 'buttons',
    buttons = list(
      list(
        label = "All",
        method = "update",
        args = list(list(visible = c(TRUE, TRUE, TRUE)),
                    list(title = "Price, SMA 50 & SMA 200"))),
       list(
        label = "price",
        method = "update",
        args = list(list(visible = c(TRUE, FALSE, FALSE)),
                    list(title = "Current Price"))),
      list(
        label = "sma 50",
        method = "update",
        args = list(list(visible = c(FALSE, TRUE, FALSE)),
                    list(title = "SMA 50"))),
      list(
        label = "sma 200",
        method = "update",
        args = list(list(visible = c(FALSE, FALSE, TRUE)),
                    list(title = "SMA 200"))))
  )
)
```

Then add to the chart with `layout(updatemenus = price_update_menus)`.

```
  rolling_average_plot %>%
  layout(updatemenus = price_update_menus)
```

20112012201320142015201620172018201920 20$100$150$200$250$300

SMA 50 v. SMA 200dateAllpricesma 50sma 200

Try clicking on the buttons to see how the chart responds.

We can do the same with the BLS reports flags.

```
volume_update_menus <- list(

  list(
    active = -1,
    type= 'buttons',
    buttons = list(
      list(
        label = "All",
        method = "update",
        args = list(list(visible = c(TRUE, TRUE, TRUE)),
                    list(title = "BLS Reports and Day after Holidays
Flagged"))),
      list(
        label = "BLS",
        method = "update",
        args = list(list(visible = c(TRUE, TRUE, FALSE)),
                    list(title = "BLS Reports"))),
      list(
        label = "Holidays",
        method = "update",
        args = list(list(visible = c(TRUE, FALSE, TRUE)),
```

```
                    list(title = "Day after Holidays"))))
  )
)

volume_plot_flags %>%
  layout(updatemenus = volume_update_menus)
```

2011201220132014201520162017201820192020 0M200M400M600M

Daily VolumedateAllBLSHolidays

Remember way back in the beginning of this post, we imported prices for several ETFs, but thus far we have been working with SPY data. If we want to apply this work with flagged dates to others, we can use a Shiny application, which we will construct next time.

That’s all for today!