Prices Periods Returns Chapter

Welcome to section 1, wherein we will perform the unglamorous work of taking raw price data for individual assets and tranforming them into monthly returns for a single portfolio. To map a data science work flow onto portfolio analysis, these 9 steps encompass data import, cleaning, wrangling, transformation and initial visualization. Even though the subtstantive issues are not complex, we will painstakingly go through the code to ensure that it is clear, reproducible and reusable. Our collaborators will thank us for this effort, including that most important of collaborators: our future self who needs to analyize risk/reward ratios, model betas and run simulations.

Here's what we need to accomplish:

1) Import daily prices from Yahoo! finance.
2) Select the adjusted prices only.
3) Transform daily prices to monthly prices.
4) Transform monthly prices to monthly returns.
5) Chart monthly returns.
6) Choose allocations or weights for each asset.
7) Calculate portfolio monthly returns based on asset monthly returns and weights.
8) Chart portfolio returns
9) Save all of our data objects for use by our collaborators and future self

Our ultimate goal is to constructe a 5-asset portfolio consisting of the following.

```
+ SPY (S&P500 fund) weighted 25%
+ EFA (a non-US equities fund) weighted 25%
+ IJS (a small-cap value fund) weighted 20%
+ EEM (an emerging-mkts fund) weighted 20%
+ AGG (a bond fund) weighted 10%
```

I chose those 5 assets because they seem to offer a balanced blend of large, small, international, emerging and bond exposure. We will include several Shiny applications in this book and those will enable you or any other end user to build a custom portfolio. For the rest of our work inside of this book, we will not change or deviate from these 5 assets and the chosen portfolio. That said, changing to different assets and weights does not involve a heavy lift and I encourage you to experiment with different asset combinations.

Let's get to step 1 wherein we import adjusted price data for the 5 ETFs to be used in our porftolio and save them to an `xts` object called `prices`.

First, we need to choose ticker symbols and store them in a vector called symbols. We do that with `symbols <- c("SPY","EFA", "IJS", "EEM","AGG")`. Those are the tickers for the 5 assets in our portfolio.

We will then pass `symbols` to Yahoo! Finance via the `getSymbols()` function from the `quantmod` package. This will return an object with the opening price, closing price, adjusted price, daily high, daily low and daily volume. We don't want to work with all of those, though. The adjusted price is what we want.

To isolate the adjusted price, we use the `map()` function from the `purrr` package and apply `Ad(get(.))` to the imported prices. This will `get()` the adjusted price from each of our individual price series. If we wanted the closing price, we would run `Cl(get(.))`. That `.` refers to our initial object. Note that if you wish to choose different stock tickers, you change the tickers in the `symbols` vector.

We could stop here and have the right substance - daily prices for 5 tickers - but the format wouldn't be great as we would have a `list` of 5 adjusted prices. Since those prices are `xts` objects, we would have a list of 5 `xts` objects. This is because the `map()` function returns a list by default.

The `reduce(merge)` function will allow us to merge the 5 lists into one `xts` object. The `merge()` function looks for the date index shared by our objects and uses that index.

Finally, we want intuitive column names and use `colnames<-` to rename the columns according to the `symbols` object. The `rename()` function from `dplyr` will not work here because the object structure is still `xts`.

```
symbols <- c("SPY","EFA", "IJS", "EEM","AGG")

# The prices object will hold our raw price data throughout this book.
prices <-
  getSymbols(symbols, src = 'yahoo', from = "2013-01-01",
             auto.assign = TRUE, warnings = FALSE) %>%
  map(~Ad(get(.))) %>%
  reduce(merge) %>%
  `colnames<-`(symbols)
```

Note that we are sourcing data from Yahoo! finance with `src = 'yahoo'` because that source is publicly available as of the time of this writing. In industry, we almost certainly wouldnn't be pulling from the internet but instead would be accesssing an internal database. In that situation, anyone wishing to reproduce or reuse or build upon our work must be able to import or update our raw data. It's a simple but oft overlooked first step that needs to be made clear. Where did the raw data come from and what code path was used to access it? Make sure it can be run in a clean R environment, meaning one in which the Global Environment has been cleared.

As to the starting date, I chose January 1, 2013. Why? This book is being published in 2018 so we will be working with 5 years or 60 months of data and I like round numbers.

Maybe my colleagues think that's cherry picking, maybe my clients think I need to go back to before the financial crisis bubble. They are entitled to to their opinions, and I to mine. The important thing is to make it easy for someone to test his/her own permutations. If a colleague looks at our work and wants to test a start date that goes back to the internet bubble, we need to enable that. And, indeed, a date change can be accomplished in the code below by changing `from = "2013-01-01"` to `from = "some other date"`.

Back to the code, we now have an `xts` object of the adjusted prices for our 5 assets. Have a quick peek.

```
head(prices)
```

```
##                 SPY      EFA      IJS      EEM      AGG
## 2013-01-02 132.7998 50.45466 77.71736 41.29057 98.66173
## 2013-01-03 132.4998 49.96531 77.61452 40.99837 98.41278
## 2013-01-04 133.0817 50.21872 78.22221 41.08055 98.51948
## 2013-01-07 132.7180 50.00027 77.76411 40.77010 98.46609
## 2013-01-08 132.3361 49.72064 77.46494 40.40486 98.55505
## 2013-01-09 132.6725 49.97405 77.68932 40.57834 98.48389
```

If you are running this code in the RStudio IDE, there will now be an object called `prices` in your Global Environment.

Convert Daily Prices to Monthly Log Returns: before the code

Next we want to turn those daily prices into monthly returns. This seems like a rather innocuous step in our work, but it involves two important decisions to be highlighted in the name of reproducibility. First, we are changing time periods from daily to monthly and thus we are transforming our data. We need to explain how that's happening.

More importantly, we will be transforming our data from its raw form, adjusted prices, to a calculated form, log returns.

This is such a standard step that the temptation is to include a few lines of code and move on to the analysis, which is the stuff our team gets paid to do. But, converting to log returns is our first major data processing

decision: why did we choose log returns instead of simple returns? It's a standard practice to use log returns but it's also a good chance to set the precedent amongst our team and within our workflow that we justify and explain decisions about our data, even decisions that are standard operating procedure in the financial world. If we have made the decision to work with log returns across our work, we should point to an R Notebook or a PDF that explains the decision and the brief substantive justification.

In this case, I know that simulating returns is in our future in the Monte Carlo chapter, and we will be assuming a normal distribution of returns. Thus, I choose to convert to log returns. Plenty of people will disagree with making this transformation, then assuming a normal distribution, then simulating based on that assumption, and that's fine.

In industry and when establishing a data science practice, an explanatory R Notebook can serve three purposes. First, for new team members, they will have a reference library that helps contextualize team-wide decisions. Second, should anyone every ask, why have we chosen log returns as the standard? The team can point to the reference material, and invite theoretical disagreements should the questioner not agree with that material. Third, it sets the standard for a best practice: this team justifies decisions that affect our data and conclusions.

To monthly log returns: 2 paths

Since financial data are generally in time series format, we will inhabit two general worlds of R code for analyzing those time series. The first is what I'll call the `xts` world. `xts` is an R package and a data format - it stands for extensible time series. `xts` objects, of which we have already created one, have a data index, not a date column. For example, if I want to look at the date for our prices `xts` object, I use `index(prices)`. The second world is the tidy world of data frames which we will get to shortly. Throughout this book we will write and run code in both worlds, to confirm that we get consistent results and also because we want our code to be reproducible by coders who have a preference and fluency for either of those worlds.

To the `xts` method for converting daily prices to monthly returns

The first observation in our `prices` object is January 2, 2013 (the first trading day of that year) and we have daily prices. We want to convert to those daily prices to monthly log returns based on the last reading of each month.

We will use `to.monthly(prices, indexAt = "last", OHLC = FALSE)` from the `quantmod` package. The argument `index = "last"` tells the function whether we want to index to the first day of the month or the last day.

```
prices_monthly <- to.monthly(prices, indexAt = "last", OHLC = FALSE)

head(prices_monthly)
```

```
##                 SPY      EFA      IJS      EEM      AGG
## 2013-01-31 136.1093 51.53820 79.68997 40.37746 98.16378
## 2013-02-28 137.8460 50.87410 80.98946 39.45523 98.74377
## 2013-03-28 143.0802 51.53820 84.31647 39.05346 98.84107
## 2013-04-30 145.8291 54.12472 84.41962 39.52828 99.79841
## 2013-05-31 149.2720 52.49066 88.03866 37.61989 97.80135
## 2013-06-28 147.2801 51.08500 87.91523 35.61634 96.27032
```

We have moved from an `xts` object of daily prices to an `xts` object of monthly prices. Note that we now have one reading per month, for the last day of each month.

Now we call `Return.calculate(prices_monthly, method = "log")` to convert to returns and save as an object called `assed_returns_xts`. Note this will give us log returns by the `method = "log"` argument. We

could have used `method = "discrete"` to get simple returns.

```
asset_returns_xts <- na.omit(Return.calculate(prices_monthly, method = "log"))

head(asset_returns_xts)
```

```
##                    SPY         EFA          IJS         EEM           AGG
## 2013-02-28  0.01267829 -0.01296935  0.016175316 -0.02310517  0.0058910455
## 2013-03-28  0.03726814  0.01296935  0.040258112 -0.01023508  0.0009848732
## 2013-04-30  0.01903020  0.04896783  0.001222619  0.01208476  0.0096390144
## 2013-05-31  0.02333508 -0.03065571  0.041976210 -0.04948336 -0.0202138300
## 2013-06-28 -0.01343412 -0.02714437 -0.001402959 -0.05472852 -0.0157783747
## 2013-07-31  0.05038573  0.05186019  0.063541553  0.01315990  0.0026878683
```

Take a quick look at the monthly returns above, to make sure things appear to be in order. Notice in particular the date of the first value. We imported prices starting "2013-01-02" yet our first monthly return is for "2013-02-28". This is because we used the argument `indexAt = "last"` when we cast to a monthly periodicity (try changing to `indexAt = "first"` and see the result). That is not necessarily good or bad, but it might matter if that first month's returns makes a difference in our analysis. More broadly, it's a good time to note how our decisions in data transformation can affect the data that ultimately survive to our analytical stage. We just lost the first two months of daily prices.

From a subtantive perspective, we have accomplished our task for the chapter: we have imported daily prices, trimmed to adjusted prices, moved to monthly prices and transformed to monthly log returns, all in the `xts` world.

Let's do the same thing but with a different coding paradigm in the tidy world.

The Tidyverse and Tidyquant World: daily prices to monthly returns

We now take the same raw data, which is the `prices` object we created upon data import and convert it to monthly returns using 3 alternative methods. We will make use of the `dplyr`, `tidyquant`, `timetk` and `tibbletime` packages.

There are lots of differences between the `xts` world and the tidy world but a very important one is the date. As noted above, `xts` objects have a date index. As we'll see, data frames have a date column. We will see this difference in action soon but it's good to keep in mind from the outset. Let's get to it.

Our conversion of the `prices` object from `xts` to a data frame will start with the very useful `tk_tbl()` function from the `timetk` package.

In the piped workflow below, our first step is to use `tk_tbl(preserve_index = TRUE, rename_index = "date")` function to convert from `xts` to `tibble`. The two arguments will convert the `xts` date index to a date column, and rename it "date". If we stopped here, we would have a new object in `tibble` format.

Next we turn to `dplyr` to `gather()` our new dataframe into long format and then `group_by` asset. We have not done any calculations yet, we have just shifted from wide format, to long, tidy format. Notice that when we gathered our data, we renamed one of the columns to `returns` even though the data are still prices. The next step will explain why we did that.

Next, we want to calculate log returns and add those returns to the data frame. We will use `mutate` and our own calculation to get log returns: `mutate(returns = (log(returns) - log(lag(returns))))`. Notice that I am putting our new log returns into the `returns` column by calling `returns = ...`. This is going to remove the price data and replace it with log returns data. This is the explanation for why, when we called `gather` in the previous step, we renamed the column to `returns`. That allows us to simply replace that column with log return data instead of having to create a new column and then delete the price data column.

Our last two steps are to `spread` the data back to wide format, which makes it easier to compare to the `xts` object and easier to read, but is not a best practice in the tidyverse. We are going to look at this new object and compare to the `xts` object above, so we will stick with wide format for now.

Finally, we want to reorder the columns so that the date column is first.

```
asset_returns_dplyr_byhand <-
  prices %>%
  to.monthly(indexAt = "last", OHLC = FALSE) %>%
  tk_tbl(preserve_index = TRUE, rename_index = "date") %>%
  gather(asset, returns, -date) %>%
  group_by(asset) %>%
  mutate(returns = (log(returns) - log(lag(returns)))) %>%
  spread(asset, returns) %>%
  select(date, symbols)
```

Have a quick peek at the new object.

```
head(asset_returns_dplyr_byhand)
```

```
## # A tibble: 6 x 6
##         date          SPY         EFA          IJS         EEM
##       <date>        <dbl>       <dbl>        <dbl>       <dbl>
## 1 2013-01-31           NA          NA           NA          NA
## 2 2013-02-28  0.01267829 -0.01296935  0.016175316 -0.02310517
## 3 2013-03-28  0.03726814  0.01296935  0.040258112 -0.01023508
## 4 2013-04-30  0.01903020  0.04896783  0.001222619  0.01208476
## 5 2013-05-31  0.02333508 -0.03065571  0.041976210 -0.04948336
## 6 2013-06-28 -0.01343412 -0.02714437 -0.001402959 -0.05472852
## # ... with 1 more variables: AGG <dbl>
```

Notice that our object now includes a reading for January 2013, whereas `xts` excluded it. Let's make them consistent by removing that first row with the `slice()` function.

```
asset_returns_dplyr_byhand <- asset_returns_dplyr_byhand %>% slice(-1)

head(asset_returns_dplyr_byhand)
```

```
## # A tibble: 6 x 6
##         date          SPY         EFA          IJS         EEM
##       <date>        <dbl>       <dbl>        <dbl>       <dbl>
## 1 2013-02-28  0.01267829 -0.01296935  0.016175316 -0.02310517
## 2 2013-03-28  0.03726814  0.01296935  0.040258112 -0.01023508
## 3 2013-04-30  0.01903020  0.04896783  0.001222619  0.01208476
## 4 2013-05-31  0.02333508 -0.03065571  0.041976210 -0.04948336
## 5 2013-06-28 -0.01343412 -0.02714437 -0.001402959 -0.05472852
## 6 2013-07-31  0.05038573  0.05186019  0.063541553  0.01315990
## # ... with 1 more variables: AGG <dbl>
```

Now our two objects are consistent and we have a data frame that we could use for further work.

Let's explore a second method in the tidy world where we'll use the `tq_transmute` function from `tidyquant`. Instead of using `to.monthly` and `mutate`, and then supplying our own calculation, we use `tq_transmute(mutate_fun = periodReturn, period = "monthly", type = "log")` and go straight from daily prices to monthly log returns. Note that we select the period as 'monthly' in that function call, which means we can pass in the raw daily price `xts` object.

```
asset_returns_tq_builtin <-
  prices %>%
  tk_tbl(preserve_index = TRUE, rename_index = "date") %>%
  gather(asset, prices, -date) %>%
  group_by(asset) %>%
  tq_transmute(mutate_fun = periodReturn, period = "monthly", type = "log") %>%
  spread(asset, monthly.returns) %>%
  select(date, symbols) %>%
  slice(-1)

head(asset_returns_tq_builtin)
```

```
## # A tibble: 6 x 6
##         date         SPY         EFA          IJS         EEM
##       <date>       <dbl>       <dbl>        <dbl>       <dbl>
## 1 2013-02-28  0.01267829 -0.01296935  0.016175316 -0.02310517
## 2 2013-03-28  0.03726814  0.01296935  0.040258112 -0.01023508
## 3 2013-04-30  0.01903020  0.04896783  0.001222619  0.01208476
## 4 2013-05-31  0.02333508 -0.03065571  0.041976210 -0.04948336
## 5 2013-06-28 -0.01343412 -0.02714437 -0.001402959 -0.05472852
## 6 2013-07-31  0.05038573  0.05186019  0.063541553  0.01315990
## # ... with 1 more variables: AGG <dbl>
```

Note that we had to again remove the first row with `slice(-1)`.

Our third method in the tidy world will produce the same output as the previous two - a `tibble` of monthly log returns - but we will also introduce the `tibbletime` package and it's function `as_period`.
As the name implies, this function allows us to cast the prices time series from daily to monthly (or weekly or quarterly etc.) in our `tibble` instead of having to apply the `to.monthly` function to the `xts` object as we did previously.

Furthermore, unlike the previous code chunk above where we went from daily prices straight to monthly returns, here we go from daily prices to monthly prices to monthly returns. That is, we will first create a `tibble` of monthly prices, then pipe to create monthly returns.

We don't have a substantive reason for doing that here, but it could prove useful if there's a time when we need to get monthly prices in isolation during a tidyverse-based piped workflow.

```
asset_returns_tbltime <- prices %>%
  tk_tbl(preserve_index = TRUE, rename_index = "date") %>%
  tbl_time(index = "date") %>%
  as_period("monthly", side = "end") %>%
  gather(asset, returns, -date) %>%
  group_by(asset) %>%
  tq_transmute(mutate_fun = periodReturn, type = "log") %>%
  spread(asset, monthly.returns) %>%
  select(date, symbols) %>%
  slice(-1)
```

Let's take a peek at our 4 monthly log return objects.

```
head(asset_returns_xts)
```

```
##                   SPY         EFA          IJS         EEM           AGG
## 2013-02-28  0.01267829 -0.01296935  0.016175316 -0.02310517  0.0058910455
## 2013-03-28  0.03726814  0.01296935  0.040258112 -0.01023508  0.0009848732
## 2013-04-30  0.01903020  0.04896783  0.001222619  0.01208476  0.0096390144
```

```
## 2013-05-31  0.02333508 -0.03065571  0.041976210 -0.04948336 -0.0202138300
## 2013-06-28 -0.01343412 -0.02714437 -0.001402959 -0.05472852 -0.0157783747
## 2013-07-31  0.05038573  0.05186019  0.063541553  0.01315990  0.0026878683
```

```
head(asset_returns_dplyr_byhand)
```

```
## # A tibble: 6 x 6
##         date         SPY         EFA          IJS         EEM
##       <date>       <dbl>       <dbl>        <dbl>       <dbl>
## 1 2013-02-28  0.01267829 -0.01296935  0.016175316 -0.02310517
## 2 2013-03-28  0.03726814  0.01296935  0.040258112 -0.01023508
## 3 2013-04-30  0.01903020  0.04896783  0.001222619  0.01208476
## 4 2013-05-31  0.02333508 -0.03065571  0.041976210 -0.04948336
## 5 2013-06-28 -0.01343412 -0.02714437 -0.001402959 -0.05472852
## 6 2013-07-31  0.05038573  0.05186019  0.063541553  0.01315990
## # ... with 1 more variables: AGG <dbl>
```

```
head(asset_returns_tq_builtin)
```

```
## # A tibble: 6 x 6
##         date         SPY         EFA          IJS         EEM
##       <date>       <dbl>       <dbl>        <dbl>       <dbl>
## 1 2013-02-28  0.01267829 -0.01296935  0.016175316 -0.02310517
## 2 2013-03-28  0.03726814  0.01296935  0.040258112 -0.01023508
## 3 2013-04-30  0.01903020  0.04896783  0.001222619  0.01208476
## 4 2013-05-31  0.02333508 -0.03065571  0.041976210 -0.04948336
## 5 2013-06-28 -0.01343412 -0.02714437 -0.001402959 -0.05472852
## 6 2013-07-31  0.05038573  0.05186019  0.063541553  0.01315990
## # ... with 1 more variables: AGG <dbl>
```

```
head(asset_returns_tbltime)
```

```
## # A tibble: 6 x 6
##         date         SPY         EFA          IJS         EEM
##       <date>       <dbl>       <dbl>        <dbl>       <dbl>
## 1 2013-02-28  0.01267829 -0.01296935  0.016175316 -0.02310517
## 2 2013-03-28  0.03726814  0.01296935  0.040258112 -0.01023508
## 3 2013-04-30  0.01903020  0.04896783  0.001222619  0.01208476
## 4 2013-05-31  0.02333508 -0.03065571  0.041976210 -0.04948336
## 5 2013-06-28 -0.01343412 -0.02714437 -0.001402959 -0.05472852
## 6 2013-07-31  0.05038573  0.05186019  0.063541553  0.01315990
## # ... with 1 more variables: AGG <dbl>
```

Do we notice anything of interest?

First, have a look at the left most column/date in each object, where the date is stored. The `asset_returns_xts` has a date index, not a column. That index doesn't have a name. It is accessed via `index(asset_returns_xts)`. The data frame objects have a column called "date", accessed via the `$date` convention, e.g. `asset_returns_dplyr_byhand$date`.

Second, each of these objects is in "wide" format, which in this case means there is a column for each of our assets.

This is the format that `xts` likes and it's the format that is easier to read as a human. However, the tidyverse wants this data to be in long or tidy format so that each variable has its own column.

For our asset_returns objects, that would mean a column called "date", a colum called "asset" and a column called "returns". To see that in action, here is how it looks.

```
asset_returns_long <-
  asset_returns_dplyr_byhand %>%
  gather(asset, returns, -date)

head(asset_returns_long)
```

```
## # A tibble: 6 x 3
##         date asset     returns
##       <date> <chr>       <dbl>
## 1 2013-02-28   SPY  0.01267829
## 2 2013-03-28   SPY  0.03726814
## 3 2013-04-30   SPY  0.01903020
## 4 2013-05-31   SPY  0.02333508
## 5 2013-06-28   SPY -0.01343412
## 6 2013-07-31   SPY  0.05038573
```

`asset_returns_long` has 3 columns, one for each variable: date, asset, return As I said, this format is harder to read as a human - we can see only the first several reading for one asset. From a tidyverse perspective, this is considered 'tidy' data or long data and it's the preferred format. When we get to visualizing and manipulating this data, it will be clearer as to why the tidyverse likes this format.

For now, spend a few minutes looking at the `xts` object `asset_returns_xts` and our various data frames, then look at the long, tidy object `asset_returns_long` object. Make sure that logic of how we got from daily prices to log returns for each object makes sense.

A Word on workflow and recap

Let's recap what we've done thus far. We have imported raw price data for 5 assets, in a reprudicible and flexible way. We have used 4 different methods for converting those daily prices to monthly, log returns. From those 4 methods, we now have 5 objects: `asset_returns_xts` (an xts object), `asset_returns_dplyr_byhand`, `asset_returns_tq_builtin` (a tibble object created with tidyquant and timetk) and `asset_returns_long` (a data frame in long instead of wide format).

We can think of our work thus far in terms of a wholistic data science workflow, that begins with data import and transformation. Data import and transformation is not the most glamorous of work but it needs to be so crystal clear that our colleagues find it stunningly easy to follow the origin of our data. If we wish for our work to lay the ground work for several potential projects or test strategies that will increase in complexity, this first step needs to be clear and accessible.

There's a high likelihood that we will encounter work from other team members who have their own methods for data import and transformation. The more methods we can master or at least practice, the better prepared we will be to reuse or expand on our colleagues' work.

Data import and transofrmation is straightforward, but it also forces us to engage with our data in its rawest form, instead of skipping ahead to the model and the R squared. A data scientist can never spend too much time getting to know his/her data. Perhaps new insights will jump out, or an error will be found, or a new hypothesis. Furthermore, when it comes time to defend or update our findings or conclusions, deep knowledge of the raw data is crucial.

Visualizing Asset Returns before they get mashed into a portfolio

We could jump straight into the process of converting these assets into a portfolio, but it's good practice to have a quick peek at the individual charts before doing so. Once a portfolio is built, we're unlikely to back track to visualizing returns on an individual basis. Yet, those individual returns are the building blocks and raw material of our portfolio. Visualizing their returns adds another chance to get to know our raw data.

For the purposes of visualizing returns, we will work with two of our monthly log returns objects, `asset_returns_xts` and `asset_returns_long` (the tidy, long formatted data frame)

First, let's use `highcharter` to visualize the `xts` formatted returns.

Highcharter is fantastic for visualizing a time series or many time series. First, we set `highchart(type = "stock")` to get a nice line format. Then we add each of our series to the highcharter code flow with `hc_add_series(asset_returns_xts[, symbols[1]], name = symbols[1])`. Notice that use our original `symbols` object to refernce the columns. This will allow the code to run should we change to different ticker symbols at the outset.

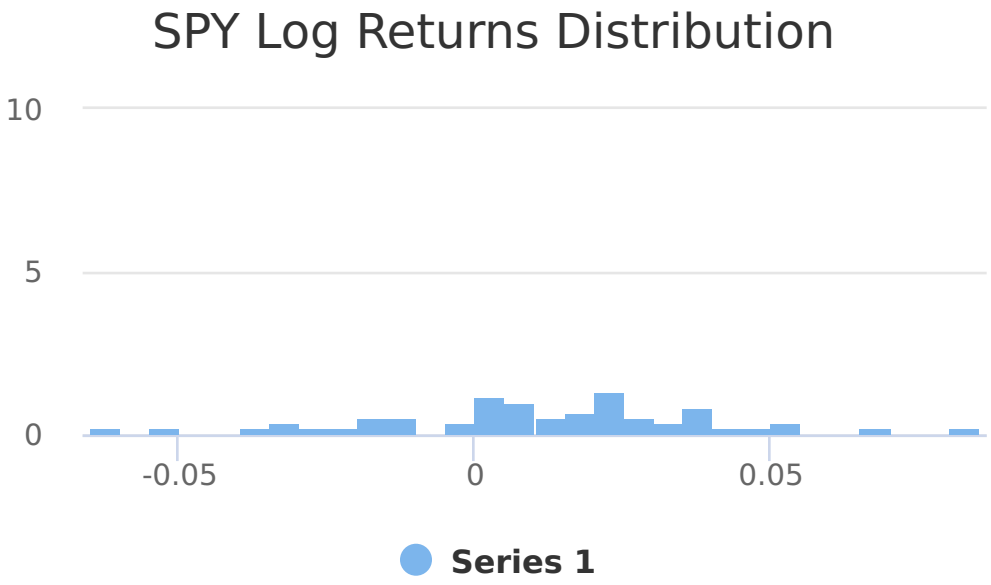
```
highchart(type = "stock") %>%
  hc_title(text = "Monthly Log Returns") %>%
  hc_add_series(asset_returns_xts[, symbols[1]],
                name = symbols[1]) %>%
  hc_add_series(asset_returns_xts[, symbols[2]],
                name = symbols[2]) %>%
  hc_add_series(asset_returns_xts[, symbols[3]],
                name = symbols[3]) %>%
  hc_add_series(asset_returns_xts[, symbols[4]],
                name = symbols[4]) %>%
  hc_add_series(asset_returns_xts[, symbols[5]],
                name = symbols[5]) %>%
  hc_add_theme(hc_theme_flat()) %>%
  hc_navigator(enabled = FALSE) %>%
  hc_scrollbar(enabled = FALSE)
```

Take a look at the chart. It has a line for the monthly log returns of our ETFs (and in my opinion it's starting to get crowded).

Highcharter also has the capacity for histogram making. One method is to first call the base function `hist` on the data along with the arguments for breaks and `plot = FALSE`. Then we can call `hchart` on that object.

```
hc_hist <- hist(asset_returns_xts[, symbols[1]], breaks = 50, plot = FALSE)
```

```
hchart(hc_hist) %>%
  hc_title(text = paste(symbols[1], "Log Returns Distribution", sep = " "))
```

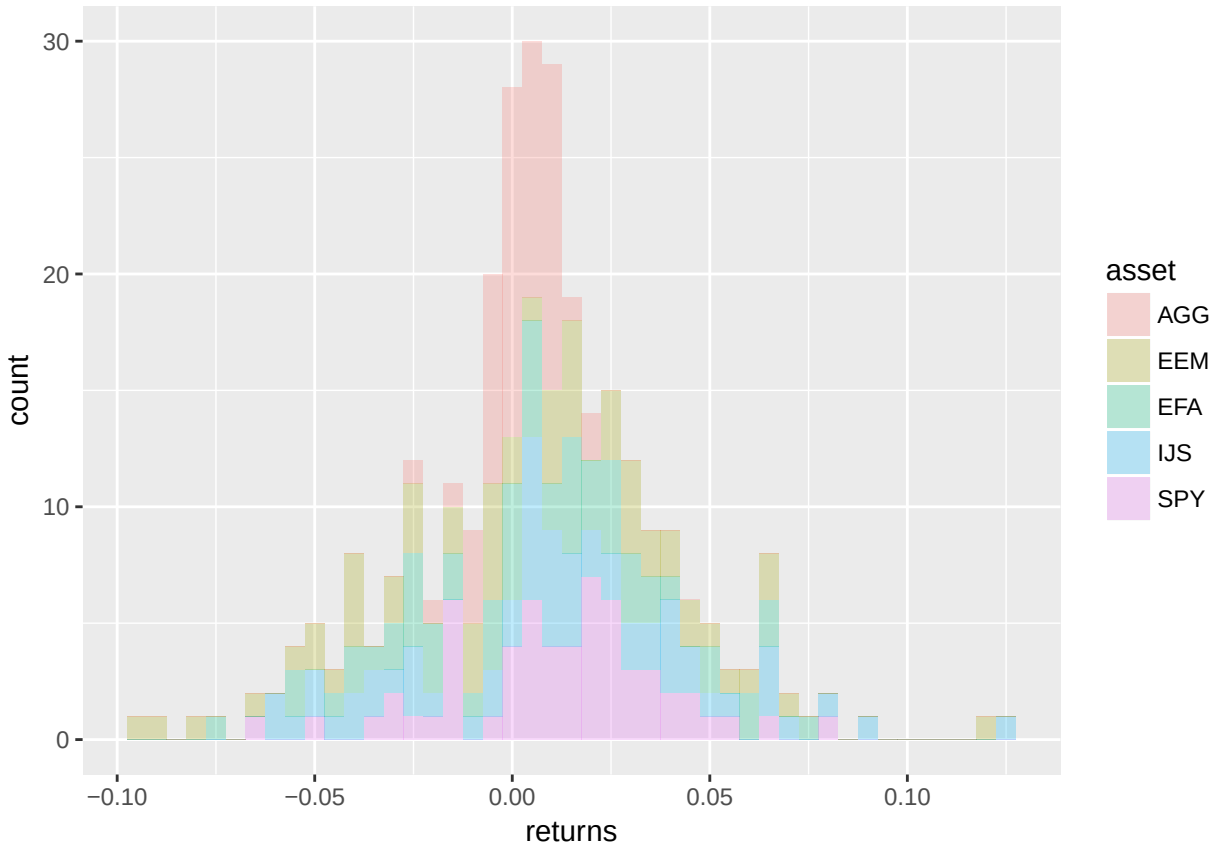


That's a nice histogram but `highcharter` doesn't have a smooth way to create 5 histograms as we to do.

For that, we will head to the tidyverse and use `ggplot2` on our tidy data frame `assets_returns_long`. We call `ggplot(aes(x = returns, fill = asset)) + geom_histogram(alpha = 0.25, binwidth = .005)` and because the data frame is grouped by the 'asset' column, `ggplot()` knows to chart a separate histogram for each asset.

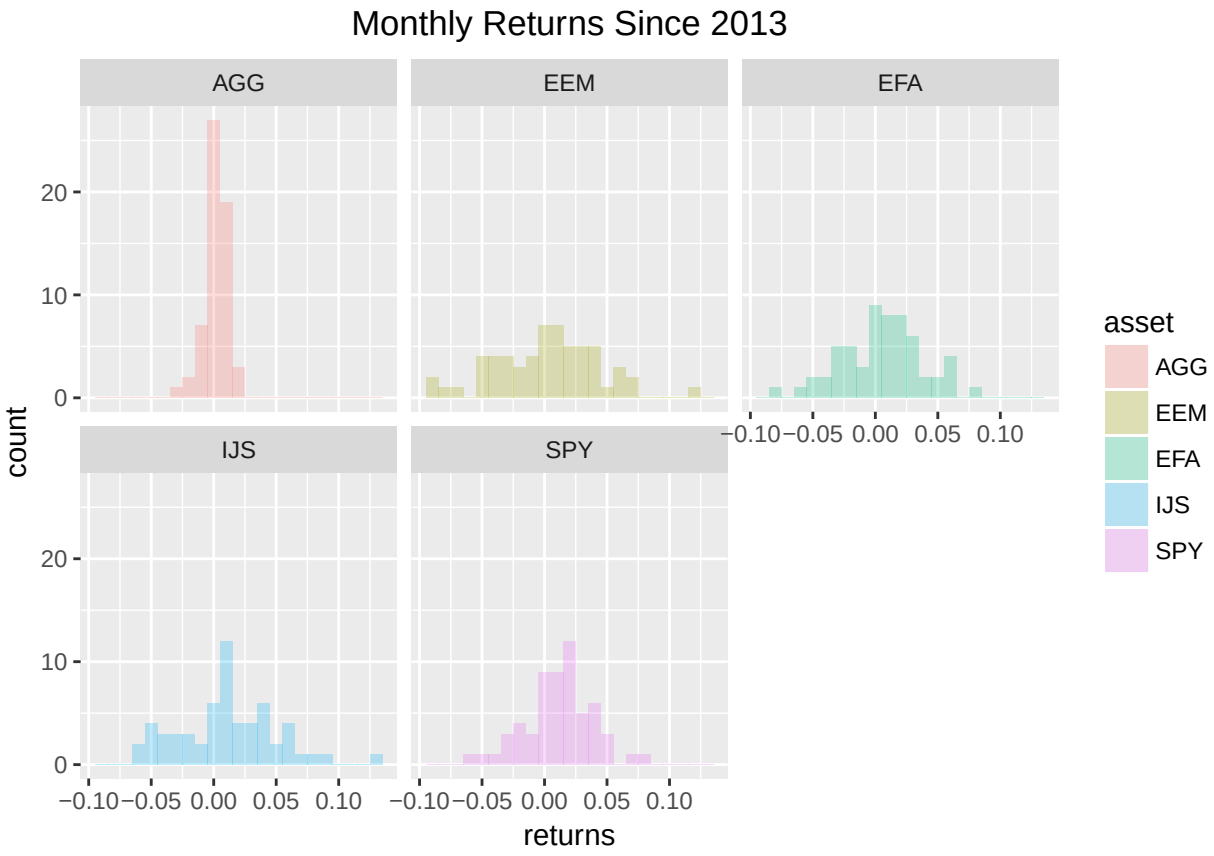
```
# Make so all titles centered in the upcoming ggplots
theme_update(plot.title = element_text(hjust = 0.5))

asset_returns_long %>%
  ggplot(aes(x = returns, fill = asset)) +
  geom_histogram(alpha = 0.25, binwidth = .005)
```



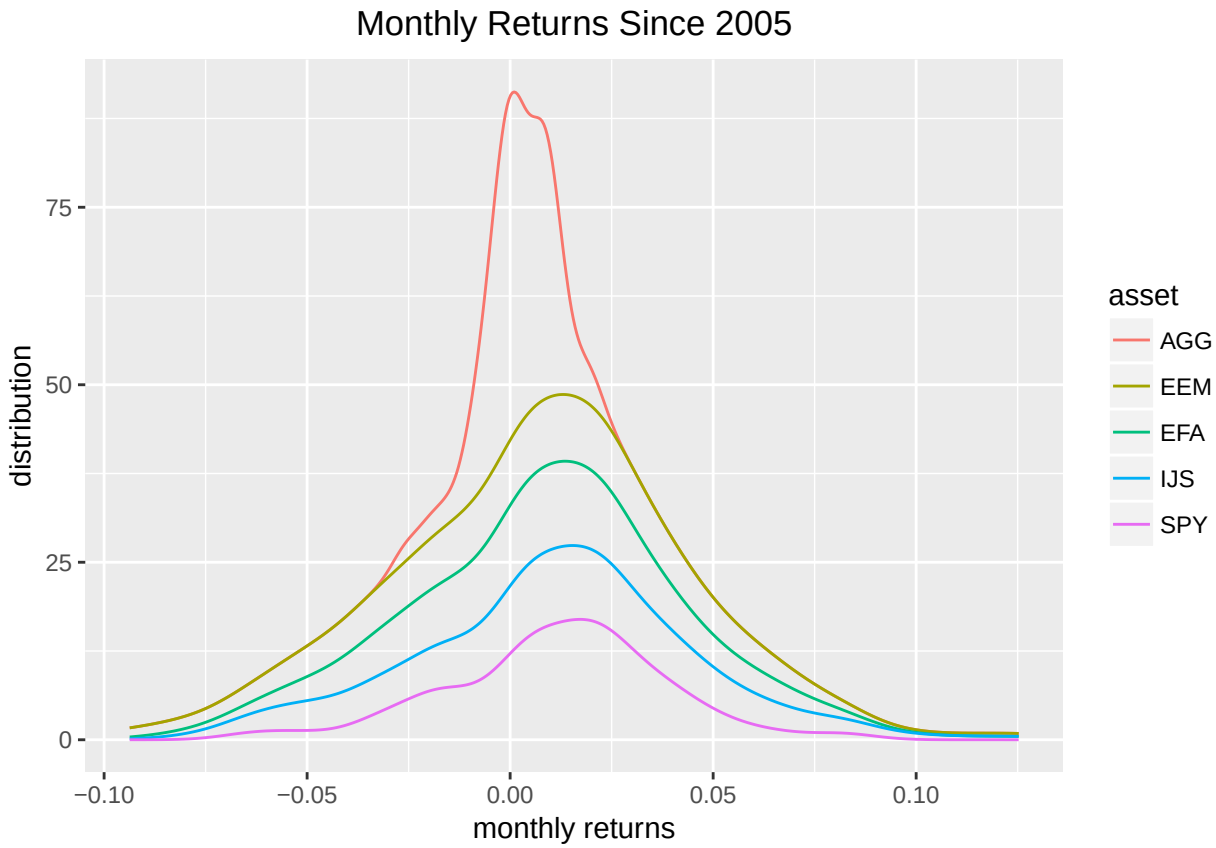
Let's use `facet_wrap(~asset)` to break these out by asset. We can add a title with `ggtitle`.

```
asset_returns_long %>%
  ggplot(aes(x = returns, fill = asset)) +
  geom_histogram(alpha = 0.25, binwidth = .01) +
  facet_wrap(~asset) +
  ggtitle("Monthly Returns Since 2013")
```



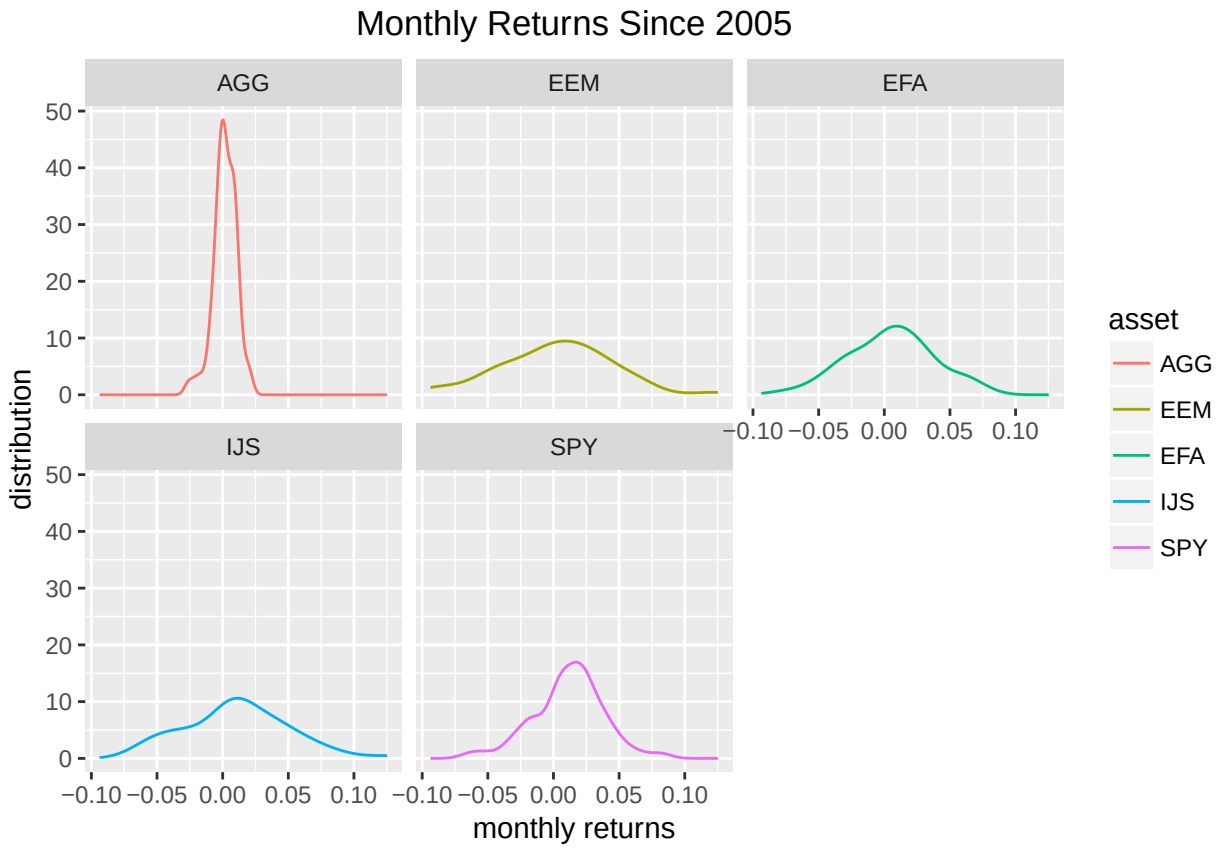
Maybe we don't want to use a histogram, but instead want to use a density line to visualize the various distributions. We can use the `stat_density(geom = "line", alpha = 1)` function to do this. The `alpha` argument is selecting a line thickness. Let's also add a label to the x and y axis with the `xlab` and `ylab` functions.

```
asset_returns_long %>%
  ggplot(aes(x = returns, colour = asset, fill = asset)) +
  stat_density(geom = "line", alpha = 1) +
  ggtitle("Monthly Returns Since 2005") +
  xlab("monthly returns") +
  ylab("distribution")
```



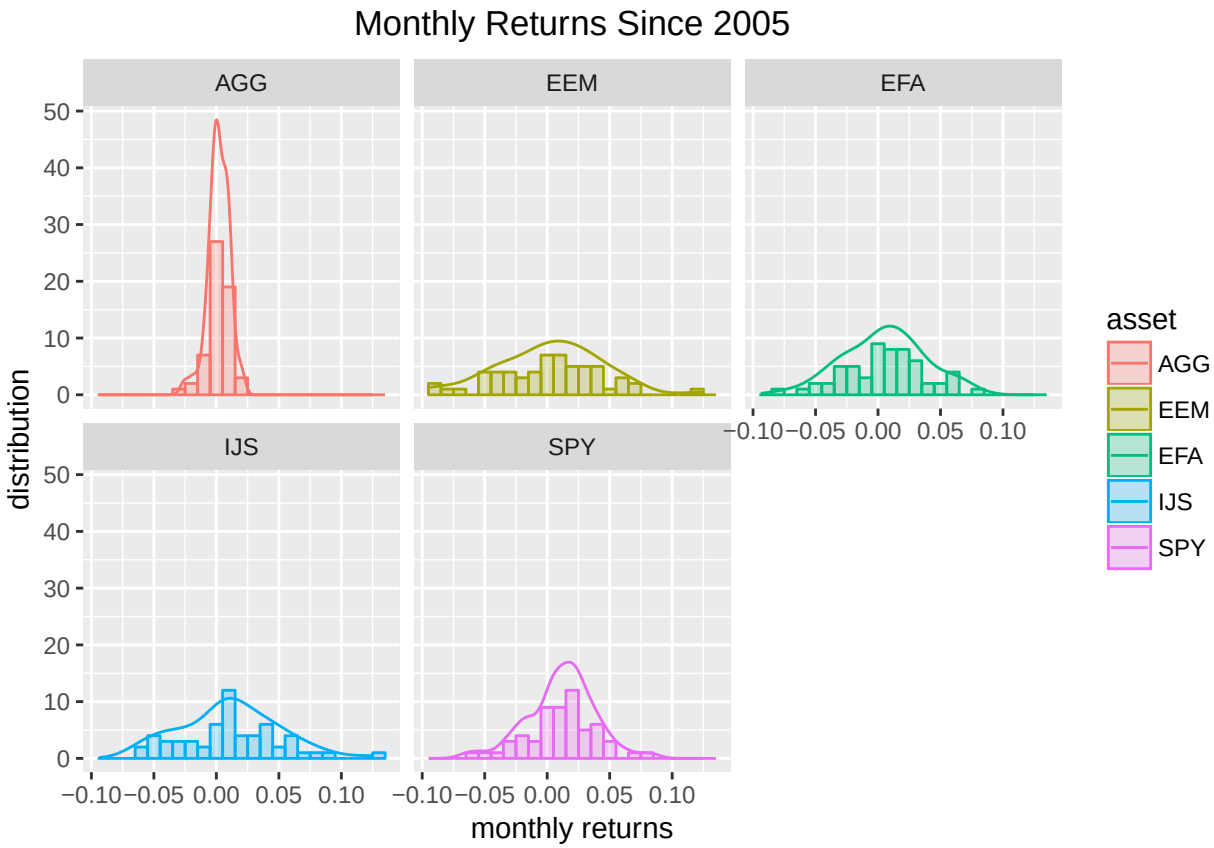
That chart is quite digestible, but we can also `facet_wrap(~asset)` to break the densities out into individual charts.

```
asset_returns_long %>%
  ggplot(aes(x = returns, colour = asset, fill = asset)) +
  stat_density(geom = "line", alpha = 1) +
  facet_wrap(~asset) +
  ggtitle("Monthly Returns Since 2005") +
  xlab("monthly returns") +
  ylab("distribution")
```



Now we can combine all of our ggplots into one nice, faceted plot.

```
asset_returns_long %>%
  ggplot(aes(x = returns, colour = asset, fill = asset)) +
  stat_density(geom = "line", alpha = 1) +
  geom_histogram(alpha = 0.25, binwidth = .01) +
  facet_wrap(~asset) +
  ggtitle("Monthly Returns Since 2005") +
  xlab("monthly returns") +
  ylab("distribution")
```



We now have one chart, with histograms and line densities broken out for each of our assets. This would scale nicely if we had more assets and wanted to peek at more distributions of returns.

To the Portfolio Station

We now turn our collection of individual returns into a portfolio, which is really a weighted collection of asset returns. Accordingly, the first thing we need to do is assign a weight to each asset. Recall that our vector of symbols is SPY, EFA, IJS, EEM, AGG. Let's create a weights vector that will allow us to assign a weight to each of our symbols. We are going for a balanced portfolio and will weight relatively little to AGG, the bond fund.

```
w <- c(0.25, 0.25, 0.20, 0.20, 0.10)
```

Before we use the weights in our calculations, a quick sanity check in the next code chunk is a good idea. This might not be necessary with 5 assets as we have today, but good practice because if we had 50 assets it could save us a lot of grief to catch a mistake early.

```
asset_weights_sanity_check <- tibble(w, symbols)
asset_weights_sanity_check
```

```
## # A tibble: 5 x 2
##       w symbols
##   <dbl>   <chr>
## 1  0.25     SPY
## 2  0.25     EFA
## 3  0.20     IJS
## 4  0.20     EEM
## 5  0.10     AGG
```

Does that tibble match up with the portfolio we want to create? Looks good to me.

Finally, make sure the weights sum to 100%, or 1. Again, we can eyeball this with 5 assets, but with 50 assets it would be easier to run the sanity check.

```
sum(asset_weights_sanity_check$w)
```

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

All looks good.

Now let's code up some portfolio returns.

The textbook equation for the return of a multi-asset portfolio is:

$$Return_{portfolio} = W_1 * Return_{asset1} + W_2 * Return_{asset2} + W_3 * Return_{asset3} + W_4 * Return_{asset4} + W_5 * Return_{asset5}$$

Let's implement that equation with R code. First, we assign weights to variables.

```
w_1 <- w[1]
w_2 <- w[2]
w_3 <- w[3]
w_4 <- w[4]
w_5 <- w[5]
```

And each asset has a return as well, stored in our `asset_returns_xts` object.

```
asset1 <- asset_returns_xts[,1]
asset2 <- asset_returns_xts[,2]
asset3 <- asset_returns_xts[,3]
asset4 <- asset_returns_xts[,4]
asset5 <- asset_returns_xts[,5]
```

Now let's use the weights and returns in the equation.

```
portfolio_returns_byhand <-
  (w_1 * asset1) +
  (w_2 * asset2) +
  (w_3 * asset3) +
  (w_4 * asset4) +
  (w_5 * asset5)

names(portfolio_returns_byhand) <- "returns"

head(portfolio_returns_byhand)
```

```
##                  returns
## 2013-02-28 -0.0008696311
## 2013-03-28  0.0186624647
## 2013-04-30  0.0206248842
## 2013-05-31 -0.0053529706
## 2013-06-28 -0.0229487552
## 2013-07-31  0.0411705578
```

Our by-hand method is complete, but let's confirm we get the same results with the built-in methods.

For our first built-in method, we will stay in the `xts` world and use the `Return.portfolio()` function from the `PerformanceAnalytics` package. You might have noticed that we didn't explicitly load that package and that is because `tidyquant` imports this package for us. The function requires two arguments for a portfolio, an `xts` object of returns and a vector of weights. It's not necessary but we are also going to set `rebalance_on = "months"` so we can confirm it matches our by hand calculations above. Remember, in

the by hand equation, we set the portfolio weights as fixed, meaning they never changed on a month to month basis. That is equivalent to rebalancing every month. In practice, that would be quite rare. Once we confirm that it matches our by hand, we can toggle over to a more realistic annual rebalancing by changing the argument to `rebalance_on = "years"`.

```
portfolio_returns_xts_rebalanced_monthly <-
  Return.portfolio(asset_returns_xts, weights = w, rebalance_on = "months") %>%
  `colnames<-`("returns")
```

Let's use the built-in `Return.portfolio` function again but we will set a more realistic annual rebalancing with the argument `rebalance_on = "years"`. This will change our results so that they no longer our by-hand calculation, which effectiely rebalanced every month.

```
portfolio_returns_xts_rebalanced_yearly <-
  Return.portfolio(asset_returns_xts, weights = w, rebalance_on = "years") %>%
  `colnames<-`("returns")
```

We can take a peek at our three portfolio objects and see how the annual rebalance made a small but important difference to our monthly returns.

```
head(portfolio_returns_byhand)
```

```
##                  returns
## 2013-02-28 -0.0008696311
## 2013-03-28  0.0186624647
## 2013-04-30  0.0206248842
## 2013-05-31 -0.0053529706
## 2013-06-28 -0.0229487552
## 2013-07-31  0.0411705578
```

```
head(portfolio_returns_xts_rebalanced_monthly)
```

```
##                  returns
## 2013-02-28 -0.0008696311
## 2013-03-28  0.0186624647
## 2013-04-30  0.0206248842
## 2013-05-31 -0.0053529706
## 2013-06-28 -0.0229487552
## 2013-07-31  0.0411705578
```

```
head(portfolio_returns_xts_rebalanced_yearly)
```

```
##                  returns
## 2013-02-28 -0.0008696311
## 2013-03-28  0.0189331170
## 2013-04-30  0.0204344954
## 2013-05-31 -0.0044738100
## 2013-06-28 -0.0218914083
## 2013-07-31  0.0425167801
```

As before, we could stop here and have accomplished our substantive task (twice already - by-hand and using the built-in function), but we want to explore alternate methods in the world of tidyverse and tidyquant.

First, we will use our long, tidy formatted `asset_returns_long` and convert to portfolio returns using the `tq_portfolio` function from `tidyquant`.

The `tq_portfolio` function takes a `tibble` and then asks for an assets column to group by, a returns column to find return data, and a weights vector It's a wrapper for `Return.portfolio()` and thus also accepts the argument `rebalance_on = "months"`. Since we are rebalancing by months, we should again

get a portfolio returns object that matches our two existing objects `portfolio_returns_byhand` and `portfolio_returns_xts_rebalanced_monthly`.

```
portfolio_returns_tq_rebalanced_monthly <-
  asset_returns_long %>%
  tq_portfolio(assets_col  = asset,
               returns_col = returns,
               weights     = w,
               col_rename  = "returns",
               rebalance_on = "months")
```

If we want to rebalance annually, it'st the same code as above, except we set `rebalance_on = "years"`.

```
portfolio_returns_tq_rebalanced_yearly <-
  asset_returns_long %>%
  tq_portfolio(assets_col  = asset,
               returns_col = returns,
               weights     = w,
               col_rename  = "returns",
               rebalance_on = "years")
```

We now have two more portfolio returns objects and they are both tidy tibbles. Let's take a quick look and compare how a tidy `tibbles` of portfolio returns compare to the `xts` object of portfolio returns.

```
head(portfolio_returns_byhand)
```

```
##                  returns
## 2013-02-28 -0.0008696311
## 2013-03-28  0.0186624647
## 2013-04-30  0.0206248842
## 2013-05-31 -0.0053529706
## 2013-06-28 -0.0229487552
## 2013-07-31  0.0411705578
```

```
head(portfolio_returns_tq_rebalanced_monthly)
```

```
## # A tibble: 6 x 2
##         date       returns
##       <date>         <dbl>
## 1 2013-02-28 -0.0008696311
## 2 2013-03-28  0.0186624647
## 3 2013-04-30  0.0206248842
## 4 2013-05-31 -0.0053529706
## 5 2013-06-28 -0.0229487552
## 6 2013-07-31  0.0411705578
```

```
head(portfolio_returns_xts_rebalanced_monthly)
```

```
##                  returns
## 2013-02-28 -0.0008696311
## 2013-03-28  0.0186624647
## 2013-04-30  0.0206248842
## 2013-05-31 -0.0053529706
## 2013-06-28 -0.0229487552
## 2013-07-31  0.0411705578
```

Huzzah, we have three objects of portfolio returns, calculated in three different ways, and with the same results.

As we move on to visualization, we will make use of those different objects for our different visualization techniques.

Visualizing Portfolio Returns

As before, let's start wiht `highcharter` to visualize the `xts` formatted portfolio returns.

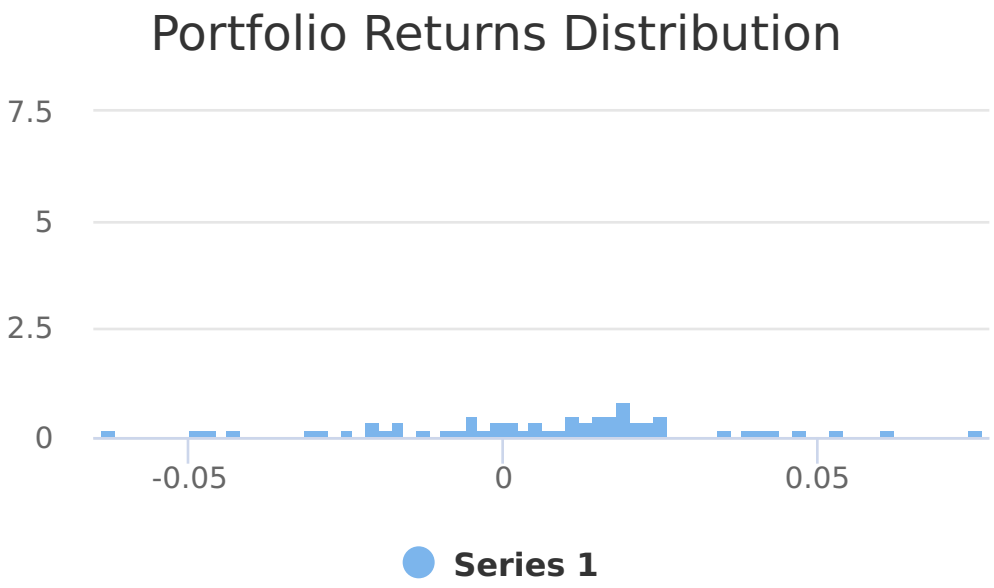
As we noted when looking at individual asset returns, `highcharter` is fantastic for visualizing a time series or many time series. First, we set `highchart(type = "stock")` to get a nice time series line. Then we add our `returns` column from the portfolio returns `xts` object. We don't have to add the date index or point to it in any way because `highcharter` recognizes the `xts` object and ports over the date index under the hood. The code below should look familiar from our work on asset returns.

```
highchart(type = "stock") %>%
  hc_title(text = "Porftolio Monthly Returns") %>%
  hc_add_series(portfolio_returns_xts_rebalanced_yearly$returns,
                name = "Rebalanced Yearly", color = "cornflowerblue") %>%
  hc_add_theme(hc_theme_flat()) %>%
  hc_navigator(enabled = FALSE) %>%
  hc_scrollbar(enabled = FALSE)
```

As before, we can use `highcharter` for histogram making, with the same code flow.

```
hc_portfolio <- hist(portfolio_returns_xts_rebalanced_yearly$returns, breaks = 50, plot = FALSE)

hchart(hc_portfolio) %>%
  hc_title(text = "Portfolio Returns Distribution")
```

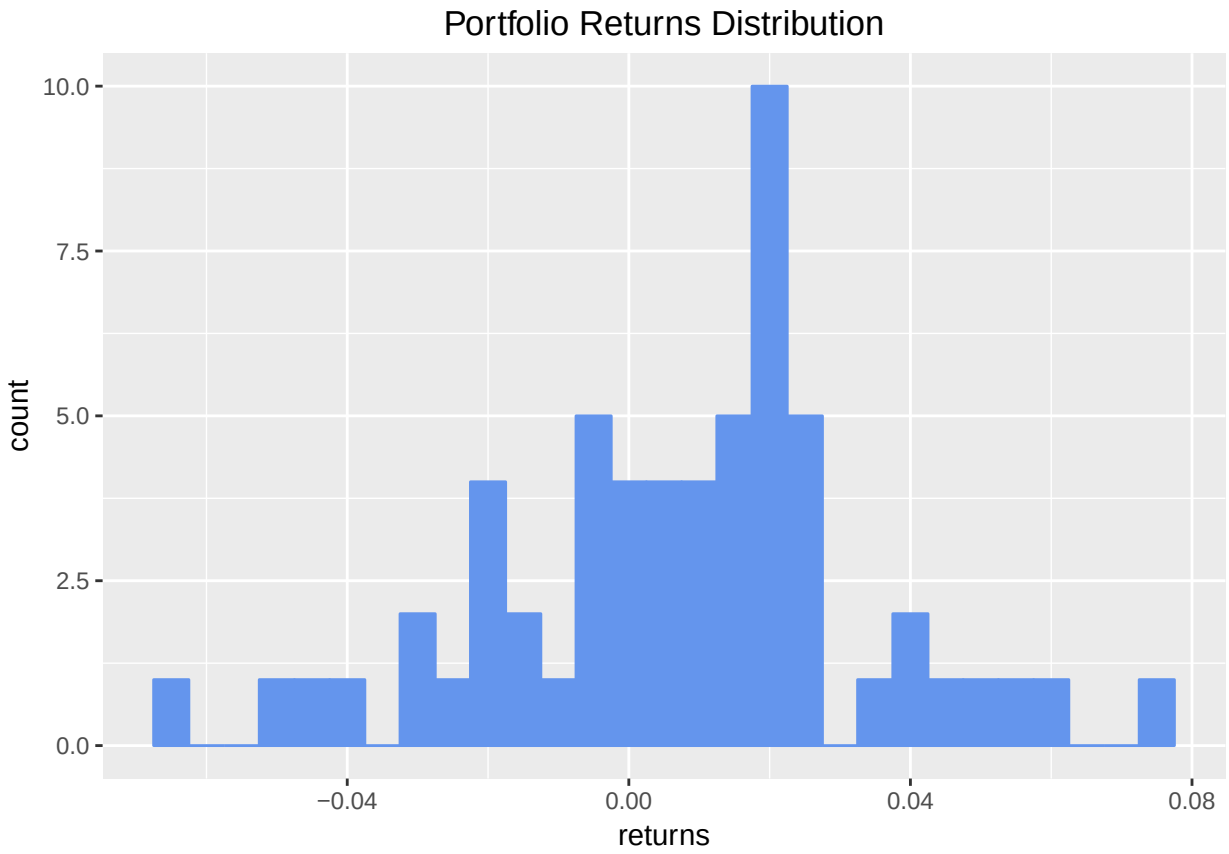


As we noted in the previous section on asset returns, there's nothing wrong with that highcharter histogram. It displays well the distribution of portfolio returns. It does not, however, offer as much flexibility as `ggplot` for adding other distributions or density lines to the same chart.

For that, we will head to the tidyverse and use `ggplot` on our tidy `tibble` 'portfolio_returns_tq_rebalanced_yearly.

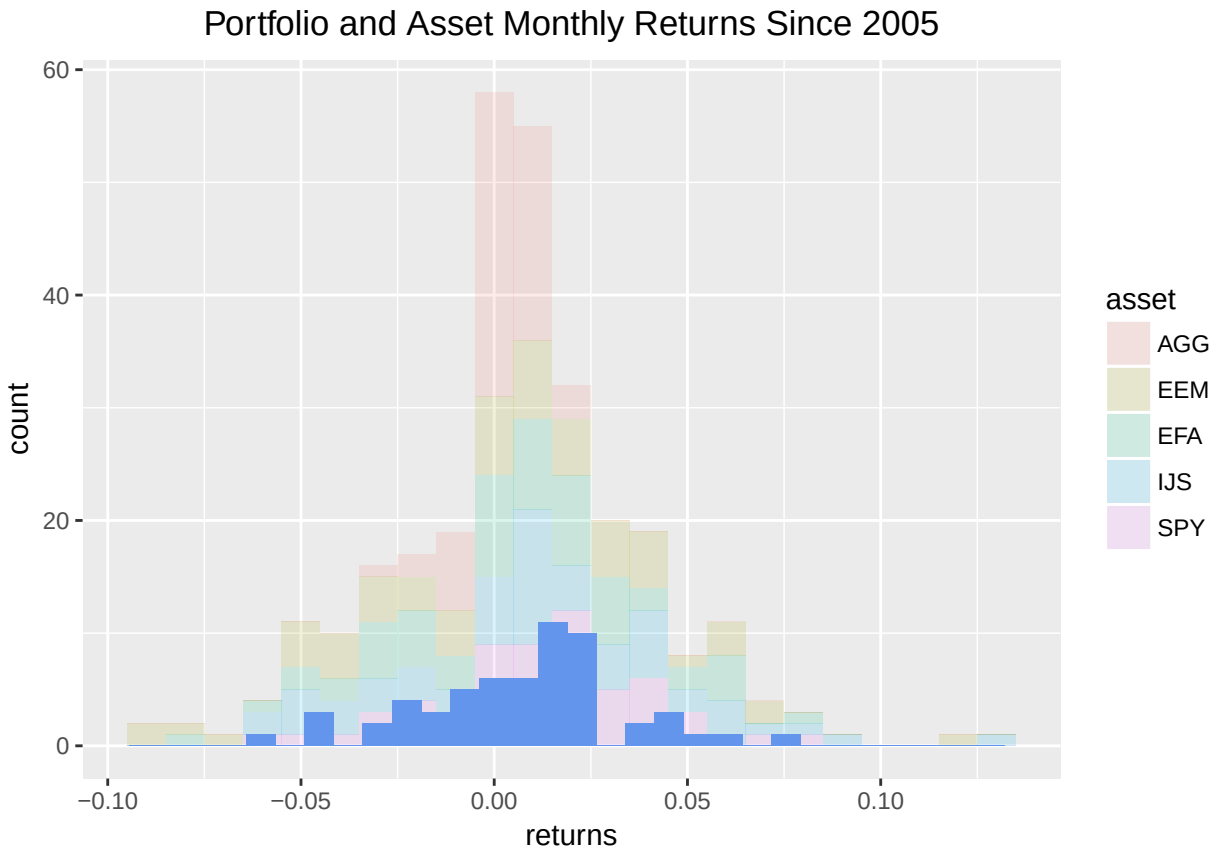
```
# Make so all titles centered in the upcoming ggplots
theme_update(plot.title = element_text(hjust = 0.5))

portfolio_returns_tq_rebalanced_yearly %>%
  ggplot(aes(x = returns)) +
  geom_histogram(binwidth = .005, fill = "cornflowerblue", color = "cornflowerblue") +
  ggtitle("Portfolio Returns Distribution")
```



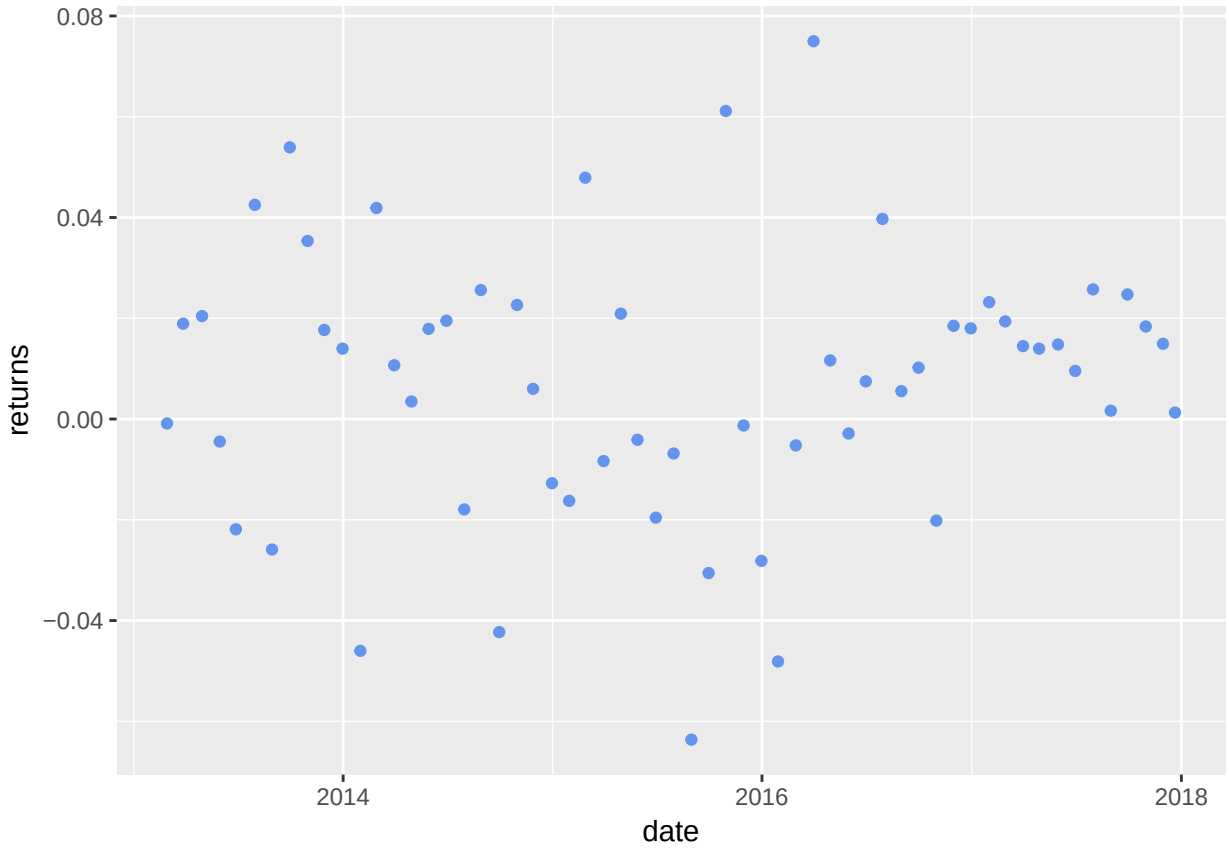
`ggplot()` makes it seemless to layer on other distributions. Let's compare the portfolio distribution to those of our individual assets. Use the alpha argument to make the asset histograms a bid faded, since there are more of them and the portfolio return is what we really want to see.

```
asset_returns_long %>%
  ggplot(aes(x = returns, fill = asset)) +
  geom_histogram(alpha = 0.15, binwidth = .01) +
  geom_histogram(data = portfolio_returns_tq_rebalanced_yearly, fill = "cornflowerblue") +
  ggtitle("Portfolio and Asset Monthly Returns Since 2005")
```



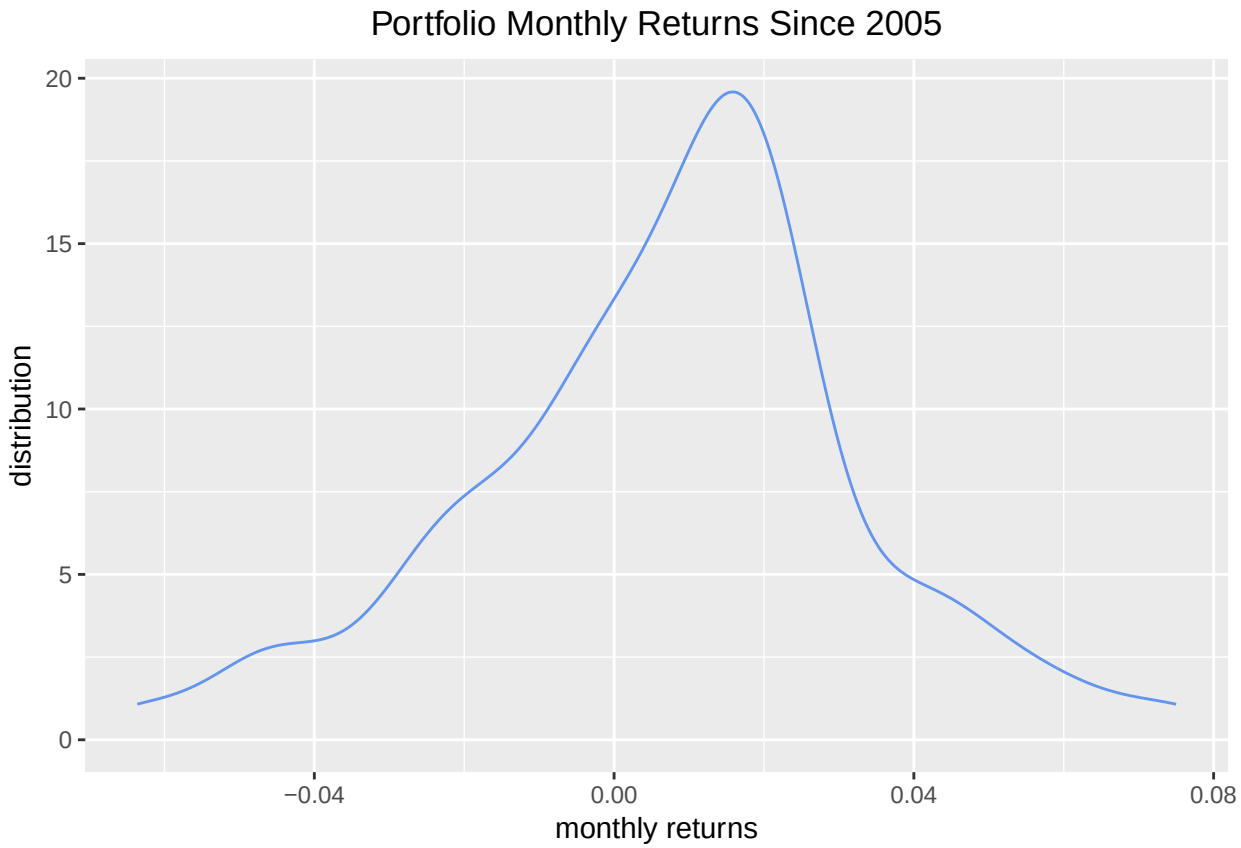
Let's turn to a new chart format and build a scatterplot of portfolio returns. I would like to see the returns over time so will put the date on the x-axis with `ggplot(aes(x = date))`. We put monthly returns on the y-axis with `geom_point(aes(y = returns), color = "cornflowerblue")`.

```
portfolio_returns_tq_rebalanced_yearly %>%
  ggplot(aes(x = date)) +
  geom_point(aes(y = returns), color = "cornflowerblue")
```



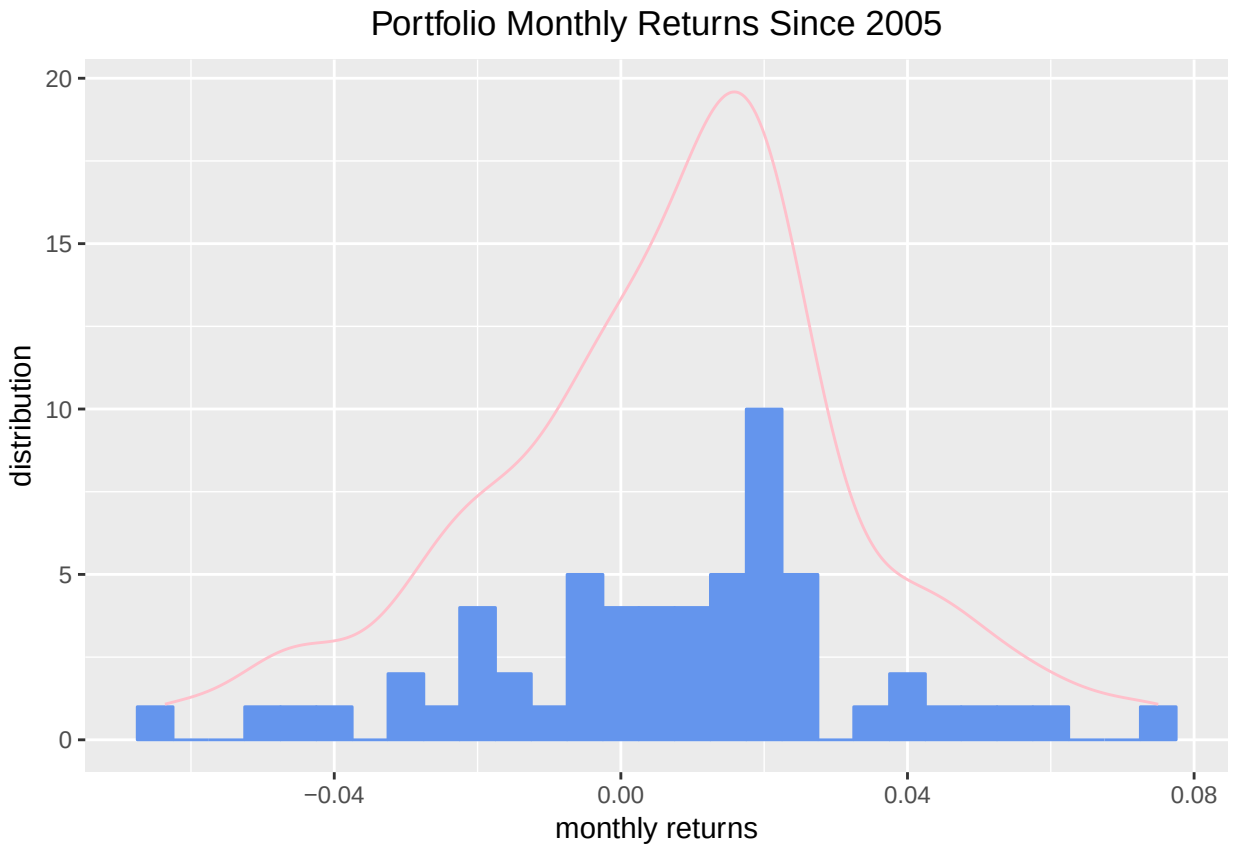
Maybe we don't want to use a histogram or scatterplot, but instead want to use a density line to visualize the portfolio returns distributions. We can use the `stat_density(geom = "line", alpha = 1)` function to do this. The `alpha` argument is selecting a line thickness. Let's also add a label to the x and y axis with the `xlab` and `ylab` functions.

```
portfolio_returns_tq_rebalanced_yearly %>%
  ggplot(aes(x = returns)) +
  stat_density(geom = "line", alpha = 1, colour = "cornflowerblue") +
  ggtitle("Portfolio Monthly Returns Since 2005") +
  xlab("monthly returns") +
  ylab("distribution")
```



Now let's put portfolio returns histogram and density on one plot. We do that by layering our geoms. First we call `geom_histogram(binwidth = .005, colour = "cornflowerblue", fill = "cornflowerblue")` then we add another layer with `stat_density(geom = "line", alpha = 1, color = "pink")`.

```
portfolio_returns_tq_rebalanced_yearly %>%
  ggplot(aes(x = returns)) +
  geom_histogram(binwidth = .005, colour = "cornflowerblue", fill = "cornflowerblue") +
  stat_density(geom = "line", alpha = 1, color = "pink") +
  ggtitle("Portfolio Monthly Returns Since 2005") +
  xlab("monthly returns") +
  ylab("distribution")
```



Summation:

Our first Shiny app

For all the reasons discussed in the introduction, a Shiny application is flexible, useful and powerful way to share our work. Let's get started in the Shiny world building app to display portfolio returns.

We want to empower an end user to do the following:

1) choose tickers and portfolio weights
2) choose a start date
3) choose a rebalancing frequency
4) calculate portfolio returns
5) visualize the portfolio returns on a scatterplot, histogram and density chart

The final app and full source code can be seen here:

www.reproduciblefinance.com/shiny/returns-distribution/

The application encompasses much of our work thus far as it requires importing daily price data, converting to monthly log returns, assigning portfolio weights, calculating portfolio returns, and visualizing with `ggplot`. This makes our work more flexible since the user can construct any 5-asset portfolio for which there's data in our data source. And, the number 5 is for illustrative purposes. Our app could easily support 50 assets (though consider the user experience there - will anyone manually enter 50 ticker symbols?).

Let's get to the code.

We will use Rmarkdown to build our Shiny applications by inserting into the yaml `runtime: shiny`. This will alert the server (or our laptop) that this is an interactive document. The yaml also gives us a space for the title.

```
---
title: "Returns Shiny"
runtime: shiny
output:
  flexdashboard::flex_dashboard:
    orientation: rows
---
```

As with other R scripts, we'll need to load the necessary packages.

```
library(tidyverse)
library(highcharter)
library(tidyquant)
library(timetk)
```

Alright, our first task is to build an input sidebar and enable users to choose five stocks and weights. We will use `textInput("stock1",...))` to create a space where the user can type a stock symbol and we will use `numericInput("w1",...)` to create a space where the user can enter a numeric weight. We want those entry spaces to be on the same line or 'row' so we will nest them inside of a call to `fluidRow()` Since we have 5 stocks and weights, we repeat this 5 times. Notice that the stock symbol field uses `textInput()` because the user needs to enter text and the weight field uses `numericInput()` because the user needs to enter a number.

```
fluidRow(
  column(6,
  textInput("stock1", "Stock 1", "SPY")),
  column(5,
  numericInput("w1", "Portf. %", 25, min = 1, max = 100))
)

fluidRow(
  column(6,
  textInput("stock2", "Stock 2", "EFA")),
  column(5,
  numericInput("w2", "Portf. %", 25, min = 1, max = 100))
)

fluidRow(
  column(6,
  textInput("stock3", "Stock 3", "IJS")),
  column(5,
  numericInput("w3", "Portf. %", 20, min = 1, max = 100))
)

fluidRow(
  column(6,
  textInput("stock4", "Stock 4", "EEM")),
  column(5,
  numericInput("w4", "Portf. %", 20, min = 1, max = 100))
)

fluidRow(
  column(6,
  textInput("stock5", "Stock 5", "AGG")),
  column(5,
  numericInput("w5", "Portf. %", 10, min = 1, max = 100))
```

```
)
```

It's worth a second look at this code to make sure it's clear becuase we will be reusing it verbatim and making no apologies. Let's dissect one of those fluid rows line-by-line.

`fluidRow()` creates the row. `column(6...)` creates a column for our stock ticker input with a length of 6. `textInput("stock1", "Stock 1", "SPY"))` creates our first text input field. We called it `stock1` which means it will be referenced in downstream code as `input$stock1`. We labeled it with "Stock 1", which is what the end user will see when viewing the app. Finally we set "SPY" as the default initial value.

We also want a row where the user can choose a start date with `dateInput("date", "Starting Date", "2010-01-01", format = "yyyy-mm-dd"))`.

```
fluidRow(
  column(7,
  dateInput("date", "Starting Date", "2010-01-01", format = "yyyy-mm-dd"))
)
```

The `dateInput()` is also quite important as we will use it in our future Shiny apps.

Finally, let's give the user the ability to rebalance the portfolio at different intervals. We will use `selectInput("rebalance", "rebal freq", c("Yearly" = "years", "Monthly" = "months", "Weekly" = "weeks"))` to create a drop down for the user.

```
fluidRow(
  column(6,
  selectInput("rebalance", "rebal freq",
              c("Yearly" = "years",
                "Monthly" = "months",
                "Weekly" = "weeks"))
  )
)
```

Finally, we include a `submit` button for our end user. This button is what takes all those inputs and passes them on to our reactive functions so the Shiny engine can start doing its work. The app won't fire until the user clicks submit.

```
actionButton("go", "Submit")
```

This is a hugely important button because it enables the use of `eventReactive()` to control our computation. That first `eventReaactive()` is where we take the user-chosen stocks and grab their daily prices.

The code should look very familiar from our previous work, except it depends on inputs from the user for ticker symbols, weights and starting date.

```
portfolio_returns_byhand <- eventReactive(input$go, {

  symbols <- c(input$stock1, input$stock2, input$stock3, input$stock4, input$stock5)

  prices <- getSymbols(symbols, src = 'yahoo', from = input$date,
             auto.assign = TRUE, warnings = FALSE) %>%
  map(~Ad(get(.))) %>%
  reduce(merge) %>%
    `colnames<-`(symbols)

  w <- c(input$w1/100, input$w2/100, input$w3/100, input$w4/100, input$w5/100)

  asset_returns_long <-
      prices %>%
```

```
      to.monthly(indexAt = "last", OHLC = FALSE) %>%
      tk_tbl(preserve_index = TRUE, rename_index = "date") %>%
      gather(asset, returns, -date) %>%
      group_by(asset) %>%
      mutate(returns = (log(returns) - log(lag(returns))))

  portfolio_returns_byhand <-
    asset_returns_long %>%
    tq_portfolio(assets_col = asset,
                 returns_col = returns,
                 weights = w,
                 col_rename = "returns")

})
```

We now have an object called `portfolio_returns_byhand()` and we can pass that object to our downstream code chunks. In fact, our substantive work has been completed. What's left is to display the distributions of portfolio returns, which we did in our previous work by passing a dataframe object to `ggplot`.

Shiny works in a similar way but it also uses a custom function for building reactive charts called `renderPlot()`. By including `renderPlot()` in the code chunks, we are alerting the app that a reactive plot is being built, one that will change when the an upstream reactive or input changes. In this case, the plot will change when the user clicks 'submit' and fires off the `eventReactive()`.

After calling `renderPlot()`, we use `ggplot()` to create a scatter plot, a histogram and a density chart of monthly returns. These will be nested in different tabs so the user can toggle between them and choose which is most helpful. That might a bit hard to envision so I recommend a quick look at the app:

www.reproduciblefinance.com/shiny/returns-distribution/

The flow for these 3 `ggplot` code chunks is going to be the same: call the reactive function `renderPlot()`, pass in`portfolio_returns_byhand()`, call `ggplot()` with an `aes(x = ...)` argument and then choose the appropriate `geom_`. The specifics of the `geom_` and other aesthetics are taken straight from our previous visualizations.

Here is the scatterplot code chunk

```
renderPlot({
portfolio_returns_byhand() %>%
  ggplot(aes(x = date)) +
  geom_point(aes(y = returns), color = "cornflowerblue") +
  ylab("percent monthly returns")
})
```

Here is the histogram code chunk.

```
renderPlot({
  portfolio_returns_byhand() %>%
    ggplot(aes(x = returns)) +
    geom_histogram(alpha = 0.25, binwidth = .01, fill = "cornflowerblue")
})
```

And finally here is the density chart code chunk.

```
renderPlot({
  portfolio_returns_byhand() %>%
    ggplot(aes(x = returns)) +
    stat_density(geom = "line", size = 1, color = "cornflowerblue")
```

```
})
```

Again, the final app and full source code can be seen here:

www.reproduciblefinance.com/shiny/returns-distribution/

This Shiny app is a good way to take all that grinding we did on portfolio returns and allow an end user to apply it to a custom portfolio.

Growth of a dollar

We have covered the process of importing daily price data for 5 assets, converting to monthly log returns for those assets, and then converting to portfolio returns after assigning weights to those assets. Next we want to convert those portfolio returns to the growth of a dollar over time, so that each month's observation is not a monthly return but rather how a dollar invested would have grown cumulatively.

One method for this calculation is to add 1 to our monthly returns and take the cumulative product:

```
portfolio_growth_byhand <- cumprod(1 + portfolio_returns_byhand$returns)
```

That `cumprod(1 + returns)` is what the built-in functions will be handling for us under the hood but we will go through them all to be sure they match and discover new code flows.

Let's turn to the `xts` world, where our first method is translate directlt from asset returns to portfolio growth with the same `Return.portfolio()` function as we used to calculate portfolio returns. We pass in the weights vector as before, but we include the argument `wealth.index = 1`. This tells the function to calculate the growth of a dollar, as if our wealth started at $1 invested.

```
portfolio_growth_xts_skip_step <-
  Return.portfolio(asset_returns_xts,
                   wealth.index = 1, weights = w, rebalance_on = "months") %>%
  `colnames<-`("growth")
```

The second method makes use of the portfolio returns object that we calculated earlier. We take the object `portfolio_returns_xts_rebalanced_monthly` and pass it to `Return.portfolio()`. We don't need to supply weights but we do again set `wealth.index = 1`.

```
portfolio_growth_xts_rebalanced_monthly <-
  Return.portfolio(portfolio_returns_xts_rebalanced_monthly,
                   wealth.index = 1) %>%
  `colnames<-`("growth")
```

These two methods yield the same result for the growth of a dollar in our portfolio.

```
tail(portfolio_growth_xts_skip_step)
```

```
##              growth
## 2017-07-31 1.392620
## 2017-08-31 1.393579
## 2017-09-29 1.430683
## 2017-10-31 1.456039
## 2017-11-30 1.478898
## 2017-12-21 1.480737
```

```
tail(portfolio_growth_xts_rebalanced_monthly)
```

```
##              growth
## 2017-07-31 1.392620
## 2017-08-31 1.393579
```

```
## 2017-09-29 1.430683
## 2017-10-31 1.456039
## 2017-11-30 1.478898
## 2017-12-21 1.480737
```

At first glance, the first method might seem to be better. It is more concise because we skip the step of converting asset returns to portfolio returns. But that skipping has a cost because we no longer have an object of monthly portfolio returns, and that means we cannot visualize those returns or calculate their standard deviations, skewness etc.

Maybe that's not important to our project, or maybe it's crucial. Either way, be aware that skipping the step of calculating monthly asset returns means we can't work with those returns.

Tidy portfolio growth

On to a tidyquant method for calculating growth of a dollar. Very similar to the `xts` method, we can go direct from asset returns to portfolio growth with the following code chunk. We use the `tq_portfolio()` function, pass it `asset_returns_long`, along with a weights vector and `wealth.index = 1`.

```
portfolio_growth_tq_rebalanced_monthly <-
  asset_returns_long %>%
  tq_portfolio(assets_col  = asset,
               returns_col = returns,
               weights     = w,
               col_rename  = "growth",
               rebalance_on = "months",
               wealth.index = 1)
```

That method again skipped our important step of first finding monthly returns. Let's combine the tidyverse and tidyquant for a code flow still relies on those monthly returns. We will use `mutate()` from the `dplyr` package and create a new column called 'growth with `mutate(growth = cumprod(1 + returns))`.

```
portfolio_growth_tidy <-
  portfolio_returns_tq_rebalanced_monthly %>%
  mutate(growth = cumprod(1 + returns)) %>%
  select(-returns)
```

Have a look at our results thus far.

```
tail(portfolio_growth_byhand)
```

```
##             returns
## 2017-07-31 1.392620
## 2017-08-31 1.393579
## 2017-09-29 1.430683
## 2017-10-31 1.456039
## 2017-11-30 1.478898
## 2017-12-21 1.480737
```

```
tail(portfolio_growth_tidy)
```

```
## # A tibble: 6 x 2
##         date   growth
##       <date>    <dbl>
## 1 2017-07-31 1.392620
## 2 2017-08-31 1.393579
## 3 2017-09-29 1.430683
## 4 2017-10-31 1.456039
```

```
## 5 2017-11-30 1.478898
## 6 2017-12-21 1.480737
```

```
tail(portfolio_growth_xts_rebalanced_monthly)
```

```
##              growth
## 2017-07-31 1.392620
## 2017-08-31 1.393579
## 2017-09-29 1.430683
## 2017-10-31 1.456039
## 2017-11-30 1.478898
## 2017-12-21 1.480737
```

Before we move on to Shiny, let's explore one more method using the `purrr` package from the tidyverse.

`purrr` contains a lot of useful functions and the `map()` family is the most commonly used, but another powerful function is `accumulate()`. As the name implies, this function allows us to accumulate calculations in a recursive way, meaning we can use prior calculations the same way we would with `cumprod()`. We will make use of this when we get to Monte Carlo simulations and now let's use `accumulate()` to get the growth of a dollar.

```
portfolio_growth_purrr <-
  portfolio_returns_tq_rebalanced_monthly %>%
  mutate(growth_1 = accumulate(1 + returns, `*`),
         growth_2 = accumulate(1 + returns, function(x, y) x * y)) %>%
  select(date, growth_1, growth_2)
```

Note that we used `accumulate()` in two ways: `accumulate(1 + returns, *)` and `accumulate(1 + returns, function(x, y) x * y)`. That is two new ways to calculate growth but in general it demonstrates a nice way to introduce a new package to our team's toolkit.

When we want to introduce a new function or package to our team's work, most important is to explicitly flag it and explain it. Without that, our collaborators might miss the new function or package altogether, especially if it is an internal package. Next, when possible, use the new tool to accomplish a task that we also have accomplished in a traditional way. Here, we introduce `accumulate()` to calculate growth of a dollar, having already done so with `cumprod()` and built-ins from `PerformanceAnalytics` and `tidyquant`. We have created a familiar environment in which to introduce a new tool and if in the future we use that tool in a complex environment (as we will when running monte carlo simulations), our collaborators will have got a prior introduction.

As a sanity check for all these methods, let's run a final comparison.

```
final_comparison <-
  merge.xts(portfolio_growth_byhand,
            portfolio_growth_xts_skip_step,
            portfolio_growth_xts_rebalanced_monthly) %>%
  tk_tbl(preserve_index = TRUE, rename_index = "date") %>%
  bind_cols(portfolio_growth_tq_rebalanced_monthly) %>%
  select(-date1) %>%
  bind_cols(portfolio_growth_tidy) %>%
  select(-date1) %>%
  bind_cols(portfolio_growth_purrr) %>%
  select(-date1) %>%
  `colnames<-`(c("date", "by_hand", "xts1", "xts2", "tq", "tidy",  "purrr1", "purrr2"))


head(final_comparison)
```

```
## # A tibble: 6 x 8
##         date   by_hand      xts1      xts2        tq      tidy    purrr1
##       <date>     <dbl>     <dbl>     <dbl>     <dbl>     <dbl>     <dbl>
## 1 2013-02-28 0.9991304 0.9991304 0.9991304 0.9991304 0.9991304 0.9991304
## 2 2013-03-28 1.0177766 1.0177766 1.0177766 1.0177766 1.0177766 1.0177766
## 3 2013-04-30 1.0387681 1.0387681 1.0387681 1.0387681 1.0387681 1.0387681
## 4 2013-05-31 1.0332076 1.0332076 1.0332076 1.0332076 1.0332076 1.0332076
## 5 2013-06-28 1.0094968 1.0094968 1.0094968 1.0094968 1.0094968 1.0094968
## 6 2013-07-31 1.0510584 1.0510584 1.0510584 1.0510584 1.0510584 1.0510584
## # ... with 1 more variables: purrr2 <dbl>
```

```
tail(final_comparison)
```

```
## # A tibble: 6 x 8
##         date  by_hand     xts1     xts2       tq     tidy   purrr1
##       <date>    <dbl>    <dbl>    <dbl>    <dbl>    <dbl>    <dbl>
## 1 2017-07-31 1.392620 1.392620 1.392620 1.392620 1.392620 1.392620
## 2 2017-08-31 1.393579 1.393579 1.393579 1.393579 1.393579 1.393579
## 3 2017-09-29 1.430683 1.430683 1.430683 1.430683 1.430683 1.430683
## 4 2017-10-31 1.456039 1.456039 1.456039 1.456039 1.456039 1.456039
## 5 2017-11-30 1.478898 1.478898 1.478898 1.478898 1.478898 1.478898
## 6 2017-12-21 1.480737 1.480737 1.480737 1.480737 1.480737 1.480737
## # ... with 1 more variables: purrr2 <dbl>
```

7 routes to the same results for dollar growth! A lot of grinding but should any of our colleagues wish to reproduce, resuse or extend our results, a plethora of code paths is available to them. Why might those paths be important? On a team of 10 R/finance ninjas, there's a good chance that one coder will prefer the `xts` world, one will prefer tidyquant and/or one will prefer the tidyverse and hand-rolled functions. It's not the case that every document needs to include all these different paths but it's good practice to have one file in the team's library that can be used as a global reference point. If language analogies are appealing, we can think of this as a Rosetta Script or where to turn when someone wonders what's the equivalent of this analytical path using another set of packages.

Beyond the flexibility of different code paths, the various object structures lend themselves to different visualizations techniques, which is a reason that some coders might prefer different paths. If `highcharter` is your axe, `xts` can be very appealing indeed.

Visualize portfolio growth

We start again in the `highcharter` world. The first two lines shoule like familiar from our previous charts. Let's add more aesthetics.

I want a title and add it with `hc_title(text = "Growth of a Dollar")` and the y-axis label to be on left-hand side with a `$` sign. We add that with `hc_yAxis(title = list(text = "growth of dollar"), opposite = FALSE, labels = list(format = "${value}")`.

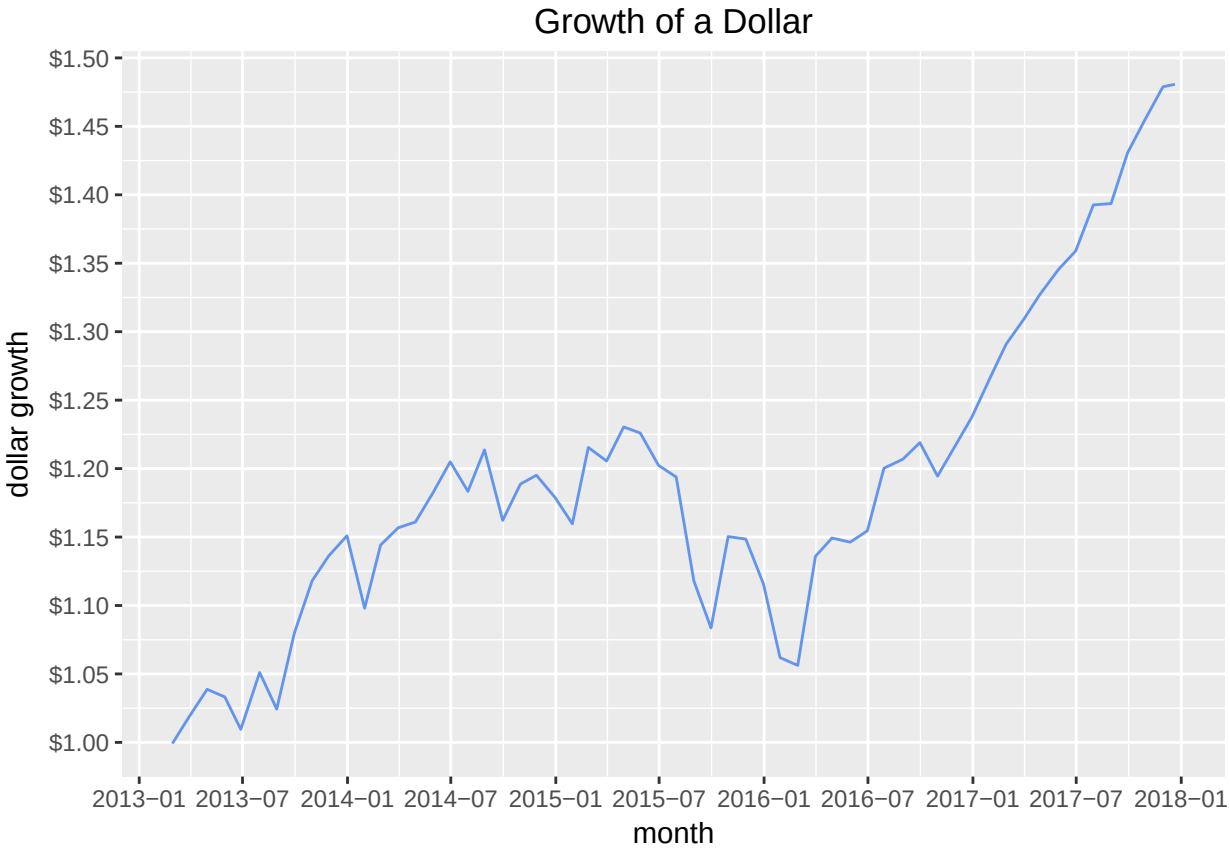
```
highchart(type = "stock") %>%
  hc_add_series(portfolio_growth_xts_rebalanced_monthly,
                name = "Portfolio", color = "cornflowerblue", lineWidth = 1) %>%
  hc_title(text = "Growth of a Dollar") %>%
  hc_yAxis(title = list(text = "growth of dollar"),
           opposite = FALSE,
           labels = list(format = "${value}")) %>%
  hc_add_theme(hc_theme_flat()) %>%
  hc_navigator(enabled = FALSE) %>%
  hc_scrollbar(enabled = FALSE)
```

That's a nice and efficient chart. An end user can quickly discern what would have happened to a dollar over time, how the financial crisis affected the portfolio and how things have recovered since.

We can also use `ggplot()` for time series plotting if we wish to stay in the tidy world.

We start with `portfolio_growth_purrr`, and set our `date` column as the x-axis and `growth_1` as the y-axis by calling `ggplot(aes(x = date, y = growth_1))`. We will add a `$` sign to the y-axis label with `scale_y_continuous(breaks = pretty_breaks(n = 10), labels = dollar)` which requires loading the `scales` package.

```
library(scales)
portfolio_growth_purrr %>%
  ggplot(aes(x = date, y = growth_1)) +
  geom_line(colour = "cornflowerblue") +
  ylab("dollar growth") +
  xlab("month") +
  ggtitle("Growth of a Dollar") +
  scale_y_continuous(breaks = pretty_breaks(n = 10), labels = dollar) +
  scale_x_date(breaks = pretty_breaks(n = 10))
```



The end result with `ggplot` or `highcharter` does not vary a lot though highcharter does offer a bit more interactivity and the built-in date selection buttons.

Shiny Growth of a Dollar

Now we want to port our Dollar Growth work to a Shiny application so that an end user is able to:

1) choose tickers and portfolio weights
2) choose a start date
3) choose a rebalancing frequency
4) chart the growth of a dollar in the portfolio since the chosen start date

The final app can be seen here with full source code:

www.reproduciblefinance.com/shiny/portfolio-growth/

The input sidebar is identical to that of our app on returns distribtution We let the user choose 5 ticker symbols, 5 weights, a start date and rebalance period. The user then clicks 'submit' to fire up the reactives. Since it's identical, we're not going to review it here.

We will, though, review the difference substantive flow that is used to get portfolio growth. We use an `eventReactive()`, pass in tickers, weights, start date and frequency, but instead of calculating portfolio returns, we calculate portfolio growth. It's not a copy paste from our first app, but it's a similar structure. Since we use `highcharter` in this app, we go with the `xts` world and use `Return.portfolio(asset_returns_xts, wealth.index = 1,...)` - this is the same method we used in our previous growth calculation.

```
portfolio_growth_xts <- eventReactive(input$go, {

  symbols <- c(input$stock1, input$stock2, input$stock3, input$stock4, input$stock5)
```

```
  prices <- getSymbols(symbols, src = 'yahoo', from = input$date,
             auto.assign = TRUE, warnings = FALSE) %>%
  map(~Ad(get(.))) %>%
  reduce(merge) %>%
  `colnames<-`(symbols)

  w <- c(input$w1/100, input$w2/100, input$w3/100, input$w4/100, input$w5/100)

  prices_monthly <- to.monthly(prices, indexAt = "last", OHLC = FALSE)

  asset_returns_xts <- na.omit(Return.calculate(prices_monthly, method = "log"))

  portfolio_growth_xts <-
    Return.portfolio(asset_returns_xts,
                     wealth.index = 1,
                     weights = w,
                     rebalance_on = input$rebalance) %>%
    `colnames<-`("growth")

})
```

At this point, portfolio growth is calculated and stored as `portfolio_growth_xts()` and our substantive work has been completed. Now we want to display the chart of the portfolio growth over time. Outside of Shiny, this would be a simple passing of the xts object to `highcharter`.

As with `ggplot`, Shiny uses a custom function for building reactive highcharter charts called `renderHighchart()`. Once we invoke that `renderHighchart()`, our code looks very similar to our previous visualization work as we use `hc_add_series(portfolio_growth_xts(), name = "Dollar Growth", color = "cornflowerblue")` to add our portfolio growth xts object to a chart.

```
renderHighchart({

  highchart(type = "stock") %>%
  hc_title(text = "Growth of a Dollar") %>%
  hc_add_series(portfolio_growth_xts(), name = "Dollar Growth", color = "cornflowerblue") %>%
  hc_navigator(enabled = FALSE) %>%
  hc_scrollbar(enabled = FALSE)
})
```

Next, we use `ggplot()` to create the same visual as above but in the tidy world. The code flow is quite similar to how we would normally create a line chart, except we first need to convert our `xts` object to a tibble with `tk_tbl(preserve_index = TRUE, rename_index = "date")`. Then we make our call to `ggplot(aes(x = date))` and add a geom with `geom_line(aes(y = growth), color = "cornflowerblue")`.

Note that the `renderPlot()` function is playing the same role as `renderHighchart()` above - it is alerting the Shiny app that a reactive plot is forthcoming after user inputs, instead of a static plot that is unchanging.

```
renderPlot({
portfolio_growth_xts() %>%
    tk_tbl(preserve_index = TRUE, rename_index = "date") %>%
    ggplot(aes(x = date) +
    geom_line(aes(y = growth), color = "cornflowerblue") +
    ylab("dollars") +
    ggtitle("Growth of Dollar over time")
})
```