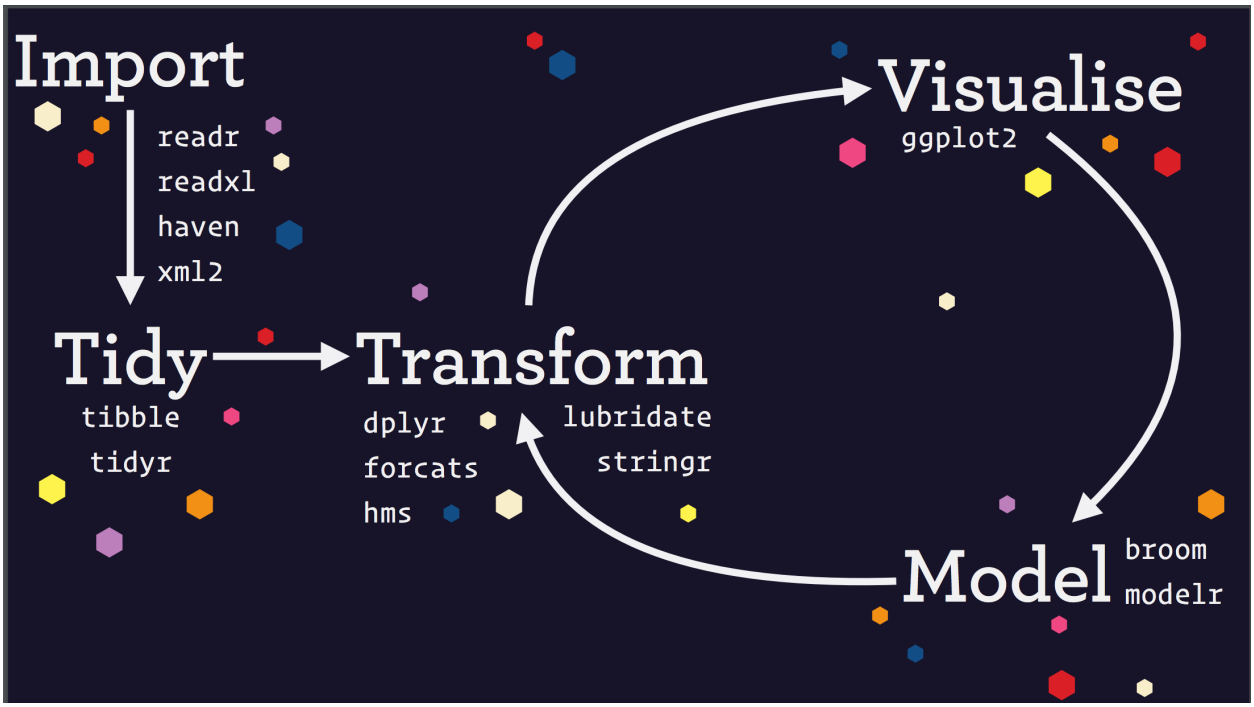
Transform Asset Prices to Log Returns

Today, we go back a bit, back to where we probably should have started in the first place, but it wouldn't have been as much fun. In our previous work on volatility, we zipped through the steps of importing data and transforming to monthly returns. If you're familiar with Garrett and Hadley's data science paradigm pictured below, we skimmed through the data import, tidying and transformation steps so that we could look at volatility.



Let's correct that oversight and do some spade work on transforming daily prices to monthly log returns. To map back to the paradigm, grabbing the daily prices is our data import, and putting to log monthly returns is our tidying and transforming.

We will use a few packages and for completeteness, we will cover several paths to getting our desired end result. Obviously it's not necessary to utilize all of our paths here. The intention is to show several methods so that the best one can be chosen for a given challenge or project.

Let's load up our packages.

```
library(tidyverse)
library(tidyquant)
library(timetk)
library(tibbletime)
```

We are building toward analyzing the returns of 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 eventually build a Shiny app to let users choose different assets and see different

results. Today, we are just going to work with the individual prices/returns of those 5 assets.

Importing the data

On to step 1, wherein we import adjusted prices for the 5 ETFs to be used in our porftolio and save them to an `xts` object called `prices`.

We need a vector of ticker symbols that we will then pass 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 need.

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 objects. We could stop here and have the right substance, but the format wouldn't be great as we would have a `list` of 5 adjusted prices. The `map` function returns a list by default but I find them to be unwieldy.

The `reduce(merge)` function will allow us to merge the lists into one object and coerce back to an `xts` structure. Finally, we want intuitive column names and use `colnames<-` to rename the columns. The `rename` function from `dplyr` will not work well here because the object structure is still `xts`.

```
# The symbols vector holds our tickers.
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 = "2005-01-01",
             auto.assign = TRUE, warnings = FALSE) %>%
  map(~Ad(get(.))) %>%
  reduce(merge) %>%
  `colnames<-`(symbols)
```

The XTS World

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
## 2005-01-03 92.93649 37.27123 50.03726 17.75115 66.46513
## 2005-01-04 91.80085 36.55673 49.14001 17.20462 66.40022
## 2005-01-05 91.16737 36.53330 48.22606 16.99400 66.37427
## 2005-01-06 91.63092 36.53330 48.51403 16.98245 66.41973
## 2005-01-07 91.49957 36.36932 47.88804 17.01533 66.40022
## 2005-01-10 91.93220 36.53330 48.30119 17.03666 66.36779
```

Our first reading is from January 3, 2005 (the first trading day of that year) and we have daily prices. Let's stay in the`xts`world and convert to monthly prices using a call to `to.monthly(prices, indexAt = "last", OHLC = FALSE)` from `quantmod`. 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
```

```
## 2005-01-31 91.28327 36.82613 49.17757 17.84268 66.78971
## 2005-02-28 93.19141 38.21999 50.60481 19.57025 66.54166
## 2005-03-31 91.48669 37.21735 49.41060 18.02219 65.89336
## 2005-04-29 89.77267 36.61529 46.88079 17.79736 67.02862
## 2005-05-31 92.66556 36.29904 49.76660 18.35988 67.58337
## 2005-06-30 92.80595 36.81910 51.71521 19.08859 68.17554
```

We now have an `xts` object, and we have moved from daily prices to monthly prices.

Now we'll 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
## 2005-02-28  0.020687956  0.037151134  0.02860907  0.09241749 -0.003720764
## 2005-03-31 -0.018462043 -0.026583839 -0.02388165 -0.08240683 -0.009790539
## 2005-04-29 -0.018912912 -0.016308976 -0.05255702 -0.01255376  0.017081924
## 2005-05-31  0.031716308 -0.008674618  0.05973616  0.03111783  0.008242225
## 2005-06-30  0.001513936  0.014225464  0.03840776  0.03892308  0.008724035
## 2005-07-29  0.037547524  0.029527500  0.05677115  0.07400828 -0.010408397
```

From a subtantive perspective, we could stop right now. Our task has been accomplished: we have imported daily prices, trimmed to adjusted prices, moved to monthly prices and transformed to monthly log returns.

If we wished to visualize these log returns, we would go straight to `highcharter` and pass in th various `xts` series.

For now, though, let's take a look at a few more methods.

The Tidyverse and Tidyquant World

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.

For our first method, we use `dplyr` and `timetk` to convert our object from an `xts` object of prices to a `tibble` of monthly returns. Once we convert to a `tibble` the tidyverse is available for all sort of manipulations.

Let's step through the logic before getting to the code chunk.

In the piped workflow below, our first step is to use the `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 prices 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 to align with the `symbols` vector. That's important because when we build a portfolio, we will use that vector to coordinate our different weights.

```
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)
```

For our second method in the tidy world, 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 `prices` `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)
```

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)
```

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

```
head(asset_returns_xts)
```

```
##                   SPY          EFA         IJS         EEM          AGG
## 2005-02-28  0.020687956  0.037151134  0.02860907  0.09241749 -0.003720764
```

```
## 2005-03-31 -0.018462043 -0.026583839 -0.02388165 -0.08240683 -0.009790539
## 2005-04-29 -0.018912912 -0.016308976 -0.05255702 -0.01255376  0.017081924
## 2005-05-31  0.031716308 -0.008674618  0.05973616  0.03111783  0.008242225
## 2005-06-30  0.001513936  0.014225464  0.03840776  0.03892308  0.008724035
## 2005-07-29  0.037547524  0.029527500  0.05677115  0.07400828 -0.010408397
```

```
head(asset_returns_dplyr_byhand)
```

```
## # A tibble: 6 x 6
##         date          SPY          EFA         IJS         EEM
##       <date>        <dbl>        <dbl>       <dbl>       <dbl>
## 1 2005-01-31           NA           NA          NA          NA
## 2 2005-02-28  0.020687956  0.037151134  0.02860907  0.09241749
## 3 2005-03-31 -0.018462043 -0.026583839 -0.02388165 -0.08240683
## 4 2005-04-29 -0.018912912 -0.016308976 -0.05255702 -0.01255376
## 5 2005-05-31  0.031716308 -0.008674618  0.05973616  0.03111783
## 6 2005-06-30  0.001513936  0.014225464  0.03840776  0.03892308
## # ... 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 2005-01-31 -0.017948838 -0.012013934 -0.01733026  0.005143039
## 2 2005-02-28  0.020687956  0.037151134  0.02860907  0.092417491
## 3 2005-03-31 -0.018462043 -0.026583839 -0.02388165 -0.082406831
## 4 2005-04-29 -0.018912912 -0.016308976 -0.05255702 -0.012553757
## 5 2005-05-31  0.031716308 -0.008674618  0.05973616  0.031117830
## 6 2005-06-30  0.001513936  0.014225464  0.03840776  0.038923082
## # ... 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 2005-01-31  0.000000000  0.000000000  0.00000000  0.00000000
## 2 2005-02-28  0.020687956  0.037151134  0.02860907  0.09241749
## 3 2005-03-31 -0.018462043 -0.026583839 -0.02388165 -0.08240683
## 4 2005-04-29 -0.018912912 -0.016308976 -0.05255702 -0.01255376
## 5 2005-05-31  0.031716308 -0.008674618  0.05973616  0.03111783
## 6 2005-06-30  0.001513936  0.014225464  0.03840776  0.03892308
## # ... with 1 more variables: AGG <dbl>
```

First, have a look at the left most column in each object, where the date is stored. The `asset_returns_xts` has a date index, not a column. 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, notice the first observation in each of the objects. Only `asset_returns_tqtbl` has an observation for January of 2005. The other 3 elided January of 2005 when we cast from daily to monthly. Do we care? Does it matter to our final calculations? Perhaps and probably. Either way, we need to be aware of the consequences of taking our raw data and reshaping it via different methods.

Third, each of these objects is in "wide" format, which in this case means there is a column for each of our assets. When we called `spread` at the end of the piped code flow, we put the data frames back to wide format.

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

head(asset_returns_long_format)
```

```
## # A tibble: 6 x 3
##         date asset      returns
##       <date> <chr>        <dbl>
## 1 2005-01-31   SPY           NA
## 2 2005-02-28   SPY  0.020687956
## 3 2005-03-31   SPY -0.018462043
## 4 2005-04-29   SPY -0.018912912
## 5 2005-05-31   SPY  0.031716308
## 6 2005-06-30   SPY  0.001513936
```

We now have 3 columns and if we want to run an operation like `mutate` on each asset, we just need to call `goup_by(asset)` in the piped flow.

Before we conclude, I would be remiss without calling out an important point that I skipped for the sake of brevity. We transformed daily prices to monthly log returns, but never explained, justified or mentioned why we chose log returns instead of simple returns.

We are making an assumption that our colleagues, collaborators or clients understand and agree with that decision. Even if they happen to do so, we should explain our transformation as a matter of best practice: justify the logic for data transformations, even if those transformations represent a standard operating procedure. Else, we might have a team that starts to skip the logic when the data transformations are a bit more non-standard, or even controversial. Since we might perform this prices to log returns in several projects, we could simply have one R Notebook that our team references to explain why we use log returns. Over time, we can build a repository of documents that explain the theory behind our data tidying and transformation practices.

That's all for today. Next time we will visualize these individual asset returns (and will notice that our log returns have a normal distribution, which will be important when we run simulations) before combining them into portfolio returns based on the different weightings. Thanks and see you next time.