returns-shiny-writeup

Now we want to port our 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) visualize the portfolio returns on a scatterplot, histogram and density chart

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, on the substance - the stuff an end user is going to see.

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. Let's have a look at that first `eventReaactive()` wherein 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 the 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 othe aesthetics are taken straight from our previous visualizations.

```r
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.

```r
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.

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