

# Quantitative Equity Research Handbook

---

## **Jason Hsu, Ph.D.**

UCLA Anderson School of Management

Rayliant Global Advisors, HK

Research Affiliates, LLC USA

## **Vivek Viswanathan**

UC Irvine Merage School of Management

Rayliant Global Advisors, HK

## **Special thanks to:**

Guanxi Yi (Rayliant Global Advisors)

Michael Wang (Rayliant Global Advisors)

Donald He (Allianz Global Investors)

Edward Sheng (Pacific Life)

Max Jiang (USC)

# Chapter 1: Building Simple Portfolios

---

Before we build complicated strategies, we need to start with the basic tools of portfolio construction. In this chapter, we will show you how to download monthly and daily return data from Wharton Research Data Services (WRDS, pronounced “words”), which your university should have access to<sup>1</sup>. Then, we will perform simple data plots and calculate basic statistics to understand the nature of individual stock returns. We will proceed to construct capitalization-weighted and equally-weighted portfolios along with their returns. Finally, we will create performance tables to see how these simple portfolios have performed.

## Downloading and Using CRSP Data

### Why CRSP?

We will show you how to download the U.S. stock return data from the Center for Research in Security Prices (CRSP, pronounced “crisp”). We will download this data from WRDS, which is a large data repository, which includes CRSP and many other data sources. Before that, we should discuss why we are using CRSP and not Yahoo! Finance, Google Finance, or Bloomberg. The most critical reason is to avoid survivorship bias. That is, companies that go bankrupt will be removed from Yahoo! Finance, Google Finance, or Bloomberg. Try finding Worldcom (ticker: WCOM) on Yahoo! Finance. You won’t be able to find it, because it was delisted in 2002. On CRSP, you will find Worldcom data up until it was delisted.

Survivorship bias is a problem because you only see the firms that did not delist due to bankruptcy, being bought out, or not meeting the requirements of the exchange. On average, delisted stocks underperformed those that did not. As one might expect, we cannot know for certain which stock will delist beforehand. Computing portfolio returns using only stocks that never delisted will exaggerate the returns tremendously. If we want to compute the 10-year return of the top 500 stocks by market capitalization in the year 2000, and we only include stocks that lasted until 2010, we would exclude Worldcom, Enron, and Lehman Brothers, who were all delisted by 2010.


Luckily, CRSP, Datastream, and other data providers will have data free of survivorship bias. You will find all of your favorite delisted companies and a corresponding delisting return, which we will discuss at length later in this chapter.


---


<sup>1</sup> If you are currently working, you may not have access to WRDS but may have access to CRSP data through an internal database. Depending on how your firm formats its data, you may need to play around with the code that we give you here. If you do not have access to CRSP at all, you may have access to another returns database like Datastream. You can certainly use that but some of the code will need to be changed significantly. Finally, if you do not have access to a returns database, much of this code will be difficult to implement

## Using WRDS to Download CRSP Data

Login to WRDS. You will see a list of subscriptions, one of which should be CRSP. Feel free to dig through the datasets that you have access to. There is a multitude of interesting information contained in WRDS. For now, though, click on CRSP.

 Your Subscriptions

 Not Subscribed

 Your Queries

» Bank Regulatory	» Fama French & Liquidity Factors	» PHLX
» Blockholders	» Federal Reserve Bank	» Public
» CBOE Indexes	» Hedge Fund Research (HFR)	» RavenPack News Analytics Trial
» Compustat - Capital IQ	» IBES	» Research Quotient
» CRSP	» IHS Global Insight	» SEC Order Execution
» CUSIP	» ISS (formerly RiskMetrics)	» TAQ
» DMEF Academic Data	» ISSM	» Thomson Reuters
» Dow Jones	» Option Metrics	» TRACE
» Event Study by WRDS	» OTC Markets	
» Factset	» Penn World Tables	

Click on Stock / Security Files under Annual Update, though if you have access to quarterly or monthly updated data, feel free to use that instead.

### Annual Update

Databases in this section are updated once each year, in early February. Update schedules should not be confused with end-of-day, end-of-month, or end-of-quarter data such as stock prices.

» Stock / Security Files	» Index / Treasury and Inflation	» Ziman REIT
» Stock / Events	» Index / CRSP Select Series	» Tools
» Stock / Portfolio Assignments	» CRSP/Compustat Merged	» Stock-1962 / Security Files
» Index / Stock File Indexes	» Treasuries	» Stock-1962 / Events
» Index / Cap-Based Portfolios	» Treasury / Daily	
» Index / S&P 500 Indexes	» Treasury / Monthly	

Now, click on “Monthly Stock File.” Generally monthly data is a bit easier to work with, so we will stick to that for now. However, we will also download daily data for the computation of volatility and beta, for which daily data provides more precise estimation than monthly data. By the way, note that we will use volatility and standard deviation interchangeably.

## Stock / Security Files

---

Monthly Stock File

Stock Market Indexes

Daily Stock File

Stock Header Info

If you haven't used CRSP before, the next page will look extremely intimidating as it did to all of us when we first looked at the page. The page is a large list of returns, price, and identification data that you may be interested in. Note that financial data (e.g. net income, book value, cash flow) is missing from this list. Financial data can be found in Compustat. We will download financial data from Compustat in the next chapter. First, we will choose our start and end date. We will include the data just from 1950 onwards.

### Step 1: Choose your date range.

Date range

1950-01

to

2015-12

Step 2 asks how we would like to search the dataset. We want to search the entire database, so skip the question about formatting our company codes and click on "Search the Entire Database."

## Step 2: Apply your company codes.

☒ TICKER ☐ PERMNO ☐ PERMCO ☐ CUSIP ☐ NCUSIP ☐ HSICCD ☐ SICCD

Select an option for entering company codes

☒  ☐

*Please enter Company codes separated by a space.  
Example: IBM MSFT AAPL [ [Code Lookup](#) ]* *Save code list to Saved Codes*

☐  No file selected

*Upload a plain text file (.txt), having one code per line.*

☐  ▼

*Choose from your saved codelists.*

☒

*This method allows you to search the entire database of records. Please be aware that this method can take a very long time to run because it is dependent upon the size of the database.*

The conditional statements below are optional but we do want to set a restriction with respect to Share Codes. We only want common shares—not deposit receipts or certificates. Set the first drop-down box to shrcd, set the second box to “=” and enter number 10 in the third box. Do the same for the second row except set the third column to 11. Share codes of 10 and 11 are common shares of stocks. Last, but certainly not least, select the “OR” radio button, not the “AND” one!

## Conditional Statements (Optional)

*How does this work?*

▼

☐ AND ☒ OR

▼

Now, we select identifiers. We are going to choose several. CUSIP and ticker may help us merge with other databases down the road. Note a company’s CUSIP or ticker may change. Next, we need the Exchange Code for reasons that we will discuss later. (NYSE stocks are on average

higher market cap than AMEX and NASDAQ stocks and therefore have slightly different characteristics.) Lastly, we want the SIC code which we will use to classify sectors.

### Step 3: Query Variables.

*How does this work?*

The screenshot shows a query builder interface with a top navigation bar and a main selection area. The navigation bar has four tabs: 'Search All' (5/62), 'Identifying Information' (5/20), 'Time Series Information' (0/11), and 'Share Information' (with a right arrow). The 'Identifying Information' tab is active and highlighted with a red border. Below the navigation bar, there are two columns: 'Select' and 'Selected'. The 'Select' column has a header 'Select' with a checked 'All' button and a count of '(15)'. It lists 15 variables with radio buttons and help icons. The 'Selected' column has a header 'Selected' with an unchecked 'Clear All' button and a count of '(5)'. It lists 5 selected variables with checked green icons. A red box highlights the 'Selected' column.

Select	Selected
<input type="radio"/> Ncusip	<input checked="" type="checkbox"/> Cusip
<input type="radio"/> CRSP Permanent Company Number	<input checked="" type="checkbox"/> Ticker
<input type="radio"/> Share Code	<input checked="" type="checkbox"/> Exchange Code
<input type="radio"/> Share Class	<input checked="" type="checkbox"/> SIC Code
<input type="radio"/> Nasdaq Issue Number	<input checked="" type="checkbox"/> Company Name
<input type="radio"/> Header Exchange Code	
<input type="radio"/> Header SIC Code	
<input type="radio"/> Header SIC Major Group	
<input type="radio"/> Header SIC Industry Group	
<input type="radio"/> Names Ending Date	
<input type="radio"/> Trading Symbol	

For time series, information, we want price, holding period return, and holding period return without dividends. Holding period return is total return (including dividends) while holding period return without dividends is price return (only the capital gains component of return). To further clarify, total return is the return you would earn if you continuously reinvested your dividends into a stock. Price return is the return you would earn if you completely ignore dividends. We will use price for calculating market capitalization. Note that the formula for market capitalization is:

$$\text{market capitalization} = \text{price} \times \text{shares outstanding}$$

### Step 3: Query Variables.

*How does this work?*

Q Search All 8/62

Identifying Information 5/20

Time Series Information 3/11

Share Information >>

Select ☒ All (8)

Selected ☐ Clear All (8)

<input type="radio"/> Price Alternate ?	<input checked="" type="checkbox"/> Cusip
<input type="radio"/> Price Alternate Date ?	<input checked="" type="checkbox"/> Ticker
<input type="radio"/> Ask or High ?	<input checked="" type="checkbox"/> Exchange Code
<input type="radio"/> Bid or Low ?	<input checked="" type="checkbox"/> SIC Code
<input type="radio"/> Closing Bid ?	<input checked="" type="checkbox"/> Company Name
<input type="radio"/> Closing Ask ?	<input checked="" type="checkbox"/> Price
<input type="radio"/> Share Volume ?	<input checked="" type="checkbox"/> Holding Period Return
<input type="radio"/> Spread Between Bid and Ask ?	<input checked="" type="checkbox"/> Holding Period Return without Dividends

Now, we need shares outstanding since it's the second piece of the formula for market capitalization.

### Step 3: Query Variables.

*How does this work?*

Search All 9/62		Identifying Information 5/20		Time Series Information 3/11		Share Information >>	
Select <input checked="" type="checkbox"/> All (2)						Selected <input type="checkbox"/> Clear All (9)	
<input type="radio"/>	Shares Observation End Date ?	<input checked="" type="checkbox"/>	Cusip				
<input type="radio"/>	Share Flag ?	<input checked="" type="checkbox"/>	Ticker				
		<input checked="" type="checkbox"/>	Exchange Code				
		<input checked="" type="checkbox"/>	SIC Code				
		<input checked="" type="checkbox"/>	Company Name				
		<input checked="" type="checkbox"/>	Price				
		<input checked="" type="checkbox"/>	Holding Period Return				
		<input checked="" type="checkbox"/>	Holding Period Return without Dividends				
		<input checked="" type="checkbox"/>	Number of Shares Outstanding				

Next we need delisting returns. Delisting returns are returns that are earned from delisting. For example, if a \$1 stock is delisted when another firm buys it for \$1.10 per share, the delisting return will be 10%. If a \$10 stock delists because it went bankrupt and its equity holders were completely wiped out, its delisting return would be -100%. We need to choose both the delisting return for total return and delisting return without dividends for price return.



### Step 3: Query Variables.

*How does this work?*

The screenshot shows a web-based query builder interface with four tabs: 'Series Information 3/11', 'Share Information 1/3', 'Delisting Information 2/8' (highlighted with a red border), and 'Distribution Information'. The 'Delisting Information' tab contains two columns: 'Select' and 'Selected'. The 'Select' column has 6 items, each with a radio button and a question mark icon. The 'Selected' column has 11 items, each with a green checkmark icon. The last two items in the 'Selected' column, 'Delisting Return' and 'Delisting Return without Dividends', are enclosed in a red rectangular box.

Select	(6)	Selected	(11)
<input type="radio"/> Delisting Code	?	<input checked="" type="checkbox"/> Cusip	
<input type="radio"/> New CRSP Permno	?	<input checked="" type="checkbox"/> Ticker	
<input type="radio"/> Date of Next Available Information	?	<input checked="" type="checkbox"/> Exchange Code	
<input type="radio"/> Amount After Delisting	?	<input checked="" type="checkbox"/> SIC Code	
<input type="radio"/> Delisting Price	?	<input checked="" type="checkbox"/> Company Name	
<input type="radio"/> Date of Delisting Payment	?	<input checked="" type="checkbox"/> Price	
		<input checked="" type="checkbox"/> Holding Period Return	
		<input checked="" type="checkbox"/> Holding Period Return without Dividends	
		<input checked="" type="checkbox"/> Number of Shares Outstanding	
		<input checked="" type="checkbox"/> Delisting Return	
		<input checked="" type="checkbox"/> Delisting Return without Dividends	

We are done with data items, so skip the remaining boxes and scroll to the bottom of the page. We need to set our data format. We will be using SAS, so choose SAS Windows\_32 dataset. If you're using the 64-bit version of SAS, choose SAS Windows\_64. You can choose to compress or not based on the speed of your Internet connection. The default date format is fine. Save the query to myWRDS as "Monthly Stock Returns" or whatever catches your fancy. Finally, click on the "Submit Query" button and wait a few moments for the data to finish loading.

#### Step 4: Select query output.

Select the desired [format](#) of the output file. For large data requests, select a compression type to expedite downloads. If you enter your email address, you will receive an email that contains a [URL](#) to the output file when the data request is finished processing.

##### Output Format

- ☐ fixed-width text (\*.txt)
- ☐ comma-delimited text (\*.csv)
- ☐ Excel spreadsheet (\*.xlsx)
- ☐ tab-delimited text (\*.txt)
- ☐ HTML table (\*.htm)
- ☒ SAS Windows\_32 dataset (\*.sas7bdat)
- ☐ SAS Windows\_64 dataset (\*.sas7bdat)
- ☐ SAS Solaris\_64 dataset (\*.sas7bdat)
- ☐ dBase file (\*.dbf)
- ☐ STATA file (\*.dta)
- ☐ SPSS file (\*.sav)

##### Compression Type

- ☒ None
- ☐ zip (\*.zip)
- ☐ gzip (\*.gz)

##### Date Format

- ☒ YYMMDDn8. (e.g. 19840725)
- ☐ DATE9. (e.g. 25JUL1984)
- ☐ DDMMYY6. (e.g. 250784)
- ☐ MMDDYY10. (e.g. 07/25/1984)
- ☐ DDMMYY10. (e.g. 25/07/1984)
- ☐ YYMMDDs10. (e.g. 1984/07/25)

E-Mail Address *(Optional)*

[Edit Preferences](#)

Custom Field *(Optional)*

☒ Save this query to myWRDS[Submit Query](#)

Once your data has finished loading, you should see this:

## Data Request Summary

Data Request ID	b2f386b64e81c424
Libraries/Data Sets	crspa/msf /
Frequency/Date Range	mon / 01Jan1950 - 31Dec2014
Search Variable	TICKER
Input Codes all item(s)	-all-
Conditional Statements	shrcd eq 10 or shrcd eq 11
Output format/Compression	saswin64 /
Variables Selected	CUSIP COMNAM TICKER EXCHCD SICCD PRC RET RETX SHROUT DLRET DLRETX
Extra Variables and Parameters Selected	

Your output is complete. Click on the link below to open the output file.

[qb2f386b64e81c424.sas7bdat](#) (441.1 MB, 3203710 observations 14 variables)

[Download instructions](#)

Internet Explorer and Firefox users... Right-click and select "Save Target As..."

Save the file and unzip it if necessary. Rename the SAS data file to something meaningful like "crsp\_monthly\_raw.sas7bdat." We will save it and read it from a folder called "C:\ Data" but feel free to place it anywhere.

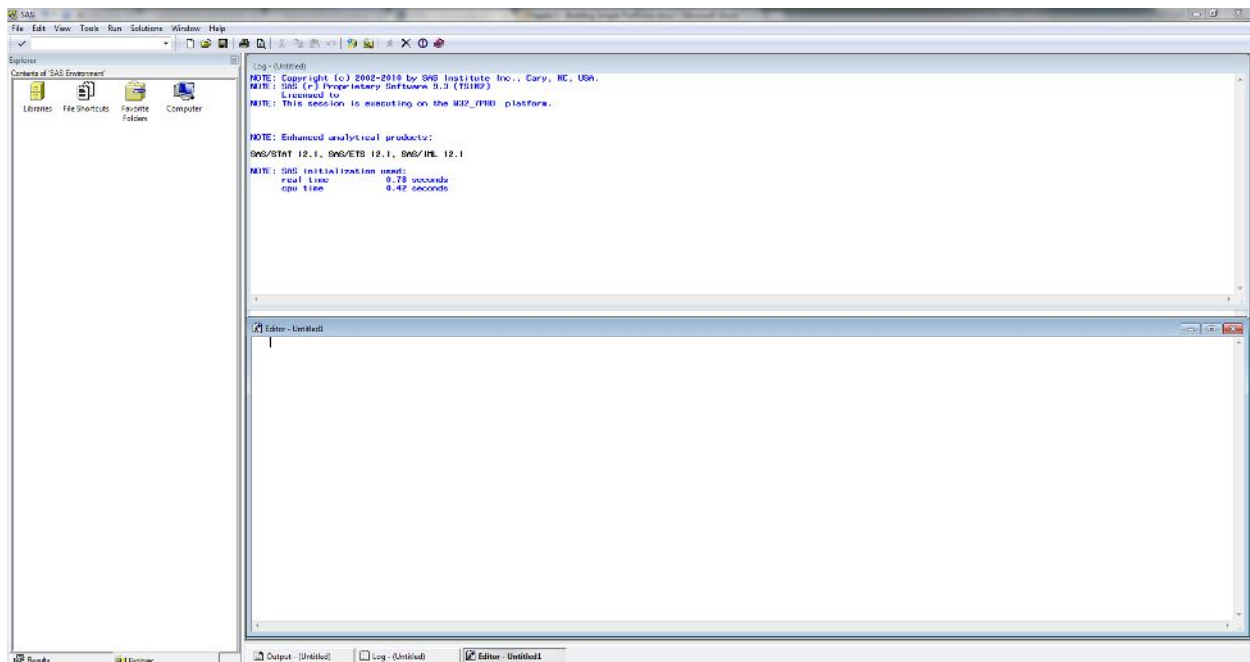
Now, do exactly the same thing for daily data. We will use this for calculating volatility and beta. This file is truly huge—about 9 GB—and will take a longer time to run. If you do not have the hard drive space to store it, skip it. You can calculate a half-decent volatility or beta estimate using monthly data. While you wait for the data to run, continue with the following sections, which use only monthly data.

## Stock / Security Files

<a href="#">Monthly Stock File</a>	<a href="#">Stock Market Indexes</a>
<a href="#">Daily Stock File</a>	<a href="#">Stock Header Info</a>

## Getting Started With SAS

First, we want to provide the barest bones introduction to SAS. Open up SAS. You should see something like this. If you are running SAS Enterprise Edition, your screen will look significantly different.



The explorer which will eventually show our datasets and working data are to the left. The log is the window on top. The code editor is at the bottom. For now, we are going to dive right in. Click on the editor and type the following:

```
* point SAS to the folder in which our CRSP returns file is stored.;
libname retdata "C:\data";
```

The `libname` statement associates a directory (`C:\data`) with a name (`retdata`), so that you can access datasets from that directory easily. We chose the name `retdata`, but you can choose any name that is 8 characters or less.

The asterisk denotes a comment. You end a comment and every line of code with a semicolon. Comments help you and other coders understand your code, but they are not *strictly* necessary. However, code without comments or with too many useless comments can be unreadable. You should always keep three things in mind when it comes to readability:

- 1) Comment your code such that people can get a quick sense of what your code does even without carefully reading through your code.
- 2) Add comments to lines that are particularly complicated or are likely to confuse others who are not familiar with the code.
- 3) Name variables and datasets intuitively.

Run this code by hitting F3 or by clicking on the picture of the running stick figure.



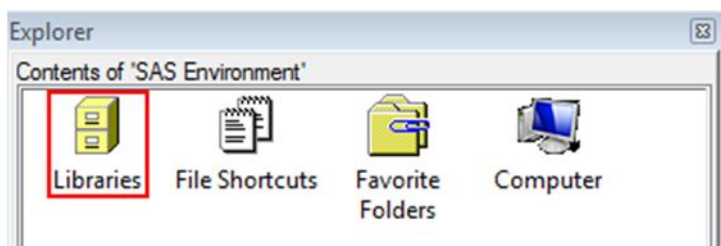
Our log should now say this:

```
1 * point SAS to the folder in which our CRSP returns file is stored.;
2 libname retdata "C:\data";
NOTE: Libref RETDATA was successfully assigned as follows:
      Engine:          V9
      Physical Name:    C:\data
```

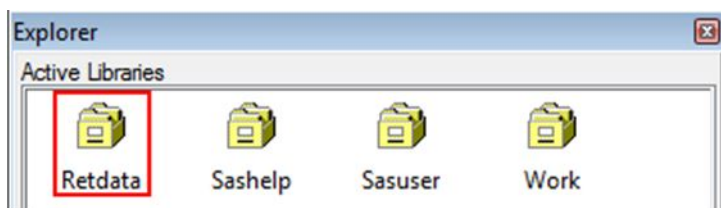
Our library `retdata` has been successfully created. You can always look at the log to ensure that code has run successfully. If you see red text, there was an error in the code and SAS was unable to run it. For example:

```
3 libname "C:\data";
ERROR: "C:\data" is not a valid SAS name.
ERROR: Error in the LIBNAME statement.
```

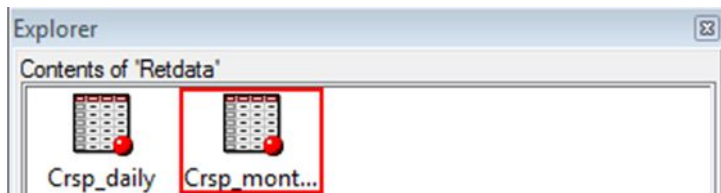
If you see red text in your log file, you know something is wrong. Read the error to figure out what the problem is. Now that our library is created, let's look at what datasets are present in the Explorer. Look at the Explorer on the left and double-click on Libraries.



Now, you will see all the active libraries. `Sashelp`, `Sasuser`, and `Work` are always there. The `Work` library is very important as it stores all of your working datasets. Everything in `Work` library is deleted every time you close SAS. The library we added is `Retdata`. Double-click on it.



You should at least see `crsp_monthly` if not `crsp_daily`. Double-click on `crsp_monthly`.



You should see a lot of data. Feel free to scroll through it a bit. Note some of the returns are letters. Note the frequent negative prices, which are bid-ask averages instead of trading prices. We'll explain this more later. Also, note the occasional delisting return, and the fact that there are no returns when there are delisting returns present.

Look at the column labels at the top. You will see PERMNO, Names Date, and so on. While these labels help us get a sense of what each column is, this is not how we will refer to the columns in our code. Click on View on the menu bar and click Column Names. Now you should see the column names as we will refer to them in the code. For example, instead of "Company Name," you should see "COMNAM." These variables are not case-sensitive, so you can refer to "COMNAM" as "comnam" (or "cOmNAM" for that matter) in your code. In short, labels are meant to be human-readable, while names are meant to be SAS-readable. It is good practice to add labels to data sets. We will do that periodically throughout this text.

Let us continue with our code. Click on Untitled1 on the bottom and type the code below.

```
* read the returns file and adjust variables;
data stockReturns;
    set retdata.crsp_monthly;
run;
```

This is what is known as a **data** step. SAS has **data** steps, which manipulate data, and **proc** steps which perform various procedures on data. We will discuss the latter more later. The line **data** stockReturns; creates a new dataset in the `Work` library called `stockReturns`. The `Work` library is the default library if you do not specify a library. Our code above is exactly equivalent to this:

```
* read the returns file and adjust variables;
data work.stockReturns;
    set retdata.crsp_monthly;
run;
```

The next line `set retdata.crsp_monthly;` sets the input to our new dataset `stockReturns` to the dataset that we downloaded. The `retdata` part specifies the library while `crsp_monthly` is the name of the dataset. If you don't specify a library, SAS assumes you are trying to access the `Work` library.

If you're used to coding in other languages, coding in SAS can be a bit strange. What we're doing here is creating a new dataset (`stockReturns`) based on another dataset (`retdata.crsp_monthly`) and will be manipulating the data within that dataset with additional code, which we will include shortly. This is the essence of the **data** step. You will become very familiar with it over the course of this book and your coding career.

Open `stockReturns` in the `Work` library and confirm that it looks exactly like `retdata.crsp_monthly`. To get to the `Work` library, click on the Explorer window and hit the "Up One Level" button on the toolbar.



Double-click on the `Work` library, and then double-click on `stockReturns`. You should see something that looks identical to `retdata.crsp_monthly`. Note that you cannot manipulate a dataset when it is open in the Explorer. If you want to test this out, try running the code again when `stockReturns` is open. The log will show an error. Note that you can run this code just fine if `retdata.crsp_monthly` is open, since we are only reading from this dataset, not writing to it.

## Basic Calculations

Let's play around a bit with our data. Add a line computing monthly dividend yields to our code.

```
* point SAS to the folder in which our CRSP returns file is stored.;
libname retdata "C:\data";

data stockReturns;
    set retdata.crsp_monthly;

    * calculate monthly dividend yield;
    div=ret-retx;
run;
```

Recall that `ret` is total return and `retx` is price return. The code above creates a new column `div` and sets it equal to the difference between the total return and price return. If we wanted to calculate dividend yield geometrically, we could do the following:

```
* point SAS to the folder in which our CRSP returns file is stored.;
libname retdata "C:\data";

data stockReturns;
    set retdata.crsp_monthly;

    * calculate monthly dividend yield;
    div=(1+ret)/(1+retx)-1;
run;
```

Imagine you wanted to calculate squared return, which is often used in calculating payouts on variance swaps (though on a daily and not monthly frequency). You could create a column called `uncenteredvar` that is equal to the square price return. Note that the sign for multiplication in SAS is `*`, while the sign for exponentiation is `**`. The sign for exponentiation in SAS is not `^`, which denotes logical NOT.

```
data stockReturns;
    set retdata.crsp_monthly;

    * calculate monthly dividend yield;
    div=(1+ret)/(1+retx)-1;
```

```

    * calculate monthly uncentered variance;
    uncenteredvar=retx**2;
run;

```

## Filtering

SAS provides many tools for filtering our data. In particular, we are going to look at keeping data given a condition, deleting data given a condition, and outputting data to a new dataset given a condition. First, let's look at selecting the data that we want.

```

* keep returns if year is 1985;
data stockReturns1985;
    set stockReturns;

    if year(date)=1985 then output;
run;

```

We use the `year` function to select *just* the year part of the date and we add those lines of data to `stockReturns1985` if the year is equal to 1985. Note that the `then output` is optional.

```

* keep returns if year is 1985;
data stockReturns1985;
    set stockReturns;

    if year(date)=1985;
run;

```

In addition, we can also use `where` instead of `if`.

```

* keep returns if year is 1985;
data stockReturns1985;
    set stockReturns;

    where year(date)=1985;
run;

```

There are differences between `where` and `if`, but we don't need to explore those at the moment, and for this particular application, they behave identically. Imagine instead that we wanted to get rid of stock returns that are from 1985. We can use the `delete` keyword.

```

* keep returns if year is not 1985;
data stockReturnsNot1985;
    set stockReturns;

    if year(date)=1985 then delete;
run;

```

Lastly, we might want to create 2 data sets: a 1985 dataset and a non-1985 dataset. We can do that in one step using `output`.

```

* split data into 1985 and not 1985;
data stockReturns1985 stockReturnsNot1985;

```



```

    set stockReturns;

    if year(date)=1985 then output stockReturns1985;
    else output stockReturnsNot1985;
run;

```

Note the if-else statement with which you may be familiar from other programming languages. Now, that we have these basics down, let's jump into some practical data manipulation.

## Dropping, Keeping, Renaming, and Relabeling Variables

You often want to drop, keep, rename, or relabel variables, and SAS makes it easy for you to do so. First, imagine that for some strange reason, we want to drop price and shares outstanding.

We can do the following:

```

* drop price and shares outstanding variables;
data stockReturnsNoPrcShare;
    set stockReturns(drop=prc shrout);
run;

```

To accomplish the same thing we can do this:

```

* drop price and shares outstanding variables;
data stockReturnsNoPrcShare(drop=prc shrout);
    set stockReturns;

run;

```

Or this:

```

* drop price and shares outstanding variables;
data stockReturnsNoPrcShare;
    set stockReturns;

    drop prc shrout;
run;

```

(Note the lack of an equal sign in the last list.) These all drop the variables `prc` and `shrout`. However, they also have functional differences. The first drops the variables immediately. This means you can't run calculations using `prc` or `shrout`. It's useful in terms of efficiency if you are working on very large dataset and you don't want to read irrelevant data from file into the data step. The second and third drop the variables at the end, so you can perform calculations using `prc` or `shrout`. In particular, both of following will run fine:

```

data stockReturnsNoPrcShare(drop=prc shrout);
    set stockReturns;

    mktcap=prc*shrout;
run;

data stockReturnsNoPrcShare;

```

```

    set stockReturns;

    drop prc shrout;
    mktcap=prc*shrout;
run;

```

Note that the last step runs perfectly fine even though `drop prc shrout` is before `mktcap=prc*shrout`. But the following will produce blank market caps:

```

* WILL NOT WORK;
data stockReturnsNoPrcShare;
    set stockReturns(drop=prc shrout);

    mktcap=prc*shrout;
run;

```

This occurs because the drop occurs before the data is even read in. The opposite of `drop` is `keep`. If we want to keep only returns, dates, and PERMNOs, we could write:

```

* keep returns, dates, and PERMNO;
data stockReturnsSparse(keep=ret retx date PERMNO);
    set stockReturns;
run;

```

We can similarly place the `keep` statement in the `set` statement and as a separate line. Lastly, we can rename variables.

```

* rename ret as TR and retx as PR;
data stockReturnsRenamed;
    set stockReturns;
    rename ret=TR;
    rename retx=PR;
run;

```

You can put these in the `data` and `set` lines as well. For example, putting it in the `data` line looks like this:

```

data stockReturnsRenamed(rename=(ret=TR retx=PR));
    set stockReturns;
run;

```

You can also label variables and remove labels from variables as follows:

```

* label ret as "Total Return" and retx as "Price Return" and remove label
from date;
data stockReturns;
    set stockReturns;
    label ret="Total Return";
    label retx="Price Return";
    label date=" ";
run;

```

Setting a blank label removes the label such that its label is simply its variable name. Now that we have covered some basics, let's focus on cleaning our data.

## Adjusting the Raw Data

We wish we could immediately start running portfolios, but there is some cleaning that needs to be done. This is always frustrating since you're adjusting a bunch of data you have just downloaded and probably don't have intuition yet, but such is life. Here is what we need to do:

- 1) When delisting returns are available, replace returns with delisting returns. (Note that when a firm delists, its returns are blank during that month, so we replace those blank returns with delisting returns.)
- 2) Remove any lines with blank or invalid returns. CRSP places delist codes in the middle of their returns. These don't represent actual returns. They just signify that the firm is delisting.
- 3) Ensure that all prices are positive. This seems like a strange one, but if a price of a security was not available, CRSP uses the average of the bid and ask of the stock, and then puts a negative sign in front of it so that you know it's derived from the bid-ask and not an actual trade. For our purposes, we don't care about this distinction between bid-ask and last trade. We just want prices to be positive!

First, open a new file and start from scratch. The code we wrote in the previous section was just for practice. Now, we are getting into useful code. The following may seem intimidating at first, but we will go through it line-by-line.

```
* point SAS to the folder in which our CRSP returns file is stored.;
libname retdata "C:\data";

* read the returns file and adjust variables;
data stockReturns;
    set retdata.crsp_monthly;

    * extract only year and month from date;
    year=year(date);
    month=month(date);

    * add delist returns;
    if missing(dlret)=0 then ret=dlret;
    if missing(dlretx)=0 then retx=dlretx;

    * only keep meaningful returns;
    if ret>=-1;

    * force price to be positive;
    prc=abs(prc);

    * drop the delist returns;
    drop dlret dlretx date;
```

```
run;
```

The two lines after `set retdata.crsp_monthly` extract the year and month components from the date.

```
* extract only year and month from date;
year=year(date);
month=month(date);
```

Since this is monthly data, having the day data that is embedded in a date is unnecessary. Indeed, breaking out the year and month separately will make our lives a bit easier at times. Now, let's look at the next two lines.

```
* add delist returns;
if missing(dlret)=0 then ret=dlret;
if missing(dlretx)=0 then retx=dlretx;
```

Note that `ret` is total returns while `retx` is price returns. The `if` statement says, "If delist returns are not missing, then set return equal to delist returns." The `missing` function returns 1 if a variable is missing and 0 if a variable is present. Recall that when we looked at the dataset, every time delist returns were present, returns were missing, so there's nothing sketchy about completely replacing returns with delist returns whenever delist returns are present.

Recall that we had letters in place of returns or sometimes missing returns. How do we get rid of these in one fell swoop? We are going to only include returns that are greater than or equal to -1, which will kick out all of the missing data and all of the letters. This is kind of a trick that we're pulling but it happens to work and it is explicable. If the variable `ret` is a letter or missing, the condition `ret>=-1` will evaluate to false (represented by a 0 in SAS).

```
* only keep meaningful returns;
if ret>=-1;
```

Recall that if you don't specify a then clause to the `if` statement, the language assumes that you want to keep the lines of data that meet the if clause's criteria. Last, we want to set price equal to the absolute value of itself.

```
* force price to be positive;
prc=abs(prc);
```

As you can guess, the `abs` function takes the absolute value of its argument. Now that we are done with the two delist total returns and delist price returns, we can safely drop them from our dataset and save ourselves a small amount of hard drive space and processing time for further steps. The drop statement can also be appended after the Data or Set statement.

```
* drop the delist returns;
drop dlret dlretx;
```

Finally, you need the run statement to tell SAS to run any previously entered statements. While you don't need to put it after every single step, we recommend that you do since it makes running blocks of code considerably easier. This entire block of code is called a data step. There are two types of steps in SAS, a data step and a proc step. The data step inputs data while the proc step executes a procedure. Don't think too hard about this right now, since it's sufficiently fuzzy that it is better to learn the differences by example.

Now, hit F3 to run the code. Look at the log to make sure the code ran without a hitch. It should look something like this:

```
NOTE: Missing values were generated as a result of performing an operation on missing values.
      Each place is given by: (Number of times) at (Line):(Column).
      17428 at 42:9
NOTE: There were 3202267 observations read from the data set RETDATA.CRSP_MONTHLY.
NOTE: The data set WORK.STOCKRETURNS has 3106811 observations and 13 variables.
NOTE: DATA statement used (Total process time):
      real time          1.35 seconds
      cpu time           1.12 second
```

Go back to the explorer and go up one level by clicking on the folder icon with an arrow.



Now double-click on the Work directory. You should see our new dataset `stockReturns`. Double-click on it and behold! The missing returns and delist returns are gone. Prices are all positive.

Now, we're ready to calculate some simple portfolios. Close the `stockReturns` tab or you won't be able to make changes to it. Save the code file as "`ew_returns.sas`" as we're going to use it to calculate equally-weighted portfolio returns. Feel free to call it something else if you don't like this name.

## Creating Equally-Weighted Portfolios

Our first portfolio, the equally weighted portfolio, will be simple. As the name suggests an equally-weighted portfolio gives the same weight to each stock in the portfolio such that it sums to 100%. For example, if you have 100 stocks in your portfolio, an equally weighted portfolio would assign 1% weight to each stock. We are going to build it off of the previous code that we have written. We have three tasks yet to complete.

- 1) Sort the returns by date.
- 2) Calculate equally-weighted returns.
- 3) Output the equally-weighted returns.

To sort, we will learn our first proc step, the proc sort.

```
* sort the returns by date;
```

```
proc sort data=stockReturns;
    by year month;
run;
```

This sorts the dataset `stockReturns` by a particular variable or variables. In this case, it sorts first by `year` and then by `month`. The above code sorts the date in ascending order. If you want to sort by ascending date but descending `PERMNO` instead, type this:

```
proc sort data=stockReturns;
    by year month descending PERMNO;
run;
```

If you want to remove duplicates, you can add `nodupkey` to the end of the first line:

```
* sort returns by date and PERMNO and remove duplicate observations;
proc sort data=stockReturns nodupkey;
    by year month PERMNO;
run;
```

If there are multiple observations with the same `year`, `month`, and `PERMNO`, it will remove one of those observations. There are no duplicates in our data set, so this `nodupkey` does nothing but it doesn't hurt either.

Note that if you want to run *just* this sorting step, highlight the code with your cursor and hit F3. We need to calculate equally-weighted returns. Equally-weighted returns are the easiest returns to calculate because you just take the average of all the returns. Luckily, there is a procedure in SAS that makes this very easy. It's called `proc means` and it can calculate a whole range of interesting results, which we will slowly reveal as time goes on or which you can Google if you are so inclined.

```
* calculate equally-weighted returns;
proc means data=stockReturns;
    var ret retx;
    by year month;
    output out=EWReturns mean=EWTR EWPR;
run;
```

The first line says the dataset on which we will be running `proc means`. The next line specifies which variables we will be taking the mean of. The third line shows we will be calculating the mean by each `year` and `month`. Finally, the fourth line says the output dataset will be `EWReturns` and the variables that will store the mean of `ret` will be called `EWTR` (for equally-weighted total return) and the mean of `retx` will be called `EWPR` (for equally-weighted price return). Run this code by highlighting it and hitting F3. You should see the following in the log:

```
NOTE: There were 3202267 observations read from the data set WORK.STOCKRETURNS.
NOTE: The data set WORK.EWRETURNS has 780 observations and 6 variables.
NOTE: PROCEDURE MEANS used (Total process time):
      real time          0.67 seconds
      cpu time           0.62 seconds
```

Note that you need `stockReturns` to be sorted by `year month` before running `proc means` or SAS will give you an error. If it's not correctly sorted, you will get an error that looks like this:

```
ERROR: Data set WORK.STOCKRETURNS is not sorted in ascending sequence. The current BY group has year
= 1987 and the next BY group has year = 1986.
NOTE: The SAS System stopped processing this step because of errors.
NOTE: There were 18 observations read from the data set WORK.STOCKRETURNS.
WARNING: The data set WORK.EWRETURNS may be incomplete. When this step was stopped there were 16
observations and 6 variables.
WARNING: Data set WORK.EWRETURNS was not replaced because this step was stopped.
NOTE: PROCEDURE MEANS used (Total process time):
      real time          0.01 seconds
      cpu time           0.01 seconds
```

If you see this error, make sure to sort your data by `year` and `month`. Note that when you run the code, the Results Viewer opens up with output giving the number of observations, mean, standard deviation, minimum, and maximum of our two variables for every date. If we don't want anything to be output to Results Viewer, we can change the code above by adding `noprint` to the end of the first line of `proc means`.

```
* calculate equally-weighted returns;
proc means data=stockReturns noprint;
    var ret retx;
    by year month;
    output out=EWReturns mean=EWTR EWPR;
run;
```

If instead of `EWTR` and `EWPR`, we were comfortable sticking with `ret` and `retx` for variable names for equally-weighted total return and price return, respectively, we could remove the `EWTR` and `EWPR` to get the following:

```
proc means data=stockReturns;
    var ret retx;
    by year month;
    output out=EWReturns mean=;
run;
```

Make sure you run the original code eventually, since we will be assuming that you are using the variable names `EWTR` and `EWPR`. Look at the dataset `EWReturns` in Explorer and click on View->Column Names, so that you can see the actual column names. You will note that the variable `_TYPE_` is always equal to 0. For our purposes, it will in fact always be 0 and is irrelevant to us, but you can read more about it online. The variable `_FREQ_` is the number of observations that each date has. In our case, it's just the number of stocks in each month. Finally, the two variables we care about most, `EWTR` and `EWPR`, are our equally-weighted total and price return respectively.

Lastly, we want to export our data. Let's create folder called "Output" in "C:\Data". Note that this is clearly not a good place for a folder called "Output". We are just putting it here, so we're

not demanding too many root directory folders from you. Feel free to put it somewhere else. Now, to export our returns to CSV:

```
* export the equally-weighted returns as a CSV;
proc export data=EWReturns outfile="C:/Data/Output/EWMonthlyReturns.csv"
  dbms=CSV replace;
run;
```

The `proc export` step exports the data set specified in the `data` option to the file specified in the `outfile` option in the format specified in the `dbms` option. Another popular option for `dbms` is `XLSX`, though make sure to change your `outfile` with the extension `xlsx`.

Hit F3 to run the code. We have calculated and exported equally-weighted stock returns. Open the file up and see if everything makes sense to you. This is a pretty impressive feat this early on.

Before looking at the answer below, try the following quick exercise. Calculate the arithmetic average equally-weighted monthly total and price return. Note that we're interested in the average return of `EWReturns` not `stockReturns`. Also, we are interested in the average over time, so the `by year month` is unnecessary. (Just omit that line.) Output this to a dataset called `avgEWReturns`. You can call the mean returns whatever you want.

Your code should look something like this, though you may have chosen different variable names or chosen to use `noprint`.

```
* mean equally-weighted returns;
proc means data=EWReturns;
  var EWTR EWPR;
  output out=avgEWReturns mean=meanEWTR meanEWPR;
run;
```

If we wanted to calculate volatility as well, we might make our code look as follows:

```
* average and volatility equally-weighted returns;
proc means data=EWReturns;
  var EWTR EWPR;
  output out=avgEWChar mean=meanEWTR meanEWPR stddev=voleWTR voleWPR;
run;
```

You should get something close to a monthly return of 1.20% and a monthly volatility of 5.51% depending on when you downloaded your CRSP returns.

## Creating Capitalization-Weighted Portfolios

We're going to make minor adjustments to our code above to allow it to handle market-capitalization. Before diving into the code, we are going to have a long-winded discussion about look-ahead bias and why we need to lag market capitalization in the data. Recall that market



capitalization is price times shares outstanding. Open up our dataset `crsp_monthly` or `stockReturns`. Look at the returns column, pick a row and look at the price column. That price is the price at the end of the month. That return is the return during that month. We do not know end-of-month price at the beginning of the month. If we used market capitalization at the end of the month to weight stocks at the beginning of the month, we would be weighting stocks more if the current month's return is higher. That is something we can't do in real life, but we can mistakenly do in our backtest. We are going to avoid it here, but at the end of this section, we will have you remove the lag in the data, so that you can see how different capitalization-weighted returns look when you introduce look-ahead bias into prices. Back to the code....

Before we begin, save the file as something else. Let's start a blank file. Hit CTRL+N while you are in the Editor or click on File->New Program. First, set the library as before:

```
* point SAS to the folder in which our CRSP returns file is stored.;  
libname retdata "C:\data";
```

Now, we want to make a variable that allows us to choose our weighting variable. At the very top of our code, underneath our definition of library `retdata`, add the following code:

```
* parameters;  
%let weightVariable = lagmktcap;
```

This defined a macro variable `weightVariable` that is equal to `lagmktcap`. What this means is that after this variable is defined whenever you type the following code `&weightVariable`, SAS treats it as if you had written `lagmktcap`. Why this is useful will be made clear shortly.

You might notice that there is no variable `lagmktcap` in our data set. We are about to change that. But first, let's use our same `data` step from before to create `stockReturns`.

```
* read the returns file and adjust variables;  
data stockReturns;  
    set retdata.crsp_monthly;  
  
    * extract only year and month from date;  
    year=year(date);  
    month=month(date);  
  
    * add delist returns;  
    if missing(dlret)=0 then ret=dlret;  
    if missing(dlretx)=0 then retx=dlretx;  
  
    * only keep meaningful returns;  
    if ret>=-1;  
  
    * force price to be positive;  
    prc=abs(prc);  
  
    * drop the delist returns;  
    drop dlret dlretx date;  
run;
```

Now, type the following:

```
* lag market cap;
data lagmktcap (keep=year month PERMNO lagmktcap);
    set stockReturns;

    * market cap is equal to shares outstanding times price;
    lagmktcap=shrout*prc;

    * lag by one month;
    month=month+1;
    if month=13 then do;
        month=1;
        year=year+1;
    end;
run;
```

The first line keeps `year`, `month`, `PERMNO`, and `lagmktcap`. The next line after the `set` statement calculates market capitalization. The reason we call the variable `lagmktcap` instead of `mktcap` is that we are going to lag the variable later in the step. Market capitalization is price multiplied by shares outstanding. If a firm has 1,000 shares, each worth \$50, the entire firm is worth \$50,000.

```
* market cap is equal to shares outstanding times price;
lagmktcap=shrout*prc;
```

The next part is the hardest to think through. We want to lag by one month. Why does adding one month lag instead of subtracting one month? To answer this question, first think of what we are trying to accomplish by lagging. We want to use last month's market cap in the current month. That means to merge last month's market cap to this month's returns, we have to add a month to the date of last month's market cap.

We should also note is that there is an alternative methodology to our code that *almost* works. SAS has a lag function which moves a column's values down one row leaving the first column blank. Why can't we just sort by `PERMNO`, `year`, and `month` and lag market caps by one month. That code would look as follows:

```
* sort the returns by PERMNO and then date;
proc sort data=stockReturns;
    by PERMNO year month;
run;

*lag market cap;
data stockReturns;
    set stockReturns;
    by PERMNO year month;
    lagmktcap=lag(mktcap);
    if first.PERMNO then lagmktcap=.;
run;
```

Run this code to see what it does. First, we sort by `PERMNO` *and then* by date. This means that when we lag market cap, each market cap gets pushed into the next month as desired. Then, we say if it's the first time a `PERMNO` is encountered, we set `lagmktcap` to empty. If we didn't have this line, we would take the market cap from the previous stock's last month, which is clearly not what we want. Note that we include the `by PERMNO`, before we can use `first.PERMNO`. Furthermore, we must sort by `PERMNO`, `year`, and `month`, before we can type `by PERMNO year month` in a data step.

Now, let's discuss the problem with this methodology. This will work in the vast majority of cases, but there are times when a stock delists temporarily and comes back on the exchange some months or years later. In such a case, if we use the `lag` function, we will lag that stock's market capitalization by however many months or years it was off the exchange. There is nothing *inherently* wrong with doing this, but the lagged market capitalization wouldn't always reflect the previous month's market capitalization.

Return to our previous method of lagging. If you ran the above code that uses the `lag` function, you will have to delete that sort step and data step and run the code up until the creation of the dataset `lagmktcap`. Now, we've created this dataset that we need to merge back into `stockReturns` in order to get the market caps with the returns. Put the following code right after the creation of `lagmktcap`.

```
* sort the lagged market caps by PERMNO and date;
proc sort data=lagmktcap;
    by PERMNO year month;
run;

* sort the returns by PERMNO and date;
proc sort data=stockReturns;
    by PERMNO year month;
run;

* merge lagged market caps into returns;
data stockReturns;
    merge stockReturns(in=k) lagmktcap;
    by PERMNO year month;
    if k;
run;
```

We need to sort stocks the relevant variable before merging them. Whenever `PERMNO`, `year`, and `month` match, the rows from `stockReturns` and `lagmktcap` are merged. We did not have to call the merged dataset `stockReturns`. We could have called it something else entirely. Furthermore, the `(in=k)` and `if k` ensure that all rows from `stockReturns` are included in the merged data set. Had the `if k` not been included some additional months that would include market caps but no returns would be included in our data set. Try running the code with and without the `if k` and observe the differences in number of observations.

Note that there is another way to merge involving `proc sql`, which is remarkably versatile. We will cover that later. Below, let's keep our sorting step the same and change our `proc means` step a bit.

```
* sort the returns by date;
proc sort data=stockReturns;
    by year month;
run;

* calculate returns;
proc means data=stockReturns;
    var ret retx;
    weight &weightVariable;
    by year month;
    output out=stratReturns mean=TR PR;
run;
```

With the addition of `&weightVariable`, we are weighting by `lagmktcap` when taking the mean return. Because we set the macro variable `weightVariable` to be equal to `lagmktcap`, the statement `weight &weightVariable` is exactly equivalent to `weight lagmktcap`. The benefit of using the macro variable is that you can just change your preferred weight variable at the top of the code instead of hunting through the code for whatever parameter it is you wish to change. Note that if there is a missing market capitalization or missing return for a particular observation, it will be disregarded when calculating the mean. However, SAS will loudly complain in the logs that there is missing weight data. You can see this in the log. We also changed the name of the output dataset to `stratReturns` and the mean return variables to `TR` and `PR`.

Now, we want to change our export dataset and filename as well. Change the export step to the following:

```
* export the returns as a CSV;
proc export data=stratReturns
    outfile="C:/Data/Output/&weightVariable.MonthlyReturns.csv" dbms=CSV replace;
run;
```

Look carefully at the filename and you will note that we have a period right after `&weightVariable`. That period denotes the end of the variable name. It will not show up in the filename. Instead the filename will be `lagmktcapMonthlyReturns.csv`.

What if we want to calculate equally-weighted returns? Easy! Add a line to our merge step, though technically you can add this at any point where we are writing to the dataset `stockReturns`.

```
* merge lagged market caps into returns;
data stockReturns;
    merge stockReturns(in=k) lagmktcap;
    by PERMNO year month;
```

```

        if k;
        ew=1;
run;

```

This creates a variable `ew` that is always set to 1. Now, if we set `weightVariable` to `ew` at the top of our code, we will produce an equally-weighted portfolio:

```
%let weightVariable = ew;
```

As an exercise, create a variable `mktcap`, which is the market capitalization without lag. To see how strong (not to mention nefarious) look-ahead bias is, weigh our portfolio by `mktcap` instead of `lagmktcap`. Once you are done, please read on.

We can calculate `mktcap` wherever we write to `stockReturns`. Since we threw `ew` into the merge step, we might as well put `mktcap` in there as well though you certainly don't have to.

```

* merge lagged market caps into returns and calculate current market cap;
data stockReturns;
    merge stockReturns(in=k) lagmktcap;
    by PERMNO year month;
    if k;
    ew=1;
    mktcap=shrout*prc;
run;

```

Now, at the top, we need only change `weightVariable` to `mktcap`, and we can run our code.

```

* parameters;
%let weightVariable = mktcap;

```

Using the market cap with look-ahead bias, you should earn an arithmetic mean total return of 1.6% per month. If you weight by lagged market cap as well we should since we cannot see the future, you will earn a return of 1.0% per month. This look-ahead bias increases return by 57%. Introducing look-ahead bias is probably the worst mistake you can commit when testing a strategy, and it is also remarkably easy. This is not the last time we will have to worry about look-ahead bias. Whenever you test a strategy, ask yourself, "Is my backtest only using information available to it at the time it is making its investment decision?"

Now, make sure to set `weightVariable` back to `lagmktcap`.

## Calculating Weights

We are weighing by market capitalization and equal weighting but we are not calculating the actual weights. We are just using the `weight` statement in `proc means`. But we want to calculate weights. What percentage of our capitalization-weighted portfolio is made up of AAPL in January 2014? That is the sort of information we want.

Let's continue working with our lagged market capitalization weighting code. You already have all the tools necessary to compute weights, so please try to do this on your own before reading ahead. That being said, it is a non-trivial exercise, so we will provide a few clues if you are having trouble.

Our goal is to sum the `weightVariable` every month (or from a coding perspective every month in every year). Then, we want to merge that sum back into `stockReturns`, where the merge is done by `year` and `month`. Finally, we want to create a new column `wt` (for "weight") that divides `weightVariable` by the sum of `weightVariable` in each month. If you couldn't get it the first time, try again with these clues. The answer is below.

Right after the sort by `year` and `month` and before the calculation of returns, add the following code:

```
* sum weight variable;
proc means data=stockReturns noprint;
    var &weightVariable;
    by year month;
    output out=sum&weightVariable(drop=_TYPE_ _FREQ_)
sum=sum&weightVariable;
run;

* calculate weights;
data stockReturns;
    merge stockReturns sum&weightVariable;
    by year month;
    wt=&weightVariable/sum&weightVariable;
run;
```

The new thing that we introduced was dropping variables in the `out` statement of a `proc means`. You didn't have to do this, but it is a good way to get rid of `_TYPE_` and `_FREQ_` immediately assuming you don't need it.

Try calculating weights for capitalization-weighted and equally-weighted portfolios. In any given month, all stocks should have the same weight in an equally-weighted portfolio.

## Varying Rebalance Frequency

In the capitalization-weighted and equally-weighted portfolios, how frequently did we rebalance? That is, how frequently did we trade back to target weights of capitalization or equal weights? Think about this question carefully before continuing.

The answer is that we rebalance monthly. Why? We calculate our weight variables that we use for weights every month since we have monthly data. What if we want to rebalance annually or quarterly instead? We will tackle that now. Note that if you want to rebalance daily, you will need daily data.

When we rebalance, we trade back to our target weights. When we don't rebalance, prices drift by a factor of 1 plus the price return. Let's walk through some examples to see why this is the case. Imagine the most trivial example. Your portfolio is made up of \$100 of Company A and \$100 of Company B.

Company A	\$100
Company B	\$100
Total	\$200

It starts out 50% Company A and 50% Company B. Imagine Company B drops by 100% to \$0 and Company A's price is constant.

Company A	\$100
Company B	\$0
Total	\$100

Now, your portfolio is 100% Company A. Now, imagine that we are at our same starting point but Company A's price increases by 10% to \$110, while Company B's price falls 10% to \$90.

Company A	\$110
Company B	\$90
Total	\$200

Company A's weight is 55% and Company B's is now 45%. Note that their weights drifted by 1 plus their returns; 55% equals 50% times 110% while 45% equals 50% times 90%. But what about the following situation: Company A goes up by 10% while Company B goes up by 5%?

Company A	\$110
Company B	\$105
Total	\$215

Company A's weight is 51.2% while Company B's is 48.8%. Let's walk through the unnormalized calculation to see how we get to the normalized weight. Company A's unnormalized weight is 50% times 1.10 which is 55%, while Company B's unnormalized weight is 50% times 1.05 which is 52.5%. Add 55% to 52.5% to get the sum of unnormalized weights 107.5%. Then, divide Company A's unnormalized 55% weight by 107.5% and you get 51.2% as before. Divide Company B's unnormalized 52.5% weight by 107.5% and you get 48.8%.

Lastly, let's imagine that Company B pays a \$5 dividend. Does this affect our calculation? That's a difficult question and depends hugely on what we do with that \$5. Practically, we might hold it as cash temporarily. Then, once we've accumulated enough cash, we would invest it to push our weights back to our target weights. However, this would be very complicated to code. In our code, we will treat it as if we had invested it at current weights. This is equivalent to ignoring the dividend and just price-drifting by price return. That is, let's imagine Company A's

price moved to \$110 and Company B's price fell to \$90, but Company B paid a \$5 dividend. If we keep Company A's weight at 55% and Company B's weight at 45%, we are implying that we invested 55% of the \$5 in Company A and 45% of the dividend in Company B. This is why we will use price return to price-drift instead of total return or some more complicated formula that involves holding cash.

The example described above is exactly how we will implement price drifting in the code. We will first drift weights and then normalize. First, we need to lag price returns. After all the unnormalized weight at the beginning of February is the weight at the beginning of January times the price return in January. To lag price returns, we will make a small modification to our `lagmktcap` dataset to create the variable `lagretx`.

```
* lag market cap and price returns;
data lagmktcap (keep=year month PERMNO lagmktcap lagretx);
    set stockReturns;

    * market cap is equal to shares outstanding times price;
    lagmktcap=shrout*prc;
    * rename price return;
    rename retx=lagretx;

    * lag by one month;
    month=month+1;
    if month=13 then do;
        month=1;
        year=year+1;
    end;
run;
```

Make sure that you change the `keep` as well to include `lagretx`. Otherwise, it will be dropped. Now, right after we calculate weights and before we calculate returns, we need to add a couple of more steps.

```
* sort by PERMNO year month;
proc sort data=stockReturns;
    by PERMNO year month;
run;

* drift weights;
data stockReturns(drop=datediff);
    set stockReturns;
    by PERMNO year month;
    retain dynwt;

    datediff=year*12+month-(lag(year)*12+lag(month));
    if month=1 then dynwt=wt;
    else if first.PERMNO or datediff>1 then dynwt=.;
    else dynwt=dynwt*(1+lagretx);

    if missing(dynwt)=0;
run;
```



The sorting step is straightforward. We want to sort by each stock and then by date. The next step contains new techniques. The `retain` `dynwt` statement pulls the `dynwt` that was calculated in the previous row. We'll see why `retain` matters shortly. We calculate the variable `datediff` to determine whether the previous observation is from one month ago. Recall that some firms delist only temporarily so we need to know whether the previous observation is from one month ago or a longer period of time.

We set the price-drifted weights in the `if` statement. This code is reasonably complicated to think through. The one thing to remember is that we want to select a set of stocks in January and price-drift only those stocks in subsequent months. A more detailed description of the code follows. If it's January, we rebalance to our target weights (`if month=1 then dynwt=wt`). If it's not January, we need to check whether this is the first time we have encountered this `PERMNO` (`if first.PERMNO`), in which case that stock does not have a weight, since it's not January. If it is the same stock, we might still not give it a weight. If the previous row is from more than one month ago (`datediff>1`), we cannot assign a weight since the stock temporarily delisted. Lastly, assuming it is not January and the previous row is from last month and from the same stock, we price-drift the previous month's weight by lag return (`dynwt=dynwt*(1+lagretx)`). Note that this is where the `retain` comes in. Because we retained `dynwt`, the line `dynwt=dynwt*(1+lagretx)` sets `dynwt` equal to the previous row's `dynwt` times `(1+lagretx)`. Why couldn't we just forget the `retain` and say: `dynwt=lag(dynwt)*(1+lagretx)`? This will not work for two reasons. The first and most important is that you can't lag a variable that is being assigned to. The second is that you shouldn't use `lag` within an `if` statement. If you are so inclined, you can search online for the precise reasons why the `lag` function does not work as expected within an `if` statement. Next, we need to ensure our dataset is sorted by date again before we calculate returns for each month. We also kick out any stocks with missing dynamic weights.

Finally, we need to change our `proc means` statement to weight by our dynamic price-drifted weight.

```
* sort by date;
proc sort data=stockReturns;
    by year month;
run;

* calculate returns;
proc means data=stockReturns;
    var ret retx;
    weight dynwt;
    by year month;
    output out=stratReturns mean=TR PR;
run;
```

Compare your results using annually rebalanced returns versus monthly rebalanced returns. You should see January matching up exactly and returns becoming slightly more divergent from February to December.

Now, change one line of the code to make the code rebalance quarterly instead of annually. Come back when you are done. The answer should involve changing

```
if month=1 then dynwt=wt
```

to

```
if month=1 or month=4 or month=7 or month=10 then dynwt=wt
```

Next, normalize the price-drifted weights to 100% in each month. That is, each month's weight should sum to 100% but after price-drifting this is not the case. Force `dynwt` to sum to 100% while maintaining the same relative weights. Try this yourself first. After you are done, read ahead.

Your code should look something like this:

```
* sum dynwt in each month;
proc means data=stockReturns noprint;
    var dynwt;
    by year month;
    output out=sumdynwt(drop=_TYPE_ _FREQ_) sum=sumdynwt;
run;

* normalize dynwt to 100% in each month;
data stockReturns(drop=sumdynwt);
    merge stockReturns sumdynwt;
    by year month;
    dynwt = dynwt/sumdynwt;
run;
```

Note that even if you do not include this code, the `proc means` step will run fine using `weight dynwt`. The act of weighting by the dynamic weight inherently normalizes to 100% when calculating the mean. However, we will want to explicitly have normalized price-drifted weights to calculate turnover.

It is great that we can change the code to make our portfolios rebalance annually or quarterly, but what we really want is a single variable that we can change near the top that will allow us to rebalance annually, quarterly, or monthly. To do so, we will use another feature of the SAS macro language, the `%macro` statement. Macros act much like functions in other languages, enclosing code that you can re-run with different parameters. Wrap all the code from the first `data stockReturns` step to the `proc export` step in the following `%macro` statement:

```
%macro calcRet();
    * read the returns file and adjust variables;
```

```

data stockReturns;
    set retdata.crsp_monthly;

    * extract only year and month from date;
    year=year(date);
    month=month(date);

    * add delist returns;
    if missing(dlret)=0 then ret=dlret;
    if missing(dlretx)=0 then retx=dlretx;

    * only keep meaningful returns;
    if ret>=-1;

    * force price to be positive;
    prc=abs(prc);

    * drop the delist returns;
    drop dlret dlretx date;
run;

[REST OF THE CODE]

* export the returns as a CSV;
proc export data=stockReturns
outfile="C:/Data/Output/&weightVariable.MonthlyReturns.csv" dbms=CSV replace;
run;
%mend calcRet;

```

Run this code now and note that it does absolutely nothing. Why? Because all it is doing is defining a `%macro` statement. To run it, we have to call the macro. Type the following after the macro definition:

```
%calcRet();
```

Why did we do all of this and what does this have to do with choosing monthly, quarterly, or annual rebalancing? Add a parameter at the top of the code under our definition of `weightVariable`:

```
%let rebfreq = annual;
```

Now, change our price-drifting code to:

```

* drift weights;
data stockReturns(drop=datediff);
    set stockReturns;
    by PERMNO year month;
    retain dynwt;

    datediff=year*12+month-(lag(year)*12+lag(month));
    %if &rbfreq=annual %then %do;
        if month=1 then dynwt=wt;
        else if first.PERMNO or datediff>1 then dynwt=.;
    %end;
run;

```

```

        else dynwt=dynwt*(1+lagretx);
    %end;
    %else %if &rebfreq=quarterly %then %do;
        if month=1 or month=4 or month=7 or month=10 then dynwt=wt;
        else if first.PERMNO or datediff>1 then dynwt=.;
        else dynwt=dynwt*(1+lagretx);
    %end;
    %else %if &rebfreq=monthly %then %do;
        dynwt=wt;
    %end;
    %else %do;
        %put ERROR: Unexpected value for rebfreq;
    %end;

    if missing(dynwt)=0;
run;

```

Here, we are using the macro `%if` statement, which can only be run within a `%macro` statement. And that is exactly why we surrounded all of our code in a `%macro` statement. In any case, the macro `%if` statement is of the form:

```

%if [CONDITION] %then %do;
    [ACTIONS]
%end;

```

If the condition is met, the actions will run. Otherwise, it will not. In this case if we set `rebfreq` to `annual`, the portfolio will annually rebalance. If we set it to `quarterly`, the portfolio will rebalance quarterly. If set to `monthly`, it will rebalance monthly. Otherwise, it will print an error to the log. The `%put` statement outputs what follows to the log. If you start the output with `ERROR:`, the log will show the output in red to indicate an error.

## Why Use Macro Statements?

If you haven't used other languages and aren't familiar with functions, it may not be immediately clear why we need macro statements. If you are familiar with functions from other languages, it is likely abundantly clear why we need macro statements, so feel free to skip this section.

Macro statements, like functions in other languages, encapsulate code and take can be re-run with a varying set of parameters. We will cover parameters later, but for now, whenever you think, "I might re-use this code," encapsulate it in a macro statement and name it appropriately. For example, if you are cleaning or preparing data that you might use repeatedly, you can create a `cleanData` or `prepareData` macro that encapsulates that code. We have *barely* touched upon the power of macro statements in this chapter. In every subsequent chapter, nearly every piece of code we will build will be encapsulated in macro statements and you will be able to build different portfolios and different factors with a mere line of code.

When you work for a firm or conduct research with other researchers, they will likely ask that all of the relevant code that you build be in the form of reusable macro statements, so that they can quickly incorporate your code. Trust us—no one likes being sent a giant block of code that they need to parse through to get any use out of. If you take away one thing from this chapter or perhaps even this book, it should be “Make your code re-usable.” And the way to make code re-usable and thus useful is to use macro statements.

## Analytics

We know how to construct basic portfolios. Next, we should learn to compute basic analytics. We will calculate industry weights, turnover and performance tables.

## Industry Weights

Let’s start with industry weights. First, note that we have Standard Industry Classification (SIC) codes in the stock returns data set. These are 4-digit industry classifiers. That means there are a lot of SIC codes. We want to group the SIC codes into fewer industries. Luckily, Fama and French have already done this<sup>2</sup>, so we can piggy back off of their insights. Change the initial `stockReturns` data step to this:

```
* read the returns file and adjust variables;
data stockReturns;
    set retdata.crsp_monthly;

    * extract only year and month from date;
    year=year(date);
    month=month(date);

    * add delist returns;
    if missing(dlret)=0 then ret=dlret;
    if missing(dlretx)=0 then retx=dlretx;

    * only keep meaningful returns;
    if ret>=-1;

    * force price to be positive;
    prc=abs(prc);

    * drop the delist returns;
    drop dlret dlretx date;

    * set sector;
    if (siccd>=0100 & siccd<=0999) | (siccd>=2000 & siccd<=2399) |
(siccd>=2700 & siccd<=2749) | (siccd>=2770 & siccd<=2799) | (siccd>=3100 &
siccd<=3199) | (siccd>=3940 & siccd<=3989) then
sector12="ConsumerNonDurable";
    else if (siccd>=2500 & siccd<=2519) | (siccd>=2590 & siccd<=2599) |
(siccd>=3630 & siccd<=3659) | (siccd>=3710 & siccd<=3711) | (siccd>=3714 &
siccd<=3714) | (siccd>=3716 & siccd<=3716) |
```

---

<sup>2</sup> Fama, Eugene F., and Kenneth R. French. "Industry costs of equity." *Journal of Financial Economics* 43.2 (1997): 153-193.

```

        (siccd>=3750 & siccd<=3751) | (siccd>=3792 & siccd<=3792) |
(siccd>=3900 & siccd<=3939) | (siccd>=3990 & siccd<=3999) then
sectorl2="ConsumerDurable";
    else if (siccd>=2520 & siccd<=2589) | (siccd>=2600 & siccd<=2699) |
(siccd>=2750 & siccd<=2769) | (siccd>=3000 & siccd<=3099) | (siccd>=3200 &
siccd<=3569) | (siccd>=3580 & siccd<=3629) |
        (siccd>=3700 & siccd<=3709) | (siccd>=3712 & siccd<=3713) |
(siccd>=3715 & siccd<=3715) | (siccd>=3717 & siccd<=3749) | (siccd>=3752 &
siccd<=3791) | (siccd>=3793 & siccd<=3799) |
        (siccd>=3830 & siccd<=3839) | (siccd>=3860 & siccd<=3899) then
sectorl2="Manufacturing";
    else if (siccd>=1200 & siccd<=1399) | (siccd>=2900 & siccd<=2999) then
sectorl2="Energy";
    else if (siccd>=2800 & siccd<=2829) | (siccd>=2840 & siccd<=2899) then
sectorl2="Chemicals";
    else if (siccd>=3570 & siccd<=3579) | (siccd>=3660 & siccd<=3692) |
(siccd>=3694 & siccd<=3699) | (siccd>=3810 & siccd<=3829) | (siccd>=7370 &
siccd<=7379) then sectorl2="BusinessEquipment";
    else if (siccd>=4800 & siccd<=4899) then sectorl2="Telecom";
    else if (siccd>=4900 & siccd<=4949) then sectorl2="Utilities";
    else if (siccd>=5000 & siccd<=5999) | (siccd>=7200 & siccd<=7299) |
(siccd>=7600 & siccd<=7699) then sectorl2="Retail";
    else if (siccd>=2830 & siccd<=2839) | (siccd>=3693 & siccd<=3693) |
(siccd>=3840 & siccd<=3859) | (siccd>=8000 & siccd<=8099) then
sectorl2="Health";
    else if (siccd>=6000 & siccd<=6999) then sectorl2="Finance";
    else sectorl2="Other";
run;

```

You don't need to devote too much thought to the new code. In effect, it is saying if the SIC codes fall within certain ranges, then we set the 12 sector classification to a relevant industry. Note that the pipe symbol | means "or" while the ampersand & means "and". Now, after we normalize `dynwt`, calculate the industry weights for each industry in each month. You should already know how to do this! Try it. Once you are done, continue reading.

Your code should look like this and should be placed after we normalized weights at the very bottom of the macro statement:

```

* sort by date and sector;
proc sort data=stockReturns;
    by year month sectorl2;
run;

* calculate industry weights;
proc means data=stockReturns noprint;
    var dynwt;
    by year month sectorl2;
    output out=colsectorweights sum=secwt;
run;

```

Open up `colsectorweights` and note that the data is in a single column. It seems more intuitive for there to be a column for each sector. Luckily, there is a way to do this:

```

* transpose to columns;
proc transpose data=colsectorweights out=sectorweights;
    var secwt;
    by year month;
    id sector12;
run;

```

This transposes the sector weights for each date. Feel free to export this to a CSV or Excel file if you would find this information useful.

## Turnover

Next, we will calculate turnover. Turnover can either be one-way or two-way. Assume we sell 5% of Stock A and buy 5% of Stock B or some other set of stocks. We can either count this as 5% turnover (one-way) or 10% turnover (two-way). Maximum two-way turnover at any rebalance is 200% assuming you buy a whole new set of stocks. Maximum one-way turnover is 100%. Two-way turnover is always double one-way turnover.

We will calculate two-way turnover for the simple reason that it is easier. Note that when there are delists, this will create turnover in our portfolio and we assume that we use the proceeds from the delisting and distribute it among all other stocks according to their relative weights in the portfolio. Is this difficult to code? No! Actually, we have already coded it like that. It's implicit. When a stock delists, its weight disappears. When we use `weight dynwt` in the `proc means` step, the weight of the missing stock is effectively re-distributed among the other stocks.

The result of delisting is that we will have turnover in the non-rebalance months as well! However, the rebalance months will have by far the greatest turnover. After we calculated portfolio returns, let's calculate turnover. You have all the coding knowledge to do this, but it's non-trivial to think through.

We will talk it through and you can give it a try. If you get it, that is truly impressive. Here are the steps:

- 1) Calculate beginning of month weights. These are lagged dynamic weights drifted by lagged price returns.
- 2) Normalize these weights to 100%.
- 3) Merge them back into the regular dynamic weights.
- 4) Set missing values of dynamic weights and lagged price-drifted dynamic weights to 0. Calculate the absolute value of the difference between the dynamic weights and lagged price-drifted dynamic weights. This is the individual stock turnover.
- 5) Sum the individual stock turnover in each year and month. This is the portfolio turnover.

The code should look something like this:

```

* sort by date and PERMNO;
proc sort data=stockReturns;

```

```

        by year month PERMNO;
run;

* create pre-turnover weights;
data lagwts(keep=PERMNO year month predynwt);
    set stockReturns;
    month=month+1;
    if month=13 then do;
        month=1;
        year=year+1;
    end;
    predynwt=dynwt*(1+retx);
run;

* sum pre-turnover weights;
proc means data=lagwts noprint;
    var predynwt;
    by year month;
    output out=sumpredynwt(drop=_TYPE_ _FREQ_) sum=sumpredynwt;
run;

* normalize pre-turnover weights;
data lagwts(drop=sumpredynwt);
    merge lagwts sumpredynwt;
    by year month;
    predynwt=predynwt/sumpredynwt;
run;

* calculate individual stock turnover;
data stockReturns;
    merge stockReturns lagwts;
    by year month PERMNO;

    if missing(predynwt) then predynwt=0;
    if missing(dynwt) then dynwt=0;
    turnover=abs(dynwt-predynwt);
run;

* calculate monthly portfolio turnover;
proc means data=stockReturns;
    var turnover;
    by year month;
    output out=turnover(drop=_TYPE_) sum=turnover;
run;

```

Open the turnover data set. Note that the first month's turnover is complete nonsense! We have to buy into the whole portfolio, so of course our turnover will appear to be 100%, but that does not reflect the turnover that we will experience from year to year. Therefore, it makes sense that we only include whole years of data for turnover. You can kick out the first 12 months using the following code. In a `data` step, `_N_` is interpreted as the row number.

Also note that the merge step between table `stockReturns` and `lagwts` will be an outer join, meaning all information between these two tables will be restored. Imagine company A is delisted on February 2014, on table `stockReturns` there will be not rows on March 2014 while



on table `lagwts` there will be a row on March 2014 for company A. When outer join the two tables, there will be a row on March 2014 for company A in the merged table and value in `dynwt` will be missing and therefore later on assigned a weight of 0. The same rules apply to companies that are first listed. Make sure to delete the extra “last month” rows created by merging with table `lagwts`.

```
* remove first year and set up removal of last observation;
data turnover;
    set turnover;
    by year month;
    if _N_>12;
    filter=1;
run;

* sort stocks by filter and date;
proc sort data=turnover;
    by filter year month;
run;

* remove last observation;
data turnover(drop = filter);
    set turnover;
    by filter year month;
    if last.filter then delete;
run;
```

As an exercise, calculate average turnover over the full period and annualize it. It should look something like this:

```
* calculate average turnover;
proc means data=turnover noprint;
    var turnover;
    output out=meanturnover(drop=_TYPE_ _FREQ_) mean=meanturnover;
run;

* annualize turnover;
data meanturnover;
    set meanturnover;
    meanturnover=meanturnover*12;
run;
```

## Performance Table

Now, we will create a new macro statement that will compute the Sharpe Ratio, geometric return, volatility, tracking error, information ratio, Jensen alpha, and Fama-French alpha. First, let’s quickly recall what these terms mean.

## Terminology

First, let’s recall what a geometric and arithmetic mean return is. The arithmetic mean return is:

$$\mu_a = \frac{1}{T} (r_1 + \dots + r_T) = \frac{1}{T} \sum_{t=1}^T r_t$$

The geometric mean return is:

$$\mu_g = \sqrt[T]{(1 + r_1) \cdots (1 + r_T)} - 1 = \sqrt[T]{\prod_{t=1}^T (1 + r_t)} - 1$$

Recall that  $\mu_g \leq \mu_a$ . The one case where geometric returns are equal to arithmetic returns is when returns are identical in every period. Indeed, the higher the volatility of returns (all other things being equal), the lower geometric returns are relative to arithmetic returns. The following is approximately accurate:

$$\mu_g \approx \mu_a - \frac{\sigma^2}{2}$$

This relationship is a critical one to consider since investors earn geometric returns. Let's consider the most extreme case. Imagine a stock increased by 100% and then fell by 100%. The arithmetic average return is 0% but the geometric return is -100%. Dropping by 100% wipes out an investor regardless of anything that happened prior. Practically, this still matters. Increasing by 10% and dropping by 10% results in an investor losing 1%. Increasing by 20% and dropping by 20% results in an investor losing 4%. (Note that the order of increasing or decreasing is irrelevant since multiplication is commutative.)

Volatility is just the standard deviation of returns.

$$\sigma = \sqrt{\frac{1}{T} \sum_{t=1}^T (r_t - \mu_a)^2}$$

Note that we use regular arithmetic standard deviation, since geometric standard deviation is dimensionless and thus harder to interpret. We won't discuss geometric standard deviation further as it is not used in finance.

Sharpe Ratio is the ratio of risk premium to its corresponding volatility. Of course, the next question is what is the risk premium? The risk premium is the excess return of a portfolio over the risk-free rate. We generally use 1-month Treasury returns since these are easy to find on the Kenneth French Data Library. We should note that nothing is actually risk-free but short-term Treasury bills are less likely to default than basically any other security. Let's assume that

the risk-free rate in period  $t$  is  $f_t$  and the arithmetic mean risk-free rate is  $f\mu_a$ . Then, the Sharpe Ratio of an asset is:

$$SR = \frac{\sqrt[T]{(1 + r_1 - f_1) \cdots (1 + r_T - f_T)} - 1}{\sqrt{\frac{1}{T} \sum_{t=1}^T (r_t - f_t - (\mu_a - f\mu_a))^2}}$$

We should ask ourselves why we even care about risk premium? Why not just care about return? What's so special about the risk-free rate? There are a variety of reasons. First, institutions can leverage returns at an interest rate close to the risk-free rate, so the ratio of risk premium to volatility gives an indication of what return one can earn relative to a given level of risk. Indeed, when you evaluate a strategy in isolation that you can cheaply leverage, the Sharpe Ratio is the most important thing. However, those are some very hard-to-meet assumptions (strategy in isolation that can be cheaply levered).

The idea of a risk premium is important even apart from Sharpe Ratio. A particular risky asset better pay more than the risk-free rate on average but not always. Otherwise, an investor could enjoy a risk-free profit by sticking with the risk-free rate. If a particular risky asset *always* paid more than the risk-free rate in all states of the world, then no sane investor would invest in the risk-free rate, so neither of these situations can exist an equilibrium. The risky asset must pay more than the risk-free rate on average but sometimes pay less.

By the way, would you guess that on average when short-term Treasury (risk-free) rates are high, stock market returns will be higher or lower than when short-term Treasury rates are low? If you were to just think about investors demanding higher risky asset returns when the risk-free rate is high, one would guess that stock market returns would be higher when the risk-free rate is higher. However, empirically, stock market returns tend to be negatively correlated with the level of the risk-free rate. There are numerous theories one could make where such a result would manifest itself, but I do not know which is correct, so we leave it to you to delve deeper into the matter if you are so inclined.

What is tracking error? The tracking error is the standard deviation of the excess returns over a benchmark. If the return of a benchmark is  $b_t$  and the arithmetic mean benchmark return is  $b\mu_a$ , then the tracking error is:

$$TE = \sqrt{\frac{1}{T} \sum_{t=1}^T (r_t - b_t - (\mu_a - b\mu_a))^2}$$

Why the hell do we care about this? We care because if we are benchmarked to a particular index—let's say the S&P 500 or Russell 1000 for large capitalization equities—our performance cannot deviate too far from it. First note that tracking error is the standard deviation of excess

returns. If you always outperform your benchmark by 0.01% per day, your tracking error is 0%. Also note that a retail investor probably does not care a great deal about tracking error. If your fund doesn't end up behaving like the S&P 500—going up when the S&P 500 goes down and going down when the S&P 500 goes up—it's unlikely that the retail investor will be angry, because a retail investor is not buying your fund because it behaves like an index. They're buying it because they want you to make them money. But an institutional investor will often care about tracking error, because they're investing in you for a very specific reason. They are allocating among asset classes and among asset managers and if your performance looks nothing like the benchmark that you are supposed to match, their risk profile is going to end up looking far different than what they were expecting. This is why tracking error matters and it matters more to institutions.

The analogue to Sharpe Ratio for benchmarked strategies is information ratio. The information ratio is the ratio of excess return over a benchmark to the volatility of that excess return.

$$IR = \frac{\sqrt[T]{(1 + r_1 - b_1) \cdots (1 + r_T - b_T)} - 1}{\sqrt{\frac{1}{T} \sum_{t=1}^T (r_t - b_t - (\mu_a - b\mu_a))^2}}$$

The information ratio tells investors how much return you are earning for every unit tracking error you take. If Fund A earns a 2% excess return at 2% tracking error while Fund B earns a 1% excess return at 0.5% tracking error, an institution may well be more impressed with Fund B. They earned lower excess returns but did it with much less risky active bets. (Of course, some institutions may still prefer Fund A if they have a higher tracking error budget, but the example is purely for illustration.)

## Calculations

First, go to the Kenneth R. French Data Library at [http://mba.tuck.dartmouth.edu/pages/faculty/ken.french/data\\_library.html](http://mba.tuck.dartmouth.edu/pages/faculty/ken.french/data_library.html). Click on this link to download the Fama French market risk premium, value factor, size factor, and risk-free rate. We will be using market returns to compute tracking error and information ratio.

### U.S. Research Returns Data (Downloadable Files)

#### Changes in CRSP Data

Fama/French 3 Factors [TXT](#) [CSV](#) [Details](#)  
 Fama/French 3 Factors [Weekly] [TXT](#) [CSV](#) [Details](#)  
 Fama/French 3 Factors [Daily] [TXT](#) [CSV](#) [Details](#)

We will unzip this file to C:\Data. In addition, download the daily data. We will use it in the next chapter. To input the file, we will use `infile` and a macro statement. But first, create a blank code file in SAS by clicking on the editor and hitting CTRL+N or going to File->New Program.

```

%macro inputFF(FFloc);
  data FF(drop=date);
    infile "&FFloc." firstobs=5;
    input date Mkt_RF SMB HML RF;

    informat date yymmn6.;

    month=month(date);
    year=year(date);

    if date=. then stop;

    Mkt_RF=Mkt_RF/100;
    SMB=SMB/100;
    HML=HML/100;
    RF=RF/100;

  run;
%mend inputFF;

```

Save the file as “performance.sas” or something similarly descriptive. This macro statement has a parameter `FFloc`. Parameters are macro variables that only exist in the macro statement. Parameters are defined when the macro is called. We can call the macro using this line:

```
%inputFF(C:/Data/F-F_Research_Data_Factors.txt);
```

For this particular run of the macro, `FFloc` is defined as `C:/Data/F-F_Research_Data_Factors.txt`. Any instance of `&FFloc` is interpreted as `C:/Data/F-F_Research_Data_Factors.txt`. If you are used to coding in other languages, you might note that it is strange to see a bunch of colons and slashes right in the middle of a function argument. This can be done in SAS!

Within the data step, we are using the `infile` statement to read a file. The option that we are interested in is `firstobs` which sets the first line to read an observation. If you open the file, you will note that the first observation occurs on the 5<sup>th</sup> line. Then, we need to use `input` to tell SAS the format and name of our columns. Note that there is no indication of format on this line. SAS can figure out these observations are numbers. Note that there is much more nuance to the `input` statement. If you need to input some other data and the data format seems incorrect, search online for more information on the `input` statement. On the next line, we need to force the input format of `date` to be of the form YYYYMM (4-digit year followed by 2-digit month). To do this we use the line:

```
informat date yymmn6.;
```

Why is the format YYYYMM denoted `yymmn6.`? I presume `yymmn6.` suggests 6 digits with a year followed by a month, but in general the formats are not incredibly intuitive, so look up date formats if you ever wonder how to format a date in a particular format.

([http://support.sas.com/documentation/cdl/en/etsug/60372/HTML/default/viewer.htm#etsug\\_intervals\\_sect010.htm](http://support.sas.com/documentation/cdl/en/etsug/60372/HTML/default/viewer.htm#etsug_intervals_sect010.htm)) Note that we used `informat` instead of `format`.

This is a somewhat confusing aspect of SAS. We use `informat` for input formatting and `format` for output formatting. In general, you want to output and input in the same format, so it often makes sense to both format and informat a variable to the same thing. In our case, we want to drop `date` and create two variables `month` and `year`, so we only care how `date` is input. Note that if you don't set the `informat` of `date` to `yymmnn6.`, the lines

```
month=month(date);  
year=year(date);
```

will not work properly. Lastly, we want to stop reading once we hit a blank line, so we run the following line:

```
if date=. then stop;
```

Try commenting out each of these lines and running the code to see the varying effects. You need all the lines and sometimes removing a line has strange effects. Removing the `informat` line causes SAS to misinterpret the `date` variable and thus create nonsensical `month` and `year` variables. Leaving out the `if date=. then stop` causes SAS to input all of the annual data below the monthly data, which we definitely do not want.

Now, we want to create a macro that will calculate our various performance analytics. Let's call this function `perfAnalytics`. Let's also give it three parameters: `stratReturns`, `FF`, and `outTable`. Make an empty macro statement that contains these three parameters. After you are done, come back. The statement should look like this:

```
%macro perfAnalytics(stratReturns,FF,outTable);  
  
%mend perfAnalytics;
```

For the purposes of debugging, we will set the parameters. We will also make sure to delete this when we are ready to use the macro. Run the following code, purely for the purposes of debugging:

```
%let stratReturns=stratReturns;  
%let FF=FF;  
%let outTable=allCapMonthly;
```

Also, run the cap-weighted monthly rebalanced returns. That will involve setting the parameters as follows:

```
%let weightVariable = lagmktcap;  
%let rebfreq = monthly;
```

Now within %macro perfAnalytics, we will show a new step, proc sql. This allows you to treat datasets like a SQL table. If you do not know SQL, then don't worry. We will also show an alternative that does not use SQL.

```
* merge strategy returns with Fama-French data;
proc sql;
    CREATE TABLE stratFF AS
    SELECT * FROM &stratReturns AS stratRet
    INNER JOIN &FF as famaFrench
    ON stratRet.year=famaFrench.year AND
    stratRet.month=famaFrench.month;
quit;
```

Note that data sets that are merged through proc sql do not need to be sorted in any particular order. These two data sets happen to be sorted by date by default though. If you prefer not to use SQL, use the following code, which no doubt should be more familiar.

```
data stratFF;
    merge stratReturns(in=k) FF(in=l);
    by year month;
    if k and l;
run;
```

The latter may seem preferable at this stage, but down the road, it is certainly worth investing time in understanding proc sql. As one might expect, it is exactly as deep and powerful as SQL.

Now, we need to calculate geometric return and arithmetic volatility for geometric Sharpe Ratios. Why use arithmetic volatility for a geometric Sharpe Ratio? Geometric standard deviation is rarely used since it is dimensionless. Why use geometric return? Geometric return is a better measure of what the investor experienced during the holding period, but it is reasonable to use arithmetic returns depending one's assumptions about the distribution of returns. The easiest way to calculate geometric means and standard deviations is to take natural logs, take arithmetic means and standard deviations, and then exponentiate them. Note that the function for natural log in SAS is just log.

```
* create various return metrics in which we are interested;
data stratFF;
    set stratFF;

    RP=TR-RF;
    ER=TR-RF-Mkt_RF;
    Mkt=Mkt_RF+RF;

    logTR=log(1+TR);
    logRP=log(1+TR-RF);
    logER=log(1+TR-RF-Mkt_RF);
    logMkt=log(1+Mkt_RF+RF);
    logMkt_RF=log(1+Mkt_RF);
```

```
run;
```

Note that RP stands for risk premium and ER stands for excess return. Now, we need to take the mean and standard deviations of all of these variables using `proc means`. Feel free to give that a shot before continuing.

```
* calculate mean and volatility of returns;
proc means data=stratFF noprint;
    var logTR logRP logER logMkt logMkt_RF TR RP ER Mkt Mkt_RF;
    output out=allSumStats
        mean=meanlogTR meanlogRP meanlogER meanlogMkt meanlogMkt_RF meanTR
        meanRP meanER meanMkt meanMkt_RF
        stddev=vollogTR vollogRP vollogER vollogMkt vollogMkt_RF volTR volRP
        volER volMkt volMkt_RF;
run;
```

Now, we will compute all of our desired annualized metrics.

```
* calculate SRs, compounded returns, and volatilities;
data allSumStats;
    set allSumStats;
    stratMean = exp(meanlogTR*12)-1;
    stratVol = volTR*sqrt(12);
    stratSR = (exp(meanlogRP*12)-1)/(volRP*sqrt(12));
    stratIR = (exp(meanlogER*12)-1)/(volER*sqrt(12));

    benchMean = exp(meanlogMkt*12)-1;
    benchVol = volMkt*sqrt(12);
    benchSR = (exp(meanlogMkt_RF*12)-1)/(volMkt_RF*sqrt(12));
run;

* tranpose data for easier viewing;
proc transpose data=allSumStats out=&outTable;
    var stratMean stratVol stratSR stratIR benchMean benchVol benchSR;
run;
```

Run this code to ensure that it works fine. We want to run this analysis in our macro `calcRet`. How do we run a macro from one code in another block of code? There is a macro function that allows us to include the code from one file into another. Add this to the top of the file that computes returns:

```
* include performance analytics;
%include "H:/Book/Ch 1 Code/performance.sas";
```

Be sure to change the location above to the appropriate location of the file. Now, move the global parameters below the function definition and above the function call like so:

```
* parameters;
%let weightVariable = lagmktcap;
%let rebfreq = monthly;
```



```
%calcRet();
```

Then, put the inputting of the Fama-French factors above this code and calculate the analytics below for both cap-weighted and equally-weighted parameters.

```
%inputFF(C:/Data/F-F_Research_Data_Factors.txt);
```

```
* cap-weighted parameters;
```

```
%let weightVariable = lagmktcap;
```

```
%let rebfreq = monthly;
```

```
%calcRet();
```

```
%perfAnalytics(stratReturns,FF,allCapMonthly);
```

```
* equally-weighted parameters;
```

```
%let weightVariable = ew;
```

```
%let rebfreq = monthly;
```

```
%calcRet();
```

```
%perfAnalytics(stratReturns,FF,allEWMonthly);
```

Now, run this code. Note that the capitalization-weighted portfolio is extremely close to the market portfolio calculated by Fama-French. That is because their portfolio is nearly identical to this one. The equally-weighted portfolio outperforms the capitalization-weighted portfolio but has considerably higher volatility such that the Sharpe Ratio is nearly identical. We will look at variants of this portfolio later.