Data Analytics in Finance

FINA 6333 for Spring 2025

Richard Herron

Table of contents

Welcome to FINA 6333 for Spring 2025 at the D'Amore-McKim School of Business at Northeastern University!

For each course topic, we will have one notebook for the pre-recorded lecture and one for the in-class practice. I will maintain these notebooks on here and everything else on Canvas. You have three choices to access these notebooks:

1. Download them from OneDrive
2. Download them from GitHub
3. Open them on Google Colab

Week 1

McKinney Chapter 2 - Python Language Basics, IPython, and Jupyter Notebooks

Introduction

We must understand the basics of Python before we can use it to analyze financial data. Chapter 2 of McKinney (2022) provides a crash course in Python's syntax, and Chapter 3 provides a crash course in Python's built-in data structures. This notebook focuses on the "Python Language Basics" in Section 2.3, which covers language semantics, scalar types, and control flow.

Note: Indented block quotes are from McKinney (2022) unless otherwise indicated. The section numbers here differ from McKinney (2022) because we will only discuss some topics.

Language Semantics

Indentation, not braces

> Python uses whitespace (tabs or spaces) to structure code instead of using braces as in many other languages like R, C++, Java, and Perl.

Spaces are more than cosmetic in Python. Here is a `for` loop with an `if` statement that shows how Python uses identation to separate code instead of parentheses and braces.

```
array = [1, 2, 3]
pivot = 2
less = []
greater = []

for x in array:
    if x < pivot:
        print(f'{x} is less than {pivot}')
        less.append(x)
    else:
```

```
        print(f'{x} is NOT less than {pivot}')
        greater.append(x)
```

```
1 is less than 2
2 is NOT less than 2
3 is NOT less than 2
```

```
less
```

```
[1]
```

```
greater
```

```
[2, 3]
```

Comments

> Any text preceded by the hash mark (pound sign) # is ignored by the Python interpreter. This is often used to add comments to code. At times you may also want to exclude certain blocks of code without deleting them.

The Python interpreter ignores any code after a hash mark `#` on a given line. We can quickly comment/un-comment lines of code with the `<Ctrl>-/` shortcut.

```
# We often use comments to leave notes for future us (or co-workers)
# 5 + 5
```

Function and object method calls

> You call functions using parentheses and passing zero or more arguments, optionally assigning the returned value to a variable:
>
> ```
> result = f(x, y, z)
> g()
> ```
>
> Almost every object in Python has attached functions, known as methods, that have access to the object's internal contents. You can call them using the following syntax:

```
obj.some_method(x, y, z)
```

Functions can take both positional and keyword arguments:

```
result = f(a, b, c, d=5, e='foo')
```

More on this later.

Here is a function named **add_numbers** that adds two numbers.

```python
def add_numbers(a, b):
    return a + b
```

```python
add_numbers(5, 5)
```

```
10
```

Here is a function named **add_strings** that adds or concatenates two strings separated by a space.

```python
def add_strings(a, b):
    return a + ' ' + b
```

```python
add_strings('5', '5')
```

```
'5 5'
```

What is the difference between `print()` and `return`?

- `print()` returns its argument to the console or "standard output"
- `return` returns its argument as an output we can assign to variables

Please see the following example.

```python
def add_strings_2(a, b):
    string_to_print = a + ' ' + b + ' (this is from the print statement)'
    string_to_return = a + ' ' + b + ' (this is from the return statement)'
    print(string_to_print)
    return string_to_return
```

```
returned = add_strings_2('5', '5')
```

```
5 5 (this is from the print statement)
```

```
returned
```

```
'5 5 (this is from the return statement)'
```

Variables and argument passing

> When assigning a variable (or name) in Python, you are creating a reference to the object on the righthand side of the equals sign.

```
a = [1, 2, 3]
b = a
```

If we assign `a` to a new variable `b`, both `a` and `b` refer to the *same* object, which is the list `[1, 2, 3]`.

```
a is b
```

```
True
```

If we modify `a` by appending 4, we also modify `b` because `a` and `b` refer to the same list.

```
a.append(4)
```

```
a
```

```
[1, 2, 3, 4]
```

```
b
```

```
[1, 2, 3, 4]
```

Likewise, if we modify `b` by appending 5, we also modify `a`.

```python
b.append(5)
```

```python
b
```

```
[1, 2, 3, 4, 5]
```

```python
a
```

```
[1, 2, 3, 4, 5]
```

Dynamic references, strong types

> In contrast with many compiled languages, such as Java and C++, object references in Python have no type associated with them.

Python has *dynamic references*. Therefore, we do not declare variable types, and we can change variable types. This behavior is because variables are names assigned to objects.

For example, above we assign `a` to a list, and below we can reassign it to an integer and then a string.

```python
a
```

```
[1, 2, 3, 4, 5]
```

```python
type(a)
```

```
list
```

```python
a = 5
type(a)
```

```
int
```

```python
a = 'foo'
type(a)
```

```
str
```

Python has *strong types.* Therefore, Python typically will not convert object types.

For example, `'5' + 5` returns either `'55'` as a string or `10` as an integer in many programming languages. However, below `'5' + 5` returns an error because Python will not implicitly convert the type of the string or integer.

```
# '5' + 5 #TypeError: can only concatenate str (not "int") to str
```

However, Python will implicitly convert integers to floats.

```
a = 4.5
b = 2
a / b
```

```
2.25
```

Attributes and methods

We can use tab completion to access attributes (characteristics stored inside objects) and methods (functions associated with objects). Tab completion is a feature of the IPython and Jupyter environments.

```
a = 'foo'
```

```
a.capitalize()
```

```
'Foo'
```

```
a.upper().lower()
```

```
'foo'
```

```
a.count('o')
```

```
2
```

Binary operators and comparisons

Binary operators operate on two arguments.

```
5 - 7
```

```
-2
```

```
12 + 21.5
```

```
33.5
```

```
5 <= 2
```

```
False
```

Table 2-1 from McKinney (2022) summarizes the binary operators.

- `a + b` : Add a and b
- `a - b` : Subtract b from a
- `a * b` : Multiply a by b
- `a / b` : Divide a by b
- `a // b` : Floor-divide a by b, dropping any fractional remainder
- `a ** b` : Raise a to the b power
- `a & b` : True if both a and b are True; for integers, take the bitwise AND
- `a | b` : True if either a or b is True; for integers, take the bitwise OR
- `a ^ b` : For booleans, True if a or b is True , but not both; for integers, take the bitwise EXCLUSIVE-OR
- `a == b` : True if a equals b
- `a != b`: True if a is not equal to b
- `a <= b, a < b` : True if a is less than (less than or equal) to b
- `a > b, a >= b`: True if a is greater than (greater than or equal) to b
- `a is b` : True if a and b reference the same Python object
- `a is not b` : True if a and b reference different Python objects

Mutable and immutable objects

> Most objects in Python, such as lists, dicts, NumPy arrays, and most user-defined types (classes), are mutable. This means that the object or values that they contain can be modified.

A list is a *mutable*, ordered collection of elements, which can be any data type. *Because lists are mutable, we can modify them.* Lists are defined using square brackets `[]` with elements separated by commas. Lists support indexing, slicing, and various methods for adding, removing, and modifying elements.

```
a_list = ['foo', 2, [4, 5]]
a_list
```

```
['foo', 2, [4, 5]]
```

Python is zero-indexed! The first element has a zero subscript `[0]`!

```
a_list[0]
```

```
'foo'
```

```
a_list[2]
```

```
[4, 5]
```

```
a_list[2][0]
```

```
4
```

```
a_list[2] = (3, 4)
a_list
```

```
['foo', 2, (3, 4)]
```

A tuple is an *immutable*, ordered collection of elements, which can be any data type. *Because tuples are immutable, we cannot modify them.* Tuples are defined using optional but helpful parentheses `()`, with elements separated by commas.

```
a_tuple = (3, 5, (4, 5))
a_tuple
```

```
(3, 5, (4, 5))
```

The Python interpreter returns an error if we try to modify `a_tuple` because tuples are immutable.

```
# a_tuple[1] = 'four' # TypeError: 'tuple' object does not support item assignment
```

The parentheses `()` are optional for tuples. However, parentheses `()` are helpful because they improve readability and remove ambiguity.

```
test = 1, 2, 3
type(test)
```

```
tuple
```

We will learn more about Python's built-in data structures in Chapter 3.

Scalar Types

> Python along with its standard library has a small set of built-in types for handling numerical data, strings, boolean (True or False) values, and dates and time. These "single value" types are sometimes called scalar types and we refer to them in this book as scalars. See Table 2-4 for a list of the main scalar types. Date and time handling will be discussed separately, as these are provided by the datetime module in the standard library.

Table 2-2 from McKinney (2022) summarizes the standard scalar types.

- `None`: The Python "null" value (only one instance of the None object exists)
- `str`: String type; holds Unicode (UTF-8 encoded) strings
- `bytes`: Raw ASCII bytes (or Unicode encoded as bytes)
- `float`: Double-precision (64-bit) floating-point number (note there is no separate double type)
- `bool`: A True or False value
- `int`: Arbitrary precision signed integer

Numeric types

Integers are unbounded in Python. The `**` binary operator raises the number on the left to the power on the right.

```
ival = 17239871
ival ** 6
```

```
262545192910924565969654629132307297011027 21
```

Floats (decimal numbers) are 64-bit in Python.

```
fval = 7.243
type(fval)
```

```
float
```

Dividing integers yields a float, if necessary.

```
3 / 2
```

```
1.5
```

We use `//` if we want integer division.

```
3 // 2
```

```
1
```

Booleans

> The two Boolean values in Python are written as True and False. Comparisons and other conditional expressions evaluate to either True or False. Boolean values are combined with the and and or keywords.

We must type Booleans as `True` and `False` because Python is case sensitive.

```
True and True
```

```
True
```

```
(5 > 1) and (10 > 5)
```

```
True
```

```
False and True
```

```
False
```

```
False or True
```

```
True
```

```
(5 > 1) or (10 > 5)
```

```
True
```

We can substitute `&` for `and` and `|` for `or`.

```
True & True
```

```
True
```

```
False & True
```

```
False
```

```
False | True
```

```
True
```

Type casting

We can "recast" variables to change their types.

```
s = '3.14159'
type(s)
```

```
str
```

```
1 + float(s)
```

```
4.14159
```

```
fval = float(s)
type(fval)
```

```
float
```

```
int(fval)
```

```
3
```

We can recast a string `'5'` to an integer or an integer `5` to a string to prevent the `5 + '5'` error above.

```
5 + int('5')
```

```
10
```

```
str(5) + '5'
```

```
'55'
```

None

`None` is null in Python. `None` is like `#N/A` or `=na()` in Excel.

```
a = None
a is None
```

```
True
```

```
b = 5
b is not None
```

```
True
```

```
type(None)
```

```
NoneType
```

Control Flow

> Python has several built-in keywords for conditional logic, loops, and other standard control flow concepts found in other programming languages.

If you understand Excel's `if()`, then you understand Python's `if`, `elif`, and `else`.

if, elif, and else

```
x = -1
type(x)
```

```
int
```

```
if x < 0:
    print("It's negative")
```

```
It's negative
```

Single quotes and double quotes (`'` and `"`) are equivalent in Python. However, in the preceding code cell, we must use double quotes to differentiate between the enclosing quotes and the apostrophe in `It's`.

Python's `elif` avoids nested `if` statements. `elif` allows another `if` condition that is tested only if the preceding `if` and `elif` conditions were not `True`. An `else` runs if no other conditions are met.

```
x = 10
if x < 0:
    print("It's negative")
elif x == 0:
    print('Equal to zero')
elif 0 < x < 5:
```

```
    print('Positive but smaller than 5')
else:
    print('Positive and larger than or equal to 5')
```

```
Positive and larger than or equal to 5
```

We can combine comparisons with `and` and `or` (or `&` and `|`).

```
a = 5
b = 7
c = 8
d = 4
if (a < b) or (c > d):
    print('Made it')
```

```
Made it
```

for loops

We use `for` loops to loop over collections, like lists or tuples.

The `continue` keyword skips the remainder of the current iteration of the `for` loop, moving to the next iteration.

The `+=` operator adds and assigns values with one operator. That is, `a += 5` is an abbreviation for `a = a + 5`. There are equivalent operators for subtraction, multiplication, and division (i.e., `-=`, `*=`, and `/=`).

```
sequence = [1, 2, None, 4, None, 5, 'Alex']
total = 0
for value in sequence:
    if value is None or isinstance(value, str):
        continue
    total += value # the += operator is equivalent to "total = total + value"
```

```
total
```

```
12
```

The `break` keyword skips the remainder of the current and all remaining iterations of the `for` loop.

```
sequence = [1, 2, 0, 4, 6, 5, 2, 1]
total_until_5 = 0
for value in sequence:
    if value == 5:
        break
    total_until_5 += value
```

```
total_until_5
```

```
13
```

range

> The range function returns an iterator that yields a sequence of evenly spaced integers.

The `range()` function quickly and efficiently generates iterators for `for` loops.

- With one argument, `range()` creates an iterator from 0 to that number *but excludes that number*, so `range(10)` is an iterator that starts at 0, stops at 9, with a length of 10
- With two arguments, the first argument is the *included* start value, and the second argument is the *excluded* stop value
- With three arguments, the third argument is the iterator step size

```
range(10)
```

```
range(0, 10)
```

We can cast a range to a list.

```
list(range(10))
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Python intervals are "closed" (included) on the left and "open" (excluded) on the right. The following is an empty list because we cannot count from 5 to 0 by steps of +1.

```
list(range(5, 0))
```

```
[]
```

However, we can count from 5 to 0 in steps of -1.

```
list(range(5, 0, -1))
```

```
[5, 4, 3, 2, 1]
```

Ternary expressions

We can complete simple comparisons on one line in Python.

```
x = -5
value = 'Non-negative' if x >= 0 else 'Negative'
value
```

```
'Negative'
```

McKinney Chapter 2 - Practice - Blank

Announcements

Five-Minute Review

Practice

Extract the year, month, and day from an 8-digit date (i.e., YYYYMMDD format) using // (integer division) and % (modulo division).

```
lb = 20080915
```

Write a function `date` that takes an 8-digit date argument and returns a year, month, and date tuple (e.g., `return (year, month, day)`).

Write a function `date_2` that takes an 8-digit date as either integer or string.

Write a function `date_3` that takes a list of 8-digit dates as integers or strings.

Write a for loop that prints the squares of integers from 1 to 10.

Write a for loop that prints the squares of *even* integers from 1 to 10.

Write a for loop that sums the squares of integers from 1 to 10.

Write a for loop that sums the squares of integers from 1 to 10 but stops before the sum exceeds 50.

FizzBuzz

Solve FizzBuzz.

Use ternary expressions to make your FizzBuzz solution more compact.

Triangle

Write a function `triangle` that accepts a positive integer N and prints a numerical triangle of height $N - 1$. For example, `triangle(N=6)` should print:

```
1
22
333
4444
55555
```

Two Sum

Write a function `two_sum` that does the following.

Given a list of integers `nums` and an integer `target`, return the indices of the two numbers that add up to target.

You may assume that each input would have exactly one solution, and you may not use the same element twice.

You can return the answer in any order.

Here are some examples:

Example 1:

Input: `nums = [2,7,11,15]`, `target = 9`
Output: `[0,1]`
Explanation: Because `nums[0] + nums[1] == 9`, we return `[0, 1]`.

Example 2:

Input: `nums = [3,2,4]`, `target = 6`
Output: `[1,2]`

Example 3:

Input: `nums = [3,3]`, `target = 6`
Output: `[0,1]`

I saw this question on LeetCode.

Best Time

Write a function `best_time` that solves the following.

You are given a list `prices` where `prices[i]` is the price of a given stock on the i^{th} day.

You want to maximize your profit by choosing a single day to buy one stock and choosing a different day in the future to sell that stock.

Return the maximum profit you can achieve from this transaction. If you cannot achieve any profit, return 0.

Here are some examples:

Example 1:

Input: `prices = [7,1,5,3,6,4]`
Output: `5`
Explanation: Buy on day 2 (price = 1) and sell on day 5 (price = 6), profit = 6-1 = 5. Note that buying on day 2 and selling on day 1 is not allowed because you must buy before you sell.

Example 2:

Input: `prices = [7,6,4,3,1]`
Output: `0`
Explanation: In this case, no transactions are done and the max profit = 0.

I saw this question on LeetCode.

McKinney Chapter 2 - Practice - Sec 02

Announcements

1. Check your email inbox for an invitation to a free six-month subscription to DataCamp
    1. I added a few short courses to our course group
    2. These short courses are completely optional
    3. DataCamp has lots of resources to help you learn Python, R, SQL, Excel, etc.
2. Here are links to a few finance newsletters I strongly suggest:
    1. Matt Levine: https://www.bloomberg.com/account/newsletters/money-stuff
    2. Byrne Hobart: https://capitalgains.thediff.co/subscribe?ref=I0N1NGdmJq&_bhlid=7fecfad9eb7fd8bcdb529e945e11346b5897acdc
    3. Clifford Asness: https://www.aqr.com/Insights/Perspectives
    4. Owen Lamont: https://www.acadian-asset.com/investment-insights/owenomics#

Five-Minute Review

Practice

Extract the year, month, and day from an 8-digit date (i.e., YYYYMMDD format) using // (integer division) and % (modulo division).

```
lb = 20080915
```

```
lb
```

```
20080915
```

```
lb // 10_000 # // is integer division
```

```
2008
```

```python
lb % 10_000 # % is modulo or remainder division
```

```
915
```

```python
(lb % 10_000) // 100
```

```
9
```

```python
lb % 100
```

```
15
```

What happened here?

- Floor or integer division `//` drops the digits on the right side (one digit per zero)
- Modulo or remainder division `%` keeps the diggits on the right side (one digit per zero)

Here is solution that approximates Excel's `LEFT()`, `MID()`, and `RIGHT()`. This works, but is not very Pythonic.

```python
int(str(lb)[:4])
```

```
2008
```

```python
int(str(lb)[4:6])
```

```
9
```

```python
int(str(lb)[7:8])
```

```
5
```

Write a function `date` that takes an 8-digit date argument and returns a year, month, and date tuple (e.g., `return (year, month, day)`).

```
def date(ymd):
    year = ymd // 10_000 # // is integer division
    month = (ymd % 10_000) // 100
    day = ymd % 100
    return (year, month, day)
```

```
date(lb)
```

```
(2008, 9, 15)
```

```
date(20250110)
```

```
(2025, 1, 10)
```

Write a function `date_2` that takes an 8-digit date as either integer or string.

```
def date_2(ymd):
    # if type(ymd) is str:
    #
    if isinstance(ymd, str):
        ymd = int(ymd)
    return date(ymd)
```

The `isinstance(ymd, str)` is better than `type(ymd) == str` because the `isinstance()` function also tests for all sub-classes of `str`. More here: https://stackoverflow.com/questions/152580/whats-the-canonical-way-to-check-for-type-in-python/.

```
date_2(str(lb))
```

```
(2008, 9, 15)
```

```
date_2('20250110')
```

```
(2025, 1, 10)
```

Write a function `date_3` that takes a list of 8-digit dates as integers or strings.

```
dates_in = [20080915, 20250110]
```

```
def date_3(dates_in):
    dates_out = []
    for d in dates_in:
        # dates_out += [date_2(d)] # alternative
        dates_out.append(date_2(d))

    return dates_out
```

I have a slight preference for `.append()` over `+= []` because `.append()` modifies the list the list in place instead of making a copy. However, the speed differences in most cases in this course will be negligible. More here: https://www.geeksforgeeks.org/difference-between-and-append-in-python/.

```
date_3(dates_in)
```

```
[(2008, 9, 15), (2025, 1, 10)]
```

Write a for loop that prints the squares of integers from 1 to 10.

```
print('a', 'b', 'c', sep = '---')
```

```
a---b---c
```

```
for i in range(1, 11):
    print(i**2, end=' ')
```

```
1 4 9 16 25 36 49 64 81 100
```

Write a for loop that prints the squares of *even* integers from 1 to 10.

```python
for i in range(1, 11):
    if i % 2 == 0:
        print(i**2, end=' ')
```

```
4 16 36 64 100
```

```python
for i in range(2, 11, 2):
    print(i**2, end=' ')
```

```
4 16 36 64 100
```

Write a for loop that sums the squares of integers from 1 to 10.

```python
total = 0
for i in range(1, 11):
    total += i**2

total
```

```
385
```

Write a for loop that sums the squares of integers from 1 to 10 but stops before the sum exceeds 50.

```python
total = 0  # Initialize sum to zero

for i in range(1, 11):  # Loop from 1 to 10
    # Check if adding the square of i would exceed 50
    if (total + i**2) > 50:
        # 'break' exits the loop completely, stopping further iterations
        # 'continue' would skip to the next iteration without executing further code in this
        break

    # Add the square of i to total
    total += i**2

# Print the final sum (implicit return in this case since it is the last line in the code cel
total
```

```
30
```

FizzBuzz

Solve FizzBuzz.

Here is some pseudo code. The test for multiples of 3 and 5 must come first, otherwise it would never run!

```
# for i in range(1, 101):
#     # test for multiple of 3 & 5
#     #     print fizzbuzz
#     # test for multiple of 3
#     #     print fizz
#     # test for multiple of 5
#     #     print buzz
#     # otherwise print i
```

Here is my favorite FizzBuzz solution.

```
for i in range(1, 101):
    if (i % 3 == 0) & (i % 5 == 0):
        print('FizzBuzz', end=' ')
    elif (i % 3 == 0):
        print('Fizz', end=' ')
    elif (i % 5 == 0):
        print('Buzz', end=' ')
    else:
        print(i, end=' ')
```

```
1 2 Fizz 4 Buzz Fizz 7 8 Fizz Buzz 11 Fizz 13 14 FizzBuzz 16 17 Fizz 19 Buzz Fizz 22 23 Fizz
```

Use ternary expressions to make your FizzBuzz solution more compact.

Here is a compact FizzBuzz solution. I consider the solution above easier to read and troubleshoot. The compact solution below uses the trick that we can multiply a string by `True` to return the string itself or by or `False` to return an empty string.

```
for i in range(1, 101):
    print('Fizz'*(i%3==0) + 'Buzz'*(i%5==0) if (i%3==0) or (i%5==0) else i, end=' ')
```

```
1 2 Fizz 4 Buzz Fizz 7 8 Fizz Buzz 11 Fizz 13 14 FizzBuzz 16 17 Fizz 19 Buzz Fizz 22 23 Fizz
```

Here is *an even more compact* FizzBuzz solution. The trick below is that Python's `or` returns its first truthy value. - If the concatenated string (`'Fizz'*(i%3==0) + 'Buzz'*(i%5==0)`) is not an empty string, which is falsy in Python, the `or` evaluates to that string. - If the string is empty, which means `i` is not divisible by 3 or 5, the `or` evaluates to `i`.

```python
for i in range(1, 101):
    print('Fizz'*(i%3==0) + 'Buzz'*(i%5==0) or i, end=' ')
```

```
1 2 Fizz 4 Buzz Fizz 7 8 Fizz Buzz 11 Fizz 13 14 FizzBuzz 16 17 Fizz 19 Buzz Fizz 22 23 Fizz
```

Triangle

Write a function `triangle` that accepts a positive integer N and prints a numerical triangle of height $N-1$. For example, `triangle(N=6)` should print:

```
1
22
333
4444
55555
```

```python
def triangle(N):
    for i in range(1, N):
        print(str(i) * i)
```

```python
triangle(6)
```

```
1
22
333
4444
55555
```

The solution above works because a multiplying a string by `i` concatenates `i` copies of that string.

```python
'Test' + 'Test' + 'Test'
```

```
'TestTestTest'
```

```python
'Test' * 3
```

```
'TestTestTest'
```

Two Sum

Write a function `two_sum` that does the following.

Given a list of integers `nums` and an integer `target`, return the indices of the two numbers that add up to target.

You may assume that each input would have exactly one solution, and you may not use the same element twice.

You can return the answer in any order.

Here are some examples:

Example 1:

Input: `nums = [2,7,11,15]`, `target = 9`
Output: `[0,1]`
Explanation: Because `nums[0] + nums[1] == 9`, we return `[0, 1]`.

Example 2:

Input: `nums = [3,2,4]`, `target = 6`
Output: `[1,2]`

Example 3:

Input: `nums = [3,3]`, `target = 6`
Output: `[0,1]`

I saw this question on LeetCode.

```python
def two_sum(nums, target):
    for i in range(1, len(nums)):
        for j in range(i):
            if nums[i] + nums[j] == target:
                return [j, i]
```

```
two_sum(nums = [2,7,11,15], target = 9)
```

```
[0, 1]
```

```
two_sum(nums = [3,2,4], target = 6)
```

```
[1, 2]
```

```
two_sum(nums = [3,3], target = 6)
```

```
[0, 1]
```

Best Time

Write a function `best_time` that solves the following.

You are given a list `prices` where `prices[i]` is the price of a given stock on the i^{th} day.

You want to maximize your profit by choosing a single day to buy one stock and choosing a different day in the future to sell that stock.

Return the maximum profit you can achieve from this transaction. If you cannot achieve any profit, return 0.

Here are some examples:

Example 1:

Input: `prices = [7,1,5,3,6,4]`
Output: `5`
Explanation: Buy on day 2 (price = 1) and sell on day 5 (price = 6), profit = 6-1 = 5. Note that buying on day 2 and selling on day 1 is not allowed because you must buy before you sell.

Example 2:

Input: `prices = [7,6,4,3,1]`
Output: `0`
Explanation: In this case, no transactions are done and the max profit = 0.

I saw this question on LeetCode.

```python
def max_profit(prices):
    # We start by assuming the first price is the lowest we've seen so far
    min_price = prices[0]

    # We initialize our maximum profit to zero, as no profit has been calculated yet
    max_profit = 0

    # Loop through each price in the list of prices
    for price in prices:
        # If the current price is lower than our lowest price seen, update min_price
        min_price = price if price < min_price else min_price

        # Calculate the profit if we were to sell at the current price
        profit = price - min_price

        # If this profit is better than our max profit so far, update max_profit
        max_profit = profit if profit > max_profit else max_profit

    # After checking all prices, return the maximum profit we've found
    return max_profit
```

```python
max_profit(prices=[7,1,5,3,6,4])
```

```
5
```

```python
max_profit(prices=[7,6,4,3,1])
```

```
0
```

We could replace the ternary statements with the `min()` and `max()` functions for a little more compact code.

```python
def max_profit_2(prices):
    min_price = prices[0]
    max_profit = 0

    for price in prices:
        # Update min_price if current price is lower
        min_price = min(min_price, price)
```

```
        # Calculate profit by selling at the current price
        current_profit = price - min_price

        # Update max_profit if the current_profit is higher
        max_profit = max(max_profit, current_profit)

    return max_profit
```

```
max_profit_2(prices=[7,1,5,3,6,4])
```

```
5
```

```
max_profit_2(prices=[7,6,4,3,1])
```

```
0
```

McKinney Chapter 2 - Practice - Sec 03

Announcements

1. Check your email inbox for an invitation to a free six-month subscription to DataCamp
    1. I added a few short courses to our course group
    2. These short courses are completely optional
    3. DataCamp has lots of resources to help you learn Python, R, SQL, Excel, etc.
2. Here are links to a few finance newsletters I strongly suggest:
    1. Matt Levine: https://www.bloomberg.com/account/newsletters/money-stuff
    2. Byrne Hobart: https://capitalgains.thediff.co/subscribe?ref=I0N1NGdmJq&_bhlid=7fecfad9eb7fd8bcdb529e945e11346b5897acdc
    3. Clifford Asness: https://www.aqr.com/Insights/Perspectives
    4. Owen Lamont: https://www.acadian-asset.com/investment-insights/owenomics#

Five-Minute Review

Practice

Extract the year, month, and day from an 8-digit date (i.e., YYYYMMDD format) using // (integer division) and % (modulo division).

```
lb = 20080915
```

```
lb
```

```
20080915
```

```
lb // 10_000 # // is integer division
```

```
2008
```

```
lb % 10_000 # % is modulo or remainder division
```

```
915
```

```
(lb % 10_000) // 100
```

```
9
```

```
lb % 100
```

```
15
```

What happened here?

- Floor or integer division `//` drops the digits on the right side (one digit per zero)
- Modulo or remainder division `%` keeps the diggits on the right side (one digit per zero)

Here is solution that approximates Excel's `LEFT()`, `MID()`, and `RIGHT()`. This works, but is not very Pythonic.

```
int(str(lb)[:4])
```

```
2008
```

```
int(str(lb)[4:6])
```

```
9
```

```
int(str(lb)[7:8])
```

```
5
```

Write a function `date` that takes an 8-digit date argument and returns a year, month, and date tuple (e.g., `return (year, month, day)`).

```
def date(x):
    year = x // 10_000 # // is integer division
    month = (x % 10_000) // 100
    day = x % 100
    return (year, month, day)
```

```
date(20250107)
```

```
(2025, 1, 7)
```

Write a function `date_2` that takes an 8-digit date as either integer or string.

```
def date_2(x):
    # if type(x) is str:
    if isinstance(x, str):
        x = int(x)

    return date(x)
```

```
date_2(str(lb))
```

```
(2008, 9, 15)
```

```
date_2(lb)
```

```
(2008, 9, 15)
```

```
date_2('20250110')
```

```
(2025, 1, 10)
```

Write a function `date_3` that takes a list of 8-digit dates as integers or strings.

```
ymds = [20080915, '20250110']
ymds
```

```
[20080915, '20250110']
```

This markdown cell is for *italicized* and **bold** text!

```
def date_3(ymds):
    ymds_out = []
    for ymd in ymds:
        ymds_out.append(date_2(ymd))

    return ymds_out
```

```
date_3(ymds)
```

```
[(2008, 9, 15), (2025, 1, 10)]
```

Write a for loop that prints the squares of integers from 1 to 10.

```
print('a', 'b', 'c', sep = '---')
```

```
a---b---c
```

```
for i in range(1, 11):
    print(i**2, end=' ')
```

```
1 4 9 16 25 36 49 64 81 100
```

Write a for loop that prints the squares of *even* integers from 1 to 10.

```
for i in range(1, 11):
    if i % 2 == 0:
        print(i**2, end=' ')
```

```
4 16 36 64 100
```

```python
for i in range(2, 11, 2):
    print(i**2, end=' ')
```

```
4 16 36 64 100
```

Write a for loop that sums the squares of integers from 1 to 10.

```python
total = 0
for i in range(1, 11):
    total += i**2

total
```

```
385
```

Write a for loop that sums the squares of integers from 1 to 10 but stops before the sum exceeds 50.

```python
total = 0  # Initialize sum to zero

for i in range(1, 11):  # Loop from 1 to 10
    # Check if adding the square of i would exceed 50
    if (total + i**2) > 50:
        # 'break' exits the loop completely, stopping further iterations
        # 'continue' would skip to the next iteration without executing further code in this
        break

    # Add the square of i to total
    total += i**2

# Print the final sum (implicit return in this case since it is the last line in the code cel
total
```

```
30
```

FizzBuzz

Solve FizzBuzz.

Here is some pseudo code. The test for multiples of 3 and 5 must come first, otherwise it would never run!

```
# for i in range(1, 101):
#     # test for multiple of 3 & 5
#     #     print fizzbuzz
#     # test for multiple of 3
#     #     print fizz
#     # test for multiple of 5
#     #     print buzz
#     # otherwise print i
```

Here is my favorite FizzBuzz solution.

```
for i in range(1, 101):
    if (i % 3 == 0) & (i % 5 == 0):
        print('FizzBuzz', end=' ')
    elif (i % 3 == 0):
        print('Fizz', end=' ')
    elif (i % 5 == 0):
        print('Buzz', end=' ')
    else:
        print(i, end=' ')
```

```
1 2 Fizz 4 Buzz Fizz 7 8 Fizz Buzz 11 Fizz 13 14 FizzBuzz 16 17 Fizz 19 Buzz Fizz 22 23 Fizz
```

Use ternary expressions to make your FizzBuzz solution more compact.

Here is a compact FizzBuzz solution. I consider the solution above easier to read and troubleshoot. The compact solution below uses the trick that we can multiply a string by `True` to return the string itself or by or `False` to return an empty string.

```
for i in range(1, 101):
    print('Fizz'*(i%3==0) + 'Buzz'*(i%5==0) if (i%3==0) or (i%5==0) else i, end=' ')
```

```
1 2 Fizz 4 Buzz Fizz 7 8 Fizz Buzz 11 Fizz 13 14 FizzBuzz 16 17 Fizz 19 Buzz Fizz 22 23 Fizz
```

Here is *an even more compact* FizzBuzz solution. The trick below is that Python's `or` returns its first truthy value. - If the concatenated string (`'Fizz'*(i%3==0) + 'Buzz'*(i%5==0)`) is not an empty string, which is falsy in Python, the `or` evaluates to that string. - If the string is empty, which means `i` is not divisible by 3 or 5, the `or` evaluates to `i`.

```
for i in range(1, 101):
    print('Fizz'*(i%3==0) + 'Buzz'*(i%5==0) or i, end=' ')
```

```
1 2 Fizz 4 Buzz Fizz 7 8 Fizz Buzz 11 Fizz 13 14 FizzBuzz 16 17 Fizz 19 Buzz Fizz 22 23 Fizz
```

Triangle

Write a function `triangle` that accepts a positive integer N and prints a numerical triangle of height $N-1$. For example, `triangle(N=6)` should print:

```
1
22
333
4444
55555
```

```
def triangle(N):
    for i in range(1, N):
        print(str(i) * i)
```

```
triangle(6)
```

```
1
22
333
4444
55555
```

The solution above works because a multiplying a string by `i` concatenates `i` copies of that string.

```
'Test' + 'Test' + 'Test'
```

```
'TestTestTest'
```

```python
'Test' * 3
```

```
'TestTestTest'
```

Two Sum

Write a function `two_sum` that does the following.

Given a list of integers `nums` and an integer `target`, return the indices of the two numbers that add up to target.

You may assume that each input would have exactly one solution, and you may not use the same element twice.

You can return the answer in any order.

Here are some examples:

Example 1:

Input: `nums = [2,7,11,15]`, `target = 9`
Output: `[0,1]`
Explanation: Because `nums[0] + nums[1] == 9`, we return `[0, 1]`.

Example 2:

Input: `nums = [3,2,4]`, `target = 6`
Output: `[1,2]`

Example 3:

Input: `nums = [3,3]`, `target = 6`
Output: `[0,1]`

I saw this question on LeetCode.

```python
def two_sum(nums, target):
    for i in range(1, len(nums)):
        for j in range(i):
            if nums[i] + nums[j] == target:
                return [j, i]
```

```python
two_sum(nums = [2,7,11,15], target = 9)
```

```
[0, 1]
```

```
two_sum(nums = [3,2,4], target = 6)
```

```
[1, 2]
```

```
two_sum(nums = [3,3], target = 6)
```

```
[0, 1]
```

Best Time

Write a function `best_time` that solves the following.

You are given a list `prices` where `prices[i]` is the price of a given stock on the i^{th} day.

You want to maximize your profit by choosing a single day to buy one stock and choosing a different day in the future to sell that stock.

Return the maximum profit you can achieve from this transaction. If you cannot achieve any profit, return 0.

Here are some examples:

Example 1:

Input: `prices = [7,1,5,3,6,4]`
Output: `5`
Explanation: Buy on day 2 (price = 1) and sell on day 5 (price = 6), profit = 6-1 = 5. Note that buying on day 2 and selling on day 1 is not allowed because you must buy before you sell.

Example 2:

Input: `prices = [7,6,4,3,1]`
Output: `0`
Explanation: In this case, no transactions are done and the max profit = 0.

I saw this question on LeetCode.

```
def max_profit(prices):
    # We start by assuming the first price is the lowest we've seen so far
    min_price = prices[0]

    # We initialize our maximum profit to zero, as no profit has been calculated yet
    max_profit = 0
```

```
    # Loop through each price in the list of prices
    for price in prices:
        # If the current price is lower than our lowest price seen, update min_price
        min_price = price if price < min_price else min_price

        # Calculate the profit if we were to sell at the current price
        profit = price - min_price

        # If this profit is better than our max profit so far, update max_profit
        max_profit = profit if profit > max_profit else max_profit

    # After checking all prices, return the maximum profit we've found
    return max_profit
```

```
max_profit(prices=[7,1,5,3,6,4])
```

```
5
```

```
max_profit(prices=[7,6,4,3,1])
```

```
0
```

We could replace the ternary statements with the `min()` and `max()` functions for a little more compact code.

```
def max_profit_2(prices):
    min_price = prices[0]
    max_profit = 0

    for price in prices:
        # Update min_price if current price is lower
        min_price = min(min_price, price)

        # Calculate profit by selling at the current price
        current_profit = price - min_price

        # Update max_profit if the current_profit is higher
        max_profit = max(max_profit, current_profit)

    return max_profit
```

```
max_profit_2(prices=[7,1,5,3,6,4])
```

```
5
```

```
max_profit_2(prices=[7,6,4,3,1])
```

```
0
```

McKinney Chapter 2 - Practice - Sec 04

HELLO! NIRAMAY!

Announcements

1. Check your email inbox for an invitation to a free six-month subscription to DataCamp
    1. I added a few short courses to our course group
    2. These short courses are completely optional
    3. DataCamp has lots of resources to help you learn Python, R, SQL, Excel, etc.
2. Here are links to a few finance newsletters I strongly suggest:
    1. Matt Levine: https://www.bloomberg.com/account/newsletters/money-stuff
    2. Byrne Hobart: https://capitalgains.thediff.co/subscribe?ref=I0N1NGdmJq&_bhlid=7fecfad9eb7fd8bcdb529e945e11346b5897acdc
    3. Clifford Asness: https://www.aqr.com/Insights/Perspectives
    4. Owen Lamont: https://www.acadian-asset.com/investment-insights/owenomics#

Five-Minute Review

Practice

Extract the year, month, and day from an 8-digit date (i.e., YYYYMMDD format) using // (integer division) and % (modulo division).

```python
lb = 20080915
```

```python
lb
```

```
20080915
```

```
lb // 10_000 # // is integer division
```

```
2008
```

```
lb % 10_000 # % is modulo or remainder division
```

```
915
```

```
(lb % 10_000) // 100
```

```
9
```

```
lb % 100
```

```
15
```

What happened here?

- Floor or integer division `//` drops the digits on the right side (one digit per zero)
- Modulo or remainder division `%` keeps the diggits on the right side (one digit per zero)

Here is solution that approximates Excel's `LEFT()`, `MID()`, and `RIGHT()`. This works, but is not very Pythonic.

```
int(str(lb)[:4])
```

```
2008
```

```
int(str(lb)[4:6])
```

```
9
```

```
int(str(lb)[7:8])
```

```
5
```

Write a function `date` that takes an 8-digit date argument and returns a year, month, and date tuple (e.g., `return (year, month, day)`).

```python
import this
```

```
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

```python
def date(ymd):
    year = ymd // 10_000 # // is integer division
    month = (ymd % 10_000) // 100
    day = ymd % 100
    return (year, month, day)
```

```python
lb
```

```
20080915
```

```python
%who
```

```
date     lb   this
```

```
date(lb)
```

```
(2008, 9, 15)
```

```
date(20250110)
```

```
(2025, 1, 10)
```

```
type(date(20250110))
```

```
tuple
```

Write a function `date_2` that takes an 8-digit date as either integer or string.

```
def date_2(ymd):
    # if type(ymd) is str:
    if isinstance(ymd, str):
        ymd = int(ymd)
    return date(ymd)
```

```
date_2(lb)
```

```
(2008, 9, 15)
```

```
date_2(str(lb))
```

```
(2008, 9, 15)
```

```
date_2(20250110)
```

```
(2025, 1, 10)
```

```
date_2('20250110')
```

```
(2025, 1, 10)
```

Write a function `date_3` that takes a list of 8-digit dates as integers or strings.

```
ymds = [20080915, '20250110']
ymds
```

```
[20080915, '20250110']
```

This markdown cell is for *italicized* and **bold** text!

```
def date_3(ymds):
    ymds_out = []
    for ymd in ymds:
        ymds_out.append(date_2(ymd))

    return ymds_out
```

```
date_3(ymds)
```

```
[(2008, 9, 15), (2025, 1, 10)]
```

Write a for loop that prints the squares of integers from 1 to 10.

```
print(1, 2, 3, sep='---')
```

```
1---2---3
```

```
for i in range(1, 11):
    print(i**2, end=' ')
```

```
1 4 9 16 25 36 49 64 81 100
```

Write a for loop that prints the squares of *even* integers from 1 to 10.

```python
for i in range(1, 11):
    if i % 2 == 0:
        print(i**2, end=' ')
```

```
4 16 36 64 100
```

```python
for i in range(2, 11, 2):
    print(i**2, end=' ')
```

```
4 16 36 64 100
```

Write a for loop that sums the squares of integers from 1 to 10.

```python
total = 0
for i in range(1, 11):
    total += i**2

total
```

```
385
```

Write a for loop that sums the squares of integers from 1 to 10 but stops before the sum exceeds 50.

```python
total = 0  # Initialize sum to zero

for i in range(1, 11):  # Loop from 1 to 10
    # Check if adding the square of i would exceed 50
    if (total + i**2) > 50:
        # 'break' exits the loop completely, stopping further iterations
        # 'continue' would skip to the next iteration without executing further code in this
        break

    # Add the square of i to total
    total += i**2

# Print the final sum (implicit return in this case since it is the last line in the code cel
total
```

30

FizzBuzz

Solve FizzBuzz.

Here is some pseudo code. The test for multiples of 3 and 5 must come first, otherwise it would never run!

```
# for i in range(1, 101):
#     # test for multiple of 3 & 5
#     #     print fizzbuzz
#     # test for multiple of 3
#     #     print fizz
#     # test for multiple of 5
#     #     print buzz
#     # otherwise print i
```

Here is my favorite FizzBuzz solution.

```
for i in range(1, 101):
    if (i % 3 == 0) & (i % 5 == 0):
        print('FizzBuzz', end=' ')
    elif (i % 3 == 0):
        print('Fizz', end=' ')
    elif (i % 5 == 0):
        print('Buzz', end=' ')
    else:
        print(i, end=' ')
```

```
1 2 Fizz 4 Buzz Fizz 7 8 Fizz Buzz 11 Fizz 13 14 FizzBuzz 16 17 Fizz 19 Buzz Fizz 22 23 Fizz
```

Use ternary expressions to make your FizzBuzz solution more compact.

Here is a compact FizzBuzz solution. I consider the solution above easier to read and troubleshoot. The compact solution below uses the trick that we can multiply a string by `True` to return the string itself or by or `False` to return an empty string.

```
for i in range(1, 101):
    print('Fizz'*(i%3==0) + 'Buzz'*(i%5==0) if (i%3==0) or (i%5==0) else i, end=' ')
```

```
1 2 Fizz 4 Buzz Fizz 7 8 Fizz Buzz 11 Fizz 13 14 FizzBuzz 16 17 Fizz 19 Buzz Fizz 22 23 Fizz
```

Here is *an even more compact* FizzBuzz solution. The trick below is that Python's `or` returns its first truthy value. - If the concatenated string (`'Fizz'*(i%3==0) + 'Buzz'*(i%5==0)`) is not an empty string, which is falsy in Python, the `or` evaluates to that string. - If the string is empty, which means `i` is not divisible by 3 or 5, the `or` evaluates to `i`.

```
for i in range(1, 101):
    print('Fizz'*(i%3==0) + 'Buzz'*(i%5==0) or i, end=' ')
```

```
1 2 Fizz 4 Buzz Fizz 7 8 Fizz Buzz 11 Fizz 13 14 FizzBuzz 16 17 Fizz 19 Buzz Fizz 22 23 Fizz
```

Triangle

Write a function `triangle` that accepts a positive integer N and prints a numerical triangle of height $N - 1$. For example, `triangle(N=6)` should print:

```
1
22
333
4444
55555
```

```
def triangle(N):
    for i in range(1, N):
        print(str(i) * i)
```

```
triangle(6)
```

```
1
22
333
4444
55555
```

The solution above works because a multiplying a string by `i` concatenates `i` copies of that string.

```python
'Test' + 'Test' + 'Test'
```

```
'TestTestTest'
```

```python
'Test' * 3
```

```
'TestTestTest'
```

Two Sum

Write a function `two_sum` that does the following.

Given a list of integers `nums` and an integer `target`, return the indices of the two numbers that add up to target.

You may assume that each input would have exactly one solution, and you may not use the same element twice.

You can return the answer in any order.

Here are some examples:

Example 1:

Input: `nums = [2,7,11,15]`, `target = 9`
Output: `[0,1]`
Explanation: Because `nums[0] + nums[1] == 9`, we return `[0, 1]`.

Example 2:

Input: `nums = [3,2,4]`, `target = 6`
Output: `[1,2]`

Example 3:

Input: `nums = [3,3]`, `target = 6`
Output: `[0,1]`

I saw this question on LeetCode.

```python
def two_sum(nums, target):
    for i in range(1, len(nums)):
        for j in range(i):
            if nums[i] + nums[j] == target:
                return [j, i]
```

```
two_sum(nums = [2,7,11,15], target = 9)
```

```
[0, 1]
```

```
two_sum(nums = [3,2,4], target = 6)
```

```
[1, 2]
```

```
two_sum(nums = [3,3], target = 6)
```

```
[0, 1]
```

Best Time

Write a function `best_time` that solves the following.

You are given a list `prices` where `prices[i]` is the price of a given stock on the i^{th} day.

You want to maximize your profit by choosing a single day to buy one stock and choosing a different day in the future to sell that stock.

Return the maximum profit you can achieve from this transaction. If you cannot achieve any profit, return 0.

Here are some examples:

Example 1:

Input: `prices = [7,1,5,3,6,4]`
Output: `5`
Explanation: Buy on day 2 (price = 1) and sell on day 5 (price = 6), profit = 6-1 = 5. Note that buying on day 2 and selling on day 1 is not allowed because you must buy before you sell.

Example 2:

Input: `prices = [7,6,4,3,1]`
Output: `0`
Explanation: In this case, no transactions are done and the max profit = 0.

I saw this question on LeetCode.

```
def max_profit(prices):
    # We start by assuming the first price is the lowest we've seen so far
    min_price = prices[0]

    # We initialize our maximum profit to zero, as no profit has been calculated yet
    max_profit = 0

    # Loop through each price in the list of prices
    for price in prices:
        # If the current price is lower than our lowest price seen, update min_price
        min_price = price if price < min_price else min_price

        # Calculate the profit if we were to sell at the current price
        profit = price - min_price

        # If this profit is better than our max profit so far, update max_profit
        max_profit = profit if profit > max_profit else max_profit

    # After checking all prices, return the maximum profit we've found
    return max_profit
```

```
max_profit(prices=[7,1,5,3,6,4])
```

```
5
```

```
max_profit(prices=[7,6,4,3,1])
```

```
0
```

We could replace the ternary statements with the `min()` and `max()` functions for a little more compact code.

```
def max_profit_2(prices):
    min_price = prices[0]
    max_profit = 0

    for price in prices:
        # Update min_price if current price is lower
        min_price = min(min_price, price)
```

```
        # Calculate profit by selling at the current price
        current_profit = price - min_price

        # Update max_profit if the current_profit is higher
        max_profit = max(max_profit, current_profit)

    return max_profit
```

```
max_profit_2(prices=[7,1,5,3,6,4])
```

```
5
```

```
max_profit_2(prices=[7,6,4,3,1])
```

```
0
```

Week 2

McKinney Chapter 3 - Built-In Data Structures, Functions, and Files

Introduction

We must understand Python's core functionality to use NumPy and pandas. Chapter 3 of McKinney (2022) discusses Python's core functionality. We will focus on the following:

1. Data structures
    1. tuples
    2. lists
    3. dicts (also known as dictionaries)
2. List comprehensions
3. Functions
    1. Returning multiple values
    2. Using anonymous functions

Note: Indented block quotes are from McKinney (2022) unless otherwise indicated. The section numbers here differ from McKinney (2022) because we will only discuss some topics.

Data Structures and Sequences

> Python's data structures are simple but powerful. Mastering their use is a critical part of becoming a proficient Python programmer.

Tuple

> A tuple is a fixed-length, immutable sequence of Python objects.

We cannot change a tuple after we create it because tuples are immutable. A tuple is ordered, so we can subset or slice it with a numerical index. We will surround tuples with parentheses but they are not required.

```
tup = (4, 5, 6)
```

Python is zero-indexed, so zero accesses the first element in tup!

```
tup[0]
```

```
4
```

```
tup[1]
```

```
5
```

```
tup[2]
```

```
6
```

```
nested_tup = ((4, 5, 6), (7, 8))
```

```
nested_tup[0]
```

```
(4, 5, 6)
```

```
nested_tup[0][0]
```

```
4
```

```
tup = ('foo', [1, 2], True)
```

If an object inside a tuple is mutable, such as a list, you can modify it in-place.

```
# tup[2] = False # gives an error, because tuples are immutable (unchangeable)
```

```
tup[1].append(3)
tup
```

```
('foo', [1, 2, 3], True)
```

You can concatenate tuples using the + operator to produce longer tuples:

Tuples are immutable, but we can combine two tuples into a new tuple.

```
(1, 2) + (1, 2)
```

```
(1, 2, 1, 2)
```

```
(4, None, 'foo') + (6, 0) + ('bar',)
```

```
(4, None, 'foo', 6, 0, 'bar')
```

> Multiplying a tuple by an integer, as with lists, has the effect of concatenating together that many copies of the tuple:

This multiplication behavior is the logical extension of the addition behavior above. The output of `tup + tup` should be the same as that of `2 * tup`.

```
('foo', 'bar') + ('foo', 'bar')
```

```
('foo', 'bar', 'foo', 'bar')
```

```
('foo', 'bar') * 2
```

```
('foo', 'bar', 'foo', 'bar')
```

Unpacking tuples

> If you try to assign to a tuple-like expression of variables, Python will attempt to unpack the value on the righthand side of the equals sign.

```
tup = (4, 5, 6)
a, b, c = tup
```

```
a
```

```
4
```

```
b
```

```
5
```

```
c
```

```
6
```

```
(d, e, f) = (7, 8, 9) # the parentheses are optional but helpful!
```

```
d
```

```
7
```

```
e
```

```
8
```

```
f
```

```
9
```

We can unpack nested tuples!

```
tup = 4, 5, (6, 7)
a, b, (c, d) = tup
```

Tuple methods

> Since the size and contents of a tuple cannot be modified, it is very light on instance methods. A particularly useful one (also available on lists) is count, which counts the number of occurrences of a value.

```
a = (1, 2, 2, 2, 3, 4, 2)
a.count(2)
```

```
4
```

Python is zero-indexed!

```
a.index(2)
```

```
1
```

List

In contrast with tuples, lists are variable-length and their contents can be modified in-place. You can define them using square brackets [ ] or using the list type function.

```
a_list = [2, 3, 7, None]
tup = ('foo', 'bar', 'baz')
b_list = list(tup)
```

```
a_list
```

```
[2, 3, 7, None]
```

```
b_list
```

```
['foo', 'bar', 'baz']
```

Python is zero-indexed!

```
a_list[0]
```

```
2
```

Concatenating and combining lists

Similar to tuples, adding two lists together with + concatenates them.

```
[4, None, 'foo'] + [7, 8, (2, 3)]
```

```
[4, None, 'foo', 7, 8, (2, 3)]
```

The `.append()` method adds its argument as the last element in a list.

```
xx = [4, None, 'foo']
xx.append([7, 8, (2, 3)])
xx
```

```
[4, None, 'foo', [7, 8, (2, 3)]]
```

> If you have a list already defined, you can append multiple elements to it using the extend method.

```
x = [4, None, 'foo']
x.extend([7, 8, (2, 3)])
x
```

```
[4, None, 'foo', 7, 8, (2, 3)]
```

Check your output! It will take you time to understand all these methods!

Slicing

Slicing is very important!

> You can select sections of most sequence types by using slice notation, which in its basic form consists of start:stop passed to the indexing operator [ ].

Recall that Python is zero-indexed, so the first element has an index of 0. A consequence of zero-indexing is that start:stop is inclusive on the left edge (start) and exclusive on the right edge (stop).

```
seq = [7, 2, 3, 7, 5, 6, 0, 1]
seq
```

```
[7, 2, 3, 7, 5, 6, 0, 1]
```

```
seq[5]
```

6

Python is zero-indexed, so left edge of slide is included and right edge is excluded!

```python
seq[1:5]
```

```
[2, 3, 7, 5]
```

Either the start or stop can be omitted, in which case they default to the start of the sequence and the end of the sequence, respectively.

```python
seq[:5]
```

```
[7, 2, 3, 7, 5]
```

```python
seq[3:]
```

```
[7, 5, 6, 0, 1]
```

Negative indices slice the sequence relative to the end.

```python
seq[-1]
```

```
1
```

```python
seq[-1:]
```

```
[1]
```

```python
seq[-4:]
```

```
[5, 6, 0, 1]
```

```python
seq[-4:-1]
```

```
[5, 6, 0]
```

```python
seq[-6:-2]
```

```
[3, 7, 5, 6]
```

A step can also be used after a second colon to, say, take every other element.

```
seq
```

```
[7, 2, 3, 7, 5, 6, 0, 1]
```

```
seq[::2]
```

```
[7, 3, 5, 0]
```

```
seq[1::2]
```

```
[2, 7, 6, 1]
```

We can think of the trailing `:2` in the preceding code cells as "count by 2". Therefore, the `1::2` slice:

- Starts at `1`
- Stops at the end because of the first `:`
- Counts by 2 becauase of the trailing `:2`

  A clever use of this is to pass -1, which has the useful effect of reversing a list or tuple.

```
seq[::-1]
```

```
[1, 0, 6, 5, 7, 3, 2, 7]
```

We will use slicing (subsetting) all semester, so we must understand the examples above.

dict

> dict is likely the most important built-in Python data structure. A more common name for it is hash map or associative array. It is a flexibly sized collection of key-value pairs, where key and value are Python objects. One approach for creating one is to use curly braces {} and colons to separate keys and values.

Elements in dictionaries have named keys, while elements in tuples and lists have numerical indices. Dictionaries are handy for passing named arguments and returning named results.

```
empty_dict = {}
empty_dict
```

```
{}
```

A dictionary is a set of key-value pairs.

```
d1 = {'a': 'some value', 'b': [1, 2, 3, 4]}
d1
```

```
{'a': 'some value', 'b': [1, 2, 3, 4]}
```

```
d1['a']
```

```
'some value'
```

```
d1[7] = 'an integer'
d1
```

```
{'a': 'some value', 'b': [1, 2, 3, 4], 7: 'an integer'}
```

We access dictionary values by key names instead of key positions.

> You can delete values either using the del keyword or the pop method (which simultaneously returns the value and deletes the key).

```
d1[5] = 'some value'
d1['dummy'] = 'another value'
d1
```

```
{'a': 'some value',
 'b': [1, 2, 3, 4],
 7: 'an integer',
 5: 'some value',
 'dummy': 'another value'}
```

```
del d1[5]
d1
```

```
{'a': 'some value',
 'b': [1, 2, 3, 4],
 7: 'an integer',
 'dummy': 'another value'}
```

```
ret = d1.pop('dummy')
```

```
ret
```

```
'another value'
```

```
d1
```

```
{'a': 'some value', 'b': [1, 2, 3, 4], 7: 'an integer'}
```

> The keys and values method give you iterators of the dict's keys and values, respectively. While the key-value pairs are not in any particular order, these functions output the keys and values in the same order.

```
d1.keys()
```

```
dict_keys(['a', 'b', 7])
```

```
d1.values()
```

```
dict_values(['some value', [1, 2, 3, 4], 'an integer'])
```

List, Set, and Dict Comprehensions

We will focus on list comprehensions, which are Pythonic.

> List comprehensions are one of the most-loved Python language features. They allow you to concisely form a new list by filtering the elements of a collection, transforming the elements passing the filter in one concise expression. They take the basic form:
>
> ```
> [expr for val in collection if condition]
> ```
>
> This is equivalent to the following for loop:

```
result = []
for val in collection:
    if condition:
        result.append(expr)
```

The filter condition can be omitted, leaving only the expression.

```
strings = ['a', 'as', 'bat', 'car', 'dove', 'python']
```

We could use a `for` loop over `strings` to keep only strings longer than two and then to capitalize them.

```
caps = []
for x in strings:
    if len(x) > 2:
        caps.append(x.upper())

caps
```

```
['BAT', 'CAR', 'DOVE', 'PYTHON']
```

A list comprehension is more Pythonic and replaces four lines of code with one. The general format of a list comprehension is `[operation on x for x in list if condition]`

```
[x.upper() for x in strings if len(x) > 2]
```

```
['BAT', 'CAR', 'DOVE', 'PYTHON']
```

Here is another example. The following code is a `for` loop and an equivalent list comprehension that squares the integers from 1 to 10.

```
squares = []
for i in range(1, 11):
    squares.append(i ** 2)

squares
```

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

```
[i**2 for i in range(1, 11)]
```

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

What if we wanted the squares of even numbers?

```
[i**2 for i in range(1, 11) if i%2==0]
```

```
[4, 16, 36, 64, 100]
```

```
[i**2 for i in range(2, 11, 2)]
```

```
[4, 16, 36, 64, 100]
```

Functions

Functions are the primary and most important method of code organization and reuse in Python. As a rule of thumb, if you anticipate needing to repeat the same or very similar code more than once, it may be worth writing a reusable function. Functions can also help make your code more readable by giving a name to a group of Python statements.

Functions are declared with the def keyword and returned from with the return keyword:

```
def my_function(x, y, z=1.5):
    if z > 1:
        return z * (x + y)
    else:
        return z / (x + y)
```

There is no issue with having multiple return statements. If Python reaches the end of a function without encountering a return statement, None is returned automatically.

Each function can have positional arguments and keyword arguments. Keyword arguments are most commonly used to specify default values or optional arguments. In the preceding function, x and y are positional arguments while z is a keyword argument. This means that the function can be called in any of these ways:

```
my_function(5, 6, z=0.7)
my_function(3.14, 7, 3.5)
my_function(10, 20)
```

The main restriction on function arguments is that the keyword arguments must follow the positional arguments (if any). You can specify keyword arguments in any order; this frees you from having to remember which order the function arguments were specified in and only what their names are.

Returning Multiple Values

We can write Python functions that return multiple objects. The function `f()` below returns one tuple that we can unpack to multiple objects.

```
def f():
    a = 5
    b = 6
    c = 7
    return (a, b, c)
```

```
f()
```

```
(5, 6, 7)
```

If we want to return multiple objects with names or labels, we can return a dictionary.

```
def f():
    a = 5
    b = 6
    c = 7
    return {'a' : a, 'b' : b, 'c' : c}
```

```
f()
```

```
{'a': 5, 'b': 6, 'c': 7}
```

```
f()['a']
```

```
5
```

Anonymous (Lambda) Functions

> Python has support for so-called anonymous or lambda functions, which are a way of writing functions consisting of a single statement, the result of which is the return value. They are defined with the lambda keyword, which has no meaning other than "we are declaring an anonymous function."
>
> I usually refer to these as lambda functions in the rest of the book. They are especially convenient in data analysis because, as you'll see, there are many cases where data transformation functions will take functions as arguments. It's often less typing (and clearer) to pass a lambda function as opposed to writing a full-out function declaration or even assigning the lambda function to a local variable.

Lambda functions are Pythonic and let us to write simple functions on the fly.

```
strings = ['foo', 'card', 'bar', 'aaaa', 'abab']
```

```
strings.sort()
strings
```

```
['aaaa', 'abab', 'bar', 'card', 'foo']
```

```
len(strings[0])
```

```
4
```

```
strings.sort(key=len)
strings
```

```
['bar', 'foo', 'aaaa', 'abab', 'card']
```

For example, we could use a lambda function to sort `strings` by the last letter of each string.

```
strings.sort(key=lambda x: x[-1])
strings
```

```
['aaaa', 'abab', 'card', 'foo', 'bar']
```

What if we want to sort by the *second* to last letter?

```
strings.sort(key=lambda x: x[-2])
strings
```

```
['aaaa', 'abab', 'bar', 'foo', 'card']
```

McKinney Chapter 3 - Practice - Blank

Announcements

Five-Minute Review

Practice

Swap the values assigned to a and b using a third variable c.

```
a = 1
```

```
b = 2
```

Swap the values assigned to a and b *without* using a third variable c.

```
a = 1
```

```
b = 2
```

What is the output of the following code and why?

```
1, 1, 1 == (1, 1, 1)
```

```
(1, 1, False)
```

Create a list `l1` of integers from 1 to 100.

Slice `l1` to create a list `l2` of integers from 60 to 50 (inclusive).

Create a list `l3` of odd integers from 1 to 21.

Create a list `l4` of the squares of integers from 1 to 100.

Create a list `l5` that contains the squares of *odd* integers from 1 to 100.

Use a lambda function to sort `strings` by the last letter in each string.

```
strings = ['Clemson', 'Hinson', 'Pillsbury', 'Shubrick']
```

Given an integer array `nums` and an integer `k`, write a function to return the k^{th} largest element in the array.

Note that it is the k^{th} largest element in the sorted order, not the k^{th} distinct element.

Example 1:

Input: `nums = [3,2,1,5,6,4]`, `k = 2`
Output: `5`

Example 2:

Input: `nums = [3,2,3,1,2,4,5,5,6]`, `k = 4`
Output: `4`

I saw this question on LeetCode.

```
nums = [3,2,3,1,2,4,5,5,6]
k = 4
```

Given an integer array `nums` and an integer `k`, write a function to return the `k` most frequent elements.

You may return the answer in any order.

Example 1:

Input: `nums = [1,1,1,2,2,3]`, `k = 2`
Output: `[1,2]`

Example 2:

Input: `nums = [1]`, `k = 1`
Output: `[1]`

I saw this question on LeetCode.

```
nums = [1,1,1,2,2,3]
k = 2
```

Test whether the given strings are palindromes.

Input: `["aba", "no"]`
Output: `[True, False]`

```
tickers = ["AAPL", "GOOG", "XOX", "XOM"]
```

Write a function `calc_returns()` that accepts lists of prices and dividends and returns a list of returns.

```
prices = [100, 150, 100, 50, 100, 150, 100, 150]
dividends = [1, 1, 1, 1, 2, 2, 2, 2]
```

Rewrite the function `calc_returns()` as `calc_returns_2()` so it returns lists of returns, capital gains yields, and dividend yields.

Write a function `rescale()` to rescale and shift numbers so that they cover the range `[0, 1]`.

Input: `[18.5, 17.0, 18.0, 19.0, 18.0]`
Output: `[0.75, 0.0, 0.5, 1.0, 0.5]`

```python
nums = [18.5, 17.0, 18.0, 19.0, 18.0]
```

Write a function `calc_portval()` that accepts a dictionary of prices and share holdings and returns the portfolio value

```python
data = {
    "AAPL": (150.25, 10),  # (price, shares)
    "GOOGL": (2750.00, 2),
    "MSFT": (300.75, 5)
}
```

McKinney Chapter 3 - Practice - Sec 02

The `%precision` magic makes it easy to round all float on print to 4 decimal places.

```
%precision 4
```

```
'%.4f'
```

Announcements

1. Keep forming groups on Canvas under *People* in the left sidebar
2. You must ask me for groups larger than four students

Five-Minute Review

List

A list is an ordered collection of objects that is changeable (mutable). You can create an empty list using either `[]` or `list()`.

```
my_list = [1, 2, 3, [1, 2, 3, [1, 2, 3]]]
my_list
```

```
[1, 2, 3, [1, 2, 3, [1, 2, 3]]]
```

Python is zero-indexed!

```
my_list[0]
```

```
1
```

```python
my_list[:3] # to get first 3 objects, :3
```

```
[1, 2, 3]
```

```python
my_list[1:4] # to get next 3 objects from 1, go from 1 to 1+3 or 1:3
```

```
[2, 3, [1, 2, 3, [1, 2, 3]]]
```

Tuple

A tuple is similar to a list but un-changeable (immutable). You can create a tuple using parentheses `()` or the `tuple()` function.

```python
my_tuple = (1, 2, 3, (1, 2, 3, (1, 2, 3)))
my_tuple
```

```
(1, 2, 3, (1, 2, 3, (1, 2, 3)))
```

Python is zero-indexed!

```python
my_tuple[0]
```

```
1
```

Tuples are immutable, so they cannot be changed!

```python
# my_tuple[0] = 2_001
# ---------------------------------------------------------------------------
# TypeError                                 Traceback (most recent call last)
# Cell In[10], line 1
# ----> 1 my_tuple[0] = 2_001

# TypeError: 'tuple' object does not support item assignment
```

Dictionary

A dictionary is an ordered collection of key-value pairs that are changeable (mutable). You can create an empty dictionary using either `{}` or the `dict()` function.

```
my_dict = {'wb': 'Warren Buffett', 'sk': 'Seth Klarman'}
my_dict
```

```
{'wb': 'Warren Buffett', 'sk': 'Seth Klarman'}
```

- The *key* can be anything hashable (string, integer, tuple), but I (almost) always make the key a string
- The *value* can be any python object

```
my_dict['wb']
```

```
'Warren Buffett'
```

```
my_dict['pl'] = 'Peter Lynch'
my_dict
```

```
{'wb': 'Warren Buffett', 'sk': 'Seth Klarman', 'pl': 'Peter Lynch'}
```

List Comprehension

A list comprehension is a concise way of creating a new list by iterating over an existing list or other iterable object. It is more time and space-efficient than traditional for loops and offers a cleaner syntax. The basic syntax of a list comprehension is `new_list = [expression for item in iterable if condition]` where:

1. `expression` is the operation to be performed on each element of the iterable
2. `item` is the current element being processed
3. `iterable` is the list or other iterable object being iterated over
4. `condition` is an optional filter that only accepts items that evaluate to True.

For example, we can use the following list comprehension to create a new list of even numbers from 0 to 8: `even_numbers = [x for x in range(9) if x % 2 == 0]`

List comprehensions are a powerful tool in Python that can help you write more efficient and readable code (i.e., more Pythonic code).

What if we wanted multiples of 3 or 5 from 1 to 25?

```
threes_fives = [i for i in range(1, 26) if (i%3==0) | (i%5==0)]
```

```
threes_fives
```

```
[3, 5, 6, 9, 10, 12, 15, 18, 20, 21, 24, 25]
```

```
threes_fives_2 = [print(i) for i in range(1, 26) if (i%3==0) | (i%5==0)]
```

```
3
5
6
9
10
12
15
18
20
21
24
25
```

```
threes_fives_2
```

```
[None, None, None, None, None, None, None, None, None, None, None, None]
```

Practice

Swap the values assigned to `a` and `b` using a third variable `c`.

```
a = 1
```

```
b = 2
```

```
c = a
```

```
a = b
```

```
b = c
```

```
print(f'a is {a} and b is {b}')
```

```
a is 2 and b is 1
```

More on f-strings!

F-strings offer a concise way to embed expressions inside string literals, using curly braces `{}`. Prefixed with `f` or `F`, these strings allow for easy formatting of variables, numbers, and expressions. For example:

```
name = "Alice"
print(f"Hello, {name}!")
```

This outputs "Hello, Alice!". F-strings simplify complex formatting, making code more readable. For a deeper understanding and more examples: https://realpython.com/python-f-strings/

Swap the values assigned to `a` and `b` *without* using a third variable `c`.

```
a = 1
b = 2
b, a = a, b
print(f'a is {a} and b is {b}')
```

```
a is 2 and b is 1
```

```
a = 1
b = 2
a, b = b, a
print(f'a is {a} and b is {b}')
```

```
a is 2 and b is 1
```

What is the output of the following code and why?

```
1, 1, 1 == (1, 1, 1)
```

```
(1, 1, False)
```

Without parentheses `()`, Python reads the final element in the tuple as `1 == (1, 1, 1)`, which is `False`. We can use parentheses `()` to force Python to do what we want!

```
(1, 1, 1) == (1, 1, 1)
```

```
True
```

For this example, we must use parentheses `()` to be unambiguous!

Create a list `l1` of integers from 1 to 100.

```
l1 = list(range(1, 101))
```

```
l1[:5]
```

```
[1, 2, 3, 4, 5]
```

```
l1[-5:]
```

```
[96, 97, 98, 99, 100]
```

Slice `l1` to create a list `l2` of integers from 60 to 50 (inclusive).

```
l2 = l1[59:48:-1]
l2
```

```
[60, 59, 58, 57, 56, 55, 54, 53, 52, 51, 50]
```

```
l1[48:59]
```

```
[49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59]
```

```
l2_alt_1 = l1[49:60][::-1]
l2_alt_1
```

```
[60, 59, 58, 57, 56, 55, 54, 53, 52, 51, 50]
```

```
l2_alt_2 = list(reversed(l1[49:60]))
l2_alt_2
```

```
[60, 59, 58, 57, 56, 55, 54, 53, 52, 51, 50]
```

Create a list `l3` of odd integers from 1 to 21.

```
l3 = list(range(1, 22, 2))
l3
```

```
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21]
```

Python `!=` is the same as Excel's `<>`.

```
l3_alt = [i for i in range(22) if i%2 != 0]
l3_alt
```

```
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21]
```

```
l3_alt_do_not_do_this = []
for i in range(22):
    if i%2 != 0:
        l3_alt_do_not_do_this.append(i)

l3_alt_do_not_do_this
```

```
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21]
```

```
l3_alt_do_not_do_this
```

```
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21]
```

Create a list `l4` of the squares of integers from 1 to 100.

```
l4 = [i**2 for i in range(1, 101)]
l4[:5]
```

```
[1, 4, 9, 16, 25]
```

Create a list `l5` that contains the squares of *odd* integers from 1 to 100.

```
l5 = [i**2 for i in range(1, 101) if i%2!=0]
l5[:5]
```

```
[1, 9, 25, 49, 81]
```

```
l5_alt = [i**2 for i in range(1, 101, 2)]
l5_alt[:5]
```

```
[1, 9, 25, 49, 81]
```

```
l5 == l5_alt
```

```
True
```

Which one is faster?!

```
%timeit [i**2 for i in range(1, 101) if i%2!=0]
```

```
12.3  s ± 2.96  s per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
```

```
%timeit [i**2 for i in range(1, 101, 2)]
```

```
4.74  s ± 956 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
```

> Premature optimization is the root of all evil
>
> `- Donald Knuth`

Use a lambda function to sort `strings` by the last letter in each string.

```
strings = ['Pillsbury', 'Shubrick', 'Clemson', 'Hinson']
```

```
strings.sort()
strings
```

```
['Clemson', 'Hinson', 'Pillsbury', 'Shubrick']
```

```
'Clemson'[-1]
```

```
'n'
```

```
strings.sort(key=lambda x: x[-1])
strings
```

```
['Shubrick', 'Clemson', 'Hinson', 'Pillsbury']
```

```
strings.sort(key=len)
strings
```

```
['Hinson', 'Clemson', 'Shubrick', 'Pillsbury']
```

Given an integer array `nums` and an integer `k`, write a function to return the k^{th} largest element in the array.

Note that it is the k^{th} largest element in the sorted order, not the k^{th} distinct element.

Example 1:

Input: `nums = [3,2,1,5,6,4]`, `k = 2`
Output: `5`

Example 2:

Input: `nums = [3,2,3,1,2,4,5,5,6]`, `k = 4`
Output: `4`

I saw this question on LeetCode.

```python
def get_klarge(nums=[3,2,3,1,2,4,5,5,6], k=4):
    return sorted(nums)[-k]
```

```python
get_klarge(nums=[3,2,1,5,6,4], k=2)
```

```
5
```

The following code follows a bad practice and uses variables in the global scope instead of passing them as arguments or parameters of the functions.

```python
nums = [3,2,3,1,2,4,5,5,6]
k = 4
```

```python
def get_klarge_donotdo():
    return sorted(nums)[-k]
```

```python
get_klarge_donotdo()
```

```
4
```

Here is an extreme example how this practice can lead to non-deterministic and confusing results that depend on how many times a function has been run.

```python
x = [1, 2, 3, 4]
```

```python
def confusing():
     x.append(1)
```

```python
confusing()
confusing()
confusing()
confusing()
confusing()
```

```python
x
```

```
[1, 2, 3, 4, 1, 1, 1, 1, 1]
```

Given an integer array `nums` and an integer `k`, write a function to return the `k` most frequent elements.

You may return the answer in any order.

Example 1:

Input: `nums = [1,1,1,2,2,3]`, `k = 2`
Output: `[1,2]`

Example 2:

Input: `nums = [1]`, `k = 1`
Output: `[1]`

I saw this question on LeetCode.

```python
def get_kfreq(nums, k):
    counts = {}
    for n in nums:
        if n in counts:
            counts[n] += 1
        else:
            counts[n] = 1
    return [x[0] for x in sorted(counts.items(), key=lambda x: x[1], reverse=True)[:k]]
```

```python
get_kfreq(nums=[1,1,1,2,2,3], k=2)
```

```
[1, 2]
```

Test whether the given strings are palindromes.

Input: `["aba", "no"]`
Output: `[True, False]`

We can reverse a string with a `[::-1]` slice just like we reverse a string!

```
def is_palindrome(x):
    return [_x == _x[::-1] for _x in x]
```

```
is_palindrome(["aba", "no"])
```

```
[True, False]
```

```
tickers = ["AAPL", "GOOG", "XOX", "XOM"]
```

```
is_palindrome(tickers)
```

```
[False, True, True, False]
```

Write a function `calc_returns()` that accepts lists of prices and dividends and returns a list of returns.

```
prices = [100, 150, 100, 50, 100, 150, 100, 150]
dividends = [1, 1, 1, 1, 2, 2, 2, 2]
```

Although loop counters are un-Pythonic, this calculation is the rare case where I found loop counters more clear.

```
def calc_returns(p, d):
    r = []
    for i in range(1, len(p)):
        # uncomment this line to watch loop iterations
        # print(f'r is {r}, p[i] is {p[i]}, p[i-1] is {p[i-1]}, d[i] is {d[i]}')
        r.append((p[i] - p[i-1] + d[i]) / p[i-1])
    return r
```

```
calc_returns(p=prices, d=dividends)
```

```
[0.5100, -0.3267, -0.4900, 1.0400, 0.5200, -0.3200, 0.5200]
```

We do not have to specify the argument names `p=` and `d=`, but they help me avoid errors.

```
calc_returns(prices, dividends)
```

```
[0.5100, -0.3267, -0.4900, 1.0400, 0.5200, -0.3200, 0.5200]
```

We can do the same calculation without indexing! Instead, we can use `zip()` to simultaneously loop over prices, lagged prices, and dividends.

```
def calc_returns_zip(p, d):
    r = []
    for _p, _plag, _d in zip(p[1:], p[:-1], d[1:]):
        r.append((_p - _plag + _d) / _plag)
    return r
```

```
calc_returns_zip(p=prices, d=dividends)
```

```
[0.5100, -0.3267, -0.4900, 1.0400, 0.5200, -0.3200, 0.5200]
```

```
calc_returns_zip(p=prices, d=dividends) == calc_returns_zip(p=prices, d=dividends)
```

```
True
```

Rewrite the function `calc_returns()` as `calc_returns_2()` so it returns lists of returns, capital gains yields, and dividend yields.

```
def calc_returns_2(p, d):
    r, cg, dp = [], [], []
    for i in range(1, len(p)):
        r.append((p[i] + d[i] - p[i-1]) / p[i-1])
        cg.append((p[i] - p[i-1]) / p[i-1])
        dp.append(d[i] / p[i-1])

    return {'r':r, 'cg':cg, 'dp': dp}
```

```python
calc_returns_2(p=prices, d=dividends)
```

```
{'r': [0.5100, -0.3267, -0.4900, 1.0400, 0.5200, -0.3200, 0.5200],
 'cg': [0.5000, -0.3333, -0.5000, 1.0000, 0.5000, -0.3333, 0.5000],
 'dp': [0.0100, 0.0067, 0.0100, 0.0400, 0.0200, 0.0133, 0.0200]}
```

```python
calc_returns(p=prices, d=dividends) == calc_returns_2(p=prices, d=dividends)['r']
```

```
True
```

Write a function `rescale()` to rescale and shift numbers so that they cover the range `[0, 1]`.

Input: `[18.5, 17.0, 18.0, 19.0, 18.0]`
Output: `[0.75, 0.0, 0.5, 1.0, 0.5]`

```python
nums = [18.5, 17.0, 18.0, 19.0, 18.0]
```

```python
def rescale(x):
    x_min = min(x)
    x_max = max(x)
    return [(i - x_min) / (x_max - x_min) for i in x]
```

```python
rescale(nums)
```

```
[0.7500, 0.0000, 0.5000, 1.0000, 0.5000]
```

Write a function `calc_portval()` that accepts a dictionary of prices and share holdings and returns the portfolio value

```python
data = {
    "AAPL": (150.25, 10),  # (price, shares)
    "GOOGL": (2750.00, 2),
    "MSFT": (300.75, 5)
}
```

```
def calc_portval(data):
    portval = 0
    for p, n in data.values():
        portval += p * n
    return portval
```

```
calc_portval(data=data)
```

```
8506.2500
```

McKinney Chapter 3 - Practice - Sec 03

The `%precision` magic makes it easy to round all float on print to 4 decimal places.

```
%precision 4
```

```
'%.4f'
```

Announcements

1. Keep forming groups on Canvas under *People* in the left sidebar
2. You must ask me for groups larger than four students

Five-Minute Review

List

A list is an ordered collection of objects that is changeable (mutable). You can create an empty list using either `[]` or `list()`.

```
my_list = [1, 2, 3, [1, 2, 3, [1, 2, 3]]]
my_list
```

```
[1, 2, 3, [1, 2, 3, [1, 2, 3]]]
```

Python is zero-indexed!

```
my_list[0]
```

```
1
```

```
my_list[:3] # to get first 3 objects, :3
```

```
[1, 2, 3]
```

```
my_list[1:4] # to get next 3 objects from 1, go from 1 to 1+3 or 1:3
```

```
[2, 3, [1, 2, 3, [1, 2, 3]]]
```

Tuple

A tuple is similar to a list but un-changeable (immutable). You can create a tuple using parentheses `()` or the `tuple()` function.

```
my_tuple = (1, 2, 3, (1, 2, 3, (1, 2, 3)))
my_tuple
```

```
(1, 2, 3, (1, 2, 3, (1, 2, 3)))
```

Python is zero-indexed!

```
my_tuple[0]
```

```
1
```

Tuples are immutable, so they cannot be changed!

```
# my_tuple[0] = 2_001
# # ---------------------------------------------------------------------------
# # TypeError                                 Traceback (most recent call last)
# # Cell In[10], line 1
# # ----> 1 my_tuple[0] = 2_001

# # TypeError: 'tuple' object does not support item assignment
```

Dictionary

A dictionary is an ordered collection of key-value pairs that are changeable (mutable). You can create an empty dictionary using either `{}` or the `dict()` function.

```python
my_dict = {'wb': 'Warren Buffett', 'sk': 'Seth Klarman'}
my_dict
```

```
{'wb': 'Warren Buffett', 'sk': 'Seth Klarman'}
```

- The *key* can be anything hashable (string, integer, tuple), but I (almost) always make the key a string
- The *value* can be any python object

```python
my_dict['wb']
```

```
'Warren Buffett'
```

```python
my_dict['pl'] = 'Peter Lynch'
my_dict
```

```
{'wb': 'Warren Buffett', 'sk': 'Seth Klarman', 'pl': 'Peter Lynch'}
```

```python
my_dict[(0, 1, 2)] = 'Risky! Do not do live demos!'
my_dict
```

```
{'wb': 'Warren Buffett',
 'sk': 'Seth Klarman',
 'pl': 'Peter Lynch',
 (0, 1, 2): 'Risky! Do not do live demos!'}
```

List Comprehension

A list comprehension is a concise way of creating a new list by iterating over an existing list or other iterable object. It is more time and space-efficient than traditional for loops and offers a cleaner syntax. The basic syntax of a list comprehension is `new_list = [expression for item in iterable if condition]` where:

1. `expression` is the operation to be performed on each element of the iterable
2. `item` is the current element being processed
3. `iterable` is the list or other iterable object being iterated over
4. `condition` is an optional filter that only accepts items that evaluate to True.

For example, we can use the following list comprehension to create a new list of even numbers from 0 to 8: `even_numbers = [x for x in range(9) if x % 2 == 0]`

List comprehensions are a powerful tool in Python that can help you write more efficient and readable code (i.e., more Pythonic code).

What if we wanted multiples of 3 or 5 from 1 to 25?

```
threes_fives = [i for i in range(1, 26) if (i%3==0) | (i%5==0)]
```

```
threes_fives
```

```
[3, 5, 6, 9, 10, 12, 15, 18, 20, 21, 24, 25]
```

```
threes_fives_2 = [print(i) for i in range(1, 26) if (i%3==0) | (i%5==0)]
```

```
3
5
6
9
10
12
15
18
20
21
24
25
```

```
threes_fives_2
```

```
[None, None, None, None, None, None, None, None, None, None, None, None]
```

Practice

Swap the values assigned to `a` and `b` using a third variable `c`.

```
a = 1
b = 2
c = [a, b]
```

```
a = c[1]
```

```
b = c[0]
```

```
print(f'a is {a} and b is {b}')
```

```
a is 2 and b is 1
```

Here is another way:

```
a = 1
b = 2
c = a
a = b
b = c
```

```
print(f'a is {a} and b is {b}')
```

```
a is 2 and b is 1
```

More on f-strings!

F-strings offer a concise way to embed expressions inside string literals, using curly braces `{}`. Prefixed with `f` or `F`, these strings allow for easy formatting of variables, numbers, and expressions. For example:

```
name = "Alice"
print(f"Hello, {name}!")
```

This outputs "Hello, Alice!". F-strings simplify complex formatting, making code more readable. For a deeper understanding and more examples: https://realpython.com/python-f-strings/

Swap the values assigned to `a` and `b` *without* using a third variable `c`.

```
a = 1
b = 2

b, a = a, b

print(f'a is {a} and b is {b}')
```

```
a is 2 and b is 1
```

What is the output of the following code and why?

```
1, 1, 1 == (1, 1, 1)
```

```
(1, 1, False)
```

Without parentheses `()`, Python reads the final element in the tuple as `1 == (1, 1, 1)`, which is `False`. We can use parentheses `()` to force Python to do what we want!

```
(1, 1, 1) == (1, 1, 1)
```

```
True
```

For this example, we must use parentheses `()` to be unambiguous!

Create a list `l1` of integers from 1 to 100.

```
l1 = list(range(1, 101))
```

```
l1[:5]
```

```
[1, 2, 3, 4, 5]
```

```python
l1[-5:]
```

```
[96, 97, 98, 99, 100]
```

Slice `l1` to create a list `l2` of integers from 60 to 50 (inclusive).

```python
l1.index(60)
```

```
59
```

```python
l2 = l1[59:48:-1]
```

```python
l2_alt_1 = l1[49:60]
l2_alt_1.reverse() # most list methods modify a list "in place"
l2_alt_1
```

```
[60, 59, 58, 57, 56, 55, 54, 53, 52, 51, 50]
```

```python
l2_alt_2 = l1[49:60][::-1]
l2_alt_2
```

```
[60, 59, 58, 57, 56, 55, 54, 53, 52, 51, 50]
```

Create a list `l3` of odd integers from 1 to 21.

```python
l3 = list(range(1, 22, 2))
l3
```

```
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21]
```

```python
l3_alt_1 = []
for i in range(1, 22):
    if i%2 != 0:
        l3_alt_1.append(i)

l3_alt_1
```

```
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21]
```

```
l3_alt_2 = [i for i in range(1, 22) if i%2!=0]
l3_alt_2
```

```
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21]
```

Create a list `l4` of the squares of integers from 1 to 100.

```
l4 = [i**2 for i in range(1, 101)]
l4[:5]
```

```
[1, 4, 9, 16, 25]
```

Create a list `l5` that contains the squares of *odd* integers from 1 to 100.

```
l5 = [i**2 for i in range(1, 101) if i%2!=0]
l5[:5]
```

```
[1, 9, 25, 49, 81]
```

```
l5_alt = [i**2 for i in range(1, 101, 2)]
l5_alt[:5]
```

```
[1, 9, 25, 49, 81]
```

```
l5 == l5_alt
```

```
True
```

Which one is faster?!

```
%timeit [i**2 for i in range(1, 101) if i%2!=0]
```

```
12.2  s ± 4.94  s per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
```

```
%timeit [i**2 for i in range(1, 101, 2)]
```

```
5.96  s ± 998 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
```

> Premature optimization is the root of all evil
>
> \- Donald Knuth

Use a lambda function to sort `strings` by the last letter in each string.

```
strings = ['Pillsbury', 'Shubrick', 'Clemson', 'Hinson']
```

```
strings.sort()
strings
```

```
['Clemson', 'Hinson', 'Pillsbury', 'Shubrick']
```

```
len('Pillsbury')
```

```
9
```

```
strings.sort(key=len)
strings
```

```
['Hinson', 'Clemson', 'Shubrick', 'Pillsbury']
```

```
'Pillsbury'[-1]
```

```
'y'
```

```
strings.sort(key=lambda x: x[-1])
strings
```

```
['Shubrick', 'Hinson', 'Clemson', 'Pillsbury']
```

Given an integer array `nums` and an integer `k`, write a function to return the k^{th} largest element in the array.

Note that it is the k^{th} largest element in the sorted order, not the k^{th} distinct element.

Example 1:

Input: `nums = [3,2,1,5,6,4]`, `k = 2`
Output: `5`

Example 2:

Input: `nums = [3,2,3,1,2,4,5,5,6]`, `k = 4`
Output: `4`

I saw this question on LeetCode.

```python
def get_klarge(nums=[3,2,3,1,2,4,5,5,6], k=4):
    return sorted(nums)[-k]
```

```python
get_klarge(nums=[3,2,1,5,6,4], k=2)
```

```
5
```

The following code follows a bad practice and uses variables in the global scope instead of passing them as arguments or parameters of the functions.

```python
nums = [3,2,3,1,2,4,5,5,6]
k = 4
```

```python
def get_klarge_donotdo():
    return sorted(nums)[-k]
```

```python
get_klarge_donotdo()
```

```
4
```

Here is an extreme example how this practice can lead to non-deterministic and confusing results that depend on how many times a function has been run.

```python
x = [1, 2, 3, 4]
```

```python
def confusing():
     x.append(1)
```

```python
confusing()
confusing()
confusing()
confusing()
confusing()
```

```python
x
```

```
[1, 2, 3, 4, 1, 1, 1, 1, 1]
```

Given an integer array `nums` and an integer `k`, write a function to return the `k` most frequent elements.

You may return the answer in any order.

Example 1:

Input: `nums = [1,1,1,2,2,3]`, `k = 2`
Output: `[1,2]`

Example 2:

Input: `nums = [1]`, `k = 1`
Output: `[1]`

I saw this question on LeetCode.

```python
def get_kfreq(nums, k):
    counts = {}
    for n in nums:
        if n in counts:
            counts[n] += 1
        else:
            counts[n] = 1
    return [x[0] for x in sorted(counts.items(), key=lambda x: x[1], reverse=True)[:k]]
```

```python
get_kfreq(nums=[1,1,1,2,2,3], k=2)
```

```
[1, 2]
```

Test whether the given strings are palindromes.

Input: `["aba", "no"]`
Output: `[True, False]`

```
def is_palindrome(x):
    return [_x == _x[::-1] for _x in x]
```

```
is_palindrome(["aba", "no"])
```

```
[True, False]
```

```
tickers = ["AAPL", "GOOG", "XOX", "XOM"]
```

```
is_palindrome(tickers)
```

```
[False, True, True, False]
```

Write a function `calc_returns()` that accepts lists of prices and dividends and returns a list of returns.

```
prices = [100, 150, 100, 50, 100, 150, 100, 150]
dividends = [1, 1, 1, 1, 2, 2, 2, 2]
```

Although loop counters are un-Pythonic, this calculation is the rare case where I found loop counters more clear.

```
def calc_returns(p, d):
    r = []
    for i in range(1, len(p)):
        # the following print statement print r, p, and d values each iteration
        # print(f'r is {r}, p[i] is {p[i]}, p[i-1] is {p[i-1]}, d[i] is {d[i]},')
        r.append((p[i] + d[i] - p[i-1]) / p[i-1])
    return r
```

```
calc_returns(p=prices, d=dividends)
```

```
[0.5100, -0.3267, -0.4900, 1.0400, 0.5200, -0.3200, 0.5200]
```

We can do the same calculation without indexing! Instead, we can use `zip()` to simultaneously loop over prices, lagged prices, and dividends.

```
def calc_returns_zip(p, d):
    r = []
    for _p, _plag, _d in zip(p[1:], p[:-1], d[1:]):
        # the following print statement print r, p, and d values each iteration
        # print(f'r is {r}, _pis {_p}, _plag is {_plag}, _d is {_d},')
        r.append((_p + _d - _plag) / _plag)
    return r
```

```
calc_returns_zip(p=prices, d=dividends)
```

```
[0.5100, -0.3267, -0.4900, 1.0400, 0.5200, -0.3200, 0.5200]
```

```
calc_returns_zip(p=prices, d=dividends) == calc_returns_zip(p=prices, d=dividends)
```

```
True
```

Rewrite the function `calc_returns()` as `calc_returns_2()` so it returns lists of returns, capital gains yields, and dividend yields.

```
def calc_returns_2(p, d):
    r = []
    cg = []
    dp = []
    # r, cg, dp = [], [], [] # tuple unpacking is very Pythonic!
    for i in range(1, len(p)):
        r.append((p[i] + d[i] - p[i-1]) / p[i-1])
        cg.append((p[i] - p[i-1]) / p[i-1])
        dp.append(d[i] / p[i-1])

    return {'r':r, 'cg':cg, 'dp':dp}
```

```
calc_returns_2(p=prices, d=dividends)
```

```
{'r': [0.5100, -0.3267, -0.4900, 1.0400, 0.5200, -0.3200, 0.5200],
 'cg': [0.5000, -0.3333, -0.5000, 1.0000, 0.5000, -0.3333, 0.5000],
 'dp': [0.0100, 0.0067, 0.0100, 0.0400, 0.0200, 0.0133, 0.0200]}
```

```python
calc_returns(p=prices, d=dividends) == calc_returns_2(p=prices, d=dividends)['r']
```

```
True
```

Write a function `rescale()` to rescale and shift numbers so that they cover the range `[0, 1]`.

Input: `[18.5, 17.0, 18.0, 19.0, 18.0]`
Output: `[0.75, 0.0, 0.5, 1.0, 0.5]`

```python
nums = [18.5, 17.0, 18.0, 19.0, 18.0]
```

```python
def rescale(x):
    x_min = min(x)
    x_max = max(x)
    return [(i - x_min) / (x_max - x_min) for i in x]
```

```python
rescale(nums)
```

```
[0.7500, 0.0000, 0.5000, 1.0000, 0.5000]
```

Write a function `calc_portval()` that accepts a dictionary of prices and share holdings and returns the portfolio value

```python
data = {
    "AAPL": (150.25, 10),  # (price, shares)
    "GOOGL": (2750.00, 2),
    "MSFT": (300.75, 5)
}
```

```python
def calc_portval(data):
    portval = 0
    for p, n in data.values():
        portval += p * n
    return portval
```

```
calc_portval(data=data)
```

```
8506.2500
```

McKinney Chapter 3 - Practice - Blank

The `%precision` magic makes it easy to round all float on print to 4 decimal places.

```
%precision 4
```

```
'%.4f'
```

Announcements

1. Keep forming groups on Canvas under *People* in the left sidebar
2. You must ask me for groups larger than four students

Five-Minute Review

List

A list is an ordered collection of objects that is changeable (mutable). You can create an empty list using either `[]` or `list()`.

```
my_list = [1, 2, 3, [1, 2, 3, [1, 2, 3]]]
my_list
```

```
[1, 2, 3, [1, 2, 3, [1, 2, 3]]]
```

Python is zero-indexed!

```
my_list[0]
```

```
1
```

```
my_list[:3] # to get first 3 objects, :3
```

```
[1, 2, 3]
```

```
my_list[1:4] # to get next 3 objects from 1, go from 1 to 1+3 or 1:3
```

```
[2, 3, [1, 2, 3, [1, 2, 3]]]
```

Tuple

A tuple is similar to a list but un-changeable (immutable). You can create a tuple using parentheses `()` or the `tuple()` function.

```
my_tuple = (1, 2, 3, (1, 2, 3, (1, 2, 3)))
my_tuple
```

```
(1, 2, 3, (1, 2, 3, (1, 2, 3)))
```

Python is zero-indexed!

```
my_tuple[0]
```

```
1
```

Tuples are immutable, so they cannot be changed!

```
# my_tuple[0] = 2_001
# # ---------------------------------------------------------------------------
# # TypeError                                 Traceback (most recent call last)
# # Cell In[10], line 1
# # ----> 1 my_tuple[0] = 2_001

# # TypeError: 'tuple' object does not support item assignment
```

Dictionary

A dictionary is an ordered collection of key-value pairs that are changeable (mutable). You can create an empty dictionary using either `{}` or the `dict()` function.

```
my_dict = {'wb': 'Warren Buffett', 'sk': 'Seth Klarman'}
my_dict
```

```
{'wb': 'Warren Buffett', 'sk': 'Seth Klarman'}
```

- The *key* can be anything hashable (string, integer, tuple), but I (almost) always make the key a string
- The *value* can be any python object

```
my_dict['wb']
```

```
'Warren Buffett'
```

```
my_dict['pl'] = 'Peter Lynch'
my_dict
```

```
{'wb': 'Warren Buffett', 'sk': 'Seth Klarman', 'pl': 'Peter Lynch'}
```

```
my_dict[(0, 1, 2)] = {'ad': 'Another dictionary!'}
my_dict
```

```
{'wb': 'Warren Buffett',
 'sk': 'Seth Klarman',
 'pl': 'Peter Lynch',
 (0, 1, 2): {'ad': 'Another dictionary!'}}
```

List Comprehension

A list comprehension is a concise way of creating a new list by iterating over an existing list or other iterable object. It is more time and space-efficient than traditional for loops and offers a cleaner syntax. The basic syntax of a list comprehension is `new_list = [expression for item in iterable if condition]` where:

1. `expression` is the operation to be performed on each element of the iterable
2. `item` is the current element being processed
3. `iterable` is the list or other iterable object being iterated over
4. `condition` is an optional filter that only accepts items that evaluate to True.

For example, we can use the following list comprehension to create a new list of even numbers from 0 to 8: `even_numbers = [x for x in range(9) if x % 2 == 0]`

List comprehensions are a powerful tool in Python that can help you write more efficient and readable code (i.e., more Pythonic code).

What if we wanted multiples of 3 or 5 from 1 to 25?

```
threes_fives = [i for i in range(1, 26) if (i%3==0) | (i%5==0)]
threes_fives
```

```
[3, 5, 6, 9, 10, 12, 15, 18, 20, 21, 24, 25]
```

```
threes_fives_2 = [print(i) for i in range(1, 26) if (i%3==0) | (i%5==0)]
threes_fives_2
```

```
3
5
6
9
10
12
15
18
20
21
24
25
```

```
[None, None, None, None, None, None, None, None, None, None, None, None]
```

Practice

Swap the values assigned to `a` and `b` using a third variable `c`.

```
a = 1
b = 2
```

```
c = a
a = b
b = c
```

```
print(f'a is {a} and b is {b}')
```

```
a is 2 and b is 1
```

More on f-strings!

F-strings offer a concise way to embed expressions inside string literals, using curly braces `{}`. Prefixed with `f` or `F`, these strings allow for easy formatting of variables, numbers, and expressions. For example:

```
name = "Alice"
print(f"Hello, {name}!")
```

This outputs "Hello, Alice!". F-strings simplify complex formatting, making code more readable. For a deeper understanding and more examples: https://realpython.com/python-f-strings/

Swap the values assigned to `a` and `b` *without* using a third variable `c`.

```
a = 1
b = 2
```

```
b, a = a, b
```

```
print(f'a is {a} and b is {b}')
```

```
a is 2 and b is 1
```

What is the output of the following code and why?

```
1, 1, 1 == (1, 1, 1)
```

```
(1, 1, False)
```

Without parentheses `()`, Python reads the final element in the tuple as `1 == (1, 1, 1)`, which is `False`. We can use parentheses `()` to force Python to do what we want!

```
(1, 1, 1) == (1, 1, 1)
```

```
True
```

For this example, we must use parentheses `()` to be unambiguous!

Create a list `l1` of integers from 1 to 100.

```
l1 = list(range(1, 101))
l1[:5]
```

```
[1, 2, 3, 4, 5]
```

Slice `l1` to create a list `l2` of integers from 60 to 50 (inclusive).

```
l1.index(60)
```

```
59
```

```
l2 = l1[59:48:-1]
l2
```

```
[60, 59, 58, 57, 56, 55, 54, 53, 52, 51, 50]
```

```
l2_alt_1 = l1[49:60]
l2_alt_1.reverse()
l2_alt_1
```

```
[60, 59, 58, 57, 56, 55, 54, 53, 52, 51, 50]
```

```
l2_alt_2 = l1[49:60]
l2_alt_2.sort(reverse=True)
l2_alt_2
```

```
[60, 59, 58, 57, 56, 55, 54, 53, 52, 51, 50]
```

```
l1[49:60][::-1]
```

```
[60, 59, 58, 57, 56, 55, 54, 53, 52, 51, 50]
```

Create a list `l3` of odd integers from 1 to 21.

```
l3 = list(range(1, 22, 2))
l3
```

```
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21]
```

```
l3_alt_1 = []
for i in range(1, 22):
    if i%2 != 0:
        l3_alt_1.append(i)

l3_alt_1
```

```
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21]
```

```
l3_alt_2 = [i for i in range(1, 22) if i%2!=0]
l3_alt_2
```

```
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21]
```

Create a list `l4` of the squares of integers from 1 to 100.

```
l4 = [i**2 for i in range(1, 101)]
l4[:5]
```

```
[1, 4, 9, 16, 25]
```

Create a list `l5` that contains the squares of *odd* integers from 1 to 100.

```
l5 = [i**2 for i in range(1, 101) if i%2!=0]
l5[:5]
```

```
[1, 9, 25, 49, 81]
```

```
l5_alt = [i**2 for i in range(1, 101, 2)]
l5_alt[:5]
```

```
[1, 9, 25, 49, 81]
```

```
l5 == l5_alt
```

```
True
```

Which one is faster?!

```
%timeit [i**2 for i in range(1, 101) if i%2!=0]
```

```
13.1  s ± 4.81  s per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
```

```
%timeit [i**2 for i in range(1, 101, 2)]
```

```
6.51  s ± 771 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
```

> Premature optimization is the root of all evil
>
> `- Donald Knuth`

Use a lambda function to sort `strings` by the last letter in each string.

```
strings = ['Pillsbury', 'Shubrick', 'Clemson', 'Hinson']
```

```
strings.sort()
strings
```

```
['Clemson', 'Hinson', 'Pillsbury', 'Shubrick']
```

```
len('Pillsbury')
```

```
9
```

```
strings.sort(key=len)
strings
```

```
['Hinson', 'Clemson', 'Shubrick', 'Pillsbury']
```

```
'Pillsbury'[-1]
```

```
'y'
```

```
strings.sort(key=lambda x: x[-1])
strings
```

```
['Shubrick', 'Hinson', 'Clemson', 'Pillsbury']
```

Given an integer array `nums` and an integer `k`, write a function to return the k^{th} largest element in the array.

Note that it is the k^{th} largest element in the sorted order, not the k^{th} distinct element.

Example 1:

Input: `nums = [3,2,1,5,6,4]`, `k = 2`
Output: `5`

Example 2:

Input: `nums = [3,2,3,1,2,4,5,5,6]`, `k = 4`
Output: `4`

I saw this question on LeetCode.

```python
def get_klarge(nums, k):
    return sorted(nums)[-k]
```

```python
get_klarge(nums=[3,2,1,5,6,4], k=2)
```

```
5
```

```python
get_klarge(nums=[3,2,3,1,2,4,5,5,6], k=4)
```

```
4
```

The following code follows a bad practice and uses variables in the global scope instead of passing them as arguments or parameters of the functions.

```python
nums = [3,2,3,1,2,4,5,5,6]
k = 4
```

```python
def get_klarge_donotdo():
    return sorted(nums)[-k]
```

```python
get_klarge_donotdo()
```

```
4
```

Here is an extreme example how this practice can lead to non-deterministic and confusing results that depend on how many times a function has been run.

```python
x = []
```

```python
def confusing():
    x.append(len(x))
```

```python
confusing()
confusing()
confusing()
```

```
x
```

```
[0, 1, 2]
```

Given an integer array `nums` and an integer `k`, write a function to return the `k` most frequent elements.

You may return the answer in any order.

Example 1:

Input: `nums = [1,1,1,2,2,3]`, `k = 2`
Output: `[1,2]`

Example 2:

Input: `nums = [1]`, `k = 1`
Output: `[1]`

I saw this question on LeetCode.

```
def get_kfreq(nums, k):
    counts = {}
    for n in nums:
        if n in counts:
            counts[n] += 1
        else:
            counts[n] = 1
    return [x[0] for x in sorted(counts.items(), key=lambda x: x[1], reverse=True)[:k]]
```

```
get_kfreq(nums=[1,1,1,2,2,3], k=2)
```

```
[1, 2]
```

Test whether the given strings are palindromes.

Input: `["aba", "no"]`
Output: `[True, False]`

```
def is_palindrome(x):
    return [_x == _x[::-1] for _x in x]
```

```
is_palindrome(["aba", "no"])
```

```
[True, False]
```

```
tickers = ["AAPL", "GOOG", "XOX", "XOM"]
```

```
is_palindrome(tickers)
```

```
[False, True, True, False]
```

Write a function `calc_returns()` that accepts lists of prices and dividends and returns a list of returns.

```
prices = [100, 150, 100, 50, 100, 150, 100, 150]
dividends = [1, 1, 1, 1, 2, 2, 2, 2]
```

Although loop counters are un-Pythonic, this calculation is the rare case where I found loop counters more clear.

```
def calc_returns(p, d):
    r =[]
    for i in range(1, len(p)):
        # the following print statement print r, p, and d values each iteration
        # print(f'r is {r}, p[i] is {p[i]}, p[i-1] is {p[i-1]}, d[i] is {d[i]},')
        r.append((p[i] + d[i] - p[i-1]) / p[i-1])
    return r
```

```
calc_returns(p=prices, d=dividends)
```

```
[0.5100, -0.3267, -0.4900, 1.0400, 0.5200, -0.3200, 0.5200]
```

We can do the same calculation without indexing! Instead, we can use `zip()` to simultaneously loop over prices, lagged prices, and dividends.

```
def calc_returns_zip(p, d):
    r = []
    for _p, _plag, _d in zip(p[1:], p[:-1], d[1:]):
        # the following print statement print r, p, and d values each iteration
        # print(f'r is {r}, _pis {_p}, _plag is {_plag}, _d is {_d},')
        r.append((_p + _d - _plag) / _plag)
    return r
```

```
calc_returns_zip(p=prices, d=dividends)
```

```
[0.5100, -0.3267, -0.4900, 1.0400, 0.5200, -0.3200, 0.5200]
```

```
calc_returns_zip(p=prices, d=dividends) == calc_returns_zip(p=prices, d=dividends)
```

```
True
```

Rewrite the function `calc_returns()` as `calc_returns_2()` so it returns lists of returns, capital gains yields, and dividend yields.

```
def calc_returns_2(p, d):
    r =[]
    cg =[]
    dp =[]
    # r, cg, dp = [], [], [] # tuple unpacking is very Pythonic!
    for i in range(1, len(p)):
        r.append((p[i] + d[i] - p[i-1]) / p[i-1])
        cg.append((p[i] - p[i-1]) / p[i-1])
        dp.append(d[i] / p[i-1])

    return {'r':r, 'cg':cg, 'dp':dp}
```

```
calc_returns_2(p=prices, d=dividends)
```

```
{'r': [0.5100, -0.3267, -0.4900, 1.0400, 0.5200, -0.3200, 0.5200],
 'cg': [0.5000, -0.3333, -0.5000, 1.0000, 0.5000, -0.3333, 0.5000],
 'dp': [0.0100, 0.0067, 0.0100, 0.0400, 0.0200, 0.0133, 0.0200]}
```

```
calc_returns(p=prices, d=dividends) == calc_returns_2(p=prices, d=dividends)['r']
```

```
True
```

Write a function `rescale()` to rescale and shift numbers so that they cover the range `[0, 1]`.

Input: `[18.5, 17.0, 18.0, 19.0, 18.0]`
Output: `[0.75, 0.0, 0.5, 1.0, 0.5]`

```
nums = [18.5, 17.0, 18.0, 19.0, 18.0]
```

```
def rescale(x):
    x_min = min(x)
    x_max = max(x)
    return [(i - x_min) / (x_max - x_min) for i in x]
```

```
rescale(nums)
```

```
[0.7500, 0.0000, 0.5000, 1.0000, 0.5000]
```

Write a function `calc_portval()` that accepts a dictionary of prices and share holdings and returns the portfolio value

```
data = {
    "AAPL": (150.25, 10),  # (price, shares)
    "GOOGL": (2750.00, 2),
    "MSFT": (300.75, 5)
}
```

```
def calc_portval(data):
    portval = 0
    for p, n in data.values():
        portval += p * n
    return portval
```

```
calc_portval(data=data)
```

```
8506.2500
```

Week 3

McKinney Chapter 4 - NumPy Basics: Arrays and Vectorized Computation

```
import numpy as np
```

```
%precision 4
```

```
'%.4f'
```

Introduction

Chapter 4 of McKinney (2022) discusses the NumPy package (an abbreviation of numerical Python), which is the foundation for numerical computing in Python, including pandas.

We will focus on:

1. Creating arrays
2. Slicing arrays
3. Applying functions and methods to arrays
4. Using conditional logic with arrays (i.e., `np.where()` and `np.select()`)

Note: Indented block quotes are from McKinney (2022) unless otherwise indicated. The section numbers here differ from McKinney (2022) because we will only discuss some topics.

Here is a simple example of NumPy's speed and syntax advantages relative to Python's built-in data structures. First, we create a list and a NumPy array with values from 0 to 999,999.

```
my_list = list(range(1_000_000))
my_arr = np.arange(1_000_000)
```

```
my_list[:5]
```

```
[0, 1, 2, 3, 4]
```

```
my_arr[:5]
```

```
array([0, 1, 2, 3, 4])
```

We must use a for loop or a list comprehension to double each value in `my_list`. We will comment this code because it prints a list with one million elements!

```
# [2 * x for x in my_list] # list comprehension to double each value
```

However, we can multiply `my_arr` by two because math "just works" with NumPy. Jupyter will pretty print the NumPy array, showing only the first and last few elements.

```
my_arr * 2
```

```
array([      0,       2,       4, ..., 1999994, 1999996, 1999998],
      shape=(1000000,))
```

We can use the "magic" function `%timeit` to time these two calculations.

```
%timeit [x * 2 for x in my_list]
```

```
72.5 ms ± 16.1 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

```
%timeit my_arr * 2
```

```
4.44 ms ± 867  s per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

The NumPy version is much faster than the list version! The NumPy version is also faster to type, read, and troubleshoot, and our time is more valuable than computer time!

The NumPy ndarray: A Multidimensional Array Object

> One of the key features of NumPy is its N-dimensional array object, or ndarray, which is a fast, flexible container for large datasets in Python. Arrays enable you to perform mathematical operations on whole blocks of data using similar syntax to the equivalent operations between scalar elements.

```
np.random.seed(42) # makes random numbers repeatable
data = np.random.randn(2, 3)
data
```

```
array([[ 0.4967, -0.1383,  0.6477],
       [ 1.523 , -0.2342, -0.2341]])
```

Multiplying `data` by 10 multiplies each element in `data` by 10, and adding `data` to itself adds each element to itself (i.e., element-wise addition). NumPy arrays must contain homogeneous data types (e.g., all floats or integers) to achieve this common-sense behavior.

```
data * 10
```

```
array([[ 4.9671, -1.3826,  6.4769],
       [15.2303, -2.3415, -2.3414]])
```

```
data_2 = data + data
data_2
```

```
array([[ 0.9934, -0.2765,  1.2954],
       [ 3.0461, -0.4683, -0.4683]])
```

NumPy arrays have attributes. Recall that IPython and Jupyter provide tab completion.

```
data.ndim
```

```
2
```

```
data.shape
```

```
(2, 3)
```

```
data.dtype
```

```
dtype('float64')
```

We slice NumPy arrays using `[]`, the same as we slice lists and tuples.

```
data[0]
```

```
array([ 0.4967, -0.1383,  0.6477])
```

We chain `[]`s with arrays, the same as we chain `[]`s with lists and tuples.

```
data[0][0]
```

```
0.4967
```

However, with NumPy arrays, we can replace n chained `[]`s with one pair of `[]`s containing n indexes or slices, separated by commas. For example, `[i][j]` becomes `[i, j]`, and `[i][j][k]` becomes `[i, j, k]`.

```
data[0, 0] # zero row, zero column
```

```
0.4967
```

```
data[0][0] == data[0, 0]
```

```
np.True_
```

Creating ndarrays

> The easiest way to create an array is to use the array function. This accepts any sequence-like object (including other arrays) and produces a new NumPy array containing the passed data

```
data1 = [6, 7.5, 8, 0, 1]
arr1 = np.array(data1)
arr1
```

```
array([6. , 7.5, 8. , 0. , 1. ])
```

```
arr1.dtype
```

```
dtype('float64')
```

Here, `np.array()` implicitly casts the integers in `data1` to floats because NumPy arrays must have homogenous data types. We could explicitly cast all values to integers but would lose information.

```
np.array(data1, dtype=np.int64)
```

```
array([6, 7, 8, 0, 1])
```

We can cast a list of lists to a two-dimensional NumPy array.

```
data2 = [[1, 2, 3, 4], [5, 6, 7, 8]]
arr2 = np.array(data2)
arr2
```

```
array([[1, 2, 3, 4],
       [5, 6, 7, 8]])
```

```
arr2.shape
```

```
(2, 4)
```

```
arr2.dtype
```

```
dtype('int64')
```

There are several other ways to create NumPy arrays.

```
np.zeros((3, 6))
```

```
array([[0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0.]])
```

```
np.ones((3, 6))
```

```
array([[1., 1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1., 1.]])
```

```
np.ones_like(arr2)
```

```
array([[1, 1, 1, 1],
       [1, 1, 1, 1]])
```

The `np.arange()` function is similar to Python's built-in `range()` but creates an array directly.

```
np.array(range(15))
```

```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14])
```

```
np.arange(15)
```

```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14])
```

Table 4-1 from McKinney (2022) summarizes NumPy array creation functions.

- `array`: Convert input data (list, tuple, array, or other sequence type) to an ndarray either by inferring a dtype or explicitly specifying a dtype; copies the input data by default
- `asarray`: Convert input to ndarray, but do not copy if the input is already an ndarray
- `arange`: Like the built-in range but returns an ndarray instead of a list
- `ones`, `ones_like`: Produce an array of all 1s with the given shape and dtype; `ones_like` takes another array and produces a `ones` array of the - same shape and dtype
- `zeros`, `zeros_like`: Like `ones` and `ones_like` but producing arrays of 0s instead
- `empty`, `empty_like`: Create new arrays by allocating new memory, but do not populate with any values like ones and zeros
- `full`, `full_like`: Produce an array of the given shape and dtype with all values set to the indicated "fill value"
- `eye`, `identity`: Create a square N-by-N identity matrix (1s on the diagonal and 0s elsewhere)

Arithmetic with NumPy Arrays

> Arrays are important because they enable you to express batch operations on data without writing any for loops. NumPy users call this vectorization. Any arithmetic operations between equal-size arrays applies the operation element-wise

```
arr = np.array([[1., 2., 3.], [4., 5., 6.]])
arr
```

```
array([[1., 2., 3.],
       [4., 5., 6.]])
```

NumPy array addition is elementwise.

```
arr + arr
```

```
array([[ 2.,  4.,  6.],
       [ 8., 10., 12.]])
```

NumPy array multiplication is elementwise.

```
arr * arr
```

```
array([[ 1.,  4.,  9.],
       [16., 25., 36.]])
```

NumPy array division is elementwise.

```
1 / arr
```

```
array([[1.    , 0.5   , 0.3333],
       [0.25  , 0.2   , 0.1667]])
```

NumPy powers are elementwise, too.

```
arr ** 2
```

```
array([[ 1.,  4.,  9.],
       [16., 25., 36.]])
```

We can also raise a single value to the elements in an array!

```
2 ** arr
```

```
array([[ 2.,  4.,  8.],
       [16., 32., 64.]])
```

Basic Indexing and Slicing

We index and slice one-dimensional arrays in the same way as lists and tuples.

```
arr = np.arange(10)
arr
```

```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```
arr[5]
```

```
np.int64(5)
```

```
arr[5:8]
```

```
array([5, 6, 7])
```

```
equiv_list = list(range(10))
equiv_list
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
equiv_list[5:8]
```

```
[5, 6, 7]
```

We must jump through some hoops to replace elements 5, 6, and 7 with the value 12 in the list `equiv_list`.

```
# # TypeError: can only assign an iterable
# equiv_list[5:8] = 12
```

```
equiv_list[5:8] = [12] * 3
equiv_list
```

```
[0, 1, 2, 3, 4, 12, 12, 12, 8, 9]
```

However, this operation is easy with NumPy array `arr`!

```
arr[5:8] = 12
arr
```

```
array([ 0,  1,  2,  3,  4, 12, 12, 12,  8,  9])
```

We call this behavior "broadcasting".

> As you can see, if you assign a scalar value to a slice, as in `arr[5:8] = 12`, the value is propagated (or broadcasted henceforth) to the entire selection. An important first distinction from Python's built-in lists is that array slices are views on the original array. This means that the data is not copied, and any modifications to the view will be reflected in the source array.

```
arr_slice = arr[5:8]
arr_slice
```

```
array([12, 12, 12])
```

```
arr_slice[1] = 12345
arr_slice
```

```
array([   12, 12345,    12])
```

```
arr
```

```
array([    0,     1,     2,     3,     4,    12, 12345,    12,     8,
           9])
```

The `:` slices every element in `arr_slice`.

```
arr_slice[:] = 64
arr_slice
```

```
array([64, 64, 64])
```

```
arr
```

```
array([ 0,  1,  2,  3,  4, 64, 64, 64,  8,  9])
```

> If you want a copy of a slice of an ndarray instead of a view, you will need to explicitly copy the array-for example, `arr[5:8].copy()`.

```
arr_slice_2 = arr[5:8].copy()
arr_slice_2
```

```
array([64, 64, 64])
```

```
arr_slice_2[:] = 2_001
arr_slice_2
```

```
array([2001, 2001, 2001])
```

```
arr
```

```
array([ 0,  1,  2,  3,  4, 64, 64, 64,  8,  9])
```

Indexing with slices

We can slice across two or more dimensions and use the `[i, j]` notation.

```
arr2d = np.array([[1,2,3], [4,5,6], [7,8,9]])
arr2d
```

```
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

```
arr2d[:2]
```

```
array([[1, 2, 3],
       [4, 5, 6]])
```

```
arr2d[:2, 1:]
```

```
array([[2, 3],
       [5, 6]])
```

A colon (`:`) by itself selects the entire dimension and is necessary to slice higher dimensions.

```
arr2d[:, :1]
```

```
array([[1],
       [4],
       [7]])
```

```
arr2d[:2, 1:] = 0
arr2d
```

```
array([[1, 0, 0],
       [4, 0, 0],
       [7, 8, 9]])
```

Always check your output!

Boolean Indexing

We can use Booleans (i.e., `True` and `False`) to slice arrays, too. Boolean indexing in Python is like combining `index()` and `match()` in Excel.

```
names = np.array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'])
np.random.seed(42)
data = np.random.randn(7, 4)
```

```
names
```

```
array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'], dtype='<U4')
```

```
data
```

```
array([[ 0.4967, -0.1383,  0.6477,  1.523 ],
       [-0.2342, -0.2341,  1.5792,  0.7674],
       [-0.4695,  0.5426, -0.4634, -0.4657],
       [ 0.242 , -1.9133, -1.7249, -0.5623],
       [-1.0128,  0.3142, -0.908 , -1.4123],
       [ 1.4656, -0.2258,  0.0675, -1.4247],
       [-0.5444,  0.1109, -1.151 ,  0.3757]])
```

Here `names` provides seven names for the seven rows in `data`.

```
names == 'Bob'
```

```
array([ True, False, False,  True, False, False, False])
```

```
data[names == 'Bob']
```

```
array([[ 0.4967, -0.1383,  0.6477,  1.523 ],
       [ 0.242 , -1.9133, -1.7249, -0.5623]])
```

We can combine Boolean slicing with `:` slicing.

```
data[names == 'Bob', 2:]
```

```
array([[ 0.6477,  1.523 ],
       [-1.7249, -0.5623]])
```

We can use `~` to invert a Boolean.

```
cond = names == 'Bob'
data[~cond]
```

```
array([[-0.2342, -0.2341,  1.5792,  0.7674],
       [-0.4695,  0.5426, -0.4634, -0.4657],
       [-1.0128,  0.3142, -0.908 , -1.4123],
       [ 1.4656, -0.2258,  0.0675, -1.4247],
       [-0.5444,  0.1109, -1.151 ,  0.3757]])
```

For NumPy arrays, we must use `&` and `|` instead of `and` and `or`.

```
cond = (names == 'Bob') | (names == 'Will')
data[cond]
```

```
array([[ 0.4967, -0.1383,  0.6477,  1.523 ],
       [-0.4695,  0.5426, -0.4634, -0.4657],
       [ 0.242 , -1.9133, -1.7249, -0.5623],
       [-1.0128,  0.3142, -0.908 , -1.4123]])
```

We can also create a Boolean for each element.

```
data
```

```
array([[ 0.4967, -0.1383,  0.6477,  1.523 ],
       [-0.2342, -0.2341,  1.5792,  0.7674],
       [-0.4695,  0.5426, -0.4634, -0.4657],
       [ 0.242 , -1.9133, -1.7249, -0.5623],
       [-1.0128,  0.3142, -0.908 , -1.4123],
       [ 1.4656, -0.2258,  0.0675, -1.4247],
       [-0.5444,  0.1109, -1.151 ,  0.3757]])
```

```
data < 0
```

```
array([[False,  True, False, False],
       [ True,  True, False, False],
       [ True, False,  True,  True],
       [False,  True,  True,  True],
       [ True, False,  True,  True],
       [False,  True, False,  True],
       [ True, False,  True, False]])
```

```python
data[data < 0] = 0
data
```

```
array([[0.4967, 0.    , 0.6477, 1.523 ],
       [0.    , 0.    , 1.5792, 0.7674],
       [0.    , 0.5426, 0.    , 0.    ],
       [0.242 , 0.    , 0.    , 0.    ],
       [0.    , 0.3142, 0.    , 0.    ],
       [1.4656, 0.    , 0.0675, 0.    ],
       [0.    , 0.1109, 0.    , 0.3757]])
```

Universal Functions: Fast Element-Wise Array Functions

> A universal function, or ufunc, is a function that performs element-wise operations on data in ndarrays. You can think of them as fast vectorized wrappers for simple functions that take one or more scalar values and produce one or more scalar results.

```python
arr = np.arange(10)
arr
```

```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

```python
np.sqrt(arr)
```

```
array([0.    , 1.    , 1.4142, 1.7321, 2.    , 2.2361, 2.4495, 2.6458,
       2.8284, 3.    ])
```

Like above, we can raise a single value to a NumPy array of powers.

```python
2**arr
```

```
array([  1,   2,   4,   8,  16,  32,  64, 128, 256, 512])
```

`np.exp(x)` is e^x.

```
np.exp(arr)
```

```
array([1.0000e+00, 2.7183e+00, 7.3891e+00, 2.0086e+01, 5.4598e+01,
       1.4841e+02, 4.0343e+02, 1.0966e+03, 2.9810e+03, 8.1031e+03])
```

Table 4-4 from McKinney (2022) summarizes fast, element-wise unary functions:

- `abs`, `fabs`: Compute the absolute value element-wise for integer, floating-point, or complex values
- `sqrt`: Compute the square root of each element (equivalent to arr ** 0.5)
- `square`: Compute the square of each element (equivalent to arr ** 2)
- `exp`: Compute the exponent e^x of each element
- `log`, `log10`, `log2`, `log1p`: Natural logarithm (base e), log base 10, log base 2, and log(1 + x), respectively
- `sign`: Compute the sign of each element: 1 (positive), 0 (zero), or –1 (negative)
- `ceil`: Compute the ceiling of each element (i.e., the smallest integer greater than or equal to that number)
- `floor`: Compute the floor of each element (i.e., the largest integer less than or equal to each element)
- `rint`: Round elements to the nearest integer, preserving the dtype
- `modf`: Return fractional and integral parts of array as a separate array
- `isnan`: Return boolean array indicating whether each value is NaN (Not a Number)
- `isfinite`, `isinf`: Return boolean array indicating whether each element is finite (non-inf, non-NaN) or infinite, respectively
- `cos`, `cosh`, `sin`, `sinh`, `tan`, `tanh`: Regular and hyperbolic trigonometric functions
- `arccos`, `arccosh`, `arcsin`, `arcsinh`, `arctan`, `arctanh`: Inverse trigonometric functions
- `logical_not`: Compute truth value of not x element-wise (equivalent to ~arr).

These "unary" functions operate on one array and return a new array with the same shape. There are also "binary" functions that operate on two arrays and return one array.

```
np.random.seed(42)
x = np.random.randn(8)
y = np.random.randn(8)
```

```
x
```

```
array([ 0.4967, -0.1383,  0.6477,  1.523 , -0.2342, -0.2341,  1.5792,
        0.7674])
```

```
y
```

```
array([-0.4695,  0.5426, -0.4634, -0.4657,  0.242 , -1.9133, -1.7249,
       -0.5623])
```

```
np.maximum(x, y)
```

```
array([ 0.4967,  0.5426,  0.6477,  1.523 ,  0.242 , -0.2341,  1.5792,
        0.7674])
```

Table 4-5 from McKinney (2022) summarizes fast, element-wise binary functions:

- `add`: Add corresponding elements in arrays
- `subtract`: Subtract elements in second array from first array
- `multiply`: Multiply array elements
- `divide`, `floor_divide`: Divide or floor divide (truncating the remainder)
- `power`: Raise elements in first array to powers indicated in second array
- `maximum`, `fmax`: Element-wise maximum; `fmax` ignores `NaN`
- `minimum`, `fmin`: Element-wise minimum; `fmin` ignores `NaN`
- `mod`: Element-wise modulus (remainder of division)
- `copysign`: Copy sign of values in second argument to values in first argument
- `greater`, `greater_equal`, `less`, `less_equal`, `equal`, `not_equal`: Perform element-wise comparison, yielding boolean array (equivalent to infix operators >, >=, <, <=, ==, !=)
- `logical_and`, `logical_or`, `logical_xor`: Compute element-wise truth value of logical operation (equivalent to infix operators & |, ^)

Array-Oriented Programming with Arrays

> Using NumPy arrays enables you to express many kinds of data processing tasks as concise array expressions that might otherwise require writing loops. This practice of replacing explicit loops with array expressions is commonly referred to as vectorization. In general, vectorized array operations will often be one or two (or more) orders of magnitude faster than their pure Python equivalents, with the biggest impact in any kind of numerical computations. Later, in Appendix A, I explain broadcasting, a powerful method for vectorizing computations.

Expressing Conditional Logic as Array Operations

> The numpy.where function is a vectorized version of the ternary expression x if condition else y.

`np.where()` is an if-else statement, like Excel's `if()`.

```
xarr = np.array([1.1, 1.2, 1.3, 1.4, 1.5])
yarr = np.array([2.1, 2.2, 2.3, 2.4, 2.5])
cond = np.array([True, False, True, True, False])
```

```
np.where(cond, xarr, yarr)
```

```
array([1.1, 2.2, 1.3, 1.4, 2.5])
```

We could use a list comprehension instead, but it takes longer to type, read, and troubleshoot.

```
np.array([(x if c else y) for x, y, c in zip(xarr, yarr, cond)])
```

```
array([1.1, 2.2, 1.3, 1.4, 2.5])
```

`np.select()` lets us test more than one condition and has a default value if no condition is met.

```
np.select(
    condlist=[cond==True, cond==False],
    choicelist=[xarr, yarr]
)
```

```
array([1.1, 2.2, 1.3, 1.4, 2.5])
```

Mathematical and Statistical Methods

> A set of mathematical functions that compute statistics about an entire array or about the data along an axis are accessible as methods of the array class. You can use aggregations (often called reductions) like sum, mean, and std (standard deviation) either by calling the array instance method or using the top-level NumPy function.

```
np.random.seed(42)
arr = np.random.randn(5, 4)
arr
```

```
array([[ 0.4967, -0.1383,  0.6477,  1.523 ],
       [-0.2342, -0.2341,  1.5792,  0.7674],
       [-0.4695,  0.5426, -0.4634, -0.4657],
       [ 0.242 , -1.9133, -1.7249, -0.5623],
       [-1.0128,  0.3142, -0.908 , -1.4123]])
```

```
arr.mean()
```

```
-0.1713
```

```
arr.sum()
```

```
-3.4260
```

The aggregation methods above aggregated the whole array. We can use the `axis` argument to aggregate columns (`axis=0`) and rows (`axis=1`).

```
arr.mean(axis=1)
```

```
array([ 0.6323,  0.4696, -0.214 , -0.9896, -0.7547])
```

```
arr[0].mean()
```

```
0.6323
```

```
arr[1].mean()
```

```
0.4696
```

```
arr.mean(axis=0)
```

```
array([-0.1956, -0.2858, -0.1739, -0.03  ])
```

```
arr[:, 0].mean()
```

```
-0.1956
```

```
arr[:, 1].mean()
```

```
-0.2858
```

The `.cumsum()` method returns the sum of all previous elements.

```
arr = np.array([0, 1, 2, 3, 4, 5, 6, 7]) # same output as np.arange(8)
arr.cumsum()
```

```
array([ 0,  1,  3,  6, 10, 15, 21, 28])
```

We can also use the `.cumsum()` method along the axis of a multi-dimensional array.

```
arr = np.array([[0, 1, 2], [3, 4, 5], [6, 7, 8]])
arr
```

```
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
```

```
arr.cumsum(axis=0)
```

```
array([[ 0,  1,  2],
       [ 3,  5,  7],
       [ 9, 12, 15]])
```

```
arr.cumprod(axis=1)
```

```
array([[  0,   0,   0],
       [  3,  12,  60],
       [  6,  42, 336]])
```

Table 4-6 from McKinney (2022) summarizes basic statistical methods:

- `sum`: Sum of all the elements in the array or along an axis; zero-length arrays have sum 0
- `mean`: Arithmetic mean; zero-length arrays have NaN mean
- `std`, `var`: Standard deviation and variance, respectively, with optional degrees of freedom adjustment (default denominator n)
- `min`, `max`: Minimum and maximum
- `argmin`, `argmax`: Indices of minimum and maximum elements, respectively
- `cumsum`: Cumulative sum of elements starting from 0
- `cumprod`: Cumulative product of elements starting from 1

McKinney Chapter 4 - Practice - Blank

```python
import numpy as np
```

```python
%precision 4
```

```
'%.4f'
```

Announcements

Five-Minute Review

Practice

Create a 1-dimensional array `a1` that counts from 0 to 24 by 1.

Create a 1-dimentional array `a2` that counts from 0 to 24 by 3.

Create a 1-dimentional array `a3` that counts from 0 to 100 by multiples of 3 or 5.

Create a 1-dimensional array `a4` that contains the squares of the even integers through 100,000.

Write a function `calc_pv()` that mimic Excel's `PV` function.

Excel's present value function is: `=PV(rate, nper, pmt, [fv], [type])`

The present value of an annuity payment is: $PV_{pmt} = \frac{pmt}{rate} \times \left(1 - \frac{1}{(1+rate)^{nper}}\right)$

The present value of a lump sum is: $PV_{fv} = \frac{fv}{(1+rate)^{nper}}$

Write a function `calc_fv()` that mimic Excel's `FV` function.

Excel's future value function is: `=FV(rate, nper, pmt, [pv], [type])`

Replace the negative values in `data` with -1 and positive values with +1.

```
np.random.seed(42)
data = np.random.randn(4, 4)
```

Write a function `calc_n()` that calculates the number of payments that generate `x`% of the present value of a perpetuity.

The present value of a growing perpetuity is $PV = \frac{C_1}{r-g}$, and the present value of a growing annuity is $PV = \frac{C_1}{r-g}\left[1 - \left(\frac{1+g}{1+r}\right)^t\right]$.

Write a function that calculates the internal rate of return of a NumPy array of cash flows.

Write a function `calc_returns()` that accepts *NumPy arrays* of prices and dividends and returns a *NumPy array* of returns.

```
prices = np.array([100, 150, 100, 50, 100, 150, 100, 150])
dividends = np.array([1, 1, 1, 1, 2, 2, 2, 2])
```

Rewrite the function `calc_returns()` as `calc_returns_2()` so it returns *NumPy arrays* of returns, capital gains yields, and dividend yields.

Write a function `rescale()` to rescale and shift numbers so that they cover the range [0, 1]

Input: `np.array([18.5, 17.0, 18.0, 19.0, 18.0])`
Output: `np.array([0.75, 0.0, 0.5, 1.0, 0.5])`

Write a function `calc_portval()` that accepts a dictionary of prices and share holdings and returns the portfolio value

First convert your dictionary to a NumPy array with one column for prices and another for shares.

```
data = {
    "AAPL": (150.25, 10),  # (price, shares)
    "GOOGL": (2750.00, 2),
    "MSFT": (300.75, 5)
}
```

Write functions `calc_var()` and `calc_std()` that calculate variance and standard deviation.

NumPy's `.var()` and `.std()` methods return *population* statistics (i.e., denominators of n). The pandas equivalents return *sample* statistics (denominators of $n-1$), which are more appropriate for financial data analysis where we have a sample instead of a population.

Your `calc_var()` and `calc_std()` functions should have a `sample` argument that is `True` by default so both functions return sample statistics by default.

Write a function `calc_ret()` to convert quantitative returns to qualitative returns

Returns within one standard deviation of the mean are "Medium". Returns less than one standard deviation below the mean are "Low", and returns greater than one standard deviation above the mean are "High".

```
np.random.seed(42)
returns = np.random.randn(100)
```

McKinney Chapter 4 - Practice - Sec 02

```
import numpy as np
```

```
%precision 4
```

```
'%.4f'
```

Announcements

1. Keep signing up for project teams on Canvas > People > Projects
2. Claim a team, then ask to increase its limit if you want more than four teammates
3. Keep adding and voting for students choice topics

Five-Minute Review

NumPy is the foundation of scientific computing in Python! NumPy is also the foundation of pandas, so all these tools in this notebook are relevant to pandas, too!

Here are some random data for us to quickly review NumPy with. The `np.random.seed()` function makes our random number draws repeatable.

```
np.random.seed(42)
my_arr = np.random.randn(3, 5)
my_arr
```

```
array([[ 0.4967, -0.1383,  0.6477,  1.523 , -0.2342],
       [-0.2341,  1.5792,  0.7674, -0.4695,  0.5426],
       [-0.4634, -0.4657,  0.242 , -1.9133, -1.7249]])
```

Slicing works the same as with lists and tuples. As well, we can replace chained slices, like `[i][j][k]`, with a Matlab notation, like `[i. j. k]`

```
my_arr[2][0]
```

```
-0.4634
```

```
my_arr[2, 0]
```

```
-0.4634
```

What if we want the *first two rows?*

```
my_arr[:2]
```

```
array([[ 0.4967, -0.1383,  0.6477,  1.523 , -0.2342],
       [-0.2341,  1.5792,  0.7674, -0.4695,  0.5426]])
```

What if we want the *first two columns?*

```
my_arr[:, :2]
```

```
array([[ 0.4967, -0.1383],
       [-0.2341,  1.5792],
       [-0.4634, -0.4657]])
```

What about linear algebra in NumPy?

```
A = np.array([[1, 2, 3], [4, 5, 6]])
B = np.array([[1, 2, 3], [4, 5, 6]])
print(f'A is:\n {A}\n B is:\n {B}')
```

```
A is:
 [[1 2 3]
 [4 5 6]]
 B is:
 [[1 2 3]
 [4 5 6]]
```

We can use the `.dot()` method to chain matrix multiplication.

```
A.dot(B.T)
```

```
array([[14, 32],
       [32, 77]])
```

The `@` operator is matrix multiplication with NumPy arrays.

```
A @ B.T
```

```
array([[14, 32],
       [32, 77]])
```

The `*` symbol is *elementwise* multiplication.

```
A * B
```

```
array([[ 1,  4,  9],
       [16, 25, 36]])
```

The `np.where()` and `np.select()` functions are the NumPy equivalents of Excel's `if()`.

Say we want to replace values in `my_arr` greater than 0.5 with 0.5.

```
my_arr
```

```
array([[ 0.4967, -0.1383,  0.6477,  1.523 , -0.2342],
       [-0.2341,  1.5792,  0.7674, -0.4695,  0.5426],
       [-0.4634, -0.4657,  0.242 , -1.9133, -1.7249]])
```

```
np.where(my_arr > 0.5, 0.5, my_arr)
```

```
array([[ 0.4967, -0.1383,  0.5   ,  0.5   , -0.2342],
       [-0.2341,  0.5   ,  0.5   , -0.4695,  0.5   ],
       [-0.4634, -0.4657,  0.242 , -1.9133, -1.7249]])
```

Or, we could use `np.minimum()` instead.

```python
np.minimum(0.5, my_arr)
```

```
array([[ 0.4967, -0.1383,  0.5   ,  0.5   , -0.2342],
       [-0.2341,  0.5   ,  0.5   , -0.4695,  0.5   ],
       [-0.4634, -0.4657,  0.242 , -1.9133, -1.7249]])
```

Now, say we want to replace values in `my_arr` greater than 0 with 0 and greater than 0.5 with 0.5.

```python
np.where(
    my_arr>0.5, # first condition is most restrictive
    0.5, # if first condition True
    np.where( # otherwise
        my_arr>0, # second condition is less restrictive
        0, # if second condition True
        my_arr # otherwise
    )
)
```

```
array([[ 0.    , -0.1383,  0.5   ,  0.5   , -0.2342],
       [-0.2341,  0.5   ,  0.5   , -0.4695,  0.5   ],
       [-0.4634, -0.4657,  0.    , -1.9133, -1.7249]])
```

We have a better way to test more than one condition! `np.select()`!

```python
np.select(
    condlist=[my_arr>0.5, my_arr>0],
    choicelist=[0.5, 0],
    default=my_arr
)
```

```
array([[ 0.    , -0.1383,  0.5   ,  0.5   , -0.2342],
       [-0.2341,  0.5   ,  0.5   , -0.4695,  0.5   ],
       [-0.4634, -0.4657,  0.    , -1.9133, -1.7249]])
```

Practice

Create a 1-dimensional array a1 that counts from 0 to 24 by 1.

```
np.array(range(25))
```

```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
       17, 18, 19, 20, 21, 22, 23, 24])
```

```
np.array(range(25))
```

```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
       17, 18, 19, 20, 21, 22, 23, 24])
```

```
a1 = np.arange(25)
a1
```

```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
       17, 18, 19, 20, 21, 22, 23, 24])
```

Create a 1-dimentional array a2 that counts from 0 to 24 by 3.

```
a2 = np.arange(0, 25, 3)
a2
```

```
array([ 0,  3,  6,  9, 12, 15, 18, 21, 24])
```

Create a 1-dimentional array a3 that counts from 0 to 100 by multiples of 3 or 5.

```
a3 = np.array([i for i in range(101) if (i%3==0) | (i%5==0)])
a3
```

```
array([  0,   3,   5,   6,   9,  10,  12,  15,  18,  20,  21,  24,  25,
        27,  30,  33,  35,  36,  39,  40,  42,  45,  48,  50,  51,  54,
        55,  57,  60,  63,  65,  66,  69,  70,  72,  75,  78,  80,  81,
        84,  85,  87,  90,  93,  95,  96,  99, 100])
```

```
toy = np.arange(5)
toy[toy%2==0]
```

```
array([0, 2, 4])
```

```
my_arr_0_100 = np.arange(101)
a3_alt = my_arr_0_100[(my_arr_0_100%3==0) | (my_arr_0_100%5==0)]
a3_alt
```

```
array([  0,   3,   5,   6,   9,  10,  12,  15,  18,  20,  21,  24,  25,
        27,  30,  33,  35,  36,  39,  40,  42,  45,  48,  50,  51,  54,
        55,  57,  60,  63,  65,  66,  69,  70,  72,  75,  78,  80,  81,
        84,  85,  87,  90,  93,  95,  96,  99, 100])
```

```
(a3 == a3_alt).all()
```

```
np.True_
```

```
np.allclose(a3, a3_alt)
```

```
True
```

Create a 1-dimensional array `a4` that contains the squares of the even integers through 100,000.

```
%timeit np.arange(0, 100_001, 2) ** 2
a4 = np.arange(0, 100_001, 2) ** 2
a4
```

```
46.7  s ± 7.59  s per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

```
array([         0,          4,         16, ...,  9999200016,
        9999600004, 10000000000], shape=(50001,))
```

```
%timeit a4_alt = np.array([i**2 for i in range(100_001) if i%2==0])
a4_alt = np.array([i**2 for i in range(100_001) if i%2==0])
a4_alt
```

```
29.7 ms ± 2.85 ms per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

```
array([          0,           4,          16, ...,  9999200016,
        9999600004, 10000000000], shape=(50001,))
```

```
(a4 == a4_alt).all()
```

```
np.True_
```

```
np.allclose(a4, a4_alt*1.0000001)
```

```
True
```

Write a function `calc_pv()` that mimic Excel's `PV` function.

Excel's present value function is: `=PV(rate, nper, pmt, [fv], [type])`

The present value of an annuity payment is: $PV_{pmt} = \frac{pmt}{rate} \times \left(1 - \frac{1}{(1+rate)^{nper}}\right)$

The present value of a lump sum is: $PV_{fv} = \frac{fv}{(1+rate)^{nper}}$

```
def calc_pv(rate, nper, pmt=None, fv=None, type='END'):
    # or we could set the pmt and fv defaults to 0
    if pmt is None:
        pmt = 0
    if fv is None:
        fv = 0
    if type not in ['BGN', 'END']:
        raise ValueError(f'type must be BGN or END; you provided {type}')

    pv_pmt = (pmt / rate) * (1 - (1 + rate) ** (-nper))
    pv_fv = fv * (1 + rate) ** (-nper)
    pv = pv_pmt + pv_fv

    if type == 'BGN':
```

```
        pv *= 1 + rate

    return -pv
```

```
calc_pv(rate=0.1, nper=10, pmt=100, fv=1_000)
```

```
-1000.0000
```

```
calc_pv(rate=0.1, nper=10, pmt=-100, fv=-1_000)
```

```
1000.0000
```

Write a function `calc_fv()` that mimic Excel's `FV` function.

Excel's future value function is: `=FV(rate, nper, pmt, [pv], [type])`

```
def calc_fv(rate, nper, pmt=None, pv=None, type='END'):
    if pmt is None:
        pmt = 0
    if pv is None:
        pv = 0
    if type not in ['BGN', 'END']:
        raise ValueError('type must be BGN or END')

    fv_pmt = (pmt / rate) * ((1 + rate)**nper - 1)
    fv_pv = pv * (1 + rate)**nper
    fv = fv_pmt + fv_pv

    if type == 'BGN':
        fv *= 1 + rate

    return -fv
```

```
calc_fv(rate=0.1, nper=10, pmt=100, pv=-1_000)
```

```
1000.0000
```

The rule of 72!

```
calc_fv(rate=0.072, nper=10, pmt=0, pv=-1_000)
```

```
2004.2314
```

Replace the negative values in `data` with -1 and positive values with +1.

```
np.random.seed(42)
data = np.random.randn(4, 4)

data
```

```
array([[ 0.4967, -0.1383,  0.6477,  1.523 ],
       [-0.2342, -0.2341,  1.5792,  0.7674],
       [-0.4695,  0.5426, -0.4634, -0.4657],
       [ 0.242 , -1.9133, -1.7249, -0.5623]])
```

We have at least three good solutions!

1. Slicing and brodacsting
2. `np.where()`
3. `np.select()`

First, here is the slicing and broadcasting solution.

```
data[data < 0] = -1
data[data > 0] = +1

data # here "1." and "-1." indicate that these values are floats
```

```
array([[ 1., -1.,  1.,  1.],
       [-1., -1.,  1.,  1.],
       [-1.,  1., -1., -1.],
       [ 1., -1., -1., -1.]])
```

Second, here is the `np.where()` solution. NumPy's `np.where()` function has the same logic as Excel's `if()` function! I will recreate `data` so we start from the same point.

```
np.random.seed(42)
data = np.random.randn(4, 4)

np.where(data < 0, -1, np.where(data > 0, +1, data))
```

```
array([[ 1., -1.,  1.,  1.],
       [-1., -1.,  1.,  1.],
       [-1.,  1., -1., -1.],
       [ 1., -1., -1., -1.]])
```

The nested `np.where()` function calls can get confusing! An option is to insert white space!

```
np.random.seed(42)
data = np.random.randn(4, 4)

np.where(
    data < 0, # condition
    -1, # result if True
    np.where(data > 0, +1, data) # result if False
)
```

```
array([[ 1., -1.,  1.,  1.],
       [-1., -1.,  1.,  1.],
       [-1.,  1., -1., -1.],
       [ 1., -1., -1., -1.]])
```

Third, here is the `np.select()` solution. NumPy's `np.select()` function lets us test *many* conditions! I will recreate `data` so we start from the same point.

```
np.random.seed(42)
data = np.random.randn(4, 4)

np.select(
    condlist=[data<0, data>0],
    choicelist=[-1, +1],
    default=data
)
```

```
array([[ 1., -1.,  1.,  1.],
       [-1., -1.,  1.,  1.],
       [-1.,  1., -1., -1.],
       [ 1., -1., -1., -1.]])
```

Write a function `calc_n()` that calculates the number of payments that generate `x`% of the present value of a perpetuity.

The present value of a growing perpetuity is $PV = \frac{C_1}{r-g}$, and the present value of a growing annuity is $PV = \frac{C_1}{r-g}\left[1 - \left(\frac{1+g}{1+r}\right)^t\right]$.

Your `npmts()` should accept arguments `c1`, `r`, and `g` that represent C_1, r, and g. The present value of a growing perpetuity is $PV = \frac{C_1}{r-g}$, and the present value of a growing annuity is $PV = \frac{C_1}{r-g}\left[1 - \left(\frac{1+g}{1+r}\right)^t\right]$.

We can use the growing annuity and perpetuity formulas to show: $x = \left[1 - \left(\frac{1+g}{1+r}\right)^t\right]$.

Then: $1 - x = \left(\frac{1+g}{1+r}\right)^t$.

Finally: $t = \frac{\log(1-x)}{\log\left(\frac{1+g}{1+r}\right)}$

We do not need to accept an argument `c1` because C_1 cancels out!

```
def npmts(x, r, g):
    return np.log(1-x) / np.log((1 + g) / (1 + r))
```

```
npmts(0.5, 0.1, 0.05)
```

```
14.9000
```

Write a function that calculates the internal rate of return of a NumPy array of cash flows.

Let us pick a set of cash flows `c` where we know the internal rate of return! For the following `c`, the IRR is 10%.

```
c = np.array([-100, +110])
r = 0.1
```

First we need an function to calculate net present value (NPV) from `c` and `r`! The `npv()` function below uses NumPy arrays to calculate NPV as:

$$NPV = \sum_{t=0}^{T} \frac{c_t}{(1+r)^t}$$

```python
def calc_npv(r, c):
    t = np.arange(len(c))
    return np.sum(c / (1 + r)**t)
```

```python
calc_npv(r=r, c=c)
```

```
-0.0000
```

We can use a `for` loop to guess IRR values until we find an NPV close to zero (or reach the maximum number of iterations in `max_iter`). We can use the Newton-Rapshon method to make smarter guesses. If we have function $f(x)$ and guess x_t, our next guess should be $x_{t+1} = x_t - \frac{f(x_t)}{f'(x_t)}$. Here our $f(x)$ is $NPV(r)$, and we can approximate $f'(x_t)$ as $\frac{NPV(r+step)-NPV(r)}{step}$. We will make guess until $|NPV| < tol$.

```python
def calc_irr(c, guess=0, step=1e-6, tol=1e-6, max_iter=1_000, verbose=False):
    irr = guess
    for i in range(max_iter):
        npv = calc_npv(r=irr, c=c)
        if abs(npv) < tol:
            if verbose:
                print(f'IRR: {irr:0.4f}\nNPV: {npv:0.4f}\nIterations: {i+1:d}')
            return irr

        deriv = (calc_npv(r=irr+step, c=c) - npv) / step
        irr -= npv / deriv

    raise ValueError(f'NPV did not converge to zero after {i+1} iterations.')
```

```python
calc_irr(c=c, verbose=True)
```

```
IRR: 0.1000
NPV: 0.0000
Iterations: 4
```

```
0.1000
```

```python
calc_irr(c=np.array([-100, 10, 10, 10, 10, 110]), verbose=True)
```

```
IRR: 0.1000
NPV: 0.0000
Iterations: 5
```

```
0.1000
```

```
# calc_irr(c=np.array([-2000, 1000, -500]))
```

Write a function `calc_returns()` that accepts *NumPy arrays* of prices and dividends and returns a *NumPy array* of returns.

```
prices = np.array([100, 150, 100, 50, 100, 150, 100, 150])
dividends = np.array([1, 1, 1, 1, 2, 2, 2, 2])
```

We want to slice our arrays to "lag" or "shift" them! For example, we slice the `prices` array to calculate capital gains as follows.

```
prices[1:] - prices[:-1]
```

```
array([ 50, -50, -50,  50,  50, -50,  50])
```

```
def calc_returns(p, d):
    return (p[1:] - p[:-1] + d[1:]) / p[:-1]
```

```
calc_returns(p=prices, d=dividends)
```

```
array([ 0.51  , -0.3267, -0.49  ,  1.04  ,  0.52  , -0.32  ,  0.52  ])
```

Rewrite the function `calc_returns()` as `calc_returns_2()` so it returns *NumPy arrays* of returns, capital gains yields, and dividend yields.

```
def calc_returns_2(p, d):
    cg = p[1:] / p[:-1] - 1
    dp = d[1:] / p[:-1]
    r = cg + dp
    return {'r': r, 'cg': cg, 'dp': dp}
```

```python
calc_returns_2(p=prices, d=dividends)
```

```
{'r': array([ 0.51  , -0.3267, -0.49  ,  1.04  ,  0.52  , -0.32  ,  0.52  ]),
 'cg': array([ 0.5   , -0.3333, -0.5   ,  1.    ,  0.5   , -0.3333,  0.5   ]),
 'dp': array([0.01  , 0.0067, 0.01  , 0.04  , 0.02  , 0.0133, 0.02  ])}
```

Write a function `rescale()` to rescale and shift numbers so that they cover the range [0, 1]

Input: `np.array([18.5, 17.0, 18.0, 19.0, 18.0])`
Output: `np.array([0.75, 0.0, 0.5, 1.0, 0.5])`

```python
x = np.array([18.5, 17.0, 18.0, 19.0, 18.0])
x
```

```
array([18.5, 17. , 18. , 19. , 18. ])
```

```python
def rescale(x):
    return (x - x.min()) / (x.max() - x.min())
```

```python
rescale(x=x)
```

```
array([0.75, 0.  , 0.5 , 1.  , 0.5 ])
```

Write a function `calc_portval()` that accepts a dictionary of prices and share holdings and returns the portfolio value

First convert your dictionary to a NumPy array with one column for prices and another for shares.

```python
data = {
    "AAPL": (150.25, 10),  # (price, shares)
    "GOOGL": (2750.00, 2),
    "MSFT": (300.75, 5)
}
```

```python
def calc_portval(data):
    x = np.array(list(data.values()))
    return x.prod(axis=1).sum()
```

```python
calc_portval(data=data)
```

```
8506.2500
```

Write functions `calc_var()` and `calc_std()` that calculate variance and standard deviation.

NumPy's `.var()` and `.std()` methods return *population* statistics (i.e., denominators of n). The pandas equivalents return *sample* statistics (denominators of $n-1$), which are more appropriate for financial data analysis where we have a sample instead of a population.

Your `calc_var()` and `calc_std()` functions should have a `sample` argument that is `True` by default so both functions return sample statistics by default.

We can use the methods `.mean()` and `.sum()` instead of the functions `np.mean()` and `np.sum()`. I find this format easier to read because it puts the data first.

```python
def calc_var(x, sample=True):
    sq_err = (x - x.mean()) ** 2
    den = len(x)
    if sample:
        den -= 1
    return sq_err.sum() / den
```

We can re-use `calc_var()` in our `calc_std()` function.

```python
def calc_std(x, sample=True):
    return calc_var(x=x, sample=sample) ** 0.5
```

```python
np.random.seed(42)
arr = np.random.randn(1_000_000)
arr
```

```
array([ 0.4967, -0.1383,  0.6477, ..., -0.113 ,  1.4691,  0.4764],
      shape=(1000000,))
```

```python
calc_var(arr)
```

```
1.0004
```

```
calc_std(arr)
```

```
1.0002
```

```
arr.var(ddof=1) == calc_var(arr)
```

```
np.True_
```

```
arr.std(ddof=1) == calc_std(arr)
```

```
np.True_
```

Write a function `calc_ret()` to convert quantitative returns to qualitative returns

Returns within one standard deviation of the mean are "Medium". Returns less than one standard deviation below the mean are "Low", and returns greater than one standard deviation above the mean are "High".

```
np.random.seed(42)
returns = np.random.randn(10)
```

```
def calc_ret(r):
    mu = r.mean()
    sigma = r.std(ddof=1)
    return np.select(
        condlist=[
            r<(mu-sigma),
            r<=(mu+sigma),
            r>(mu+sigma)
        ],
        choicelist=[
            'Low',
            'Medium',
            'High'
        ],
        default=''
    )
```

```python
calc_ret(returns)
```

```
array(['Medium', 'Medium', 'Medium', 'High', 'Medium', 'Medium', 'High',
       'Medium', 'Low', 'Medium'], dtype='<U6')
```

McKinney Chapter 4 - Practice - Sec 03

```
import numpy as np
```

```
%precision 4
```

```
'%.4f'
```

Announcements

1. Keep signing up for project teams on Canvas > People > Projects
2. Claim a team, then ask to increase its limit if you want more than four teammates
3. Keep adding and voting for students choice topics

Five-Minute Review

NumPy is the foundation of scientific computing in Python! NumPy is also the foundation of pandas, so all these tools in this notebook are relevant to pandas, too!

Here are some random data for us to quickly review NumPy with. The `np.random.seed()` function makes our random number draws repeatable.

```
np.random.seed(42)
my_arr = np.random.randn(3, 5)
my_arr
```

```
array([[ 0.4967, -0.1383,  0.6477,  1.523 , -0.2342],
       [-0.2341,  1.5792,  0.7674, -0.4695,  0.5426],
       [-0.4634, -0.4657,  0.242 , -1.9133, -1.7249]])
```

Slicing works the same as with lists and tuples. As well, we can replace chained slices, like `[i][j][k]`, with a Matlab notation, like `[i. j. k]`

```
my_arr[2][4]
```

```
-1.7249
```

```
my_arr[2, 4]
```

```
-1.7249
```

What if we want the *first two rows?*

```
my_arr[:2]
```

```
array([[ 0.4967, -0.1383,  0.6477,  1.523 , -0.2342],
       [-0.2341,  1.5792,  0.7674, -0.4695,  0.5426]])
```

What if we want the *first two columns?*

```
my_arr[:, :2]
```

```
array([[ 0.4967, -0.1383],
       [-0.2341,  1.5792],
       [-0.4634, -0.4657]])
```

What about linear algebra in NumPy?

```
A = np.array([[1, 2, 3], [4, 5, 6]])
A
```

```
array([[1, 2, 3],
       [4, 5, 6]])
```

```
B = np.array([[1, 2, 3], [4, 5, 6]])
B
```

```
array([[1, 2, 3],
       [4, 5, 6]])
```

We can use the `.dot()` method to chain matrix multiplication.

```
A.dot(B.T)
```

```
array([[14, 32],
       [32, 77]])
```

The `@` operator is matrix multiplication with NumPy arrays.

```
A @ B.T
```

```
array([[14, 32],
       [32, 77]])
```

The `*` symbol is *elementwise* multiplication.

```
A * B
```

```
array([[ 1,  4,  9],
       [16, 25, 36]])
```

There is *another* way, too! The `np.matmul()` function is similar to the `.dot()` method!

```
np.matmul(A, B.T)
```

```
array([[14, 32],
       [32, 77]])
```

The `np.where()` and `np.select()` functions are the NumPy equivalents of Excel's `if()`.

Say we want to replace values in `my_arr` greater than 0.5 with 0.5.

```
my_arr
```

```
array([[ 0.4967, -0.1383,  0.6477,  1.523 , -0.2342],
       [-0.2341,  1.5792,  0.7674, -0.4695,  0.5426],
       [-0.4634, -0.4657,  0.242 , -1.9133, -1.7249]])
```

```
np.where(my_arr>0.5, 0.5, my_arr)
```

```
array([[ 0.4967, -0.1383,  0.5   ,  0.5   , -0.2342],
       [-0.2341,  0.5   ,  0.5   , -0.4695,  0.5   ],
       [-0.4634, -0.4657,  0.242 , -1.9133, -1.7249]])
```

Or, we could use `np.minimum()` instead.

```
np.minimum(0.5, my_arr)
```

```
array([[ 0.4967, -0.1383,  0.5   ,  0.5   , -0.2342],
       [-0.2341,  0.5   ,  0.5   , -0.4695,  0.5   ],
       [-0.4634, -0.4657,  0.242 , -1.9133, -1.7249]])
```

Now, say we want to replace values in `my_arr` greater than 0 with 0 and greater than 0.5 with 0.5.

```
np.where(
    my_arr>0.5, # first condition is most restrictive
    0.5, # if first condition True
    np.where( # otherwise
        my_arr>0, # second condition is less restrictive
        0, # if second condition True
        my_arr # otherwise
    )
)
```

```
array([[ 0.    , -0.1383,  0.5   ,  0.5   , -0.2342],
       [-0.2341,  0.5   ,  0.5   , -0.4695,  0.5   ],
       [-0.4634, -0.4657,  0.    , -1.9133, -1.7249]])
```

We have a better way to test more than one condition! `np.select()`!

```
np.select(
    condlist=[my_arr>0.5, my_arr>0],
    choicelist=[0.5, 0],
    default=my_arr
)
```

```
array([[ 0.    , -0.1383,  0.5   ,  0.5   , -0.2342],
       [-0.2341,  0.5   ,  0.5   , -0.4695,  0.5   ],
       [-0.4634, -0.4657,  0.    , -1.9133, -1.7249]])
```

Practice

Create a 1-dimensional array `a1` that counts from 0 to 24 by 1.

```
np.array(range(25))
```

```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
       17, 18, 19, 20, 21, 22, 23, 24])
```

```
a1 = np.arange(25)
a1
```

```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
       17, 18, 19, 20, 21, 22, 23, 24])
```

Create a 1-dimentional array `a2` that counts from 0 to 24 by 3.

```
a2 = np.arange(0, 25, 3)
a2
```

```
array([ 0,  3,  6,  9, 12, 15, 18, 21, 24])
```

Create a 1-dimentional array `a3` that counts from 0 to 100 by multiples of 3 or 5.

```
a3 = np.array([i for i in range(101) if (i%3==0) | (i%5==0)])
a3
```

```
array([  0,   3,   5,   6,   9,  10,  12,  15,  18,  20,  21,  24,  25,
        27,  30,  33,  35,  36,  39,  40,  42,  45,  48,  50,  51,  54,
        55,  57,  60,  63,  65,  66,  69,  70,  72,  75,  78,  80,  81,
        84,  85,  87,  90,  93,  95,  96,  99, 100])
```

```
zero_to_100 = np.arange(101)
a3_alt = zero_to_100[(zero_to_100 % 3 == 0) | (zero_to_100 % 5 == 0)]
a3_alt
```

```
array([  0,   3,   5,   6,   9,  10,  12,  15,  18,  20,  21,  24,  25,
        27,  30,  33,  35,  36,  39,  40,  42,  45,  48,  50,  51,  54,
        55,  57,  60,  63,  65,  66,  69,  70,  72,  75,  78,  80,  81,
        84,  85,  87,  90,  93,  95,  96,  99, 100])
```

```
(a3 == a3_alt).all()
```

```
np.True_
```

```
np.allclose(a3, a3_alt)
```

```
True
```

Create a 1-dimensional array a4 that contains the squares of the even integers through 100,000.

```
%timeit np.arange(0, 100_001, 2) ** 2
a4 = np.arange(0, 100_001, 2) ** 2
a4
```

```
43.3  s ± 10.3  s per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

```
array([          0,           4,          16, ...,  9999200016,
        9999600004, 10000000000], shape=(50001,))
```

```
%timeit a4_alt = np.array([i**2 for i in range(100_001) if i%2==0])
a4_alt = np.array([i**2 for i in range(100_001) if i%2==0])
a4_alt
```

```
29.5 ms ± 2.69 ms per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

```
array([          0,           4,          16, ...,  9999200016,
        9999600004, 10000000000], shape=(50001,))
```

```
(a4 == a4_alt).all()
```

```
np.True_
```

```
np.allclose(a4, a4_alt*1.0000001)
```

```
True
```

Write a function `calc_pv()` that mimic Excel's `PV` function.

Excel's present value function is: `=PV(rate, nper, pmt, [fv], [type])`

The present value of an annuity payment is: $PV_{pmt} = \frac{pmt}{rate} \times \left(1 - \frac{1}{(1+rate)^{nper}}\right)$

The present value of a lump sum is: $PV_{fv} = \frac{fv}{(1+rate)^{nper}}$

```
def calc_pv(rate, nper, pmt=None, fv=None, type=0):
    # or we could set the pmt and fv defaults to 0
    if pmt is None:
        pmt = 0
    if fv is None:
        fv = 0
    if type not in [0, 1]:
        raise ValueError(f'type must be 0 or 1; you provided {type}')

    pv_pmt = (pmt / rate) * (1 - (1 + rate)**(-nper))
    pv_fv = fv * (1 + rate)**(-nper)
    pv = pv_pmt + pv_fv

    if type == 1:
        pv *= 1 + rate

    return -pv
```

```
calc_pv(rate=0.1, nper=10, pmt=10, fv=100)
```

```
-100.0000
```

```
calc_pv(rate=0.1, nper=10, pmt=10, fv=100, type=1)
```

```
-110.0000
```

Write a function `calc_fv()` that mimic Excel's `FV` function.

Excel's future value function is: `=FV(rate, nper, pmt, [pv], [type])`

```
def calc_fv(rate, nper, pmt=None, pv=None, type='END'):
    if pmt is None:
        pmt = 0
    if pv is None:
        pv = 0
    if type not in ['BGN', 'END']:
        raise ValueError('type must be BGN or END')

    fv_pmt = (pmt / rate) * ((1 + rate)**nper - 1)
    fv_pv = pv * (1 + rate)**nper
    fv = fv_pmt + fv_pv

    if type == 'BGN':
        fv *= 1 + rate

    return -fv
```

```
calc_fv(rate=0.1, nper=10, pmt=10, pv=-100)
```

```
100.0000
```

The rule of 72!

```
calc_fv(rate=0.072, nper=10, pmt=0, pv=-100)
```

```
200.4231
```

Replace the negative values in `data` with -1 and positive values with +1.

```
np.random.seed(42)
data = np.random.randn(4, 4)

data
```

```
array([[ 0.4967, -0.1383,  0.6477,  1.523 ],
       [-0.2342, -0.2341,  1.5792,  0.7674],
       [-0.4695,  0.5426, -0.4634, -0.4657],
       [ 0.242 , -1.9133, -1.7249, -0.5623]])
```

We have at least three good solutions!

1. Slicing and brodacsting
2. `np.where()`
3. `np.select()`

First, here is the slicing and broadcasting solution.

```
data[data < 0] = -1
data[data > 0] = +1

data # here "1." and "-1." indicate that these values are floats
```

```
array([[ 1., -1.,  1.,  1.],
       [-1., -1.,  1.,  1.],
       [-1.,  1., -1., -1.],
       [ 1., -1., -1., -1.]])
```

Second, here is the `np.where()` solution. NumPy's `np.where()` function has the same logic as Excel's `if()` function! I will recreate `data` so we start from the same point.

```
np.random.seed(42)
data = np.random.randn(4, 4)

np.where(data < 0, -1, np.where(data > 0, +1, data))
```

```
array([[ 1., -1.,  1.,  1.],
       [-1., -1.,  1.,  1.],
       [-1.,  1., -1., -1.],
       [ 1., -1., -1., -1.]])
```

The nested `np.where()` function calls can get confusing! An option is to insert white space!

```
np.random.seed(42)
data = np.random.randn(4, 4)

np.where(
    data < 0, # condition
    -1, # result if True
    np.where(data > 0, +1, data) # result if False
)
```

```
array([[ 1., -1.,  1.,  1.],
       [-1., -1.,  1.,  1.],
       [-1.,  1., -1., -1.],
       [ 1., -1., -1., -1.]])
```

Third, here is the `np.select()` solution. NumPy's `np.select()` function lets us test *many* conditions! I will recreate `data` so we start from the same point.

```
np.random.seed(42)
data = np.random.randn(4, 4)

np.select(
    condlist=[data<0, data>0],
    choicelist=[-1, +1],
    default=data
)
```

```
array([[ 1., -1.,  1.,  1.],
       [-1., -1.,  1.,  1.],
       [-1.,  1., -1., -1.],
       [ 1., -1., -1., -1.]])
```

Write a function `calc_n()` that calculates the number of payments that generate `x`% of the present value of a perpetuity.

The present value of a growing perpetuity is $PV = \frac{C_1}{r-g}$, and the present value of a growing annuity is $PV = \frac{C_1}{r-g} \left[1 - \left(\frac{1+g}{1+r}\right)^t\right]$.

We can use the growing annuity and perpetuity formulas to show: $x = \left[1 - \left(\frac{1+g}{1+r}\right)^t\right]$.

Then: $1 - x = \left(\frac{1+g}{1+r}\right)^t$.

Finally: $t = \frac{\log(1-x)}{\log\left(\frac{1+g}{1+r}\right)}$

We do not need to accept an argument `c1` because C_1 cancels out!

```
def npmts(x, r, g):
    return np.log(1-x) / np.log((1 + g) / (1 + r))
```

```
npmts(0.5, 0.1, 0.05)
```

```
14.9000
```

Write a function that calculates the internal rate of return of a NumPy array of cash flows.

Let us pick a set of cash flows `c` where we know the internal rate of return! For the following `c`, the IRR is 10%.

```
c = np.array([-100, 110])
r = 0.1
```

First we need an function to calculate net present value (NPV) from `c` and `r`! The `npv()` function below uses NumPy arrays to calculate NPV as:

$$NPV = \sum_{t=0}^{T} \frac{c_t}{(1+r)^t}$$

```
def calc_npv(r, c):
    t = np.arange(len(c))
    return np.sum(c / (1 + r)**t)
```

```
calc_npv(r=r, c=c)
```

```
-0.0000
```

```
def calc_irr(c, guess=0, step=1e-6, tol=1e-6, max_iter=1_000, verbose=False):
    irr = guess
    for i in range(max_iter):
        npv = calc_npv(r=irr, c=c)
        if abs(npv) < tol:
```

```
            if verbose:
                print(f'IRR: {irr:0.4f}\nNPV: {npv:0.4f}\nIterations: {i+1:d}')
            return irr
        deriv = (calc_npv(r=irr+step, c=c) - npv) / step
        irr -= npv / deriv

    raise ValueError(f'NPV did not converge to zero after {i+1:d} iterations')
```

```
calc_irr(c=c, verbose=True)
```

```
IRR: 0.1000
NPV: 0.0000
Iterations: 4
```

```
0.1000
```

```
calc_irr(c=np.array([-100, 10, 10, 10, 10, 110]), verbose=True)
```

```
IRR: 0.1000
NPV: 0.0000
Iterations: 5
```

```
0.1000
```

```
# calc_irr(c=np.array([-2000, 1000, -500]))
```

Write a function `calc_returns()` that accepts *NumPy arrays* of prices and dividends and returns a *NumPy array* of returns.

```
prices = np.array([100, 150, 100, 50, 100, 150, 100, 150])
dividends = np.array([1, 1, 1, 1, 2, 2, 2, 2])
```

We want to slice our arrays to "lag" or "shift" them! For example, we slice the `prices` array to calculate capital gains as follows.

```
prices[1:] - prices[:-1]
```

```
array([ 50, -50, -50,  50,  50, -50,  50])
```

```
def calc_returns(p, d):
    return (p[1:] - p[:-1] + d[1:]) / p[:-1]
```

```
calc_returns(p=prices, d=dividends)
```

```
array([ 0.51  , -0.3267, -0.49  ,  1.04  ,  0.52  , -0.32  ,  0.52  ])
```

Rewrite the function `calc_returns()` as `calc_returns_2()` so it returns ***NumPy arrays*** of returns, capital gains yields, and dividend yields.

```
def calc_returns_2(p, d):
    cg = p[1:] / p[:-1] - 1
    dp = d[1:] / p[:-1]
    r = cg + dp
    return {'r': r, 'cg': cg, 'dp': dp}
```

```
calc_returns_2(p=prices, d=dividends)
```

```
{'r': array([ 0.51  , -0.3267, -0.49  ,  1.04  ,  0.52  , -0.32  ,  0.52  ]),
 'cg': array([ 0.5   , -0.3333, -0.5   ,  1.    ,  0.5   , -0.3333,  0.5   ]),
 'dp': array([0.01  , 0.0067, 0.01  , 0.04  , 0.02  , 0.0133, 0.02  ])}
```

Write a function `rescale()` to rescale and shift numbers so that they cover the range [0, 1]

Input: `np.array([18.5, 17.0, 18.0, 19.0, 18.0])`
Output: `np.array([0.75, 0.0, 0.5, 1.0, 0.5])`

```
x = np.array([18.5, 17.0, 18.0, 19.0, 18.0])
x
```

```
array([18.5, 17. , 18. , 19. , 18. ])
```

```python
def rescale(x):
    return (x - x.min()) / (x.max() - x.min())
```

```python
rescale(x=x)
```

```
array([0.75, 0.  , 0.5 , 1.  , 0.5 ])
```

Write a function `calc_portval()` that accepts a dictionary of prices and share holdings and returns the portfolio value

First convert your dictionary to a NumPy array with one column for prices and another for shares.

```python
data = {
    "AAPL": (150.25, 10),  # (price, shares)
    "GOOGL": (2750.00, 2),
    "MSFT": (300.75, 5)
}
```

```python
def calc_portval(data):
    x = np.array(list(data.values()))
    return x.prod(axis=1).sum()
```

```python
calc_portval(data=data)
```

```
8506.2500
```

Write functions `calc_var()` and `calc_std()` that calculate variance and standard deviation.

NumPy's `.var()` and `.std()` methods return *population* statistics (i.e., denominators of n). The pandas equivalents return *sample* statistics (denominators of $n-1$), which are more appropriate for financial data analysis where we have a sample instead of a population.

Your `calc_var()` and `calc_std()` functions should have a `sample` argument that is `True` by default so both functions return sample statistics by default.

```
def calc_var(x, sample=True):
    sq_err = (x - x.mean()) ** 2
    den = len(x)
    if sample:
        den -= 1
    return sq_err.sum() / den
```

We can re-use `calc_var()` in our `calc_std()` function.

```
def calc_std(x, sample=True):
    return calc_var(x=x, sample=sample) ** 0.5
```

```
np.random.seed(42)
arr = np.random.randn(1_000_000)
arr
```

```
array([ 0.4967, -0.1383,  0.6477, ..., -0.113 ,  1.4691,  0.4764],
      shape=(1000000,))
```

```
calc_var(arr)
```

```
1.0004
```

```
calc_std(arr)
```

```
1.0002
```

```
arr.var(ddof=1) == calc_var(arr)
```

```
np.True_
```

```
arr.std(ddof=1) == calc_std(arr)
```

```
np.True_
```

Write a function `calc_ret()` to convert quantitative returns to qualitative returns

Returns within one standard deviation of the mean are "Medium". Returns less than one standard deviation below the mean are "Low", and returns greater than one standard deviation above the mean are "High".

```
np.random.seed(42)
returns = np.random.randn(10)
```

```
def calc_ret(r):
    mu = r.mean()
    sigma = r.std(ddof=1)
    return np.select(
        condlist=[
            r<(mu-sigma),
            r<=(mu+sigma),
            r>(mu+sigma)
        ],
        choicelist=[
            'Low',
            'Medium',
            'High'
        ],
        default=''
    )
```

```
calc_ret(returns)
```

```
array(['Medium', 'Medium', 'Medium', 'High', 'Medium', 'Medium', 'High',
       'Medium', 'Low', 'Medium'], dtype='<U6')
```

McKinney Chapter 4 - Practice - Sec 04

```
import numpy as np
```

```
%precision 4
```

```
'%.4f'
```

Announcements

1. Keep signing up for project teams on Canvas > People > Projects
2. Claim a team, then ask to increase its limit if you want more than four teammates
3. Keep adding and voting for students choice topics

Five-Minute Review

NumPy is the foundation of scientific computing in Python! NumPy is also the foundation of pandas, so all these tools in this notebook are relevant to pandas, too!

Here are some random data for us to quickly review NumPy with. The `np.random.seed()` function makes our random number draws repeatable.

```
np.random.seed(42)
my_arr = np.random.randn(3, 5)
my_arr
```

```
array([[ 0.4967, -0.1383,  0.6477,  1.523 , -0.2342],
       [-0.2341,  1.5792,  0.7674, -0.4695,  0.5426],
       [-0.4634, -0.4657,  0.242 , -1.9133, -1.7249]])
```

Slicing works the same as with lists and tuples. As well, we can replace chained slices, like `[i][j][k]`, with a Matlab notation, like `[i. j. k]`

```
my_arr[2][3]
```

```
-1.9133
```

```
my_arr[2, 3]
```

```
-1.9133
```

What if we want the *first two rows?*

```
my_arr[:2]
```

```
array([[ 0.4967, -0.1383,  0.6477,  1.523 , -0.2342],
       [-0.2341,  1.5792,  0.7674, -0.4695,  0.5426]])
```

What if we want the *first two columns?*

```
my_arr[:, :2]
```

```
array([[ 0.4967, -0.1383],
       [-0.2341,  1.5792],
       [-0.4634, -0.4657]])
```

What about linear algebra in NumPy?

```
A = np.array([[1,2,3], [4,5,6]])
A
```

```
array([[1, 2, 3],
       [4, 5, 6]])
```

```
B = np.array([[1,2,3], [4,5,6]])
B
```

```
array([[1, 2, 3],
       [4, 5, 6]])
```

We can use the `.dot()` method to chain matrix multiplication.

```
A.dot(B.T)
```

```
array([[14, 32],
       [32, 77]])
```

The `@` operator is matrix multiplication with NumPy arrays.

```
A @ B.T
```

```
array([[14, 32],
       [32, 77]])
```

The `*` operator is elementwise multiplication with NumPy arrays.

```
A * B
```

```
array([[ 1,  4,  9],
       [16, 25, 36]])
```

The `np.where()` and `np.select()` functions are the NumPy equivalents of Excel's `if()`.

Say we want to replace values in `my_arr` greater than 0.5 with 0.5.

```
my_arr
```

```
array([[ 0.4967, -0.1383,  0.6477,  1.523 , -0.2342],
       [-0.2341,  1.5792,  0.7674, -0.4695,  0.5426],
       [-0.4634, -0.4657,  0.242 , -1.9133, -1.7249]])
```

```
np.where(my_arr>0.5, 0.5, my_arr)
```

```
array([[ 0.4967, -0.1383,  0.5   ,  0.5   , -0.2342],
       [-0.2341,  0.5   ,  0.5   , -0.4695,  0.5   ],
       [-0.4634, -0.4657,  0.242 , -1.9133, -1.7249]])
```

Or, we could use `np.minimum()` instead.

```
np.minimum(my_arr, 0.5)
```

```
array([[ 0.4967, -0.1383,  0.5   ,  0.5   , -0.2342],
       [-0.2341,  0.5   ,  0.5   , -0.4695,  0.5   ],
       [-0.4634, -0.4657,  0.242 , -1.9133, -1.7249]])
```

Now, say we want to replace values in `my_arr` greater than 0 with 0 and greater than 0.5 with 0.5.

```
np.where(
    my_arr>0.5, # first condition is most restrictive
    0.5, # if first condition True
    np.where( # otherwise
        my_arr>0, # second condition is less restrictive
        0, # if second condition True
        my_arr # otherwise
    )
)
```

```
array([[ 0.    , -0.1383,  0.5   ,  0.5   , -0.2342],
       [-0.2341,  0.5   ,  0.5   , -0.4695,  0.5   ],
       [-0.4634, -0.4657,  0.    , -1.9133, -1.7249]])
```

We have a better way to test more than one condition! `np.select()`!

```
np.select(
    condlist=[my_arr>0.5, my_arr>0],
    choicelist=[0.5, 0],
    default=my_arr
)
```

```
array([[ 0.    , -0.1383,  0.5   ,  0.5   , -0.2342],
       [-0.2341,  0.5   ,  0.5   , -0.4695,  0.5   ],
       [-0.4634, -0.4657,  0.    , -1.9133, -1.7249]])
```

Practice

Create a 1-dimensional array a1 that counts from 0 to 24 by 1.

```
a1 = np.arange(25)
a1
```

```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
       17, 18, 19, 20, 21, 22, 23, 24])
```

Create a 1-dimentional array a2 that counts from 0 to 24 by 3.

```
a2 = np.arange(0, 25, 3)
a2
```

```
array([ 0,  3,  6,  9, 12, 15, 18, 21, 24])
```

Create a 1-dimentional array a3 that counts from 0 to 100 by multiples of 3 or 5.

```
a3 = np.array([i for i in range(101) if (i%3==0) | (i%5==0)])
a3
```

```
array([  0,   3,   5,   6,   9,  10,  12,  15,  18,  20,  21,  24,  25,
        27,  30,  33,  35,  36,  39,  40,  42,  45,  48,  50,  51,  54,
        55,  57,  60,  63,  65,  66,  69,  70,  72,  75,  78,  80,  81,
        84,  85,  87,  90,  93,  95,  96,  99, 100])
```

```
zero_to_100 = np.arange(101)
a3_alt = zero_to_100[(zero_to_100%3==0) | (zero_to_100%5==0)]
a3_alt
```

```
array([  0,   3,   5,   6,   9,  10,  12,  15,  18,  20,  21,  24,  25,
        27,  30,  33,  35,  36,  39,  40,  42,  45,  48,  50,  51,  54,
        55,  57,  60,  63,  65,  66,  69,  70,  72,  75,  78,  80,  81,
        84,  85,  87,  90,  93,  95,  96,  99, 100])
```

```
(a3 == a3_alt).all()
```

```
np.True_
```

```
np.allclose(a3, a3_alt * 1.000_001)
```

```
True
```

Create a 1-dimensional array `a4` that contains the squares of the even integers through 100,000.

```
%timeit np.arange(0, 100_001, 2) ** 2
a4 = np.arange(0, 100_001, 2) ** 2
a4
```

```
30.1  s ± 6.69  s per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

```
array([          0,           4,          16, ...,  9999200016,
        9999600004, 10000000000], shape=(50001,))
```

```
%timeit a4_alt = np.array([i**2 for i in range(100_001) if i%2==0])
a4_alt = np.array([i**2 for i in range(100_001) if i%2==0])
a4_alt
```

```
22.1 ms ± 7.74 ms per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

```
array([          0,           4,          16, ...,  9999200016,
        9999600004, 10000000000], shape=(50001,))
```

```
(a4 == a4_alt).all()
```

```
np.True_
```

```
np.allclose(a4, a4_alt*1.0000001)
```

```
True
```

Write a function `calc_pv()` that mimic Excel's `PV` function.

Excel's present value function is: `=PV(rate, nper, pmt, [fv], [type])`

The present value of an annuity payment is: $PV_{pmt} = \frac{pmt}{rate} \times \left(1 - \frac{1}{(1+rate)^{nper}}\right)$

The present value of a lump sum is: $PV_{fv} = \frac{fv}{(1+rate)^{nper}}$

```
def calc_pv(rate, nper, pmt=None, fv=None, type=0):
    # or we could set the pmt and fv defaults to 0
    if pmt is None:
        pmt = 0
    if fv is None:
        fv = 0
    if type not in [0, 1]:
        raise ValueError(f'type must be 0 or 1; you provided {type}')

    pv_pmt = (pmt / rate) * (1 - (1 + rate)**(-nper))
    pv_fv = fv * (1 + rate)**(-nper)
    pv = pv_pmt + pv_fv

    if type == 1:
        pv *= 1 + rate

    return -pv
```

```
calc_pv(rate=0.1, nper=10, pmt=10, fv=100)
```

```
-100.0000
```

```
calc_pv(rate=0.1, nper=10, pmt=10, fv=100, type=1)
```

```
-110.0000
```

Write a function `calc_fv()` that mimic Excel's `FV` function.

Excel's future value function is: `=FV(rate, nper, pmt, [pv], [type])`

```
def calc_fv(rate, nper, pmt=None, pv=None, type='END'):
    if pmt is None:
        pmt = 0
    if pv is None:
        pv = 0
    if type not in ['BGN', 'END']:
        raise ValueError('type must be BGN or END')

    fv_pmt = (pmt / rate) * ((1 + rate)**nper - 1)
    fv_pv = pv * (1 + rate)**nper
    fv = fv_pmt + fv_pv

    if type == 'BGN':
        fv *= 1 + rate

    return -fv
```

```
calc_fv(rate=0.1, nper=10, pmt=10, pv=-100)
```

```
100.0000
```

The rule of 72!

```
calc_fv(rate=0.072, nper=10, pmt=0, pv=-100)
```

```
200.4231
```

Replace the negative values in `data` with -1 and positive values with +1.

```
np.random.seed(42)
data = np.random.randn(4, 4)

data
```

```
array([[ 0.4967, -0.1383,  0.6477,  1.523 ],
       [-0.2342, -0.2341,  1.5792,  0.7674],
       [-0.4695,  0.5426, -0.4634, -0.4657],
       [ 0.242 , -1.9133, -1.7249, -0.5623]])
```

We have at least three good solutions!

1. Slicing and brodacsting
2. `np.where()`
3. `np.select()`

First, here is the slicing and broadcasting solution.

```
data[data < 0] = -1
data[data > 0] = +1

data # here "1." and "-1." indicate that these values are floats
```

```
array([[ 1., -1.,  1.,  1.],
       [-1., -1.,  1.,  1.],
       [-1.,  1., -1., -1.],
       [ 1., -1., -1., -1.]])
```

Second, here is the `np.where()` solution. NumPy's `np.where()` function has the same logic as Excel's `if()` function! I will recreate `data` so we start from the same point.

```
np.random.seed(42)
data = np.random.randn(4, 4)

np.where(data < 0, -1, np.where(data > 0, +1, data))
```

```
array([[ 1., -1.,  1.,  1.],
       [-1., -1.,  1.,  1.],
       [-1.,  1., -1., -1.],
       [ 1., -1., -1., -1.]])
```

The nested `np.where()` function calls can get confusing! An option is to insert white space!

```
np.random.seed(42)
data = np.random.randn(4, 4)

np.where(
    data < 0, # condition
    -1, # result if True
    np.where(data > 0, +1, data) # result if False
)
```

```
array([[ 1., -1.,  1.,  1.],
       [-1., -1.,  1.,  1.],
       [-1.,  1., -1., -1.],
       [ 1., -1., -1., -1.]])
```

Third, here is the `np.select()` solution. NumPy's `np.select()` function lets us test *many* conditions! I will recreate `data` so we start from the same point.

```python
np.random.seed(42)
data = np.random.randn(4, 4)

np.select(
    condlist=[data<0, data>0],
    choicelist=[-1, +1],
    default=data
)
```

```
array([[ 1., -1.,  1.,  1.],
       [-1., -1.,  1.,  1.],
       [-1.,  1., -1., -1.],
       [ 1., -1., -1., -1.]])
```

Write a function `calc_n()` that calculates the number of payments that generate `x`% of the present value of a perpetuity.

The present value of a growing perpetuity is $PV = \frac{C_1}{r-g}$, and the present value of a growing annuity is $PV = \frac{C_1}{r-g}\left[1 - \left(\frac{1+g}{1+r}\right)^t\right]$.

We can use the growing annuity and perpetuity formulas to show: $x = \left[1 - \left(\frac{1+g}{1+r}\right)^t\right]$.

Then: $1 - x = \left(\frac{1+g}{1+r}\right)^t$.

Finally: $t = \frac{\log(1-x)}{\log\left(\frac{1+g}{1+r}\right)}$

We do not need to accept an argument `c1` because C_1 cancels out!

```python
def npmts(x, r, g):
    return np.log(1-x) / np.log((1 + g) / (1 + r))
```

```
npmts(0.5, 0.1, 0.05)
```

```
14.9000
```

Write a function that calculates the internal rate of return of a NumPy array of cash flows.

Let us pick a set of cash flows `c` where we know the internal rate of return! For the following `c`, the IRR is 10%.

```
c = np.array([-100, 110])
r = 0.1
```

First we need an function to calculate net present value (NPV) from `c` and `r`! The `npv()` function below uses NumPy arrays to calculate NPV as:

$$NPV = \sum_{t=0}^{T} \frac{c_t}{(1+r)^t}$$

```
def calc_npv(r, c):
    t = np.arange(len(c))
    return np.sum(c / (1 + r)**t)
```

```
calc_npv(r=r, c=c)
```

```
-0.0000
```

```
def calc_irr(c, guess=0, tol=1e-6, step=1e-6, max_iter=1_000, verbose=False):
    irr = guess
    for i in range(max_iter):
        npv = calc_npv(r=irr, c=c)
        if abs(npv) < tol:
            if verbose:
                print(f'IRR: {irr:0.4f}\nNPV: {npv:0.4f}\nIterations: {i+1:d}')
            return irr
        deriv = (calc_npv(r=irr+step, c=c) - npv) / step
        irr -= npv / deriv

    raise ValueError(f'NPV did not converge to zero after {i+1} iterations')
```

```
calc_irr(c)
```

```
0.1000
```

```
calc_irr(c=np.array([-100, 10, 10, 10, 10, 110]), verbose=True)
```

```
IRR: 0.1000
NPV: 0.0000
Iterations: 5
```

```
0.1000
```

```
# calc_irr(c=np.array([-2000, 1000, -500]))
```

Write a function `calc_returns()` that accepts *NumPy arrays* of prices and dividends and returns a *NumPy array* of returns.

```
prices = np.array([100, 150, 100, 50, 100, 150, 100, 150])
dividends = np.array([1, 1, 1, 1, 2, 2, 2, 2])
```

We want to slice our arrays to "lag" or "shift" them! For example, we slice the `prices` array to calculate capital gains as follows.

```
prices[1:] - prices[:-1]
```

```
array([ 50, -50, -50,  50,  50, -50,  50])
```

```
def calc_returns(p, d):
    return (p[1:] - p[:-1] + d[1:]) / p[:-1]
```

```
calc_returns(p=prices, d=dividends)
```

```
array([ 0.51  , -0.3267, -0.49  ,  1.04  ,  0.52  , -0.32  ,  0.52  ])
```

Rewrite the function `calc_returns()` as `calc_returns_2()` so it returns *NumPy arrays* of returns, capital gains yields, and dividend yields.

```
def calc_returns_2(p, d):
    cg = p[1:] / p[:-1] - 1
    dp = d[1:] / p[:-1]
    r = cg + dp
    return {'r': r, 'cg': cg, 'dp': dp}
```

```
calc_returns_2(p=prices, d=dividends)
```

```
{'r': array([ 0.51  , -0.3267, -0.49  ,  1.04  ,  0.52  , -0.32  ,  0.52  ]),
 'cg': array([ 0.5   , -0.3333, -0.5   ,  1.    ,  0.5   , -0.3333,  0.5   ]),
 'dp': array([0.01  , 0.0067, 0.01  , 0.04  , 0.02  , 0.0133, 0.02  ])}
```

Write a function `rescale()` to rescale and shift numbers so that they cover the range [0, 1]

Input: `np.array([18.5, 17.0, 18.0, 19.0, 18.0])`
Output: `np.array([0.75, 0.0, 0.5, 1.0, 0.5])`

```
x = np.array([18.5, 17.0, 18.0, 19.0, 18.0])
x
```

```
array([18.5, 17. , 18. , 19. , 18. ])
```

```
def rescale(x):
    return (x - x.min()) / (x.max() - x.min())
```

```
rescale(x=x)
```

```
array([0.75, 0.  , 0.5 , 1.  , 0.5 ])
```

Write a function `calc_portval()` that accepts a dictionary of prices and share holdings and returns the portfolio value

First convert your dictionary to a NumPy array with one column for prices and another for shares.

```python
data = {
    "AAPL": (150.25, 10),  # (price, shares)
    "GOOGL": (2750.00, 2),
    "MSFT": (300.75, 5)
}
```

```python
def calc_portval(data):
    x = np.array(list(data.values()))
    return x.prod(axis=1).sum()
```

```python
calc_portval(data=data)
```

```
8506.2500
```

Write functions `calc_var()` and `calc_std()` that calculate variance and standard deviation.

NumPy's `.var()` and `.std()` methods return *population* statistics (i.e., denominators of n). The pandas equivalents return *sample* statistics (denominators of $n-1$), which are more appropriate for financial data analysis where we have a sample instead of a population.

Your `calc_var()` and `calc_std()` functions should have a `sample` argument that is `True` by default so both functions return sample statistics by default.

```python
def calc_var(x, sample=True):
    sq_err = (x - x.mean()) ** 2
    den = len(x)
    if sample:
        den -= 1
    return sq_err.sum() / den
```

We can re-use `calc_var()` in our `calc_std()` function.

```python
def calc_std(x, sample=True):
    return calc_var(x=x, sample=sample) ** 0.5
```

```python
np.random.seed(42)
arr = np.random.randn(1_000_000)
arr
```

```
array([ 0.4967, -0.1383,  0.6477, ..., -0.113 ,  1.4691,  0.4764],
      shape=(1000000,))
```

```
calc_var(arr)
```

```
1.0004
```

```
calc_std(arr)
```

```
1.0002
```

```
arr.var(ddof=1) == calc_var(arr)
```

```
np.True_
```

```
arr.std(ddof=1) == calc_std(arr)
```

```
np.True_
```

Write a function `calc_ret()` to convert quantitative returns to qualitative returns

Returns within one standard deviation of the mean are "Medium". Returns less than one standard deviation below the mean are "Low", and returns greater than one standard deviation above the mean are "High".

```
np.random.seed(42)
returns = np.random.randn(10)
```

```
def calc_ret(r):
    mu = r.mean()
    sigma = r.std(ddof=1)
    return np.select(
        condlist=[
            r<(mu-sigma),
            r<=(mu+sigma),
            r>(mu+sigma)
        ],
        choicelist=[
```

```
            'Low',
            'Medium',
            'High'
        ],
        default=''
    )
```

```
calc_ret(returns)
```

```
array(['Medium', 'Medium', 'Medium', 'High', 'Medium', 'Medium', 'High',
       'Medium', 'Low', 'Medium'], dtype='<U6')
```

Week 4

McKinney Chapter 5 - Getting Started with pandas

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import yfinance as yf
```

```
%precision 4
pd.options.display.float_format = '{:.4f}'.format
# %config InlineBackend.figure_format = 'retina'
```

Introduction

Chapter 5 of McKinney (2022) discusses the fundamentals of pandas, which will be our main tool for the rest of the semester. pandas is an abbreviation of *pan*el *da*ta. Panel data contain observations on multiple entities (e.g., individuals, firms, or countries) over multiple time periods. Panel data combine cross-sectional data (i.e., data across entities at a single point in time) with time-series data (i.e., data over time for a single entity). Panel data are widely used in finance and economics to analyze trends, relationships, and behaviors over time and across entities. We will use pandas every day for the rest of the course!

> pandas will be a major tool of interest throughout much of the rest of the book. It contains data structures and data manipulation tools designed to make data cleaning and analysis fast and easy in Python. pandas is often used in tandem with numerical computing tools like NumPy and SciPy, analytical libraries like statsmodels and scikit-learn, and data visualization libraries like matplotlib. pandas adopts significant parts of NumPy's idiomatic style of array-based computing, especially array-based functions and a preference for data processing without for loops.
>
> While pandas adopts many coding idioms from NumPy, the biggest difference is that pandas is designed for working with tabular or heterogeneous data. NumPy, by contrast, is best suited for working with homogeneous numerical array data.

Note: Indented block quotes are from McKinney (2022) unless otherwise indicated. The section numbers here differ from McKinney (2022) because we will only discuss some topics.

Introduction to pandas Data Structures

> To get started with pandas, you will need to get comfortable with its two workhorse data structures: Series and DataFrame. While they are not a universal solution for every problem, they provide a solid, easy-to-use basis for most applications.

Series

> A Series is a one-dimensional array-like object containing a sequence of values (of similar types to NumPy types) and an associated array of data labels, called its index. The simplest Series is formed from only an array of data.

The early examples use integer and string labels, but date-time labels are most useful.

```
obj = pd.Series([4, 7, -5, 3])
obj
```

```
0    4
1    7
2   -5
3    3
dtype: int64
```

Contrast `obj` with its NumPy array equivalent:

```
np.array([4, 7, -5, 3])
```

```
array([ 4,  7, -5,  3])
```

```
obj.values
```

```
array([ 4,  7, -5,  3])
```

```python
obj.index  # similar to range(4)
```

```
RangeIndex(start=0, stop=4, step=1)
```

We did not explicitly create an index for `obj`, so `obj` has an integer index that starts at 0. We can explicitly create an index with the `index=` argument.

```python
obj2 = pd.Series([4, 7, -5, 3], index=['d', 'b', 'a', 'c'])
obj2
```

```
d    4
b    7
a   -5
c    3
dtype: int64
```

```python
obj2.index
```

```
Index(['d', 'b', 'a', 'c'], dtype='object')
```

```python
obj2['a']
```

```
np.int64(-5)
```

```python
obj2.loc['a']
```

```
np.int64(-5)
```

```python
obj2.iloc[2]
```

```
np.int64(-5)
```

```python
obj2['d'] = 6
obj2
```

```
d    6
b    7
a   -5
c    3
dtype: int64
```

```
obj2[['c', 'a', 'd']]
```

```
c    3
a   -5
d    6
dtype: int64
```

```
obj2.loc[['c', 'a', 'd']]
```

```
c    3
a   -5
d    6
dtype: int64
```

A pandas series is like a NumPy array, and we can use Boolean filters and perform vectorized mathematical operations.

```
obj2 > 0
```

```
d     True
b     True
a    False
c     True
dtype: bool
```

```
obj2[obj2 > 0]
```

```
d    6
b    7
c    3
dtype: int64
```

```
obj2.loc[obj2 > 0]
```

```
d    6
b    7
c    3
dtype: int64
```

```
obj2 * 2
```

```
d    12
b    14
a   -10
c     6
dtype: int64
```

We can also create a pandas series from a dictionary. The dictionary keys become the series index.

```
sdata = {'Ohio': 35000, 'Texas': 71000, 'Oregon': 16000, 'Utah': 5000}
obj3 = pd.Series(sdata)
obj3
```

```
Ohio      35000
Texas     71000
Oregon    16000
Utah       5000
dtype: int64
```

If we also specify an index with list `states`, pandas will:

1. Respect the index order
2. Keep California because it was in the index
3. Drop Utah because it was not in the index

```
states = ['California', 'Ohio', 'Oregon', 'Texas']
obj4 = pd.Series(sdata, index=states)
obj4
```

```
California           NaN
Ohio          35000.0000
Oregon        16000.0000
Texas         71000.0000
dtype: float64
```

When we perform mathematical operations, pandas aligns series by their indexes. Here `NaN` is "not a number", indicating missing values. `NaN` is a float, so the data type switches from int64 to float64.

```
obj3 + obj4
```

```
California            NaN
Ohio           70000.0000
Oregon         32000.0000
Texas         142000.0000
Utah                  NaN
dtype: float64
```

DataFrame

A pandas data frame is like a worksheet in an Excel workbook with row labels and column names that provide fast indexing.

> A DataFrame represents a rectangular table of data and contains an ordered collection of columns, each of which can be a different value type (numeric, string, boolean, etc.). The DataFrame has both a row and column index; it can be thought of as a dict of Series all sharing the same index. Under the hood, the data is stored as one or more two-dimensional blocks rather than a list, dict, or some other collection of one-dimensional arrays. The exact details of DataFrame's internals are outside the scope of this book.
>
> There are many ways to construct a DataFrame, though one of the most common is from a dict of equal-length lists or NumPy arrays:

```
data = {
    'state': ['Ohio', 'Ohio', 'Ohio', 'Nevada', 'Nevada', 'Nevada'],
    'year': [2000, 2001, 2002, 2001, 2002, 2003],
    'pop': [1.5, 1.7, 3.6, 2.4, 2.9, 3.2]
}
frame = pd.DataFrame(data)

frame
```

	state	year	pop
0	Ohio	2000	1.5000
1	Ohio	2001	1.7000
2	Ohio	2002	3.6000
3	Nevada	2001	2.4000
4	Nevada	2002	2.9000
5	Nevada	2003	3.2000

We did not specify an index, so `frame` has the default index of integers starting at 0.

```
frame2 = pd.DataFrame(
    data,
    columns=['year', 'state', 'pop', 'debt'],
    index=['one', 'two', 'three', 'four', 'five', 'six']
)

frame2
```

	year	state	pop	debt
one	2000	Ohio	1.5000	NaN
two	2001	Ohio	1.7000	NaN
three	2002	Ohio	3.6000	NaN
four	2001	Nevada	2.4000	NaN
five	2002	Nevada	2.9000	NaN
six	2003	Nevada	3.2000	NaN

If we extract one column with `df.column` or `df['column']`, we get a series. We can use the `df.colname` or `df['colname']` syntax to *extract* a column from a data frame as a series. ***However, we must use the `df['colname']` syntax to add a column to a data frame.*** Also, we must use the `df['colname']` syntax to extract or add a column whose name contains whitespace.

```
frame2['state']
```

```
one        Ohio
two        Ohio
three      Ohio
four     Nevada
five     Nevada
```

```
six      Nevada
Name: state, dtype: object
```

```
frame2.state
```

```
one        Ohio
two        Ohio
three      Ohio
four     Nevada
five     Nevada
six      Nevada
Name: state, dtype: object
```

Data frames have two dimensions, so we must slice data frames more precisely than series.

1. The `.loc[]` method slices by row labels and column names
2. The `.iloc[]` method slices by *integer* row and label indexes

```
frame2.loc['three']
```

```
year       2002
state      Ohio
pop      3.6000
debt        NaN
Name: three, dtype: object
```

```
frame2.iloc[2]
```

```
year       2002
state      Ohio
pop      3.6000
debt        NaN
Name: three, dtype: object
```

We can use NumPy's `[row, column]` syntax within `.loc[]` and `.iloc[]`.

```
frame2.loc['three', 'state'] # row, column
```

```
'Ohio'
```

```
frame2.iloc[2, 1] # row, column
```

```
'Ohio'
```

```
frame2.loc['three', ['state', 'pop']] # row, column
```

```
state     Ohio
pop     3.6000
Name: three, dtype: object
```

```
frame2.iloc[2, [1, 2]] # row, column
```

```
state     Ohio
pop     3.6000
Name: three, dtype: object
```

We can assign either scalars or arrays to data frame columns.

1. Scalars will broadcast to every row in the data frame
2. Arrays must have the same length as the column

```
frame2['debt'] = 16.5
frame2
```

	year	state	pop	debt
one	2000	Ohio	1.5000	16.5000
two	2001	Ohio	1.7000	16.5000
three	2002	Ohio	3.6000	16.5000
four	2001	Nevada	2.4000	16.5000
five	2002	Nevada	2.9000	16.5000
six	2003	Nevada	3.2000	16.5000

```
frame2['debt'] = np.arange(6.)
frame2
```

	year	state	pop	debt
one	2000	Ohio	1.5000	0.0000
two	2001	Ohio	1.7000	1.0000
three	2002	Ohio	3.6000	2.0000
four	2001	Nevada	2.4000	3.0000
five	2002	Nevada	2.9000	4.0000
six	2003	Nevada	3.2000	5.0000

If we assign a series to a data frame column, pandas will use the index to align it with the data frame. Data frame rows not in the series will be `NaN`.

```
val = pd.Series([-1.2, -1.5, -1.7], index=['two', 'four', 'five'])
val
```

```
two    -1.2000
four   -1.5000
five   -1.7000
dtype: float64
```

```
frame2['debt'] = val
frame2
```

	year	state	pop	debt
one	2000	Ohio	1.5000	NaN
two	2001	Ohio	1.7000	-1.2000
three	2002	Ohio	3.6000	NaN
four	2001	Nevada	2.4000	-1.5000
five	2002	Nevada	2.9000	-1.7000
six	2003	Nevada	3.2000	NaN

We can add columns to our data frame, then delete them with `del`.

```
frame2['eastern'] = (frame2.state == 'Ohio')
frame2
```

	year	state	pop	debt	eastern
one	2000	Ohio	1.5000	NaN	True

	year	state	pop	debt	eastern
two	2001	Ohio	1.7000	-1.2000	True
three	2002	Ohio	3.6000	NaN	True
four	2001	Nevada	2.4000	-1.5000	False
five	2002	Nevada	2.9000	-1.7000	False
six	2003	Nevada	3.2000	NaN	False

```
del frame2['eastern']
frame2
```

	year	state	pop	debt
one	2000	Ohio	1.5000	NaN
two	2001	Ohio	1.7000	-1.2000
three	2002	Ohio	3.6000	NaN
four	2001	Nevada	2.4000	-1.5000
five	2002	Nevada	2.9000	-1.7000
six	2003	Nevada	3.2000	NaN

Index Objects

```
obj = pd.Series(range(3), index=['a', 'b', 'c'])
index = obj.index
index
```

```
Index(['a', 'b', 'c'], dtype='object')
```

Index objects are immutable!

```
# # TypeError: Index does not support mutable operations
# index[1] = 'd'
```

Indexes can contain duplicates, so an index does not guarantee that our data are duplicate-free.

```
dup_labels = pd.Index(['foo', 'foo', 'bar', 'bar'])
dup_labels
```

```
Index(['foo', 'foo', 'bar', 'bar'], dtype='object')
```

Essential Functionality

This section provides the most common pandas operations. It is difficult to provide an exhaustive reference, but this section introduces the most common operations.

Indexing, Selection, and Filtering

Indexing, selecting, and filtering will be among our most-used pandas features.

```
obj = pd.Series(np.arange(4.), index=['a', 'b', 'c', 'd'])
obj
```

```
a   0.0000
b   1.0000
c   2.0000
d   3.0000
dtype: float64
```

```
obj['b']
```

```
1.0000
```

```
obj.loc['b']
```

```
1.0000
```

```
obj.iloc[1]
```

```
1.0000
```

```
obj.iloc[1:3]
```

```
b   1.0000
c   2.0000
dtype: float64
```

When we slice with labels, the left and right endpoints are inclusive.

```
obj.loc['b':'c']
```

```
b    1.0000
c    2.0000
dtype: float64
```

```
obj.loc['b':'c'] = 5
obj
```

```
a    0.0000
b    5.0000
c    5.0000
d    3.0000
dtype: float64
```

```
data = pd.DataFrame(
    data=np.arange(16).reshape((4, 4)),
    index=['Ohio', 'Colorado', 'Utah', 'New York'],
    columns=['one', 'two', 'three', 'four']
)

data
```

	one	two	three	four
Ohio	0	1	2	3
Colorado	4	5	6	7
Utah	8	9	10	11
New York	12	13	14	15

Indexing one column returns a series.

```
data['two']
```

```
Ohio         1
Colorado     5
Utah         9
New York    13
Name: two, dtype: int64
```

Indexing columns with a list returns a data frame.

```
data[['three']]
```

	three
Ohio	2
Colorado	6
Utah	10
New York	14

```
data[['three', 'one']]
```

	three	one
Ohio	2	0
Colorado	6	4
Utah	10	8
New York	14	12

Table 5-4 summarizes data frame indexing and slicing options:

- `df[val]`: Select single column or sequence of columns from the DataFrame; special case conveniences: boolean array (filter rows), slice (slice rows), or boolean DataFrame (set values based on some criterion)
- `df.loc[val]`: Selects single row or subset of rows from the DataFrame by label
- `df.loc[:, val]`: Selects single column or subset of columns by label
- `df.loc[val1, val2]`: Select both rows and columns by label
- `df.iloc[where]`: Selects single row or subset of rows from the DataFrame by integer position
- `df.iloc[:, where]`: Selects single column or subset of columns by integer position
- `df.iloc[where_i, where_j]`: Select both rows and columns by integer position
- `df.at[label_i, label_j]`: Select a single scalar value by row and column label
- `df.iat[i, j]`: Select a single scalar value by row and column position (integers) reindex method Select either rows or columns by labels
- `get_value`, `set_value` methods: Select single value by row and column label

pandas is powerful and these options can be overwhelming! We will typically use `df[val]` to select columns (here `val` is either a string or list of strings), `df.loc[val]` to select rows (here `val` is a row label), and `df.loc[val1, val2]` to select both rows and columns. The other options add flexibility, and we may occasionally use them. However, our data will be large enough that counting row and column number will be tedious, making `.iloc[]` impractical.

Arithmetic and Data Alignment

An important pandas feature for some applications is the behavior of arithmetic between objects with different indexes. When you are adding together objects, if any index pairs are not the same, the respective index in the result will be the union of the index pairs. For users with database experience, this is similar to an automatic outer join on the index labels.

```
s1 = pd.Series(
    data=[7.3, -2.5, 3.4, 1.5],
    index=['a', 'c', 'd', 'e']
)
s2 = pd.Series(
    data=[-2.1, 3.6, -1.5, 4, 3.1],
    index=['a', 'c', 'e', 'f', 'g']
)
```

```
s1
```

```
a    7.3000
c   -2.5000
d    3.4000
e    1.5000
dtype: float64
```

```
s2
```

```
a   -2.1000
c    3.6000
e   -1.5000
f    4.0000
g    3.1000
dtype: float64
```

```
s1 + s2
```

```
a   5.2000
c   1.1000
d      NaN
e   0.0000
f      NaN
```

```
g        NaN
dtype: float64
```

```
df1 = pd.DataFrame(
    data=np.arange(9.).reshape((3, 3)),
    columns=list('bcd'),
    index=['Ohio', 'Texas', 'Colorado']
)
df2 = pd.DataFrame(
    data=np.arange(12.).reshape((4, 3)),
    columns=list('bde'),
    index=['Utah', 'Ohio', 'Texas', 'Oregon']
)
```

```
df1
```

	b	c	d
Ohio	0.0000	1.0000	2.0000
Texas	3.0000	4.0000	5.0000
Colorado	6.0000	7.0000	8.0000

```
df2
```

	b	d	e
Utah	0.0000	1.0000	2.0000
Ohio	3.0000	4.0000	5.0000
Texas	6.0000	7.0000	8.0000
Oregon	9.0000	10.0000	11.0000

```
df1 + df2
```

	b	c	d	e
Colorado	NaN	NaN	NaN	NaN
Ohio	3.0000	NaN	6.0000	NaN
Oregon	NaN	NaN	NaN	NaN
Texas	9.0000	NaN	12.0000	NaN
Utah	NaN	NaN	NaN	NaN

Always check your output!

Arithmetic methods with fill values

```
df1 = pd.DataFrame(
    data=np.arange(12.).reshape((3, 4)),
    columns=list('abcd')
)
df2 = pd.DataFrame(
    data=np.arange(20.).reshape((4, 5)),
    columns=list('abcde')
)
df2.loc[1, 'b'] = np.nan
```

```
df1
```

	a	b	c	d
0	0.0000	1.0000	2.0000	3.0000
1	4.0000	5.0000	6.0000	7.0000
2	8.0000	9.0000	10.0000	11.0000

```
df2
```

	a	b	c	d	e
0	0.0000	1.0000	2.0000	3.0000	4.0000
1	5.0000	NaN	7.0000	8.0000	9.0000
2	10.0000	11.0000	12.0000	13.0000	14.0000
3	15.0000	16.0000	17.0000	18.0000	19.0000

```
df1 + df2
```

	a	b	c	d	e
0	0.0000	2.0000	4.0000	6.0000	NaN
1	9.0000	NaN	13.0000	15.0000	NaN
2	18.0000	20.0000	22.0000	24.0000	NaN
3	NaN	NaN	NaN	NaN	NaN

We can specify a fill value for `NaN` values. pandas fills would-be `NaN` values in each data frame *before* the arithmetic operation.

```
df1.add(df2, fill_value=0)
```

	a	b	c	d	e
0	0.0000	2.0000	4.0000	6.0000	4.0000
1	9.0000	5.0000	13.0000	15.0000	9.0000
2	18.0000	20.0000	22.0000	24.0000	14.0000
3	15.0000	16.0000	17.0000	18.0000	19.0000

Operations between DataFrame and Series

```
arr = np.arange(12.).reshape((3, 4))
arr
```

```
array([[ 0.,  1.,  2.,  3.],
       [ 4.,  5.,  6.,  7.],
       [ 8.,  9., 10., 11.]])
```

```
arr[0]
```

```
array([0., 1., 2., 3.])
```

```
arr - arr[0]
```

```
array([[0., 0., 0., 0.],
       [4., 4., 4., 4.],
       [8., 8., 8., 8.]])
```

Arithmetic operations between series and data frames behave the same as in the example above.

```
frame = pd.DataFrame(
    data=np.arange(12.).reshape((4, 3)),
    columns=list('bde'),
    index=['Utah', 'Ohio', 'Texas', 'Oregon']
)

series = frame.iloc[0]
```

```
frame
```

	b	d	e
Utah	0.0000	1.0000	2.0000
Ohio	3.0000	4.0000	5.0000
Texas	6.0000	7.0000	8.0000
Oregon	9.0000	10.0000	11.0000

```
series
```

```
b   0.0000
d   1.0000
e   2.0000
Name: Utah, dtype: float64
```

```
frame - series
```

	b	d	e
Utah	0.0000	0.0000	0.0000
Ohio	3.0000	3.0000	3.0000
Texas	6.0000	6.0000	6.0000
Oregon	9.0000	9.0000	9.0000

```
series2 = pd.Series(data=range(3), index=['b', 'e', 'f'])
```

```
frame
```

	b	d	e
Utah	0.0000	1.0000	2.0000
Ohio	3.0000	4.0000	5.0000
Texas	6.0000	7.0000	8.0000
Oregon	9.0000	10.0000	11.0000

```
series2
```

```
b    0
e    1
f    2
dtype: int64
```

```
frame + series2
```

	b	d	e	f
Utah	0.0000	NaN	3.0000	NaN
Ohio	3.0000	NaN	6.0000	NaN
Texas	6.0000	NaN	9.0000	NaN
Oregon	9.0000	NaN	12.0000	NaN

pandas has a `.sub()` method that lets us chain operations, but we might need the `axis` argument to get the result we want!

```
series3 = frame['d']
```

```
frame
```

	b	d	e
Utah	0.0000	1.0000	2.0000
Ohio	3.0000	4.0000	5.0000
Texas	6.0000	7.0000	8.0000
Oregon	9.0000	10.0000	11.0000

```
series3
```

```
Utah       1.0000
Ohio       4.0000
Texas      7.0000
Oregon    10.0000
Name: d, dtype: float64
```

```
frame - series
```

	b	d	e
Utah	0.0000	0.0000	0.0000
Ohio	3.0000	3.0000	3.0000
Texas	6.0000	6.0000	6.0000
Oregon	9.0000	9.0000	9.0000

```
frame.sub(series3, axis=0)
```

	b	d	e
Utah	-1.0000	0.0000	1.0000
Ohio	-1.0000	0.0000	1.0000
Texas	-1.0000	0.0000	1.0000
Oregon	-1.0000	0.0000	1.0000

Function Application and Mapping

```
np.random.seed(42)
frame = pd.DataFrame(
    data=np.random.randn(4, 3),
    columns=list('bde'),
    index=['Utah', 'Ohio', 'Texas', 'Oregon']
)

frame
```

	b	d	e
Utah	0.4967	-0.1383	0.6477
Ohio	1.5230	-0.2342	-0.2341
Texas	1.5792	0.7674	-0.4695
Oregon	0.5426	-0.4634	-0.4657

```
frame.abs()
```

	b	d	e
Utah	0.4967	0.1383	0.6477
Ohio	1.5230	0.2342	0.2341
Texas	1.5792	0.7674	0.4695
Oregon	0.5426	0.4634	0.4657

Another frequent operation is applying a function on one-dimensional arrays to each column or row. DataFrame's apply method does exactly this:

```
frame.apply(lambda x: x.max() - x.min()) # implied axis=0
```

```
b   1.0825
d   1.2309
e   1.1172
dtype: float64
```

```
frame.apply(lambda x: x.max() - x.min(), axis=1) # explicit axis=1
```

```
Utah      0.7860
Ohio      1.7572
Texas     2.0487
Oregon    1.0083
dtype: float64
```

However, under the hood, the `.apply()` method is a `for` loop and slower than built-in methods.

```
%timeit frame['e'].abs()
```

```
44.9  s ± 10.7  s per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
```

```
%timeit frame['e'].apply(np.abs)
```

```
96.9  s ± 16.6  s per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

Summarizing and Computing Descriptive Statistics

```
df = pd.DataFrame(
    [[1.4, np.nan], [7.1, -4.5], [np.nan, np.nan], [0.75, -1.3]],
    index=['a', 'b', 'c', 'd'],
    columns=['one', 'two']
)

df
```

	one	two
a	1.4000	NaN
b	7.1000	-4.5000
c	NaN	NaN
d	0.7500	-1.3000

```
df.sum() # implied axis=0
```

```
one    9.2500
two   -5.8000
dtype: float64
```

```
df.sum(axis=1)
```

```
a    1.4000
b    2.6000
c    0.0000
d   -0.5500
dtype: float64
```

```
df.mean(axis=1, skipna=False)
```

```
a       NaN
b    1.3000
c       NaN
d   -0.2750
dtype: float64
```

The `.idxmax()` method returns the label for the maximum observation.

```
df
```

	one	two
a	1.4000	NaN
b	7.1000	-4.5000
c	NaN	NaN
d	0.7500	-1.3000

```
df.idxmax()
```

```
one      b
two      d
dtype: object
```

The `.describe()` returns summary statistics for each numerical column in a data frame.

```
df.describe()
```

	one	two
count	3.0000	2.0000
mean	3.0833	-2.9000
std	3.4937	2.2627
min	0.7500	-4.5000
25%	1.0750	-3.7000
50%	1.4000	-2.9000
75%	4.2500	-2.1000
max	7.1000	-1.3000

For non-numerical data, `.describe()` returns alternative summary statistics.

```
obj = pd.Series(['a', 'a', 'b', 'c'] * 4)
obj.describe()
```

```
count     16
unique     3
top        a
freq       8
dtype: object
```

Correlation and Covariance

> **Note**
>
> Starting with version 0.2.51, the `yfinance` package changed the default behavior of the `auto_adjust` argument from `False` to `True`. By default, the `yf.download()` function now returns adjusted prices, without including the `Adj Close` column.
> We prefer to work with raw data from Yahoo! Finance and explicitly calculate returns using the `Adj Close` column. Therefore, we will set `auto_adjust=False` in our `yf.download()` calls. See the yfinance changelog for release version 0.2.51.
> Also, I will use the `progress=False` argument to improve the readability of the PDF and website I render from these notebooks.

```
data = yf.download(tickers='AAPL IBM MSFT GOOG', auto_adjust=False, progress=False)
```

```
data['Adj Close'].tail()
```

Ticker Date	AAPL	GOOG	IBM	MSFT
2025-02-24	247.1000	181.1900	261.8700	404.0000
2025-02-25	247.0400	177.3700	257.7500	397.9000
2025-02-26	240.3600	174.7000	255.8400	399.7300
2025-02-27	237.3000	170.2100	253.2300	392.5300
2025-02-28	241.8400	172.2200	252.4400	396.9900

Data frame `data` contains daily prices and volume for AAPL, IBM, MSFT, and GOOG. The `Adj Close` columns are reverse-engineered daily closing prices that account for dividends and stock splits (and reverse splits). As a result, the `.pct_change()` of `Adj Close` correctly considers dividends and price changes, so $r_t = \frac{(P_t+D_t)-P_{t-1}}{P_{t-1}} = \frac{\text{Adj Close}_t - \text{Adj Close}_{t-1}}{\text{Adj Close}_{t-1}}$.

```
returns = data['Adj Close'].pct_change().dropna()
returns
```

Ticker Date	AAPL	GOOG	IBM	MSFT
2004-08-20	0.0029	0.0794	0.0042	0.0029
2004-08-23	0.0091	0.0101	-0.0070	0.0044
2004-08-24	0.0280	-0.0414	0.0007	0.0000

Ticker Date	AAPL	GOOG	IBM	MSFT
2004-08-25	0.0344	0.0108	0.0043	0.0114
2004-08-26	0.0487	0.0180	-0.0045	-0.0040
...	...	...	...	...
2025-02-24	0.0063	-0.0021	0.0015	-0.0103
2025-02-25	-0.0002	-0.0211	-0.0157	-0.0151
2025-02-26	-0.0270	-0.0151	-0.0074	0.0046
2025-02-27	-0.0127	-0.0257	-0.0102	-0.0180
2025-02-28	0.0191	0.0118	-0.0031	0.0114

We multiply by 252 to annualize mean daily returns because means grow linearly with time and there are (about) 252 trading days per year.

```
returns.mean().mul(252)
```

```
Ticker
AAPL    0.3579
GOOG    0.2533
IBM     0.1112
MSFT    0.1905
dtype: float64
```

We multiply by $\sqrt{252}$ to annualize the volatility of daily returns because standard deviation is the square root of variance, variances grow linearly with time, and there are (about) 252 trading days per year. Ivo Welch explains this calculation at the bottom of Page 7 of Chapter 8 his free corporate finance textbook.

```
returns.std().mul(np.sqrt(252))
```

```
Ticker
AAPL    0.3234
GOOG    0.3061
IBM     0.2281
MSFT    0.2693
dtype: float64
```

We can calculate pairwise correlations.

```
returns['MSFT'].corr(returns['IBM'])
```

```
0.4733
```

We can also calculate correlation matrices.

```
returns.corr()
```

Ticker Ticker	AAPL	GOOG	IBM	MSFT
AAPL	1.0000	0.5116	0.4121	0.5209
GOOG	0.5116	1.0000	0.3807	0.5603
IBM	0.4121	0.3807	1.0000	0.4733
MSFT	0.5209	0.5603	0.4733	1.0000

```
returns.corr().loc['MSFT', 'IBM']
```

```
0.4733
```

```
np.allclose(
    a=returns['MSFT'].corr(returns['IBM']),
    b=returns.corr().loc['MSFT', 'IBM']
)
```

```
True
```

McKinney Chapter 5 - Practice - Blank

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import yfinance as yf
```

```
%precision 4
pd.options.display.float_format = '{:.4f}'.format
# %config InlineBackend.figure_format = 'retina'
```

Announcements

Five-Minute Review

Practice

What are the mean daily returns for these four stocks?

```
tickers = 'AAPL IBM MSFT GOOG'
```

What are the standard deviations of daily returns for these four stocks?

What are the *annualized* means and standard deviations of daily returns for these four stocks?

Plot *annualized* means versus standard deviations of daily returns for these four stocks

Repeat the previous calculations and plot for the stocks in the Dow-Jones Industrial Index (DJIA)

We can find the current DJIA stocks on Wikipedia. We must download new data, into `tickers_2`, `data_2`, and `returns_2`.

Calculate total returns for the stocks in the DJIA

Plot the distribution of total returns for the stocks in the DJIA

Which stocks have the minimum and maximum total returns?

Plot the cumulative returns for the stocks in the DJIA

Repeat the plot above with only the minimum and maximum total returns

McKinney Chapter 5 - Practice - Sec 02

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import yfinance as yf
```

```
%precision 4
pd.options.display.float_format = '{:.4f}'.format
# %config InlineBackend.figure_format = 'retina'
```

Announcements

1. Please keep forming groups on Canvas > People > Projects. If you want a group with more than four students, please fill a group with four students, then email with the group number and size.
2. Please keep proposing and voting for students' choice topics here.

Five-Minute Review

The pandas package makes it easy to manipulate panel data and we will use it all semester. Its name is an abbreviation of *pan*el *da*ta:

> In statistics and econometrics, panel data and longitudinal data[1][2] are both multi-dimensional data involving measurements over time. Panel data is a subset of longitudinal data where observations are for the same subjects each time.

We will download some financial data from Yahoo! Finance to cover three important tools in pandas.

First, we can use the yfinance package to easily download stock data from Yahoo! Finance.

> **Note**
>
> Starting with version 0.2.51, the `yfinance` package changed the default behavior of the `auto_adjust` argument from `False` to `True`. By default, the `yf.download()` function now returns adjusted prices, without including the `Adj Close` column.
> We prefer to work with raw data from Yahoo! Finance and explicitly calculate returns using the `Adj Close` column. Therefore, we will set `auto_adjust=False` in our `yf.download()` calls. See the yfinance changelog for release version 0.2.51.
> Also, I will use the `progress=False` argument to improve the readability of the PDF and website I render from these notebooks.

```
df0 = yf.download(tickers='AAPL MSFT', auto_adjust=False, progress=False)
```

```
df0
```

Price	Adj Close		Close		High		Low		Open
Ticker	AAPL	MSFT	AAPL	MSFT	AAPL	MSFT	AAPL	MSFT	AAPL
Date									
1980-12-12	0.0987	NaN	0.1283	NaN	0.1289	NaN	0.1283	NaN	0.1283
1980-12-15	0.0936	NaN	0.1217	NaN	0.1222	NaN	0.1217	NaN	0.1222
1980-12-16	0.0867	NaN	0.1127	NaN	0.1133	NaN	0.1127	NaN	0.1133
1980-12-17	0.0889	NaN	0.1155	NaN	0.1161	NaN	0.1155	NaN	0.1155
1980-12-18	0.0914	NaN	0.1189	NaN	0.1194	NaN	0.1189	NaN	0.1189
...	...	...	...	...	...	...	...	...	...
2025-02-24	247.1000	404.0000	247.1000	404.0000	248.8600	409.3700	244.4200	399.3200	244.9300
2025-02-25	247.0400	397.9000	247.0400	397.9000	250.0000	401.9200	244.9100	396.7000	248.0000
2025-02-26	240.3600	399.7300	240.3600	399.7300	244.9800	403.6000	239.1300	394.2500	244.3300
2025-02-27	237.3000	392.5300	237.3000	392.5300	242.4600	405.7400	237.0600	392.1700	239.4100
2025-02-28	241.8400	396.9900	241.8400	396.9900	242.0900	397.6300	230.2000	386.5700	236.9500

Second, we can slice rows and columns two ways: by integer locations with `.iloc[]` and by labels with `.loc[]`

```
df0.iloc[:6, :6]
```

Price	Adj Close		Close		High	
Ticker	AAPL	MSFT	AAPL	MSFT	AAPL	MSFT
Date						
1980-12-12	0.0987	NaN	0.1283	NaN	0.1289	NaN
1980-12-15	0.0936	NaN	0.1217	NaN	0.1222	NaN
1980-12-16	0.0867	NaN	0.1127	NaN	0.1133	NaN
1980-12-17	0.0889	NaN	0.1155	NaN	0.1161	NaN
1980-12-18	0.0914	NaN	0.1189	NaN	0.1194	NaN
1980-12-19	0.0970	NaN	0.1261	NaN	0.1267	NaN

```
df0.loc[:'1980-12-19', :'High']
```

Price	Adj Close		Close		High	
Ticker	AAPL	MSFT	AAPL	MSFT	AAPL	MSFT
Date						
1980-12-12	0.0987	NaN	0.1283	NaN	0.1289	NaN
1980-12-15	0.0936	NaN	0.1217	NaN	0.1222	NaN
1980-12-16	0.0867	NaN	0.1127	NaN	0.1133	NaN
1980-12-17	0.0889	NaN	0.1155	NaN	0.1161	NaN
1980-12-18	0.0914	NaN	0.1189	NaN	0.1194	NaN
1980-12-19	0.0970	NaN	0.1261	NaN	0.1267	NaN

Note

To slice a DataFrame:

- Use `['Name']` to select specific columns by their names.
- Use `.loc[]` to slice rows, or rows and columns together, with labels or conditional expressions.

```
df0['High']
```

Ticker	AAPL	MSFT
Date		
1980-12-12	0.1289	NaN
1980-12-15	0.1222	NaN
1980-12-16	0.1133	NaN

Ticker Date	AAPL	MSFT
1980-12-17	0.1161	NaN
1980-12-18	0.1194	NaN
...	...	...
2025-02-24	248.8600	409.3700
2025-02-25	250.0000	401.9200
2025-02-26	244.9800	403.6000
2025-02-27	242.4600	405.7400
2025-02-28	242.0900	397.6300

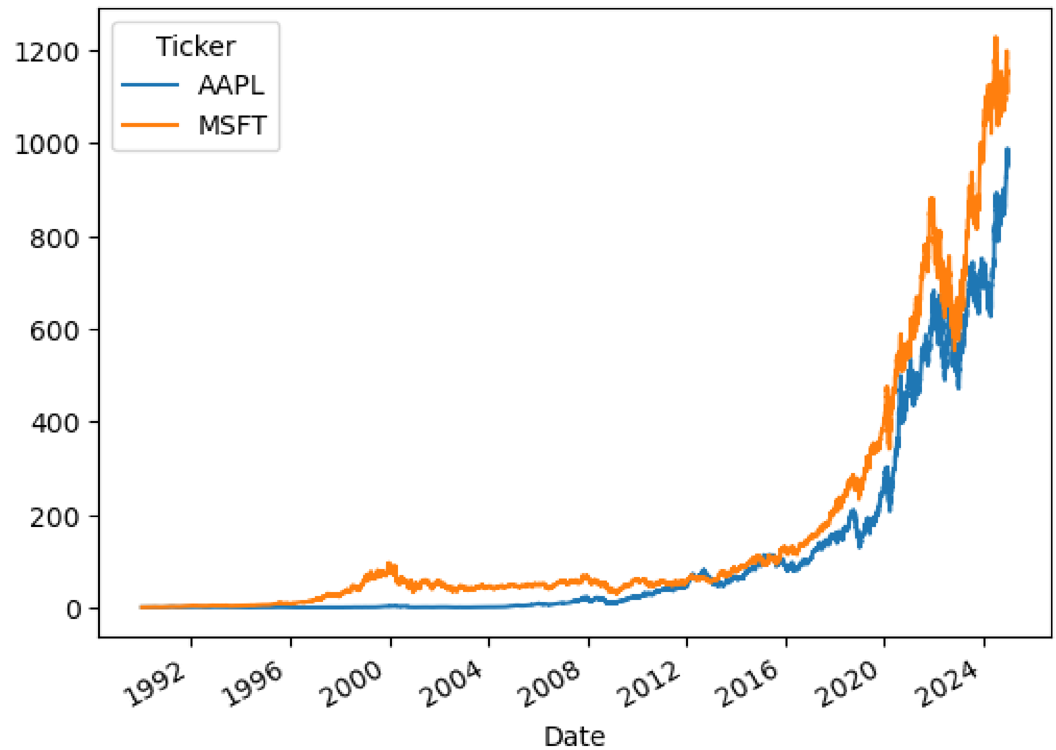
```
# # KeyError: '1980-12-12'
# df0['1980-12-12']
```

Note, if we use string labels, like dates and words, pandas includes left and right edges! This string label behavior differs from the integer location behavior everywhere else in Python. However, it is easy to figure out the sequence of integer locations. It is difficult to figure our the sequence of string labels.

Third, there many methods we can apply to pandas objects (and chain)! At this point in the course, our most common methods will be:

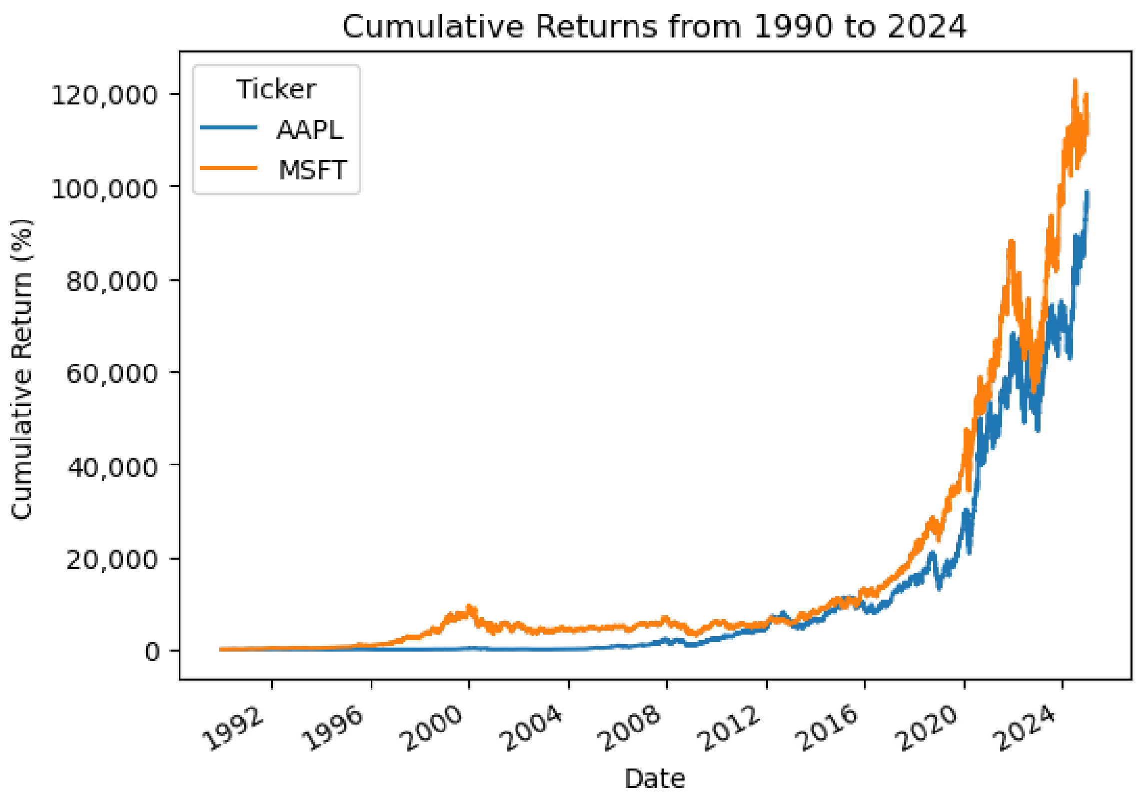
1. `.pct_change()` to calculate simple returns from adjusted close prices
2. `.plot()` to quickly plot pandas objects
3. `.mean()`, `.std()`, `.describe()`, etc. to calculate summary statistics

```
(
    df0  # DataFrame containing AAPL and MSFT data from 1980-12-12 through today
    .loc['1990':'2024', 'Adj Close']  # Slice rows for 1990-2024 (inclusive) and the 'Adj Clo
    .pct_change()  # Calculate daily percentage changes in 'Adj Close' (includes dividends an
    .add(1)  # Prepare for compounding by adding 1 to daily returns
    .cumprod()  # Compute cumulative product to get total return for each day since the start
    .sub(1)  # Convert back to cumulative returns
    .plot()  # Plot cumulative returns
)
```



The plot above is in decimal returns! pandas makes it easy to generate plots, but getting them beautiful and readable takes more work. The following code adds a title, labels, and formats the y axis.

```
from matplotlib.ticker import FuncFormatter

# Plot the data
ax = df0.loc['1990':'2024', 'Adj Close'].pct_change().add(1).cumprod().sub(1).plot()

# Format y-axis as percentages with comma separators
ax.yaxis.set_major_formatter(FuncFormatter(lambda x, _: f'{x*100:,.0f}'))

# Add labels and title if needed
plt.ylabel('Cumulative Return (%)')
plt.title('Cumulative Returns from 1990 to 2024')

# Show the plot and suppress text output (<Axes: xlabel='Date'> above)
plt.show()
```



Practice

What are the mean daily returns for these four stocks?

```
tickers = 'AAPL IBM MSFT GOOG'
```

```
returns = (
    yf.download(tickers=tickers, auto_adjust=False, progress=False)
    ['Adj Close']
    .iloc[:-1]
    .pct_change()
)

returns
```

Ticker Date	AAPL	GOOG	IBM	MSFT
1962-01-02	NaN	NaN	NaN	NaN
1962-01-03	NaN	NaN	0.0087	NaN
1962-01-04	NaN	NaN	-0.0100	NaN
1962-01-05	NaN	NaN	-0.0197	NaN
1962-01-08	NaN	NaN	-0.0188	NaN
...	...	...	...	...
2025-02-21	-0.0011	-0.0271	-0.0123	-0.0190
2025-02-24	0.0063	-0.0021	0.0015	-0.0103
2025-02-25	-0.0002	-0.0211	-0.0157	-0.0151
2025-02-26	-0.0270	-0.0151	-0.0074	0.0046
2025-02-27	-0.0127	-0.0257	-0.0102	-0.0180

```
(
    returns # daily returns from 1962 through today
    .dropna() # drop days with incomplete returns
    .iloc[:-1] # drop today, which is likely a partial-day return
    .mean() # calculate mean of daily returns from GOOG IPO through yesterday
)
```

```
Ticker
AAPL   0.0014
GOOG   0.0010
IBM    0.0004
MSFT   0.0008
dtype: float64
```

What are the standard deviations of daily returns for these four stocks?

```
(
    returns # daily returns from 1962 through today
    .dropna() # drop days with incomplete returns
    .iloc[:-1] # drop today, which is likely a partial-day return
    .std() # calculate standard deviation (volatility) of daily returns from GOOG IPO through
)
```

```
Ticker
AAPL   0.0204
```

```
GOOG   0.0193
IBM    0.0144
MSFT   0.0170
dtype: float64
```

What are the ***annualized*** means and standard deviations of daily returns for these four stocks?

```
ann_means = (
    returns # daily returns from 1962 through today
    .dropna() # drop days with missing returns
    .iloc[:-1] # drop today, which is likely a partial-day return
    .mean() # calculate mean of daily returns from close of GOOG IPO through yesterday
    .mul(252) # means grow linearly with time, so multiply by 252
)

ann_means
```

```
Ticker
AAPL   0.3577
GOOG   0.2541
IBM    0.1119
MSFT   0.1909
dtype: float64
```

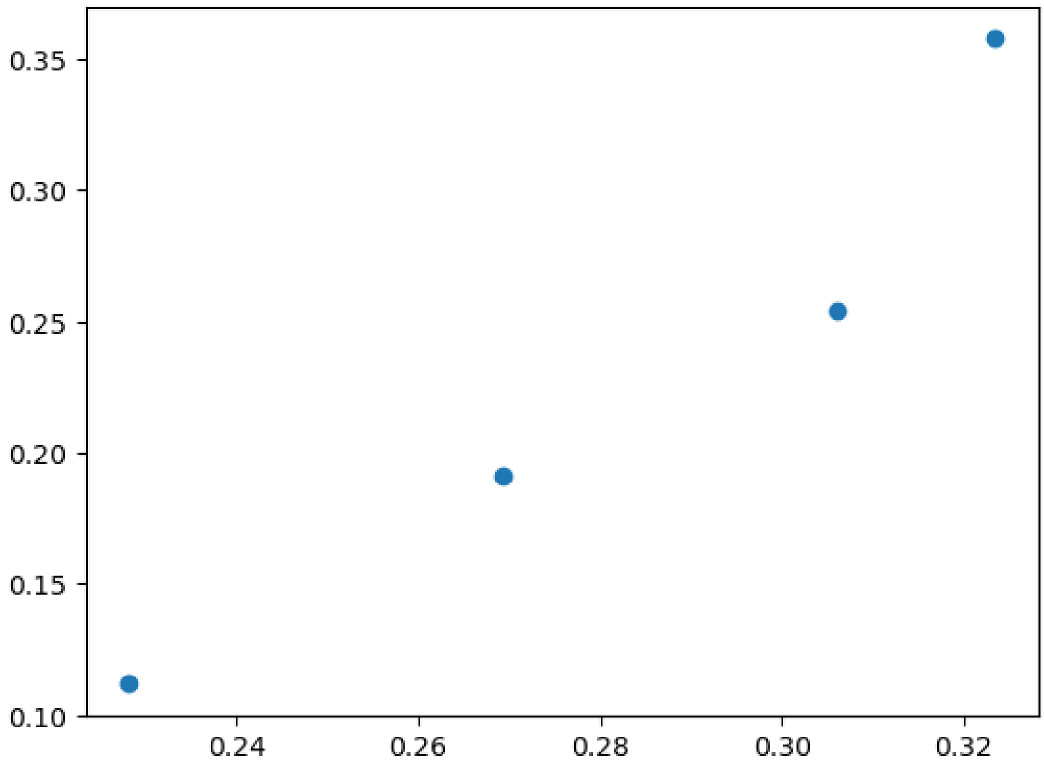
```
ann_stds = (
    returns # daily returns from 1962 through today
    .dropna() # drop days with incomplete returns
    .iloc[:-1] # drop today, which is likely a partial-day return
    .std() # calculate mean of daily returns from close of GOOG IPO through yesterday
    .mul(np.sqrt(252)) # variances grow linearly with time, so standard deviations grow sqrt
)

ann_stds
```

```
Ticker
AAPL   0.3234
GOOG   0.3060
IBM    0.2282
MSFT   0.2693
dtype: float64
```

Plot *annualized* means versus standard deviations of daily returns for these four stocks

Here is a crude plot!

```
plt.scatter(x=ann_stds, y=ann_means)
```



But we can do better than a crude plot! We will typically combine data into a data frame to make plotting easier. Because `ann_std` and `ann_means` are pandas' series, so we can use `pd.DataFrame` to combine them into a data frame.

```
df = pd.DataFrame({'Volatility': ann_stds, 'Mean Return': ann_means})

df
```

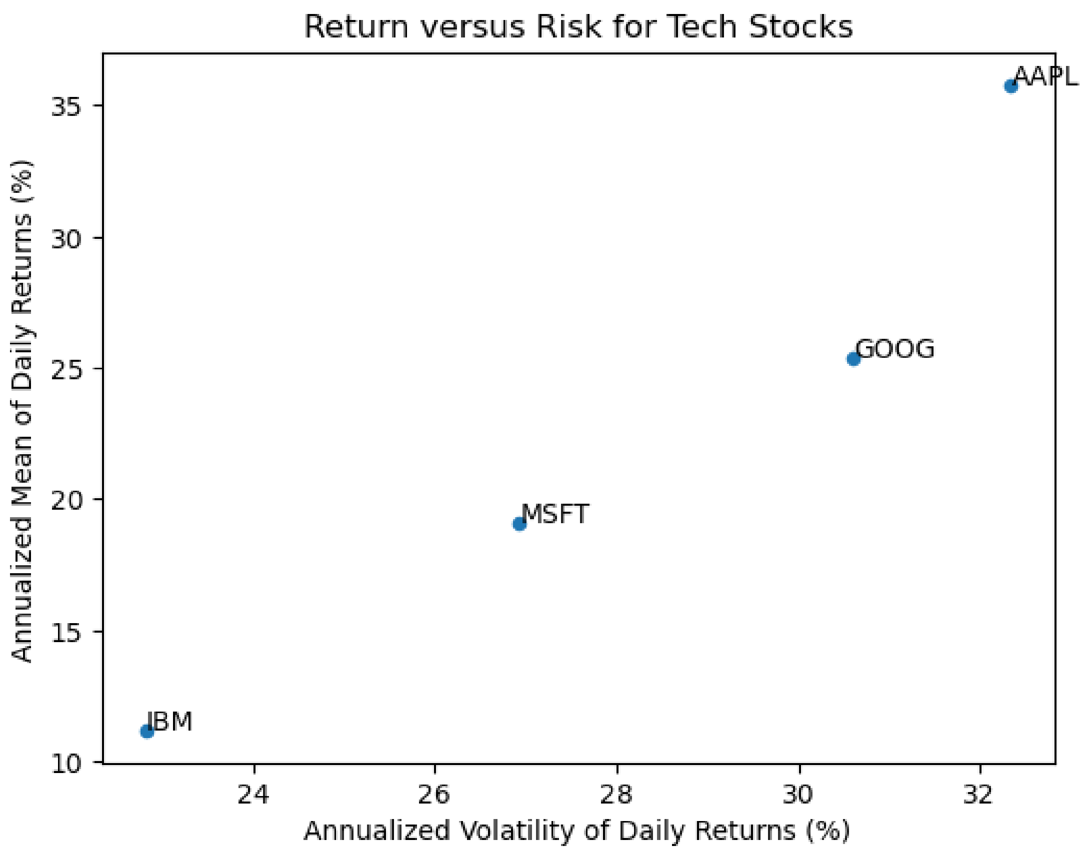
Ticker	Volatility	Mean Return
AAPL	0.3234	0.3577

	Volatility	Mean Return
Ticker		
GOOG	0.3060	0.2541
IBM	0.2282	0.1119
MSFT	0.2693	0.1909

Note

Below, we could use `enumerate()` instead of `.iterrows()`. However, `enumerate()` loops over *column names* instead of row indexes and contents. Therefore, with `enumerate()`, we would have to `.transpose()` our data frame, then use the tickers to slice the rows of our original data frame. Here `iterrows()` combines these several steps into one.

```
ax = df.plot(kind='scatter', x='Volatility', y='Mean Return')

for s, (v, mr) in df.iterrows():
    plt.text(s=s, x=v, y=mr)

ax.xaxis.set_major_formatter(FuncFormatter(lambda x, _: f'{x*100:,.0f}'))
ax.yaxis.set_major_formatter(FuncFormatter(lambda x, _: f'{x*100:,.0f}'))

plt.ylabel('Annualized Mean of Daily Returns (%)')
plt.xlabel('Annualized Volatility of Daily Returns (%)')

plt.title('Return versus Risk for Tech Stocks')
plt.show()
```



Repeat the previous calculations and plot for the stocks in the Dow-Jones Industrial Index (DJIA)

We can find the current DJIA stocks on Wikipedia. We must download new data, into `tickers_2`, `data_2`, and `returns_2`.

```
url_2 = 'https://en.wikipedia.org/wiki/Dow_Jones_Industrial_Average'
wiki_2 = pd.read_html(io=url_2)
```

```
type(wiki_2)
```

```
list
```

```
tickers_2 = wiki_2[2]['Symbol'].to_list()
```

```
data_2 = yf.download(tickers=tickers_2, auto_adjust=False, progress=False)
```

```
returns_2 = (
    data_2
    ['Adj Close']
    .iloc[:-1]
    .pct_change()
    .dropna()
)
```

```
df_2 = pd.DataFrame({
    'Volatility': returns_2.std().mul(np.sqrt(252)),
    'Mean Return': returns_2.mean().mul(252)
})
```

```
dates_2 = returns_2.index
```

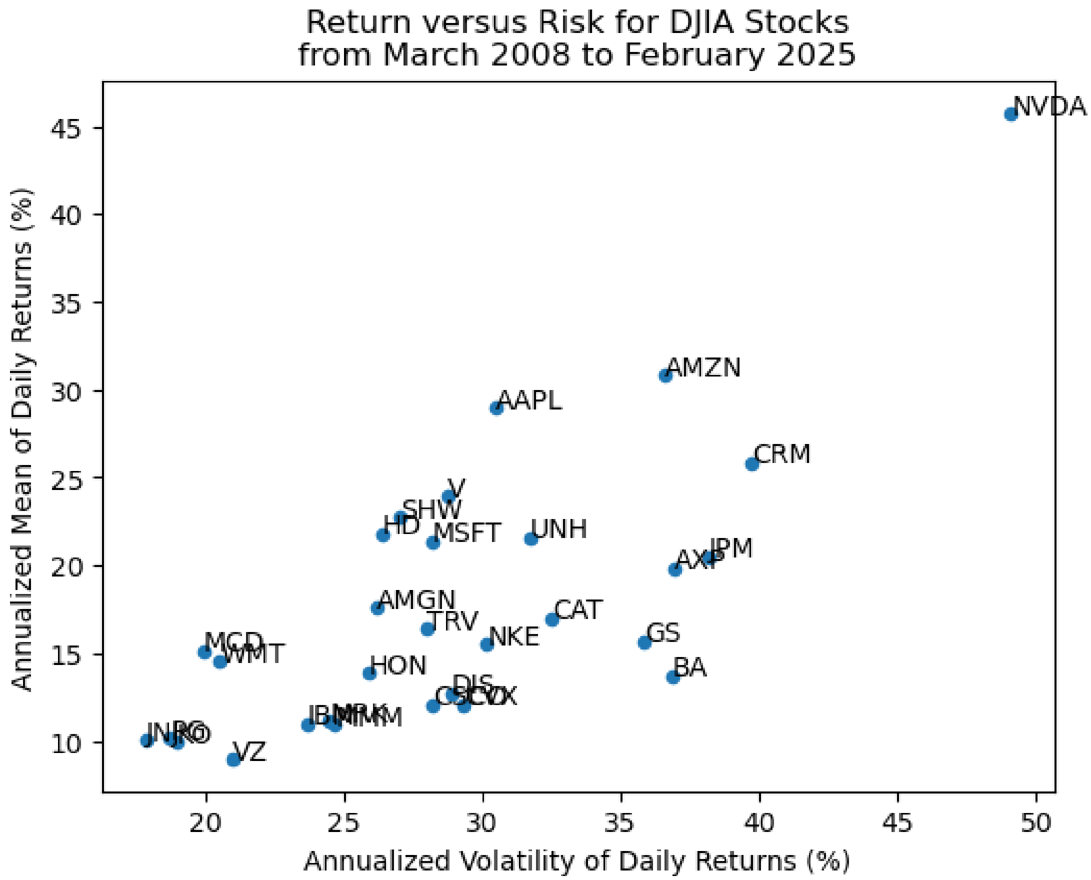
```
dates_2[0]
```

```
Timestamp('2008-03-20 00:00:00')
```

```
dates_2[-1]
```

```
Timestamp('2025-02-27 00:00:00')
```

```
ax = df_2.plot(kind='scatter', x='Volatility', y='Mean Return')

for s, (v, mr) in df_2.iterrows():
    plt.text(s=s, x=v, y=mr)

dates_2 = returns_2.index

ax.xaxis.set_major_formatter(FuncFormatter(lambda x, _: f'{x*100:,.0f}'))
ax.yaxis.set_major_formatter(FuncFormatter(lambda x, _: f'{x*100:,.0f}'))

plt.ylabel('Annualized Mean of Daily Returns (%)')
plt.xlabel('Annualized Volatility of Daily Returns (%)')

plt.title(f'Return versus Risk for DJIA Stocks\nfrom {dates_2[0]:%B %Y} to {dates_2[-1]:%B %
plt.show()
```

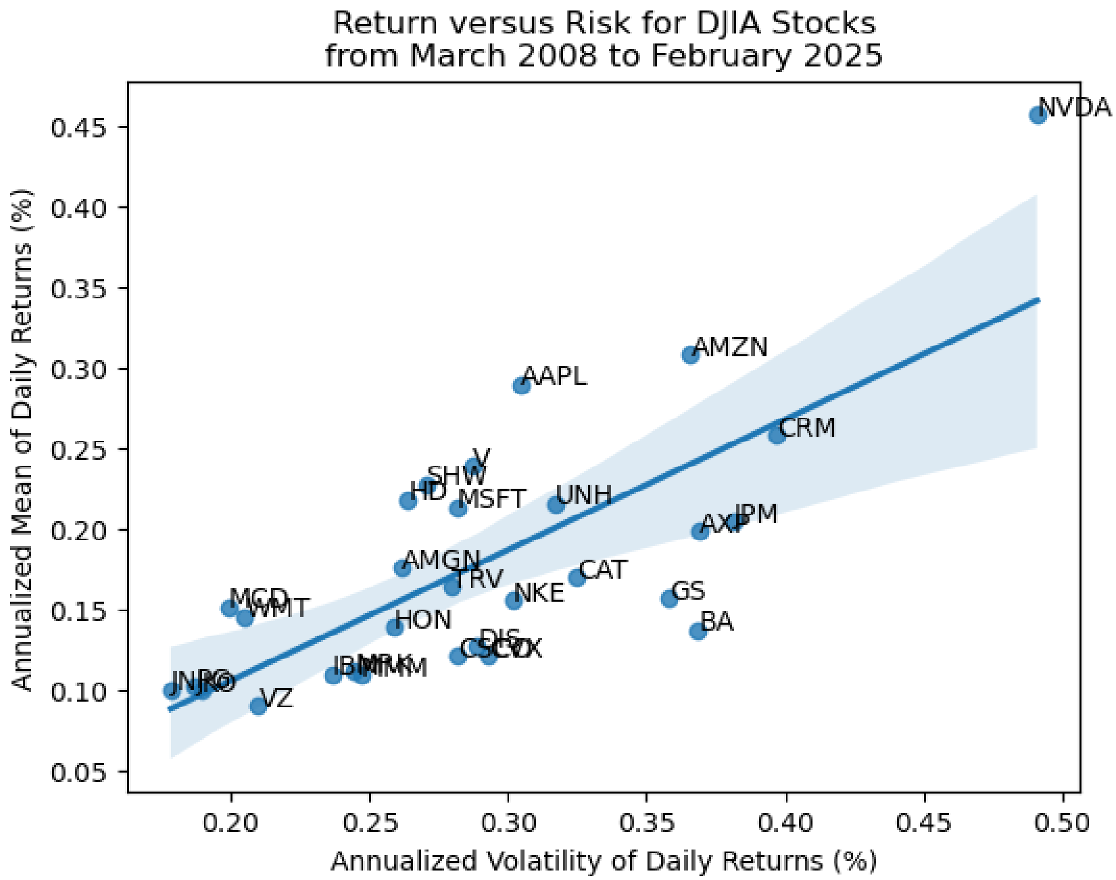


We can use the seaborn package to add a best-fit line! More on seaborn here: https://seaborn.pydata.org/index.html

```
import seaborn as sns
```

```
sns.regplot(
    data=df_2,
    x='Volatility',
    y='Mean Return'
)

for s, (v, mr) in df_2.iterrows():
    plt.text(s=s, x=v, y=mr)

ax.xaxis.set_major_formatter(FuncFormatter(lambda x, _: f'{x*100:,.0f}'))
ax.yaxis.set_major_formatter(FuncFormatter(lambda x, _: f'{x*100:,.0f}'))
```

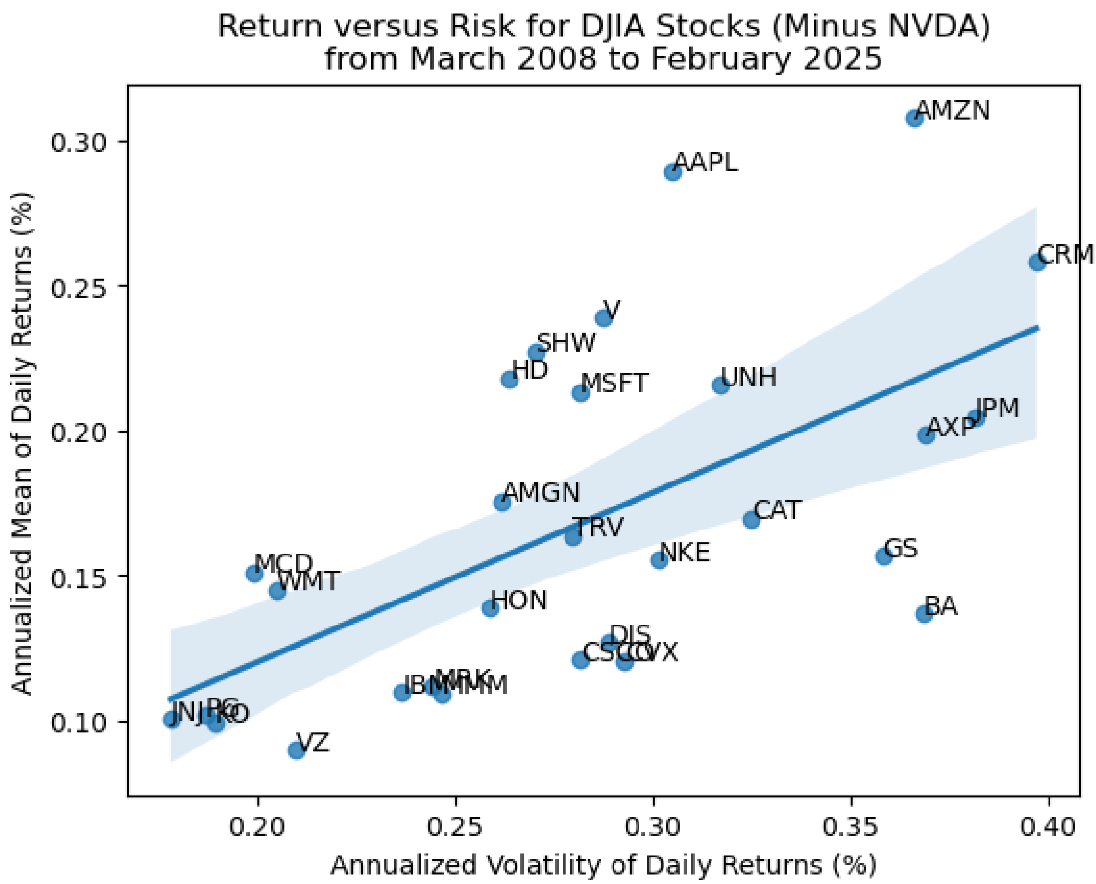
```
plt.ylabel('Annualized Mean of Daily Returns (%)')
plt.xlabel('Annualized Volatility of Daily Returns (%)')

plt.title(f'Return versus Risk for DJIA Stocks\nfrom {dates_2[0]:%B %Y} to {dates_2[-1]:%B %Y
plt.show()
```



NVDA is a real outlier! We can use the `.drop()` method to quickly drop NVDA. Theory predicts no relation between μ and σ for single stocks because single-stock risk is diversifiable. If we use a larger sample (or a different time period), we would see a flat or negatively sloped best-fit line.

```
sns.regplot(
    data=df_2.drop('NVDA'),
    x='Volatility',
    y='Mean Return'
)
```

```
for s, (v, mr) in df_2.drop('NVDA').iterrows():
    plt.text(s=s, x=v, y=mr)

dates_2 = returns_2.index

ax.xaxis.set_major_formatter(FuncFormatter(lambda x, _: f'{x*100:,.0f}'))
ax.yaxis.set_major_formatter(FuncFormatter(lambda x, _: f'{x*100:,.0f}'))

plt.ylabel('Annualized Mean of Daily Returns (%)')
plt.xlabel('Annualized Volatility of Daily Returns (%)')

plt.title(f'Return versus Risk for DJIA Stocks (Minus NVDA)\nfrom {dates_2[0]:%B %Y} to {date
plt.show()
```



Calculate total returns for the stocks in the DJIA

We can use the `.prod()` method to compound returns as $1 + R_T = \prod_{t=1}^T (1 + R_t)$. Technically, we should write R_T as $R_{0,T}$, but we typically omit the subscript 0.

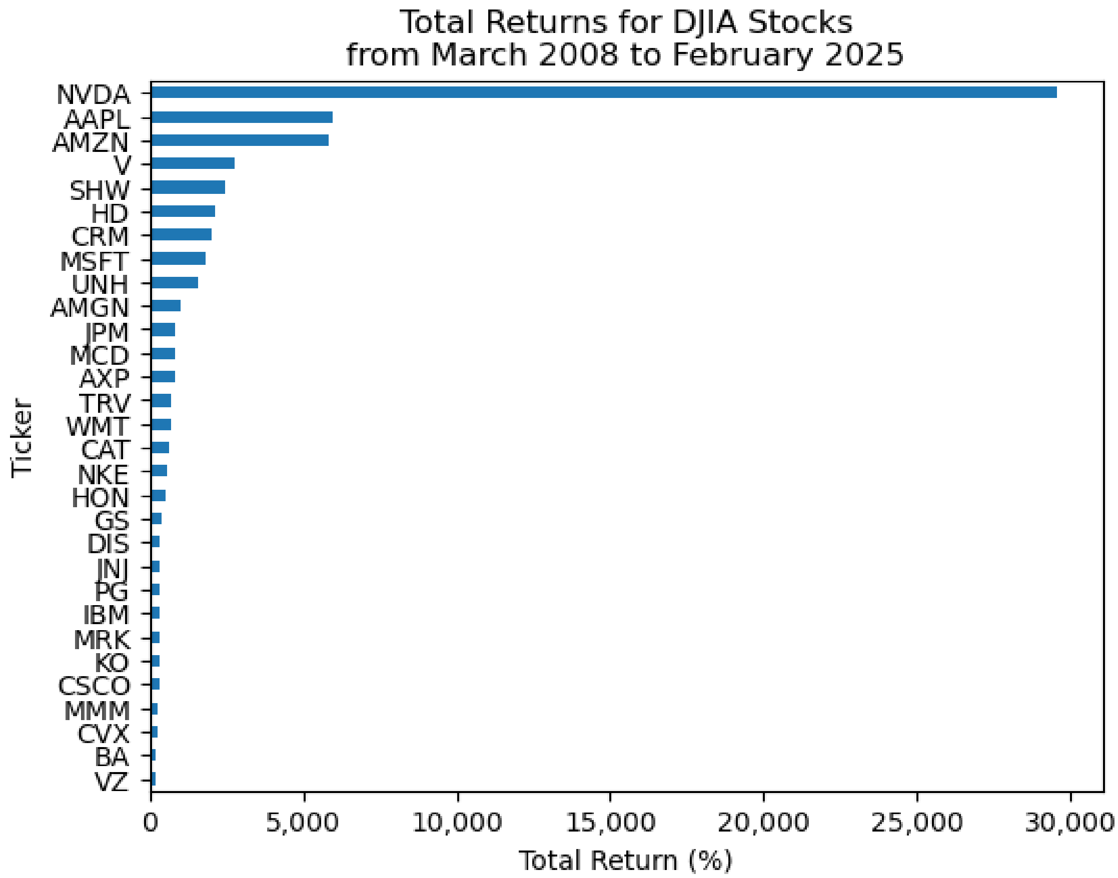
In general, I prefer to do simple math on pandas objects (data frames and series) with methods instead of operators:

For example:

1. `.add(1)` instead of `+ 1`
2. `.sub(1)` instead of `- 1`
3. `.div(1)` instead of `/ 1`
4. `.mul(1)` instead of `* 1`

The advantage of methods over operators, is that we can easily chain methods without lots of parentheses.

```
total_returns_2 = returns_2.add(1).prod().sub(1)

ax = total_returns_2.sort_values().plot(kind='barh')

ax.xaxis.set_major_formatter(FuncFormatter(lambda x, _: f'{x*100:,.0f}'))

plt.xlabel('Total Return (%)')

plt.title(f'Total Returns for DJIA Stocks\nfrom {dates_2[0]:%B %Y} to {dates_2[-1]:%B %Y}')
plt.show()
```

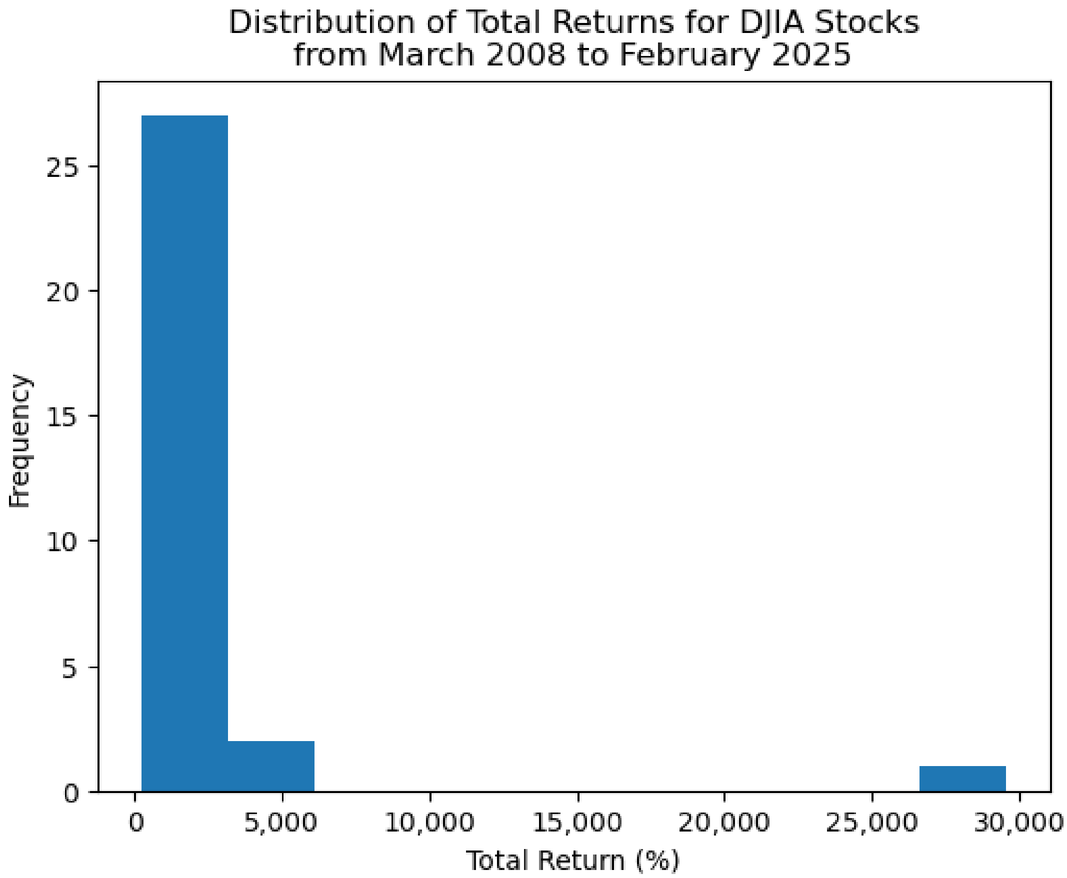


Plot the distribution of total returns for the stocks in the DJIA

We can plot a histogram, using either the `plt.hist()` function or the `.plot(kind='hist')` method.

A histogram is a great way to visualize data!

```
ax = total_returns_2.plot(kind='hist')

ax.xaxis.set_major_formatter(FuncFormatter(lambda x, _: f'{x*100:,.0f}'))

plt.xlabel('Total Return (%)')

plt.title(f'Distribution of Total Returns for DJIA Stocks\nfrom {dates_2[0]:%B %Y} to {dates_
plt.show()
```



Which stocks have the minimum and maximum total returns?

If we want the *values*, the `.min()` and `.max()` methods are the way to go!

```
total_returns_2.min()
```

```
2.1510
```

```
total_returns_2.max()
```

```
295.7450
```

The `.min()` and `.max()` methods give the values but not the tickers (or index). We use the `.idxmin()` and `.idxmax()` to get the tickers (or index).

```
total_returns_2.idxmin()
```

```
'VZ'
```

```
total_returns_2.idxmax()
```

```
'NVDA'
```

Here is what I would use to capture values and tickers!

```
total_returns_2.sort_values().iloc[[0, -1]]
```

```
Ticker
VZ        2.1510
NVDA    295.7450
dtype: float64
```

Not the exactly right tool here, but the `.nsmallest()' and`.nlargest()‘ methods are really useful!

```
total_returns_2.nsmallest(3)
```

```
Ticker
VZ     2.1510
BA     2.2134
CVX    2.7258
dtype: float64
```

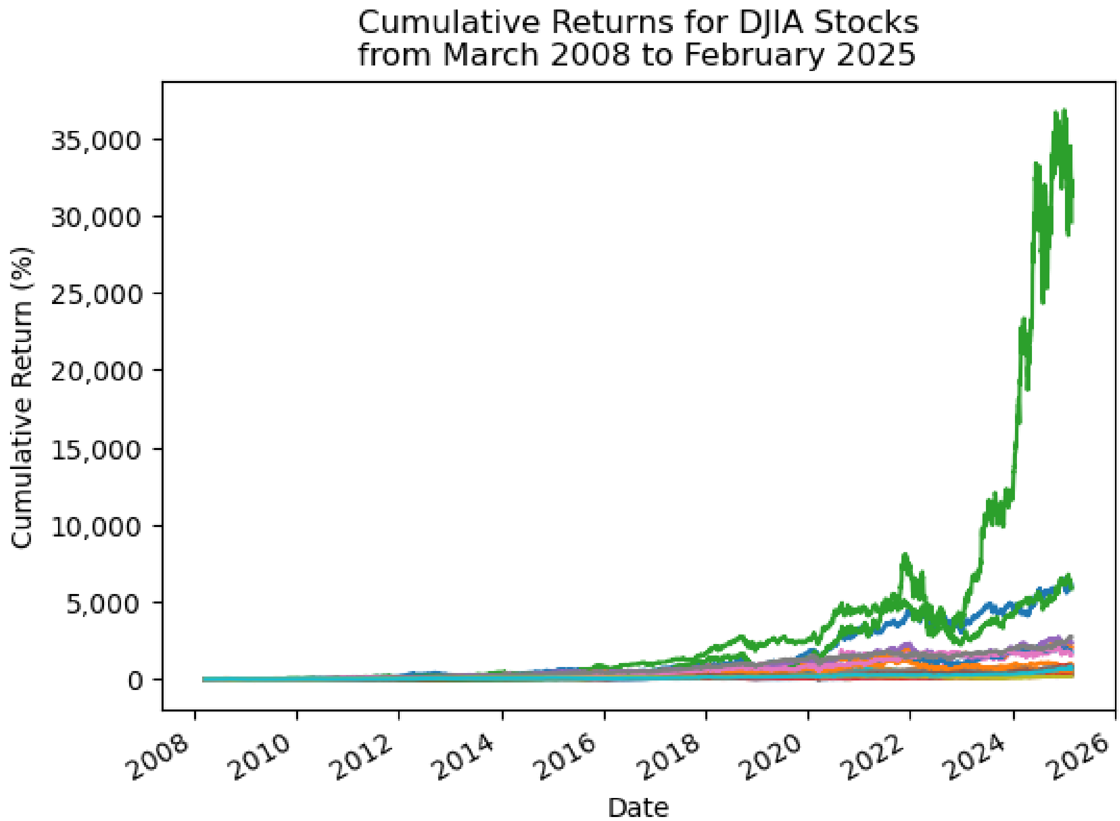
```
total_returns_2.nlargest(3)
```

```
Ticker
NVDA    295.7450
AAPL     59.8113
AMZN     58.4955
dtype: float64
```

Plot the cumulative returns for the stocks in the DJIA

We can use the cumulative product method `.cumprod()` to calculate the right hand side of the formula above.

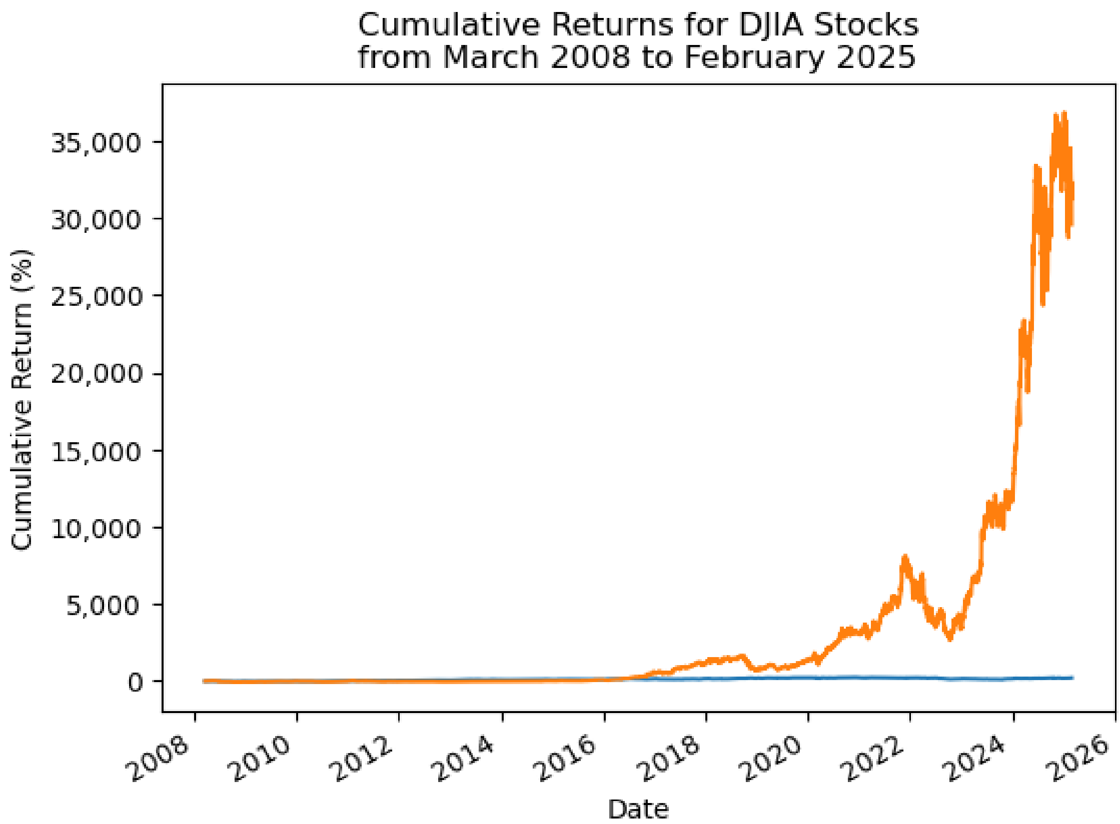
```
ax = (
    returns_2
    .add(1)
    .cumprod()
    .sub(1)
    .plot(legend=False) # with 30 stocks, this legend is too big to be useful
)

ax.yaxis.set_major_formatter(FuncFormatter(lambda x, _: f'{x*100:,.0f}'))

plt.ylabel('Cumulative Return (%)')

plt.title(f'Cumulative Returns for DJIA Stocks\nfrom {dates_2[0]:%B %Y} to {dates_2[-1]:%B %Y
plt.show()
```



Repeat the plot above with only the minimum and maximum total returns

```
ax = (
    returns_2
    [total_returns_2.sort_values().iloc[[0, -1]].index] # slice min and max total return stoc
    .add(1)
    .cumprod()
    .sub(1)
    .plot(legend=False) # with 30 stocks, this legend is too big to be useful
)

ax.yaxis.set_major_formatter(FuncFormatter(lambda x, _: f'{x*100:,.0f}'))

plt.ylabel('Cumulative Return (%)')
```

```
plt.title(f'Cumulative Returns for DJIA Stocks\nfrom {dates_2[0]:%B %Y} to {dates_2[-1]:%B %Y
plt.show()
```



McKinney Chapter 5 - Practice - Sec 03

```python
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import yfinance as yf
```

```python
%precision 4
pd.options.display.float_format = '{:.4f}'.format
# %config InlineBackend.figure_format = 'retina'
```

Announcements

1. Please keep forming groups on Canvas > People > Projects. If you want a group with more than four students, please fill a group with four students, then email with the group number and size.
2. Please keep proposing and voting for students' choice topics here.

Five-Minute Review

The pandas package makes it easy to manipulate panel data and we will use it all semester. Its name is an abbreviation of *pan*el *da*ta:

> In statistics and econometrics, panel data and longitudinal data[1][2] are both multi-dimensional data involving measurements over time. Panel data is a subset of longitudinal data where observations are for the same subjects each time.

We will download some financial data from Yahoo! Finance to cover three important tools in pandas.

First, we can use the yfinance package to easily download stock data from Yahoo! Finance.

> **Note**
>
> Starting with version 0.2.51, the `yfinance` package changed the default behavior of the `auto_adjust` argument from `False` to `True`. By default, the `yf.download()` function now returns adjusted prices, without including the `Adj Close` column.
> We prefer to work with raw data from Yahoo! Finance and explicitly calculate returns using the `Adj Close` column. Therefore, we will set `auto_adjust=False` in our `yf.download()` calls. See the yfinance changelog for release version 0.2.51.
> Also, I will use the `progress=False` argument to improve the readability of the PDF and website I render from these notebooks.

```
df0 = yf.download(tickers='AAPL MSFT', auto_adjust=False, progress=False)
```

```
df0
```

Price	Adj Close		Close		High		Low		Open
Ticker	AAPL	MSFT	AAPL	MSFT	AAPL	MSFT	AAPL	MSFT	AAPL
Date									
1980-12-12	0.0987	NaN	0.1283	NaN	0.1289	NaN	0.1283	NaN	0.1283
1980-12-15	0.0936	NaN	0.1217	NaN	0.1222	NaN	0.1217	NaN	0.1222
1980-12-16	0.0867	NaN	0.1127	NaN	0.1133	NaN	0.1127	NaN	0.1133
1980-12-17	0.0889	NaN	0.1155	NaN	0.1161	NaN	0.1155	NaN	0.1155
1980-12-18	0.0914	NaN	0.1189	NaN	0.1194	NaN	0.1189	NaN	0.1189
...	...	...	...	...	...	...	...	...	...
2025-02-24	247.1000	404.0000	247.1000	404.0000	248.8600	409.3700	244.4200	399.3200	244.9300
2025-02-25	247.0400	397.9000	247.0400	397.9000	250.0000	401.9200	244.9100	396.7000	248.0000
2025-02-26	240.3600	399.7300	240.3600	399.7300	244.9800	403.6000	239.1300	394.2500	244.3300
2025-02-27	237.3000	392.5300	237.3000	392.5300	242.4600	405.7400	237.0600	392.1700	239.4100
2025-02-28	241.8400	396.9900	241.8400	396.9900	242.0900	397.6300	230.2000	386.5700	236.9500

Second, we can slice rows and columns two ways: by integer locations with `.iloc[]` and by labels with `.loc[]`

```
df0.iloc[:6, :6]
```

Price	Adj Close		Close		High	
Ticker	AAPL	MSFT	AAPL	MSFT	AAPL	MSFT
Date						
1980-12-12	0.0987	NaN	0.1283	NaN	0.1289	NaN
1980-12-15	0.0936	NaN	0.1217	NaN	0.1222	NaN
1980-12-16	0.0867	NaN	0.1127	NaN	0.1133	NaN
1980-12-17	0.0889	NaN	0.1155	NaN	0.1161	NaN
1980-12-18	0.0914	NaN	0.1189	NaN	0.1194	NaN
1980-12-19	0.0970	NaN	0.1261	NaN	0.1267	NaN

```
df0.loc[:'1980-12-19', :'High']
```

Price	Adj Close		Close		High	
Ticker	AAPL	MSFT	AAPL	MSFT	AAPL	MSFT
Date						
1980-12-12	0.0987	NaN	0.1283	NaN	0.1289	NaN
1980-12-15	0.0936	NaN	0.1217	NaN	0.1222	NaN
1980-12-16	0.0867	NaN	0.1127	NaN	0.1133	NaN
1980-12-17	0.0889	NaN	0.1155	NaN	0.1161	NaN
1980-12-18	0.0914	NaN	0.1189	NaN	0.1194	NaN
1980-12-19	0.0970	NaN	0.1261	NaN	0.1267	NaN

Note

To slice a DataFrame:

- Use `['Name']` to select specific columns by their names.
- Use `.loc[]` to slice rows, or rows and columns together, with labels or conditional expressions.

```
df0['High']
```

Ticker	AAPL	MSFT
Date		
1980-12-12	0.1289	NaN
1980-12-15	0.1222	NaN
1980-12-16	0.1133	NaN

Ticker Date	AAPL	MSFT
1980-12-17	0.1161	NaN
1980-12-18	0.1194	NaN
...	...	...
2025-02-24	248.8600	409.3700
2025-02-25	250.0000	401.9200
2025-02-26	244.9800	403.6000
2025-02-27	242.4600	405.7400
2025-02-28	242.0900	397.6300

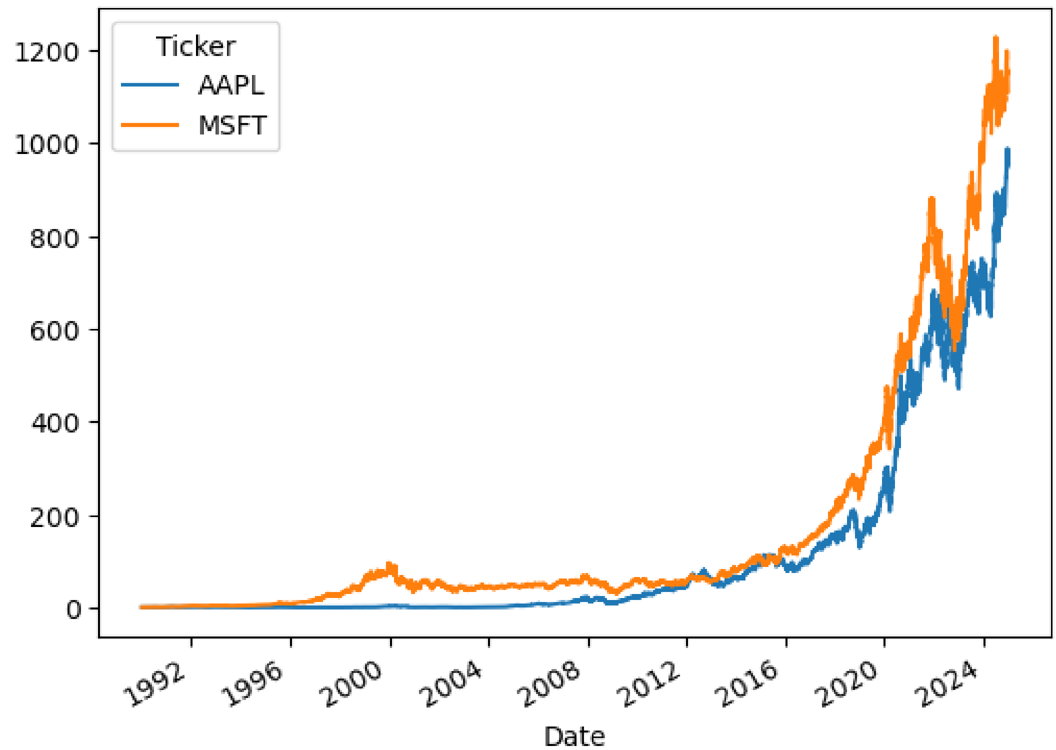
```
# # KeyError: '1980-12-12'
# df0['1980-12-12']
```

Note, if we use string labels, like dates and words, pandas includes left and right edges! This string label behavior differs from the integer location behavior everywhere else in Python. However, it is easy to figure out the sequence of integer locations. It is difficult to figure our the sequence of string labels.

Third, there many methods we can apply to pandas objects (and chain)! At this point in the course, our most common methods will be:

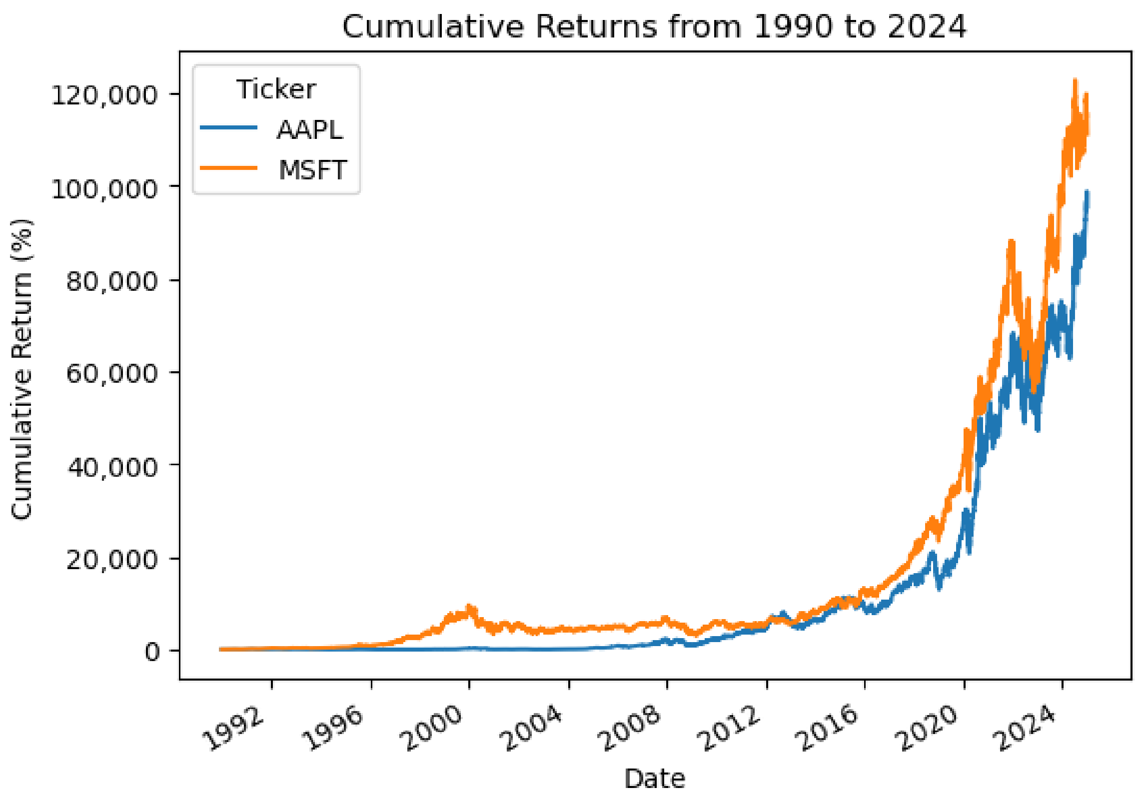
1. `.pct_change()` to calculate simple returns from adjusted close prices
2. `.plot()` to quickly plot pandas objects
3. `.mean()`, `.std()`, `.describe()`, etc. to calculate summary statistics

```
(
    df0  # DataFrame containing AAPL and MSFT data from 1980-12-12 through today
    .loc['1990':'2024', 'Adj Close']  # Slice rows for 1990-2024 (inclusive) and the 'Adj Clo
    .pct_change()  # Calculate daily percentage changes in 'Adj Close' (includes dividends an
    .add(1)  # Prepare for compounding by adding 1 to daily returns
    .cumprod()  # Compute cumulative product to get total return for each day since the start
    .sub(1)  # Convert back to cumulative returns
    .plot()  # Plot cumulative returns
)
```



The plot above is in decimal returns! pandas makes it easy to generate plots, but getting them beautiful and readable takes more work. The following code adds a title, labels, and formats the y axis.

```
from matplotlib.ticker import FuncFormatter

# Plot the data
ax = df0.loc['1990':'2024', 'Adj Close'].pct_change().add(1).cumprod().sub(1).plot()

# Format y-axis as percentages with comma separators
ax.yaxis.set_major_formatter(FuncFormatter(lambda x, _: f'{x*100:,.0f}'))

# Add labels and title if needed
plt.ylabel('Cumulative Return (%)')
plt.title('Cumulative Returns from 1990 to 2024')

# Show the plot and suppress text output (<Axes: xlabel='Date'> above)
plt.show()
```



Practice

What are the mean daily returns for these four stocks?

```
tickers = 'AAPL IBM MSFT GOOG'
```

```
returns = (
    yf.download(tickers=tickers, auto_adjust=False, progress=False)
    ['Adj Close']
    .iloc[:-1]
    .pct_change()
)
```

```
(
    returns # daily returns from 1962 through today
    .dropna() # drop days with incomplete returns
```

```
    .iloc[:-1] # drop today, which is likely a partial-day return
    .mean() # calculate mean of daily returns from GOOG IPO through yesterday
)
```

```
Ticker
AAPL   0.0014
GOOG   0.0010
IBM    0.0004
MSFT   0.0008
dtype: float64
```

What are the standard deviations of daily returns for these four stocks?

```
(
    returns # daily returns from 1962 through today
    .dropna() # drop days with incomplete returns
    .iloc[:-1] # drop today, which is likely a partial-day return
    .std() # calculate standard deviation (volatility) of daily returns from GOOG IPO throug
)
```

```
Ticker
AAPL   0.0204
GOOG   0.0193
IBM    0.0144
MSFT   0.0170
dtype: float64
```

What are the *annualized* means and standard deviations of daily returns for these four stocks?

```
ann_means = (
    returns # daily returns from 1962 through today
    .dropna() # drop days with missing returns
    .iloc[:-1] # drop today, which is likely a partial-day return
    .mean() # calculate mean of daily returns from close of GOOG IPO through yesterday
    .mul(252) # means grow linearly with time, so multiply by 252
)
```

```
ann_means
```

```
Ticker
AAPL   0.3577
GOOG   0.2541
IBM    0.1119
MSFT   0.1909
dtype: float64
```

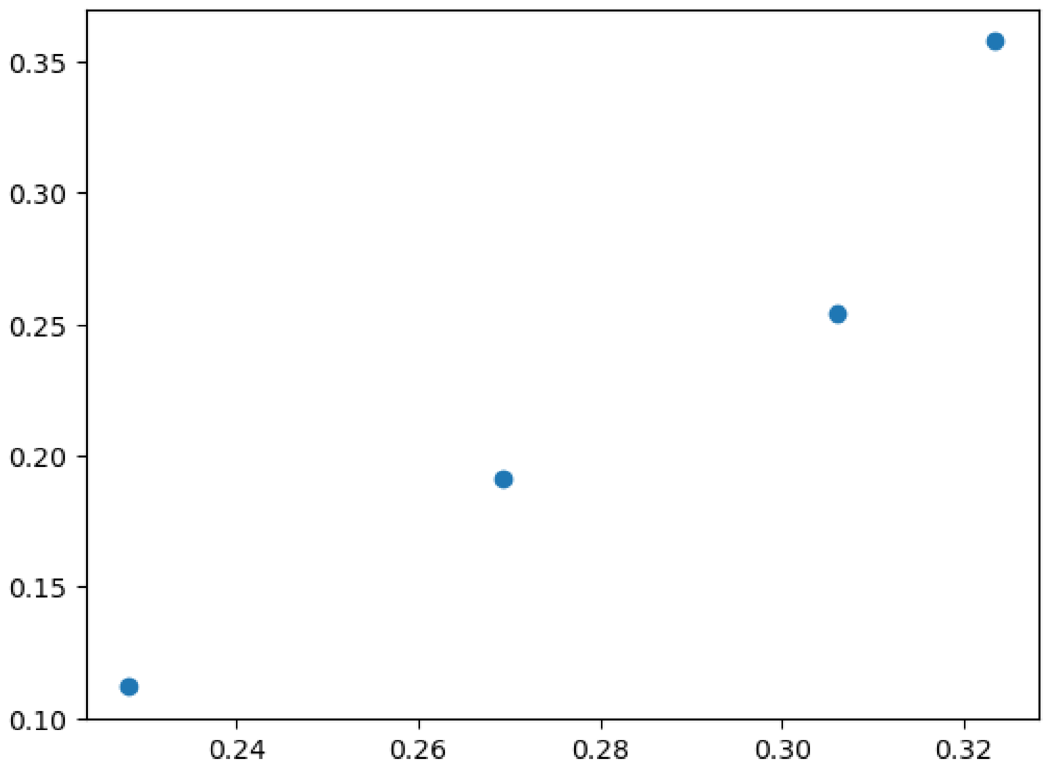
```
ann_stds = (
    returns # daily returns from 1962 through today
    .dropna() # drop days with incomplete returns
    .iloc[:-1] # drop today, which is likely a partial-day return
    .std() # calculate mean of daily returns from close of GOOG IPO through yesterday
    .mul(np.sqrt(252)) # variances grow linearly with time, so standard deviations grow sqrt
)

ann_stds
```

```
Ticker
AAPL   0.3234
GOOG   0.3060
IBM    0.2282
MSFT   0.2693
dtype: float64
```

Plot *annualized* means versus standard deviations of daily returns for these four stocks

Here is a crude plot!

```
plt.scatter(x=ann_stds, y=ann_means)
```



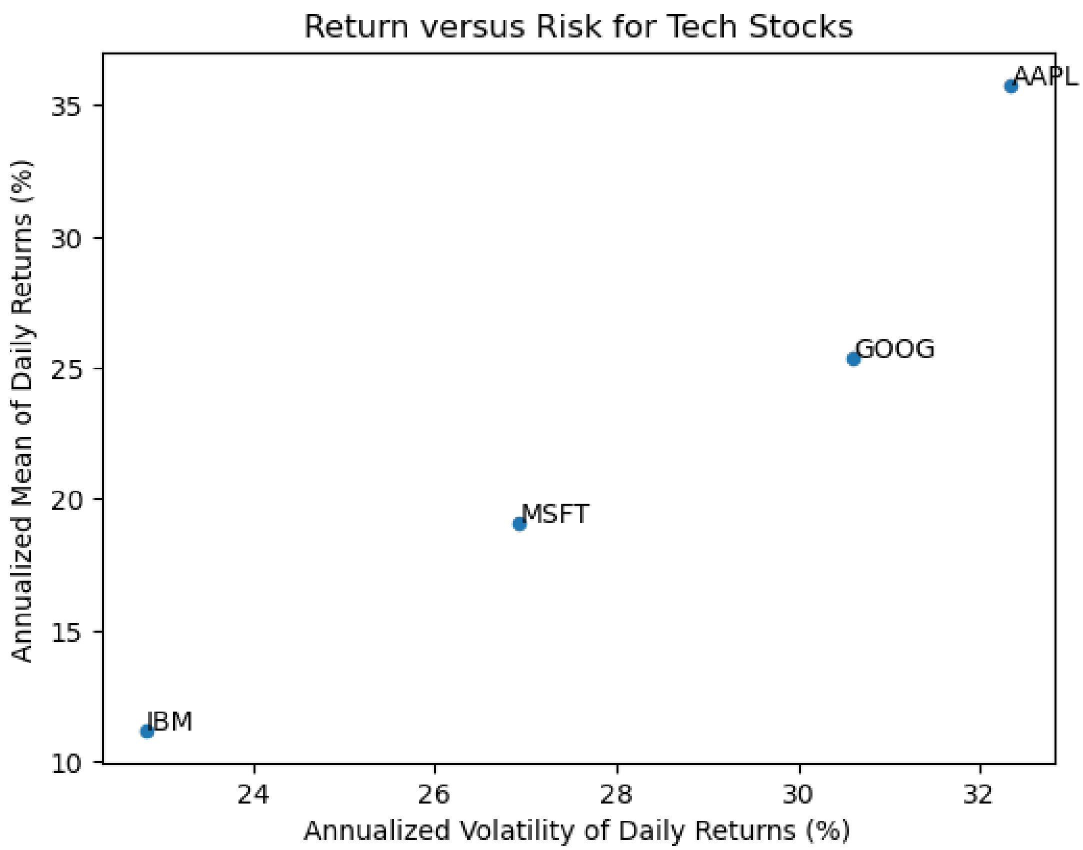
But we can do better than a crude plot! We will typically combine data into a data frame to make plotting easier. Because `ann_std` and `ann_means` are pandas' series, so we can use `pd.DataFrame` to combine them into a data frame.

```
df = pd.DataFrame({'Volatility': ann_stds, 'Mean Return': ann_means})

df
```

	Volatility	Mean Return
Ticker		
AAPL	0.3234	0.3577
GOOG	0.3060	0.2541
IBM	0.2282	0.1119
MSFT	0.2693	0.1909

> **Note**
>
> Below, we could use `enumerate()` instead of `.iterrows()`. However, `enumerate()` loops over *column names* instead of row indexes and contents. Therefore, with `enumerate()`, we would have to `.transpose()` our data frame, then use the tickers to slice the rows of our original data frame. Here `iterrows()` combines these several steps into one.

```
ax = df.plot(kind='scatter', x='Volatility', y='Mean Return')

for i, (v, mr) in df.iterrows():
    plt.text(x=v, y=mr, s=i)

ax.xaxis.set_major_formatter(FuncFormatter(lambda x, _: f'{x*100:,.0f}'))
ax.yaxis.set_major_formatter(FuncFormatter(lambda x, _: f'{x*100:,.0f}'))

plt.ylabel('Annualized Mean of Daily Returns (%)')
plt.xlabel('Annualized Volatility of Daily Returns (%)')

plt.title('Return versus Risk for Tech Stocks')
plt.show()
```



Repeat the previous calculations and plot for the stocks in the Dow-Jones Industrial Index (DJIA)

We can find the current DJIA stocks on Wikipedia. We must download new data, into `tickers_2`, `data_2`, and `returns_2`.

```
url_2 = 'https://en.wikipedia.org/wiki/Dow_Jones_Industrial_Average'
wiki_2 = pd.read_html(io=url_2)
```

```
type(wiki_2)
```

```
list
```

```
tickers_2 = wiki_2[2]['Symbol'].to_list()
```

```
returns_2 = (
    yf.download(tickers=tickers_2, auto_adjust=False, progress=False)
    ['Adj Close']
    .iloc[:-1]
    .pct_change()
    .dropna()
)
```

```
df_2 = pd.DataFrame({
    'Volatility': returns_2.std().mul(np.sqrt(252)),
    'Mean Return': returns_2.mean().mul(252),
})
```

```
dates_2 = returns_2.index
```

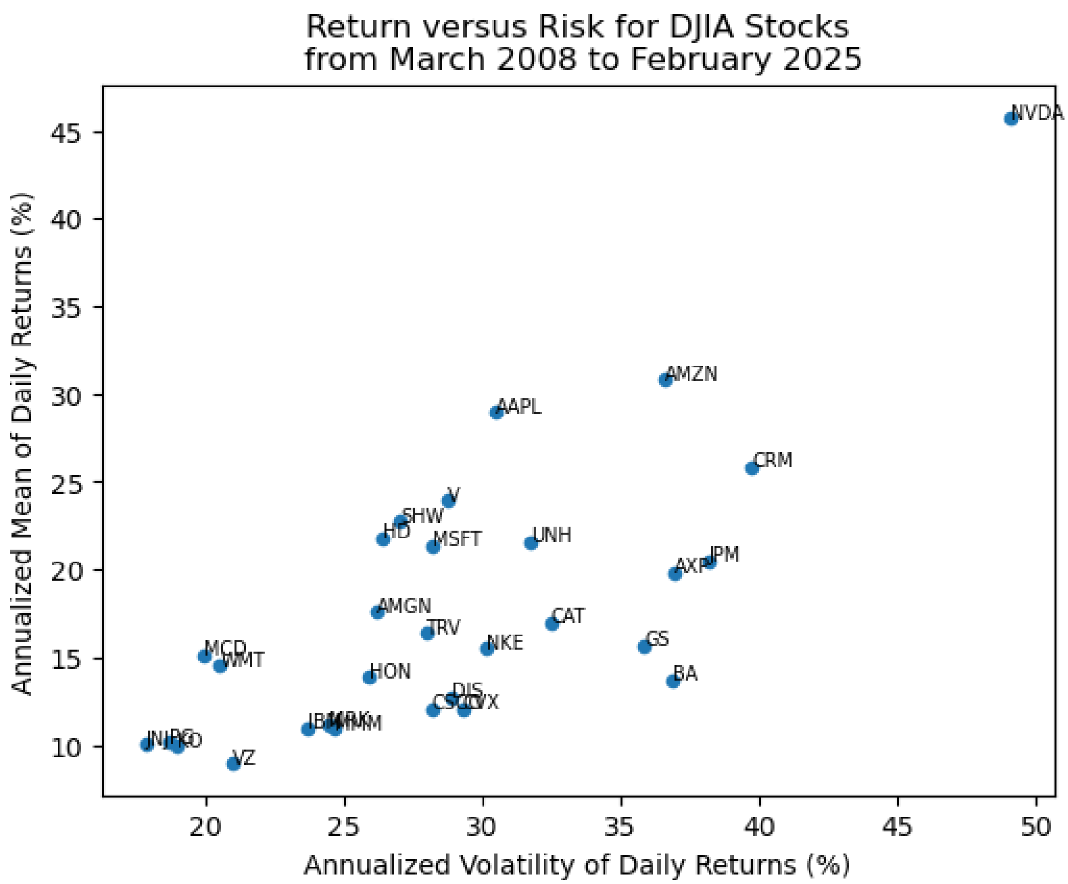
```
dates_2[0]
```

```
Timestamp('2008-03-20 00:00:00')
```

```
dates_2[-1]
```

```
Timestamp('2025-02-27 00:00:00')
```

```
ax = df_2.plot(kind='scatter', x='Volatility', y='Mean Return')

for i, (v, mr) in df_2.iterrows():
    plt.text(x=v, y=mr, s=i, fontdict={'fontsize': 'x-small'})

ax.xaxis.set_major_formatter(FuncFormatter(lambda x, _: f'{x*100:,.0f}'))
ax.yaxis.set_major_formatter(FuncFormatter(lambda x, _: f'{x*100:,.0f}'))

plt.ylabel('Annualized Mean of Daily Returns (%)')
plt.xlabel('Annualized Volatility of Daily Returns (%)')

plt.title(f'Return versus Risk for DJIA Stocks\n from {dates_2[0]:%B %Y} to {dates_2[-1]:%B %
plt.show()
```

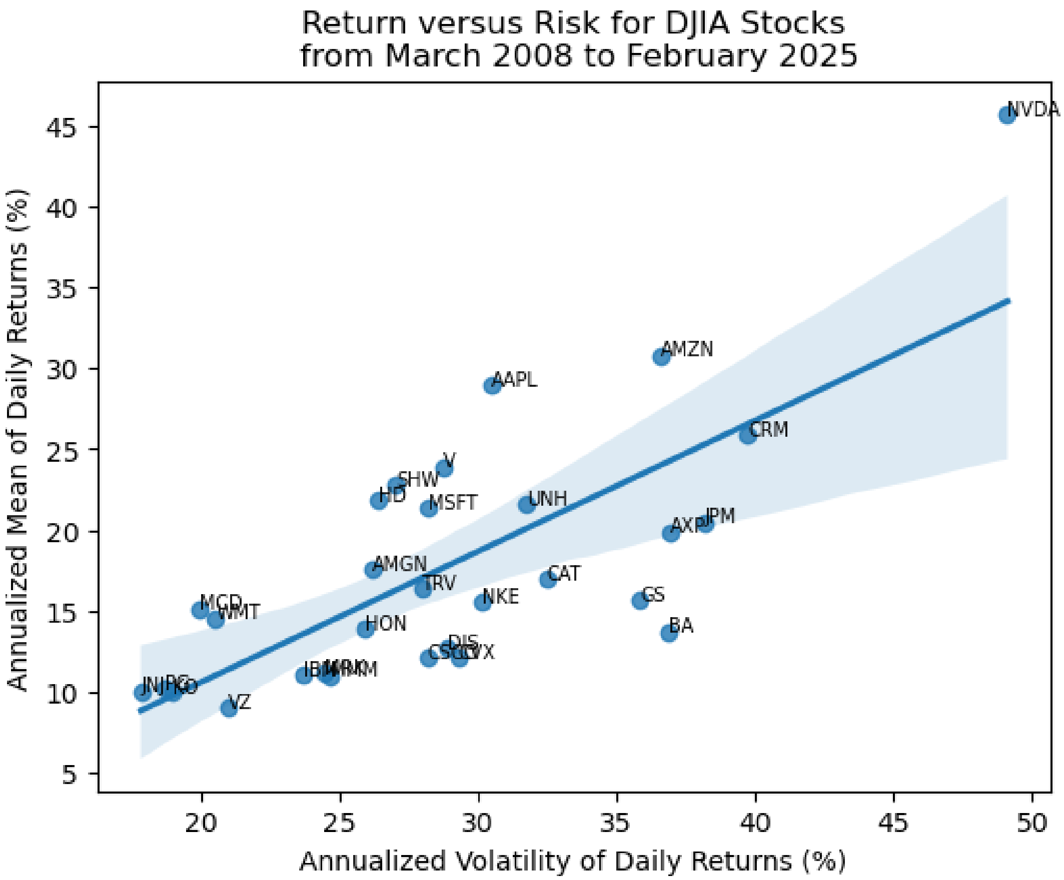


We can use the seaborn package to add a best-fit line! More on seaborn here: https://seaborn.pydata.org/index.html

```
import seaborn as sns
```

```
ax = sns.regplot(
    data=df_2,
    x='Volatility',
    y='Mean Return'
)

for i, (v, mr) in df_2.iterrows():
    plt.text(x=v, y=mr, s=i, fontdict={'fontsize': 'x-small'})

ax.xaxis.set_major_formatter(FuncFormatter(lambda x, _: f'{x*100:,.0f}'))
ax.yaxis.set_major_formatter(FuncFormatter(lambda x, _: f'{x*100:,.0f}'))
```

```
plt.ylabel('Annualized Mean of Daily Returns (%)')
plt.xlabel('Annualized Volatility of Daily Returns (%)')

plt.title(f'Return versus Risk for DJIA Stocks\n from {dates_2[0]:%B %Y} to {dates_2[-1]:%B %
plt.show()
```



Calculate total returns for the stocks in the DJIA

We can use the `.prod()` method to compound returns as $1 + R_T = \prod_{t=1}^{T} (1 + R_t)$. Technically, we should write R_T as $R_{0,T}$, but we typically omit the subscript 0.

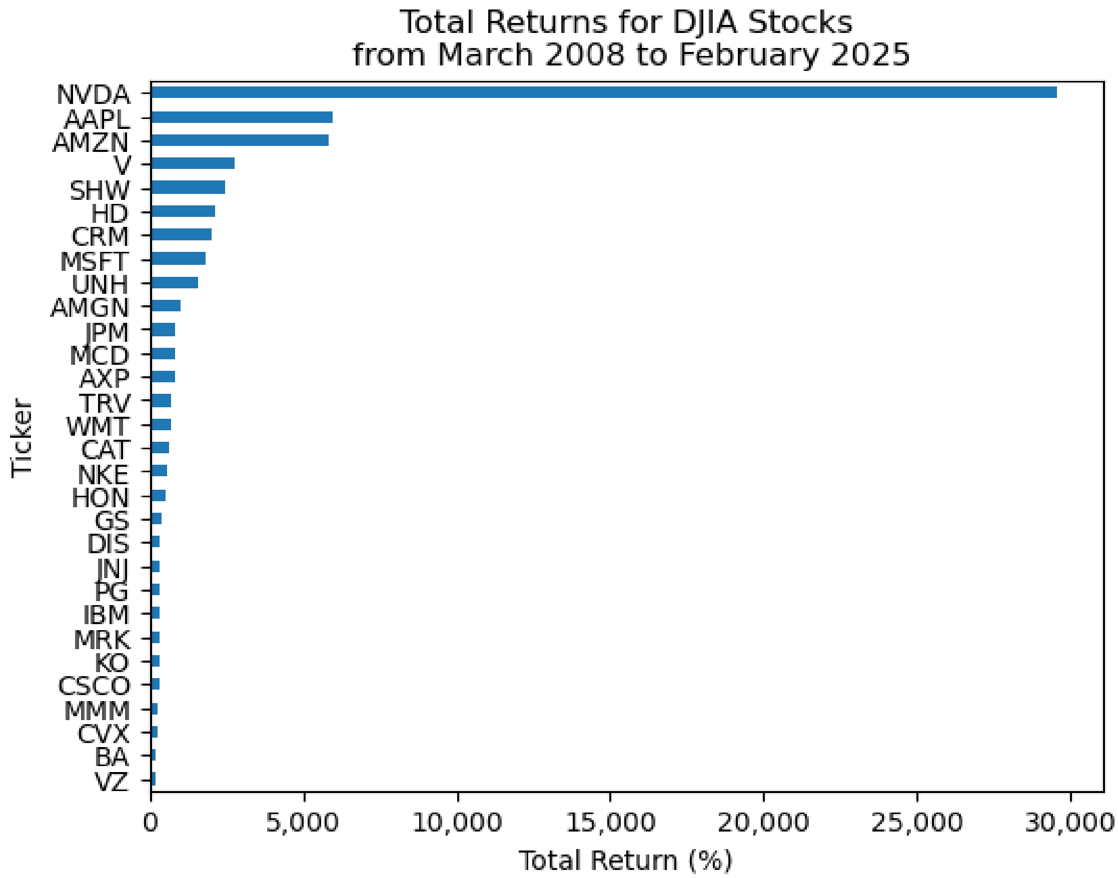
In general, I prefer to do simple math on pandas objects (data frames and series) with methods instead of operators:

For example:

1. `.add(1)` instead of `+ 1`
2. `.sub(1)` instead of `- 1`
3. `.div(1)` instead of `/ 1`
4. `.mul(1)` instead of `* 1`

The advantage of methods over operators, is that we can easily chain methods without lots of parentheses.

```
total_returns_2 = returns_2.add(1).prod().sub(1)

ax = total_returns_2.sort_values().plot(kind='barh')

ax.xaxis.set_major_formatter(FuncFormatter(lambda x, _: f'{x*100:,.0f}'))

plt.xlabel('Total Return (%)')

plt.title(f'Total Returns for DJIA Stocks\n from {dates_2[0]:%B %Y} to {dates_2[-1]:%B %Y}')
plt.show()
```

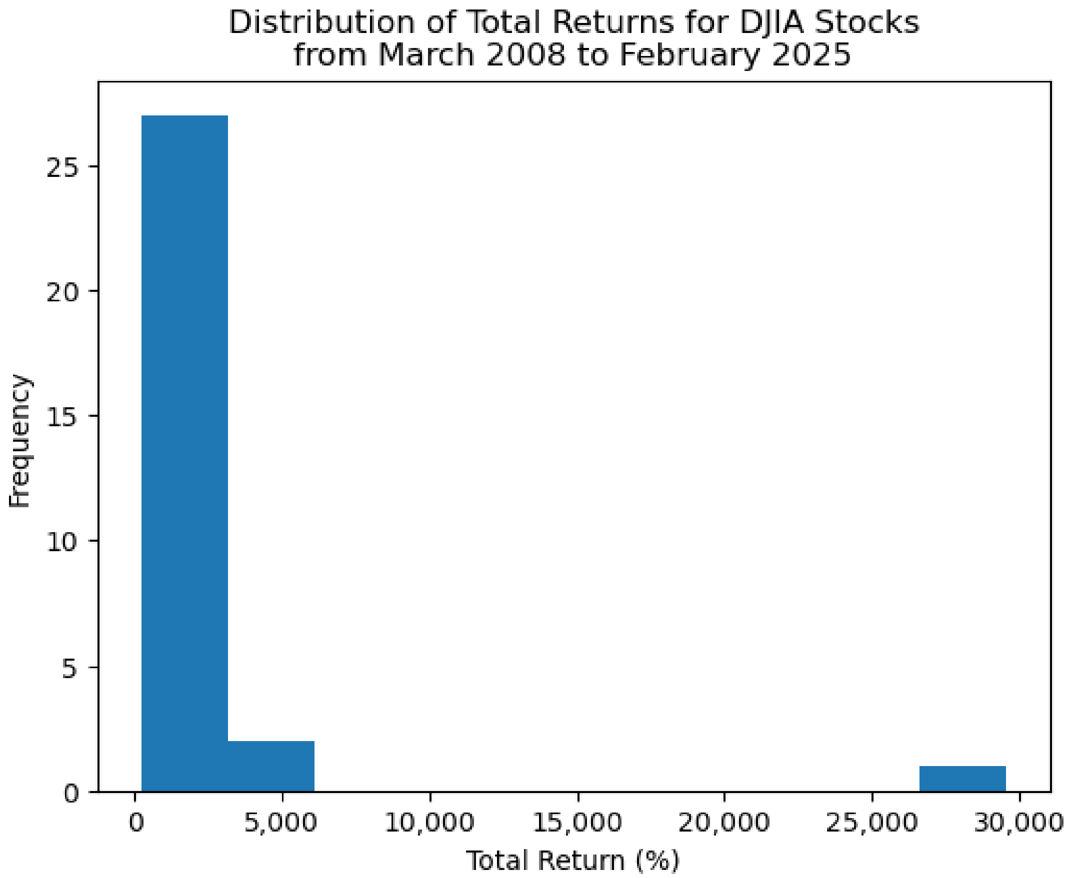


Plot the distribution of total returns for the stocks in the DJIA

We can plot a histogram, using either the `plt.hist()` function or the `.plot(kind='hist')` method.

A histogram is a great way to visualize data!

```
ax = total_returns_2.plot(kind='hist')

ax.xaxis.set_major_formatter(FuncFormatter(lambda x, _: f'{x*100:,.0f}'))

plt.xlabel('Total Return (%)')

plt.title(f'Distribution of Total Returns for DJIA Stocks\nfrom {dates_2[0]:%B %Y} to {dates_
plt.show()
```



Which stocks have the minimum and maximum total returns?

If we want the *values*, the `.min()` and `.max()` methods are the way to go!

```
total_returns_2.min()
```

```
2.1510
```

```
total_returns_2.max()
```

```
295.7450
```

The `.min()` and `.max()` methods give the values but not the tickers (or index). We use the `.idxmin()` and `.idxmax()` to get the tickers (or index).

```python
total_returns_2.idxmin()
```

```
'VZ'
```

```python
total_returns_2.idxmax()
```

```
'NVDA'
```

Here is what I would use to capture values and tickers!

```python
total_returns_2.sort_values().iloc[[0, -1]]
```

```
Ticker
VZ        2.1510
NVDA    295.7450
dtype: float64
```

Not the exactly right tool here, but the `.nsmallest()' and`.nlargest()‘ methods are really useful!

```python
total_returns_2.nsmallest(3)
```

```
Ticker
VZ     2.1510
BA     2.2134
CVX    2.7258
dtype: float64
```

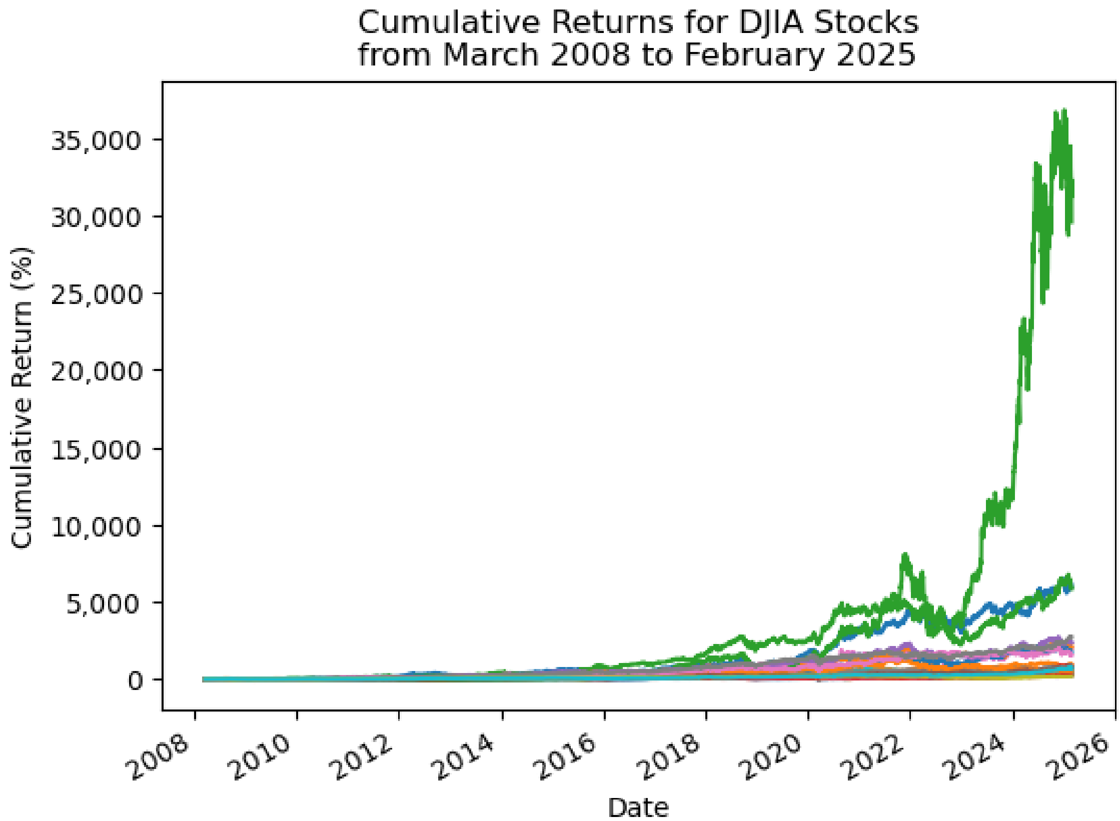
```python
total_returns_2.nlargest(3)
```

```
Ticker
NVDA    295.7450
AAPL     59.8113
AMZN     58.4955
dtype: float64
```

Plot the cumulative returns for the stocks in the DJIA

We can use the cumulative product method `.cumprod()` to calculate the right hand side of the formula above.

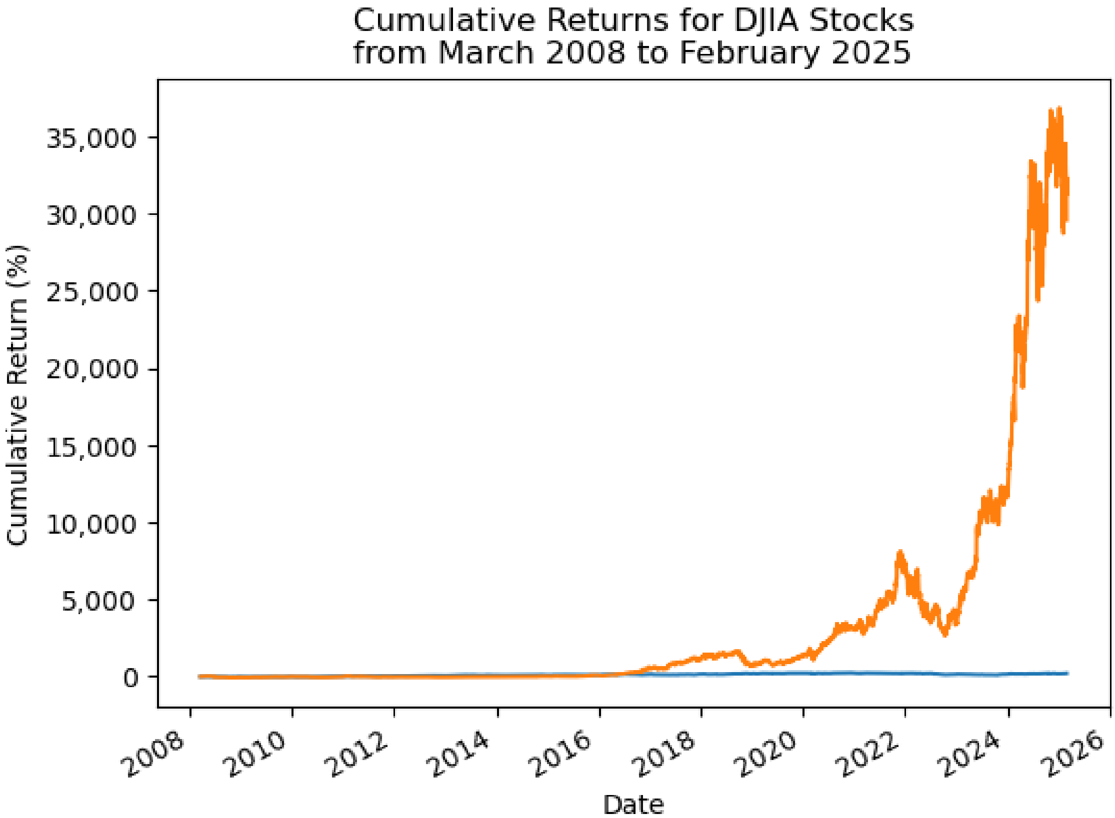
```
ax = (
    returns_2
    .add(1)
    .cumprod()
    .sub(1)
    .plot(legend=False) # with 30 stocks, this legend is too big to be useful
)

ax.yaxis.set_major_formatter(FuncFormatter(lambda x, _: f'{x*100:,.0f}'))

plt.ylabel('Cumulative Return (%)')

plt.title(f'Cumulative Returns for DJIA Stocks\nfrom {dates_2[0]:%B %Y} to {dates_2[-1]:%B %Y
plt.show()
```



Repeat the plot above with only the minimum and maximum total returns

```
ax = (
    returns_2
    [total_returns_2.sort_values().iloc[[0, -1]].index] # slice min and max total return stoc
    .add(1)
    .cumprod()
    .sub(1)
    .plot(legend=False) # with 30 stocks, this legend is too big to be useful
)

ax.yaxis.set_major_formatter(FuncFormatter(lambda x, _: f'{x*100:,.0f}'))

plt.ylabel('Cumulative Return (%)')
```

```
plt.title(f'Cumulative Returns for DJIA Stocks\nfrom {dates_2[0]:%B %Y} to {dates_2[-1]:%B %Y
plt.show()
```



McKinney Chapter 5 - Practice - Sec 04

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import yfinance as yf
```

```
%precision 4
pd.options.display.float_format = '{:.4f}'.format
# %config InlineBackend.figure_format = 'retina'
```

Announcements

1. Please keep forming groups on Canvas > People > Projects. If you want a group with more than four students, please fill a group with four students, then email with the group number and size.
2. Please keep proposing and voting for students' choice topics here.

Five-Minute Review

The pandas package makes it easy to manipulate panel data and we will use it all semester. Its name is an abbreviation of *pan*el *da*ta:

> In statistics and econometrics, panel data and longitudinal data[1][2] are both multi-dimensional data involving measurements over time. Panel data is a subset of longitudinal data where observations are for the same subjects each time.

We will download some financial data from Yahoo! Finance to cover three important tools in pandas.

First, we can use the yfinance package to easily download stock data from Yahoo! Finance.

Note

Starting with version 0.2.51, the `yfinance` package changed the default behavior of the `auto_adjust` argument from `False` to `True`. By default, the `yf.download()` function now returns adjusted prices, without including the `Adj Close` column.
We prefer to work with raw data from Yahoo! Finance and explicitly calculate returns using the `Adj Close` column. Therefore, we will set `auto_adjust=False` in our `yf.download()` calls. See the yfinance changelog for release version 0.2.51.
Also, I will use the `progress=False` argument to improve the readability of the PDF and website I render from these notebooks.

```
df0 = yf.download(tickers='AAPL MSFT', auto_adjust=False, progress=False)
```

```
df0
```

Price	Adj Close		Close		High		Low		Open
Ticker	AAPL	MSFT	AAPL	MSFT	AAPL	MSFT	AAPL	MSFT	AAPL
Date									
1980-12-12	0.0987	NaN	0.1283	NaN	0.1289	NaN	0.1283	NaN	0.1283
1980-12-15	0.0936	NaN	0.1217	NaN	0.1222	NaN	0.1217	NaN	0.1222
1980-12-16	0.0867	NaN	0.1127	NaN	0.1133	NaN	0.1127	NaN	0.1133
1980-12-17	0.0889	NaN	0.1155	NaN	0.1161	NaN	0.1155	NaN	0.1155
1980-12-18	0.0914	NaN	0.1189	NaN	0.1194	NaN	0.1189	NaN	0.1189
...	...	...	...	...	...	...	...	...	...
2025-02-24	247.1000	404.0000	247.1000	404.0000	248.8600	409.3700	244.4200	399.3200	244.9300
2025-02-25	247.0400	397.9000	247.0400	397.9000	250.0000	401.9200	244.9100	396.7000	248.0000
2025-02-26	240.3600	399.7300	240.3600	399.7300	244.9800	403.6000	239.1300	394.2500	244.3300
2025-02-27	237.3000	392.5300	237.3000	392.5300	242.4600	405.7400	237.0600	392.1700	239.4100
2025-02-28	241.8400	396.9900	241.8400	396.9900	242.0900	397.6300	230.2000	386.5700	236.9500

Second, we can slice rows and columns two ways: by integer locations with `.iloc[]` and by labels with `.loc[]`

```
df0.iloc[:6, :6]
```

Price	Adj Close		Close		High	
Ticker	AAPL	MSFT	AAPL	MSFT	AAPL	MSFT
Date						
1980-12-12	0.0987	NaN	0.1283	NaN	0.1289	NaN
1980-12-15	0.0936	NaN	0.1217	NaN	0.1222	NaN
1980-12-16	0.0867	NaN	0.1127	NaN	0.1133	NaN
1980-12-17	0.0889	NaN	0.1155	NaN	0.1161	NaN
1980-12-18	0.0914	NaN	0.1189	NaN	0.1194	NaN
1980-12-19	0.0970	NaN	0.1261	NaN	0.1267	NaN

```
df0.loc[:'1980-12-19', :'High']
```

Price	Adj Close		Close		High	
Ticker	AAPL	MSFT	AAPL	MSFT	AAPL	MSFT
Date						
1980-12-12	0.0987	NaN	0.1283	NaN	0.1289	NaN
1980-12-15	0.0936	NaN	0.1217	NaN	0.1222	NaN
1980-12-16	0.0867	NaN	0.1127	NaN	0.1133	NaN
1980-12-17	0.0889	NaN	0.1155	NaN	0.1161	NaN
1980-12-18	0.0914	NaN	0.1189	NaN	0.1194	NaN
1980-12-19	0.0970	NaN	0.1261	NaN	0.1267	NaN

Note

To slice a DataFrame:

- Use `['Name']` to select specific columns by their names.
- Use `.loc[]` to slice rows, or rows and columns together, with labels or conditional expressions.

```
df0['High']
```

Ticker	AAPL	MSFT
Date		
1980-12-12	0.1289	NaN
1980-12-15	0.1222	NaN
1980-12-16	0.1133	NaN

Ticker Date	AAPL	MSFT
1980-12-17	0.1161	NaN
1980-12-18	0.1194	NaN
...	...	...
2025-02-24	248.8600	409.3700
2025-02-25	250.0000	401.9200
2025-02-26	244.9800	403.6000
2025-02-27	242.4600	405.7400
2025-02-28	242.0900	397.6300

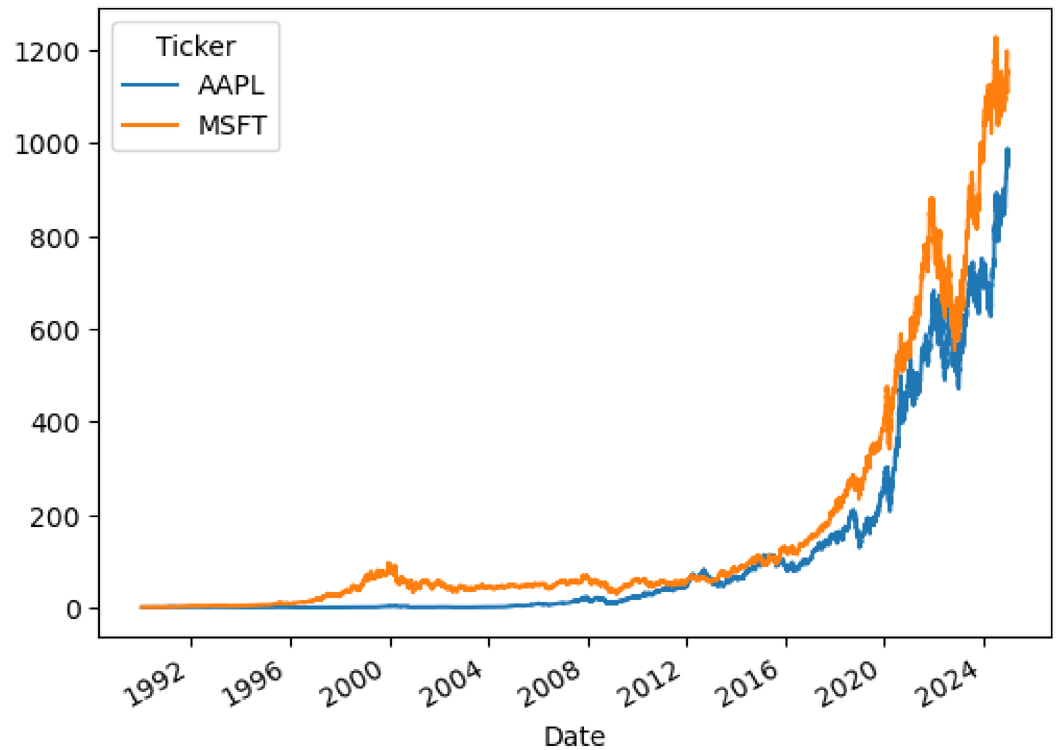
```
# # KeyError: '1980-12-12'
# df0['1980-12-12']
```

Note, if we use string labels, like dates and words, pandas includes left and right edges! This string label behavior differs from the integer location behavior everywhere else in Python. However, it is easy to figure out the sequence of integer locations. It is difficult to figure our the sequence of string labels.

Third, there many methods we can apply to pandas objects (and chain)! At this point in the course, our most common methods will be:

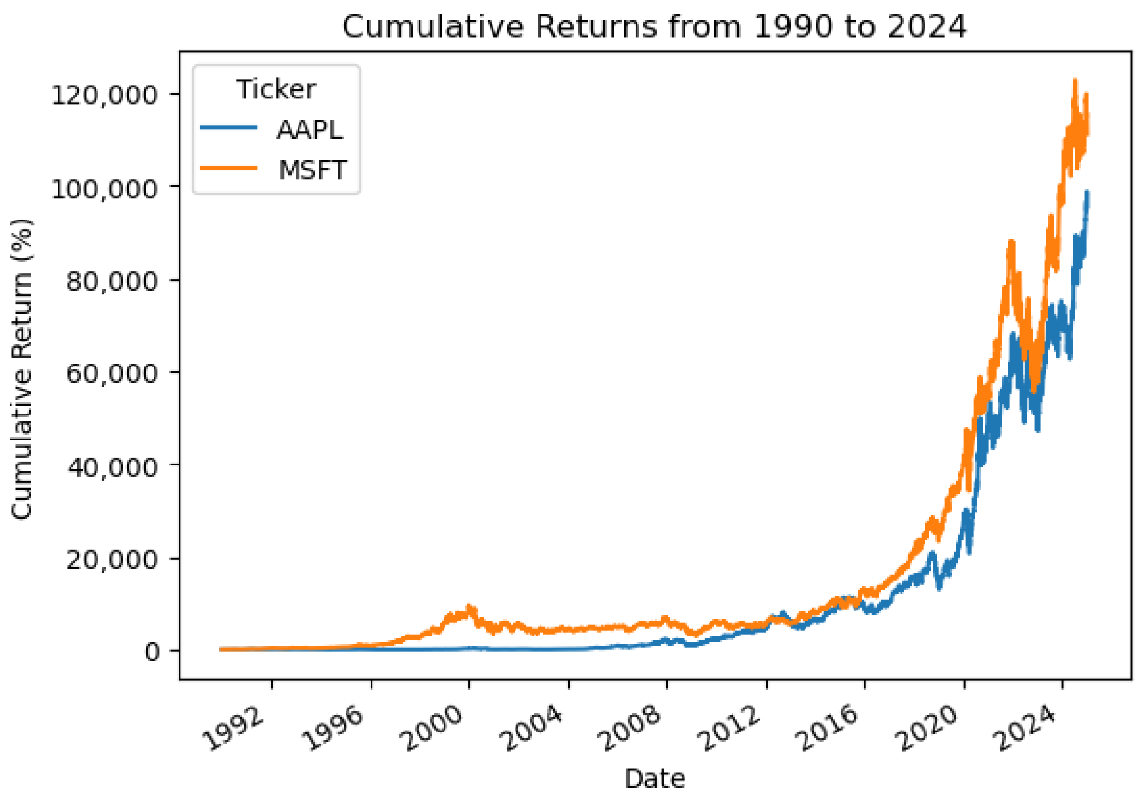
1. `.pct_change()` to calculate simple returns from adjusted close prices
2. `.plot()` to quickly plot pandas objects
3. `.mean()`, `.std()`, `.describe()`, etc. to calculate summary statistics

```
(
    df0  # DataFrame containing AAPL and MSFT data from 1980-12-12 through today
    .loc['1990':'2024', 'Adj Close']  # Slice rows for 1990-2024 (inclusive) and the 'Adj Clo
    .pct_change()  # Calculate daily percentage changes in 'Adj Close' (includes dividends an
    .add(1)  # Prepare for compounding by adding 1 to daily returns
    .cumprod()  # Compute cumulative product to get total return for each day since the start
    .sub(1)  # Convert back to cumulative returns
    .plot()  # Plot cumulative returns
)
```



The plot above is in decimal returns! pandas makes it easy to generate plots, but getting them beautiful and readable takes more work. The following code adds a title, labels, and formats the y axis.

```
from matplotlib.ticker import FuncFormatter

# Plot the data
ax = df0.loc['1990':'2024', 'Adj Close'].pct_change().add(1).cumprod().sub(1).plot()

# Format y-axis as percentages with comma separators
ax.yaxis.set_major_formatter(FuncFormatter(lambda x, _: f'{x*100:,.0f}'))

# Add labels and title if needed
plt.ylabel('Cumulative Return (%)')
plt.title('Cumulative Returns from 1990 to 2024')

# Show the plot and suppress text output (<Axes: xlabel='Date'> above)
plt.show()
```



Practice

What are the mean daily returns for these four stocks?

```
tickers = 'AAPL IBM MSFT GOOG'
```

```
returns = (
    yf.download(tickers=tickers, auto_adjust=False, progress=False)
    ['Adj Close']
    .pct_change()
)
```

```
(
    returns # daily returns from 1962 through today
    .dropna() # drop days with incomplete returns
    .iloc[:-1] # drop today, which is likely a partial-day return
```

```
    .mean() # calculate mean of daily returns from GOOG IPO through yesterday
)
```

```
Ticker
AAPL   0.0014
GOOG   0.0010
IBM    0.0004
MSFT   0.0008
dtype: float64
```

What are the standard deviations of daily returns for these four stocks?

```
(
    returns # daily returns from 1962 through today
    .dropna() # drop days with incomplete returns
    .iloc[:-1] # drop today, which is likely a partial-day return
    .std() # calculate standard deviation (volatility) of daily returns from GOOG IPO throug
)
```

```
Ticker
AAPL   0.0204
GOOG   0.0193
IBM    0.0144
MSFT   0.0170
dtype: float64
```

What are the *annualized* means and standard deviations of daily returns for these four stocks?

```
ann_means = (
    returns # daily returns from 1962 through today
    .dropna() # drop days with missing returns
    .iloc[:-1] # drop today, which is likely a partial-day return
    .mean() # calculate mean of daily returns from close of GOOG IPO through yesterday
    .mul(252) # means grow linearly with time, so multiply by 252
)

ann_means
```

```
Ticker
AAPL   0.3570
GOOG   0.2528
IBM    0.1113
MSFT   0.1900
dtype: float64
```

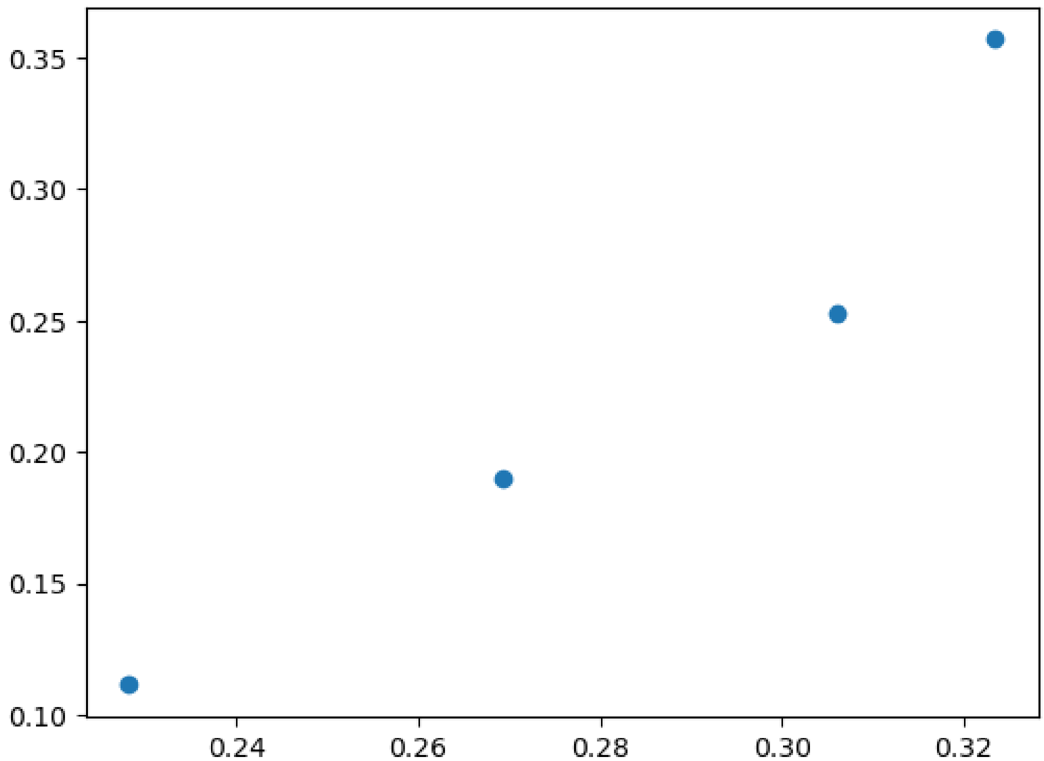
```
ann_stds = (
    returns # daily returns from 1962 through today
    .dropna() # drop days with incomplete returns
    .iloc[:-1] # drop today, which is likely a partial-day return
    .std() # calculate mean of daily returns from close of GOOG IPO through yesterday
    .mul(np.sqrt(252)) # variances grow linearly with time, so standard deviations grow sqrt
)

ann_stds
```

```
Ticker
AAPL   0.3234
GOOG   0.3061
IBM    0.2282
MSFT   0.2693
dtype: float64
```

Plot *annualized* means versus standard deviations of daily returns for these four stocks

Here is a crude plot!

```
plt.scatter(x=ann_stds, y=ann_means)
```



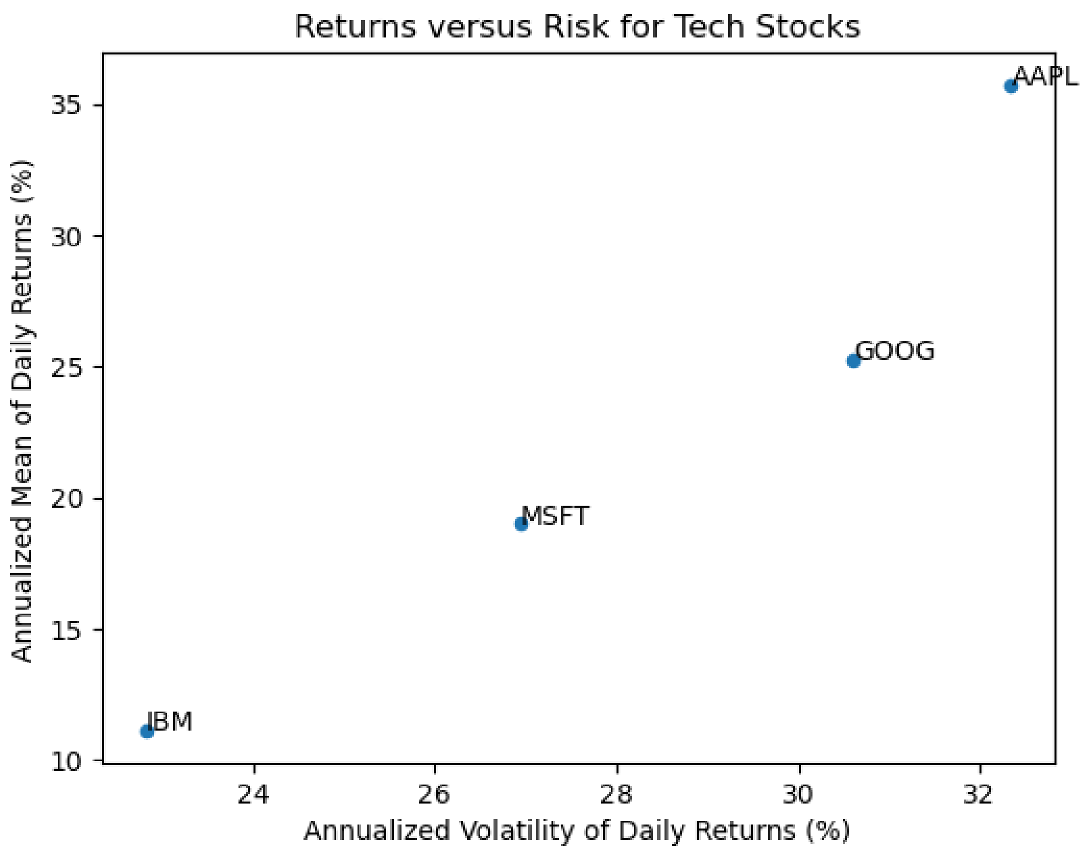
But we can do better than a crude plot! We will typically combine data into a data frame to make plotting easier. Because `ann_std` and `ann_means` are pandas' series, so we can use `pd.DataFrame` to combine them into a data frame.

```
df = pd.DataFrame({'Volatility': ann_stds, 'Mean Return': ann_means})

df
```

Ticker	Volatility	Mean Return
AAPL	0.3234	0.3570
GOOG	0.3061	0.2528
IBM	0.2282	0.1113
MSFT	0.2693	0.1900

> **Note**
>
> Below, we could use `enumerate()` instead of `.iterrows()`. However, `enumerate()` loops over *column names* instead of row indexes and contents. Therefore, with `enumerate()`, we would have to `.transpose()` our data frame, then use the tickers to slice the rows of our original data frame. Here `iterrows()` combines these several steps into one.

```
ax = df.plot(kind='scatter', x='Volatility', y='Mean Return')

for i, (v, mr) in df.iterrows():
    plt.text(s=i, x=v, y=mr)

ax.xaxis.set_major_formatter(FuncFormatter(lambda x, _: f'{x*100:,.0f}'))
ax.yaxis.set_major_formatter(FuncFormatter(lambda x, _: f'{x*100:,.0f}'))

plt.xlabel('Annualized Volatility of Daily Returns (%)')
plt.ylabel('Annualized Mean of Daily Returns (%)')

plt.title('Returns versus Risk for Tech Stocks')
plt.show()
```



Repeat the previous calculations and plot for the stocks in the Dow-Jones Industrial Index (DJIA)

We can find the current DJIA stocks on Wikipedia. We must download new data, into `tickers_2`, `data_2`, and `returns_2`.

```
url_2 = 'https://en.wikipedia.org/wiki/Dow_Jones_Industrial_Average'
wiki_2 = pd.read_html(io=url_2)
```

```
type(wiki_2)
```

```
list
```

```
tickers_2 = wiki_2[2]['Symbol'].to_list()
```

```
returns_2 = (
    yf.download(tickers=tickers_2, auto_adjust=False, progress=False)
    ['Adj Close']
    .iloc[:-1]
    .pct_change()
    .dropna()
)
```

```
df_2 = pd.DataFrame({
    'Volatility': returns_2.std().mul(np.sqrt(252)),
    'Mean Return': returns_2.mean().mul(252)
})
```

```
dates_2 = returns_2.index
```

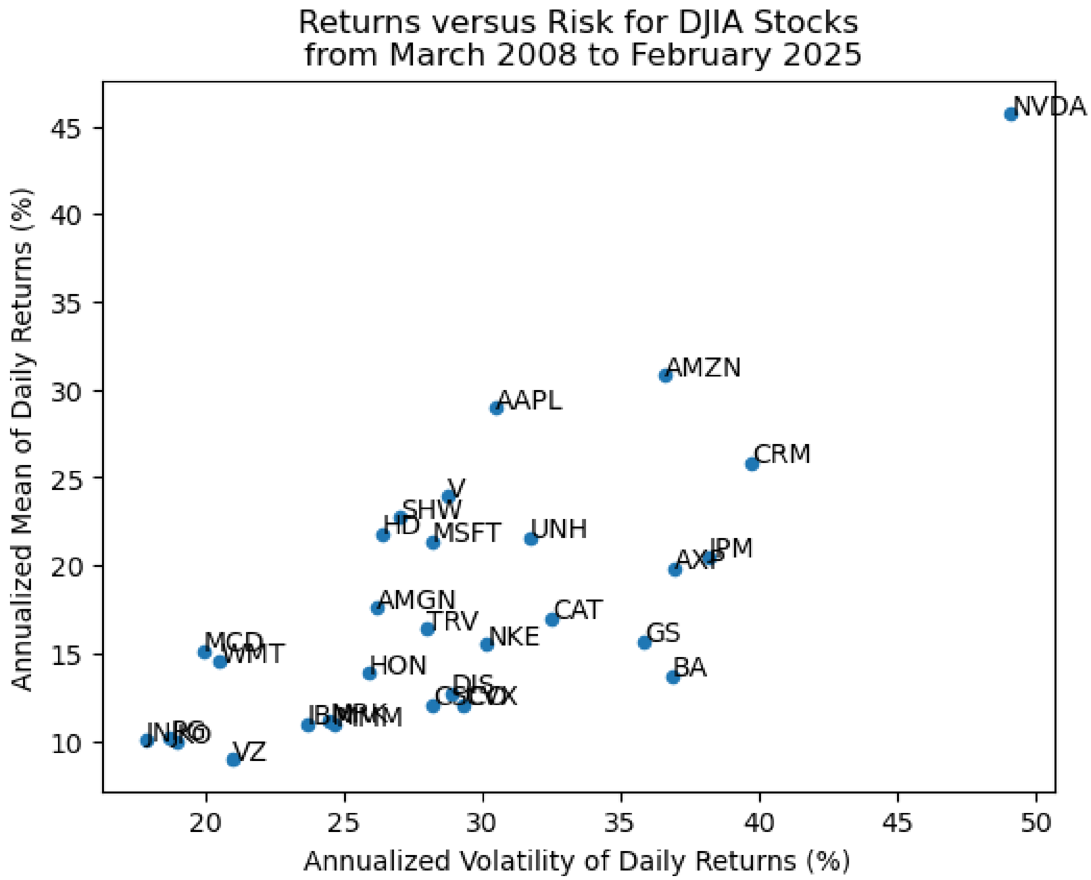
```
dates_2[0]
```

```
Timestamp('2008-03-20 00:00:00')
```

```
dates_2[-1]
```

```
Timestamp('2025-02-27 00:00:00')
```

```
ax = df_2.plot(kind='scatter', x='Volatility', y='Mean Return')

for i, (v, mr) in df_2.iterrows():
    plt.text(s=i, x=v, y=mr)

ax.xaxis.set_major_formatter(FuncFormatter(lambda x, _: f'{x*100:,.0f}'))
ax.yaxis.set_major_formatter(FuncFormatter(lambda x, _: f'{x*100:,.0f}'))

plt.xlabel('Annualized Volatility of Daily Returns (%)')
plt.ylabel('Annualized Mean of Daily Returns (%)')

plt.title(f'Returns versus Risk for DJIA Stocks\n from {dates_2[0]:%B %Y} to {dates_2[-1]:%B
plt.show()
```

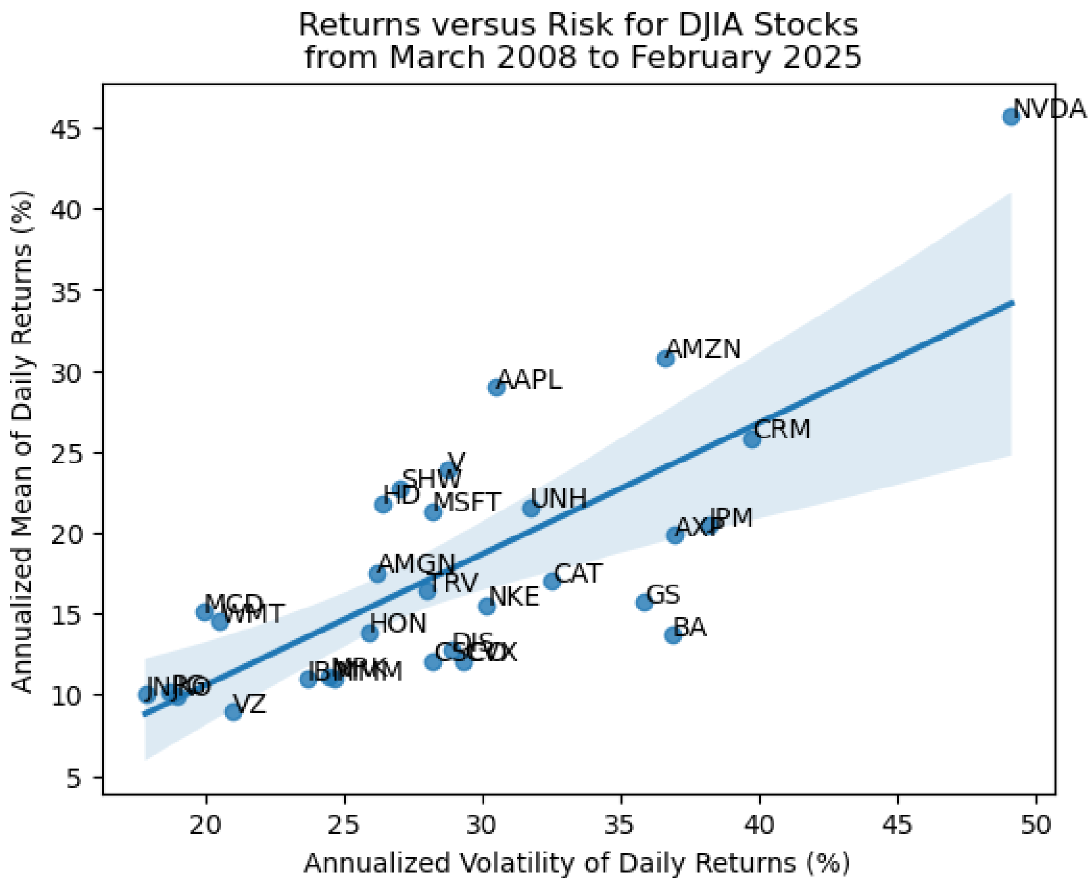


We can use the seaborn package to add a best-fit line! More on seaborn here: https://seaborn.pydata.org/index.html

```
import seaborn as sns
```

```
ax = sns.regplot(data=df_2, x='Volatility', y='Mean Return')

for i, (v, mr) in df_2.iterrows():
    plt.text(s=i, x=v, y=mr)

ax.xaxis.set_major_formatter(FuncFormatter(lambda x, _: f'{x*100:,.0f}'))
ax.yaxis.set_major_formatter(FuncFormatter(lambda x, _: f'{x*100:,.0f}'))

plt.xlabel('Annualized Volatility of Daily Returns (%)')
plt.ylabel('Annualized Mean of Daily Returns (%)')
```

```
plt.title(f'Returns versus Risk for DJIA Stocks\n from {dates_2[0]:%B %Y} to {dates_2[-1]:%B
plt.show()
```



Calculate total returns for the stocks in the DJIA

We can use the `.prod()` method to compound returns as $1 + R_T = \prod_{t=1}^T (1 + R_t)$. Technically, we should write R_T as $R_{0,T}$, but we typically omit the subscript 0.

In general, I prefer to do simple math on pandas objects (data frames and series) with methods instead of operators:

For example:

1. `.add(1)` instead of `+ 1`
2. `.sub(1)` instead of `- 1`
3. `.div(1)` instead of `/ 1`

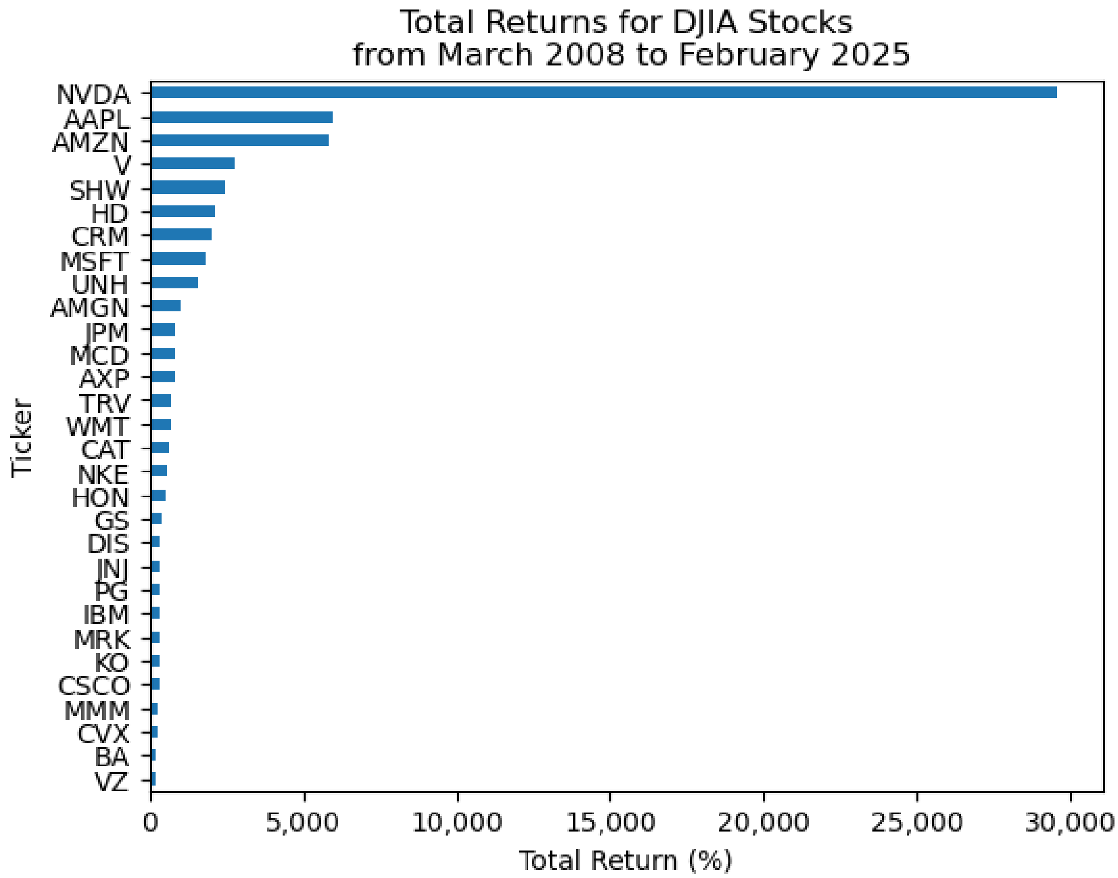
4. `.mul(1)` instead of `* 1`

The advantage of methods over operators, is that we can easily chain methods without lots of parentheses.

We can use the `.clipboard()` method to quickly move data to Excel!

```
returns_2.to_clipboard()
```

```
total_returns_2 = returns_2.add(1).prod().sub(1)

ax = total_returns_2.sort_values().plot(kind='barh')

ax.xaxis.set_major_formatter(FuncFormatter(lambda x, _: f'{x*100:,.0f}'))

plt.xlabel('Total Return (%)')

plt.title(f'Total Returns for DJIA Stocks\n from {dates_2[0]:%B %Y} to {dates_2[-1]:%B %Y}')
plt.show()
```

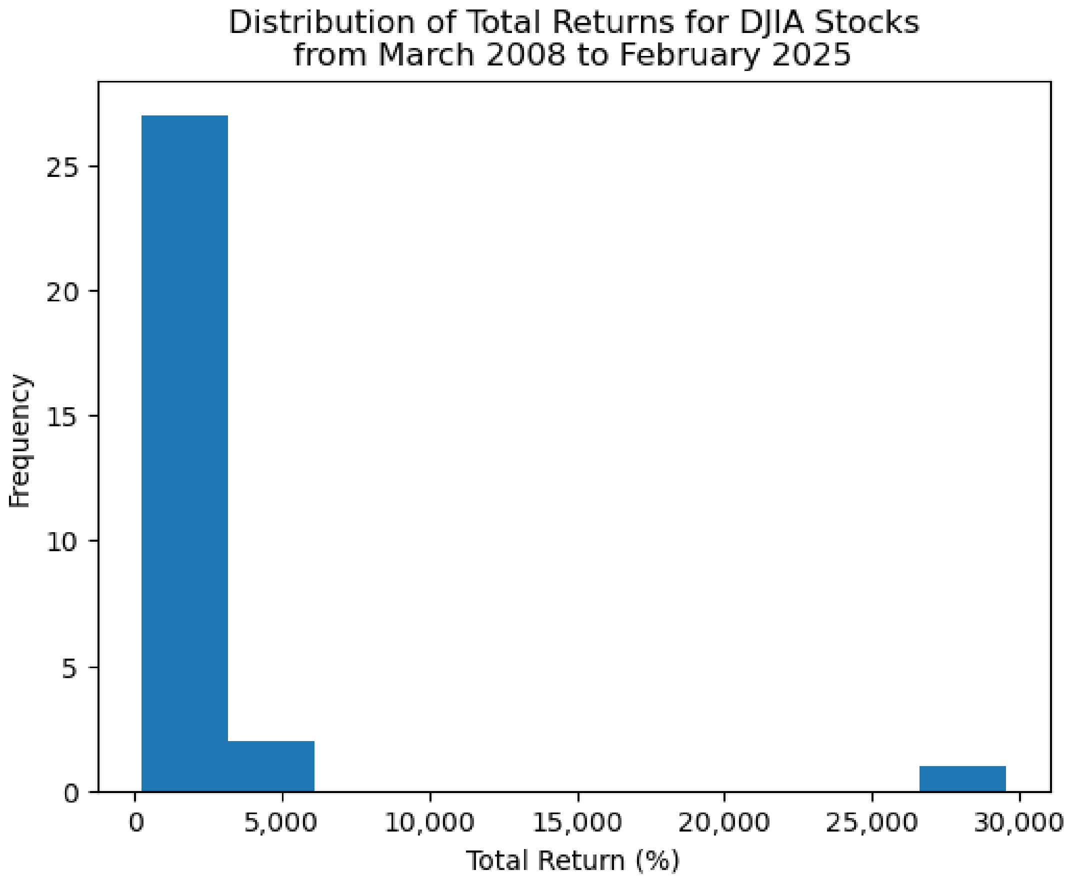


Plot the distribution of total returns for the stocks in the DJIA

We can plot a histogram, using either the `plt.hist()` function or the `.plot(kind='hist')` method.

A histogram is a great way to visualize data!

```
ax = total_returns_2.plot(kind='hist')

ax.xaxis.set_major_formatter(FuncFormatter(lambda x, _: f'{x*100:,.0f}'))

plt.xlabel('Total Return (%)')

plt.title(f'Distribution of Total Returns for DJIA Stocks\nfrom {dates_2[0]:%B %Y} to {dates_
plt.show()
```



Which stocks have the minimum and maximum total returns?

If we want the *values*, the `.min()` and `.max()` methods are the way to go!

```
total_returns_2.min()
```

```
2.1510
```

```
total_returns_2.max()
```

```
295.7450
```

The `.min()` and `.max()` methods give the values but not the tickers (or index). We use the `.idxmin()` and `.idxmax()` to get the tickers (or index).

```python
total_returns_2.idxmin()
```

```
'VZ'
```

```python
total_returns_2.idxmax()
```

```
'NVDA'
```

Here is what I would use to capture values and tickers!

```python
total_returns_2.sort_values().iloc[[0, -1]]
```

```
Ticker
VZ        2.1510
NVDA    295.7450
dtype: float64
```

Not the exactly right tool here, but the `.nsmallest()' and`.nlargest()‘ methods are really useful!

```python
total_returns_2.nsmallest(3)
```

```
Ticker
VZ     2.1510
BA     2.2134
CVX    2.7258
dtype: float64
```

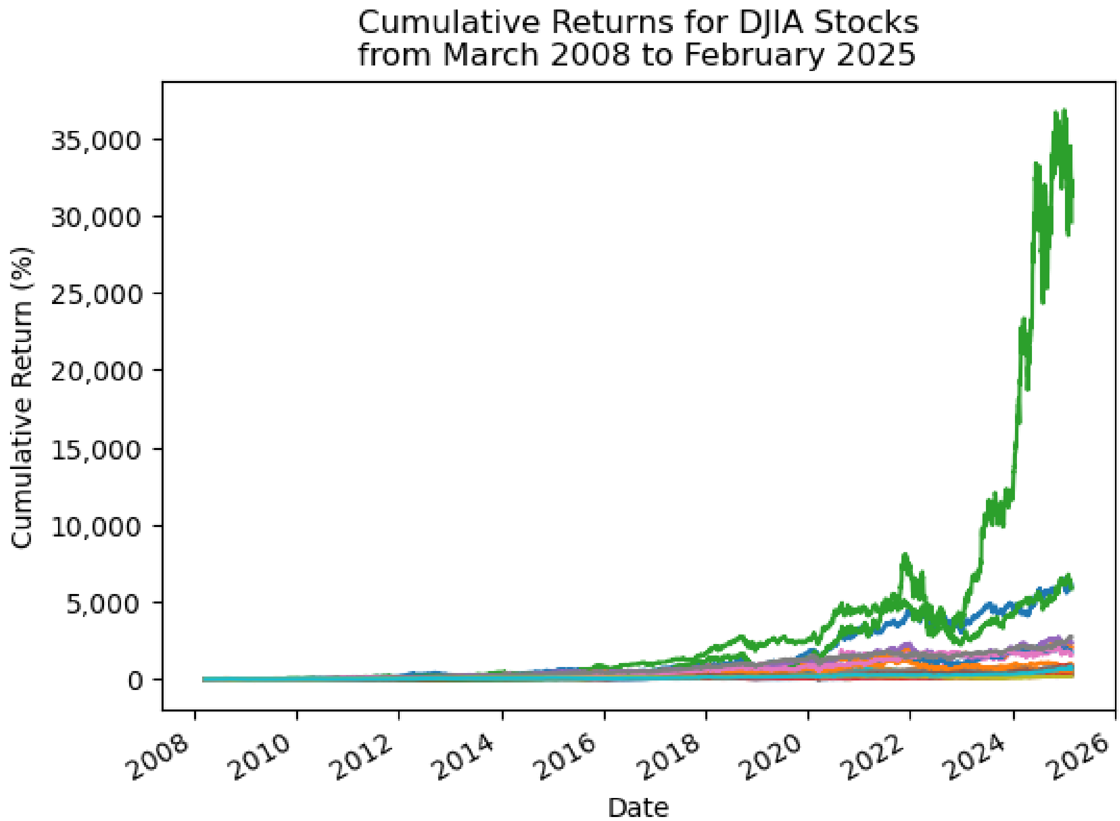
```python
total_returns_2.nlargest(3)
```

```
Ticker
NVDA    295.7450
AAPL     59.8113
AMZN     58.4955
dtype: float64
```

Plot the cumulative returns for the stocks in the DJIA

We can use the cumulative product method `.cumprod()` to calculate the right hand side of the formula above.

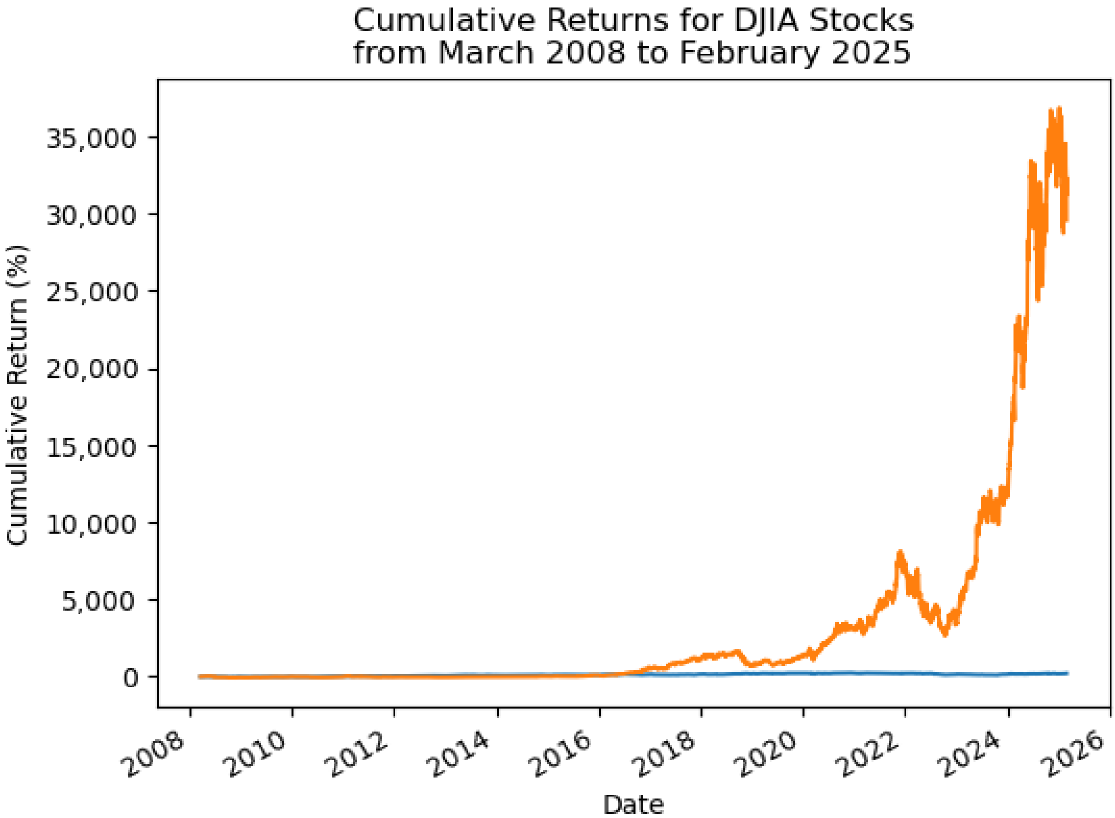
```
ax = (
    returns_2
    .add(1)
    .cumprod()
    .sub(1)
    .plot(legend=False) # with 30 stocks, this legend is too big to be useful
)

ax.yaxis.set_major_formatter(FuncFormatter(lambda x, _: f'{x*100:,.0f}'))

plt.ylabel('Cumulative Return (%)')

plt.title(f'Cumulative Returns for DJIA Stocks\nfrom {dates_2[0]:%B %Y} to {dates_2[-1]:%B %Y
plt.show()
```



Repeat the plot above with only the minimum and maximum total returns

```
ax = (
    returns_2
    [total_returns_2.sort_values().iloc[[0, -1]].index] # slice min and max total return stoc
    .add(1)
    .cumprod()
    .sub(1)
    .plot(legend=False) # with 30 stocks, this legend is too big to be useful
)

ax.yaxis.set_major_formatter(FuncFormatter(lambda x, _: f'{x*100:,.0f}'))

plt.ylabel('Cumulative Return (%)')
```

```
plt.title(f'Cumulative Returns for DJIA Stocks\nfrom {dates_2[0]:%B %Y} to {dates_2[-1]:%B %Y
plt.show()
```



Week 5

McKinney Chapter 8 - Data Wrangling: Join, Combine, and Reshape

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import pandas_datareader as pdr
import yfinance as yf
```

```
%precision 4
pd.options.display.float_format = '{:.4f}'.format
# %config InlineBackend.figure_format = 'retina'
```

Introduction

Chapter 8 of McKinney (2022) introduces a few important pandas concepts:

1. Joining or merging is combining 2+ data frames on 1+ indexes or columns into 1 data frame
2. Reshaping is rearranging a data frame so it has fewer columns and more rows (wide to long) or more columns and fewer rows (long to wide)

Note: Indented block quotes are from McKinney (2022) unless otherwise indicated. The section numbers here differ from McKinney (2022) because we will only discuss some topics.

Hierarchical Indexing

We must learn about hierarchical indexing before we learn about combining and reshaping data. A hierarchical index has two or more levels. For example, we could index rows by ticker and date. Or we could index columns by variable and ticker. Hierarchical indexing helps us work with high-dimensional data in a low-dimensional form.

```
np.random.seed(42)
data = pd.Series(
    data=np.random.randn(9),
    index=[
        ['a', 'a', 'a', 'b', 'b', 'c', 'c', 'd', 'd'],
        [1, 2, 3, 1, 3, 1, 2, 2, 3]
    ]
)

data
```

```
a  1    0.4967
   2   -0.1383
   3    0.6477
b  1    1.5230
   3   -0.2342
c  1   -0.2341
   2    1.5792
d  2    0.7674
   3   -0.4695
dtype: float64
```

We can index this series to subset it.

```
data['b']
```

```
1    1.5230
3   -0.2342
dtype: float64
```

```
data.loc['b']
```

```
1    1.5230
3   -0.2342
dtype: float64
```

```
data['b':'c']
```

```
b  1    1.5230
   3   -0.2342
c  1   -0.2341
   2    1.5792
dtype: float64
```

```
data.loc['b':'c']
```

```
b  1    1.5230
   3   -0.2342
c  1   -0.2341
   2    1.5792
dtype: float64
```

We can subset on the index inner level, too. Here, the `:` slices all values in the outer index, and the `2` slices the three values with `2` indexes.

```
data.loc[:, 2]
```

```
a   -0.1383
c    1.5792
d    0.7674
dtype: float64
```

Here, `data` has a stacked or long format. We have multiple observations for each outer index (letters) with different inner indexes (numbers). We can un-stack `data` to convert the inner index level to columns. New, we have an unstacked or wide format.

```
data.unstack()
```

	1	2	3
a	0.4967	-0.1383	0.6477
b	1.5230	NaN	-0.2342
c	-0.2341	1.5792	NaN
d	NaN	0.7674	-0.4695

We can create a data frame with hierarchical indexes or multi-indexes on rows *and* columns.

```python
frame = pd.DataFrame(
    data=np.arange(12).reshape((4, 3)),
    index=[['a', 'a', 'b', 'b'], [1, 2, 1, 2]],
    columns=[['Ohio', 'Ohio', 'Colorado'], ['Green', 'Red', 'Green']]
)

frame
```

		Ohio	
		Green	Re
a	1	0	1
	2	3	4
b	1	6	7
	2	9	10

We can name these multi-indexes, but index names are not required.

```python
frame.index.names = ['key1', 'key2']
frame.columns.names = ['state', 'color']

frame
```

	state	Ohio
	color	Green
key1	key2	
a	1	0
	2	3
b	1	6
	2	9

Recall that `df[val]` selects the `val` column. Here, `frame` has a multi-index for the columns, so `frame['Ohio']` selects all columns with Ohio as the outer index.

```python
frame['Ohio']
```

	color	Green
key1	key2	
a	1	0
	2	3
b	1	6
	2	9

We can pass a tuple if we only want one column.

```
frame[[('Ohio', 'Green')]]
```

	state	Ohio
	color	Green
key1	key2	
a	1	0
	2	3
b	1	6
	2	9

We must do more work to slice the inner level of the column index.

```
frame.loc[:, (slice(None), 'Green')]
```

	state	Ohio
	color	Green
key1	key2	
a	1	0
	2	3
b	1	6
	2	9

We can use `pd.IndexSlice[:, 'Green']` an alternative to `(slice(None), 'Green')`.

```
frame.loc[:, pd.IndexSlice[:, 'Green']]
```

	state	Ohio
	color	Green
key1	key2	
a	1	0
	2	3
b	1	6
	2	9

Reordering and Sorting Levels

We can swap index levels with the `.swaplevel()` method. The default arguments are `i=-2` and `j=-1`, which swap the two innermost index levels.

```
frame
```

	state	Ohio
	color	Green
key1	key2	
a	1	0
	2	3
b	1	6
	2	9

```
frame.swaplevel().sort_index()
```

	state	Ohio
	color	Green
key2	key1	
1	a	0
	b	6
2	a	3
	b	9

We can use index *names*, too.

```
frame.swaplevel('key1', 'key2').sort_index()
```

key2	state color key1	Ohio Green
1	a	0
	b	6
2	a	3
	b	9

Indexing with a DataFrame's columns

We can convert a column into an index and an index into a column with the `.set_index()` and `.reset_index()` methods.

```
frame = pd.DataFrame({
    'a': range(7),
    'b': range(7, 0, -1),
    'c': ['one', 'one', 'one', 'two', 'two','two', 'two'],
    'd': [0, 1, 2, 0, 1, 2, 3]
})

frame
```

	a	b	c	d
0	0	7	one	0
1	1	6	one	1
2	2	5	one	2
3	3	4	two	0
4	4	3	two	1
5	5	2	two	2
6	6	1	two	3

The `.set_index()` method converts columns to indexes and drops these columns by default.

```
frame2 = frame.set_index(['c', 'd'])

frame2
```

		a	b
c	d		
one	0	0	7
	1	1	6
	2	2	5
two	0	3	4
	1	4	3
	2	5	2
	3	6	1

The `.reset_index()` method drops indexes, adds them as columns by default, and sets an integer index.

```
frame2.reset_index()
```

	c	d	a	b
0	one	0	0	7
1	one	1	1	6
2	one	2	2	5
3	two	0	3	4
4	two	1	4	3
5	two	2	5	2
6	two	3	6	1

Combining and Merging Datasets

pandas provides several methods and functions to combine and merge data. We can typically create the same output with several these methods or functions, but one may be more efficient.

When we want to combine data frames with similar indexes, we will tend to use the `.join()` method. The `.join()` can also combine three or more data frames.

Otherwise, we will use the `.merge()` method or `pd.merge()` function. The `pd.merge()` function is more flexible than the `.join()` method, so we will start with the `pd.merge()` function.

The pandas website provides helpful visualizations.

Database-Style DataFrame Joins

Merge or join operations combine datasets by linking rows using one or more keys. These operations are central to relational databases (e.g., SQL-based). The merge function in pandas is the main entry point for using these algorithms on your data.

```
df1 = pd.DataFrame({'key': ['b', 'b', 'a', 'c', 'a', 'a', 'b'], 'data1': range(7)})
df2 = pd.DataFrame({'key': ['a', 'b', 'd'], 'data2': range(3)})
```

```
df1
```

	key	data1
0	b	0
1	b	1
2	a	2
3	c	3
4	a	4
5	a	5
6	b	6

```
df2
```

	key	data2
0	a	0
1	b	1
2	d	2

```
pd.merge(df1, df2)
```

	key	data1	data2
0	b	0	1
1	b	1	1
2	a	2	0
3	a	4	0
4	a	5	0
5	b	6	1

The default is `how='inner'`, so `pd.merge()` inner joins left and right data frames by default, keeping only rows that appear in both. We can specify `how='outer'`, so `pd.merge()` outer joins left and right data frames, keeping all rows that appear in either.

```
pd.merge(df1, df2, how='outer')
```

	key	data1	data2
0	a	2.0000	0.0000
1	a	4.0000	0.0000
2	a	5.0000	0.0000
3	b	0.0000	1.0000
4	b	1.0000	1.0000
5	b	6.0000	1.0000
6	c	3.0000	NaN
7	d	NaN	2.0000

A `how='left'` merge keeps only rows that appear in the left data frame.

```
pd.merge(df1, df2, how='left')
```

	key	data1	data2
0	b	0	1.0000
1	b	1	1.0000
2	a	2	0.0000
3	c	3	NaN
4	a	4	0.0000
5	a	5	0.0000
6	b	6	1.0000

A `how='right'` merge keeps only rows that appear in the right data frame.

```
pd.merge(df1, df2, how='right')
```

	key	data1	data2
0	a	2.0000	0
1	a	4.0000	0
2	a	5.0000	0

	key	data1	data2
3	b	0.0000	1
4	b	1.0000	1
5	b	6.0000	1
6	d	NaN	2

By default, `pd.merge()` merges on any columns that appear in both data frames.

> `on` : label or list Column or index level names to join on. These must be found in both DataFrames. If `on` is None and not merging on indexes then this defaults to the intersection of the columns in both DataFrames.

Here, `key` is the only common column between `df1` and `df2`. We *should* specify `on='key'` to avoid unexpected results.

```
pd.merge(df1, df2, on='key')
```

	key	data1	data2
0	b	0	1
1	b	1	1
2	a	2	0
3	a	4	0
4	a	5	0
5	b	6	1

We *must* specify `left_on` and `right_on` if our left and right data frames do not have a common column.

```
df3 = pd.DataFrame({'lkey': ['b', 'b', 'a', 'c', 'a', 'a', 'b'], 'data1': range(7)})
df4 = pd.DataFrame({'rkey': ['a', 'b', 'd'], 'data2': range(3)})
```

```
df3
```

	lkey	data1
0	b	0
1	b	1
2	a	2
3	c	3

	lkey	data1
4	a	4
5	a	5
6	b	6

```
df4
```

	rkey	data2
0	a	0
1	b	1
2	d	2

```
# pd.merge(df3, df4) # this code fails/errors because there are not common columns
# MergeError: No common columns to perform merge on. Merge options: left_on=None, right_on=No
```

```
pd.merge(df3, df4, left_on='lkey', right_on='rkey')
```

	lkey	data1	rkey	data2
0	b	0	b	1
1	b	1	b	1
2	a	2	a	0
3	a	4	a	0
4	a	5	a	0
5	b	6	b	1

Here, `pd.merge()` drops row `c` from `df3` and row `d` from `df4` because `pd.merge()` *inner* joins by default. An inner join keeps the intersection of the left and right data frame keys. If we want to keep rows `c` and `d`, we can *outer* join `df3` and `df4` with `how='outer'`.

```
pd.merge(df3, df4, left_on='lkey', right_on='rkey', how='outer')
```

	lkey	data1	rkey	data2
0	a	2.0000	a	0.0000
1	a	4.0000	a	0.0000
2	a	5.0000	a	0.0000

	lkey	data1	rkey	data2
3	b	0.0000	b	1.0000
4	b	1.0000	b	1.0000
5	b	6.0000	b	1.0000
6	c	3.0000	NaN	NaN
7	NaN	NaN	d	2.0000

Many-to-many merges have well-defined, though not necessarily intuitive, behavior.

```
df1 = pd.DataFrame({'key': ['b', 'b', 'a', 'c', 'a', 'b'], 'data1': range(6)})
df2 = pd.DataFrame({'key': ['a', 'b', 'a', 'b', 'd'], 'data2': range(5)})
```

```
df1
```

	key	data1
0	b	0
1	b	1
2	a	2
3	c	3
4	a	4
5	b	5

```
df2
```

	key	data2
0	a	0
1	b	1
2	a	2
3	b	3
4	d	4

```
pd.merge(df1, df2, on='key')
```

	key	data1	data2
0	b	0	1
1	b	0	3

	key	data1	data2
2	b	1	1
3	b	1	3
4	a	2	0
5	a	2	2
6	a	4	0
7	a	4	2
8	b	5	1
9	b	5	3

> Many-to-many joins form the Cartesian product of the rows. Since there were three `b` rows in the left DataFrame and two in the right one, there are six `b` rows in the result. The join method only affects the distinct key values appearing in the result.

Be careful with many-to-many joins! In finance, we do not expect many-to-many joins because we expect at least one of the data frames to have unique observations. *pandas will not warn us if we accidentally perform a many-to-many join instead of a one-to-one or many-to-one join.*

```
# pd.merge(df1, df2, on='key', validate='1:1')
# MergeError: Merge keys are not unique in either left or right dataset; not a one-to-one me
```

We can merge on more than one key. For example, we can merge two data sets on ticker-date pairs or industry-date pairs.

```
left = pd.DataFrame({'key1': ['foo', 'foo', 'bar'],
                     'key2': ['one', 'two', 'one'],
                     'lval': [1, 2, 3]})
right = pd.DataFrame({'key1': ['foo', 'foo', 'bar', 'bar'],
                      'key2': ['one', 'one', 'one', 'two'],
                      'rval': [4, 5, 6, 7]})
```

```
left
```

	key1	key2	lval
0	foo	one	1
1	foo	two	2
2	bar	one	3

```
right
```

	key1	key2	rval
0	foo	one	4
1	foo	one	5
2	bar	one	6
3	bar	two	7

```
pd.merge(left, right, on=['key1', 'key2'], how='outer')
```

	key1	key2	lval	rval
0	bar	one	3.0000	6.0000
1	bar	two	NaN	7.0000
2	foo	one	1.0000	4.0000
3	foo	one	1.0000	5.0000
4	foo	two	2.0000	NaN

When column names overlap between the left and right data frames, `pd.merge()` appends `_x` and `_y` to the left and right versions of the overlapping column names.

```
pd.merge(left, right, on='key1')
```

	key1	key2_x	lval	key2_y	rval
0	foo	one	1	one	4
1	foo	one	1	one	5
2	foo	two	2	one	4
3	foo	two	2	one	5
4	bar	one	3	one	6
5	bar	one	3	two	7

I typically specify the `suffixes` argument to avoid confusion.

```
pd.merge(left, right, on='key1', suffixes=('_left', '_right'))
```

	key1	key2_left	lval	key2_right	rval
0	foo	one	1	one	4
1	foo	one	1	one	5
2	foo	two	2	one	4
3	foo	two	2	one	5
4	bar	one	3	one	6
5	bar	one	3	two	7

I read the `pd.merge()` docstring frequently! ***Table 8-2*** summarizes the commonly used arguments for `pd.merge()`.

- `left`: DataFrame to be merged on the left side.
- `right`: DataFrame to be merged on the right side.
- `how`: One of 'inner', 'outer', 'left', or 'right'; defaults to 'inner'.
- `on`: Column names to join on. Must be found in both DataFrame objects. If not specified and no other join keys given will use the intersection of the column names in left and right as the join keys.
- `left_on`: Columns in left DataFrame to use as join keys.
- `right_on`: Analogous to left_on for left DataFrame.
- `left_index`: Use row index in left as its join key (or keys, if a MultiIndex).
- `right_index`: Analogous to left_index.
- `sort`: Sort merged data lexicographically by join keys; True by default (disable to get better performance in some cases on large datasets).
- `suffixes`: Tuple of string values to append to column names in case of overlap; defaults to ('_x', '_y') (e.g., if 'data' in both DataFrame objects, would appear as 'data_x' and 'data_y' in result).
- `copy`: If False, avoid copying data into resulting data structure in some exceptional cases; by default always copies.
- `indicator`: Adds a special column _merge that indicates the source of each row; values will be 'left_only', 'right_only', or 'both' based on the origin of the joined data in each row.

Merging on Index

If we want to use `pd.merge()` to join on row indexes, we can use the `left_index` and `right_index` arguments.

```
left1 = pd.DataFrame({'key': ['a', 'b', 'a', 'a', 'b', 'c'], 'value': range(6)})
right1 = pd.DataFrame({'group_val': [3.5, 7]}, index=['a', 'b'])
```

```
left1
```

	key	value
0	a	0
1	b	1
2	a	2
3	a	3
4	b	4
5	c	5

```
right1
```

	group_val
a	3.5000
b	7.0000

```
pd.merge(left1, right1, left_on='key', right_index=True, how='outer')
```

	key	value	group_val
0	a	0	3.5000
2	a	2	3.5000
3	a	3	3.5000
1	b	1	7.0000
4	b	4	7.0000
5	c	5	NaN

The index arguments work for hierarchical indexes (multi indexes), too.

```
lefth = pd.DataFrame({'key1': ['Ohio', 'Ohio', 'Ohio', 'Nevada', 'Nevada'],
                      'key2': [2000, 2001, 2002, 2001, 2002],
                      'data': np.arange(5.)})
righth = pd.DataFrame(np.arange(12).reshape((6, 2)),
                      index=[['Nevada', 'Nevada', 'Ohio', 'Ohio', 'Ohio', 'Ohio'],
                             [2001, 2000, 2000, 2000, 2001, 2002]],
                      columns=['event1', 'event2'])
```

```
pd.merge(lefth, righth, left_on=['key1', 'key2'], right_index=True, how='outer')
```

	key1	key2	data	event1	event2
4	Nevada	2000	NaN	2.0000	3.0000
3	Nevada	2001	3.0000	0.0000	1.0000
4	Nevada	2002	4.0000	NaN	NaN
0	Ohio	2000	0.0000	4.0000	5.0000
0	Ohio	2000	0.0000	6.0000	7.0000
1	Ohio	2001	1.0000	8.0000	9.0000
2	Ohio	2002	2.0000	10.0000	11.0000

```
left2 = pd.DataFrame([[1., 2.], [3., 4.], [5., 6.]],
                     index=['a', 'c', 'e'],
                     columns=['Ohio', 'Nevada'])
right2 = pd.DataFrame([[7., 8.], [9., 10.], [11., 12.], [13, 14]],
                      index=['b', 'c', 'd', 'e'],
                      columns=['Missouri', 'Alabama'])
```

If we use both indexes, `pd.merge()` will keep the index.

```
pd.merge(left2, right2, how='outer', left_index=True, right_index=True)
```

	Ohio	Nevada	Missouri	Alabama
a	1.0000	2.0000	NaN	NaN
b	NaN	NaN	7.0000	8.0000
c	3.0000	4.0000	9.0000	10.0000
d	NaN	NaN	11.0000	12.0000
e	5.0000	6.0000	13.0000	14.0000

> DataFrame has a convenient join instance for merging by index. It can also be used to combine together many DataFrame objects having the same or similar indexes but non-overlapping columns.

We can use the `.join()` method if both data frames have similar indexes.

```
left2
```

	Ohio	Nevada
a	1.0000	2.0000
c	3.0000	4.0000
e	5.0000	6.0000

```
right2
```

	Missouri	Alabama
b	7.0000	8.0000
c	9.0000	10.0000
d	11.0000	12.0000
e	13.0000	14.0000

```
left2.join(right2, how='outer')
```

	Ohio	Nevada	Missouri	Alabama
a	1.0000	2.0000	NaN	NaN
b	NaN	NaN	7.0000	8.0000
c	3.0000	4.0000	9.0000	10.0000
d	NaN	NaN	11.0000	12.0000
e	5.0000	6.0000	13.0000	14.0000

The `.join()` method left joins by default. Because the `.join()` method uses indexes, it requires fewer arguments than `.merge()`. The `.join()` method can also accept a list of data frames.

```
another = pd.DataFrame(
    data=[[7., 8.], [9., 10.], [11., 12.], [16., 17.]],
    index=['a', 'c', 'e', 'f'],
    columns=['New York', 'Oregon']
)

another
```

	New York	Oregon
a	7.0000	8.0000
c	9.0000	10.0000
e	11.0000	12.0000
f	16.0000	17.0000

```
left2.join([right2, another])
```

	Ohio	Nevada	Missouri	Alabama	New York	Oregon
a	1.0000	2.0000	NaN	NaN	7.0000	8.0000
c	3.0000	4.0000	9.0000	10.0000	9.0000	10.0000
e	5.0000	6.0000	13.0000	14.0000	11.0000	12.0000

```
left2.join([right2, another], how='outer')
```

	Ohio	Nevada	Missouri	Alabama	New York	Oregon
a	1.0000	2.0000	NaN	NaN	7.0000	8.0000
c	3.0000	4.0000	9.0000	10.0000	9.0000	10.0000
e	5.0000	6.0000	13.0000	14.0000	11.0000	12.0000
b	NaN	NaN	7.0000	8.0000	NaN	NaN
d	NaN	NaN	11.0000	12.0000	NaN	NaN
f	NaN	NaN	NaN	NaN	16.0000	17.0000

Concatenating Along an Axis

The `pd.concat()` function provides a flexible way to combine data frames and series along an axis. I typically use `pd.concat()` to combine:

1. A list of data frames with similar layouts
2. A list of series because series do not have `.join()` or `.merge()` methods

```
s1 = pd.Series([0, 1], index=['a', 'b'])
s2 = pd.Series([2, 3, 4], index=['c', 'd', 'e'])
s3 = pd.Series([5, 6], index=['f', 'g'])
```

```
s1
```

```
a    0
b    1
dtype: int64
```

```
s2
```

```
c    2
d    3
e    4
dtype: int64
```

```
s3
```

```
f    5
g    6
dtype: int64
```

```
pd.concat([s1, s2, s3]) # implicit axis=0
```

```
a    0
b    1
c    2
d    3
e    4
f    5
g    6
dtype: int64
```

```
pd.concat([s1, s2, s3], axis=1) # explicit axis=1
```

	0	1	2
a	0.0000	NaN	NaN
b	1.0000	NaN	NaN
c	NaN	2.0000	NaN
d	NaN	3.0000	NaN
e	NaN	4.0000	NaN

	0	1	2
f	NaN	NaN	5.0000
g	NaN	NaN	6.0000

```
result = pd.concat([s1, s2, s3], keys=['one', 'two', 'three']) # implicit axis=0
result
```

```
one    a    0
       b    1
two    c    2
       d    3
       e    4
three  f    5
       g    6
dtype: int64
```

```
result.unstack(level=0)
```

	one	two	three
a	0.0000	NaN	NaN
b	1.0000	NaN	NaN
c	NaN	2.0000	NaN
d	NaN	3.0000	NaN
e	NaN	4.0000	NaN
f	NaN	NaN	5.0000
g	NaN	NaN	6.0000

```
pd.concat([s1, s2, s3], axis=1, keys=['one', 'two', 'three']) # explicit axis=1
```

	one	two	three
a	0.0000	NaN	NaN
b	1.0000	NaN	NaN
c	NaN	2.0000	NaN
d	NaN	3.0000	NaN
e	NaN	4.0000	NaN
f	NaN	NaN	5.0000
g	NaN	NaN	6.0000

```
df1 = pd.DataFrame(
    data=np.arange(6).reshape(3, 2),
    index=['a', 'b', 'c'],
    columns=['one', 'two']
)
df2 = pd.DataFrame(
    data=5 + np.arange(4).reshape(2, 2),
    index=['a', 'c'],
    columns=['three', 'four']
)
```

```
pd.concat([df1, df2], axis=1, keys=['level1', 'level2'])
```

	level1		level2	
	one	two	three	four
a	0	1	5.0000	6.0000
b	2	3	NaN	NaN
c	4	5	7.0000	8.0000

```
pd.concat([df1, df2], axis=1, keys=['level1', 'level2'], names=['upper', 'lower'])
```

upper	level1		level2	
lower	one	two	three	four
a	0	1	5.0000	6.0000
b	2	3	NaN	NaN
c	4	5	7.0000	8.0000

Reshaping and Pivoting

Above, we briefly explore reshaping data with `.stack()` and `.unstack()`. Here, we more deeply explore reshaping data.

Reshaping with Hierarchical Indexing

Hierarchical indexes (multi-indexes) help reshape data.

There are two primary actions: - stack: This "rotates" or pivots from the columns in the data to the rows - unstack: This pivots from the rows into the columns

```
data = pd.DataFrame(np.arange(6).reshape((2, 3)),
                    index=pd.Index(['Ohio', 'Colorado'], name='state'),
                    columns=pd.Index(['one', 'two', 'three'],
                    name='number'))

data
```

number state	one	two	three
Ohio	0	1	2
Colorado	3	4	5

```
result = data.stack()

result
```

```
state     number
Ohio      one       0
          two       1
          three     2
Colorado  one       3
          two       4
          three     5
dtype: int64
```

```
result.unstack()
```

number state	one	two	three
Ohio	0	1	2
Colorado	3	4	5

```
s1 = pd.Series([0, 1, 2, 3], index=['a', 'b', 'c', 'd'])
s2 = pd.Series([4, 5, 6], index=['c', 'd', 'e'])
data2 = pd.concat([s1, s2], keys=['one', 'two'])

data2
```

```
one  a    0
     b    1
     c    2
     d    3
two  c    4
     d    5
     e    6
dtype: int64
```

Un-stacking may introduce missing values because data frames are rectangular.

```
data2.unstack()
```

	a	b	c	d	e
one	0.0000	1.0000	2.0000	3.0000	NaN
two	NaN	NaN	4.0000	5.0000	6.0000

Stacking drops these missing values by default. However, this behavior may change soon, so check your output!

```
data2.unstack().stack()
```

```
one  a   0.0000
     b   1.0000
     c   2.0000
     d   3.0000
two  c   4.0000
     d   5.0000
     e   6.0000
dtype: float64
```

McKinney provides two more subsections on reshaping data with the `.pivot()` and `.melt()` methods. Unlike, the stacking methods, the pivoting methods can aggregate data and do not require an index. We will skip these additional aggregation methods for now.

McKinney Chapter 8 - Practice - Blank

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import pandas_datareader as pdr
import yfinance as yf
```

```
%precision 4
pd.options.display.float_format = '{:.4f}'.format
# %config InlineBackend.figure_format = 'retina'
```

Announcements

Five-Minute Review

Practice

Download data from Yahoo! Finance for BAC, C, GS, JPM, MS, and PNC and assign to data frame `stocks_wide`.

Reshape `stocks_wide` from wide to long with dates and tickers as row indexes and assign to data frame `stocks_long`.

Add daily returns to both `stocks_wide` and `stocks_long` under the name `Returns`.

Hint: Use `pd.MultiIndex()` to create a multi index for the wide data frame `stocks_wide`.

Download the daily benchmark return factors from Ken French's data library.

Hint: Use the `DataReader()` function in the pandas-datareader package. We imported this package above with the `pdr.` prefix.

Add the daily benchmark return factors to `stocks_wide` and `stocks_long`.

Write a function `download()` that accepts tickers and returns a wide data frame of returns with the daily benchmark return factors.

We can even add a `shape` argument to return a wide or long data frame!

Download earnings per share for the stocks in `stocks_long` and combine to a long data frame `earnings`.

Use the `.earnings_dates` method described here. Use `pd.concat()` to combine the result of each the `.earnings_date` data frames and assign them to a new data frame `earnings`. Name the row indexes `Ticker` and `Date` and swap to match the order of the row index in `stocks_long`.

Combine `earnings` with the returns from `stocks_long`.

Use the `.earnings_dates` method described here. Use `pd.concat()` to combine the result of each the `.earnings_date` data frames and assign them to a new data frame `earnings`. Name the row indexes `Ticker` and `Date` and swap to match the order of the row index in `stocks_long`.

Plot the relation between daily returns and earnings surprises

Repeat the earnings exercise with the S&P 100 stocks

With more data, we can more clearly see the positive relation between earnings surprises and returns!

McKinney Chapter 8 - Practice - Sec 02

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import pandas_datareader as pdr
import yfinance as yf
```

```
%precision 4
pd.options.display.float_format = '{:.4f}'.format
# %config InlineBackend.figure_format = 'retina'
```

Announcements

1. ***The deadline for forming project groups is Tuesday, 2/11.*** That evening, I will create random project groups from the unassigned students.
2. ***The deadline for proposing (and voting on) students' choice topics is Tuesday, 2/25.*** That evening, I will finalize our schedule for the second half of the semester.

Five-Minute Review

Chapter 8 of McKinney covers 3 important topics.

1. ***Hierarchical Indexing:*** Hierarchical indexes (or multi-indexes) organize data at multiple levels instead of just a flat, two-dimensional structure. They help us work with high-dimensional data in a low-dimensional form. For example, we can index rows by multiple levels like `Ticker` and `Date`, or columns by `Variable` and `Ticker`.
2. ***Combining Data:*** We will use three functions and methods to combine datasets on one or more keys. All three offer `inner`, `outer`, `left`, or `right` combinations.
    1. The `pd.merge()` function (or the `.merge()` method) provides the most flexible way to perform database-style joins on data frames.
    2. The `.join()` method combines data frames with similar indexes.

3. The `pd.concat()` function combines similarly-shaped series and data frames.

3. ***Reshaping Data:*** We can reshape data to change its structure, such as pivoting from wide to long format or vice versa. We will most often use the `.stack()` and `.unstack()` methods, which pivot columns to rows and rows to columns, respectively. Laster in the course we will learn about the `.pivot()` method for aggregating data and the `.melt()` method for more advanced reshaping.

Practice

Download data from Yahoo! Finance for BAC, C, GS, JPM, MS, and PNC and assign to data frame `stocks_wide`.

```
stocks_wide = yf.download(tickers='BAC, C, GS, JPM, MS, PNC', auto_adjust=False, progress=Fal
```

```
stocks_wide.tail()
```

Price	Adj Close						Close			
Ticker	BAC	C	GS	JPM	MS	PNC	BAC	C	GS	JI
Date										
2025-02-24	44.4600	78.5400	623.0505	261.3400	129.9700	186.9600	44.4600	78.5400	626.1400	26
2025-02-25	43.9400	78.1400	611.8759	257.4000	129.6000	186.4700	43.9400	78.1400	614.9100	25
2025-02-26	43.9400	79.0700	614.7218	258.7900	131.0500	187.0200	43.9400	79.0700	617.7700	25
2025-02-27	44.1200	78.8700	605.0000	259.0500	129.2400	188.5600	44.1200	78.8700	608.0000	25
2025-02-28	46.1000	79.9500	622.2900	264.6500	133.1100	191.9200	46.1000	79.9500	622.2900	26

Side Note: I do not know of a good variable inspector for JupyterLab. When I need to visually interact with a data frame, I use the `.to_clipboard()` method to copy it to my clipboard, and then I paste it into Excel.

```
stocks_wide.to_clipboard()
```

I can quickly move from Excel to pandas with the `pd.read_clipboard()` function.

Reshape `stocks_wide` from wide to long with dates and tickers as row indexes and assign to data frame `stocks_long`.

We use the `.stack()` method to go from wider to longer, and the `.unstack()` method to go from long longer to wider. Note that we set `future_stack=True` to accep the future default arguments for `.stack()` and suppress the `FutureWarning`. A `FutureWarning` is not an error, just a warning about some expected change that could cause and error in the future.

```
stocks_long = stocks_wide.stack(future_stack=True)
```

```
stocks_long.tail()
```

Date	Price Ticker	Adj Cl
2025-02-28	C	79.950
	GS	622.29
	JPM	264.65
	MS	133.11
	PNC	191.92

Add daily returns to both `stocks_wide` and `stocks_long` under the name `Returns`.

Hint: Use `pd.MultiIndex()` to create a multi index for the wide data frame `stocks_wide`.

We can use create a multi-index from all the combinations of `['Returns']` and the tickers in `stocks_wide['Adj Close']`! This approach is much easier than manually creating all the variable-ticker pairs we would need, even with only size tickers!

We assign this multi-index to the variable _ (underscore). We only assign and use _ in the same code cell, so we do not have to worry about accidentally using it elsewhere in the notebook.

```
_ = pd.MultiIndex.from_product([['Returns'], stocks_wide['Adj Close'].columns])

stocks_wide[_] = (
    stocks_wide
    ['Adj Close']
    .iloc[:-1] # do not use mid-day Adj Close for returns calculation
    .pct_change()
)
```

```
stocks_wide.tail()
```

Price	Adj Close						Close			
Ticker	BAC	C	GS	JPM	MS	PNC	BAC	C	GS	JI
Date										
2025-02-24	44.4600	78.5400	623.0505	261.3400	129.9700	186.9600	44.4600	78.5400	626.1400	26
2025-02-25	43.9400	78.1400	611.8759	257.4000	129.6000	186.4700	43.9400	78.1400	614.9100	25
2025-02-26	43.9400	79.0700	614.7218	258.7900	131.0500	187.0200	43.9400	79.0700	617.7700	25
2025-02-27	44.1200	78.8700	605.0000	259.0500	129.2400	188.5600	44.1200	78.8700	608.0000	25
2025-02-28	46.1000	79.9500	622.2900	264.6500	133.1100	191.9200	46.1000	79.9500	622.2900	26

To add returns to `stocks_long` we have two options. I prefer the first option, but I will present the second option to show an application of the `.join()` method. I will assign the results of these two options to `stocks_long_1` and `stocks_long_2` so we can keep the original `stocks_long` as-is.

Option 1: Re-use `stocks_wide`!

```
stocks_long_1 = stocks_wide.stack(future_stack=True)
```

Recall, we omitted returns for the most recent trading day, which could include a partial data return.

```
stocks_long_1.tail(12)
```

	Price	Adj Cl
Date	Ticker	
2025-02-27	BAC	44.120
	C	78.870
	GS	605.00
	JPM	259.05
	MS	129.24
	PNC	188.56
2025-02-28	BAC	46.100
	C	79.950
	GS	622.29
	JPM	264.65
	MS	133.11
	PNC	191.92

	Price	Adj Cl
Date	Ticker	

Option 2: Use `.join()`!

Here we will use `_` as a temporary variable for our long data frame with daily returns.

```
_ = (
    stocks_wide
    ['Adj Close']
    .iloc[:-1]
    .pct_change()
    .stack(future_stack=True)
    .to_frame('Returns')
)

stocks_long_2 = stocks_long.join(_)
```

Recall, we omitted returns for the most recent trading day, which could include a partial data return.

```
stocks_long_2.tail(12)
```

		Adj Cl
Date	Ticker	
2025-02-27	BAC	44.120
	C	78.870
	GS	605.00
	JPM	259.05
	MS	129.24
	PNC	188.56
2025-02-28	BAC	46.100
	C	79.950
	GS	622.29
	JPM	264.65
	MS	133.11
	PNC	191.92

We can test the equality of `stocks_long_1` and `stocks_long_2` most easily with the `.equals()` method.

```
stocks_long_1.equals(stocks_long_2)
```

```
True
```

Download the daily benchmark return factors from Ken French's data library.

Hint: Use the `DataReader()` function in the pandas-datareader package. We imported this package above with the `pdr.` prefix.

I often cannot remember the exact name for the daily factors. We can use the `pdr.famafrench.get_available_datasets()` to list all the data in Kenneth French's data library.

```
pdr.famafrench.get_available_datasets()[:5]
```

```
['F-F_Research_Data_Factors',
 'F-F_Research_Data_Factors_weekly',
 'F-F_Research_Data_Factors_daily',
 'F-F_Research_Data_5_Factors_2x3',
 'F-F_Research_Data_5_Factors_2x3_daily']
```

I set `start=1900` to keep all available data. Otherwise, `pdr.DataReader()` keeps only the most recent five years in any data set.

```
ff = pdr.DataReader(
    name='F-F_Research_Data_Factors_daily',
    data_source='famafrench',
    start='1900'
)
```

```
C:\Users\r.herron\AppData\Local\Temp\ipykernel_14108\875599436.py:1: FutureWarning: The argu
  ff = pdr.DataReader(
```

```
type(ff)
```

```
dict
```

The daily factors only have one data frame (in the `0` key) and the data set description (in the `DESCR` key).

```
ff.keys()
```

```
dict_keys([0, 'DESCR'])
```

> **Note**
>
> Data from the Kenneth French data library are *percent* returns instead of *decimal* returns!

```
ff[0]
```

	Mkt-RF	SMB	HML	RF
Date				
1926-07-01	0.1000	-0.2500	-0.2700	0.0090
1926-07-02	0.4500	-0.3300	-0.0600	0.0090
1926-07-06	0.1700	0.3000	-0.3900	0.0090
1926-07-07	0.0900	-0.5800	0.0200	0.0090
1926-07-08	0.2100	-0.3800	0.1900	0.0090
...	...	...	...	...
2024-12-24	1.1100	-0.0900	-0.0500	0.0170
2024-12-26	0.0200	1.0400	-0.1900	0.0170
2024-12-27	-1.1700	-0.6600	0.5600	0.0170
2024-12-30	-1.0900	0.1200	0.7400	0.0170
2024-12-31	-0.4600	0.0000	0.7100	0.0170

```
print(ff['DESCR'])
```

```
F-F Research Data Factors daily
-------------------------------

This file was created by CMPT_ME_BEME_RETS_DAILY using the 202412 CRSP database. The Tbill re

  0 : (25901 rows x 4 cols)
```

Add the daily benchmark return factors to `stocks_wide` and `stocks_long`.

Since both `ff[0]` and `stocks_long_2` have date indexes, we can easily combine them with the `.join()` method.

```
ff[0].tail()
```

	Mkt-RF	SMB	HML	RF
Date				
2024-12-24	1.1100	-0.0900	-0.0500	0.0170
2024-12-26	0.0200	1.0400	-0.1900	0.0170
2024-12-27	-1.1700	-0.6600	0.5600	0.0170
2024-12-30	-1.0900	0.1200	0.7400	0.0170
2024-12-31	-0.4600	0.0000	0.7100	0.0170

```
stocks_long_2.tail(12)
```

		Adj C
Date	Ticker	
2025-02-27	BAC	44.120
	C	78.870
	GS	605.00
	JPM	259.05
	MS	129.24
	PNC	188.56
2025-02-28	BAC	46.100
	C	79.950
	GS	622.29
	JPM	264.65
	MS	133.11
	PNC	191.92

We can quickly combine `stocks_long_2` and `ff[0]` because both have indexes with daily dates named `Date`. Two notes:

1. The `.join()` method left joins by default, so the combined output has only dates in `stocks_long_2`
2. Kenneth French provides *percent* returns, so we divide them by 100 to convert them to *decimal* returns to match our Yahoo! Finance data

```
stocks_long_2.join(ff[0].div(100))
```

Date	Ticker	Adj Cl
1973-02-21	BAC	1.5426
	C	NaN
	GS	NaN
	JPM	NaN
	MS	NaN
...	...	...
2025-02-28	C	79.950
	GS	622.29
	JPM	264.65
	MS	133.11
	PNC	191.92

We could instead convert the Yahoo! Finance *decimal* returns to *percent* returns. I do not have a strong preference on all decimal returns or all percent returns, but all returns should have the same form.

```
(
    stocks_long_2
    .assign(Returns=lambda x: 100 * x['Returns'])
    .join(ff[0])
)
```

Date	Ticker	Adj Cl
1973-02-21	BAC	1.5426
	C	NaN
	GS	NaN
	JPM	NaN
	MS	NaN
...	...	...
2025-02-28	C	79.950
	GS	622.29
	JPM	264.65
	MS	133.11
	PNC	191.92

With `stocks_wide`, we have to do a little more work becuase of its column multi-index! We will use the `pd.MultiIndex.from_product()` trick from above.

```
_ = pd.MultiIndex.from_product([['Factors'], ff[0].columns])
stocks_wide[_] = ff[0].div(100)
```

```
stocks_wide.loc[:'2024'].tail()
```

Price	Adj Close						Close			
Ticker	BAC	C	GS	JPM	MS	PNC	BAC	C	GS	JI
Date										
2024-12-24	44.3800	70.5117	579.9144	241.0650	126.2201	192.4933	44.3800	71.0000	582.7900	24
2024-12-26	44.5500	70.8593	578.3621	241.8907	127.1837	193.1777	44.5500	71.3500	581.2300	24
2024-12-27	44.3400	70.5117	573.3370	239.9308	125.9221	191.6999	44.3400	71.0000	576.1800	24
2024-12-30	43.9100	69.9059	570.7200	238.0904	124.9188	190.9560	43.9100	70.3900	573.5500	23
2024-12-31	43.9500	69.9059	569.7946	238.4783	124.8890	191.2734	43.9500	70.3900	572.6200	23

Write a function `download()` that accepts tickers and returns a wide data frame of returns with the daily benchmark return factors.

We can even add a `shape` argument to return a wide or long data frame!

```
import warnings

def download(tickers, shape='wide'):
    """
    Download stock price data and Fama-French factors, returning in either 'wide' or 'long' f

    Parameters:
    - tickers (str or list of str): Stock ticker(s) to download.
    - shape (str): Output format, either 'wide' (default) or 'long'.

    Returns:
    - pd.DataFrame: A DataFrame containing stock prices, returns, and Fama-French factors.
    """

    # shape must be wide or long
    if shape not in ['wide', 'long']:
```

```
        raise ValueError('Invalid shape: must be "wide" or "long".')

    # Download stock data
    stocks = yf.download(tickers=tickers, auto_adjust=False, progress=False)

    # Download Fama-French factors
    # (suppressing FutureWarning for 'date_parser')
    with warnings.catch_warnings():
        warnings.simplefilter('ignore', category=FutureWarning)
        factors = pdr.DataReader(
            name='F-F_Research_Data_Factors_daily',
            data_source='famafrench',
            start='1900'
        )[0].div(100) # Convert percentages to decimals

    # Multi-index case
    if isinstance(stocks.columns, pd.MultiIndex):
        # Compute daily returns
        _ = pd.MultiIndex.from_product([['Returns'], stocks['Adj Close'].columns])
        stocks[_] = stocks['Adj Close'].pct_change()

        if shape == 'wide':
            # Add factors with multi-index
            _ = pd.MultiIndex.from_product([['Factors'], factors.columns])
            stocks[_] = factors
            return stocks

        # Convert to long format then add factors
        else:
            return stocks.stack(future_stack=True).join(factors)

    # Single index case
    # (redundant with recent versions of yfinance that always return a multi-index)
    stocks['Returns'] = stocks['Adj Close'].pct_change()
    return stocks.join(factors)
```

```
download(tickers='AAPL TSLA')
```

Price	Adj Close		Close		High		Low		Open
Ticker	AAPL	TSLA	AAPL	TSLA	AAPL	TSLA	AAPL	TSLA	AAPL
Date									
1980-12-12	0.0987	NaN	0.1283	NaN	0.1289	NaN	0.1283	NaN	0.1283
1980-12-15	0.0936	NaN	0.1217	NaN	0.1222	NaN	0.1217	NaN	0.1222
1980-12-16	0.0867	NaN	0.1127	NaN	0.1133	NaN	0.1127	NaN	0.1133
1980-12-17	0.0889	NaN	0.1155	NaN	0.1161	NaN	0.1155	NaN	0.1155
1980-12-18	0.0914	NaN	0.1189	NaN	0.1194	NaN	0.1189	NaN	0.1189
...	...	...	...	...	...	...	...	...	...
2025-02-24	247.1000	330.5300	247.1000	330.5300	248.8600	342.4000	244.4200	324.7000	244.9300
2025-02-25	247.0400	302.8000	247.0400	302.8000	250.0000	328.8900	244.9100	297.2500	248.0000
2025-02-26	240.3600	290.8000	240.3600	290.8000	244.9800	309.0000	239.1300	288.0400	244.3300
2025-02-27	237.3000	281.9500	237.3000	281.9500	242.4600	297.2300	237.0600	280.8800	239.4100
2025-02-28	241.8400	292.9800	241.8400	292.9800	242.0900	293.8800	230.2000	273.6000	236.9500

> **Note**
>
> The `yfinance` package is a powerful tool for downloading market data, financial statements, and analyst estimates from Yahoo! Finance.
> However, because `yfinance` relies on Yahoo! Finance's API, changes to the API can disrupt its functionality.
> Recently, Yahoo! Finance changed API access to earnings forecasts and announcement dates, so we **cannot complete the earnings announcement exercise I had planned**.
> Instead, I will prepare an alternative set of exercises for us to work on in class on Friday.
> Thank you for your flexibility!

Download earnings per share for the stocks in `stocks_long` and combine to a long data frame `earnings`.

Use the `.earnings_dates` method described here. Use `pd.concat()` to combine the result of each the `.earnings_date` data frames and assign them to a new data frame `earnings`. Name the row indexes `Ticker` and `Date` and swap to match the order of the row index in `stocks_long`.

Combine `earnings` with the returns from `stocks_long`.

Use the `.earnings_dates` method described here. Use `pd.concat()` to combine the result of each the `.earnings_date` data frames and assign them to a new data frame `earnings`.

Name the row indexes `Ticker` and `Date` and swap to match the order of the row index in `stocks_long`.

Plot the relation between daily returns and earnings surprises

Repeat the earnings exercise with the S&P 100 stocks

With more data, we can more clearly see the positive relation between earnings surprises and returns!

McKinney Chapter 8 - Practice - Sec 03

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import pandas_datareader as pdr
import yfinance as yf
```

```
%precision 4
pd.options.display.float_format = '{:.4f}'.format
# %config InlineBackend.figure_format = 'retina'
```

Announcements

1. ***The deadline for forming project groups is Tuesday, 2/11.*** That evening, I will create random project groups from the unassigned students.
2. ***The deadline for proposing (and voting on) students' choice topics is Tuesday, 2/25.*** That evening, I will finalize our schedule for the second half of the semester.

Five-Minute Review

Chapter 8 of McKinney covers 3 important topics.

1. ***Hierarchical Indexing:*** Hierarchical indexes (or multi-indexes) organize data at multiple levels instead of just a flat, two-dimensional structure. They help us work with high-dimensional data in a low-dimensional form. For example, we can index rows by multiple levels like `Ticker` and `Date`, or columns by `Variable` and `Ticker`.
2. ***Combining Data:*** We will use three functions and methods to combine datasets on one or more keys. All three offer `inner`, `outer`, `left`, or `right` combinations.
    1. The `pd.merge()` function (or the `.merge()` method) provides the most flexible way to perform database-style joins on data frames.
    2. The `.join()` method combines data frames with similar indexes.

  3. The `pd.concat()` function combines similarly-shaped series and data frames.

3. ***Reshaping Data:*** We can reshape data to change its structure, such as pivoting from wide to long format or vice versa. We will most often use the `.stack()` and `.unstack()` methods, which pivot columns to rows and rows to columns, respectively. Laster in the course we will learn about the `.pivot()` method for aggregating data and the `.melt()` method for more advanced reshaping.

Practice

Download data from Yahoo! Finance for BAC, C, GS, JPM, MS, and PNC and assign to data frame `stocks_wide`.

```
stocks_wide = yf.download(tickers='BAC, C, GS, JPM, MS, PNC', auto_adjust=False, progress=Fal
```

```
stocks_wide.tail()
```

Price	Adj Close						Close			
Ticker	BAC	C	GS	JPM	MS	PNC	BAC	C	GS	JI
Date										
2025-02-24	44.4600	78.5400	623.0505	261.3400	129.9700	186.9600	44.4600	78.5400	626.1400	26
2025-02-25	43.9400	78.1400	611.8759	257.4000	129.6000	186.4700	43.9400	78.1400	614.9100	25
2025-02-26	43.9400	79.0700	614.7218	258.7900	131.0500	187.0200	43.9400	79.0700	617.7700	25
2025-02-27	44.1200	78.8700	605.0000	259.0500	129.2400	188.5600	44.1200	78.8700	608.0000	25
2025-02-28	46.1000	79.9500	622.2900	264.6500	133.1100	191.9200	46.1000	79.9500	622.2900	26

Reshape `stocks_wide` from wide to long with dates and tickers as row indexes and assign to data frame `stocks_long`.

We use the `.stack()` method to go from wider to longer, and the `.unstack()` method to go from long longer to wider. Note that we set `future_stack=True` to accep the future default arguments for `.stack()` and suppress the `FutureWarning`. A `FutureWarning` is not an error, just a warning about some expected change that could cause and error in the future.

```
stocks_long = stocks_wide.stack(future_stack=True)
```

```
stocks_long.tail()
```

	Price	Adj C
Date	Ticker	
2025-02-28	C	79.950
	GS	622.29
	JPM	264.65
	MS	133.11
	PNC	191.92

> **Note**
>
> The `.melt()` methods can reshape data frames from wide to long. However, our data frame has a column multi-index, which makes `.melt()` difficult to use and `.stack()` a better option.

Add daily returns to both `stocks_wide` and `stocks_long` under the name `Returns`.

Hint: Use `pd.MultiIndex()` to create a multi index for the wide data frame `stocks_wide`.

```
stocks_wide['Adj Close'].columns
```

```
Index(['BAC', 'C', 'GS', 'JPM', 'MS', 'PNC'], dtype='object', name='Ticker')
```

```
_ = pd.MultiIndex.from_product([['Returns'], stocks_wide['Adj Close'].columns])

stocks_wide[_] = (
    stocks_wide
    ['Adj Close']
    .iloc[:-1] # do not use mid-day Adj Close for returns calculation
    .pct_change()
)
```

```
stocks_wide.tail()
```

Price	Adj Close						Close			
Ticker	BAC	C	GS	JPM	MS	PNC	BAC	C	GS	JI
Date										
2025-02-24	44.4600	78.5400	623.0505	261.3400	129.9700	186.9600	44.4600	78.5400	626.1400	26
2025-02-25	43.9400	78.1400	611.8759	257.4000	129.6000	186.4700	43.9400	78.1400	614.9100	25
2025-02-26	43.9400	79.0700	614.7218	258.7900	131.0500	187.0200	43.9400	79.0700	617.7700	25
2025-02-27	44.1200	78.8700	605.0000	259.0500	129.2400	188.5600	44.1200	78.8700	608.0000	25
2025-02-28	46.1000	79.9500	622.2900	264.6500	133.1100	191.9200	46.1000	79.9500	622.2900	26

To add returns to `stocks_long` we have two options. I prefer the first option, but I will present the second option to show an application of the `.join()` method. I will assign the results of these two options to `stocks_long_1` and `stocks_long_2` so we can keep the original `stocks_long` as-is.

Option 1: Make `stocks_wide` long!

```
stocks_long_1 = stocks_wide.stack(future_stack=True)
```

Recall, we omitted returns for the most recent trading day, which could include a partial data return.

```
stocks_long_1.tail(12)
```

	Price	Adj Cl
Date	Ticker	
2025-02-27	BAC	44.120
	C	78.870
	GS	605.00
	JPM	259.05
	MS	129.24
	PNC	188.56
2025-02-28	BAC	46.100
	C	79.950
	GS	622.29
	JPM	264.65
	MS	133.11
	PNC	191.92

Option 2: Calculate returns from `stocks_wide`, make them long, then `.join()` them to `stocks_long`!

```
_ = stocks_wide['Adj Close'].iloc[:-1].pct_change().stack().to_frame('Returns')

stocks_long_2 = stocks_long.join(_)
```

Recall, we omitted returns for the most recent trading day, which could include a partial data return.

```
stocks_long_2.tail(12)
```

		Adj C
Date	Ticker	
2025-02-27	BAC	44.120
	C	78.870
	GS	605.00
	JPM	259.05
	MS	129.24
	PNC	188.56
2025-02-28	BAC	46.100
	C	79.950
	GS	622.29
	JPM	264.65
	MS	133.11
	PNC	191.92

We can test the equality of `stocks_long_1` and `stocks_long_2` most easily with the `.equals()` method.

```
stocks_long_1.equals(stocks_long_2)
```

```
True
```

Download the daily benchmark return factors from Ken French's data library.

Hint: Use the `DataReader()` function in the pandas-datareader package. We imported this package above with the `pdr.` prefix.

I often cannot remember the exact name for the daily factors. We can use the `pdr.famafrench.get_available_datasets()` to list all the data in Kenneth French's data library.

```
pdr.famafrench.get_available_datasets()[:5]
```

```
['F-F_Research_Data_Factors',
 'F-F_Research_Data_Factors_weekly',
 'F-F_Research_Data_Factors_daily',
 'F-F_Research_Data_5_Factors_2x3',
 'F-F_Research_Data_5_Factors_2x3_daily']
```

```
ff = pdr.DataReader(
    name='F-F_Research_Data_Factors_daily',
    data_source='famafrench',
    start='1900'
)
```

```
C:\Users\r.herron\AppData\Local\Temp\ipykernel_15448\875599436.py:1: FutureWarning: The argu
  ff = pdr.DataReader(
```

```
type(ff)
```

```
dict
```

The daily factors only have one data frame (in the `0` key) and the data set description (in the `DESCR` key).

```
ff.keys()
```

```
dict_keys([0, 'DESCR'])
```

> **Note**
>
> Data from the Kenneth French data library are *percent* returns instead of *decimal* returns!

```
ff[0]
```

Date	Mkt-RF	SMB	HML	RF
1926-07-01	0.1000	-0.2500	-0.2700	0.0090
1926-07-02	0.4500	-0.3300	-0.0600	0.0090
1926-07-06	0.1700	0.3000	-0.3900	0.0090
1926-07-07	0.0900	-0.5800	0.0200	0.0090
1926-07-08	0.2100	-0.3800	0.1900	0.0090
...	...	...	...	...
2024-12-24	1.1100	-0.0900	-0.0500	0.0170
2024-12-26	0.0200	1.0400	-0.1900	0.0170
2024-12-27	-1.1700	-0.6600	0.5600	0.0170
2024-12-30	-1.0900	0.1200	0.7400	0.0170
2024-12-31	-0.4600	0.0000	0.7100	0.0170

```
print(ff['DESCR'])
```

```
F-F Research Data Factors daily
-------------------------------

This file was created by CMPT_ME_BEME_RETS_DAILY using the 202412 CRSP database. The Tbill r

  0 : (25901 rows x 4 cols)
```

Add the daily benchmark return factors to `stocks_wide` and `stocks_long`.

Since both `ff[0]` and `stocks_long_2` have date indexes, we can easily combine them with the `.join()` method.

```
ff[0].tail()
```

Date	Mkt-RF	SMB	HML	RF
2024-12-24	1.1100	-0.0900	-0.0500	0.0170
2024-12-26	0.0200	1.0400	-0.1900	0.0170
2024-12-27	-1.1700	-0.6600	0.5600	0.0170
2024-12-30	-1.0900	0.1200	0.7400	0.0170
2024-12-31	-0.4600	0.0000	0.7100	0.0170

```
stocks_long_2.tail(12)
```

Date	Ticker	Adj Cl
2025-02-27	BAC	44.120
	C	78.870
	GS	605.00
	JPM	259.05
	MS	129.24
	PNC	188.56
2025-02-28	BAC	46.100
	C	79.950
	GS	622.29
	JPM	264.65
	MS	133.11
	PNC	191.92

We can quickly combine `stocks_long_2` and `ff[0]` because both have indexes with daily dates named `Date`. Two notes:

1. The `.join()` method left joins by default, so the combined output has only dates in `stocks_long_2`
2. Kenneth French provides *percent* returns, so we divide them by 100 to convert them to *decimal* returns to match our Yahoo! Finance data

```
stocks_long_2.join(ff[0].div(100))
```

Date	Ticker	Adj Cl
1973-02-21	BAC	1.5426
	C	NaN
	GS	NaN
	JPM	NaN
	MS	NaN
...	...	...
2025-02-28	C	79.950
	GS	622.29
	JPM	264.65
	MS	133.11

Date	Ticker	Adj Cl
	PNC	191.92

We could instead convert the Yahoo! Finance *decimal* returns to *percent* returns. I do not have a strong preference on all decimal returns or all percent returns, but all returns should have the same form.

```
(
    stocks_long_2
    .assign(Returns=lambda x: 100 * x['Returns'])
    .join(ff[0])
)
```

Date	Ticker	Adj Cl
1973-02-21	BAC	1.5426
	C	NaN
	GS	NaN
	JPM	NaN
	MS	NaN
...	...	...
2025-02-28	C	79.950
	GS	622.29
	JPM	264.65
	MS	133.11
	PNC	191.92

With `stocks_wide`, we have to do a little more work becuase of its column multi-index! We will use the `pd.MultiIndex.from_product()` trick from above.

```
_ = pd.MultiIndex.from_product([['Factors'], ff[0].columns])
stocks_wide[_] = ff[0].div(100)
```

```
stocks_wide.loc[:'2024'].tail()
```

Price	Adj Close						Close			
Ticker	BAC	C	GS	JPM	MS	PNC	BAC	C	GS	JI
Date										
2024-12-24	44.3800	70.5117	579.9144	241.0650	126.2201	192.4933	44.3800	71.0000	582.7900	24
2024-12-26	44.5500	70.8593	578.3621	241.8907	127.1837	193.1777	44.5500	71.3500	581.2300	24
2024-12-27	44.3400	70.5117	573.3370	239.9308	125.9221	191.6999	44.3400	71.0000	576.1800	24
2024-12-30	43.9100	69.9059	570.7200	238.0904	124.9188	190.9560	43.9100	70.3900	573.5500	23
2024-12-31	43.9500	69.9059	569.7946	238.4783	124.8890	191.2734	43.9500	70.3900	572.6200	23

Write a function `download()` that accepts tickers and returns a wide data frame of returns with the daily benchmark return factors.

We can even add a `shape` argument to return a wide or long data frame!

We can even add a `shape` argument to return a wide or long data frame!

```
import warnings

def download(tickers, shape='wide'):
    """
    Download stock price data and Fama-French factors, returning in either 'wide' or 'long' f

    Parameters:
    - tickers (str or list of str): Stock ticker(s) to download.
    - shape (str): Output format, either 'wide' (default) or 'long'.

    Returns:
    - pd.DataFrame: A DataFrame containing stock prices, returns, and Fama-French factors.
    """

    # shape must be wide or long
    if shape not in ['wide', 'long']:
        raise ValueError('Invalid shape: must be "wide" or "long".')

    # Download stock data
    stocks = yf.download(tickers=tickers, auto_adjust=False, progress=False)

    # Download Fama-French factors
```

```
    # (suppressing FutureWarning for 'date_parser')
    with warnings.catch_warnings():
        warnings.simplefilter('ignore', category=FutureWarning)
        factors = pdr.DataReader(
            name='F-F_Research_Data_Factors_daily',
            data_source='famafrench',
            start='1900'
        )[0].div(100) # Convert percentages to decimals

    # Multi-index case
    if isinstance(stocks.columns, pd.MultiIndex):
        # Compute daily returns
        _ = pd.MultiIndex.from_product([['Returns'], stocks['Adj Close'].columns])
        stocks[_] = stocks['Adj Close'].pct_change()

        if shape == 'wide':
            # Add factors with multi-index
            _ = pd.MultiIndex.from_product([['Factors'], factors.columns])
            stocks[_] = factors
            return stocks

        # Convert to long format then add factors
        else:
            return stocks.stack(future_stack=True).join(factors)

    # Single index case
    # (redundant with recent versions of yfinance that always return a multi-index)
    stocks['Returns'] = stocks['Adj Close'].pct_change()
    return stocks.join(factors)
```

```
download(tickers='AAPL TSLA')
```

Price	Adj Close		Close		High		Low		Open
Ticker	AAPL	TSLA	AAPL	TSLA	AAPL	TSLA	AAPL	TSLA	AAPL
Date									
1980-12-12	0.0987	NaN	0.1283	NaN	0.1289	NaN	0.1283	NaN	0.1283
1980-12-15	0.0936	NaN	0.1217	NaN	0.1222	NaN	0.1217	NaN	0.1222
1980-12-16	0.0867	NaN	0.1127	NaN	0.1133	NaN	0.1127	NaN	0.1133
1980-12-17	0.0889	NaN	0.1155	NaN	0.1161	NaN	0.1155	NaN	0.1155
1980-12-18	0.0914	NaN	0.1189	NaN	0.1194	NaN	0.1189	NaN	0.1189
...	...	...	...	...	...	...	...	...	...

Price	Adj Close		Close		High		Low		Open
Ticker	AAPL	TSLA	AAPL	TSLA	AAPL	TSLA	AAPL	TSLA	AAPL
Date									
2025-02-24	247.1000	330.5300	247.1000	330.5300	248.8600	342.4000	244.4200	324.7000	244.9300
2025-02-25	247.0400	302.8000	247.0400	302.8000	250.0000	328.8900	244.9100	297.2500	248.0000
2025-02-26	240.3600	290.8000	240.3600	290.8000	244.9800	309.0000	239.1300	288.0400	244.3300
2025-02-27	237.3000	281.9500	237.3000	281.9500	242.4600	297.2300	237.0600	280.8800	239.4100
2025-02-28	241.8400	292.9800	241.8400	292.9800	242.0900	293.8800	230.2000	273.6000	236.9500

Note

The `yfinance` package is a powerful tool for downloading market data, financial statements, and analyst estimates from Yahoo! Finance.
However, because `yfinance` relies on Yahoo! Finance's API, changes to the API can disrupt its functionality.
Recently, Yahoo! Finance changed API access to earnings forecasts and announcement dates, so we **cannot complete the earnings announcement exercise I had planned**.
Instead, I will prepare an alternative set of exercises for us to work on in class on Friday.
Thank you for your flexibility!

Download earnings per share for the stocks in `stocks_long` and combine to a long data frame `earnings`.

Use the `.earnings_dates` method described here. Use `pd.concat()` to combine the result of each the `.earnings_date` data frames and assign them to a new data frame `earnings`. Name the row indexes `Ticker` and `Date` and swap to match the order of the row index in `stocks_long`.

Combine `earnings` with the returns from `stocks_long`.

Use the `.earnings_dates` method described here. Use `pd.concat()` to combine the result of each the `.earnings_date` data frames and assign them to a new data frame `earnings`. Name the row indexes `Ticker` and `Date` and swap to match the order of the row index in `stocks_long`.

Plot the relation between daily returns and earnings surprises

Repeat the earnings exercise with the S&P 100 stocks

With more data, we can more clearly see the positive relation between earnings surprises and returns!

McKinney Chapter 8 - Practice - Sec 04

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import pandas_datareader as pdr
import yfinance as yf
```

```
%precision 4
pd.options.display.float_format = '{:.4f}'.format
# %config InlineBackend.figure_format = 'retina'
```

Announcements

1. ***The deadline for forming project groups is Tuesday, 2/11.*** That evening, I will create random project groups from the unassigned students.
2. ***The deadline for proposing (and voting on) students' choice topics is Tuesday, 2/25.*** That evening, I will finalize our schedule for the second half of the semester.

Five-Minute Review

Chapter 8 of McKinney covers 3 important topics.

1. ***Hierarchical Indexing:*** Hierarchical indexes (or multi-indexes) organize data at multiple levels instead of just a flat, two-dimensional structure. They help us work with high-dimensional data in a low-dimensional form. For example, we can index rows by multiple levels like `Ticker` and `Date`, or columns by `Variable` and `Ticker`.
2. ***Combining Data:*** We will use three functions and methods to combine datasets on one or more keys. All three offer `inner`, `outer`, `left`, or `right` combinations.
    1. The `pd.merge()` function (or the `.merge()` method) provides the most flexible way to perform database-style joins on data frames.
    2. The `.join()` method combines data frames with similar indexes.

3. The `pd.concat()` function combines similarly-shaped series and data frames.

3. ***Reshaping Data:*** We can reshape data to change its structure, such as pivoting from wide to long format or vice versa. We will most often use the `.stack()` and `.unstack()` methods, which pivot columns to rows and rows to columns, respectively. Laster in the course we will learn about the `.pivot()` method for aggregating data and the `.melt()` method for more advanced reshaping.

Practice

Download data from Yahoo! Finance for BAC, C, GS, JPM, MS, and PNC and assign to data frame `stocks_wide`.

```
stocks_wide = yf.download(tickers='BAC, C, GS, JPM, MS, PNC', auto_adjust=False, progress=Fal
```

```
stocks_wide.tail()
```

Price	Adj Close						Close			
Ticker	BAC	C	GS	JPM	MS	PNC	BAC	C	GS	JI
Date										
2025-02-24	44.4600	78.5400	623.0505	261.3400	129.9700	186.9600	44.4600	78.5400	626.1400	26
2025-02-25	43.9400	78.1400	611.8759	257.4000	129.6000	186.4700	43.9400	78.1400	614.9100	25
2025-02-26	43.9400	79.0700	614.7218	258.7900	131.0500	187.0200	43.9400	79.0700	617.7700	25
2025-02-27	44.1200	78.8700	605.0000	259.0500	129.2400	188.5600	44.1200	78.8700	608.0000	25
2025-02-28	46.1000	79.9500	622.2900	264.6500	133.1100	191.9200	46.1000	79.9500	622.2900	26

Reshape `stocks_wide` from wide to long with dates and tickers as row indexes and assign to data frame `stocks_long`.

We use the `.stack()` method to go from wider to longer, and the `.unstack()` method to go from long longer to wider. Note that we set `future_stack=True` to accep the future default arguments for `.stack()` and suppress the `FutureWarning`. A `FutureWarning` is not an error, just a warning about some expected change that could cause and error in the future.

```
stocks_long = stocks_wide.stack(future_stack=True)
```

```
stocks_long.tail()
```

Date	Price Ticker	Adj Cl
2025-02-28	C	79.950
	GS	622.29
	JPM	264.65
	MS	133.11
	PNC	191.92

Add daily returns to both `stocks_wide` and `stocks_long` under the name `Returns`.

Hint: Use `pd.MultiIndex()` to create a multi index for the wide data frame `stocks_wide`.

```
stocks_wide['Adj Close'].columns
```

```
Index(['BAC', 'C', 'GS', 'JPM', 'MS', 'PNC'], dtype='object', name='Ticker')
```

```
_ = pd.MultiIndex.from_product([['Returns'], stocks_wide['Adj Close'].columns])

stocks_wide[_] = (
    stocks_wide
    ['Adj Close']
    .iloc[:-1] # do not use mid-day Adj Close for returns calculation
    .pct_change()
)
```

```
stocks_wide.tail()
```

Price	Adj Close						Close			
Ticker	BAC	C	GS	JPM	MS	PNC	BAC	C	GS	JI
Date										
2025-02-24	44.4600	78.5400	623.0505	261.3400	129.9700	186.9600	44.4600	78.5400	626.1400	26
2025-02-25	43.9400	78.1400	611.8759	257.4000	129.6000	186.4700	43.9400	78.1400	614.9100	25
2025-02-26	43.9400	79.0700	614.7218	258.7900	131.0500	187.0200	43.9400	79.0700	617.7700	25
2025-02-27	44.1200	78.8700	605.0000	259.0500	129.2400	188.5600	44.1200	78.8700	608.0000	25

Price	Adj Close						Close			
Ticker	BAC	C	GS	JPM	MS	PNC	BAC	C	GS	JI
Date										
2025-02-28	46.1000	79.9500	622.2900	264.6500	133.1100	191.9200	46.1000	79.9500	622.2900	26

To add returns to `stocks_long` we have two options. I prefer the first option, but I will present the second option to show an application of the `.join()` method. I will assign the results of these two options to `stocks_long_1` and `stocks_long_2` so we can keep the original `stocks_long` as-is.

Option 1: Stacl `stocks_wide`!

```
stocks_long_1 = stocks_wide.stack(future_stack=True)
```

Recall, we omitted returns for the most recent trading day, which could include a partial data return.

```
stocks_long_1.tail(12)
```

	Price	Adj C
Date	Ticker	
2025-02-27	BAC	44.120
	C	78.870
	GS	605.00
	JPM	259.05
	MS	129.24
	PNC	188.56
2025-02-28	BAC	46.100
	C	79.950
	GS	622.29
	JPM	264.65
	MS	133.11
	PNC	191.92

Option 2: Calculate returns, make them long, then join them!

```
_ = stocks_wide['Adj Close'].iloc[:-1].pct_change().stack(future_stack=True).to_frame('Retur
stocks_long_2 = stocks_long.join(_)
```

Recall, we omitted returns for the most recent trading day, which could include a partial data return.

```
stocks_long_2.tail(12)
```

		Adj Cl
Date	Ticker	
2025-02-27	BAC	44.120
	C	78.870
	GS	605.00
	JPM	259.05
	MS	129.24
	PNC	188.56
2025-02-28	BAC	46.100
	C	79.950
	GS	622.29
	JPM	264.65
	MS	133.11
	PNC	191.92

We can test the equality of `stocks_long_1` and `stocks_long_2` most easily with the `.equals()` method.

```
stocks_long_1.equals(stocks_long_2)
```

```
True
```

Download the daily benchmark return factors from Ken French's data library.

Hint: Use the `DataReader()` function in the pandas-datareader package. We imported this package above with the `pdr.` prefix.

I often cannot remember the exact name for the daily factors. We can use the `pdr.famafrench.get_available_datasets()` to list all the data in Kenneth French's data library.

```
pdr.famafrench.get_available_datasets()[:5]
```

```
['F-F_Research_Data_Factors',
 'F-F_Research_Data_Factors_weekly',
 'F-F_Research_Data_Factors_daily',
 'F-F_Research_Data_5_Factors_2x3',
 'F-F_Research_Data_5_Factors_2x3_daily']
```

```
ff = pdr.DataReader(
    name='F-F_Research_Data_Factors_daily',
    data_source='famafrench',
    start='1900'
)
```

```
C:\Users\r.herron\AppData\Local\Temp\ipykernel_28432\875599436.py:1: FutureWarning: The argu
  ff = pdr.DataReader(
```

```
type(ff)
```

```
dict
```

The daily factors only have one data frame (in the `0` key) and the data set description (in the `DESCR` key).

```
ff.keys()
```

```
dict_keys([0, 'DESCR'])
```

> **Note**
>
> Data from the Kenneth French data library are *percent* returns instead of *decimal* returns!

```
ff[0]
```

	Mkt-RF	SMB	HML	RF
Date				
1926-07-01	0.1000	-0.2500	-0.2700	0.0090
1926-07-02	0.4500	-0.3300	-0.0600	0.0090
1926-07-06	0.1700	0.3000	-0.3900	0.0090

Date	Mkt-RF	SMB	HML	RF
1926-07-07	0.0900	-0.5800	0.0200	0.0090
1926-07-08	0.2100	-0.3800	0.1900	0.0090
...	...	...	...	...
2024-12-24	1.1100	-0.0900	-0.0500	0.0170
2024-12-26	0.0200	1.0400	-0.1900	0.0170
2024-12-27	-1.1700	-0.6600	0.5600	0.0170
2024-12-30	-1.0900	0.1200	0.7400	0.0170
2024-12-31	-0.4600	0.0000	0.7100	0.0170

```
print(ff['DESCR'])
```

```
F-F Research Data Factors daily
-------------------------------

This file was created by CMPT_ME_BEME_RETS_DAILY using the 202412 CRSP database. The Tbill re

  0 : (25901 rows x 4 cols)
```

Add the daily benchmark return factors to `stocks_wide` and `stocks_long`.

Since both `ff[0]` and `stocks_long_2` have date indexes, we can easily combine them with the `.join()` method.

```
ff[0].tail()
```

Date	Mkt-RF	SMB	HML	RF
2024-12-24	1.1100	-0.0900	-0.0500	0.0170
2024-12-26	0.0200	1.0400	-0.1900	0.0170
2024-12-27	-1.1700	-0.6600	0.5600	0.0170
2024-12-30	-1.0900	0.1200	0.7400	0.0170
2024-12-31	-0.4600	0.0000	0.7100	0.0170

```
stocks_long_2.tail(12)
```

Date	Ticker	Adj Cl
2025-02-27	BAC	44.120
	C	78.870
	GS	605.00
	JPM	259.05
	MS	129.24
	PNC	188.56
2025-02-28	BAC	46.100
	C	79.950
	GS	622.29
	JPM	264.65
	MS	133.11
	PNC	191.92

We can quickly combine `stocks_long_2` and `ff[0]` because both have indexes with daily dates named `Date`. Two notes:

1. The `.join()` method left joins by default, so the combined output has only dates in `stocks_long_2`
2. Kenneth French provides *percent* returns, so we divide them by 100 to convert them to *decimal* returns to match our Yahoo! Finance data

```
stocks_long_2.join(ff[0].div(100))
```

Date	Ticker	Adj Cl
1973-02-21	BAC	1.5426
	C	NaN
	GS	NaN
	JPM	NaN
	MS	NaN
...	...	...
2025-02-28	C	79.950
	GS	622.29
	JPM	264.65
	MS	133.11
	PNC	191.92

We could instead convert the Yahoo! Finance *decimal* returns to *percent* returns. I do not have a strong preference on all decimal returns or all percent returns, but all returns should have the same form.

```
(
    stocks_long_2
    .assign(Returns=lambda x: 100 * x['Returns'])
    .join(ff[0])
)
```

		Adj Cl
Date	Ticker	
1973-02-21	BAC	1.5426
	C	NaN
	GS	NaN
	JPM	NaN
	MS	NaN
...	...	...
2025-02-28	C	79.950
	GS	622.29
	JPM	264.65
	MS	133.11
	PNC	191.92

With `stocks_wide`, we have to do a little more work becuase of its column multi-index! We will use the `pd.MultiIndex.from_product()` trick from above.

```
_ = pd.MultiIndex.from_product([['Factors'], ff[0].columns])
stocks_wide[_] = ff[0].div(100)
```

```
stocks_wide.loc[:'2024'].tail()
```

Price	Adj Close						Close			
Ticker	BAC	C	GS	JPM	MS	PNC	BAC	C	GS	JI
Date										
2024-12-24	44.3800	70.5117	579.9144	241.0650	126.2201	192.4933	44.3800	71.0000	582.7900	24

Price	Adj Close						Close			
Ticker	BAC	C	GS	JPM	MS	PNC	BAC	C	GS	JI
Date										
2024-12-26	44.5500	70.8593	578.3621	241.8907	127.1837	193.1777	44.5500	71.3500	581.2300	24
2024-12-27	44.3400	70.5117	573.3370	239.9308	125.9221	191.6999	44.3400	71.0000	576.1800	24
2024-12-30	43.9100	69.9059	570.7200	238.0904	124.9188	190.9560	43.9100	70.3900	573.5500	23
2024-12-31	43.9500	69.9059	569.7946	238.4783	124.8890	191.2734	43.9500	70.3900	572.6200	23

Write a function `download()` that accepts tickers and returns a wide data frame of returns with the daily benchmark return factors.

We can even add a `shape` argument to return a wide or long data frame!

We can even add a `shape` argument to return a wide or long data frame!

```
import warnings

def download(tickers, shape='wide'):
    """
    Download stock price data and Fama-French factors, returning in either 'wide' or 'long' f

    Parameters:
    - tickers (str or list of str): Stock ticker(s) to download.
    - shape (str): Output format, either 'wide' (default) or 'long'.

    Returns:
    - pd.DataFrame: A DataFrame containing stock prices, returns, and Fama-French factors.
    """

    # shape must be wide or long
    if shape not in ['wide', 'long']:
        raise ValueError('Invalid shape: must be "wide" or "long".')

    # Download stock data
    stocks = yf.download(tickers=tickers, auto_adjust=False, progress=False)

    # Download Fama-French factors
    # (suppressing FutureWarning for 'date_parser')
    with warnings.catch_warnings():
        warnings.simplefilter('ignore', category=FutureWarning)
        factors = pdr.DataReader(
```

```
            name='F-F_Research_Data_Factors_daily',
            data_source='famafrench',
            start='1900'
        )[0].div(100) # Convert percentages to decimals

    # Multi-index case
    if isinstance(stocks.columns, pd.MultiIndex):
        # Compute daily returns
        _ = pd.MultiIndex.from_product([['Returns'], stocks['Adj Close'].columns])
        stocks[_] = stocks['Adj Close'].pct_change()

        if shape == 'wide':
            # Add factors with multi-index
            _ = pd.MultiIndex.from_product([['Factors'], factors.columns])
            stocks[_] = factors
            return stocks

        # Convert to long format then add factors
        else:
            return stocks.stack(future_stack=True).join(factors)

    # Single index case
    # (redundant with recent versions of yfinance that always return a multi-index)
    stocks['Returns'] = stocks['Adj Close'].pct_change()
    return stocks.join(factors)
```

```
download(tickers='AAPL TSLA')
```

Price	Adj Close		Close		High		Low		Open
Ticker	AAPL	TSLA	AAPL	TSLA	AAPL	TSLA	AAPL	TSLA	AAPL
Date									
1980-12-12	0.0987	NaN	0.1283	NaN	0.1289	NaN	0.1283	NaN	0.1283
1980-12-15	0.0936	NaN	0.1217	NaN	0.1222	NaN	0.1217	NaN	0.1222
1980-12-16	0.0867	NaN	0.1127	NaN	0.1133	NaN	0.1127	NaN	0.1133
1980-12-17	0.0889	NaN	0.1155	NaN	0.1161	NaN	0.1155	NaN	0.1155
1980-12-18	0.0914	NaN	0.1189	NaN	0.1194	NaN	0.1189	NaN	0.1189
...	...	...	...	...	...	...	...	...	...
2025-02-24	247.1000	330.5300	247.1000	330.5300	248.8600	342.4000	244.4200	324.7000	244.9300
2025-02-25	247.0400	302.8000	247.0400	302.8000	250.0000	328.8900	244.9100	297.2500	248.0000
2025-02-26	240.3600	290.8000	240.3600	290.8000	244.9800	309.0000	239.1300	288.0400	244.3300
2025-02-27	237.3000	281.9500	237.3000	281.9500	242.4600	297.2300	237.0600	280.8800	239.4100

Price	Adj Close		Close		High		Low		Open
Ticker	AAPL	TSLA	AAPL	TSLA	AAPL	TSLA	AAPL	TSLA	AAPL
Date									
2025-02-28	241.8400	292.9800	241.8400	292.9800	242.0900	293.8800	230.2000	273.6000	236.9500

> **Note**
>
> The `yfinance` package is a powerful tool for downloading market data, financial statements, and analyst estimates from Yahoo! Finance.
> However, because `yfinance` relies on Yahoo! Finance's API, changes to the API can disrupt its functionality.
> Recently, Yahoo! Finance changed API access to earnings forecasts and announcement dates, so we **cannot complete the earnings announcement exercise I had planned**.
> Instead, I will prepare an alternative set of exercises for us to work on in class on Friday.
> Thank you for your flexibility!

Download earnings per share for the stocks in `stocks_long` and combine to a long data frame `earnings`.

Use the `.earnings_dates` method described here. Use `pd.concat()` to combine the result of each the `.earnings_date` data frames and assign them to a new data frame `earnings`. Name the row indexes `Ticker` and `Date` and swap to match the order of the row index in `stocks_long`.

Combine `earnings` with the returns from `stocks_long`.

Use the `.earnings_dates` method described here. Use `pd.concat()` to combine the result of each the `.earnings_date` data frames and assign them to a new data frame `earnings`. Name the row indexes `Ticker` and `Date` and swap to match the order of the row index in `stocks_long`.

Plot the relation between daily returns and earnings surprises

Repeat the earnings exercise with the S&P 100 stocks

With more data, we can more clearly see the positive relation between earnings surprises and returns!

Herron Topic 1 - Log and Simple Returns, Portfolio Math, and Applications - Blank

This notebook covers two topics:

1. Log and simple returns
2. Portfolio returns, plus two applications of portfolio returns

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import pandas_datareader as pdr
import yfinance as yf
```

```
%precision 4
pd.options.display.float_format = '{:.4f}'.format
# %config InlineBackend.figure_format = 'retina'
```

Log and Simple Returns

We will typically use *simple* returns, calculated as $r_{simple,t} = \frac{p_t + d_t - p_{t-1}}{p_{t-1}}$. This simple return is the return that investors earn on their investments. We can calculate simple returns from Yahoo Finance data with the `.pct_change()` method on the adjusted close column (i.e., `Adj Close`), which adjusts for dividends and splits. The adjusted close column is a reverse-engineered close price (i.e., end-of-trading-day price) that incorporates dividends and splits, making simple return calculations easy.

However, we may see *log* returns elsewhere, which are the (natural) log of one plus simple returns: $r_{log,t} = \log(1 + r_{simple,t})$. Therefore, we calculate log returns as either the log of one plus simple returns or the difference of the logs of the adjusted close column. Log returns are also known as *continuously-compounded* returns.

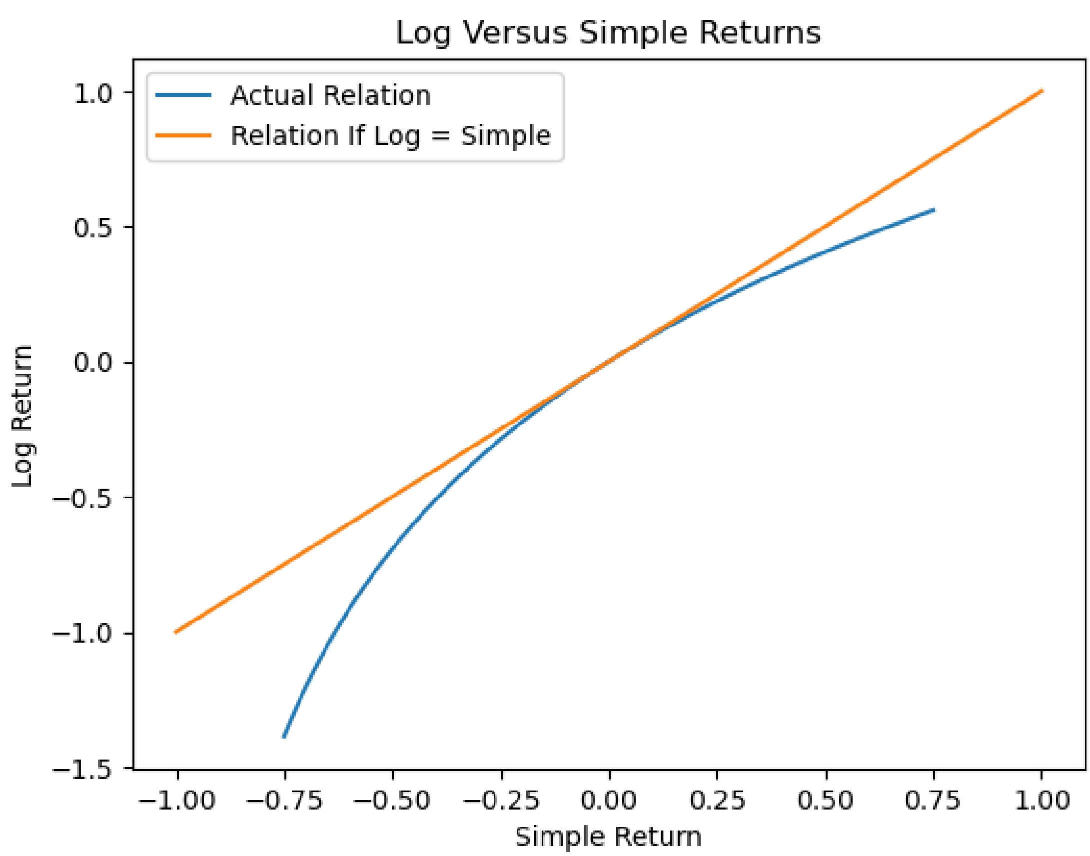
This section explains the differences between simple and log returns and where each is appropriate.

Simple and Log Returns are Similar for Small Returns

$x \approx \log(1+x)$ for small values of x, so simple returns and log returns are similar for small returns. Returns are typically small at daily and monthly horizons, so the difference between simple and log returns is small at daily and monthly horizons. The following figure shows that $r_{simple,t} \approx r_{log,t}$ for small values of r.

```
simpler = np.linspace(-0.75, 0.75, 100)
logr = np.log1p(simpler)
```

```
plt.plot(simpler, logr)
plt.plot([-1, 1], [-1, 1])
plt.xlabel('Simple Return')
plt.ylabel('Log Return')
plt.title('Log Versus Simple Returns')
plt.legend(['Actual Relation', 'Relation If Log = Simple'])
plt.show()
```



Simple Return Advantage: Portfolio Calculations

For a portfolio of N assets with portfolio weights w_i, the portfolio return r_p is the weighted average of the returns of its assets: $r_p = \sum_{i=1}^N w_i r_i$. For example, for an equal-weighted portfolio with two stocks, $r_p = 0.5r_1 + 0.5r_2 = \frac{r_1+r_2}{2}$. Therefore, we cannot calculate portfolio returns with log returns because the sum of logs is the log of products. That is $\log(1+r_i) + \log(1+r_j) = \log((1+r_i) \times (1+r_j))$, which is not what we want to measure! ***We cannot perform portfolio calculations with log returns!***

Log Return Advantage: Log Returns are Additive

We compound simple returns with multiplication, *but we compound log returns with addition.* This additive property of log returns makes code simple, computations fast, and proofs easy when we must compound returns.

We compound returns from $t = 0$ to $t = T$ as follows:

$$1 + r_{0,T} = (1 + r_1) \times (1 + r_2) \times \cdots \times (1 + r_T)$$

Next, we take the log of both sides of the previous equation and use the property that the log of products is the sum of logs:

$$\log(1+r_{0,T}) = \log((1+r_1)\times(1+r_2)\times\cdots\times(1+r_T)) = \log(1+r_1)+\log(1+r_2)+\cdots+\log(1+r_T) = \sum_{t=1}^{T} \log(1+r_t)$$

Next, we exponentiate both sides of the previous equation:

$$e^{\log(1+r_{0,T})} = e^{\sum_{t=0}^T \log(1+r_t)}$$

Next, we use the property that $e^{\log(x)} = x$ to simplify the previous equation:

$$1 + r_{0,T} = e^{\sum_{t=0}^T \log(1+r_t)}$$

Finally, we subtract 1 from both sides:

$$r_{0,T} = e^{\sum_{t=0}^T \log(1+r_t)} - 1$$

So, the return $r_{0,T}$ from $t = 0$ to $t = T$ is the exponentiated sum of log returns. The pandas developers assume users understand the math above and focus on optimizing sums!

```
np.random.seed(42)
df = pd.DataFrame(data={'r': np.exp(np.random.randn(10_000)) - 1})

df.describe()
```

	r
count	10000.0000
mean	0.6529
std	2.1918
min	-0.9802
25%	-0.4896
50%	-0.0026
75%	0.9564
max	49.7158

We can time the calculation of 10-observation rolling returns. We use `.apply()` for the simple return version because `.rolling()` does not have a product method. We find that `.rolling()` is slower with `.apply()` than with `.sum()` by a factor of about 1,000. ***We will learn about `.rolling()` and `.apply()` in a few weeks, but they provide the best example of when to use log returns.***

```
%%timeit
df['r10_via_prod'] = (
    df['r']
    .add(1)
    .rolling(10)
    .apply(lambda x: x.prod())
    .sub(1)
)
```

```
936 ms ± 119 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

```
%%timeit
df['r10_via_sum'] = (
    df['r']
    .add(1)
    .pipe(np.log)
    .rolling(10)
    .sum()
    .pipe(np.exp)
    .sub(1)
)
```

```
1.15 ms ± 159  s per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

```
df.head(15)
```

	r	r10_via_prod	r10_via_sum
0	0.6433	NaN	NaN
1	-0.1291	NaN	NaN
2	0.9111	NaN	NaN
3	3.5861	NaN	NaN
4	-0.2088	NaN	NaN
5	-0.2087	NaN	NaN
6	3.8511	NaN	NaN
7	1.1542	NaN	NaN
8	-0.3747	NaN	NaN
9	0.7204	87.2886	87.2886
10	-0.3709	32.8006	32.8006
11	-0.3723	23.3617	23.3617
12	0.2737	15.2369	15.2369
13	-0.8524	-0.4774	-0.4774
14	-0.8218	-0.8823	-0.8823

```
np.allclose(df['r10_via_prod'], df['r10_via_sum'], equal_nan=True)
```

```
True
```

These two approaches calculate the same return series, but the simple-return approach using `.prod()` is about 1,000 times slower than the log-return approach using `.sum()`! ***We can use log returns to calculate total returns very quickly!***

Portfolio Math

Portfolio return r_p is the weighted average of its asset returns, so $r_p = \sum_{i=1}^N w_i r_i$. Here N is the number of assets, w_i is the weight on asset i, and $\sum_{i=1}^N w_i = 1$.

The 1/N Portfolio

The $\frac{1}{N}$ portfolio equally weights portfolio assets, so $w_1 = w_2 = \dots = w_N = \frac{1}{N}$. If $w_i = \frac{1}{N}$, then $r_p = \sum_{i=1}^N \frac{1}{N} r_i = \frac{\sum_{i=1}^N r_i}{N} = \bar{r}$. Therefore, we can use `.mean(axis=1)` to calculate $\frac{1}{N}$ portfolio returns!

```python
df2 = yf.download(tickers='AAPL AMZN GOOG MSFT NVDA TSLA', auto_adjust=False, progress=False)
returns2 = df2['Adj Close'].pct_change().dropna()

returns2.describe()
```

Ticker	AAPL	AMZN	GOOG	MSFT	NVDA	TSLA
count	3689.0000	3689.0000	3689.0000	3689.0000	3689.0000	3689.0000
mean	0.0011	0.0012	0.0009	0.0010	0.0021	0.0021
std	0.0175	0.0205	0.0173	0.0161	0.0288	0.0362
min	-0.1286	-0.1405	-0.1110	-0.1474	-0.1876	-0.2106
25%	-0.0074	-0.0089	-0.0071	-0.0070	-0.0122	-0.0164
50%	0.0010	0.0010	0.0009	0.0007	0.0017	0.0012
75%	0.0102	0.0119	0.0093	0.0093	0.0161	0.0194
max	0.1198	0.1575	0.1605	0.1422	0.2981	0.2440

```python
returns2.mean() # implied axis=0
```

```
Ticker
AAPL   0.0011
AMZN   0.0012
GOOG   0.0009
MSFT   0.0010
NVDA   0.0021
TSLA   0.0021
dtype: float64
```

```python
rp2_via_mean = returns2.mean(axis=1)

rp2_via_mean
```

```
Date
2010-06-30   -0.0123
2010-07-01   -0.0107
2010-07-02   -0.0271
2010-07-06   -0.0223
2010-07-07    0.0254
                ...
2025-02-21   -0.0272
2025-02-24   -0.0127
```

```
2025-02-25   -0.0247
2025-02-26   -0.0055
2025-02-27   -0.0330
Length: 3689, dtype: float64
```

Note that when we apply the same portfolio weights every period, we rebalance at the same frequency as the returns data. If we have daily data, rebalance daily. If we have monthly data, we rebalance monthly, and so on.

A More General Solution

If we combine portfolio weights into vector w and the time series of asset returns into matrix $\mathbf{R}$, then we can calculate the time series of portfolio returns as $r_p = w^T \mathbf{R}$. The pandas version of this calculation is `R.dot(w)`, where `R` is a data frame of asset returns and `w` is a series or an array of portfolio weights. We can use this approach to calculate $\frac{1}{N}$ portfolio returns, too.

```python
weights2 = np.ones(returns2.shape[1]) / returns2.shape[1]

weights2
```

```
array([0.1667, 0.1667, 0.1667, 0.1667, 0.1667, 0.1667])
```

```python
rp2_via_dot = returns2.dot(weights2)

rp2_via_dot
```

```
Date
2010-06-30   -0.0123
2010-07-01   -0.0107
2010-07-02   -0.0271
2010-07-06   -0.0223
2010-07-07    0.0254
                ...
2025-02-21   -0.0272
2025-02-24   -0.0127
2025-02-25   -0.0247
2025-02-26   -0.0055
2025-02-27   -0.0330
Length: 3689, dtype: float64
```

Both approaches give the same answer!

```
np.allclose(rp2_via_mean, rp2_via_dot, equal_nan=True)
```

```
True
```

Portfolio Math Application 1: All stocks half the time or half stocks all the time?

Are you better off investing:

1. 100% in stocks 50% of the time and the riskless asset the other 50% of the time *or*
2. 50% in stocks and 50% in the riskless asset 100% of the time?

Here is a roadmap for convincing yourself with data!

Download *annual* market and risk-free asset returns from Kenneth French's data library

Convert these factors to decimal returns and calculate the market return series

Add a portfolio return series that is half stocks all the time

You might call this portfolio return series `Balanced`

Add a portfolio return series that switches between stocks and bills every year with stocks in odd years

You might call this portfolio return series `Switching Stocks Odd`

Add a portfolio return series that switches between stocks and bills every year with stocks in even years

You might call this portfolio return series `Switching Stocks Even`

Plot the cumulative returns calculate the summary statistics for the `Balanced` and `Switching` series

Use the `.describe()` method to calcualte summary statistics.

Which strategy do you prefer?

Why? How sure are you?

Use the `simulate()` function to simulate 10,000 different outcomes for the U.S. market

`simulate()` calculates one `Switching` return series because the randomization also randomizes the odd-year and even-year choice.

```
def simulate(df, cols=['Mkt', 'RF'], n_iter=10_000):
    """
    Simulates resampling of the given DataFrame columns and computes bal;anced and switching

    Parameters:
    df (pd.DataFrame): The input DataFrame.
    cols (list): List of column names to sample.
    n_iter (int): Number of iterations for simulation.

    Returns:
    pd.DataFrame: A concatenated DataFrame with simulation results.
    """
    return pd.concat(
        objs=[(
            df[cols]
            .sample(frac=1, ignore_index=True, random_state=i)
            .assign(
                Balanced=lambda x: x[cols].mean(axis=1),
                Switching=lambda x: np.where(x.index % 2 == 0, x[cols[0]], x[cols[1]])
            )
        ) for i in range(n_iter)],
        keys=range(n_iter),
        names=['Simulation', 'Year']
    )
```

Calculate the summary statistics for these new `Balanced` and `Switching` series

Which strategy do you prefer?

Why? How sure are you?

Portfolio Math Application 2: What are the benefits of diversification?

Use random portfolios of S&P 100 stocks of various portfolio sizes to show that portfolio volatility falls quickly, then slowly, then not at all as we increase portfolio size.

Download daily data for the stocks in the S&P 100

Wikipedia provides tickers for the stocks in the S&P 100. Use a list comprehension to replace `.` in tickers with `-` for compatability with Yahoo! Finance.

Calculate the past five years of dailty returns for these stocks

Calculate the volatilies of 20 equal-weighted random portfolios of various portfolio sizes

Random portfolios should have portfolio sizes of 1, 2, 4, 6, 8, 10, 20, 30, 40, or 50 stocks each.

You can combine the `.sample(n=?, axis=1, random_state=?)`, `.mean(axis=1)`, and `.std()` to calculate the volatilities of equal-weighted portfolios. You can collect these volatilies in a list of lists built with two `for` loops or list comprehensions. Replace the `?`s in `.sample()` with loop counters. The inner loop will calculate a portfolio volatilty for each portfolio size, and the outer loop will collect 20 versions of each portfolio. Using the outer loop counter for `random_state=` makes your analysis repeatable!

Combine this list of list into a data frame

Calculate the mean volatility for each portfolio size and replicate the plot above

Herron Topic 1 - Log and Simple Returns, Portfolio Math, and Applications - Sec 02

This notebook covers two topics:

1. Log and simple returns
2. Portfolio returns, plus two applications of portfolio returns

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import pandas_datareader as pdr
import yfinance as yf
```

```
%precision 4
pd.options.display.float_format = '{:.4f}'.format
# %config InlineBackend.figure_format = 'retina'
```

Log and Simple Returns

We will typically use *simple* returns, calculated as $r_{simple,t} = \frac{p_t + d_t - p_{t-1}}{p_{t-1}}$. This simple return is the return that investors earn on their investments. We can calculate simple returns from Yahoo Finance data with the `.pct_change()` method on the adjusted close column (i.e., `Adj Close`), which adjusts for dividends and splits. The adjusted close column is a reverse-engineered close price (i.e., end-of-trading-day price) that incorporates dividends and splits, making simple return calculations easy.

However, we may see *log* returns elsewhere, which are the (natural) log of one plus simple returns: $r_{log,t} = \log(1 + r_{simple,t})$. Therefore, we calculate log returns as either the log of one plus simple returns or the difference of the logs of the adjusted close column. Log returns are also known as *continuously-compounded* returns.

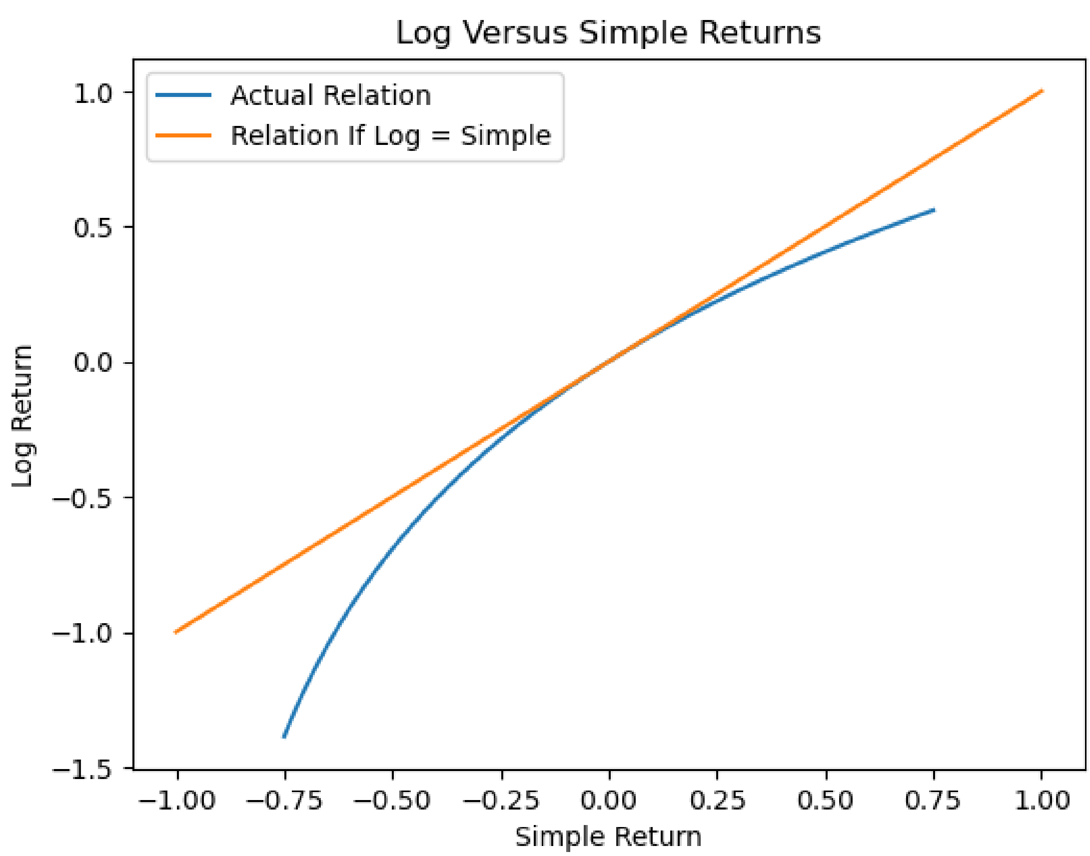
This section explains the differences between simple and log returns and where each is appropriate.

Simple and Log Returns are Similar for Small Returns

$x \approx \log(1+x)$ for small values of x, so simple returns and log returns are similar for small returns. Returns are typically small at daily and monthly horizons, so the difference between simple and log returns is small at daily and monthly horizons. The following figure shows that $r_{simple,t} \approx r_{log,t}$ for small values of r.

```
simpler = np.linspace(-0.75, 0.75, 100)
logr = np.log1p(simpler)
```

```
plt.plot(simpler, logr)
plt.plot([-1, 1], [-1, 1])
plt.xlabel('Simple Return')
plt.ylabel('Log Return')
plt.title('Log Versus Simple Returns')
plt.legend(['Actual Relation', 'Relation If Log = Simple'])
plt.show()
```



Simple Return Advantage: Portfolio Calculations

For a portfolio of N assets with portfolio weights w_i, the portfolio return r_p is the weighted average of the returns of its assets: $r_p = \sum_{i=1}^{N} w_i r_i$. For example, for an equal-weighted portfolio with two stocks, $r_p = 0.5r_1 + 0.5r_2 = \frac{r_1+r_2}{2}$. Therefore, we cannot calculate portfolio returns with log returns because the sum of logs is the log of products. That is $\log(1 + r_i) + \log(1 + r_j) = \log((1 + r_i) \times (1 + r_j))$, which is not what we want to measure! ***We cannot perform portfolio calculations with log returns!***

Log Return Advantage: Log Returns are Additive

We compound simple returns with multiplication, *but we compound log returns with addition.* This additive property of log returns makes code simple, computations fast, and proofs easy when we must compound returns.

We compound returns from $t = 0$ to $t = T$ as follows:

$$1 + r_{0,T} = (1 + r_1) \times (1 + r_2) \times \dots \times (1 + r_T)$$

Next, we take the log of both sides of the previous equation and use the property that the log of products is the sum of logs:

$$\log(1+r_{0,T}) = \log((1+r_1)\times(1+r_2)\times\dots\times(1+r_T)) = \log(1+r_1)+\log(1+r_2)+\dots+\log(1+r_T) = \sum_{t=1}^{T} \log(1+r_t)$$

Next, we exponentiate both sides of the previous equation:

$$e^{\log(1+r_{0,T})} = e^{\sum_{t=0}^{T} \log(1+r_t)}$$

Next, we use the property that $e^{\log(x)} = x$ to simplify the previous equation:

$$1 + r_{0,T} = e^{\sum_{t=0}^{T} \log(1+r_t)}$$

Finally, we subtract 1 from both sides:

$$r_{0,T} = e^{\sum_{t=0}^{T} \log(1+r_t)} - 1$$

So, the return $r_{0,T}$ from $t = 0$ to $t = T$ is the exponentiated sum of log returns. The pandas developers assume users understand the math above and focus on optimizing sums!

```
np.random.seed(42)
df = pd.DataFrame(data={'r': np.exp(np.random.randn(10_000)) - 1})

df.describe()
```

	r
count	10000.0000
mean	0.6529
std	2.1918
min	-0.9802
25%	-0.4896
50%	-0.0026
75%	0.9564
max	49.7158

We can time the calculation of 10-observation rolling returns. We use `.apply()` for the simple return version because `.rolling()` does not have a product method. We find that `.rolling()` is slower with `.apply()` than with `.sum()` by a factor of about 1,000. ***We will learn about `.rolling()` and `.apply()` in a few weeks, but they provide the best example of when to use log returns.***

```
%%timeit
df['r10_via_prod'] = (
    df['r']
    .add(1)
    .rolling(10)
    .apply(lambda x: x.prod())
    .sub(1)
)
```

```
233 ms ± 84.1 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
%%timeit
df['r10_via_sum'] = (
    df['r']
    .add(1)
    .pipe(np.log)
    .rolling(10)
    .sum()
    .pipe(np.exp)
    .sub(1)
)
```

```
1.71 ms ± 297  s per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

```
df.head(15)
```

	r	r10_via_prod	r10_via_sum
0	0.6433	NaN	NaN
1	-0.1291	NaN	NaN
2	0.9111	NaN	NaN
3	3.5861	NaN	NaN
4	-0.2088	NaN	NaN
5	-0.2087	NaN	NaN
6	3.8511	NaN	NaN
7	1.1542	NaN	NaN
8	-0.3747	NaN	NaN
9	0.7204	87.2886	87.2886
10	-0.3709	32.8006	32.8006
11	-0.3723	23.3617	23.3617
12	0.2737	15.2369	15.2369
13	-0.8524	-0.4774	-0.4774
14	-0.8218	-0.8823	-0.8823

```
np.allclose(df['r10_via_prod'], df['r10_via_sum'], equal_nan=True)
```

```
True
```

These two approaches calculate the same return series, but the simple-return approach using `.prod()` is about 1,000 times slower than the log-return approach using `.sum()`! ***We can use log returns to calculate total returns very quickly!***

Portfolio Math

Portfolio return r_p is the weighted average of its asset returns, so $r_p = \sum_{i=1}^N w_i r_i$. Here N is the number of assets, w_i is the weight on asset i, and $\sum_{i=1}^N w_i = 1$.

The 1/N Portfolio

The $\frac{1}{N}$ portfolio equally weights portfolio assets, so $w_1 = w_2 = \cdots = w_N = \frac{1}{N}$. If $w_i = \frac{1}{N}$, then $r_p = \sum_{i=1}^N \frac{1}{N} r_i = \frac{\sum_{i=1}^N r_i}{N} = \bar{r}$. Therefore, we can use `.mean(axis=1)` to calculate $\frac{1}{N}$ portfolio returns!

```
df2 = yf.download(tickers='AAPL AMZN GOOG MSFT NVDA TSLA', auto_adjust=False, progress=False)
returns2 = df2['Adj Close'].pct_change().dropna()

returns2.describe()
```

Ticker	AAPL	AMZN	GOOG	MSFT	NVDA	TSLA
count	3689.0000	3689.0000	3689.0000	3689.0000	3689.0000	3689.0000
mean	0.0011	0.0012	0.0009	0.0010	0.0021	0.0021
std	0.0175	0.0205	0.0173	0.0161	0.0288	0.0362
min	-0.1286	-0.1405	-0.1110	-0.1474	-0.1876	-0.2106
25%	-0.0074	-0.0089	-0.0071	-0.0070	-0.0122	-0.0164
50%	0.0010	0.0010	0.0009	0.0007	0.0017	0.0012
75%	0.0102	0.0119	0.0093	0.0093	0.0161	0.0194
max	0.1198	0.1575	0.1605	0.1422	0.2981	0.2440

```
returns2.mean() # implied axis=0
```

```
Ticker
AAPL   0.0011
AMZN   0.0012
GOOG   0.0009
MSFT   0.0010
NVDA   0.0021
TSLA   0.0021
dtype: float64
```

```
rp2_via_mean = returns2.mean(axis=1)

rp2_via_mean
```

```
Date
2010-06-30   -0.0123
2010-07-01   -0.0107
2010-07-02   -0.0271
2010-07-06   -0.0223
2010-07-07    0.0254
               ...
2025-02-21   -0.0272
2025-02-24   -0.0127
```

```
2025-02-25   -0.0247
2025-02-26   -0.0055
2025-02-27   -0.0330
Length: 3689, dtype: float64
```

Note that when we apply the same portfolio weights every period, we rebalance at the same frequency as the returns data. If we have daily data, rebalance daily. If we have monthly data, we rebalance monthly, and so on.

A More General Solution

If we combine portfolio weights into vector w and the time series of asset returns into matrix $\mathbf{R}$, then we can calculate the time series of portfolio returns as $r_p = w^T\mathbf{R}$. The pandas version of this calculation is `R.dot(w)`, where `R` is a data frame of asset returns and `w` is a series or an array of portfolio weights. We can use this approach to calculate $\frac{1}{N}$ portfolio returns, too.

```python
weights2 = np.ones(returns2.shape[1]) / returns2.shape[1]

weights2
```

```
array([0.1667, 0.1667, 0.1667, 0.1667, 0.1667, 0.1667])
```

```python
rp2_via_dot = returns2.dot(weights2)

rp2_via_dot
```

```
Date
2010-06-30   -0.0123
2010-07-01   -0.0107
2010-07-02   -0.0271
2010-07-06   -0.0223
2010-07-07    0.0254
               ...
2025-02-21   -0.0272
2025-02-24   -0.0127
2025-02-25   -0.0247
2025-02-26   -0.0055
2025-02-27   -0.0330
Length: 3689, dtype: float64
```

Both approaches give the same answer!

```
np.allclose(rp2_via_mean, rp2_via_dot, equal_nan=True)
```

```
True
```

Portfolio Math Application 1: All stocks half the time or half stocks all the time?

Are you better off investing:

1. 100% in stocks 50% of the time and the riskless asset the other 50% of the time *or*
2. 50% in stocks and 50% in the riskless asset 100% of the time?

Here is a roadmap for convincing yourself with data!

Please see Kritzman (2000, Chapter 5) for a more detailed solution!

Download *annual* market and risk-free asset returns from Kenneth French's data library

```
pdr.famafrench.get_available_datasets()[:5]
```

```
['F-F_Research_Data_Factors',
 'F-F_Research_Data_Factors_weekly',
 'F-F_Research_Data_Factors_daily',
 'F-F_Research_Data_5_Factors_2x3',
 'F-F_Research_Data_5_Factors_2x3_daily']
```

```
ff = pdr.DataReader(
    name='F-F_Research_Data_Factors',
    data_source='famafrench',
    start='1900'
)
```

```
C:\Users\r.herron\AppData\Local\Temp\ipykernel_2148\304025689.py:1: FutureWarning: The argume
  ff = pdr.DataReader(
C:\Users\r.herron\AppData\Local\Temp\ipykernel_2148\304025689.py:1: FutureWarning: The argume
  ff = pdr.DataReader(
```

```
print(ff['DESCR'])
```

```
F-F Research Data Factors
-------------------------

This file was created by CMPT_ME_BEME_RETS using the 202412 CRSP database. The 1-month TBill

  0 : (1182 rows x 4 cols)
  1 : Annual Factors: January-December (98 rows x 4 cols)
```

Convert these factors to decimal returns and calculate the market return series

```
df3 = ff[1].div(100)
```

```
df3['Mkt'] = df3['Mkt-RF'] + df3['RF']
```

Add a portfolio return series that is half stocks all the time

You might call this portfolio return series `Balanced`

```
df3['Balanced'] = df3[['Mkt', 'RF']].mean(axis=1)
```

Add a portfolio return series that switches between stocks and bills every year with stocks in odd years

You might call this portfolio return series `Switching Stocks Odd`

```
df3['Switching Stocks Odd'] = np.where(df3.index.year % 2 == 1, df3['Mkt'], df3['RF'])
```

Add a portfolio return series that switches between stocks and bills every year with stocks in even years

You might call this portfolio return series `Switching Stocks Even`

```
df3['Switching Stocks Even'] = np.where(df3.index.year % 2 == 0, df3['Mkt'], df3['RF'])
```

```
df3.head()
```

Date	Mkt-RF	SMB	HML	RF	Mkt	Balanced	Switching Stocks Odd	Switching Stocks E
1927	0.2947	-0.0204	-0.0454	0.0312	0.3259	0.1785	0.3259	0.0312
1928	0.3539	0.0451	-0.0617	0.0356	0.3895	0.2126	0.0356	0.3895
1929	-0.1954	-0.3070	0.1167	0.0475	-0.1479	-0.0502	-0.1479	0.0475
1930	-0.3123	-0.0517	-0.1154	0.0241	-0.2882	-0.1321	0.0241	-0.2882
1931	-0.4511	0.0370	-0.1395	0.0107	-0.4404	-0.2149	-0.4404	0.0107

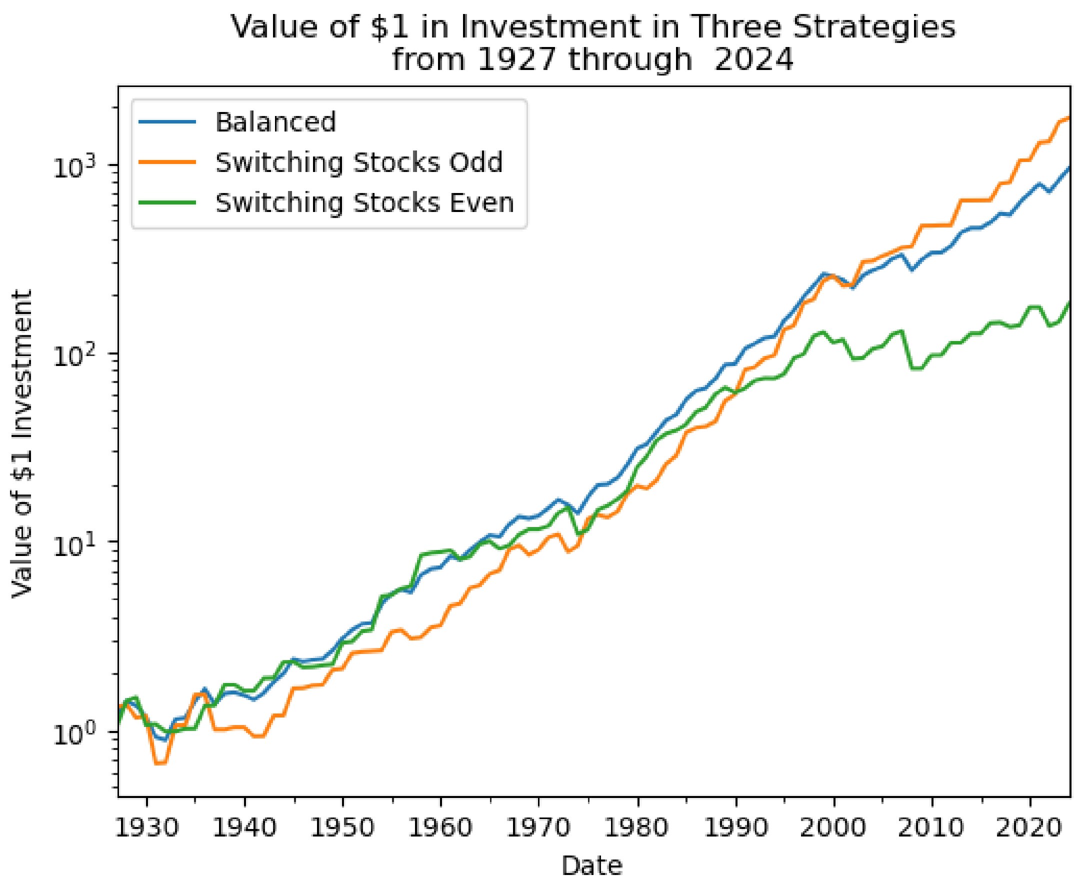
Plot the cumulative returns on a $1 investment and calculate the summary statistics for the `Balanced` and `Switching` series

Use the `.describe()` method to calculate summary statistics.

```
portfolios = ['Balanced', 'Switching Stocks Odd', 'Switching Stocks Even']
```

```
import matplotlib.ticker as ticker
```

```
(
    df3[portfolios]
    .add(1)
    .cumprod()
    .plot()
)

plt.yscale('log')
plt.ylabel('Value of $1 Investment')
plt.title(f'Value of $1 in Investment in Three Strategies\nfrom {df3.index.year[0]} through

plt.show()
```



Which strategy do you prefer?

Why? How sure are you?

The `Switching Stocks Odd` strategy *seems* best in this example! However, this apparent superiority is largely a matter of **luck**, specifically this combination of the sample period and starting year. In this sample, the number of even and odd years is equal, but the market happened to perform better in odd years. This outcome is entirely driven by **this particular historical sequence**.

To draw broader conclusions, we must break free from the "luck of the draw" tied to this single realization of history. Below, we will explore two approaches to determine which strategy is better:

1. **Theory:** We will analyze the expected returns and variances mathematically to assess the underlying return-risk tradeoff for each strategy.

2. **Simulation:** To rely on data instead of theory, we will simulate thousands of alternative historical sequences.

By shuffling the 98 years of market data repeatedly, we can generate 10,000 random samples and evaluate the performance of each strategy across these simulations.
This approach allows us to:

- Confirm that **both strategies have the same arithmetic average return**.
- Show that the **balanced strategy consistently delivers a better return-risk tradeoff (Sharpe ratio)** when evaluated across many possible outcomes.

We will do the simulation first.

Use the `simulate()` function to simulate 10,000 different outcomes for the U.S. market

`simulate()` calculates one `Switching` return series because the randomization also randomizes the odd-year and even-year choice.

```
df3[['Mkt', 'RF']].sample(frac=1, ignore_index=True, random_state=42).head()
```

	Mkt	RF
0	0.2886	0.0837
1	0.2870	0.0421
2	0.2361	0.0004
3	0.3871	0.0033
4	-0.3674	0.0160

```
def simulate(df, cols=['Mkt', 'RF'], n_iter=10_000):
    """
    Simulates resampling of the given DataFrame columns and computes bal;anced and switching

    Parameters:
    df (pd.DataFrame): The input DataFrame.
    cols (list): List of column names to sample.
    n_iter (int): Number of iterations for simulation.

    Returns:
    pd.DataFrame: A concatenated DataFrame with simulation results.
    """
    return pd.concat(
        objs=[(
```

```
            df[cols]
            .sample(frac=1, ignore_index=True, random_state=i)
            .assign(
                Balanced=lambda x: x[cols].mean(axis=1),
                Switching=lambda x: np.where(x.index % 2 == 0, x[cols[0]], x[cols[1]])
            )
        ) for i in range(n_iter)],
        keys=range(n_iter),
        names=['Simulation', 'Year']
    )
```

```
df4 = simulate(df3)
```

Calculate the summary statistics for these new `Balanced` and `Switching` series

```
df4[['Balanced', 'Switching']].agg(['mean', 'std'])
```

	Balanced	Switching
mean	0.0772	0.0774
std	0.1001	0.1487

We see that `Balanced` and `Switching` have the same mean return, but `Balanced` has much lower volatility than `Switching`! A risk-averse investor prefers `Balanced` because it has a higher return/risk ratio than `Switching`. We can quantify this ratio as the Sharpe ratio, which is the mean excess return divided by the volatility of excess returns: $S_i = \frac{\overline{r_i - r_f}}{\sigma(r_i - r_f)}$. We can calcualte excess returns and Sharpe ratios in one code snippet.

```
(
    df4
    [['Balanced', 'Switching']]
    .sub(df4['RF'], axis=0)
    .agg(lambda x: x.mean() / x.std())
    .to_frame('Sharpe ratio')
)
```

	Sharpe ratio
Balanced	0.4368
Switching	0.2961

We have to do a little more work if we want to combine mean and volatility of *raw* returns with the Sharpe ratio of *excess* returns.

```
pd.concat(
    objs=[
        df4[['Balanced', 'Switching']].agg(['mean', 'std']).transpose(),
        df4[['Balanced', 'Switching']].sub(df4['RF'], axis=0).agg(lambda x: x.mean() / x.std
    ],
    axis=1
)
```

	mean	std	Sharpe ratio
Balanced	0.0772	0.1001	0.4368
Switching	0.0774	0.1487	0.2961

Which strategy do you prefer?

Why? How sure are you?

We prefer `Balanced` because it has a Sharpe ratio about 50% greater than `Switching`! We can see this in the data above, and here is the theory.

For `Balanced`, $\sigma_p^2 = w_m^2 \sigma_m^2 + w_f^2 \sigma_f^2 + 2 w_m w_f \sigma_m \sigma_f \rho_{m,f}$. Because $\sigma_f^2 \approx 0$ and $\rho_{m,f} \approx 0$, $\sigma_p^2 \approx w_m^2 \sigma_m^2$. Therefore, for `Balanced` $\sigma_p \approx w_m \sigma_m = \frac{1}{2} \sigma_m$. We see this in the data!

```
0.5 * df3['Mkt'].std()
```

```
0.0998
```

```
df4['Balanced'].std()
```

```
0.1001
```

For `Switching`, $\sigma_p^2 = w_m \sigma_m^2 + w_f \sigma_f^2 + w_m w_f (\mu_m - \mu_f)^2$. We have a different formula because `Switching` is diversified *over time* instead at a point in time! Because $\sigma_f^2 \approx 0$ and $(\mu_m - \mu_f)^2 \approx 0$, $\sigma_p^2 \approx w_m \sigma_m^2$. Therefore, for `Balanced` $\sigma_p \approx \sqrt{w_m} \sigma_m = \sqrt{\frac{1}{2}} \sigma_m$. We see this in the data!

```
np.sqrt(0.5) * df3['Mkt'].std()
```

```
0.1411
```

```
df4['Switching'].std()
```

```
0.1487
```

The $(\mu_m - \mu_f)^2$ is close to zero but not exactly zero. We can get even closed to the observed `Switching` portfolio volatility if we consider this term!

```
np.sqrt(0.5 * df3['Mkt'].var() + 0.5 * 0.5 * (df3['Mkt'].mean() - df3['RF'].mean())**2)
```

```
0.1478
```

Please see Kritzman (2000, Chapter 5) for a more detailed solution!

Here are two after-class additions to this question.

Can we add a progress bar?

Yes! We have to rewrite the `simulate()` function and replace the list comprehension with a for loop. However, this addition requires the `tqdm` package. To avoid any conflicts, I will not install this package and provide the code as markdown instead of executable code.

```
from tqdm import tqdm

def simulate(df, cols=['Mkt', 'RF'], n_iter=10_000):
    """
    Simulates resampling of the given DataFrame columns and computes balanced
    and switching portfolio returns.
```

```
    Parameters:
    -----------
    df : pd.DataFrame
        The input DataFrame.
    cols : list
        List of column names to sample.
    n_iter : int
        Number of iterations for simulation.

    Returns:
    --------
    pd.DataFrame
        A concatenated DataFrame with simulation results.
    """
    all_resamples = []

    # Use tqdm to visualize progress
    for i in tqdm(range(n_iter), desc="Simulating"):
        # Random re-sampling
        resampled = (
            df[cols]
            .sample(frac=1, ignore_index=True, random_state=i)
            .assign(
                Balanced=lambda x: x[cols].mean(axis=1),
                Switching=lambda x: np.where(
                    x.index % 2 == 0,
                    x[cols[0]],
                    x[cols[1]]
                )
            )
        )
        all_resamples.append(resampled)

    # Concatenate final results
    return pd.concat(all_resamples, keys=range(n_iter), names=['Simulation', 'Year'])
```

Ideally, we calculate the mean and volatility of *each simulation*, then take the average

We can do this easily with either `.groupby()` of `.pivot_table()`!

```
(
    df4
    .groupby(level='Simulation')
    [['Balanced', 'Switching']]
    .agg(['mean', 'std'])
    .mean()
    .unstack()
)
```

	mean	std
Balanced	0.0772	0.1006
Switching	0.0774	0.1488

These statistics are similar, but not identical!

Portfolio Math Application 2: What are the benefits of diversification?

Use random portfolios of S&P 100 stocks of various portfolio sizes to show that portfolio volatility falls quickly, then slowly, then not at all as we increase portfolio size.

Download daily data for the stocks in the S&P 100

Wikipedia provides tickers for the stocks in the S&P 100. Use a list comprehension to replace `.` in tickers with `-` for compatability with Yahoo! Finance.

```
wiki = pd.read_html('https://en.wikipedia.org/wiki/S%26P_100')
tickers = [i.replace('.', '-') for i in wiki[2]['Symbol']]
data = yf.download(tickers=tickers, auto_adjust=False, progress=False).iloc[:-1]
```

Calculate the past five years of dailty returns for these stocks

```
returns = (
    data
    ['Adj Close']
    .dropna(axis=1, how='all')
    .pct_change()
    .iloc[-5*252:]
)
```

Calculate the volatilies of 20 equal-weighted random portfolios of various portfolio sizes

Random portfolios should have portfolio sizes of 1, 2, 4, 6, 8, 10, 20, 30, 40, or 50 stocks each.

You can combine the `.sample(n=?, axis=1, random_state=?)`, `.mean(axis=1)`, and `.std()` to calculate the volatilities of equal-weighted portfolios. You can collect these volatilies in a list of lists built with two `for` loops or list comprehensions. Replace the `?`s in `.sample()` with loop counters. The inner loop will calculate a portfolio volatilty for each portfolio size, and the outer loop will collect 20 versions of each portfolio. Using the outer loop counter for `random_state=` makes your analysis repeatable!

```
portfolio_size = [1, 2, 4, 6, 8, 10, 20, 30, 40, 50]
portfolio_number = 20

list_of_volatilities = []
for i in range(portfolio_number):
    list_of_volatilities.append([returns.sample(n=j, axis=1, random_state=i).mean(axis=1).std
```

```
list_of_volatilities[0]
```

```
[0.0153,
 0.0127,
 0.0121,
 0.0118,
 0.0134,
 0.0129,
 0.0128,
 0.0129,
 0.0129,
 0.0129]
```

Combine this list of lists into a data frame

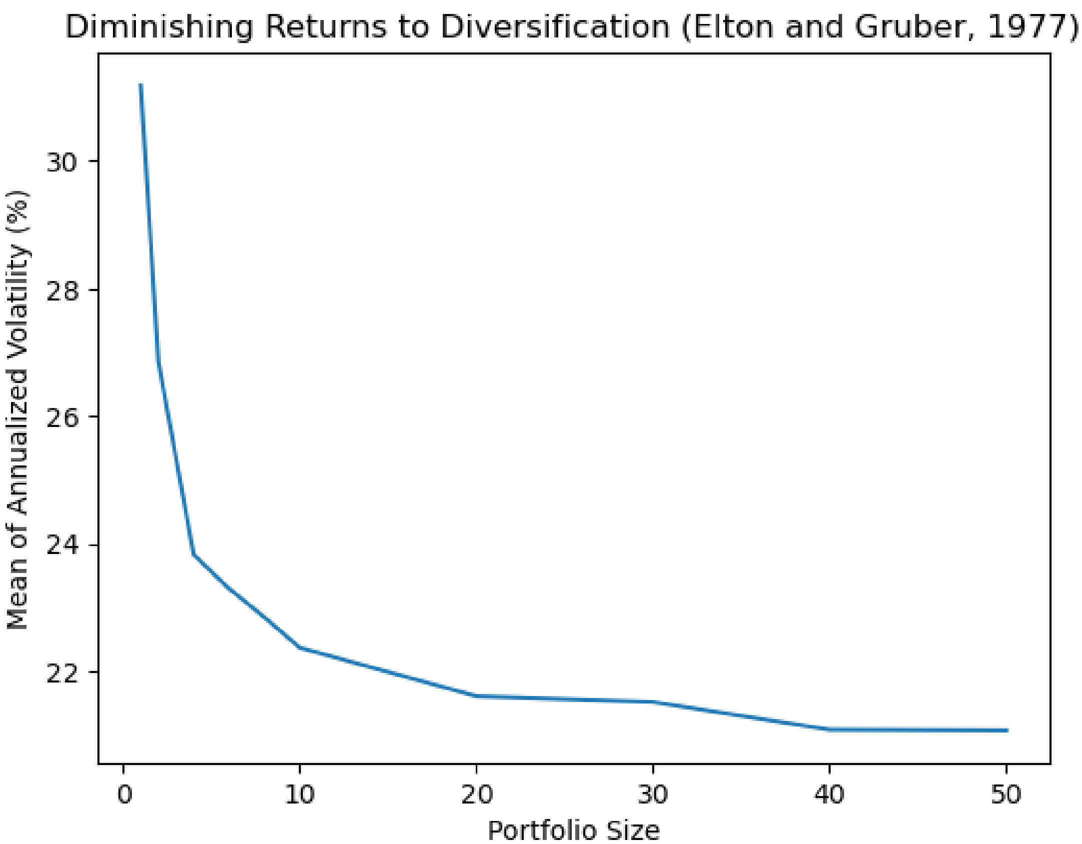
```
volatilities = (
    pd.DataFrame(
        data=list_of_volatilities,
        index=range(1, 1+portfolio_number),
        columns=portfolio_size
    )
    .rename_axis(index='Portfolio Number', columns='Portfolio Size')
)
```

```
volatilities
```

Portfolio Size Portfolio Number	1	2	4	6	8	10	20	30	40	50
1	0.0153	0.0127	0.0121	0.0118	0.0134	0.0129	0.0128	0.0129	0.0129	0.0129
2	0.0193	0.0144	0.0127	0.0145	0.0144	0.0145	0.0137	0.0136	0.0136	0.0133
3	0.0160	0.0165	0.0161	0.0151	0.0142	0.0144	0.0132	0.0134	0.0133	0.0134
4	0.0193	0.0177	0.0172	0.0171	0.0157	0.0156	0.0141	0.0141	0.0142	0.0139
5	0.0205	0.0199	0.0151	0.0164	0.0141	0.0133	0.0127	0.0131	0.0128	0.0131
6	0.0182	0.0157	0.0123	0.0136	0.0136	0.0139	0.0137	0.0143	0.0137	0.0136
7	0.0199	0.0174	0.0171	0.0160	0.0148	0.0142	0.0124	0.0133	0.0130	0.0128
8	0.0280	0.0178	0.0142	0.0149	0.0144	0.0131	0.0141	0.0138	0.0134	0.0134
9	0.0150	0.0136	0.0115	0.0116	0.0117	0.0119	0.0131	0.0132	0.0131	0.0133
10	0.0141	0.0171	0.0164	0.0150	0.0146	0.0147	0.0138	0.0131	0.0128	0.0126
11	0.0285	0.0210	0.0185	0.0170	0.0163	0.0167	0.0143	0.0142	0.0136	0.0137
12	0.0172	0.0193	0.0148	0.0154	0.0137	0.0140	0.0137	0.0133	0.0125	0.0128
13	0.0150	0.0131	0.0124	0.0126	0.0142	0.0134	0.0138	0.0133	0.0131	0.0135
14	0.0280	0.0198	0.0160	0.0147	0.0147	0.0137	0.0137	0.0131	0.0133	0.0134
15	0.0282	0.0197	0.0152	0.0142	0.0148	0.0142	0.0143	0.0140	0.0133	0.0134
16	0.0188	0.0150	0.0148	0.0136	0.0138	0.0140	0.0138	0.0141	0.0137	0.0136
17	0.0165	0.0132	0.0149	0.0141	0.0133	0.0128	0.0130	0.0133	0.0133	0.0131
18	0.0213	0.0198	0.0177	0.0166	0.0161	0.0156	0.0145	0.0142	0.0135	0.0139
19	0.0132	0.0136	0.0143	0.0140	0.0150	0.0145	0.0139	0.0134	0.0131	0.0127
20	0.0205	0.0209	0.0169	0.0154	0.0151	0.0143	0.0137	0.0134	0.0132	0.0132

Calculate the mean volatility for each portfolio size and replicate the plot above

```
volatilities.mul(100 * np.sqrt(252)).mean().plot()
plt.xlabel('Portfolio Size')
plt.ylabel('Mean of Annualized Volatility (%)')
plt.title('Diminishing Returns to Diversification (Elton and Gruber, 1977)')
plt.show()
```



Herron Topic 1 - Log and Simple Returns, Portfolio Math, and Applications - Sec 03

This notebook covers two topics:

1. Log and simple returns
2. Portfolio returns, plus two applications of portfolio returns

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import pandas_datareader as pdr
import yfinance as yf
```

```
%precision 4
pd.options.display.float_format = '{:.4f}'.format
# %config InlineBackend.figure_format = 'retina'
```

Log and Simple Returns

We will typically use *simple* returns, calculated as $r_{simple,t} = \frac{p_t + d_t - p_{t-1}}{p_{t-1}}$. This simple return is the return that investors earn on their investments. We can calculate simple returns from Yahoo Finance data with the `.pct_change()` method on the adjusted close column (i.e., `Adj Close`), which adjusts for dividends and splits. The adjusted close column is a reverse-engineered close price (i.e., end-of-trading-day price) that incorporates dividends and splits, making simple return calculations easy.

However, we may see *log* returns elsewhere, which are the (natural) log of one plus simple returns: $r_{log,t} = \log(1 + r_{simple,t})$. Therefore, we calculate log returns as either the log of one plus simple returns or the difference of the logs of the adjusted close column. Log returns are also known as *continuously-compounded* returns.

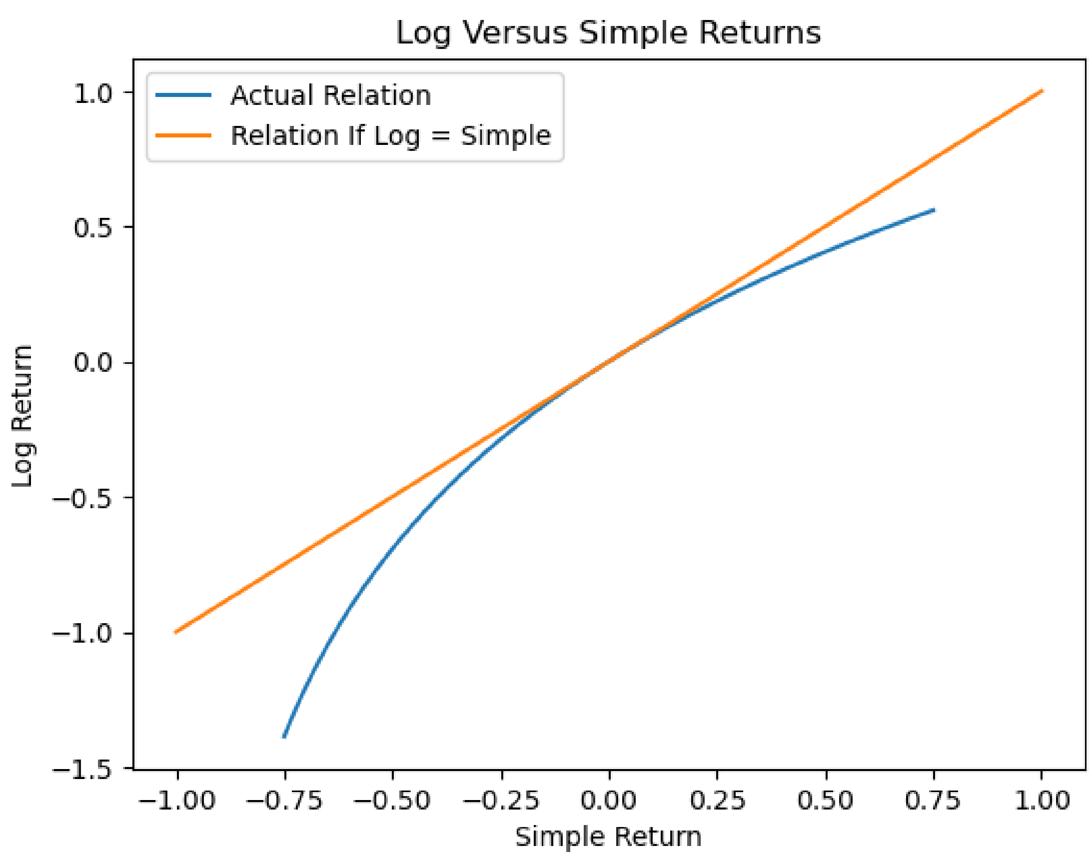
This section explains the differences between simple and log returns and where each is appropriate.

Simple and Log Returns are Similar for Small Returns

$x \approx \log(1 + x)$ for small values of x, so simple returns and log returns are similar for small returns. Returns are typically small at daily and monthly horizons, so the difference between simple and log returns is small at daily and monthly horizons. The following figure shows that $r_{simple,t} \approx r_{log,t}$ for small values of r.

```
simpler = np.linspace(-0.75, 0.75, 100)
logr = np.log1p(simpler)
```

```
plt.plot(simpler, logr)
plt.plot([-1, 1], [-1, 1])
plt.xlabel('Simple Return')
plt.ylabel('Log Return')
plt.title('Log Versus Simple Returns')
plt.legend(['Actual Relation', 'Relation If Log = Simple'])
plt.show()
```



Simple Return Advantage: Portfolio Calculations

For a portfolio of N assets with portfolio weights w_i, the portfolio return r_p is the weighted average of the returns of its assets: $r_p = \sum_{i=1}^{N} w_i r_i$. For example, for an equal-weighted portfolio with two stocks, $r_p = 0.5r_1 + 0.5r_2 = \frac{r_1+r_2}{2}$. Therefore, we cannot calculate portfolio returns with log returns because the sum of logs is the log of products. That is $\log(1+r_i) + \log(1+r_j) = \log((1+r_i) \times (1+r_j))$, which is not what we want to measure! ***We cannot perform portfolio calculations with log returns!***

Log Return Advantage: Log Returns are Additive

We compound simple returns with multiplication, *but we compound log returns with addition.* This additive property of log returns makes code simple, computations fast, and proofs easy when we must compound returns.

We compound returns from $t = 0$ to $t = T$ as follows:

$$1 + r_{0,T} = (1 + r_1) \times (1 + r_2) \times \cdots \times (1 + r_T)$$

Next, we take the log of both sides of the previous equation and use the property that the log of products is the sum of logs:

$$\log(1+r_{0,T}) = \log((1+r_1)\times(1+r_2)\times\cdots\times(1+r_T)) = \log(1+r_1)+\log(1+r_2)+\cdots+\log(1+r_T) = \sum_{t=1}^{T}\log(1+r_t)$$

Next, we exponentiate both sides of the previous equation:

$$e^{\log(1+r_{0,T})} = e^{\sum_{t=0}^{T}\log(1+r_t)}$$

Next, we use the property that $e^{\log(x)} = x$ to simplify the previous equation:

$$1 + r_{0,T} = e^{\sum_{t=0}^{T}\log(1+r_t)}$$

Finally, we subtract 1 from both sides:

$$r_{0,T} = e^{\sum_{t=0}^{T}\log(1+r_t)} - 1$$

So, the return $r_{0,T}$ from $t = 0$ to $t = T$ is the exponentiated sum of log returns. The pandas developers assume users understand the math above and focus on optimizing sums!

```
np.random.seed(42)
df = pd.DataFrame(data={'r': np.exp(np.random.randn(10_000)) - 1})

df.describe()
```

	r
count	10000.0000
mean	0.6529
std	2.1918
min	-0.9802
25%	-0.4896
50%	-0.0026
75%	0.9564
max	49.7158

We can time the calculation of 10-observation rolling returns. We use `.apply()` for the simple return version because `.rolling()` does not have a product method. We find that `.rolling()` is slower with `.apply()` than with `.sum()` by a factor of about 1,000. ***We will learn about .rolling() and .apply() in a few weeks, but they provide the best example of when to use log returns.***

```
%%timeit
df['r10_via_prod'] = (
    df['r']
    .add(1)
    .rolling(10)
    .apply(lambda x: x.prod())
    .sub(1)
)
```

```
980 ms ± 90 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

```
%%timeit
df['r10_via_sum'] = (
    df['r']
    .add(1)
    .pipe(np.log)
    .rolling(10)
    .sum()
    .pipe(np.exp)
    .sub(1)
)
```

```
1.38 ms ± 69.6  s per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

```
df.head(15)
```

	r	r10_via_prod	r10_via_sum
0	0.6433	NaN	NaN
1	-0.1291	NaN	NaN
2	0.9111	NaN	NaN
3	3.5861	NaN	NaN
4	-0.2088	NaN	NaN
5	-0.2087	NaN	NaN
6	3.8511	NaN	NaN
7	1.1542	NaN	NaN
8	-0.3747	NaN	NaN
9	0.7204	87.2886	87.2886
10	-0.3709	32.8006	32.8006
11	-0.3723	23.3617	23.3617
12	0.2737	15.2369	15.2369
13	-0.8524	-0.4774	-0.4774
14	-0.8218	-0.8823	-0.8823

```
np.allclose(df['r10_via_prod'], df['r10_via_sum'], equal_nan=True)
```

```
True
```

These two approaches calculate the same return series, but the simple-return approach using `.prod()` is about 1,000 times slower than the log-return approach using `.sum()`! ***We can use log returns to calculate total returns very quickly!***

Portfolio Math

Portfolio return r_p is the weighted average of its asset returns, so $r_p = \sum_{i=1}^N w_i r_i$. Here N is the number of assets, w_i is the weight on asset i, and $\sum_{i=1}^N w_i = 1$.

The 1/N Portfolio

The $\frac{1}{N}$ portfolio equally weights portfolio assets, so $w_1 = w_2 = \cdots = w_N = \frac{1}{N}$. If $w_i = \frac{1}{N}$, then $r_p = \sum_{i=1}^N \frac{1}{N} r_i = \frac{\sum_{i=1}^N r_i}{N} = \bar{r}$. Therefore, we can use `.mean(axis=1)` to calculate $\frac{1}{N}$ portfolio returns!

```python
df2 = yf.download(tickers='AAPL AMZN GOOG MSFT NVDA TSLA', auto_adjust=False, progress=False)
returns2 = df2['Adj Close'].pct_change().dropna()

returns2.describe()
```

Ticker	AAPL	AMZN	GOOG	MSFT	NVDA	TSLA
count	3689.0000	3689.0000	3689.0000	3689.0000	3689.0000	3689.0000
mean	0.0011	0.0012	0.0009	0.0010	0.0021	0.0021
std	0.0175	0.0205	0.0173	0.0161	0.0288	0.0362
min	-0.1286	-0.1405	-0.1110	-0.1474	-0.1876	-0.2106
25%	-0.0074	-0.0089	-0.0071	-0.0070	-0.0122	-0.0164
50%	0.0010	0.0010	0.0009	0.0007	0.0017	0.0012
75%	0.0102	0.0119	0.0093	0.0093	0.0161	0.0194
max	0.1198	0.1575	0.1605	0.1422	0.2981	0.2440

```python
returns2.mean() # implied axis=0
```

```
Ticker
AAPL   0.0011
AMZN   0.0012
GOOG   0.0009
MSFT   0.0010
NVDA   0.0021
TSLA   0.0021
dtype: float64
```

```python
rp2_via_mean = returns2.mean(axis=1)

rp2_via_mean
```

```
Date
2010-06-30   -0.0123
2010-07-01   -0.0107
2010-07-02   -0.0271
2010-07-06   -0.0223
2010-07-07    0.0254
               ...
2025-02-21   -0.0272
2025-02-24   -0.0127
```

```
2025-02-25   -0.0247
2025-02-26   -0.0055
2025-02-27   -0.0330
Length: 3689, dtype: float64
```

Note that when we apply the same portfolio weights every period, we rebalance at the same frequency as the returns data. If we have daily data, rebalance daily. If we have monthly data, we rebalance monthly, and so on.

A More General Solution

If we combine portfolio weights into vector w and the time series of asset returns into matrix $\mathbf{R}$, then we can calculate the time series of portfolio returns as $r_p = w^T\mathbf{R}$. The pandas version of this calculation is `R.dot(w)`, where `R` is a data frame of asset returns and `w` is a series or an array of portfolio weights. We can use this approach to calculate $\frac{1}{N}$ portfolio returns, too.

```python
weights2 = np.ones(returns2.shape[1]) / returns2.shape[1]

weights2
```

```
array([0.1667, 0.1667, 0.1667, 0.1667, 0.1667, 0.1667])
```

```python
rp2_via_dot = returns2.dot(weights2)

rp2_via_dot
```

```
Date
2010-06-30   -0.0123
2010-07-01   -0.0107
2010-07-02   -0.0271
2010-07-06   -0.0223
2010-07-07    0.0254
                ...
2025-02-21   -0.0272
2025-02-24   -0.0127
2025-02-25   -0.0247
2025-02-26   -0.0055
2025-02-27   -0.0330
Length: 3689, dtype: float64
```

Both approaches give the same answer!

```
np.allclose(rp2_via_mean, rp2_via_dot, equal_nan=True)
```

```
True
```

Portfolio Math Application 1: All stocks half the time or half stocks all the time?

Are you better off investing:

1. 100% in stocks 50% of the time and the riskless asset the other 50% of the time *or*
2. 50% in stocks and 50% in the riskless asset 100% of the time?

Here is a roadmap for convincing yourself with data!

Please see Kritzman (2000, Chapter 5) for a more detailed solution!

Download *annual* market and risk-free asset returns from Kenneth French's data library

```
pdr.famafrench.get_available_datasets()[:5]
```

```
['F-F_Research_Data_Factors',
 'F-F_Research_Data_Factors_weekly',
 'F-F_Research_Data_Factors_daily',
 'F-F_Research_Data_5_Factors_2x3',
 'F-F_Research_Data_5_Factors_2x3_daily']
```

```
ff = pdr.DataReader(
    name='F-F_Research_Data_Factors',
    data_source='famafrench',
    start='1900'
)
```

```
C:\Users\r.herron\AppData\Local\Temp\ipykernel_17392\304025689.py:1: FutureWarning: The argu
  ff = pdr.DataReader(
C:\Users\r.herron\AppData\Local\Temp\ipykernel_17392\304025689.py:1: FutureWarning: The argu
  ff = pdr.DataReader(
```

```python
print(ff['DESCR'])
```

```
F-F Research Data Factors
-------------------------

This file was created by CMPT_ME_BEME_RETS using the 202412 CRSP database. The 1-month TBill

  0 : (1182 rows x 4 cols)
  1 : Annual Factors: January-December (98 rows x 4 cols)
```

Convert these factors to decimal returns and calculate the market return series

```python
df3 = ff[1].div(100)
```

```python
df3['Mkt'] = df3['Mkt-RF'] + df3['RF']
```

Add a portfolio return series that is half stocks all the time

You might call this portfolio return series `Balanced`

```python
df3['Balanced'] = df3[['Mkt', 'RF']].mean(axis=1)
```

Add a portfolio return series that switches between stocks and bills every year with stocks in odd years

You might call this portfolio return series `Switching Stocks Odd`

```python
df3['Switching Stocks Odd'] = np.where(df3.index.year % 2 == 1, df3['Mkt'], df3['RF'])
```

Add a portfolio return series that switches between stocks and bills every year with stocks in even years

You might call this portfolio return series `Switching Stocks Even`

```python
df3['Switching Stocks Even'] = np.where(df3.index.year % 2 == 0, df3['Mkt'], df3['RF'])
```

```
df3.head()
```

Date	Mkt-RF	SMB	HML	RF	Mkt	Balanced	Switching Stocks Odd	Switching Stocks E
1927	0.2947	-0.0204	-0.0454	0.0312	0.3259	0.1785	0.3259	0.0312
1928	0.3539	0.0451	-0.0617	0.0356	0.3895	0.2126	0.0356	0.3895
1929	-0.1954	-0.3070	0.1167	0.0475	-0.1479	-0.0502	-0.1479	0.0475
1930	-0.3123	-0.0517	-0.1154	0.0241	-0.2882	-0.1321	0.0241	-0.2882
1931	-0.4511	0.0370	-0.1395	0.0107	-0.4404	-0.2149	-0.4404	0.0107

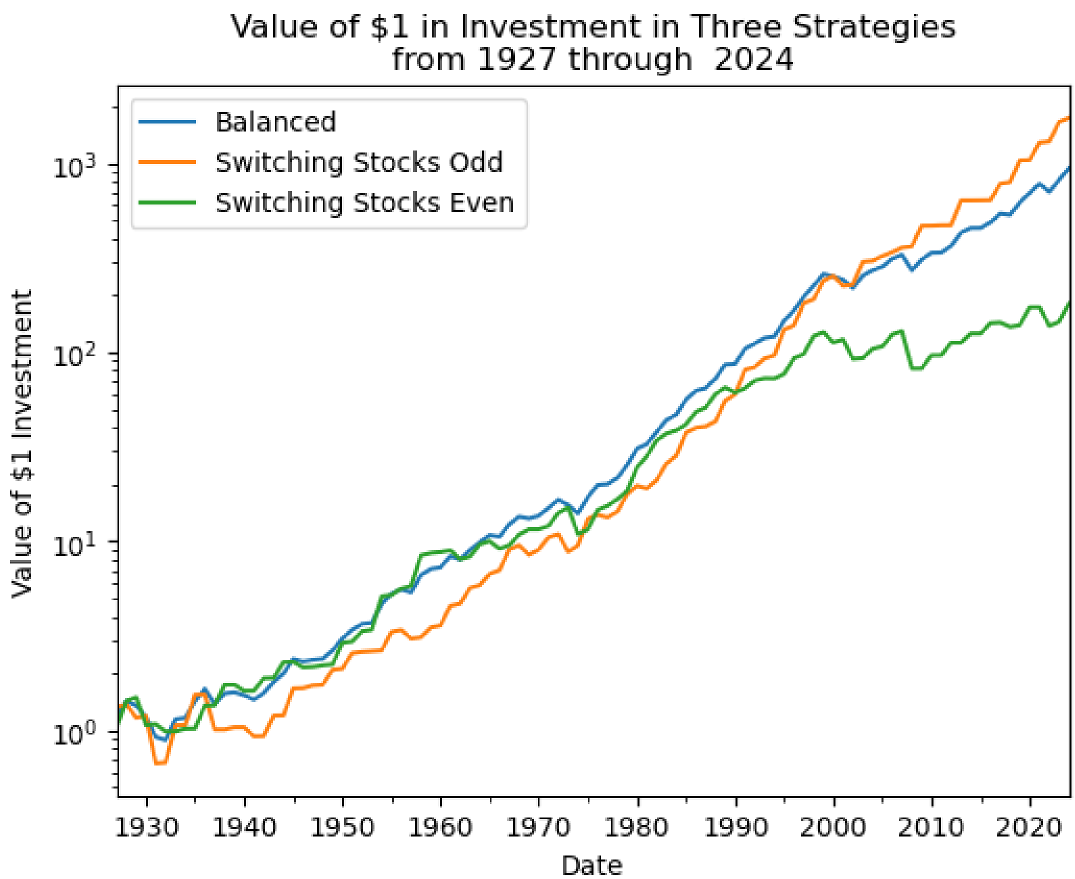
Plot the cumulative returns on a $1 investment and calculate the summary statistics for the `Balanced` and `Switching` series

Use the `.describe()` method to calcualte summary statistics.

```
portfolios = ['Balanced', 'Switching Stocks Odd', 'Switching Stocks Even']
```

```
import matplotlib.ticker as ticker
```

```
(
    df3[portfolios]
    .add(1)
    .cumprod()
    .plot()
)

plt.yscale('log')
plt.ylabel('Value of $1 Investment')
plt.title(f'Value of $1 in Investment in Three Strategies\nfrom {df3.index.year[0]} through

plt.show()
```



Which strategy do you prefer?

Why? How sure are you?

The `Switching Stocks Odd` strategy *seems* best in this example! However, this apparent superiority is largely a matter of **luck**, specifically this combination of the sample period and starting year. In this sample, the number of even and odd years is equal, but the market happened to perform better in odd years. This outcome is entirely driven by **this particular historical sequence**.

To draw broader conclusions, we must break free from the "luck of the draw" tied to this single realization of history. Below, we will explore two approaches to determine which strategy is better:

1. **Theory:** We will analyze the expected returns and variances mathematically to assess the underlying return-risk tradeoff for each strategy.

2. **Simulation:** To rely on data instead of theory, we will simulate thousands of alternative historical sequences.

By shuffling the 98 years of market data repeatedly, we can generate 10,000 random samples and evaluate the performance of each strategy across these simulations.
This approach allows us to:

- Confirm that **both strategies have the same arithmetic average return**.
- Show that the **balanced strategy consistently delivers a better return-risk tradeoff (Sharpe ratio)** when evaluated across many possible outcomes.

We will do the simulation first.

Use the `simulate()` function to simulate 10,000 different outcomes for the U.S. market

`simulate()` calculates one `Switching` return series because the randomization also randomizes the odd-year and even-year choice.

```
df3[['Mkt', 'RF']].sample(frac=1, ignore_index=True, random_state=42).head()
```

	Mkt	RF
0	0.2886	0.0837
1	0.2870	0.0421
2	0.2361	0.0004
3	0.3871	0.0033
4	-0.3674	0.0160

```
def simulate(df, cols=['Mkt', 'RF'], n_iter=10_000):
    """
    Simulates resampling of the given DataFrame columns and computes bal;anced and switching

    Parameters:
    df (pd.DataFrame): The input DataFrame.
    cols (list): List of column names to sample.
    n_iter (int): Number of iterations for simulation.

    Returns:
    pd.DataFrame: A concatenated DataFrame with simulation results.
    """
    return pd.concat(
        objs=[(
```

```
            df[cols]
            .sample(frac=1, ignore_index=True, random_state=i)
            .assign(
                Balanced=lambda x: x[cols].mean(axis=1),
                Switching=lambda x: np.where(x.index % 2 == 0, x[cols[0]], x[cols[1]])
            )
        ) for i in range(n_iter)],
        keys=range(n_iter),
        names=['Simulation', 'Year']
    )
```

```
df4 = simulate(df3)
```

```
df4.head()
```

		Mkt
Simulation	Year	
0	0	0.0077
	1	0.1633
	2	-0.1479
	3	0.2120
	4	0.3682

Calculate the summary statistics for these new `Balanced` and `Switching` series

```
df4[['Balanced', 'Switching']].agg(['mean', 'std'])
```

	Balanced	Switching
mean	0.0772	0.0774
std	0.1001	0.1487

We see that `Balanced` and `Switching` have the same mean return, but `Balanced` has much lower volatility than `Switching`! A risk-averse investor prefers `Balanced` because it has a higher return/risk ratio than `Switching`. We can quantify this ratio as the Sharpe ratio, which is the mean excess return divided by the volatility of excess returns: $S_i = \frac{\overline{r_i - r_f}}{\sigma(r_i - r_f)}$. We can calcualte excess returns and Sharpe ratios in one code snippet.

```
(
    df4
    [['Balanced', 'Switching']]
    .sub(df4['RF'], axis=0)
    .agg(lambda x: x.mean() / x.std())
    .to_frame('Sharpe ratio')
)
```

	Sharpe ratio
Balanced	0.4368
Switching	0.2961

We have to do a little more work if we want to combine mean and volatility of *raw* returns with the Sharpe ratio of *excess* returns.

```
pd.concat(
    objs=[
        df4[['Balanced', 'Switching']].agg(['mean', 'std']).transpose(),
        df4[['Balanced', 'Switching']].sub(df4['RF'], axis=0).agg(lambda x: x.mean() / x.std
    ],
    axis=1
)
```

	mean	std	Sharpe ratio
Balanced	0.0772	0.1001	0.4368
Switching	0.0774	0.1487	0.2961

Which strategy do you prefer?

Why? How sure are you?

We prefer `Balanced` because it has a Sharpe ratio about 50% greater than `Switching`! We can see this in the data above, and here is the theory.

For `Balanced`, $\sigma_p^2 = w_m^2 \sigma_m^2 + w_f^2 \sigma_f^2 + 2 w_m w_f \sigma_m \sigma_f \rho_{m,f}$. Because $\sigma_f^2 \approx 0$ and $\rho_{m,f} \approx 0$, $\sigma_p^2 \approx w_m^2 \sigma_m^2$. Therefore, for `Balanced` $\sigma_p \approx w_m \sigma_m = \frac{1}{2} \sigma_m$. We see this in the data!

```
0.5 * df3['Mkt'].std()
```

```
0.0998
```

```
df4['Balanced'].std()
```

```
0.1001
```

For `Switching`, $\sigma_p^2 = w_m \sigma_m^2 + w_f \sigma_f^2 + w_m w_f (\mu_m - \mu_f)^2$. We have a different formula because `Switching` is diversified *over time* instead at a point in time! Because $\sigma_f^2 \approx 0$ and $(\mu_m - \mu_f)^2 \approx 0$, $\sigma_p^2 \approx w_m \sigma_m^2$. Therefore, for `Balanced` $\sigma_p \approx \sqrt{w_m} \sigma_m = \sqrt{\frac{1}{2}} \sigma_m$. We see this in the data!

```
np.sqrt(0.5) * df3['Mkt'].std()
```

```
0.1411
```

```
df4['Switching'].std()
```

```
0.1487
```

The $(\mu_m - \mu_f)^2$ is close to zero but not exactly zero. We can get even closed to the observed `Switching` portfolio volatility if we consider this term!

```
np.sqrt(0.5 * df3['Mkt'].var() + 0.5 * 0.5 * (df3['Mkt'].mean() - df3['RF'].mean())**2)
```

```
0.1478
```

Here are two after-class additions to this question.

Can we add a progress bar?

Yes! We have to rewrite the `simulate()` function and replace the list comprehension with a for loop. However, this addition requires the `tqdm` package. To avoid any conflicts, I will not install this package and provide the code as markdown instead of executable code.

```
from tqdm import tqdm

def simulate(df, cols=['Mkt', 'RF'], n_iter=10_000):
    """
    Simulates resampling of the given DataFrame columns and computes balanced
    and switching portfolio returns.

    Parameters:
    -----------
    df : pd.DataFrame
        The input DataFrame.
    cols : list
        List of column names to sample.
    n_iter : int
        Number of iterations for simulation.

    Returns:
    --------
    pd.DataFrame
        A concatenated DataFrame with simulation results.
    """
    all_resamples = []

    # Use tqdm to visualize progress
    for i in tqdm(range(n_iter), desc="Simulating"):
        # Random re-sampling
        resampled = (
            df[cols]
            .sample(frac=1, ignore_index=True, random_state=i)
            .assign(
                Balanced=lambda x: x[cols].mean(axis=1),
                Switching=lambda x: np.where(
                    x.index % 2 == 0,
                    x[cols[0]],
                    x[cols[1]]
                )
            )
        )
        all_resamples.append(resampled)

    # Concatenate final results
    return pd.concat(all_resamples, keys=range(n_iter), names=['Simulation', 'Year'])
```

Ideally, we calculate the mean and volatility of *each simulation*, then take the average

We can do this easily with either `.groupby()` of `.pivot_table()`!

```
(
    df4
    .groupby(level='Simulation')
    [['Balanced', 'Switching']]
    .agg(['mean', 'std'])
    .mean()
    .unstack()
)
```

	mean	std
Balanced	0.0772	0.1006
Switching	0.0774	0.1488

These statistics are similar, but not identical!

Portfolio Math Application 2: What are the benefits of diversification?

Use random portfolios of S&P 100 stocks of various portfolio sizes to show that portfolio volatility falls quickly, then slowly, then not at all as we increase portfolio size.

Download daily data for the stocks in the S&P 100

Wikipedia provides tickers for the stocks in the S&P 100. Use a list comprehension to replace `.` in tickers with `-` for compatability with Yahoo! Finance.

```
wiki = pd.read_html('https://en.wikipedia.org/wiki/S%26P_100')
tickers = [i.replace('.', '-') for i in wiki[2]['Symbol']]
data = yf.download(tickers=tickers, auto_adjust=False, progress=False).iloc[:-1]
```

Calculate the past five years of dailty returns for these stocks

```
returns = (
    data
    ['Adj Close']
    .dropna(axis=1, how='all')
    .pct_change()
    .iloc[-5*252:]
)
```

Calculate the volatilies of 20 equal-weighted random portfolios of various portfolio sizes

Random portfolios should have portfolio sizes of 1, 2, 4, 6, 8, 10, 20, 30, 40, or 50 stocks each.

You can combine the `.sample(n=?, axis=1, random_state=?)`, `.mean(axis=1)`, and `.std()` to calculate the volatilities of equal-weighted portfolios. You can collect these volatilies in a list of lists built with two `for` loops or list comprehensions. Replace the `?`s in `.sample()` with loop counters. The inner loop will calculate a portfolio volatilty for each portfolio size, and the outer loop will collect 20 versions of each portfolio. Using the outer loop counter for `random_state=` makes your analysis repeatable!

```
portfolio_size = [1, 2, 4, 6, 8, 10, 20, 30, 40, 50]
portfolio_number = 20

list_of_volatilities = []
for i in range(portfolio_number):
    list_of_volatilities.append([returns.sample(n=j, axis=1, random_state=i).mean(axis=1).st
```

```
list_of_volatilities[0]
```

```
[0.0153,
 0.0127,
 0.0121,
 0.0118,
 0.0134,
 0.0129,
 0.0128,
 0.0129,
 0.0129,
 0.0129]
```

Combine this list of lists into a data frame

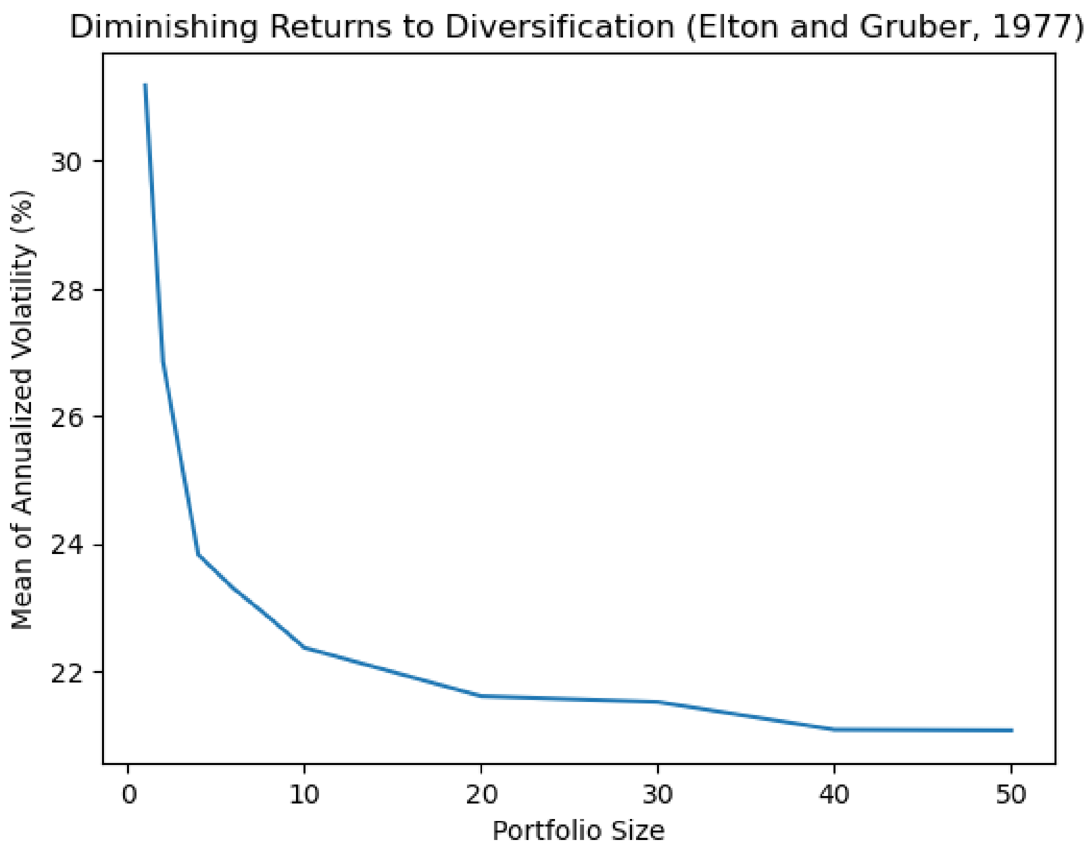
```
volatilities = (
    pd.DataFrame(
        data=list_of_volatilities,
        index=range(1, 1+portfolio_number),
        columns=portfolio_size
    )
    .rename_axis(index='Portfolio Number', columns='Portfolio Size')
)
```

```
volatilities
```

Portfolio Size Portfolio Number	1	2	4	6	8	10	20	30	40	50
1	0.0153	0.0127	0.0121	0.0118	0.0134	0.0129	0.0128	0.0129	0.0129	0.0129
2	0.0193	0.0144	0.0127	0.0145	0.0144	0.0145	0.0137	0.0136	0.0136	0.0133
3	0.0160	0.0165	0.0161	0.0151	0.0142	0.0144	0.0132	0.0134	0.0133	0.0134
4	0.0193	0.0177	0.0172	0.0171	0.0157	0.0156	0.0141	0.0141	0.0142	0.0139
5	0.0205	0.0199	0.0151	0.0164	0.0141	0.0133	0.0127	0.0131	0.0128	0.0131
6	0.0182	0.0157	0.0123	0.0136	0.0136	0.0139	0.0137	0.0143	0.0137	0.0136
7	0.0199	0.0174	0.0171	0.0160	0.0148	0.0142	0.0124	0.0133	0.0130	0.0128
8	0.0280	0.0178	0.0142	0.0149	0.0144	0.0131	0.0141	0.0138	0.0134	0.0134
9	0.0150	0.0136	0.0115	0.0116	0.0117	0.0119	0.0131	0.0132	0.0131	0.0133
10	0.0141	0.0171	0.0164	0.0150	0.0146	0.0147	0.0138	0.0131	0.0128	0.0126
11	0.0285	0.0210	0.0185	0.0170	0.0163	0.0167	0.0143	0.0142	0.0136	0.0137
12	0.0172	0.0193	0.0148	0.0154	0.0137	0.0140	0.0137	0.0133	0.0125	0.0128
13	0.0150	0.0131	0.0124	0.0126	0.0142	0.0134	0.0138	0.0133	0.0131	0.0135
14	0.0280	0.0198	0.0160	0.0147	0.0147	0.0137	0.0137	0.0131	0.0133	0.0134
15	0.0282	0.0197	0.0152	0.0142	0.0148	0.0142	0.0143	0.0140	0.0133	0.0134
16	0.0188	0.0150	0.0148	0.0136	0.0138	0.0140	0.0138	0.0141	0.0137	0.0136
17	0.0165	0.0132	0.0149	0.0141	0.0133	0.0128	0.0130	0.0133	0.0133	0.0131
18	0.0213	0.0198	0.0177	0.0166	0.0161	0.0156	0.0145	0.0142	0.0135	0.0139
19	0.0132	0.0136	0.0143	0.0140	0.0150	0.0145	0.0139	0.0134	0.0131	0.0127
20	0.0205	0.0209	0.0169	0.0154	0.0151	0.0143	0.0137	0.0134	0.0132	0.0132

Calculate the mean volatility for each portfolio size and replicate the plot above

```
volatilities.mul(100 * np.sqrt(252)).mean().plot()
plt.xlabel('Portfolio Size')
plt.ylabel('Mean of Annualized Volatility (%)')
plt.title('Diminishing Returns to Diversification (Elton and Gruber, 1977)')
plt.show()
```



Herron Topic 1 - Log and Simple Returns, Portfolio Math, and Applications - Sec 04

This notebook covers two topics:

1. Log and simple returns
2. Portfolio returns, plus two applications of portfolio returns

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import pandas_datareader as pdr
import yfinance as yf
```

```
%precision 4
pd.options.display.float_format = '{:.4f}'.format
# %config InlineBackend.figure_format = 'retina'
```

Log and Simple Returns

We will typically use *simple* returns, calculated as $r_{simple,t} = \frac{p_t + d_t - p_{t-1}}{p_{t-1}}$. This simple return is the return that investors earn on their investments. We can calculate simple returns from Yahoo Finance data with the `.pct_change()` method on the adjusted close column (i.e., `Adj Close`), which adjusts for dividends and splits. The adjusted close column is a reverse-engineered close price (i.e., end-of-trading-day price) that incorporates dividends and splits, making simple return calculations easy.

However, we may see *log* returns elsewhere, which are the (natural) log of one plus simple returns: $r_{log,t} = \log(1 + r_{simple,t})$. Therefore, we calculate log returns as either the log of one plus simple returns or the difference of the logs of the adjusted close column. Log returns are also known as *continuously-compounded* returns.

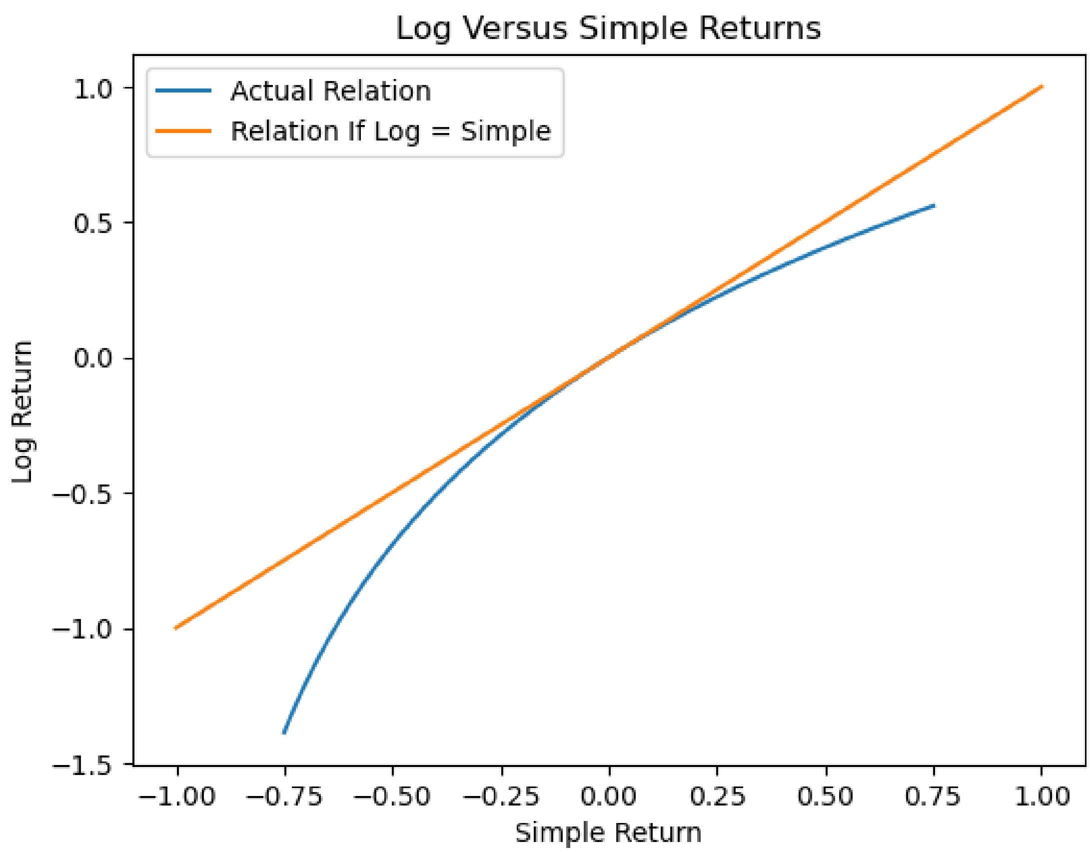
This section explains the differences between simple and log returns and where each is appropriate.

Simple and Log Returns are Similar for Small Returns

$x \approx \log(1+x)$ for small values of x, so simple returns and log returns are similar for small returns. Returns are typically small at daily and monthly horizons, so the difference between simple and log returns is small at daily and monthly horizons. The following figure shows that $r_{simple,t} \approx r_{log,t}$ for small values of r.

```
simpler = np.linspace(-0.75, 0.75, 100)
logr = np.log1p(simpler)
```

```
plt.plot(simpler, logr)
plt.plot([-1, 1], [-1, 1])
plt.xlabel('Simple Return')
plt.ylabel('Log Return')
plt.title('Log Versus Simple Returns')
plt.legend(['Actual Relation', 'Relation If Log = Simple'])
plt.show()
```



Simple Return Advantage: Portfolio Calculations

For a portfolio of N assets with portfolio weights w_i, the portfolio return r_p is the weighted average of the returns of its assets: $r_p = \sum_{i=1}^N w_i r_i$. For example, for an equal-weighted portfolio with two stocks, $r_p = 0.5 r_1 + 0.5 r_2 = \frac{r_1 + r_2}{2}$. Therefore, we cannot calculate portfolio returns with log returns because the sum of logs is the log of products. That is $\log(1 + r_i) + \log(1 + r_j) = \log((1 + r_i) \times (1 + r_j))$, which is not what we want to measure! ***We cannot perform portfolio calculations with log returns!***

Log Return Advantage: Log Returns are Additive

We compound simple returns with multiplication, *but we compound log returns with addition.* This additive property of log returns makes code simple, computations fast, and proofs easy when we must compound returns.

We compound returns from $t = 0$ to $t = T$ as follows:

$$1 + r_{0,T} = (1 + r_1) \times (1 + r_2) \times \dots \times (1 + r_T)$$

Next, we take the log of both sides of the previous equation and use the property that the log of products is the sum of logs:

$$\log(1+r_{0,T}) = \log((1+r_1)\times(1+r_2)\times\dots\times(1+r_T)) = \log(1+r_1)+\log(1+r_2)+\dots+\log(1+r_T) = \sum_{t=1}^{T} \log(1+r_t)$$

Next, we exponentiate both sides of the previous equation:

$$e^{\log(1+r_{0,T})} = e^{\sum_{t=0}^{T} \log(1+r_t)}$$

Next, we use the property that $e^{\log(x)} = x$ to simplify the previous equation:

$$1 + r_{0,T} = e^{\sum_{t=0}^{T} \log(1+r_t)}$$

Finally, we subtract 1 from both sides:

$$r_{0,T} = e^{\sum_{t=0}^{T} \log(1+r_t)} - 1$$

So, the return $r_{0,T}$ from $t = 0$ to $t = T$ is the exponentiated sum of log returns. The pandas developers assume users understand the math above and focus on optimizing sums!

```
np.random.seed(42)
df = pd.DataFrame(data={'r': np.exp(np.random.randn(10_000)) - 1})

df.describe()
```

	r
count	10000.0000
mean	0.6529
std	2.1918
min	-0.9802
25%	-0.4896
50%	-0.0026
75%	0.9564
max	49.7158

We can time the calculation of 10-observation rolling returns. We use `.apply()` for the simple return version because `.rolling()` does not have a product method. We find that `.rolling()` is slower with `.apply()` than with `.sum()` by a factor of about 1,000. ***We will learn about `.rolling()` and `.apply()` in a few weeks, but they provide the best example of when to use log returns.***

```
%%timeit
df['r10_via_prod'] = (
    df['r']
    .add(1)
    .rolling(10)
    .apply(lambda x: x.prod())
    .sub(1)
)
```

```
208 ms ± 40.1 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
%%timeit
df['r10_via_sum'] = (
    df['r']
    .add(1)
    .pipe(np.log)
    .rolling(10)
    .sum()
    .pipe(np.exp)
    .sub(1)
)
```

```
1.49 ms ± 443  s per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

```
df.head(15)
```

	r	r10_via_prod	r10_via_sum
0	0.6433	NaN	NaN
1	-0.1291	NaN	NaN
2	0.9111	NaN	NaN
3	3.5861	NaN	NaN
4	-0.2088	NaN	NaN
5	-0.2087	NaN	NaN
6	3.8511	NaN	NaN
7	1.1542	NaN	NaN
8	-0.3747	NaN	NaN
9	0.7204	87.2886	87.2886
10	-0.3709	32.8006	32.8006
11	-0.3723	23.3617	23.3617
12	0.2737	15.2369	15.2369
13	-0.8524	-0.4774	-0.4774
14	-0.8218	-0.8823	-0.8823

```
np.allclose(df['r10_via_prod'], df['r10_via_sum'], equal_nan=True)
```

```
True
```

These two approaches calculate the same return series, but the simple-return approach using `.prod()` is about 1,000 times slower than the log-return approach using `.sum()`! ***We can use log returns to calculate total returns very quickly!***

Portfolio Math

Portfolio return r_p is the weighted average of its asset returns, so $r_p = \sum_{i=1}^N w_i r_i$. Here N is the number of assets, w_i is the weight on asset i, and $\sum_{i=1}^N w_i = 1$.

The 1/N Portfolio

The $\frac{1}{N}$ portfolio equally weights portfolio assets, so $w_1 = w_2 = \dots = w_N = \frac{1}{N}$. If $w_i = \frac{1}{N}$, then $r_p = \sum_{i=1}^N \frac{1}{N} r_i = \frac{\sum_{i=1}^N r_i}{N} = \bar{r}$. Therefore, we can use `.mean(axis=1)` to calculate $\frac{1}{N}$ portfolio returns!

```python
df2 = yf.download(tickers='AAPL AMZN GOOG MSFT NVDA TSLA', auto_adjust=False, progress=False)
returns2 = df2['Adj Close'].pct_change().dropna()

returns2.describe()
```

Ticker	AAPL	AMZN	GOOG	MSFT	NVDA	TSLA
count	3689.0000	3689.0000	3689.0000	3689.0000	3689.0000	3689.0000
mean	0.0011	0.0012	0.0009	0.0010	0.0021	0.0021
std	0.0175	0.0205	0.0173	0.0161	0.0288	0.0362
min	-0.1286	-0.1405	-0.1110	-0.1474	-0.1876	-0.2106
25%	-0.0074	-0.0089	-0.0071	-0.0070	-0.0122	-0.0164
50%	0.0010	0.0010	0.0009	0.0007	0.0017	0.0012
75%	0.0102	0.0119	0.0093	0.0093	0.0161	0.0194
max	0.1198	0.1575	0.1605	0.1422	0.2981	0.2440

```python
returns2.mean() # implied axis=0
```

```
Ticker
AAPL   0.0011
AMZN   0.0012
GOOG   0.0009
MSFT   0.0010
NVDA   0.0021
TSLA   0.0021
dtype: float64
```

```python
rp2_via_mean = returns2.mean(axis=1)

rp2_via_mean
```

```
Date
2010-06-30   -0.0123
2010-07-01   -0.0107
2010-07-02   -0.0271
2010-07-06   -0.0223
2010-07-07    0.0254
               ...
2025-02-21   -0.0272
2025-02-24   -0.0127
```

```
2025-02-25   -0.0247
2025-02-26   -0.0055
2025-02-27   -0.0330
Length: 3689, dtype: float64
```

Note that when we apply the same portfolio weights every period, we rebalance at the same frequency as the returns data. If we have daily data, rebalance daily. If we have monthly data, we rebalance monthly, and so on.

A More General Solution

If we combine portfolio weights into vector w and the time series of asset returns into matrix $\mathbf{R}$, then we can calculate the time series of portfolio returns as $r_p = w^T\mathbf{R}$. The pandas version of this calculation is `R.dot(w)`, where `R` is a data frame of asset returns and `w` is a series or an array of portfolio weights. We can use this approach to calculate $\frac{1}{N}$ portfolio returns, too.

```python
returns2.shape
```

```
(3689, 6)
```

```python
weights2 = np.ones(returns2.shape[1]) / returns2.shape[1]

weights2
```

```
array([0.1667, 0.1667, 0.1667, 0.1667, 0.1667, 0.1667])
```

```python
rp2_via_dot = returns2.dot(weights2)

rp2_via_dot
```

```
Date
2010-06-30   -0.0123
2010-07-01   -0.0107
2010-07-02   -0.0271
2010-07-06   -0.0223
2010-07-07    0.0254
               ...
2025-02-21   -0.0272
2025-02-24   -0.0127
```

```
2025-02-25   -0.0247
2025-02-26   -0.0055
2025-02-27   -0.0330
Length: 3689, dtype: float64
```

Both approaches give the same answer!

```
np.allclose(rp2_via_mean, rp2_via_dot, equal_nan=True)
```

```
True
```

Portfolio Math Application 1: All stocks half the time or half stocks all the time?

Are you better off investing:

1. 100% in stocks 50% of the time and the riskless asset the other 50% of the time *or*
2. 50% in stocks and 50% in the riskless asset 100% of the time?

Here is a roadmap for convincing yourself with data!

Please see Kritzman (2000, Chapter 5) for a more detailed solution!

Download *annual* market and risk-free asset returns from Kenneth French's data library

```
pdr.famafrench.get_available_datasets()[:5]
```

```
['F-F_Research_Data_Factors',
 'F-F_Research_Data_Factors_weekly',
 'F-F_Research_Data_Factors_daily',
 'F-F_Research_Data_5_Factors_2x3',
 'F-F_Research_Data_5_Factors_2x3_daily']
```

```
ff = pdr.DataReader(
    name='F-F_Research_Data_Factors',
    data_source='famafrench',
    start='1900'
)
```

```
C:\Users\r.herron\AppData\Local\Temp\ipykernel_24988\304025689.py:1: FutureWarning: The argu
  ff = pdr.DataReader(
C:\Users\r.herron\AppData\Local\Temp\ipykernel_24988\304025689.py:1: FutureWarning: The argu
  ff = pdr.DataReader(
```

```
print(ff['DESCR'])
```

```
F-F Research Data Factors
-------------------------

This file was created by CMPT_ME_BEME_RETS using the 202412 CRSP database. The 1-month TBill

  0 : (1182 rows x 4 cols)
  1 : Annual Factors: January-December (98 rows x 4 cols)
```

Convert these factors to decimal returns and calculate the market return series

```
df3 = ff[1].div(100)
```

```
df3['Mkt'] = df3['Mkt-RF'] + df3['RF']
```

Add a portfolio return series that is half stocks all the time

You might call this portfolio return series `Balanced`

```
df3['Balanced'] = df3[['Mkt', 'RF']].mean(axis=1)
```

Add a portfolio return series that switches between stocks and bills every year with stocks in odd years

You might call this portfolio return series `Switching Stocks Odd`

```
df3['Switching Stocks Odd'] = np.where(df3.index.year % 2 == 1, df3['Mkt'], df3['RF'])
```

Add a portfolio return series that switches between stocks and bills every year with stocks in even years

You might call this portfolio return series `Switching Stocks Even`

```
df3['Switching Stocks Even'] = np.where(df3.index.year % 2 == 0, df3['Mkt'], df3['RF'])
```

```
df3.head()
```

Date	Mkt-RF	SMB	HML	RF	Mkt	Balanced	Switching Stocks Odd	Switching Stocks E
1927	0.2947	-0.0204	-0.0454	0.0312	0.3259	0.1785	0.3259	0.0312
1928	0.3539	0.0451	-0.0617	0.0356	0.3895	0.2126	0.0356	0.3895
1929	-0.1954	-0.3070	0.1167	0.0475	-0.1479	-0.0502	-0.1479	0.0475
1930	-0.3123	-0.0517	-0.1154	0.0241	-0.2882	-0.1321	0.0241	-0.2882
1931	-0.4511	0.0370	-0.1395	0.0107	-0.4404	-0.2149	-0.4404	0.0107

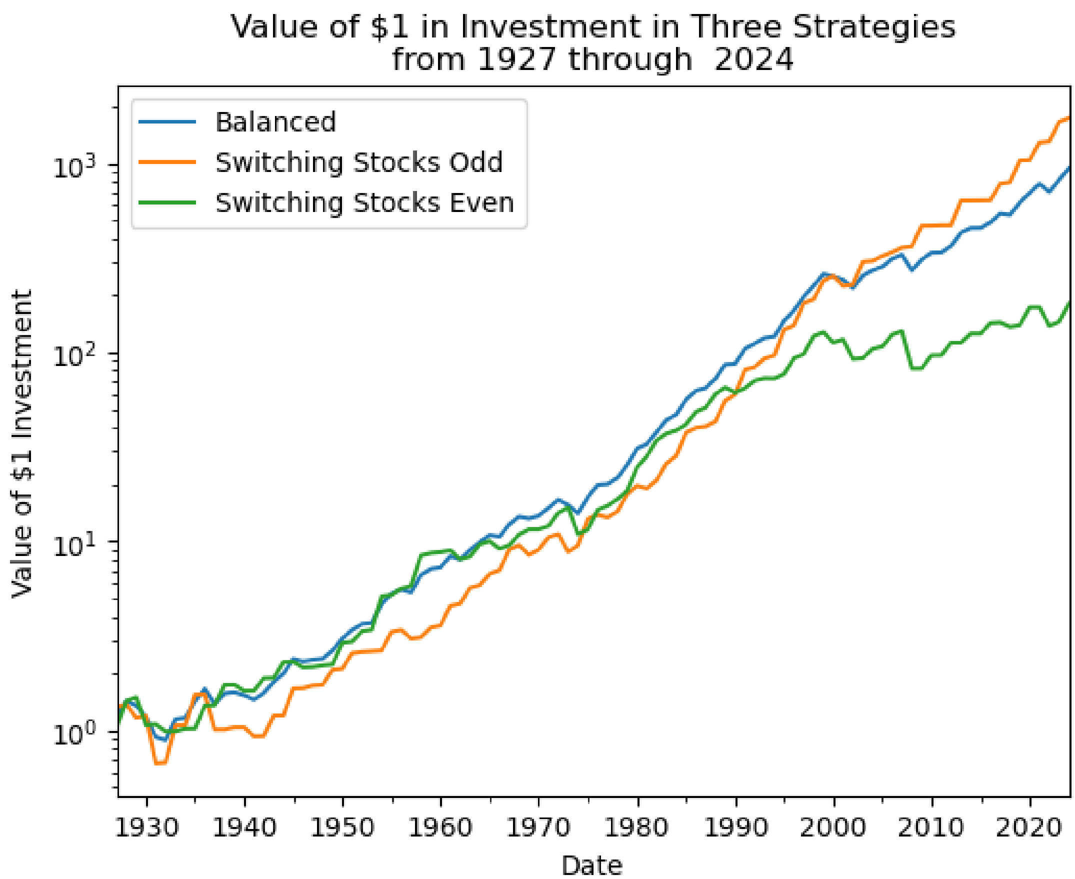
Plot the cumulative returns on a $1 investment and calculate the summary statistics for the `Balanced` and `Switching` series

Use the `.describe()` method to calculate summary statistics.

```
portfolios = ['Balanced', 'Switching Stocks Odd', 'Switching Stocks Even']
```

```
import matplotlib.ticker as ticker
```

```
(
    df3[portfolios]
    .add(1)
    .cumprod()
    .plot()
)

plt.yscale('log')
plt.ylabel('Value of $1 Investment')
plt.title(f'Value of $1 in Investment in Three Strategies\nfrom {df3.index.year[0]} through

plt.show()
```



Which strategy do you prefer?

Why? How sure are you?

The `Switching Stocks Odd` strategy *seems* best in this example! However, this apparent superiority is largely a matter of **luck**, specifically this combination of the sample period and starting year. In this sample, the number of even and odd years is equal, but the market happened to perform better in odd years. This outcome is entirely driven by **this particular historical sequence**.

To draw broader conclusions, we must break free from the "luck of the draw" tied to this single realization of history. Below, we will explore two approaches to determine which strategy is better:

1. **Theory:** We will analyze the expected returns and variances mathematically to assess the underlying return-risk tradeoff for each strategy.

2. **Simulation:** To rely on data instead of theory, we will simulate thousands of alternative historical sequences.

By shuffling the 98 years of market data repeatedly, we can generate 10,000 random samples and evaluate the performance of each strategy across these simulations.
This approach allows us to:

- Confirm that **both strategies have the same arithmetic average return**.
- Show that the **balanced strategy consistently delivers a better return-risk tradeoff (Sharpe ratio)** when evaluated across many possible outcomes.

We will do the simulation first.

Use the `simulate()` function to simulate 10,000 different outcomes for the U.S. market

`simulate()` calculates one `Switching` return series because the randomization also randomizes the odd-year and even-year choice.

```
df3[['Mkt', 'RF']].sample(frac=1, ignore_index=True, random_state=42).head()
```

	Mkt	RF
0	0.2886	0.0837
1	0.2870	0.0421
2	0.2361	0.0004
3	0.3871	0.0033
4	-0.3674	0.0160

```
def simulate(df, cols=['Mkt', 'RF'], n_iter=10_000):
    """
    Simulates resampling of the given DataFrame columns and computes bal;anced and switching

    Parameters:
    df (pd.DataFrame): The input DataFrame.
    cols (list): List of column names to sample.
    n_iter (int): Number of iterations for simulation.

    Returns:
    pd.DataFrame: A concatenated DataFrame with simulation results.
    """
    return pd.concat(
        objs=[(
```

```
            df[cols]
            .sample(frac=1, ignore_index=True, random_state=i)
            .assign(
                Balanced=lambda x: x[cols].mean(axis=1),
                Switching=lambda x: np.where(x.index % 2 == 0, x[cols[0]], x[cols[1]])
            )
        ) for i in range(n_iter)],
        keys=range(n_iter),
        names=['Simulation', 'Year']
    )
```

```
df4 = simulate(df3)
```

```
df4.head()
```

		Mkt
Simulation	Year	
0	0	0.0077
	1	0.1633
	2	-0.1479
	3	0.2120
	4	0.3682

Calculate the summary statistics for these new `Balanced` and `Switching` series

```
df4[['Balanced', 'Switching']].agg(['mean', 'std'])
```

	Balanced	Switching
mean	0.0772	0.0774
std	0.1001	0.1487

We see that `Balanced` and `Switching` have the same mean return, but `Balanced` has much lower volatility than `Switching`! A risk-averse investor prefers `Balanced` because it has a higher return/risk ratio than `Switching`. We can quantify this ratio as the Sharpe ratio, which is the mean excess return divided by the volatility of excess returns: $S_i = \frac{\overline{r_i - r_f}}{\sigma(r_i - r_f)}$. We can calcualte excess returns and Sharpe ratios in one code snippet.

```
(
    df4
    [['Balanced', 'Switching']]
    .sub(df4['RF'], axis=0)
    .agg(lambda x: x.mean() / x.std())
    .to_frame('Sharpe ratio')
)
```

	Sharpe ratio
Balanced	0.4368
Switching	0.2961

We have to do a little more work if we want to combine mean and volatility of *raw* returns with the Sharpe ratio of *excess* returns.

```
pd.concat(
    objs=[
        df4[['Balanced', 'Switching']].agg(['mean', 'std']).transpose(),
        df4[['Balanced', 'Switching']].sub(df4['RF'], axis=0).agg(lambda x: x.mean() / x.std
    ],
    axis=1
)
```

	mean	std	Sharpe ratio
Balanced	0.0772	0.1001	0.4368
Switching	0.0774	0.1487	0.2961

Which strategy do you prefer?

Why? How sure are you?

We prefer `Balanced` because it has a Sharpe ratio about 50% greater than `Switching`! We can see this in the data above, and here is the theory.

For `Balanced`, $\sigma_p^2 = w_m^2\sigma_m^2 + w_f^2\sigma_f^2 + 2w_m w_f \sigma_m \sigma_f \rho_{m,f}$. Because $\sigma_f^2 \approx 0$ and $\rho_{m,f} \approx 0$, $\sigma_p^2 \approx w_m^2\sigma_m^2$. Therefore, for `Balanced` $\sigma_p \approx w_m \sigma_m = \frac{1}{2}\sigma_m$. We see this in the data!

```
0.5 * df3['Mkt'].std()
```

```
0.0998
```

```
df4['Balanced'].std()
```

```
0.1001
```

For `Switching`, $\sigma_p^2 = w_m \sigma_m^2 + w_f \sigma_f^2 + w_m w_f (\mu_m - \mu_f)^2$. We have a different formula because `Switching` is diversified *over time* instead at a point in time! Because $\sigma_f^2 \approx 0$ and $(\mu_m - \mu_f)^2 \approx 0$, $\sigma_p^2 \approx w_m \sigma_m^2$. Therefore, for `Balanced` $\sigma_p \approx \sqrt{w_m} \sigma_m = \sqrt{\frac{1}{2}} \sigma_m$. We see this in the data!

```
np.sqrt(0.5) * df3['Mkt'].std()
```

```
0.1411
```

```
df4['Switching'].std()
```

```
0.1487
```

The $(\mu_m - \mu_f)^2$ is close to zero but not exactly zero. We can get even closed to the observed `Switching` portfolio volatility if we consider this term!

```
np.sqrt(0.5 * df3['Mkt'].var() + 0.5 * 0.5 * (df3['Mkt'].mean() - df3['RF'].mean())**2)
```

```
0.1478
```

Here are two after-class additions to this question.

Can we add a progress bar?

Yes! We have to rewrite the `simulate()` function and replace the list comprehension with a for loop. However, this addition requires the `tqdm` package. To avoid any conflicts, I will not install this package and provide the code as markdown instead of executable code.

```
from tqdm import tqdm

def simulate(df, cols=['Mkt', 'RF'], n_iter=10_000):
    """
    Simulates resampling of the given DataFrame columns and computes balanced
    and switching portfolio returns.

    Parameters:
    -----------
    df : pd.DataFrame
        The input DataFrame.
    cols : list
        List of column names to sample.
    n_iter : int
        Number of iterations for simulation.

    Returns:
    --------
    pd.DataFrame
        A concatenated DataFrame with simulation results.
    """
    all_resamples = []

    # Use tqdm to visualize progress
    for i in tqdm(range(n_iter), desc="Simulating"):
        # Random re-sampling
        resampled = (
            df[cols]
            .sample(frac=1, ignore_index=True, random_state=i)
            .assign(
                Balanced=lambda x: x[cols].mean(axis=1),
                Switching=lambda x: np.where(
                    x.index % 2 == 0,
                    x[cols[0]],
                    x[cols[1]]
                )
            )
        )
        all_resamples.append(resampled)

    # Concatenate final results
    return pd.concat(all_resamples, keys=range(n_iter), names=['Simulation', 'Year'])
```

Ideally, we calculate the mean and volatility of *each simulation*, then take the average

We can do this easily with either `.groupby()` of `.pivot_table()`!

```
(
    df4
    .groupby(level='Simulation')
    [['Balanced', 'Switching']]
    .agg(['mean', 'std'])
    .mean()
    .unstack()
)
```

	mean	std
Balanced	0.0772	0.1006
Switching	0.0774	0.1488

These statistics are similar, but not identical!

Portfolio Math Application 2: What are the benefits of diversification?

Use random portfolios of S&P 100 stocks of various portfolio sizes to show that portfolio volatility falls quickly, then slowly, then not at all as we increase portfolio size.

Download daily data for the stocks in the S&P 100

Wikipedia provides tickers for the stocks in the S&P 100. Use a list comprehension to replace `.` in tickers with `-` for compatability with Yahoo! Finance.

```
wiki = pd.read_html('https://en.wikipedia.org/wiki/S%26P_100')
tickers = [i.replace('.', '-') for i in wiki[2]['Symbol']]
data = yf.download(tickers=tickers, auto_adjust=False, progress=False).iloc[:-1]
```

Calculate the past five years of dailty returns for these stocks

```python
returns = (
    data
    ['Adj Close']
    .dropna(axis=1, how='all')
    .pct_change()
    .iloc[-5*252:]
)
```

Calculate the volatilies of 20 equal-weighted random portfolios of various portfolio sizes

Random portfolios should have portfolio sizes of 1, 2, 4, 6, 8, 10, 20, 30, 40, or 50 stocks each.

You can combine the `.sample(n=?, axis=1, random_state=?)`, `.mean(axis=1)`, and `.std()` to calculate the volatilities of equal-weighted portfolios. You can collect these volatilies in a list of lists built with two `for` loops or list comprehensions. Replace the `?`s in `.sample()` with loop counters. The inner loop will calculate a portfolio volatilty for each portfolio size, and the outer loop will collect 20 versions of each portfolio. Using the outer loop counter for `random_state=` makes your analysis repeatable!

```python
portfolio_size = [1, 2, 4, 6, 8, 10, 20, 30, 40, 50]
portfolio_number = 20

list_of_volatilities = []
for i in range(portfolio_number):
    list_of_volatilities.append([returns.sample(n=j, axis=1, random_state=i).mean(axis=1).st
```

```python
list_of_volatilities[0]
```

```
[0.0153,
 0.0127,
 0.0121,
 0.0118,
 0.0134,
 0.0129,
 0.0128,
 0.0129,
 0.0129,
 0.0129]
```

Combine this list of lists into a data frame

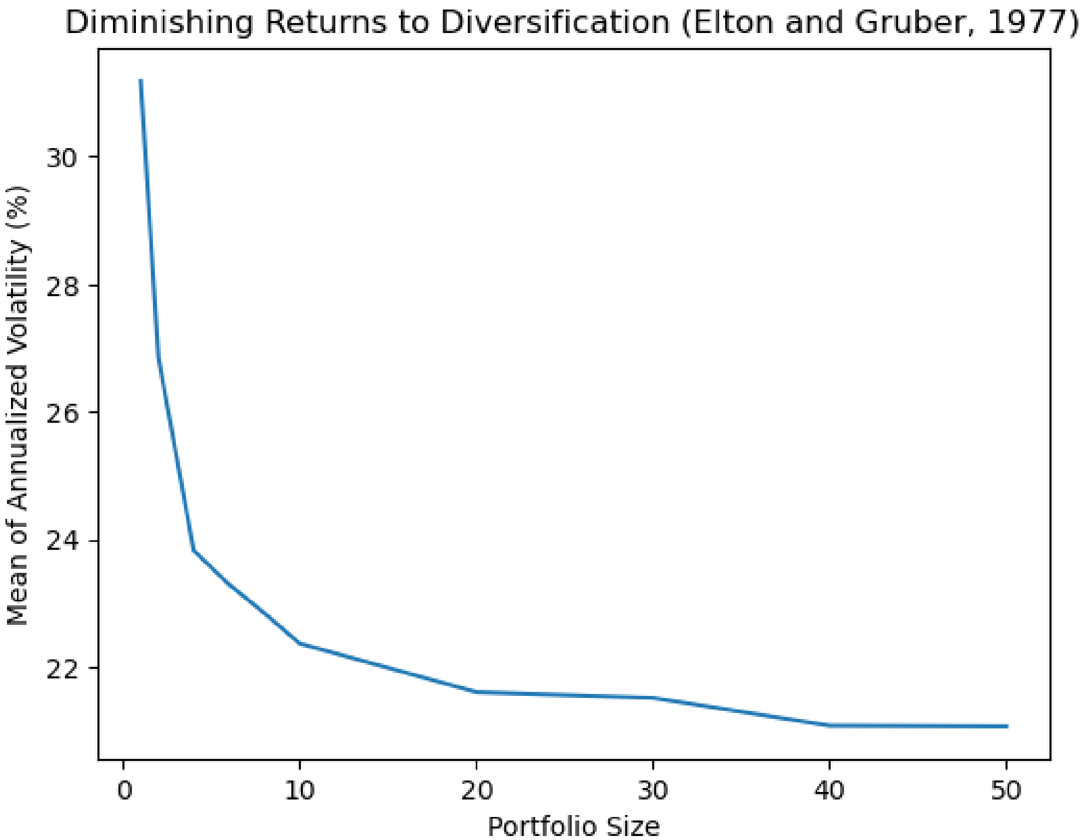
```
volatilities = (
    pd.DataFrame(
        data=list_of_volatilities,
        index=range(1, 1+portfolio_number),
        columns=portfolio_size
    )
    .rename_axis(index='Portfolio Number', columns='Portfolio Size')
)
```

```
volatilities
```

Portfolio Size Portfolio Number	1	2	4	6	8	10	20	30	40	50
1	0.0153	0.0127	0.0121	0.0118	0.0134	0.0129	0.0128	0.0129	0.0129	0.0129
2	0.0193	0.0144	0.0127	0.0145	0.0144	0.0145	0.0137	0.0136	0.0136	0.0133
3	0.0160	0.0165	0.0161	0.0151	0.0142	0.0144	0.0132	0.0134	0.0133	0.0134
4	0.0193	0.0177	0.0172	0.0171	0.0157	0.0156	0.0141	0.0141	0.0142	0.0139
5	0.0205	0.0199	0.0151	0.0164	0.0141	0.0133	0.0127	0.0131	0.0128	0.0131
6	0.0182	0.0157	0.0123	0.0136	0.0136	0.0139	0.0137	0.0143	0.0137	0.0136
7	0.0199	0.0174	0.0171	0.0160	0.0148	0.0142	0.0124	0.0133	0.0130	0.0128
8	0.0280	0.0178	0.0142	0.0149	0.0144	0.0131	0.0141	0.0138	0.0134	0.0134
9	0.0150	0.0136	0.0115	0.0116	0.0117	0.0119	0.0131	0.0132	0.0131	0.0133
10	0.0141	0.0171	0.0164	0.0150	0.0146	0.0147	0.0138	0.0131	0.0128	0.0126
11	0.0285	0.0210	0.0185	0.0170	0.0163	0.0167	0.0143	0.0142	0.0136	0.0137
12	0.0172	0.0193	0.0148	0.0154	0.0137	0.0140	0.0137	0.0133	0.0125	0.0128
13	0.0150	0.0131	0.0124	0.0126	0.0142	0.0134	0.0138	0.0133	0.0131	0.0135
14	0.0280	0.0198	0.0160	0.0147	0.0147	0.0137	0.0137	0.0131	0.0133	0.0134
15	0.0282	0.0197	0.0152	0.0142	0.0148	0.0142	0.0143	0.0140	0.0133	0.0134
16	0.0188	0.0150	0.0148	0.0136	0.0138	0.0140	0.0138	0.0141	0.0137	0.0136
17	0.0165	0.0132	0.0149	0.0141	0.0133	0.0128	0.0130	0.0133	0.0133	0.0131
18	0.0213	0.0198	0.0177	0.0166	0.0161	0.0156	0.0145	0.0142	0.0135	0.0139
19	0.0132	0.0136	0.0143	0.0140	0.0150	0.0145	0.0139	0.0134	0.0131	0.0127
20	0.0205	0.0209	0.0169	0.0154	0.0151	0.0143	0.0137	0.0134	0.0132	0.0132

Calculate the mean volatility for each portfolio size and replicate the plot above

```
volatilities.mul(100 * np.sqrt(252)).mean().plot()
plt.xlabel('Portfolio Size')
plt.ylabel('Mean of Annualized Volatility (%)')
plt.title('Diminishing Returns to Diversification (Elton and Gruber, 1977)')
plt.show()
```



Week 6

McKinney Chapter 10 - Data Aggregation and Group Operations

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import pandas_datareader as pdr
import yfinance as yf
```

```
%precision 4
pd.options.display.float_format = '{:.4f}'.format
# %config InlineBackend.figure_format = 'retina'
```

Introduction

Chapter 10 of McKinney (2022) discusses groupby operations, the pandas equivalent of pivot tables in Excel. Pivot tables calculate statistics (e.g., sums, means, and medians) for one set of variables by groups of other variables (e.g., weekdays and tickers). For example, we could use a pivot table to calculate mean daily stock returns by weekday.

We will focus on:

1. The `.groupby()` method to group by columns and indexes
2. The `.agg()` method to aggregate columns to single values
3. The `.pivot_table()` method as an alternative to `.groupby()`

Note: Indented block quotes are from McKinney (2022) unless otherwise indicated. The section numbers here differ from McKinney (2022) because we will only discuss some topics.

GroupBy Mechanics

"Split-apply-combine" is an excellent way to describe pandas groupby operations.

> Hadley Wickham, an author of many popular packages for the R programming language, coined the term split-apply-combine for describing group operations. In the first stage of the process, data contained in a pandas object, whether a Series, DataFrame, or otherwise, is split into groups based on one or more keys that you provide. The splitting is performed on a particular axis of an object. For example, a DataFrame can be grouped on its rows (axis=0) or its columns (axis=1). Once this is done, a function is applied to each group, producing a new value. Finally, the results of all those function applications are combined into a result object. The form of the resulting object will usually depend on what's being done to the data. See Figure 10-1 for a mockup of a simple group aggregation.

Figure 10-1 visualizes a split-apply-combine operation that:

1. Splits by the `key` column (i.e., "groups by `key`")
2. Applies the sum operation to the `data` column (i.e., "and sums `data`")
3. Combines the grouped sums (i.e., "combines the output")

We could describe this operation as "sum the `data` column by groups of the `key` column then combines the output."

```
np.random.seed(42)
df = pd.DataFrame({'key1' : ['a', 'a', 'b', 'b', 'a'],
                   'key2' : ['one', 'two', 'one', 'two', 'one'],
                   'data1' : np.random.randn(5),
                   'data2' : np.random.randn(5)})

df
```

	key1	key2	data1	data2
0	a	one	0.4967	-0.2341
1	a	two	-0.1383	1.5792
2	b	one	0.6477	0.7674
3	b	two	1.5230	-0.4695
4	a	one	-0.2342	0.5426

Here is the manual way to calculate the means of `data1` by groups of `key1`.

```
df.loc[df['key1'] == 'a', 'data1'].mean()
```

```
0.0414
```

```
df.loc[df['key1'] == 'b', 'data1'].mean()
```

```
1.0854
```

We can do this calculation more easily!

1. Use the `.groupby()` method to group by `key1`
2. Use the `.mean()` method to calculate the mean of `data1` within each value of `key1`

```
df['data1'].groupby(df['key1']).mean()
```

```
key1
a   0.0414
b   1.0854
Name: data1, dtype: float64
```

We can wrap `data1` with two sets of square brackets if we prefer our result as a data frame instead of a series.

```
df[['data1']].groupby(df['key1']).mean()
```

	data1
key1	
a	0.0414
b	1.0854

We can group by more than one variable!

```
means = df['data1'].groupby([df['key1'], df['key2']]).mean()
means
```

```
key1  key2
a     one      0.1313
      two     -0.1383
b     one      0.6477
      two      1.5230
Name: data1, dtype: float64
```

We can use the `.unstack()` method if we want to use both rows and columns to organize data. Recall that the `.unstack()` method un-stacks the inner index level (i.e., `level = -1`) by default so that `key2` values become the columns.

```
means.unstack()
```

key2 key1	one	two
a	0.1313	-0.1383
b	0.6477	1.5230

Our grouping variables are typically columns in the data frame we want to group, so the following syntax is more compact and easier to read.

```
df.groupby(['key1', 'key2'])['data1'].mean().unstack()
```

key2 key1	one	two
a	0.1313	-0.1383
b	0.6477	1.5230

We can wrap long chains in parentheses to insert line breaks and improve readability.

```
(
    df
    .groupby(['key1', 'key2'])
    ['data1']
    .mean()
    .unstack()
)
```

key2 key1	one	two
a	0.1313	-0.1383
b	0.6477	1.5230

However, we must pass only numerical columns to numerical aggregation methods. Otherwise, pandas will give a type error. For example, in the following code, pandas unsuccessfully tries to calculate the mean of string `key2`.

```
# # TypeError: agg function failed [how->mean,dtype->object]
# df.groupby('key1').mean()
```

We avoid this error by slicing the numerical columns.

```
df.groupby('key1')[['data1', 'data2']].mean()
```

	data1	data2
key1		
a	0.0414	0.6292
b	1.0854	0.1490

Grouping with Functions

We can also group with functions. Below, we group with the `len` function, which calculates the lengths of the labels in the row index.

```
np.random.seed(42)
people = pd.DataFrame(
    data=np.random.randn(5, 5),
    columns=['a', 'b', 'c', 'd', 'e'],
    index=['Joe', 'Steve', 'Wes', 'Jim', 'Travis']
)

people
```

	a	b	c	d	e
Joe	0.4967	-0.1383	0.6477	1.5230	-0.2342
Steve	-0.2341	1.5792	0.7674	-0.4695	0.5426
Wes	-0.4634	-0.4657	0.2420	-1.9133	-1.7249
Jim	-0.5623	-1.0128	0.3142	-0.9080	-1.4123
Travis	1.4656	-0.2258	0.0675	-1.4247	-0.5444

```
people.groupby(len).sum()
```

	a	b	c	d	e
3	-0.5290	-1.6168	1.2039	-1.2983	-3.3714
5	-0.2341	1.5792	0.7674	-0.4695	0.5426
6	1.4656	-0.2258	0.0675	-1.4247	-0.5444

We can mix functions, lists, dictionaries, etc., as arguments to the `.groupby()` method.

```
key_list = ['one', 'one', 'one', 'two', 'two']
people.groupby([len, key_list]).min()
```

		a
3	one	-0.4634
	two	-0.5623
5	one	-0.2341
6	two	1.4656

```
d = {'Joe': 'a', 'Jim': 'b'}
people.groupby([len, d]).min()
```

		a	b
3	a	0.4967	-0
	b	-0.5623	-1

```
d_2 = {'Joe': 'Cool', 'Jim': 'Nerd', 'Travis': 'Cool'}
people.groupby([len, d_2]).min()
```

		a
3	Cool	0.4967
	Nerd	-0.5623
6	Cool	1.4656

Grouping by Index Levels

We can also group by index levels.

```
columns = pd.MultiIndex.from_arrays([['US', 'US', 'US', 'JP', 'JP'],
                                    [1, 3, 5, 1, 3]],
                                    names=['cty', 'tenor'])
hier_df = pd.DataFrame(np.random.randn(4, 5), columns=columns).transpose()
hier_df
```

		0
cty	tenor	
US	1	0.1109
	3	-1.1510
	5	0.3757
JP	1	-0.6006
	3	-0.2917

```
hier_df.groupby(level='cty').count()
```

	0	1	2	3
cty				
JP	2	2	2	2
US	3	3	3	3

```
hier_df.groupby(level='tenor').count()
```

	0	1	2	3
tenor				
1	2	2	2	2
3	2	2	2	2
5	1	1	1	1

Data Aggregation

Table 10-1 summarizes the optimized groupby methods:

- `count`: Number of non-NA values in the group
- `sum`: Sum of non-NA values
- `mean`: Mean of non-NA values

- `median`: Arithmetic median of non-NA values
- `std`, `var`: Unbiased (n – 1 denominator) standard deviation and variance
- `min`, `max`: Minimum and maximum of non-NA values
- `prod`: Product of non-NA values
- `first`, `last`: First and last non-NA values

These optimized methods are fast and efficient. Still, pandas lets us use non-optimized methods. First, any series method is available.

```
df
```

	key1	key2	data1	data2
0	a	one	0.4967	-0.2341
1	a	two	-0.1383	1.5792
2	b	one	0.6477	0.7674
3	b	two	1.5230	-0.4695
4	a	one	-0.2342	0.5426

```
df.groupby('key1')['data1'].quantile(0.9)
```

```
key1
a   0.3697
b   1.4355
Name: data1, dtype: float64
```

```
0.6477 + 0.9 * (1.5230 - 0.6477)
```

```
1.4355
```

Second, we can write functions and pass them to the `.agg()` method. These functions should accept an array and return a single value.

```
def max_minus_min(arr):
    return arr.max() - arr.min()
```

```
df.sort_values(by=['key1', 'data1'])
```

	key1	key2	data1	data2
4	a	one	-0.2342	0.5426
1	a	two	-0.1383	1.5792
0	a	one	0.4967	-0.2341
2	b	one	0.6477	0.7674
3	b	two	1.5230	-0.4695

```
df.groupby('key1')['data1'].agg(max_minus_min)
```

```
key1
a   0.7309
b   0.8753
Name: data1, dtype: float64
```

```
1.5230 - 0.6477
```

```
0.8753
```

Some other methods work, too, even if they do not aggregate an array to a scalar.

```
df.groupby('key1')['data1'].describe()
```

	count	mean	std	min	25%	50%	75%	max
key1								
a	3.0000	0.0414	0.3972	-0.2342	-0.1862	-0.1383	0.1792	0.4967
b	2.0000	1.0854	0.6190	0.6477	0.8665	1.0854	1.3042	1.5230

The `.agg()` method provides two more handy features:

1. We can pass multiple functions to operate on all columns
2. We can pass specific functions to operate on specific columns

First, here are examples of multiple functions that operate on all columns.

```
df.groupby('key1')['data1'].agg(['mean', 'median', 'min', 'max'])
```

	mean	median	min	max
key1				
a	0.0414	-0.1383	-0.2342	0.4967
b	1.0854	1.0854	0.6477	1.5230

```
df.groupby('key1')[['data1', 'data2']].agg(['mean', 'median', 'min', 'max'])
```

	data1				data2			
	mean	median	min	max	mean	median	min	max
key1								
a	0.0414	-0.1383	-0.2342	0.4967	0.6292	0.5426	-0.2341	1.5792
b	1.0854	1.0854	0.6477	1.5230	0.1490	0.1490	-0.4695	0.7674

Second, here are examples of specific functions that operate on specific columns.

```
df.groupby('key1').agg({'data1': 'mean', 'data2': 'median'})
```

	data1	data2
key1		
a	0.0414	0.5426
b	1.0854	0.1490

We can calculate the mean *and standard deviation* of `data1` and the median of `data2` by `key1`.

```
df.groupby('key1').agg({'data1': ['mean', 'std'], 'data2': 'median'})
```

	data1		data2
	mean	std	median
key1			
a	0.0414	0.3972	0.5426
b	1.0854	0.6190	0.1490

Apply: General split-apply-combine

The `.agg()` method aggregates an array to a scalar. We can use the `.apply()` method for more general calculations that do not return a scalar. For example, the following `top()` function selects the top `n` rows in data frame `x` sorted by column `col`. The `.sort_values()` method sorts from low to high by default.

```
def top(x, col, n=1):
    return x.sort_values(col).head(n)
```

```
df
```

	key1	key2	data1	data2
0	a	one	0.4967	-0.2341
1	a	two	-0.1383	1.5792
2	b	one	0.6477	0.7674
3	b	two	1.5230	-0.4695
4	a	one	-0.2342	0.5426

```
top(
    x=df.loc[df['key1'] == 'a'],
    col='data1',
    n=2
)
```

	key1	key2	data1	data2
4	a	one	-0.2342	0.5426
1	a	two	-0.1383	1.5792

The following code returns the one row with the smallest value of `data1` within each group of `key1`. *Note:* we include the `include_groups=False` to suppress the `FutureWarning` and adopt the future default behavior now.

The following code returns the *two rows* with the smallest values of `data1` within each group of `key1`.

```
df.groupby('key1').apply(top, col='data1', include_groups=False)
```

		key2	data1	data2
key1				
a	4	one	-0.2342	0.5426
b	2	one	0.6477	0.7674

```
df.groupby('key1').apply(top, col='data1', n=2, include_groups=False)
```

		key2	data
key1			
a	4	one	-0.23
	1	two	-0.13
b	2	one	0.64
	3	two	1.52

We must use the `.reset_index()` method with the `drop=True` argument if we want to drop the index from `df`.

```
(
    df
    .groupby('key1')
    .apply(top, col='data1', n=2, include_groups=False)
    .reset_index(level=1, drop=True)
)
```

	key2	data1	data2
key1			
a	one	-0.2342	0.5426
a	two	-0.1383	1.5792
b	one	0.6477	0.7674
b	two	1.5230	-0.4695

Note

The `.agg()` and `.apply()` methods both operate on groups created by the `.groupby()` method. However, they serve different purposes and have distinct use cases.
The `.agg()` method is designed for aggregating data, meaning it applies functions that reduce a group to a single value (e.g., mean, sum, or custom functions that return a

single scalar). This method is useful for summarizing data across groups.
In contrast, the `.apply()` method is more general and flexible. The `.apply()` method returns results of varying shapes. The `.agg()` method is limited to scalar outputs for each group, but the `.apply()` method is not.

Pivot Tables and Cross-Tabulation

Above, we manually made pivot tables with the `.groupby()`, `.agg()`, `.apply()` and `.unstack()` methods. pandas provides Excel-style aggregations with the `.pivot_table()` method and the `pandas.pivot_table()` function. It is worthwhile to read the `.pivot_table()` docstring several times.

```
ind = (
    yf.download(
        tickers='^GSPC ^DJI ^IXIC ^FTSE ^N225 ^HSI',
        auto_adjust=False,
        progress=False
    )
    .iloc[:-1]
    .stack(future_stack=True)
)

ind.head()
```

	Price	Adj C
Date	Ticker	
1927-12-30	^DJI	NaN
	^FTSE	NaN
	^GSPC	17.66
	^HSI	NaN
	^IXIC	NaN

The default aggregation function for `.pivot_table()` is `.mean()`. For the remaining examples, we will only consider data from 2015 and later.

```
(
    ind
    .loc['2015':]
```

```
    .pivot_table(
        index='Ticker'
    )
)
```

Price Ticker	Adj Close	Close	High	Low	Open	Volume
^DJI	27935.0069	27935.0069	28079.9469	27774.3386	27931.3049	303142509.7886
^FTSE	7163.3634	7163.3634	7203.9626	7121.6113	7162.5127	814461430.5924
^GSPC	3395.6074	3395.6074	3413.1090	3375.7649	3395.1843	4015744083.7901
^HSI	23785.6004	23785.6004	23950.5568	23615.3132	23798.3853	2145570432.1457
^IXIC	9999.4784	9999.4784	10065.3920	9924.6220	9998.5261	3567504600.6265
^N225	25038.6233	25038.6233	25175.5387	24893.2712	25039.7854	98144699.7179

We can specificy a different aggregation function with the `aggfunc` argument. We can use `values` to select specific variables, `pd.Grouper()` to sample different date windows, and `aggfunc` to select specific aggregation functions.

```
(
    ind
    .loc['2015':]
    .reset_index()
    .pivot_table(
        values='Close',
        index=pd.Grouper(key='Date', freq='YE'),
        columns='Ticker',
        aggfunc=['min', 'max']
    )
)
```

 Ticker Date	min ^DJI	 ^FTSE	 ^GSPC	 ^HSI	 ^IXIC	 ^N225	max ^DJI	 ^FTS
2015-12-31	15666.4404	5874.1001	1867.6100	20556.5996	4506.4902	16795.9609	18312.3906	7104.0
2016-12-31	15660.1797	5537.0000	1829.0800	18319.5801	4266.8398	14952.0195	19974.6191	7142.7
2017-12-31	19732.4004	7099.2002	2257.8301	22134.4707	5429.0801	18335.6309	24837.5098	7687.7
2018-12-31	21792.1992	6584.7002	2351.1001	24585.5293	6192.9199	19155.7402	26828.3906	7877.5
2019-12-31	22686.2207	6692.7002	2447.8899	25064.3594	6463.5000	19561.9609	28645.2598	7686.6
2020-12-31	18591.9297	4993.8999	2237.3999	21696.1309	6860.6699	16552.8301	30606.4805	7674.6

	min						max	
Ticker	^DJI	^FTSE	^GSPC	^HSI	^IXIC	^N225	^DJI	^FTS
Date								
2021-12-31	29982.6191	6407.5000	3700.6499	22744.8594	12609.1602	27013.2500	36488.6289	7420.7
2022-12-31	28725.5098	6826.2002	3577.0300	14687.0195	10213.2900	24717.5293	36799.6484	7672.3
2023-12-31	31819.1406	7256.8999	3808.1001	16201.4902	10305.2402	25716.8594	37710.1016	8014.2
2024-12-31	37266.6719	7446.2998	4688.6802	14961.1797	14510.2998	31458.4199	45014.0391	8445.7
2025-12-31	41938.4492	8201.5000	5827.0400	18874.1406	18544.4199	38142.3711	44882.1289	8807.4

McKinney Chapter 10 - Practice - Blank

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import pandas_datareader as pdr
import yfinance as yf
```

```
%precision 4
pd.options.display.float_format = '{:.4f}'.format
# %config InlineBackend.figure_format = 'retina'
```

Announcements

Five-Minute Review

Practice

Replicate the following `.pivot_table()` output with `.groupby()`

```
ind = (
    yf.download(
        tickers='^GSPC ^DJI ^IXIC ^FTSE ^N225 ^HSI',
        auto_adjust=False,
        progress=False
    )
    .rename_axis(columns=['Variable', 'Index'])
    .stack(future_stack=True)
)
```

```
[****************      33%                       ]  2 of 6 completed[*********************5
```

```
a = (
    ind
    .loc['2015':]
    .reset_index()
    .pivot_table(
        values='Close',
        index=pd.Grouper(key='Date', freq='YE'),
        columns='Index',
        aggfunc=['min', 'max']
    )
)
```

Calulate the mean and standard deviation of returns by ticker for the MATANA (MSFT, AAPL, TSLA, AMZN, NVDA, and GOOG) stocks

Consider only dates with complete returns data. Try this calculation with wide and long data frames, and confirm your results are the same.

```
matana = (
    yf.download(
        tickers='MSFT AAPL TSLA AMZN NVDA GOOG',
        auto_adjust=False,
        progress=False
    )
    .rename_axis(columns=['Variable', 'Ticker'])
)
```

```
[****************      33%                       ]  2 of 6 completed[*********************50
```

Calculate the mean and standard deviation of returns and the maximum of closing prices by ticker for the MATANA stocks

Calculate monthly means and volatilities for SPY and GOOG returns

Plot the monthly means and volatilities from the previous exercise

Assign the Dow Jones stocks to five portfolios based on the *preceding* month's volatility

Plot the time-series volatilities of these five portfolios

Calculate the *mean* monthly correlation between the Dow Jones stocks

Is market volatility higher during wars?

Here is some guidance:

1. Download the daily factor data from Ken French's website
2. Calculate daily market returns by summing the market risk premium and risk-free rates (`Mkt-RF` and `RF`, respectively)
3. Calculate the volatility (standard deviation) of daily returns *every month* by combining `pd.Grouper()` and `.groupby()`)
4. Multiply by $\sqrt{252}$ to annualize these volatilities of daily returns
5. Plot these annualized volatilities

Is market volatility higher during wars? Consider the following dates:

1. WWII: December 1941 to September 1945
2. Korean War: 1950 to 1953
3. Viet Nam War: 1959 to 1975
4. Gulf War: 1990 to 1991
5. War in Afghanistan: 2001 to 2021

Week 7

McKinney Chapter 11 - Time Series

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import pandas_datareader as pdr
import yfinance as yf
```

```
%precision 4
pd.options.display.float_format = '{:.4f}'.format
# %config InlineBackend.figure_format = 'retina'
```

Introduction

Chapter 11 of McKinney (2022) discusses time series and panel data, which is where pandas *excels!* We will use these time series and panel tools every day for the rest of the course.

We will focus on:

1. Slicing a data frame or series by date or date range
2. Using `.shift()` to create leads and lags of variables
3. Using `.resample()` to change the frequency of variables
4. Using `.rolling()` to aggregate data over moving or rolling windows

Note: Indented block quotes are from McKinney (2022) unless otherwise indicated. The section numbers here differ from McKinney (2022) because we will only discuss some topics.

Time Series Basics

Let us create a time series to play with.

```
from datetime import datetime
dates = [
    datetime(2011, 1, 2),
    datetime(2011, 1, 5),
    datetime(2011, 1, 7),
    datetime(2011, 1, 8),
    datetime(2011, 1, 10),
    datetime(2011, 1, 12)
]
np.random.seed(42)
ts = pd.Series(np.random.randn(6), index=dates)

ts
```

```
2011-01-02    0.4967
2011-01-05   -0.1383
2011-01-07    0.6477
2011-01-08    1.5230
2011-01-10   -0.2342
2011-01-12   -0.2341
dtype: float64
```

Note that pandas converts the `datetime` objects to a pandas `DatetimeIndex` object and a single index value is a `Timestamp` object.

```
ts.index
```

```
DatetimeIndex(['2011-01-02', '2011-01-05', '2011-01-07', '2011-01-08',
               '2011-01-10', '2011-01-12'],
              dtype='datetime64[ns]', freq=None)
```

```
ts.index[0]
```

```
Timestamp('2011-01-02 00:00:00')
```

Recall that pandas automatically aligns objects on indexes.

```
ts
```

```
2011-01-02    0.4967
2011-01-05   -0.1383
2011-01-07    0.6477
2011-01-08    1.5230
2011-01-10   -0.2342
2011-01-12   -0.2341
dtype: float64
```

```
ts.iloc[::2]
```

```
2011-01-02    0.4967
2011-01-07    0.6477
2011-01-10   -0.2342
dtype: float64
```

```
ts + ts.iloc[::2]
```

```
2011-01-02    0.9934
2011-01-05       NaN
2011-01-07    1.2954
2011-01-08       NaN
2011-01-10   -0.4683
2011-01-12       NaN
dtype: float64
```

If we want to assign a default for missing values on the fly, we can use the `.add()` method.

```
ts.add(ts.iloc[::2], fill_value=1_000_000)
```

```
2011-01-02          0.9934
2011-01-05     999999.8617
2011-01-07          1.2954
2011-01-08    1000001.5230
2011-01-10         -0.4683
2011-01-12     999999.7659
dtype: float64
```

Indexing, Selection, Subsetting

pandas uses U.S.-style date strings (e.g., "M/D/Y") or unambiguous date strings (e.g., "YYYY-MM-DD") to select data.

```
ts.loc['1/10/2011'] # M/D/YYYY
```

```
-0.2342
```

```
ts.loc['2011-01-10'] # YYYY-MM-DD
```

```
-0.2342
```

```
ts.loc['20110110'] # YYYYMMDD
```

```
-0.2342
```

```
ts.loc['10-Jan-2011'] # D-Mon-YYYY
```

```
-0.2342
```

```
ts.loc['Jan-10-2011'] # Mon-D-YYYY
```

```
-0.2342
```

pandas *does not* use U.K.-style date strings.

```
# ts.loc['10/1/2011'] # D/M/YYYY # KeyError: '10/1/2011'
```

Let us create a *longer* time series to play with.

```
np.random.seed(42)
longer_ts = pd.Series(
    data=np.random.randn(1000),
    index=pd.date_range('1/1/2000', periods=1000)
)
```

```
longer_ts
```

```
2000-01-01    0.4967
2000-01-02   -0.1383
2000-01-03    0.6477
2000-01-04    1.5230
2000-01-05   -0.2342
               ...
2002-09-22   -0.2811
2002-09-23    1.7977
2002-09-24    0.6408
2002-09-25   -0.5712
2002-09-26    0.5726
Freq: D, Length: 1000, dtype: float64
```

We can specify a year-month to slice all of the observations in May of 2001.

```
longer_ts.loc['2001-05']
```

```
2001-05-01   -0.6466
2001-05-02   -1.0815
2001-05-03    1.6871
2001-05-04    0.8816
2001-05-05   -0.0080
2001-05-06    1.4799
2001-05-07    0.0774
2001-05-08   -0.8613
2001-05-09    1.5231
2001-05-10    0.5389
2001-05-11   -1.0372
2001-05-12   -0.1903
2001-05-13   -0.8756
2001-05-14   -1.3828
2001-05-15    0.9262
2001-05-16    1.9094
2001-05-17   -1.3986
2001-05-18    0.5630
2001-05-19   -0.6506
2001-05-20   -0.4871
2001-05-21   -0.5924
2001-05-22   -0.8640
```

```
2001-05-23    0.0485
2001-05-24   -0.8310
2001-05-25    0.2705
2001-05-26   -0.0502
2001-05-27   -0.2389
2001-05-28   -0.9076
2001-05-29   -0.5768
2001-05-30    0.7554
2001-05-31    0.5009
Freq: D, dtype: float64
```

We can also specify a year to slice all observations in 2001.

```
longer_ts.loc['2001']
```

```
2001-01-01    0.2241
2001-01-02    0.0126
2001-01-03    0.0977
2001-01-04   -0.7730
2001-01-05    0.0245
               ...
2001-12-27    0.0184
2001-12-28    0.3476
2001-12-29   -0.5398
2001-12-30   -0.7783
2001-12-31    0.1958
Freq: D, Length: 365, dtype: float64
```

If we sort our data chronologically, we can also slice with a range of date strings.

```
ts.loc['1/6/2011':'1/10/2011']
```

```
2011-01-07    0.6477
2011-01-08    1.5230
2011-01-10   -0.2342
dtype: float64
```

However, we cannot date slice if our data are not sorted chronologically.

```
ts2 = ts.sort_values()
```

```
ts2
```

```
2011-01-10   -0.2342
2011-01-12   -0.2341
2011-01-05   -0.1383
2011-01-02    0.4967
2011-01-07    0.6477
2011-01-08    1.5230
dtype: float64
```

The following date slice fails because `ts2` is not sorted chronologically.

```
# ts2.loc['1/6/2011':'1/11/2011'] # KeyError: 'Value based partial slicing on non-monotonic D
```

We can use the `.sort_index()` method first to allow date slices.

```
ts2.sort_index()['1/6/2011':'1/11/2011']
```

```
2011-01-07    0.6477
2011-01-08    1.5230
2011-01-10   -0.2342
dtype: float64
```

As with label slices, date slices are inclusive on both ends.

```
longer_ts.loc['1/6/2001':'1/11/2001']
```

```
2001-01-06    0.4980
2001-01-07    1.4511
2001-01-08    0.9593
2001-01-09    2.1532
2001-01-10   -0.7673
2001-01-11    0.8723
Freq: D, dtype: float64
```

Recall that if we modify a slice, we modify the original series or data frame.

Remember that slicing in this manner produces views on the source time series like slicing NumPy arrays. This means that no data is copied and modifications on the slice will be reflected in the original data.

```
ts3 = ts.copy()
```

```
ts3
```

```
2011-01-02    0.4967
2011-01-05   -0.1383
2011-01-07    0.6477
2011-01-08    1.5230
2011-01-10   -0.2342
2011-01-12   -0.2341
dtype: float64
```

```
ts4 = ts3.iloc[:3]
```

```
ts4
```

```
2011-01-02    0.4967
2011-01-05   -0.1383
2011-01-07    0.6477
dtype: float64
```

```
ts4.iloc[:] = 2001
```

```
ts4
```

```
2011-01-02   2001.0000
2011-01-05   2001.0000
2011-01-07   2001.0000
dtype: float64
```

```
ts3
```

```
2011-01-02   2001.0000
2011-01-05   2001.0000
2011-01-07   2001.0000
```

```
2011-01-08      1.5230
2011-01-10     -0.2342
2011-01-12     -0.2341
dtype: float64
```

Series `ts` is unchanged because `ts3` is a *copy* of `ts`!

```
ts
```

```
2011-01-02    0.4967
2011-01-05   -0.1383
2011-01-07    0.6477
2011-01-08    1.5230
2011-01-10   -0.2342
2011-01-12   -0.2341
dtype: float64
```

Time Series with Duplicate Indices

Most our of data in this course will be well-formed with one observation per date-time for series or one observation per individual per date-time for data frames. However, we may later receive poorly-formed data with duplicate observations. Here, series `dup_ts` has three observations on February 2nd.

```
dates = pd.DatetimeIndex(['1/1/2000', '1/2/2000', '1/2/2000', '1/2/2000', '1/3/2000'])
dup_ts = pd.Series(data=np.arange(5), index=dates)
dup_ts
```

```
2000-01-01    0
2000-01-02    1
2000-01-02    2
2000-01-02    3
2000-01-03    4
dtype: int64
```

The `.is_unique` attribute tells us if an index is unique.

```
dup_ts.index.is_unique
```

```
False
```

```
dup_ts.loc['1/3/2000'] # not duplicated
```

```
np.int64(4)
```

```
dup_ts.loc['1/2/2000'] # duplicated
```

```
2000-01-02    1
2000-01-02    2
2000-01-02    3
dtype: int64
```

The solution to duplicate data depends on the context. For example, we may want the mean of all observations on a given date. The `.groupby()` method can help us here.

```
dup_ts.groupby(level=0).mean()
```

```
2000-01-01   0.0000
2000-01-02   2.0000
2000-01-03   4.0000
dtype: float64
```

Or keep the first value on each date.

```
dup_ts.groupby(level=0).first()
```

```
2000-01-01    0
2000-01-02    1
2000-01-03    4
dtype: int64
```

Date Ranges, Frequencies, and Shifting

> Generic time series in pandas are assumed to be irregular; that is, they have no fixed frequency. For many applications this is sufficient. However, it's often desirable to work relative to a fixed frequency, such as daily, monthly, or every 15 minutes, even if that means introducing missing values into a time series. Fortunately pandas has a full suite of standard time series frequencies and tools for resampling, inferring frequencies, and generating fixed-frequency date ranges.

Shifting Data

Shifting is an important feature! Shifting is moving data backward (or forward) through time.

```
np.random.seed(42)
ts = pd.Series(
    data=np.random.randn(4),
    index=pd.date_range('1/1/2000', periods=4, freq='ME')
)
```

```
ts
```

```
2000-01-31    0.4967
2000-02-29   -0.1383
2000-03-31    0.6477
2000-04-30    1.5230
Freq: ME, dtype: float64
```

If we specify a *positive integer* N to the `.shift()` method:

1. The date index remains the same
2. Values *shift down* N observations

The `.shift()` method defaults to $N = 1$.

```
ts.shift()
```

```
2000-01-31       NaN
2000-02-29    0.4967
2000-03-31   -0.1383
2000-04-30    0.6477
Freq: ME, dtype: float64
```

```
ts.shift(1)
```

```
2000-01-31       NaN
2000-02-29    0.4967
2000-03-31   -0.1383
2000-04-30    0.6477
Freq: ME, dtype: float64
```

```python
ts.shift(2)
```

```
2000-01-31       NaN
2000-02-29       NaN
2000-03-31    0.4967
2000-04-30   -0.1383
Freq: ME, dtype: float64
```

If we specify a *negative integer* N to the `.shift()` method, values *shift up* N observations.

```python
ts.shift(-2)
```

```
2000-01-31   0.6477
2000-02-29   1.5230
2000-03-31      NaN
2000-04-30      NaN
Freq: ME, dtype: float64
```

> **Note**
>
> We almost never shift with negative values to prevent a look-ahead bias. That is, assuming chronological sorting, we almost never bring values from the future back to the present. We do not want to assume that financial market participants know the future.

The `.shift()` examples above shift by N observations without considering time stamps. As a result, the time stamps are unchanged, and values shift down for positive `periods` or up for negative `periods`. However, we can specify the `freq` argument to consider time stamps. With the `freq` argument, time stamps shift by `periods` multiples of the `freq` argument.

```python
ts
```

```
2000-01-31    0.4967
2000-02-29   -0.1383
2000-03-31    0.6477
2000-04-30    1.5230
Freq: ME, dtype: float64
```

```python
ts.shift(periods=2, freq='ME')
```

```
2000-03-31    0.4967
2000-04-30   -0.1383
2000-05-31    0.6477
2000-06-30    1.5230
Freq: ME, dtype: float64
```

```
ts.shift(periods=3, freq='D')
```

```
2000-02-03    0.4967
2000-03-03   -0.1383
2000-04-03    0.6477
2000-05-03    1.5230
dtype: float64
```

`M` is already months, so `min` is minutes.

```
ts.shift(periods=1, freq='90min')
```

```
2000-01-31 01:30:00    0.4967
2000-02-29 01:30:00   -0.1383
2000-03-31 01:30:00    0.6477
2000-04-30 01:30:00    1.5230
dtype: float64
```

Calculating returns

We can calculate returns in two ways. First, easily with the `.pct_change()` method.

```
ts.pct_change()
```

```
2000-01-31       NaN
2000-02-29   -1.2784
2000-03-31   -5.6844
2000-04-30    1.3515
Freq: ME, dtype: float64
```

Second, manaully with the `.shift()` method.

```
(ts - ts.shift()) / ts.shift()
```

```
2000-01-31       NaN
2000-02-29   -1.2784
2000-03-31   -5.6844
2000-04-30    1.3515
Freq: ME, dtype: float64
```

These two return calculations are the same.

```
np.allclose(
    a=ts.pct_change(),
    b=(ts - ts.shift()) / ts.shift(),
    equal_nan=True
)
```

```
True
```

Two observations on these return calculations:

1. The first percent change is `NaN` because there is no previous value to change from
2. The default for `.shift()` and `.pct_change()` is `periods=1`

Shifting dates with offsets

We can also shift time stamps to the beginning or end of a period.

```
from pandas.tseries.offsets import MonthEnd
now = datetime(2011, 11, 17)
```

```
now
```

```
datetime.datetime(2011, 11, 17, 0, 0)
```

`MonthEnd(0)` moves to the end of the month *but does not leave the current month.*

```
now + MonthEnd(0)
```

```
Timestamp('2011-11-30 00:00:00')
```

`MonthEnd(1)` moves to the end of the *current* month. If already at the end of the *current* month, it moves to the end of the *next* month.

```
now + MonthEnd(1)
```

```
Timestamp('2011-11-30 00:00:00')
```

```
now + MonthEnd(1) + MonthEnd(1)
```

```
Timestamp('2011-12-31 00:00:00')
```

```
now + MonthEnd(0) + MonthEnd(1)
```

```
Timestamp('2011-12-31 00:00:00')
```

Be careful! The `MonthEnd()` default is `n=1`!

```
datetime(2021, 10, 31) + MonthEnd(0)
```

```
Timestamp('2021-10-31 00:00:00')
```

```
datetime(2021, 10, 31) + MonthEnd(1)
```

```
Timestamp('2021-11-30 00:00:00')
```

```
datetime(2021, 10, 31) + MonthEnd()
```

```
Timestamp('2021-11-30 00:00:00')
```

Always check your output!

Resampling and Frequency Conversion

Resampling is an important feature!

> Resampling refers to the process of converting a time series from one frequency to another. Aggregating higher frequency data to lower frequency is called downsampling, while converting lower frequency to higher frequency is called upsampling. Not all resampling falls into either of these categories; for example, converting W-WED (weekly on Wednesday) to W-FRI is neither upsampling nor downsampling.

We can resample both series and data frames. The `.resample()` method syntax is similar to `.groupby()`.

Downsampling

> Aggregating data to a regular, lower frequency is a pretty normal time series task. The data you're aggregating doesn't need to be fixed frequently; the desired frequency defines bin edges that are used to slice the time series into pieces to aggregate. For example, to convert to monthly, 'M' or 'BM', you need to chop up the data into one-month intervals. Each interval is said to be half-open; a data point can only belong to one interval, and the union of the intervals must make up the whole time frame. There are a couple things to think about when using resample to downsample data:
>
> - Which side of each interval is closed
> - How to label each aggregated bin, either with the start of the interval or the end

```
rng = pd.date_range(start='2000-01-01', periods=12, freq='min')
ts = pd.Series(np.arange(12), index=rng)
```

```
ts
```

```
2000-01-01 00:00:00    0
2000-01-01 00:01:00    1
2000-01-01 00:02:00    2
2000-01-01 00:03:00    3
2000-01-01 00:04:00    4
2000-01-01 00:05:00    5
2000-01-01 00:06:00    6
2000-01-01 00:07:00    7
2000-01-01 00:08:00    8
```

```
2000-01-01 00:09:00     9
2000-01-01 00:10:00    10
2000-01-01 00:11:00    11
Freq: min, dtype: int64
```

We can aggregate the one-minute frequency data above to five-minute frequency data. Resampling requires an aggregation method. Here, we use the `.sum()` method.

```python
ts.resample('5min').sum()
```

```
2000-01-01 00:00:00    10
2000-01-01 00:05:00    35
2000-01-01 00:10:00    21
Freq: 5min, dtype: int64
```

When we resample with a minute-frequency:

1. Left edges of the resampling interval are closed (included) and right edges are open (excluded)
2. Labels are by the left edge of the resampling interval by default

In the example above, the first value of 10 at midnight is the sum of values at midnight until 00:05, excluding the value at 00:05. That is, the sums are $10 = 0 + 1 + 2 + 3 + 4$ at 00:00, $35 = 5 + 6 + 7 + 8 + 9$ at 00:05, and so on. We can use the `closed` and `label` arguments to change this behavior.

In finance, we generally prefer `closed='right'` and `label='right'` to avoid a lookahead bias.

```python
ts.resample('5min', closed='right', label='right').sum()
```

```
2000-01-01 00:00:00     0
2000-01-01 00:05:00    15
2000-01-01 00:10:00    40
2000-01-01 00:15:00    11
Freq: 5min, dtype: int64
```

These defaults for minute-frequency data may seem odd, but any choice is arbitrary. The defaults for weekly and lower frequencies are `closed='right'` and `label='right'`, which are corect for finance. Still, we should read the docstring and check our output whenever we use the `.resample()` method!

Upsampling and Interpolation

To downsample (i.e., resample from higher to lower frequency), we must aggregate (e.g., `.mean()`, `.sum()`, `.first()`, or `.last()`). To upsample (i.e., resample from lower to higher frequency), we must choose how to fill in the new, higher-frequency observations.

```
np.random.seed(42)
frame = pd.DataFrame(
    data=np.random.randn(2, 4),
    index=pd.date_range('1/1/2000', periods=2, freq='W-WED'),
    columns=['Colorado', 'Texas', 'New York', 'Ohio']
)
```

```
frame
```

	Colorado	Texas	New York	Ohio
2000-01-05	0.4967	-0.1383	0.6477	1.5230
2000-01-12	-0.2342	-0.2341	1.5792	0.7674

We use the `.asfreq()` method to convert to the new frequency and leave the new observations as missing.

```
df_daily = frame.resample('D').asfreq()
```

```
df_daily
```

	Colorado	Texas	New York	Ohio
2000-01-05	0.4967	-0.1383	0.6477	1.5230
2000-01-06	NaN	NaN	NaN	NaN
2000-01-07	NaN	NaN	NaN	NaN
2000-01-08	NaN	NaN	NaN	NaN
2000-01-09	NaN	NaN	NaN	NaN
2000-01-10	NaN	NaN	NaN	NaN
2000-01-11	NaN	NaN	NaN	NaN
2000-01-12	-0.2342	-0.2341	1.5792	0.7674

We use the `.ffill()` method to forward fill values to replace missing values.

```
frame.resample('D').ffill()
```

	Colorado	Texas	New York	Ohio
2000-01-05	0.4967	-0.1383	0.6477	1.5230
2000-01-06	0.4967	-0.1383	0.6477	1.5230
2000-01-07	0.4967	-0.1383	0.6477	1.5230
2000-01-08	0.4967	-0.1383	0.6477	1.5230
2000-01-09	0.4967	-0.1383	0.6477	1.5230
2000-01-10	0.4967	-0.1383	0.6477	1.5230
2000-01-11	0.4967	-0.1383	0.6477	1.5230
2000-01-12	-0.2342	-0.2341	1.5792	0.7674

```
frame.resample('D').ffill(limit=2)
```

	Colorado	Texas	New York	Ohio
2000-01-05	0.4967	-0.1383	0.6477	1.5230
2000-01-06	0.4967	-0.1383	0.6477	1.5230
2000-01-07	0.4967	-0.1383	0.6477	1.5230
2000-01-08	NaN	NaN	NaN	NaN
2000-01-09	NaN	NaN	NaN	NaN
2000-01-10	NaN	NaN	NaN	NaN
2000-01-11	NaN	NaN	NaN	NaN
2000-01-12	-0.2342	-0.2341	1.5792	0.7674

```
frame.resample('W-THU').ffill()
```

	Colorado	Texas	New York	Ohio
2000-01-06	0.4967	-0.1383	0.6477	1.5230
2000-01-13	-0.2342	-0.2341	1.5792	0.7674

Moving Window Functions

Moving or rolling window functions are one of the neatest features of pandas.!

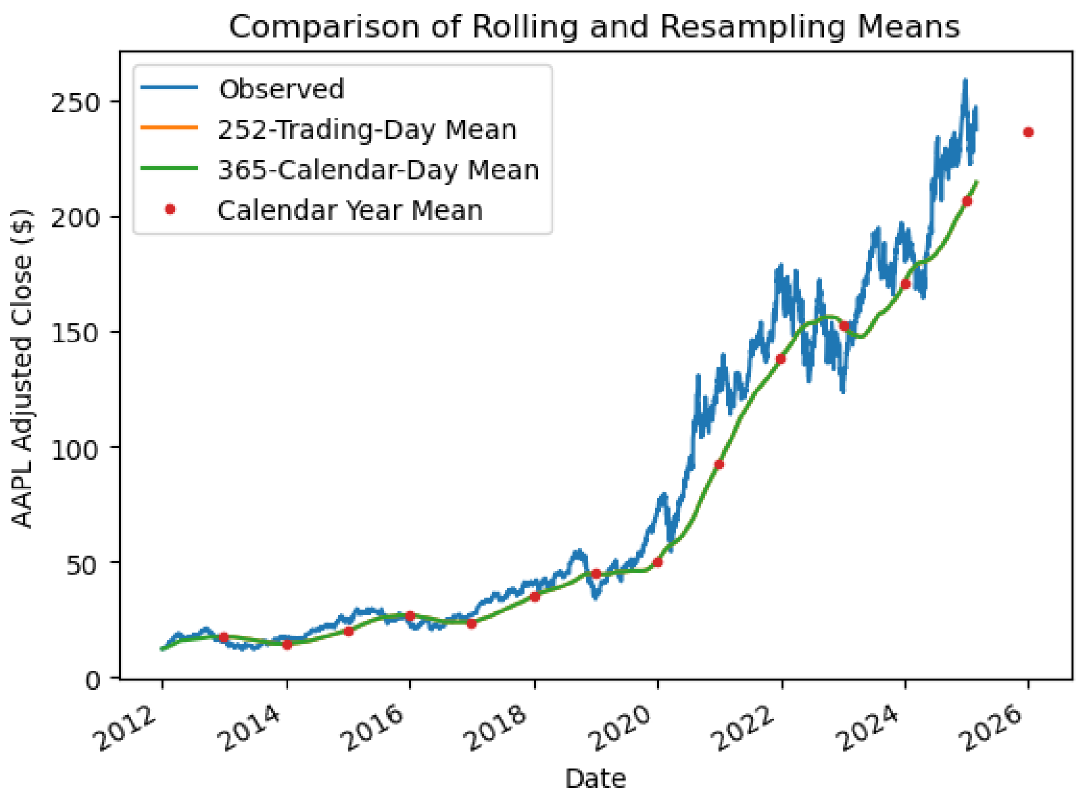
```
df = (
    yf.download(
        tickers=['AAPL', 'MSFT', 'SPY'],
        auto_adjust=False,
        progress=False
    )
    .iloc[:-1]
)
```

```
df
```

Price	Adj Close			Close			High		
Ticker	AAPL	MSFT	SPY	AAPL	MSFT	SPY	AAPL	MSFT	SPY
Date									
1980-12-12	0.0987	NaN	NaN	0.1283	NaN	NaN	0.1289	NaN	NaN
1980-12-15	0.0936	NaN	NaN	0.1217	NaN	NaN	0.1222	NaN	NaN
1980-12-16	0.0867	NaN	NaN	0.1127	NaN	NaN	0.1133	NaN	NaN
1980-12-17	0.0889	NaN	NaN	0.1155	NaN	NaN	0.1161	NaN	NaN
1980-12-18	0.0914	NaN	NaN	0.1189	NaN	NaN	0.1194	NaN	NaN
...	...	...	...	...	...	...	...	...	...
2025-02-21	245.5500	408.2100	599.9400	245.5500	408.2100	599.9400	248.6900	418.0500	610.3000
2025-02-24	247.1000	404.0000	597.2100	247.1000	404.0000	597.2100	248.8600	409.3700	603.0300
2025-02-25	247.0400	397.9000	594.2400	247.0400	397.9000	594.2400	250.0000	401.9200	597.8900
2025-02-26	240.3600	399.7300	594.5400	240.3600	399.7300	594.5400	244.9800	403.6000	599.5800
2025-02-27	237.3000	392.5300	585.0500	237.3000	392.5300	585.0500	242.4600	405.7400	598.0200

The `.rolling()` method accepts a window-width and requires an aggregation method. The following example plots AAPL's observed daily price alongside its 252-trading-day rolling mean, 365-calendar-day rolling mean, and calendar year mean.

```
aapl = df.loc['2012':, ('Adj Close', 'AAPL')]
aapl.plot(label='Observed')
aapl.rolling(window=252).mean().plot(label='252-Trading-Day Mean') # min_periods defaults to
aapl.rolling(window='365D').mean().plot(label='365-Calendar-Day Mean') # min_periods default
aapl.resample('YE').mean().plot(style='.', label='Calendar Year Mean')
plt.legend()
plt.ylabel('AAPL Adjusted Close ($)')
plt.title('Comparison of Rolling and Resampling Means')
plt.show()
```



> **Note**
>
> If we specify the rolling window width as an integer:
>
> 1. Each rolling window is that many observations wide and ignores time stamps
> 2. Each rolling window must have that many non-missing observations
>
> We can specify `min_periods` to allow incomplete windows. For integer window widths, `min_periods` defaults to the given integer window width. For string date offsets, `min_periods` defaults to `1`.

Binary Moving Window Functions

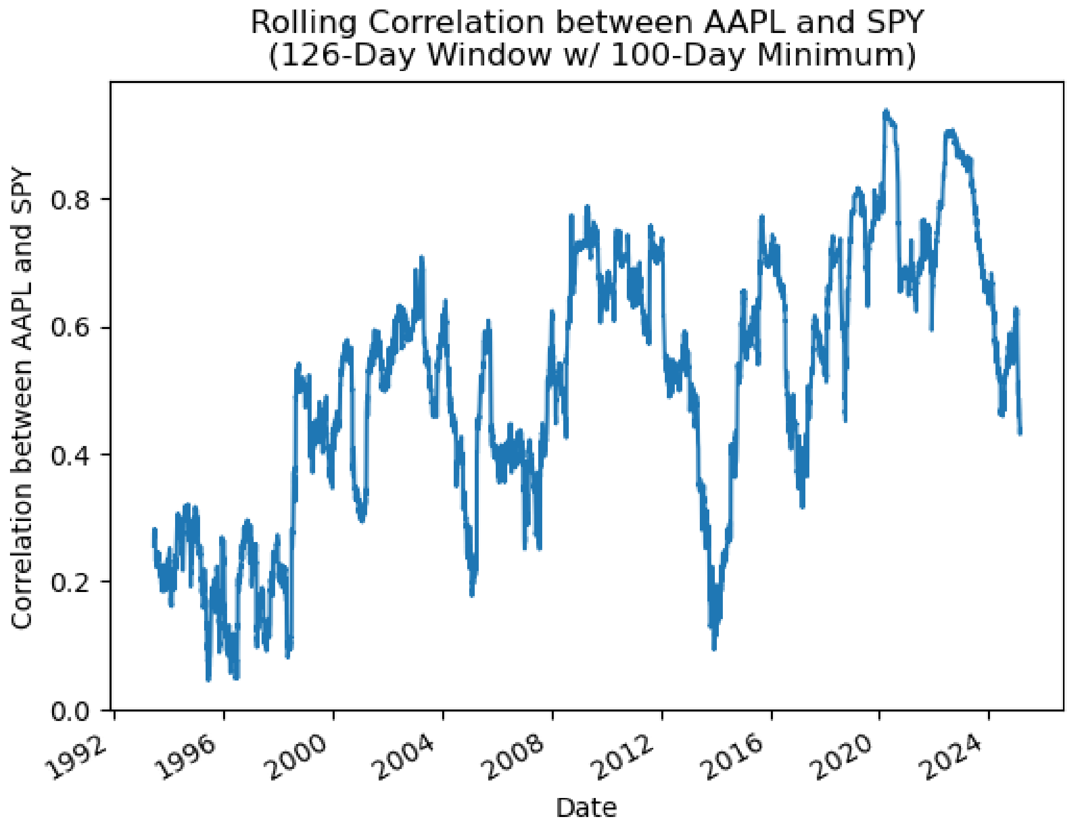
Binary moving window functions accept two inputs. The most common example is the rolling correlation between two return series.

```
returns = df['Adj Close'].pct_change()
```

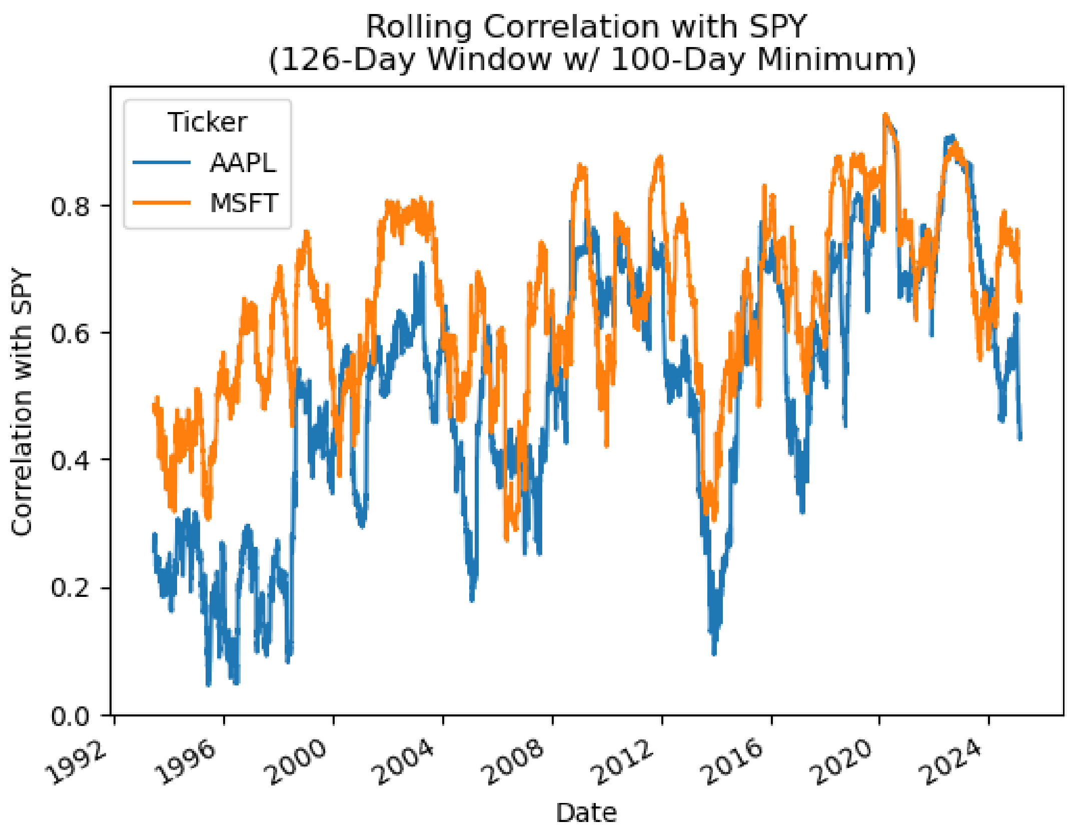
```
returns
```

Ticker Date	AAPL	MSFT	SPY
1980-12-12	NaN	NaN	NaN
1980-12-15	-0.0522	NaN	NaN
1980-12-16	-0.0734	NaN	NaN
1980-12-17	0.0248	NaN	NaN
1980-12-18	0.0290	NaN	NaN
...	...	...	...
2025-02-21	-0.0011	-0.0190	-0.0171
2025-02-24	0.0063	-0.0103	-0.0046
2025-02-25	-0.0002	-0.0151	-0.0050
2025-02-26	-0.0270	0.0046	0.0005
2025-02-27	-0.0127	-0.0180	-0.0160

```
(
    returns['AAPL']
    .rolling(126, min_periods=100)
    .corr(returns['SPY'])
    .plot()
)
plt.ylabel('Correlation between AAPL and SPY')
plt.title('Rolling Correlation between AAPL and SPY\n (126-Day Window w/ 100-Day Minimum)')
plt.show()
```



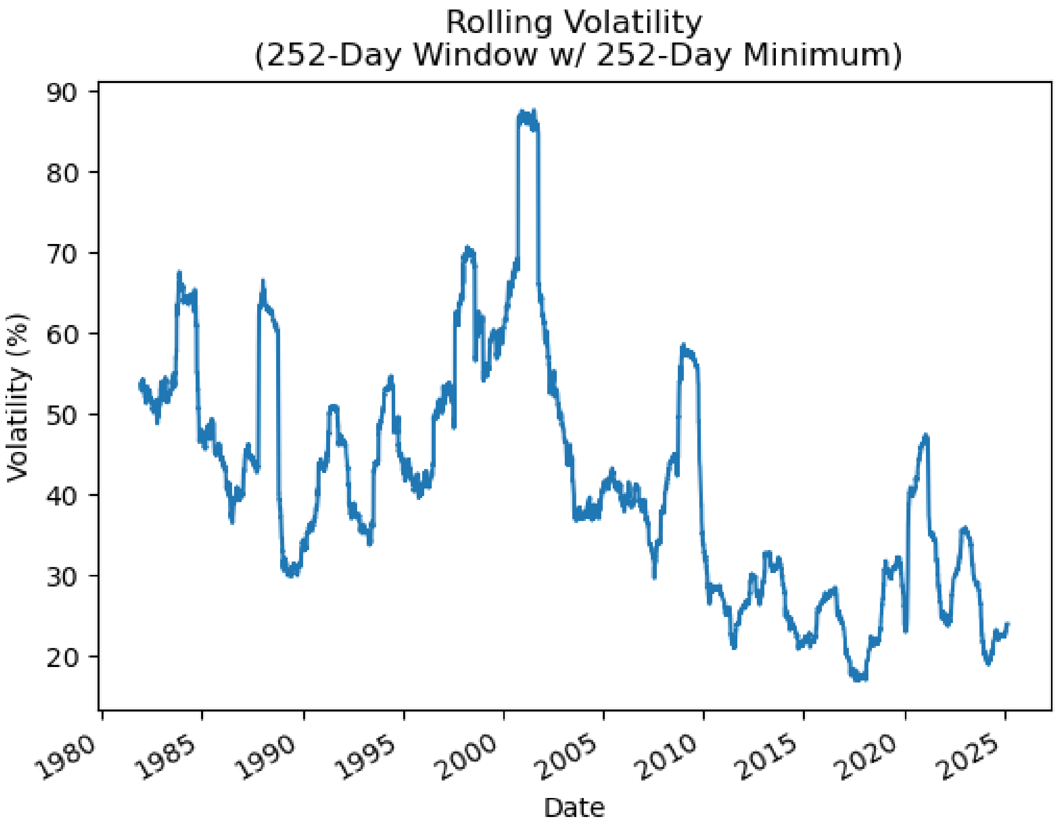

```
(
    returns[['AAPL', 'MSFT']]
    .rolling(126, min_periods=100)
    .corr(returns['SPY'])
    .plot()
)
plt.ylabel('Correlation with SPY')
plt.title('Rolling Correlation with SPY\n (126-Day Window w/ 100-Day Minimum)')
plt.show()
```



User-Defined Moving Window Functions

We can define our own moving window functions and use them with the `.apply()` method. However, note that `.apply()` method will be much slower than the optimized methods, like `.mean()` and `.std()`. Here, we will calculate rolling volatility with `.apply()` and `.std()` and compare their speeds.

```
(
    returns['AAPL']
    .rolling(252)
    .apply(np.std)
    .mul(np.sqrt(252) * 100)
    .plot()
)
plt.ylabel('Volatility (%)')
plt.title('Rolling Volatility\n (252-Day Window w/ 252-Day Minimum)')
plt.show()
```



Do not be afraid to use `.apply()`, but realize that `.apply()` is often 1000 times slower than the optimized method!

```
%timeit returns['AAPL'].rolling(252).apply(np.std)
```

```
2.28 s ± 109 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
%timeit returns['AAPL'].rolling(252).std()
```

```
613  s ± 80  s per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

McKinney Chapter 11 - Practice - Blank

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import pandas_datareader as pdr
import yfinance as yf
```

```
%precision 4
pd.options.display.float_format = '{:.4f}'.format
# %config InlineBackend.figure_format = 'retina'
```

Announcements

Five-Minute Review

Practice

Download daily returns for ten portfolios formed on book-to-market ratios

Plot cumulative returns for all available data

Calculate total returns for each calendar year

Calculate total returns for all 252-trading-day windows

Calculate total returns for 12-months windows with monthly data

Calculate Sharpe Ratios for each calendar year

Calculate rolling betas

We can calculate CAPM betas as: $\beta_i = \frac{Cov(r_i - r_f, r_M - r_f)}{Var(r_M - r_f)}$

Calculate rolling Sharpe Ratios

Week 8

Project 1

Purpose

I have two goals for this project:

1. Help you learn to compare two different foreign exchange (forex) trading strategies in Python
2. Help you work on a real-world task that an industry partner considers interesting and worthwhile

Assignment

Our industry partner trades forex for a multinational client. This client wants our partner to compare its current active trading approach to a new zero-basis banking approach with passive or automatic trading.

Our assignment: Analyze these two approaches, summarize their advantages and disadvantages, and recommend a plan of action for our partner. Your plan should include an executive summary and supporting figures and tables. Here are ideas to consider:

1. Our partner provided her thought process in `Student Python Research.docx` and plans to visit one section virtually
2. Consider only the USD/CAD pair
3. ~~Consider all transaction costs~~
    1. ~~Use a variable `spread` to account for transaction costs that increase linearly with trade size (e.g., market orders pay one-half of the bid-ask spread or `0.5 * spread * size`)~~
    2. ~~Use a variable `gamma` to account for transaction costs that increase with trade size *squared* (e.g., large orders have market impacts that increase quickly with size or `gamma * size * size`)~~
    3. ~~Set `spread` and `gamma` to 0.0001 to approximate the real world~~
4. At a minimum, compare the final dollar values with both approaches
5. You might also consider the best final dollar value that an active trader could achieve, *plus any other interesting aspects of these data*

6. Consider all scenarios that our industry partner provides
7. Title, label, and caption your figures and tables and reference them in your report

Criteria

Table 1 provides the project grading rubric. The project is worth 200 points. The peer reviews are worth 100 points, and students will receive their median score. Almost all students earn perfect peer review scores, so I will factor that into project scores. For example, a project score *without peer review scores* of 77.5% converts to a project score *with perfect peer review scores* of 85% because $\frac{0.775 \times 200 + 1.00 \times 100}{300} = 0.85 = 85\%$.

Table 1: This table provides the project grading rubric

Topic	Points
Clarity, correctness, and completeness of calculations	60
Clarity, correctness, and completeness of visualizations	60
Clarity, correctness, and completeness of discussions	60
Correctness of submission according to the deliverables section	20
Total	200

Deliverables

Upload the following as unzipped files to Canvas by 11:59 PM on 2/28:

1. One Jupyter notebook that contains your report and performs *all* your analysis
    1. Name this file `project_1.ipynb` for me to run your code
    2. Your notebook must run on my computer; I will place the data file in the same folder as your notebook
    3. You may not edit the Word documents after you create them
2. One Quarto-generated Word document *without code* for our partner to review
    1. Name this file `project_1_without_code.docx`
    2. Typing `echo: false` in the first cell of this Jupyter notebook hides code in your Word document
3. One Quarto-generated Word document *with code* for me to grade
    1. Name this file `project_1_with_code.docx`
    2. Typing `echo: true` in the first cell of this Jupyter notebook displays code in your Word document

Here is some additional guidance:

1. Provide a one-page executive summary on your first page; this summary can run up to halfway down the second page to avoid having to render your notebook several times to check lengths
2. Your Word document must not exceed 15 pages in length
3. Do not include your names anywhere in your submission

Data

This project requires two data files.

1. `USDCAD 021345.xlsx` provides recent prices of US dollars in terms of Canadian dollars. For example, 1.3875 indicates that 1.3875 Canadian dollars buy 1 US dollar
2. `NE University Trades.xlsx` provides a summary of our industry partner's trades

We need to install the `openpyxl` package to read these files. To install it, follow these steps:

1. `conda activate fina6333` (activate your course environment)
2. `conda install openpyxl` (install `openpyxl` into your course environment
3. `jupyter lab` (open JupyterLab to begin your analysis)

We can use `pd.read_excel()` to read these files. For example, the following code reads the exchange rate file:

```
import matplotlib.pyplot as plt
import matplotlib.ticker as ticker
import numpy as np
import pandas as pd
import yfinance as yf
```

Edit 2025-02-24: I tweaked this code cell to set `Date` as the index and added comments.

```
df_1 = (
    pd.read_excel(  # Load data from an Excel file into a pandas DataFrame
        io='USDCAD 021325.xlsx',  # Specify the file path or name of the Excel file to read
        sheet_name='Sheet1',  # Select the specific sheet in the Excel file (Sheet1) to load
        names=['Date', 'USDCAD'],  # Assign custom column names 'Date' and 'USDCAD' to the da
        parse_dates=['Date'],  # Tell pandas to automatically parse the 'Date' column as date
    )
    .assign(  # Add or modify columns in the DataFrame
        Date=lambda x: x['Date'].dt.tz_localize('UTC')  # Take the 'Date' column, already a d
```

```
        # 'x' is the DataFrame; .dt accesses datetime properties; .tz_localize('UTC') makes i
    )
    .set_index('Date')  # Set the 'Date' column (now UTC timezone-aware) as the DataFrame's i
)
```

```
df_1.head()
```

	USDCAD
Date	
2024-08-01 17:05:00+00:00	1.3877
2024-08-01 17:10:00+00:00	1.3877
2024-08-01 17:15:00+00:00	1.3875
2024-08-01 17:20:00+00:00	1.3874
2024-08-01 17:25:00+00:00	1.3875

Edit 2025-02-24: I added this code to parse the trades spreadsheet.

```
df_2 = (
    pd.read_excel(  # Load data from an Excel file into a pandas DataFrame
        io='NE University Trades.xlsx',  # Specify the file path or name of the Excel file
        sheet_name='Sheet1',  # Select the specific sheet in the Excel file to read
        usecols='A:I',  # Read only columns A through I (first 9 columns) to focus on relevan
        skiprows=2,  # Skip the first 2 rows (e.g., headers or metadata) to get to the actual
    )
    .assign(  # Add or modify columns in the DataFrame
        Date=lambda x: (  # Create or overwrite the 'Date' column using a lambda function; 'x
            pd.to_datetime(  # Convert a string to a pandas datetime object
                x['Date'].astype(str) + ' ' +  # Convert the 'Date' column to strings and add
                x['Time CST'].apply(lambda t: t.strftime('%H:%M:%S'))  # Convert 'Time CST'
            )  # The result is a combined string like '2025-02-04 15:28:09'
            .dt.tz_localize('America/Chicago')  # Assign the 'America/Chicago' timezone (CST/
            .dt.tz_convert('UTC')  # Convert the timezone-aware datetime from CST/CDT to UTC
        )  # The 'Date' column now holds UTC timestamps
    )
    .set_index('Date')  # Set the 'Date' column (now in UTC) as the DataFrame's index for tim
    .drop(columns='Time CST')  # Remove the original 'Time CST' column since it’s now part of
)
```

```
df_2
```

Date	Interbank Rate	Value Date	Client Rate	Sell CAD	Buy USD	Spread BP
2025-02-04 21:28:09+00:00	1.4329	2025-02-05	1.4374	2M	1391420.50	31
2025-02-12 15:42:37+00:00	1.4295	2025-02-12	1.4341	2M	1394641.79	32
2025-01-29 14:29:55+00:00	1.4453	2025-01-29	1.4496	1M	689840.72	29
2025-01-28 16:18:07+00:00	1.4394	2025-01-29	1.4438	1M	692616.71	30

Quarto

Basics

1. Use Quarto to generate your Word document from your notebook
2. Use `#` to create a title and `##` to create sections
3. Use `-` or `1.` to create lists
4. Use the first cell in this notebook to hide or display code with `echo=false` or `echo=true`, respectively
5. This first cell must be a `raw` cell instead of a `code` or `markdown` cell
6. Use `quarto render project_1.ipynb` in the same folder as your notebook to render it to a Word document
7. Use the `cd` command in the terminal to change the working directory to the directory with your notebook

Examples

This section provides a sample analysis highlighting how code and formatting work with Quarto. Figure 1 provides a line plot of the value of a $10,000 investment in SPY, the S&P 500 SPDR ETF. Note that `#| label:` and `#| fig-cap:` comments at the top of the figure cell create the figure reference/link and the figure caption, respectively. You can learn more about cross-referencing figures and tables here.

```
data = (
    yf.download(tickers='SPY', auto_adjust=False, progress=False)
    .assign(
        ret=lambda x: x['Adj Close'].pct_change(),
        value=lambda x: x['ret'].pipe(np.log1p).cumsum().pipe(np.exp).mul(10_000)
    )
)
```

```
data['value'].plot()
plt.ylabel('Value ($)')
plt.suptitle('Value of $10,000 Investment in SPY')
plt.title(f'Investment made at the market close on {data.index[0]: %b %d, %Y}')
plt.gca().yaxis.set_major_formatter(ticker.FuncFormatter(lambda x, p: format(int(x), ',')))

plt.show()
```

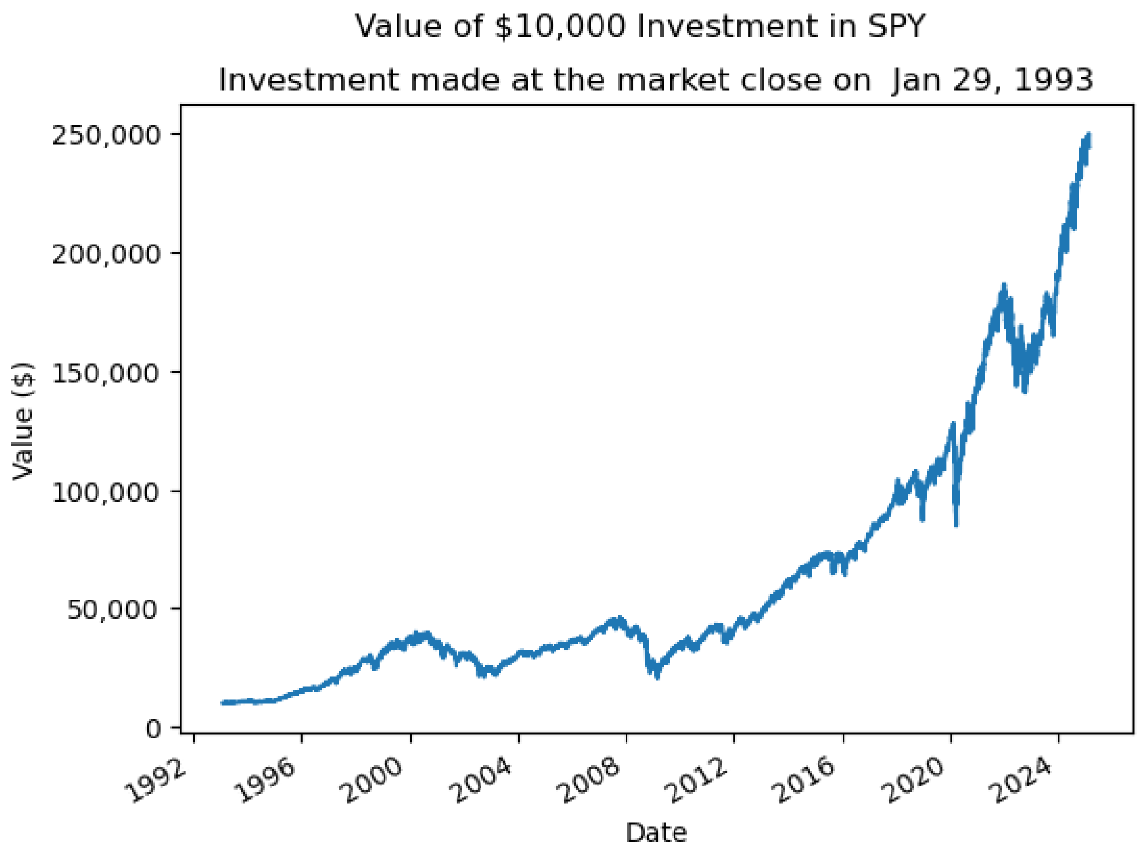


Figure 1: This line plot shows the value of a $10,000 investment in the S&P 500 SPDR ETF at the close of its first day of trading

Artificial Intelligence (AI)

You may use AI (e.g., ChatGPT) to *help* you prepare your analysis and discussion. However:

1. AI will not do very well on this project without significant input from your team

2. AI will not be a defense against plagiarism because AI should not *write* your code and slides; If you plagiarize an AI that plagiarizes other sources, you are responsible for plagiarizing the AI and its sources

Week 9

Herron Topic 2 - Trading Strategies Based on Technical Analysis

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import pandas_datareader as pdr
import statsmodels.api as sm
import yfinance as yf
```

```
%precision 4
pd.options.display.float_format = '{:.4f}'.format
# %config InlineBackend.figure_format = 'retina'
```

Introduction

This notebook covers trading strategies based on technical analysis in three parts:

1. What is technical analysis?
2. Why might trading strategies based on technical analysis work (or not work)?
3. Implement a simple moving average (SMA) trading strategy

I based this lecture notebook on Welch (2022, Chapter 12), Lewinson (2020, Chapter 2), and Murphy (1999). If you want to learn technical analysis, Murphy (1999) is the best reference and covers more than we can in a week or semester. The practice notebook will cover several other trading strategies based on technical analysis.

What is technical analysis?

Technical analysis is a methodology that analyzes past market data (e.g., prices and volume, plus open interest in futures and options markets) in an attempt to forecast future price movements. If technical analysis can predict future price movements, the market is not weak-form efficient. Welch (2022, Section 12.2) provides the three degrees of market efficiency:

> The Traditional Classification The traditional definition of market efficiency focuses on information. In the traditional classification, market efficiency comes in one of three primary degrees: weak, semi-strong, and strong.
>
> **Weak market efficiency** says that all information in past prices is reflected in today's stock prices so that technical analysis (trading based solely on historical price patterns) cannot be used to beat the market. Put differently, the market is the best technical analyst.
>
> **Semistrong market efficiency** says that all public information is reflected in today's stock prices, so that neither fundamental trading (based on underlying firm fundamentals, such as cash flows or discount rates) nor technical analysis can be used to beat the market. Put differently, the market is both the best technical and the best fundamental analyst.
>
> **Strong market efficiency** says that all information, both public and private, is reflected in today's stock prices, so that nothing — not even private insider information — can be used to beat the market. Put differently, the market is the best analyst and cannot be beat.
>
> In this traditional classification, all finance professors nowadays believe that most U.S. financial markets are not strong-form efficient: Insider trading may be illegal, but it works. However, there are still arguments as to which markets are only semi-strong-form efficient or even only weak-form efficient.

Welch (2022, Section 12.2) goes on to provide his own taxonomy of true, firm, mild, and nonbelievers in market efficiency. Chapter 12 summarizes market efficiency, classical finance, behavioral finance, arbitrage, limits to arbitrage, and their consequences for managers and investors. You can read Chapter 12 here. We will focus on technical analysis in this notebook, but Welch (2022) is excellent.

Why might trading strategies based on technical analysis work or not?

...Work?

Technical analysis relies on a few ideas:

1. Market prices and volume reflect all relevant information, so we can focus on past prices and volume instead of fundamentals and news.
2. Market prices move in trends and patterns driven by market participants.
3. These trends and patterns tend to repeat themselves because market participants create them.

…Or Not?

The logic above is reasonable. However, if past market prices reflect all relevant information, they should also reflect any prices trends they predict. Therefore, any patterns should be self-defeating, and market prices should follow a random walk. As well, the signal-to-noise ratio in market prices is high! Still, technical analysis provides an opportunity to learn how to implement and back-test trading strategies in Python.

A Random Walk

In a random walk, the price tomorrow equals the price today plus a tiny drift plus noise. In math terms, a random walk is

$$P_t = \rho P_{t-1} + m P_{t-1} + \varepsilon_t$$

where m is a small drift term and $\mathrm{E}[\varepsilon] = 0$. If $\rho > 1$, prices would quickly increase, and, if $\rho < 1$, prices would quickly decrease. Let us examine the historical record.

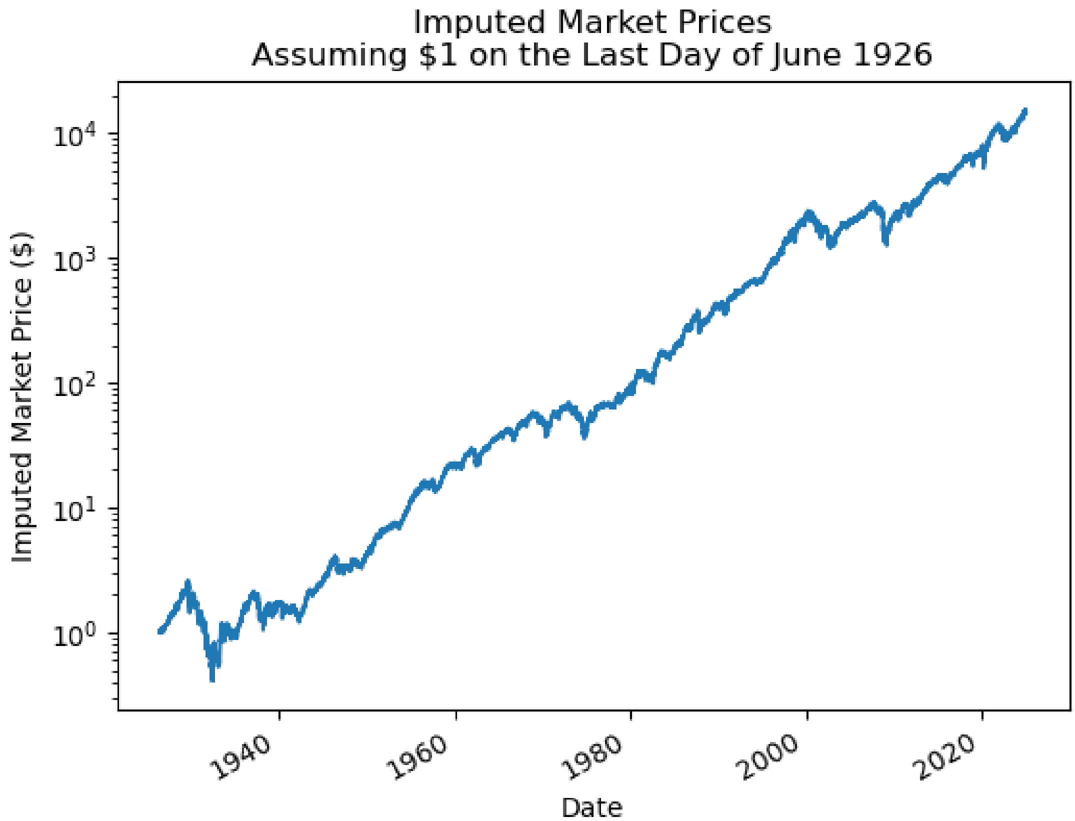
```
ff = (
    pdr.DataReader(
        name='F-F_Research_Data_Factors_daily',
        data_source='famafrench',
        start='1900'
    )
    [0]
    .assign(Mkt=lambda x: x['Mkt-RF'] + x['RF'])
    .div(100)
)
```

```
C:\Users\richa\AppData\Local\Temp\ipykernel_6816\3102596058.py:2: FutureWarning: The argumen
  pdr.DataReader(
```

We can compound market returns to impute market prices relative to the last day of June 1926.

```
prices = ff['Mkt'].add(1).cumprod()
```

```
prices.plot()
plt.title('Imputed Market Prices\nAssuming $1 on the Last Day of June 1926')
plt.ylabel('Imputed Market Price ($)')
plt.semilogy()
plt.show()
```



We need lagged prices to estimate ρ. We will add 10 lags of P to help us understand the relation between past and future prices.

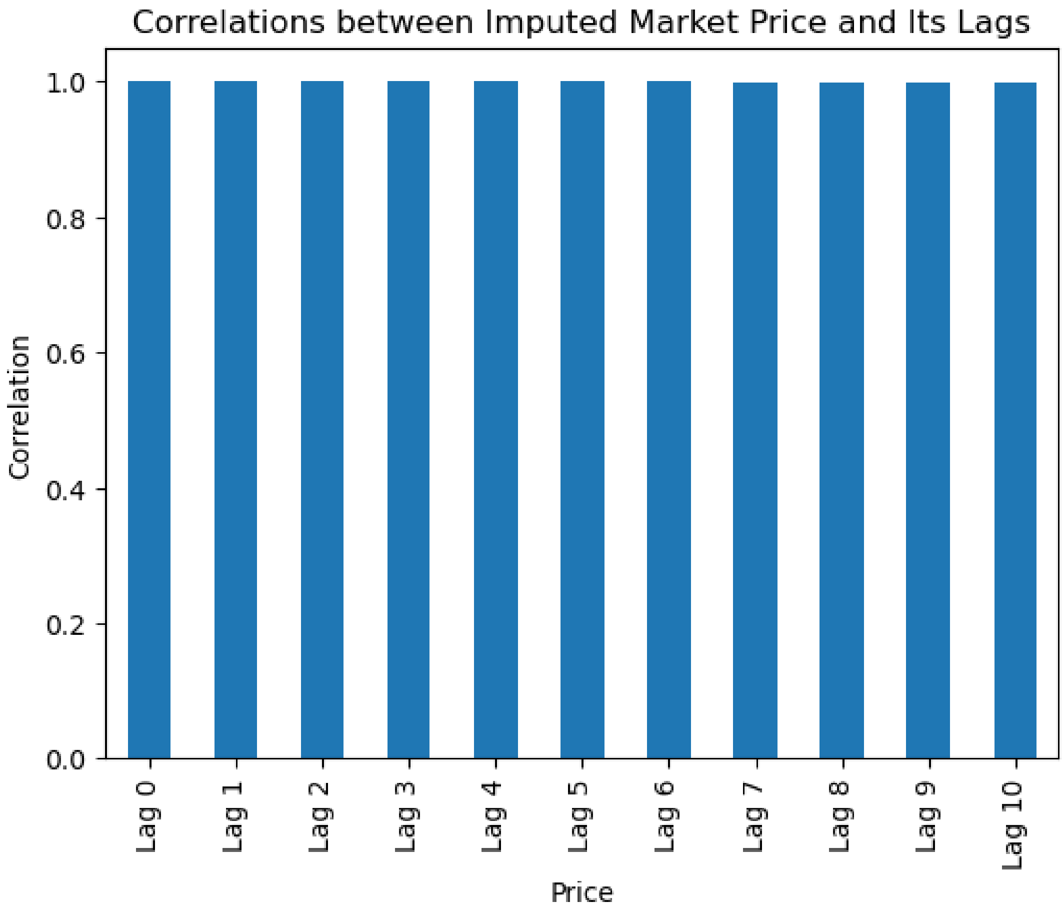
```
prices_w_lags = (
    pd.concat(
        objs=[prices.shift(t) for t in range(11)],
        keys=[f'Lag {t}' for t in range(11)],
        names=['Price'],
        axis=1,
    )
)
```

```
prices_w_lags.tail()
```

Price Date	Lag 0	Lag 1	Lag 2	Lag 3	Lag 4	Lag 5	Lag 6	Lag
2024-12-24	15038.5668	14870.9709	14778.3109	14617.9520	14633.0240	15110.9845	15182.7991	151
2024-12-26	15044.1311	15038.5668	14870.9709	14778.3109	14617.9520	14633.0240	15110.9845	151
2024-12-27	14870.6722	15044.1311	15038.5668	14870.9709	14778.3109	14617.9520	14633.0240	151
2024-12-30	14711.1099	14870.6722	15044.1311	15038.5668	14870.9709	14778.3109	14617.9520	146
2024-12-31	14645.9397	14711.1099	14870.6722	15044.1311	15038.5668	14870.9709	14778.3109	146

Now we can plot the correlation of price with its lags.

```
(
    prices_w_lags
    .dropna()
    .corr()
    .loc['Lag 0']
    .plot(kind='bar')
)
plt.title('Correlations between Imputed Market Price and Its Lags')
plt.ylabel('Correlation')
plt.show()
```



But these are *pairwise* correlations. If we estimate *conditional* correlations, we see that most of the price information is in the first lag!

```
y = prices_w_lags.dropna()['Lag 0'] # Pt
X = prices_w_lags.dropna().drop('Lag 0', axis=1).pipe(sm.add_constant) # Pt-1, Pt-2, ..., Pt-
model = sm.OLS(endog=y, exog=X)
fit = model.fit(cov_type='HAC', cov_kwds={'maxlags': 10})
fit.summary()
```

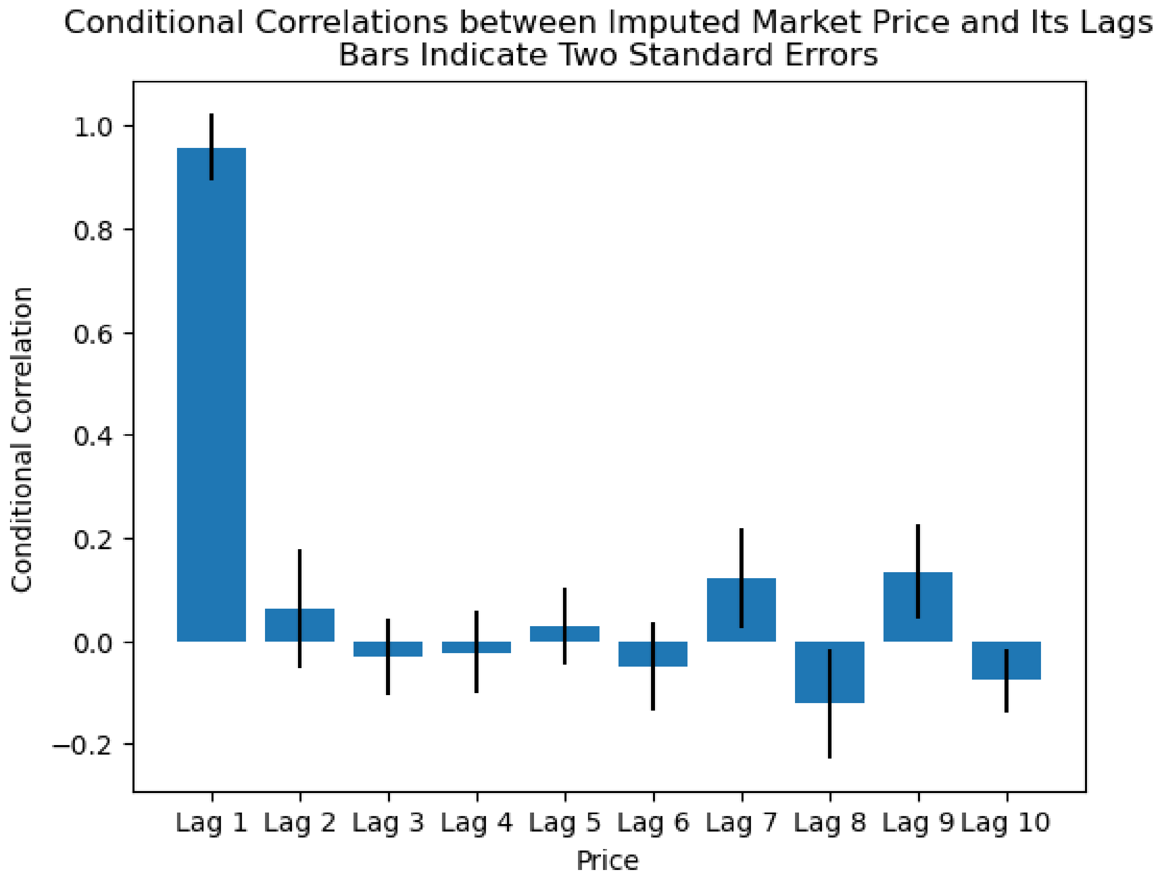
Dep. Variable:	Lag 0	**R-squared:**	1.000
Model:	OLS	**Adj. R-squared:**	1.000
Method:	Least Squares	**F-statistic:**	2.195e+06
Date:	Thu, 03 Apr 2025	**Prob (F-statistic):**	0.00
Time:	16:42:37	**Log-Likelihood:**	-1.2518e+05
No. Observations:	25891	**AIC:**	2.504e+05
Df Residuals:	25880	**BIC:**	2.505e+05
Df Model:	10		
Covariance Type:	HAC		

	coef	**std err**	**z**	**P> |z|**	**[0.025**	**0.975]**
const	-0.0144	0.144	-0.100	0.920	-0.297	0.268
Lag 1	0.9574	0.033	29.042	0.000	0.893	1.022
Lag 2	0.0621	0.058	1.077	0.281	-0.051	0.175
Lag 3	-0.0315	0.037	-0.860	0.390	-0.103	0.040
Lag 4	-0.0220	0.040	-0.553	0.580	-0.100	0.056
Lag 5	0.0290	0.038	0.767	0.443	-0.045	0.103
Lag 6	-0.0498	0.042	-1.181	0.237	-0.132	0.033
Lag 7	0.1215	0.048	2.518	0.012	0.027	0.216
Lag 8	-0.1222	0.053	-2.302	0.021	-0.226	-0.018
Lag 9	0.1331	0.046	2.908	0.004	0.043	0.223
Lag 10	-0.0771	0.030	-2.535	0.011	-0.137	-0.017

Omnibus:	15929.955	**Durbin-Watson:**	1.994
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	4213325.277
Skew:	-1.820	**Prob(JB):**	0.00
Kurtosis:	65.389	**Cond. No.**	9.90e+03

Notes:
[1] Standard Errors are heteroscedasticity and autocorrelation robust (HAC) using 10 lags and without small sample correction
[2] The condition number is large, 9.9e+03. This might indicate that there are strong multicollinearity or other numerical problems.

```
plt.bar(
    x=fit.params.index[1:],
    height=fit.params[1:],
    yerr=2*fit.bse[1:]
)
plt.title('Conditional Correlations between Imputed Market Price and Its Lags\nBars Indicate
plt.ylabel('Conditional Correlation')
plt.xlabel('Price')
plt.show()
```



Signal-to-Noise Ratio

Recall, we can express a random walk as $P_t = \rho P_{t-1} + m P_{t-1} + \varepsilon_t$. Since $\rho = 1$, we can subtract P_{t-1} from both sides, then divide by P_{t-1} on both sides. This transformation expresses a random walk in terms of returns: $r_{t-1,t} = m + e_t$, where $\mathrm{E}[e_t] = 0$ and $\mathrm{SD}[e_t] = s$, so $\mathrm{E}[r_{t-1,t}] = m$. We can think of the signal-to-noise ratio as $\frac{m}{s}$. How high is this ratio?

```
m, s = ff['Mkt'].mean(), ff['Mkt'].std()
```

Here m is about 4 basis points per day!

```
m
```

```
0.0004
```

However, s is about 108 basis points per day!

```
s
```

```
0.0108
```

Therefore, the signal-to-noise ratio is less than 0.04! We want this ratio above 2 to reject that a drift (or a strategy) is zero.

```
m/s
```

```
0.0397
```

Recall that means grow linearly with time and standard deviations growth with the square-root of time. So, if we want $\sqrt{t} \times \frac{m}{s} \geq 2$, we need $t \geq \left(2 \times \frac{s}{m}\right)^2$ days! Even with market portfolio noise, which is diversified and low, we needat least a decade! During this decade, the true values of m and s can change!

```
(2 * s / m)**2 / 252
```

```
10.0486
```

Implement a simple moving average (SMA) trading strategy

The over-simplified goal of technical analysis is to "buy low, and sell high." The n-day SMA reduces noise in market prices, removing market fluctuations and providing estimates of "true" prices. While the market price is *above* the SMA, the SMA *rises.* While the market price is *below* the SMA, the SMA *falls.* So, if we buy the stock as it cross the SMA from below and sell the stock as it crosses the SMA from above, we mechanically buy low and sell high! Here, we will implement a long-only 20-day SMA strategy with Bitcoin:

1. Buy when the closing price crosses SMA(20) from below
2. Sell when the closing price crosses SMA(20) from above
3. No short-selling

We can simplify this strategy to "long if above SMA(20), otherwise neutral". First, we will need Bitcoin returns data.

```
btc = (
    yf.download(
        tickers='BTC-USD',
        auto_adjust=False,
        progress=False,
        multi_level_index=False
    )
    .assign(Return=lambda x: x['Adj Close'].pct_change())
)
```

```
btc.head()
```

Date	Adj Close	Close	High	Low	Open	Volume	Return
2014-09-17	457.3340	457.3340	468.1740	452.4220	465.8640	21056800	NaN
2014-09-18	424.4400	424.4400	456.8600	413.1040	456.8600	34483200	-0.0719
2014-09-19	394.7960	394.7960	427.8350	384.5320	424.1030	37919700	-0.0698
2014-09-20	408.9040	408.9040	423.2960	389.8830	394.6730	36863600	0.0357
2014-09-21	398.8210	398.8210	412.4260	393.1810	408.0850	26580100	-0.0247

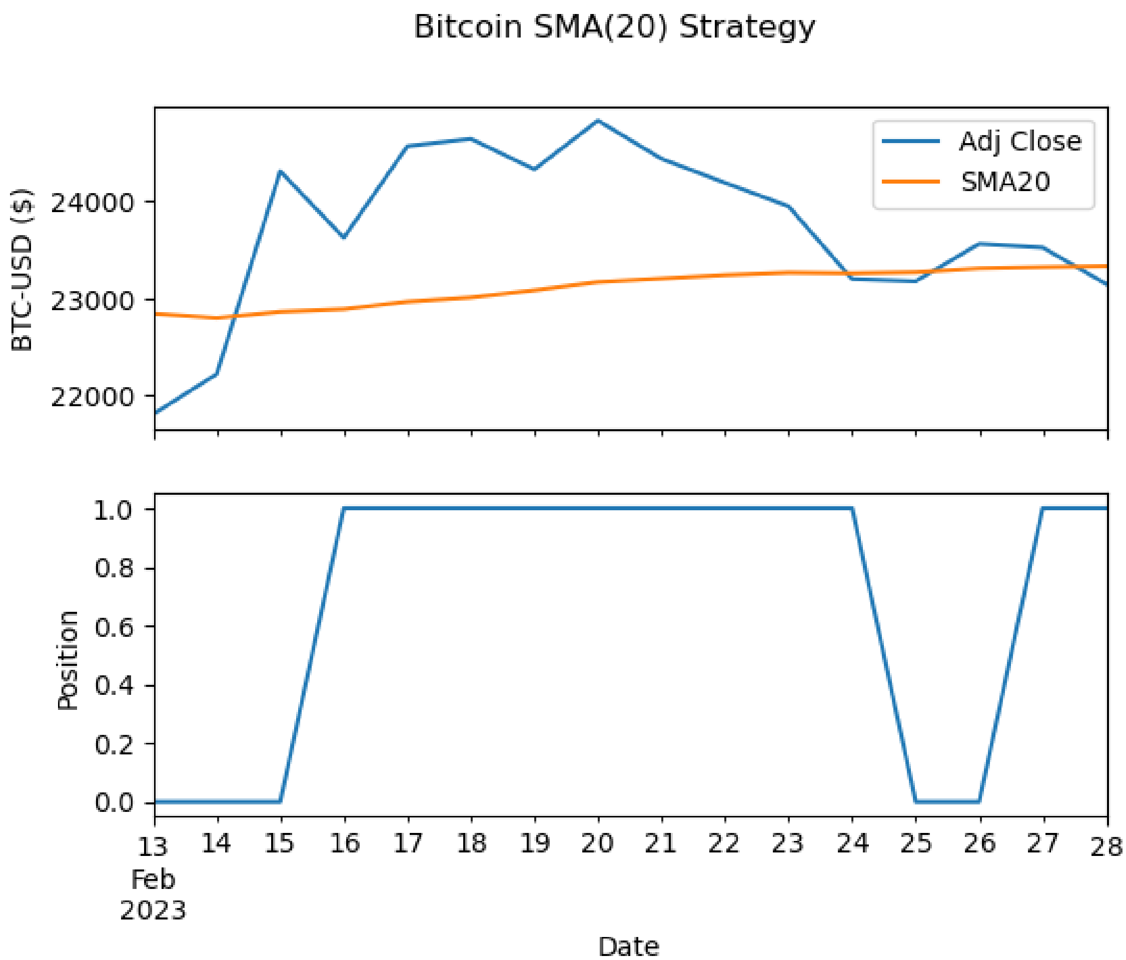
Next we:

1. Use `.rolling(20).mean()` to add a `SMA20` column containing SMA(20) to our `btc` data frame
2. Use `np.select()` to add a `Position` column containing:
    1. `1` (long) when the adjusted close is greater than SMA(20)
    2. `0` (neutral) when the adjusted close is less than (or equal to) SMA(20)
    3. **We use `.shift()` to compare yesterday's closing prices, avoiding a look-ahead bias**
    4. `np.select()` tests multiple conditions and provides a default, making it more flexible framework than `np.where()`
3. Add a `Strategy` column containing:
    1. `Return` if `Position == 1`
    2. `0` if `Position == 0`
    3. We could earn the risk-free rate instead of 0 percent, but earning 0 percent simplifies this example

```
btc = (
    btc
    .assign(
        SMA20=lambda x: x['Adj Close'].rolling(20).mean(),
        Position=lambda x: np.select(
            condlist=[
                x['Adj Close'].shift() > x['SMA20'].shift(),
                x['Adj Close'].shift() <= x['SMA20'].shift()
            ],
            choicelist=[1, 0],
            default=np.nan
        ),
        Strategy=lambda x: x['Position'] * x['Return']
    )
)
```

I find it helpful to plot `Adj Close`, `SMA20`, and `Position` for a sort window with one or more crossings.

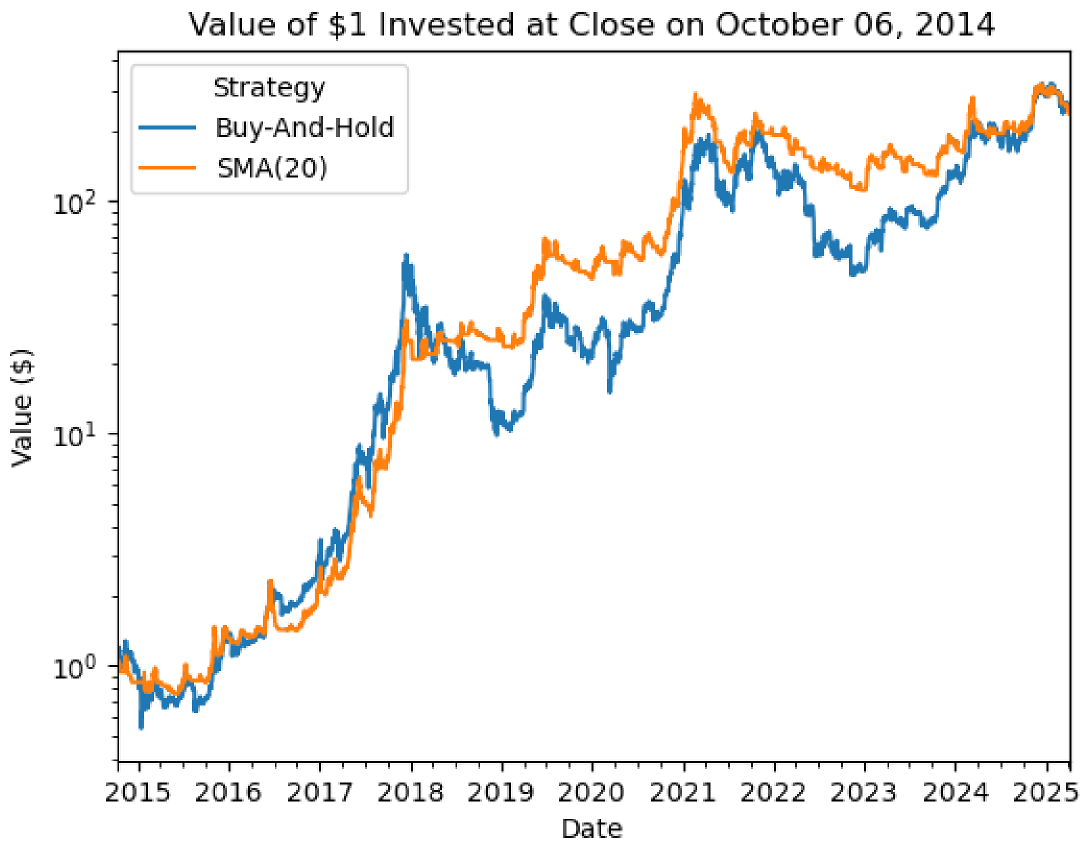
```
fig, ax = plt.subplots(2, 1, sharex=True)
df = btc.loc['2023-02-13':'2023-02-28']
df[['Adj Close', 'SMA20']].plot(ax=ax[0], ylabel='BTC-USD ($)')
df[['Position']].plot(ax=ax[1], ylabel='Position', legend=False)
plt.suptitle('Bitcoin SMA(20) Strategy')
plt.show()
```



We can compare the long-run performance of buy-and-hold and SMA(20).

```
df = btc[['Return', 'Strategy']].dropna()

(
    df
    .add(1)
    .cumprod()
    .rename_axis(columns='Strategy')
    .rename(columns={'Return': 'Buy-And-Hold', 'Strategy': 'SMA(20)'})
    .plot()
)
plt.semilogy()
plt.ylabel('Value ($)')
```

```
plt.title(f'Value of $1 Invested at Close on {df.index[0] - pd.offsets.Day(1):%B %d, %Y}')
plt.show()
```



In the practice notebook, we will dig deeper on this strategy and others.

```
df.add(1).prod()
```

```
Return      248.1171
Strategy    237.1394
dtype: float64
```

Herron Topic 2 - Practice

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import pandas_datareader as pdr
import statsmodels.api as sm
import yfinance as yf
```

```
%precision 4
pd.options.display.float_format = '{:.4f}'.format
# %config InlineBackend.figure_format = 'retina'
```

Announcements

Five-Minute Review

Practice

Implement the SMA(20) strategy with Bitcoin from the lecture notebook

Try to create the `btc` data frame in one code cell with one assignment (i.e., one `=`).

Investigate how SMA(20) generates returns

Consider the following:

1. Does SMA(20) avoid the worst performing days? How many of the worst 20 days does SMA(20) avoid? Try the `.sort_values()` or `.nlargest()` method.
2. Does SMA(20) preferentially avoid low-return days? Try to combine the `.groupby()` method and `pd.qcut()` function.
3. Does SMA(20) preferentially avoid high-volatility days? Try to combine the `.groupby()` method and `pd.qcut()` function.

Implement the SMA(20) strategy with IBM

How often does SMA(20) outperform buy-and-hold with 10-year rolling windows?

Implement a long-only BB(20, 2) strategy with Bitcoin

Bollinger Bands are bands around a trend, typically defined in terms of simple moving averages and volatilities. A long-only BB(20, 2) strategy has upper and lower bands at 2 standard deviations above and below the SMA(20). It invests as follows:

1. Buy when the closing price crosses LB(20) from below
2. Sell when the closing price crosses UB(20) from above
3. No short-selling

The long-only BB(20, 2) is more difficult to implement than the long-only SMA(20) because we need to track buys and sells. For example, if the closing price is between LB(20) and BB(20), we need to know if our last trade was a buy or a sell. Further, if the closing price is below LB(20), we can still be long because we sell when the closing price crosses UB(20) from above.

More on Bollinger Bands here and here.

Implement a long-short RSI(14) strategy with Bitcoin

From Fidelity:

> The Relative Strength Index (RSI), developed by J. Welles Wilder, is a momentum oscillator that measures the speed and change of price movements. The RSI oscillates between zero and 100. Traditionally the RSI is considered overbought when above 70 and oversold when below 30. Signals can be generated by looking for divergences and failure swings. RSI can also be used to identify the general trend.

The RSI formula: $RSI(n) = 100 - \frac{100}{1+RS(n)}$, where $RS(n) = \frac{SMA(U,n)}{SMA(D,n)}$. For "up days", $U = \Delta \text{Adj Close}$ and $D = 0$. For "down days", $U = 0$ and $D = -\Delta \text{Adj Close}$.

We will implement a long-short RSI(14) as follows:

1. Buy when the RSI crosses 30 from below, and sell when the RSI crosses 50 from below
2. Short when the RSI crosses 70 from above, and cover when the RSI crosses 50 from above

More about RSI here.

Implement a golden cross with Bitcoin

From Grok:

> In technical analysis, a golden cross is a bullish chart pattern that occurs when a short-term moving average (typically the 50-day moving average) crosses above a long-term moving average (typically the 200-day moving average). This crossover is considered a signal that a stock, index, or other asset may be entering a sustained upward trend, suggesting potential buying opportunities for traders and investors.

More here.

Compare all strategies with Bitcoin

Herron Topic 2 - Practice - Sec 02

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import pandas_datareader as pdr
import seaborn as sns
import statsmodels.api as sm
import yfinance as yf
```

```
%precision 4
pd.options.display.float_format = '{:.4f}'.format
# %config InlineBackend.figure_format = 'retina'
```

Announcements

1. I am still grading your projects; I hope to finish them by next Tuesday
2. I posted 50 practice problems to prepare for the end-of-course Programming Assessment here: https://northeastern.instructure.com/courses/207607/discussion_topics/2727917
    1. I built 5 autograded notebooks to help you prepare for the assessment, and I will build more in the coming weeks
    2. To run these notebooks, install the `otter-grader` package: In the Anaconda command prompt (or Terminal on Mac) run `conda activate fina6333` then `conda install otter-grader`
    3. See the video at the link above for details

Five-Minute Review

1. Technical analysis (TA) is a method of evaluating trends in trading prices and volume (and open interest in the futures and options markets only)
2. The three tenants of TA are:
    1. Markets discount everything

   2. Prices move in trends
   3. History repeats itself, so these trends are recurring

3. Academics have not found much evidence that TA generate profits that exceed transaction costs, but we will spend a week on it for three reasons:

   1. You asked for it, and it will help build our data analytics skills
   2. It is a small part of the Chartered Financial Analyst (CFA) curriculum
   3. That TA still receives attention 60 years after the Efficient Markets Hypothesis (EMH) suggests that it has some value that academics have been unable to measure

Practice

Implement the SMA(20) strategy with BTC-USD from the lecture notebook

Try to create the `btc_sma` data frame from the `btc` data frame in one code cell with one assignment (i.e., one `=`).

```
btc = (
    yf.download(
        tickers='BTC-USD',
        auto_adjust=False,
        progress=False,
        multi_level_index=False
    )
    .iloc[:-1] # drop incomplete trading day
)
```

After class, I converted our code to a function. We might want to test different moving average parameters, and a function makes these tests easier.

The following `calc_sma()` function accepts:

1. A data frame `df` of daily values from `yfinance.download()`
2. An integer `window` that specifies the number of trading days in the SMA window

And returns the original data frame `df` plus the following columns:

1. `Return` with daily returns
2. `SMA` for the `window`-trading-day moving average
3. `Position` for the weight on the security each day
4. `Strategy` for the return on the strategy each day

```
def calc_sma(df, window=20):
    return (
        df
        .assign(
            Return=lambda x: x['Adj Close'].pct_change(),
            SMA=lambda x: x['Adj Close'].rolling(window=window).mean(),
            Position=lambda x: np.select(
                condlist=[
                    x['Adj Close'].shift(1) > x['SMA'].shift(1),
                    x['Adj Close'].shift(1) <= x['SMA'].shift(1)
                ],
                choicelist=[1, 0],
                default=np.nan
            ),
            Strategy=lambda x: x['Position'] * x['Return']
        )
    )
```

```
btc_sma = btc.pipe(calc_sma, window=20)
```

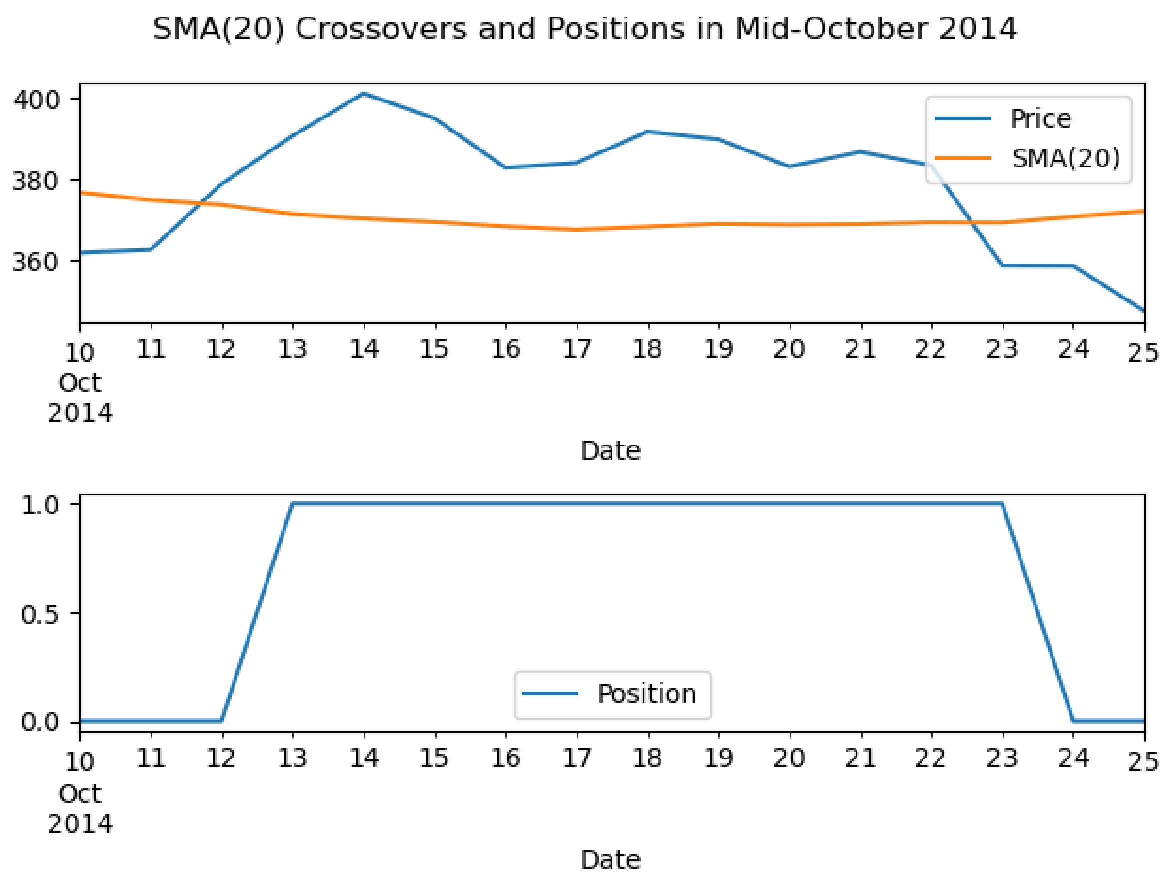
It can be helpful to visualize a few places where the price (here `Adj Close`) crosses the moving average (here SMA(20)). In class, we found that there are a few changes in position in the middle of October, 2014. The following code create two subplots in one figure, then puts one plot in each of the subplots.

For the first crossover on October 12, we *buy* at the close on October 12 and earn the security return on October 13. For the second crossover on October 23, we *sell* at the close on October 23 and earn zero return on October 24.

```
columns_sma20 = {
    'Adj Close': 'Price',
    'SMA': 'SMA(20)',
    'Return': 'Buy-And-Hold',
    'Strategy': 'SMA(20)'
}
```

```
fig, ax = plt.subplots(2, 1)
df = btc_sma.loc['2014-10-10':'2014-10-25']
df[['Adj Close', 'SMA']].rename(columns=columns_sma20).plot(ax=ax[0])
df[['Position']].plot(ax=ax[1])
plt.suptitle('SMA(20) Crossovers and Positions in Mid-October 2014')
```

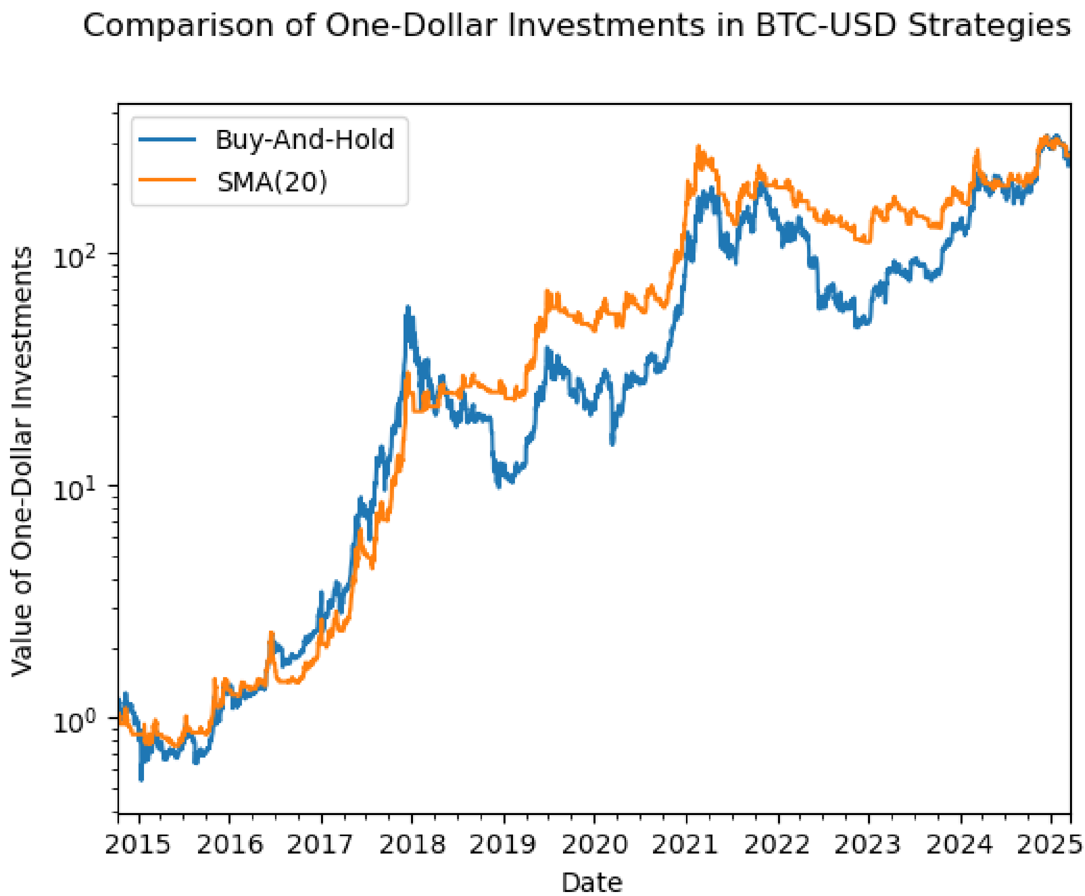
```
plt.tight_layout()
plt.show()
```



We can compare the total returns on BTC-USD and our SMA(20) strategy! We `.dropna()` first because the SMA(20) strategy needs 20 days of data to make its first investing decision.

```
(
    btc_sma
    [['Return', 'Strategy']]
    .dropna()
    .rename(columns=columns_sma20)
    .add(1)
    .cumprod()
    .plot()
)
plt.semilogy()
```

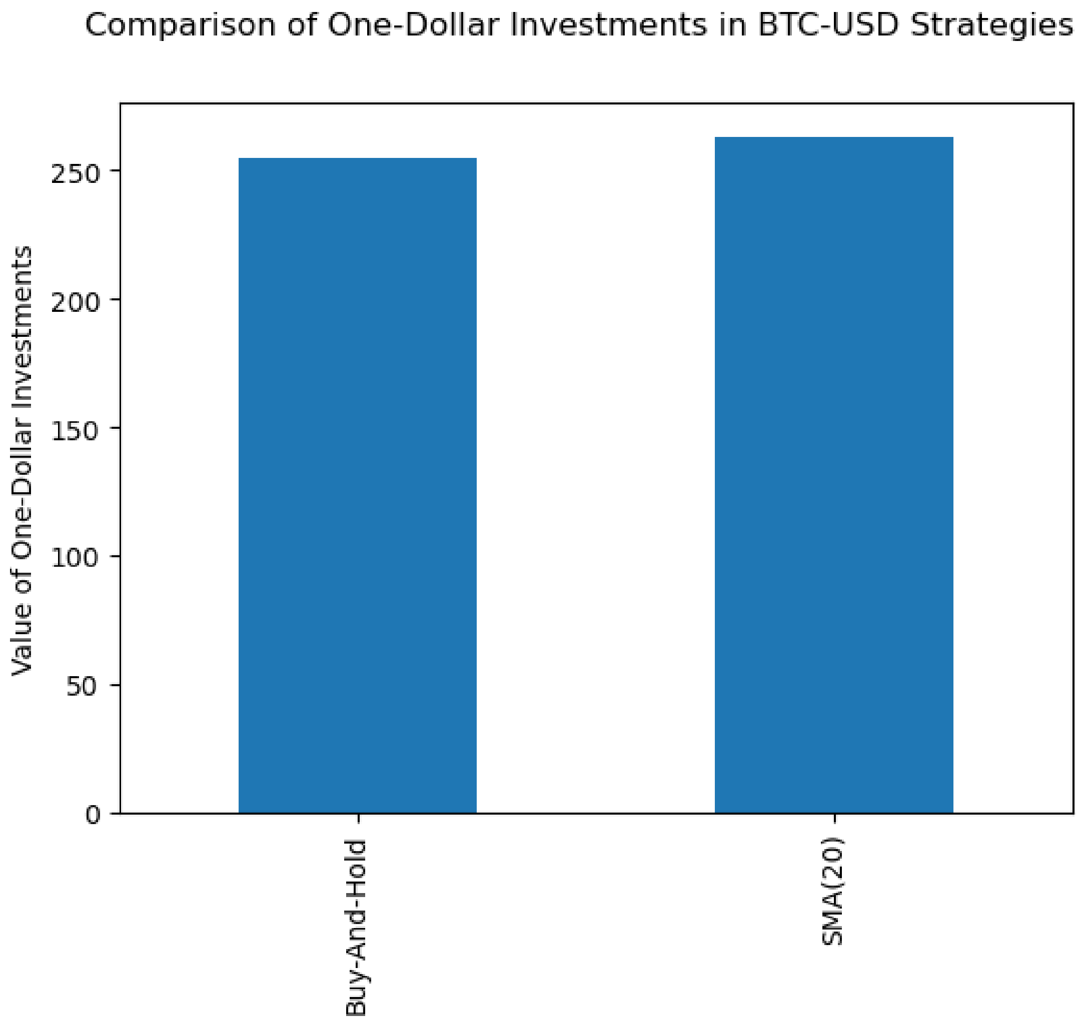
```
plt.suptitle('Comparison of One-Dollar Investments in BTC-USD Strategies')
plt.ylabel('Value of One-Dollar Investments')
plt.show()
```



These total returns, at least today, are similar!

```
(
    btc_sma
    [['Return', 'Strategy']]
    .dropna()
    .rename(columns=columns_sma20)
    .add(1)
    .prod()
    .plot(kind='bar')
)
plt.suptitle('Comparison of One-Dollar Investments in BTC-USD Strategies')
```

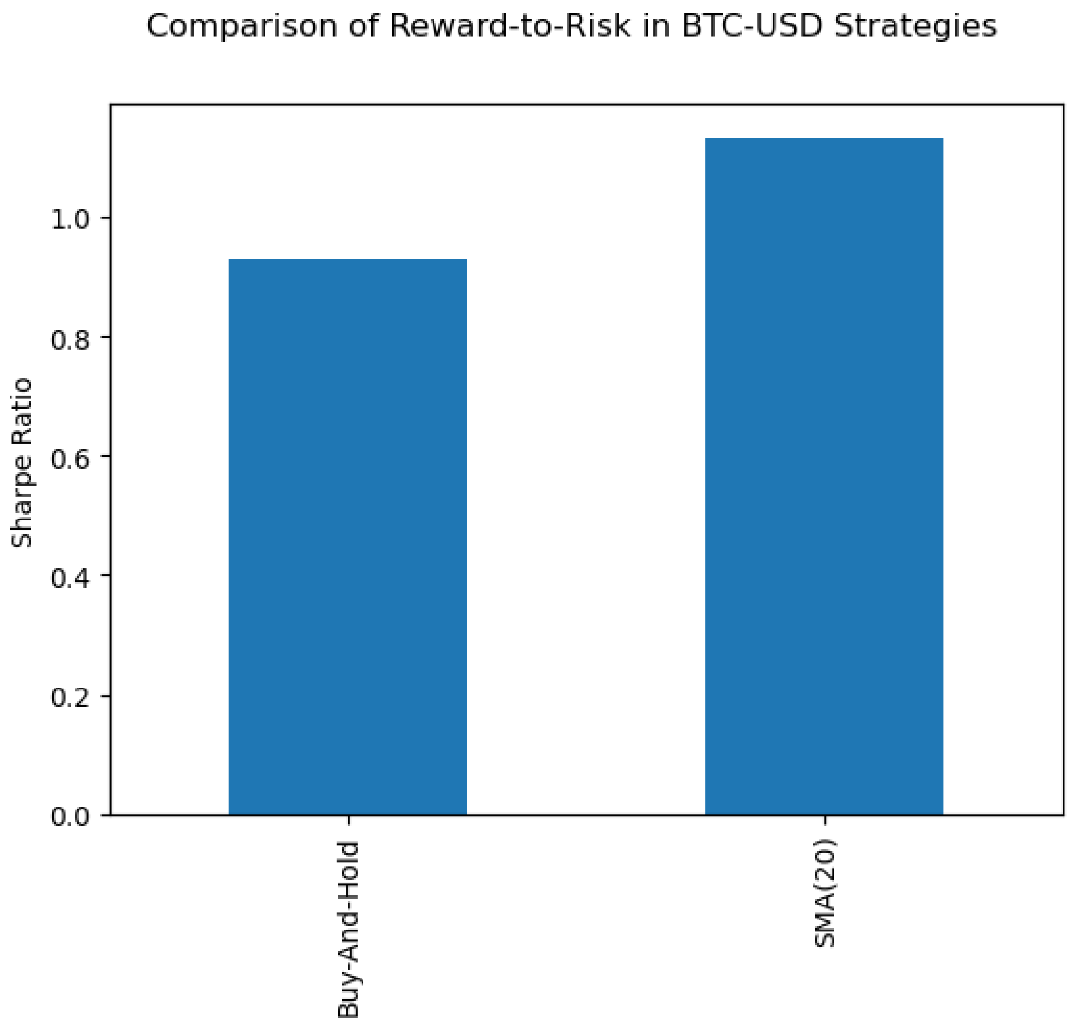
```
plt.ylabel('Value of One-Dollar Investments')
plt.show()
```



However, the Sharpe ratio of the SMA(20) is higher, because time out of the market when `Position=0` reduces risk. For simplicity, we will ignore the risk-free rate in the Sharpe ratio formula.

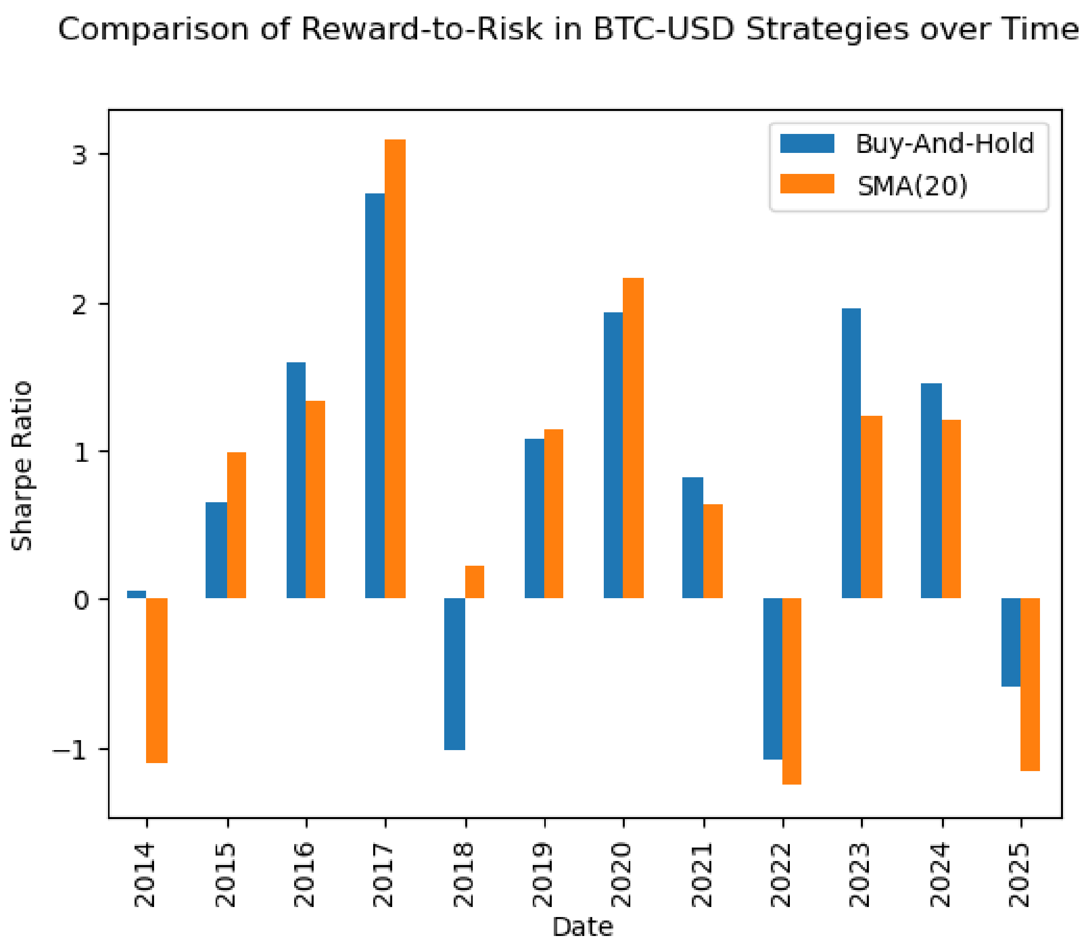
```
(
    btc_sma
    [['Return', 'Strategy']]
    .dropna()
    .rename(columns=columns_sma20)
```

```
    .apply(lambda x: np.sqrt(252) * x.mean() / x.std())
    .plot(kind='bar')
)
plt.suptitle('Comparison of Reward-to-Risk in BTC-USD Strategies')
plt.ylabel('Sharpe Ratio')
plt.show()
```



But how persistent is this outperformance in terms of reward-to-risk ratios? We can quickly modify the code above to calculate Sharpe ratios each year! We find that SMA(20) Sharpe ratio edge is not persistent.

```
df = (
    btc_sma
    [['Return', 'Strategy']]
    .dropna()
    .rename(columns=columns_sma20)
    .resample('YE')
    .apply(lambda x: np.sqrt(252) * x.mean() / x.std())
)
df.index = df.index.year
df.plot(kind='bar')
plt.suptitle('Comparison of Reward-to-Risk in BTC-USD Strategies over Time')
plt.ylabel('Sharpe Ratio')
plt.show()
```



We can use our list comprehension skills to easily try several window sizes!

```
def CAGR(x):
    T = x.count()
    return (x.add(1).prod() ** (252 / T)) - 1
```

```
def Sharpe(x, ann_fac=np.sqrt(252)):
    return ann_fac * x.mean() / x.std()
```

```
columns_sman = {
    'Adj Close': 'Price',
    'SMA': 'SMA(N)',
    'Return': 'Buy-And-Hold',
    'Strategy': 'SMA(N)'
}
```

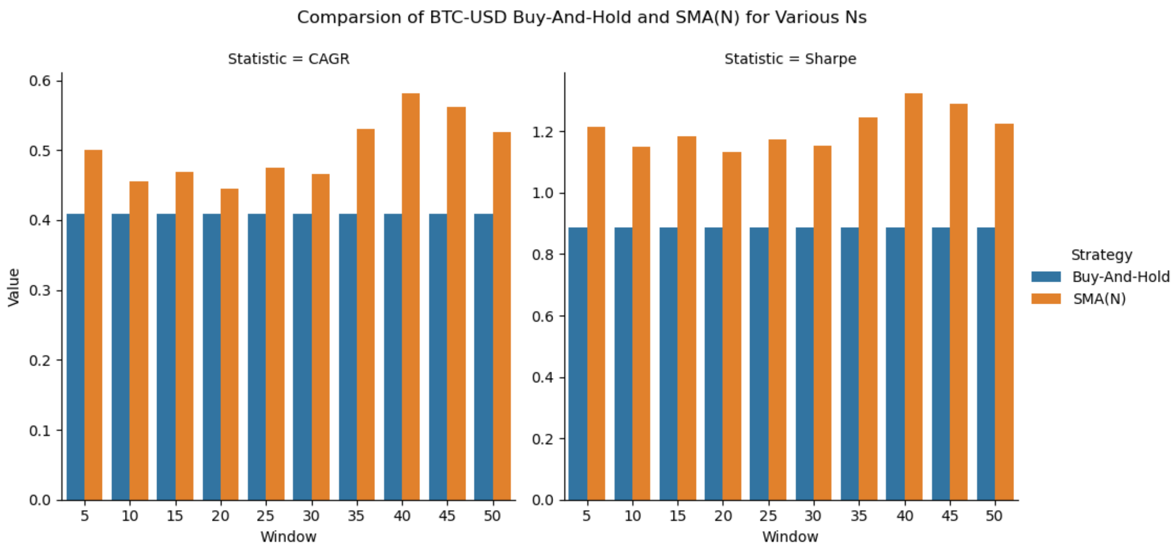
```
windows = list(range(5, 55, 5))

btc_smas = (
    pd.concat(
        objs=[
            btc.pipe(calc_sma, window=w)[['Return', 'Strategy']].agg([CAGR, Sharpe])
            for w in windows
        ],
        keys=windows,
        names=['Window', 'Statistic']
    )
    .rename(columns=columns_sman)
)
```

```
btc_smas
```

		Buy-
Window	Statistic	
5	CAGR	0.40
	Sharpe	0.88
10	CAGR	0.40
	Sharpe	0.88

Window	Statistic	Buy-
15	CAGR	0.40
	Sharpe	0.88
20	CAGR	0.40
	Sharpe	0.88
25	CAGR	0.40
	Sharpe	0.88
30	CAGR	0.40
	Sharpe	0.88
35	CAGR	0.40
	Sharpe	0.88
40	CAGR	0.40
	Sharpe	0.88
45	CAGR	0.40
	Sharpe	0.88
50	CAGR	0.40
	Sharpe	0.88

```
(
    btc_smas
    .reset_index()
    .melt(
        id_vars=['Window', 'Statistic'],
        value_vars=['Buy-And-Hold', 'SMA(N)'],
        var_name='Strategy',
        value_name='Value'
    )
    .pipe(
        sns.catplot,
        x='Window',
        col='Statistic',
        hue='Strategy',
        y='Value',
        kind='bar',
        sharey=False
    )
)
plt.suptitle('Comparsion of BTC-USD Buy-And-Hold and SMA(N) for Various Ns', y=1.05)
plt.show()
```



Investigate how SMA(20) generates returns

Consider the following:

1. Does SMA(20) avoid the worst performing days? How many of the worst 20 days does SMA(20) avoid? Try the `.nlargest()` method.
2. Does SMA(20) preferentially avoid low-return days? Try to combine the `.groupby()` method and `pd.qcut()` function.
3. Does SMA(20) preferentially avoid high-volatility days? Try to combine the `.groupby()` method and `pd.qcut()` function.

The SMA(20) does well here because it avoids 17 of the 20 worst days, without avoiding the best days.

```python
btc_sma.loc[btc_sma['Return'].nsmallest(20).index, ['Position']].value_counts()
```

```
Position
0.0000      17
1.0000       3
Name: count, dtype: int64
```

```
btc_sma.loc[btc_sma['Return'].nlargest(20).index, ['Position']].value_counts()
```

```
Position
0.0000      10
1.0000      10
Name: count, dtype: int64
```

We can also look at the descriptive statistics by `Postiion` and strategy. Two observations:

1. The `min` column shows that SMA(20) misses the bad days, like we see above
2. The `mean` in `Position=1` is about 5 times higher than in `Position=0`

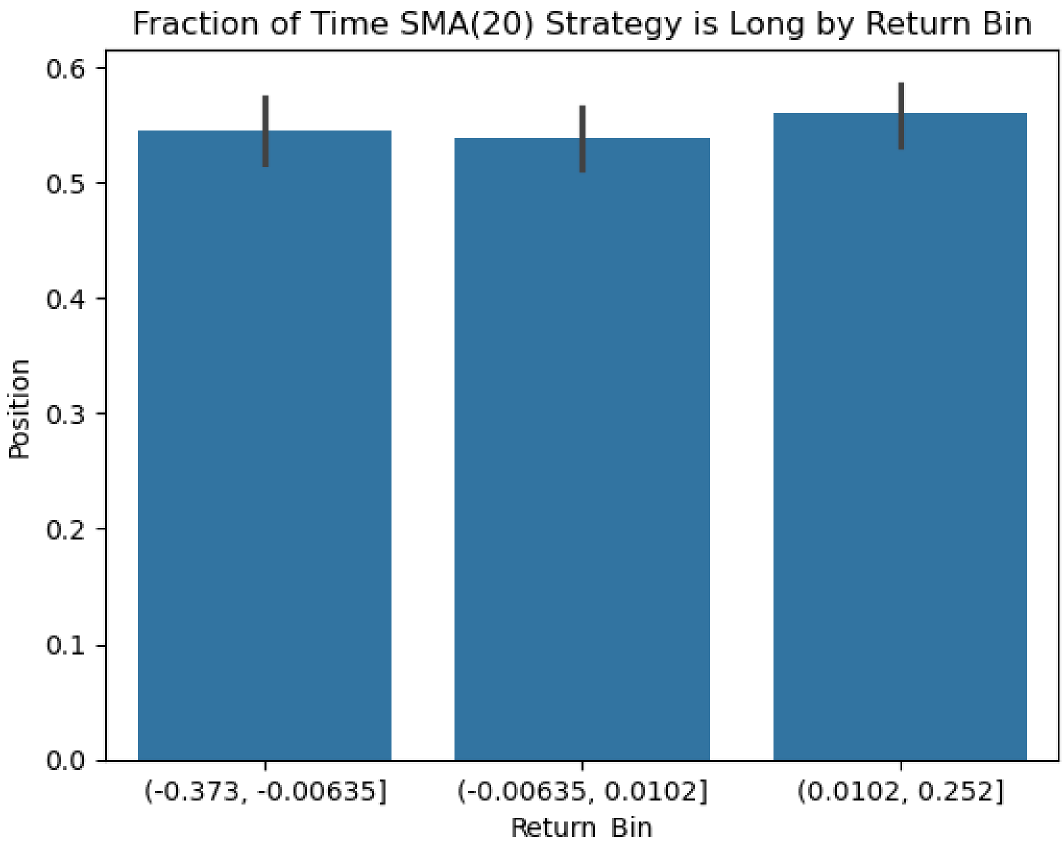
```
(
    btc_sma
    [['Position', 'Return', 'Strategy']]
    .dropna()
    .groupby('Position')
    .describe()
    .rename(columns=columns_sma20)
    .rename_axis(columns=['Strategy', 'Statistic'])
    .stack('Strategy', future_stack=True)
)
```

Position	Statistic Strategy
0.0000	Buy-And-Hold
	SMA(20)
1.0000	Buy-And-Hold
	SMA(20)

We can also use the seaborn package to visualize `Position` (i.e., the portfolio weight on Bitcoin) during periods of high and low Bitcoin returns and volatility. The SMA(20) strategy is long Bitcoin about 55% of the time, whether Bitcoin returns are high (bin 2) or low (bin 1).

```
(
    btc_sma
    [['Return', 'Position']]
    .dropna()
    .assign(Return_Bin=lambda x: pd.qcut(x['Return'], q=3))
```

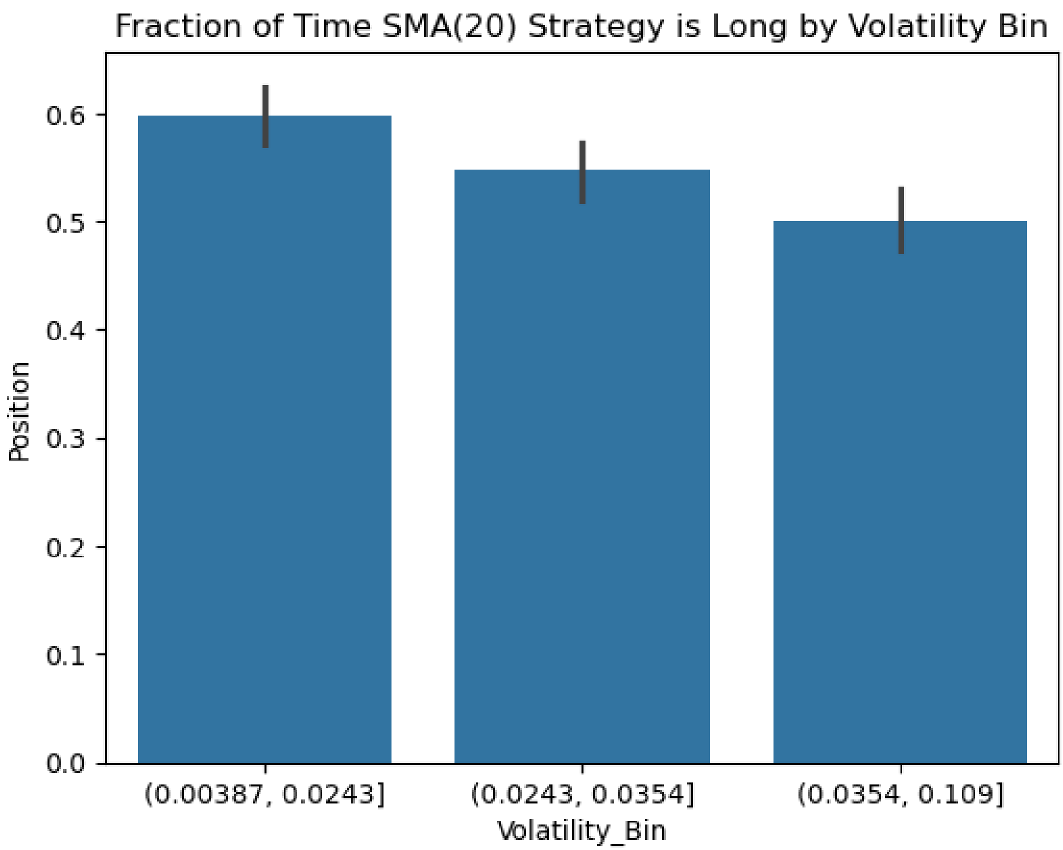
```
    .pipe(
        sns.barplot,
        x='Return_Bin',
        y='Position'
    )
)

plt.title('Fraction of Time SMA(20) Strategy is Long by Return Bin')
plt.show()
```



However, the SMA(20) stateegy, *for this security, sample, and window,* spends less times in Bitcoind during volatilit times.

```
(
    btc_sma
    [['Return', 'Position']]
    .dropna()
```

```
    .assign(Volatility_Bin=lambda x: pd.qcut(x['Return'].rolling(20).std(), q=3))
    .pipe(
        sns.barplot,
        x='Volatility_Bin',
        y='Position'
    )
)

plt.title('Fraction of Time SMA(20) Strategy is Long by Volatility Bin')
plt.show()
```



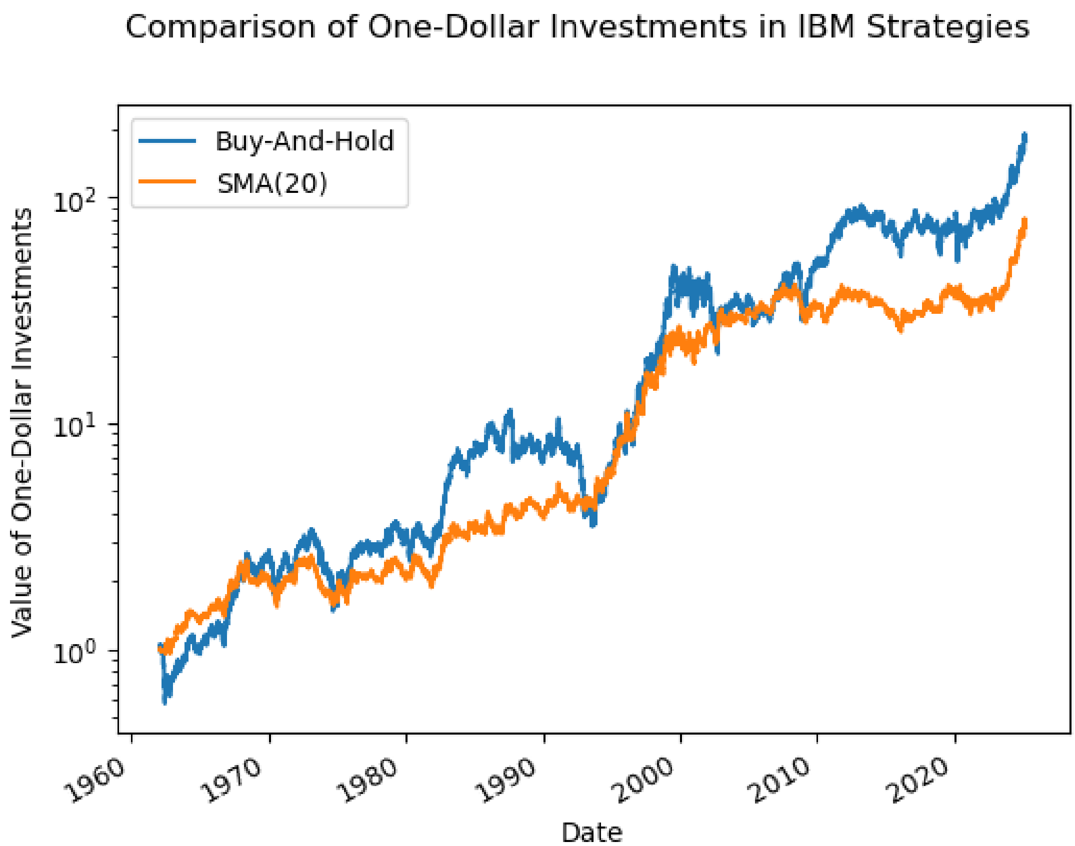
Implement the SMA(20) strategy with IBM

How often does SMA(20) outperform buy-and-hold with 10-year rolling windows?

```
ibm = (
    yf.download(
        tickers='IBM',
        auto_adjust=False,
        progress=False,
        multi_level_index=False
    )
    .iloc[:-1] # drop incomplete trading day
)
```

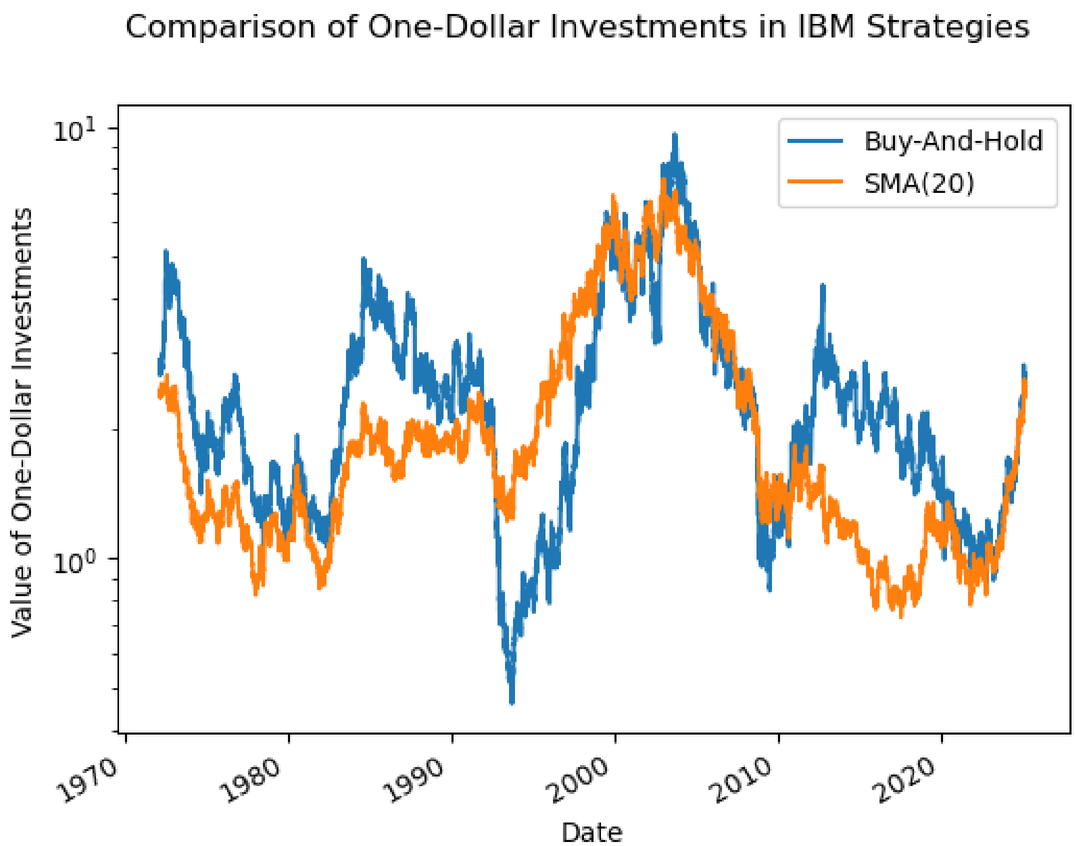
```
ibm_sma = ibm.pipe(calc_sma, window=20)
```

```
(
    ibm_sma
    [['Return', 'Strategy']]
    .dropna()
    .rename(columns=columns_sma20)
    .add(1)
    .cumprod()
    .plot()
)
plt.semilogy()
plt.suptitle('Comparison of One-Dollar Investments in IBM Strategies')
plt.ylabel('Value of One-Dollar Investments')
plt.show()
```



Over the full, 60-year sample, SMA(20) underperforms buy-and-hold. What about on rolling ten-year windows? The results look balanced, and neither clearly outperforms.

```
(
    ibm_sma
    [['Return', 'Strategy']]
    .dropna()
    .rename(columns=columns_sma20)
    .pipe(np.log1p)
    .rolling(window=10*252)
    .sum()
    .pipe(np.exp)
    .plot()
)
plt.semilogy()
plt.suptitle('Comparison of One-Dollar Investments in IBM Strategies')
plt.ylabel('Value of One-Dollar Investments')
plt.show()
```



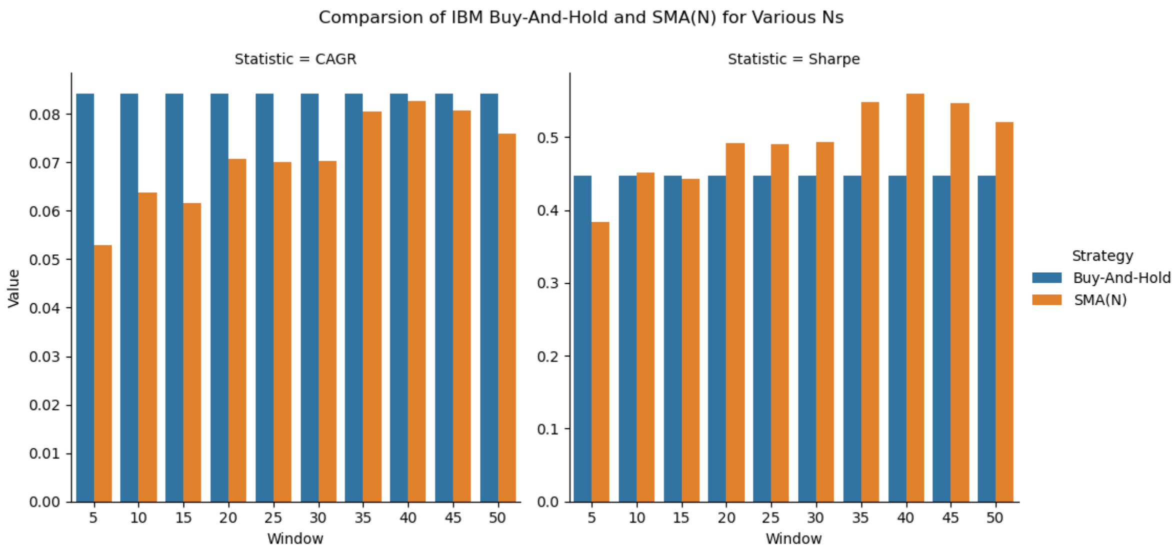
We can quantify how often SMA(20) outperform buy-and-hold. SMA(20) outperforms only 24% of the time!

```
(
    ibm_sma
    [['Return', 'Strategy']]
    .dropna()
    .pipe(np.log1p)
    .rolling(window=10*252)
    .sum()
    .pipe(np.exp)
    .pipe(lambda x: x['Strategy'] > x['Return'])
    .mean()
)
```

```
0.2382
```

```
windows = list(range(5, 55, 5))

ibm_smas = (
    pd.concat(
        objs=[
            ibm.pipe(calc_sma, window=w)[['Return', 'Strategy']].agg([CAGR, Sharpe])
            for w in windows
        ],
        keys=windows,
        names=['Window', 'Statistic']
    )
    .rename(columns=columns_sman)
)
```

```
(
    ibm_smas
    .reset_index()
    .melt(
        id_vars=['Window', 'Statistic'],
        value_vars=['Buy-And-Hold', 'SMA(N)'],
        var_name='Strategy',
        value_name='Value'
    )
    .pipe(
        sns.catplot,
        x='Window',
        col='Statistic',
        hue='Strategy',
        y='Value',
        kind='bar',
        sharey=False
    )
)
plt.suptitle('Comparsion of IBM Buy-And-Hold and SMA(N) for Various Ns', y=1.05)
plt.show()
```



Implement a long-only BB(20, 2) strategy with Bitcoin

Bollinger Bands are bands around a trend, typically defined in terms of simple moving averages and volatilities. A long-only BB(20, 2) strategy has upper and lower bands at 2 standard deviations above and below the SMA(20). It invests as follows:

1. Buy when the closing price crosses LB(20) from below
2. Sell when the closing price crosses UB(20) from above
3. No short-selling

The long-only BB(20, 2) is more difficult to implement than the long-only SMA(20) because we need to track buys and sells. For example, if the closing price is between LB(20) and BB(20), we need to know if our last trade was a buy or a sell. Further, if the closing price is below LB(20), we can still be long because we sell when the closing price crosses UB(20) from above.

More on Bollinger Bands here and here.

```
def calc_bb(df, m=20, n=2):
    return (
        df
        .assign(
            Return=lambda x: x['Adj Close'].pct_change(),
            SMA=lambda x: x['Adj Close'].rolling(window=m).mean(),
            SMV=lambda x: x['Adj Close'].rolling(window=m).std(),
            UB=lambda x: x['SMA'] + n*x['SMV'],
```

```
            LB=lambda x: x['SMA'] - n*x['SMV'],
            Position_w_nan=lambda x: np.select(
                condlist=[
                    (x['Adj Close'].shift(1) > x['LB'].shift(1)) & (x['Adj Close'].shift(2)
                    (x['Adj Close'].shift(1) < x['UB'].shift(1)) & (x['Adj Close'].shift(2)
                ],
                choicelist=[1, 0],
                default=np.nan
            ),
            Position=lambda x: x['Position_w_nan'].ffill(),
            Strategy=lambda x: x['Position'] * x['Return']
        )
    )
```

```
btc_bb = btc.pipe(calc_bb)
```

The BB(20, 2) only spends 40% of its time long BTC!

```
btc_bb['Position'].mean()
```

```
0.4013
```

And BTC-USD performance is worse when it long that when its neutral!

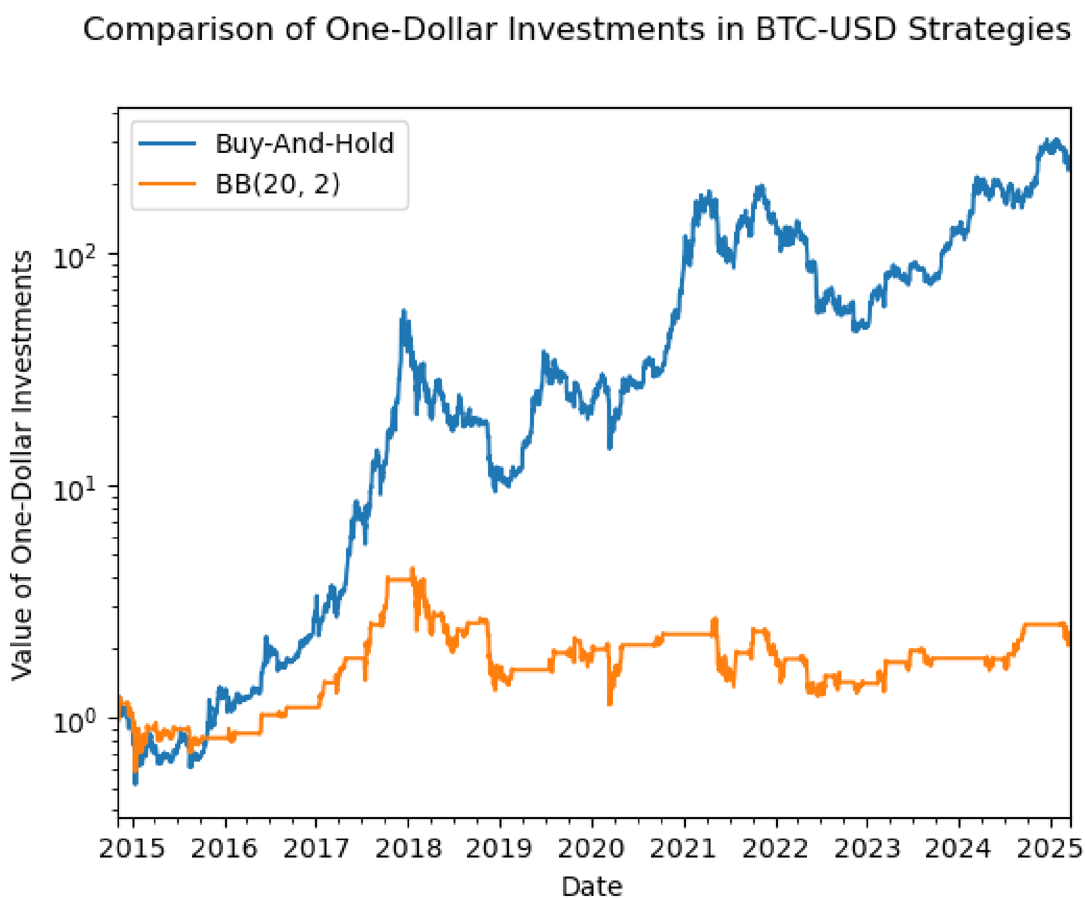
- Mean daily return is lower
- Volatility of daily returns is higher
- Every percentile of the distribution is worse

```
btc_bb.groupby('Position')['Return'].describe()
```

	count	mean	std	min	25%	50%	75%	max
Position								
0.0000	2268.0000	0.0027	0.0343	-0.1874	-0.0110	0.0019	0.0172	0.2525
1.0000	1520.0000	0.0013	0.0386	-0.3717	-0.0149	0.0007	0.0161	0.2394

```
columns_bb = {
    'Adj Close': 'Price',
    'Return': 'Buy-And-Hold',
    'Strategy': 'BB(20, 2)'
}
```

```
(
    btc_bb
    [['Return', 'Strategy']]
    .dropna()
    .rename(columns=columns_bb)
    .add(1)
    .cumprod()
    .plot()
)
plt.semilogy()
plt.suptitle('Comparison of One-Dollar Investments in BTC-USD Strategies')
plt.ylabel('Value of One-Dollar Investments')
plt.show()
```



Here are the final values of $1 investments, which are difficult to read on the log scale above.

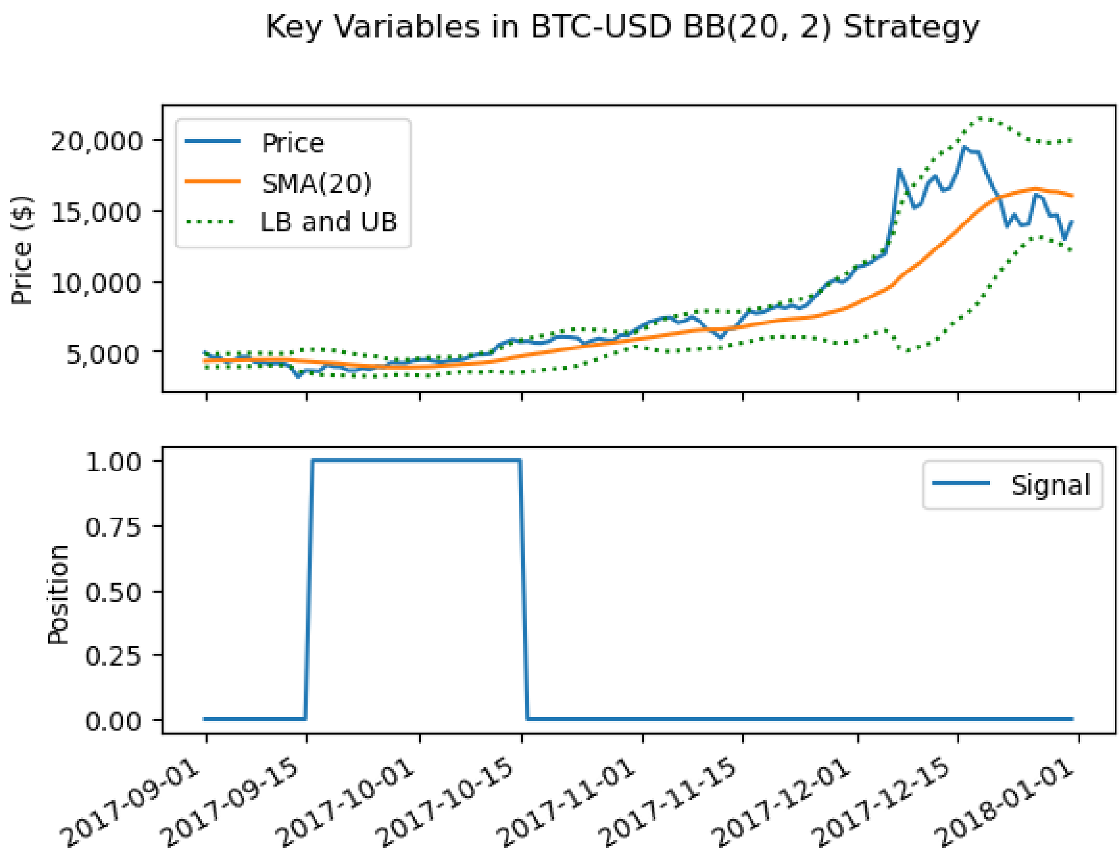
```
(
    btc_bb
    [['Return', 'Strategy']]
    .rename(columns=columns_bb)
    .add(1)
    .prod()
    .rename_axis('Strategy')
    .to_frame('Value of One-Dollar Investment')
)
```

	Value of One-Dollar Investment
Strategy	
Buy-And-Hold	183.6056
BB(20, 2)	2.1877

We need a more complex plot to better understand what is going on!

```
import matplotlib.ticker as ticker
```

```
fig, ax = plt.subplots(nrows=2, ncols=1, sharex=True)
df = btc_bb.loc['2017-09':'2017-12']

ax[0].plot(df[['Adj Close']], label='Price')
ax[0].plot(df[['SMA']], label='SMA(20)')
ax[0].plot(df[['UB']], label='LB and UB', color='green', linestyle=':')
ax[0].plot(df[['LB']], color='green', linestyle=':')
ax[0].legend()
ax[0].set_ylabel('Price ($)')

ax[1].plot(df[['Position']], label='Signal')
ax[1].legend()
ax[1].set_ylabel('Position')

ax[0].yaxis.set_major_formatter(ticker.StrMethodFormatter('{x:,.0f}'))
fig.autofmt_xdate()

plt.suptitle('Key Variables in BTC-USD BB(20, 2) Strategy')
plt.show()
```



Implement a long-short RSI(14) strategy with Bitcoin

From Fidelity:

> The Relative Strength Index (RSI), developed by J. Welles Wilder, is a momentum oscillator that measures the speed and change of price movements. The RSI oscillates between zero and 100. Traditionally the RSI is considered overbought when above 70 and oversold when below 30. Signals can be generated by looking for divergences and failure swings. RSI can also be used to identify the general trend.

The RSI formula: $RSI(n) = 100 - \frac{100}{1+RS(n)}$, where $RS(n) = \frac{SMA(U,n)}{SMA(D,n)}$. For "up days", $U = \Delta\text{Adj Close}$ and $D = 0$. For "down days", $U = 0$ and $D = -\Delta\text{Adj Close}$.

We will implement a long-short RSI(14) as follows:

1. Buy when the RSI crosses 30 from below, and sell when the RSI crosses 50 from below

2. Short when the RSI crosses 70 from above, and cover when the RSI crosses 50 from above

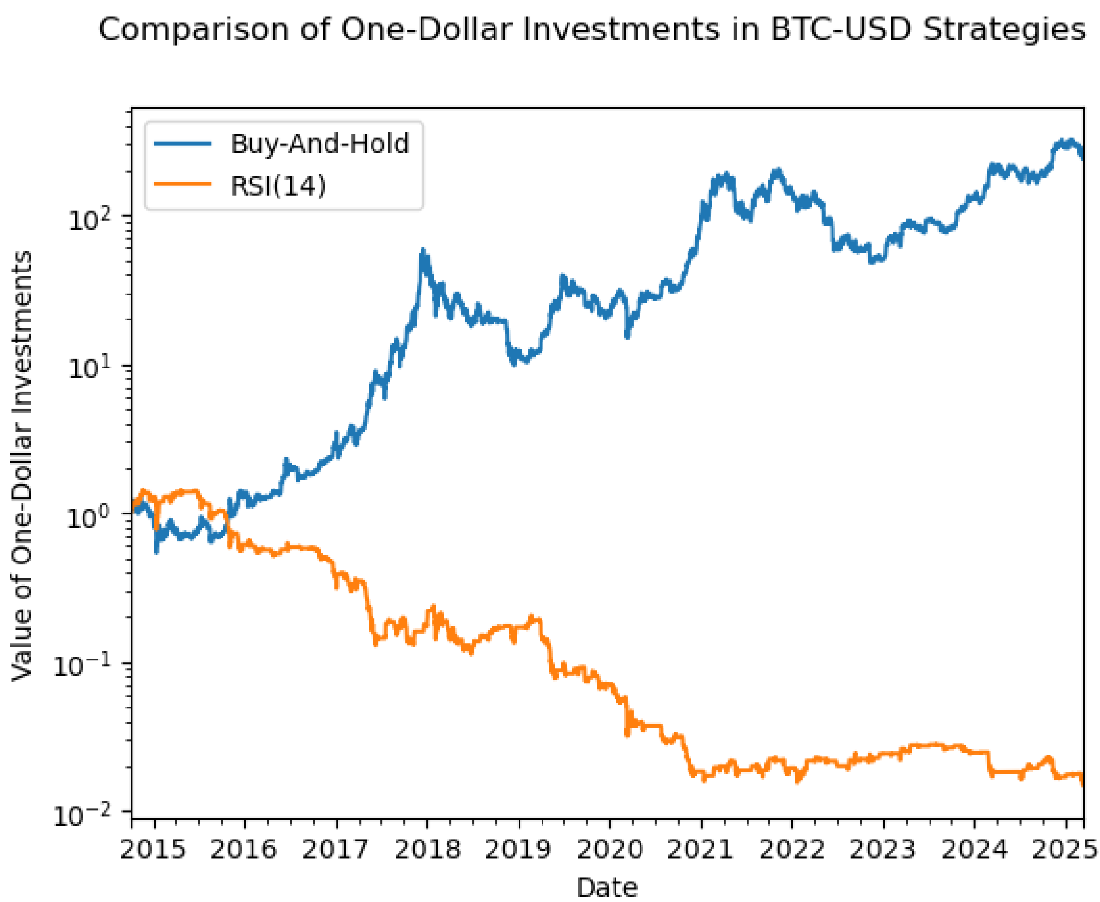
More about RSI here.

```
def calc_rsi(df, window=14, lo=30, mid=50, hi=70):
    return (
        df
        .assign(
            Return=lambda x: x['Adj Close'].pct_change(),
            # This approach with .max() and .min() handles NA values better than the in-class
            Diff=lambda x: x['Adj Close'].diff(),
            Zero=0,
            U=lambda x: x[['Diff', 'Zero']].max(axis=1, skipna=False),
            D=lambda x: -x[['Diff', 'Zero']].min(axis=1, skipna=False),
            SMAU=lambda x: x['U'].rolling(window=window).mean(),
            SMAD=lambda x: x['D'].rolling(window=window).mean(),
            RS=lambda x: x['SMAU'] / x['SMAD'],
            RSI=lambda x: 100 - 100 / (1 + x['RS']),
            Position_w_nan=lambda x: np.select(
                condlist=[
                    (x['RSI'].shift(1) > lo) & (x['RSI'].shift(2) <= lo),
                    (x['RSI'].shift(1) > mid) & (x['RSI'].shift(2) <= mid),
                    (x['RSI'].shift(1) < hi) & (x['RSI'].shift(2) >= hi),
                    (x['RSI'].shift(1) < mid) & (x['RSI'].shift(2) >= mid),
                ],
                choicelist=[1, 0, -1, 0],
                default=np.nan
            ),
            Position=lambda x: x['Position_w_nan'].ffill(),
            Strategy=lambda x: x['Position'] * x['Return']
        )
    )
```

```
btc_rsi = btc.pipe(calc_rsi)
```

```
columns_rsi = {
    'Adj Close': 'Price',
    'Return': 'Buy-And-Hold',
    'Strategy': 'RSI(14)'
}
```

```
(
    btc_rsi
    [['Return', 'Strategy']]
    .dropna()
    .rename(columns=columns_rsi)
    .add(1)
    .cumprod()
    .plot()
)
plt.semilogy()
plt.suptitle('Comparison of One-Dollar Investments in BTC-USD Strategies')
plt.ylabel('Value of One-Dollar Investments')
plt.show()
```



Implement a golden cross with Bitcoin

Someone in Section 04 mentioned two-moving average strategies, so I added this golden cross, where the 50-day SMA crosses the 200-day SMA, to every section.

From Grok:

> In technical analysis, a golden cross is a bullish chart pattern that occurs when a short-term moving average (typically the 50-day moving average) crosses above a long-term moving average (typically the 200-day moving average). This crossover is considered a signal that a stock, index, or other asset may be entering a sustained upward trend, suggesting potential buying opportunities for traders and investors.

More here.

```
def calc_cross(df, long=200, short=50):
    return (
        df
        .assign(
            Return=lambda x: x['Adj Close'].pct_change(),
            SMAL=lambda x: x['Adj Close'].rolling(window=long).mean(),
            SMAS=lambda x: x['Adj Close'].rolling(window=short).mean(),
            Position=lambda x: np.select(
                condlist=[
                    x['SMAS'].shift(1) > x['SMAL'].shift(1),
                    x['SMAS'].shift(1) <= x['SMAL'].shift(1)
                ],
                choicelist=[1, 0],
                default=np.nan
            ),
            Strategy=lambda x: x['Position'] * x['Return']
        )
    )
```

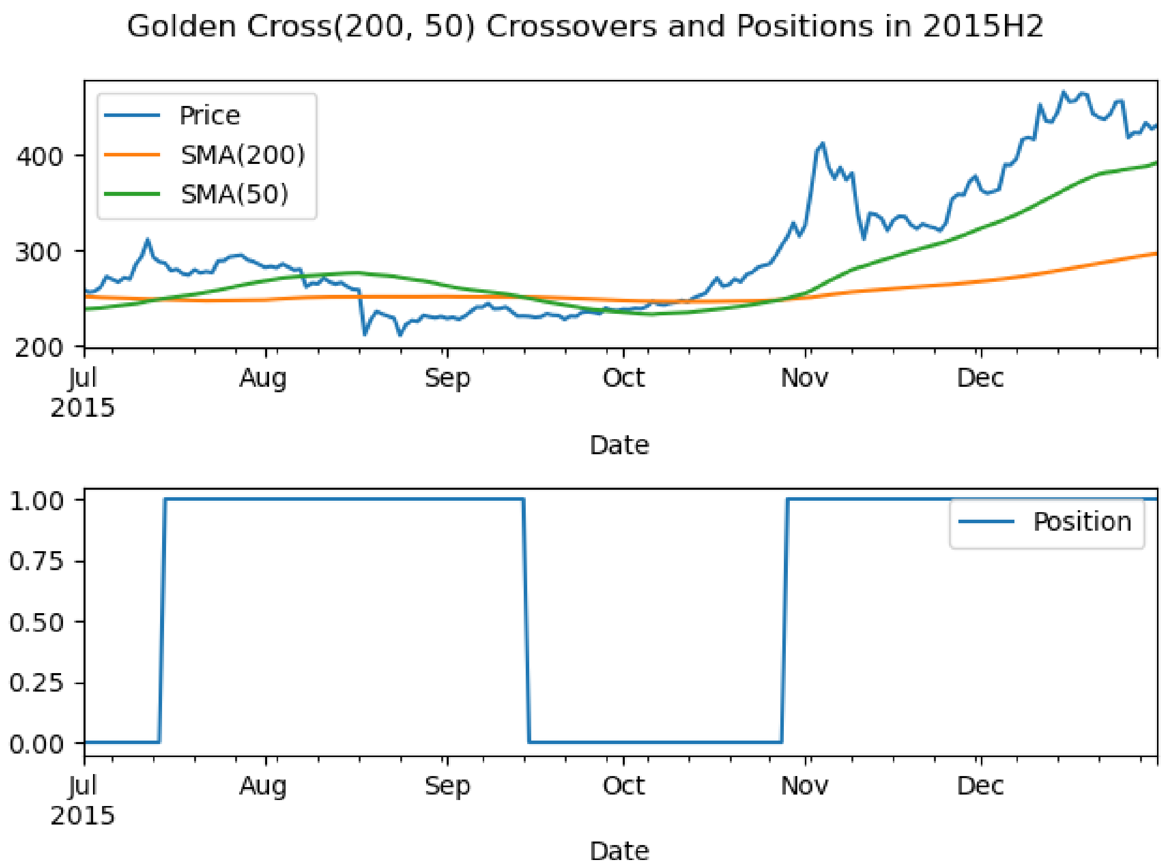
```
btc_cross = btc.pipe(calc_cross, long=200, short=50)
```

```
columns_cross = {
    'Adj Close': 'Price',
    'SMAL': 'SMA(200)',
    'SMAS': 'SMA(50)',
    'Return': 'Buy-And-Hold',
    'Strategy': 'Golden Cross(200, 50)'
}
```

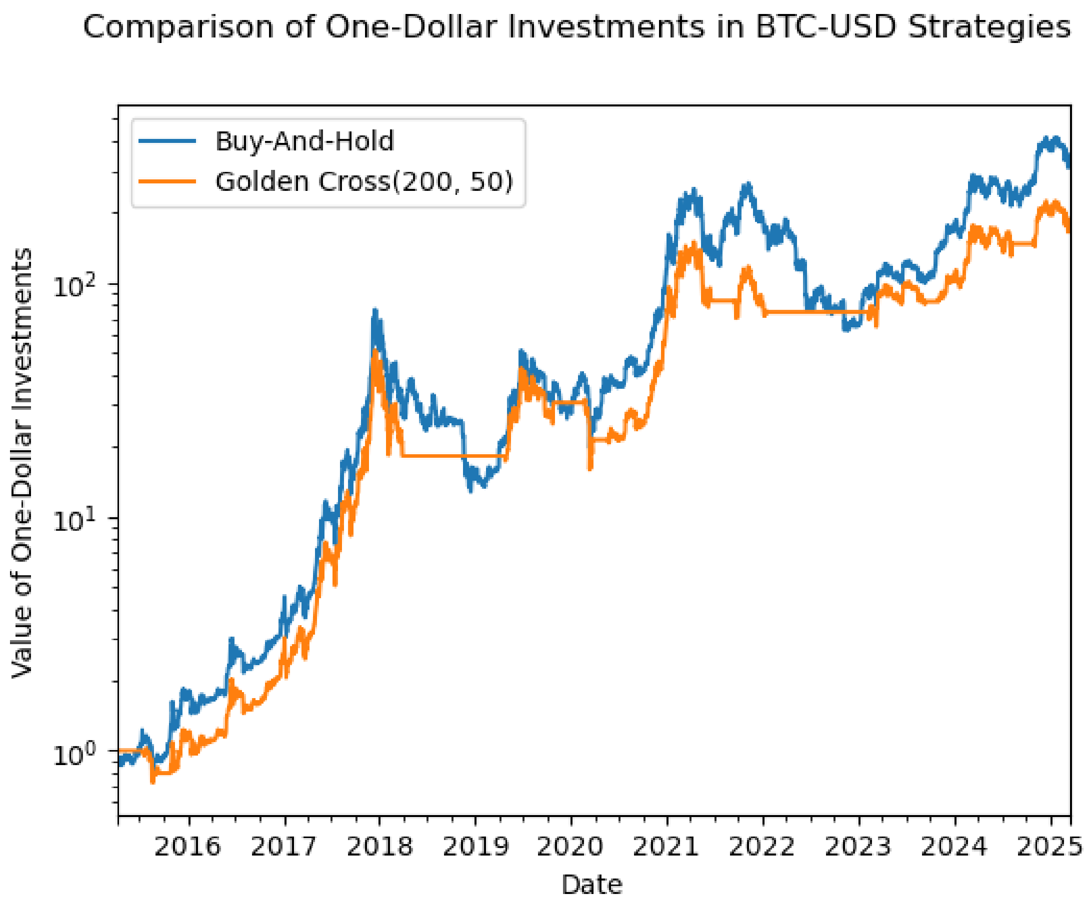
```
btc_cross.query('Position == 1')
```

	Adj Close	Close	High	Low	Open	Volume	Return	SMAI
Date								
2015-07-15	285.8290	285.8290	293.2480	285.3670	288.0450	27486600	-0.0057	247.78
2015-07-16	278.0890	278.0890	291.1830	275.2400	286.0420	49482600	-0.0271	247.58
2015-07-17	279.4720	279.4720	280.2800	272.0430	278.0910	27591400	0.0050	247.42
2015-07-18	274.9010	274.9010	282.5270	274.0750	279.3310	25187100	-0.0164	247.24
2015-07-19	273.6140	273.6140	275.6700	272.5130	274.7670	15332500	-0.0047	247.00
...	...	...	...	...	...	...	...	...
2025-03-10	78532.0000	78532.0000	83955.9297	77420.5938	80597.1484	54061099422	-0.0257	83357
2025-03-11	82862.2109	82862.2109	83577.7578	76624.2500	78523.8750	54702837196	0.0551	83451
2025-03-12	83722.3594	83722.3594	84358.5781	80635.2500	82857.3750	40353484454	0.0104	83549
2025-03-13	81066.7031	81066.7031	84301.6953	79931.8516	83724.9219	31412940153	-0.0317	83633
2025-03-14	83969.1016	83969.1016	85263.2891	80797.5625	81066.9922	29588112414	0.0358	83738

```
fig, ax = plt.subplots(2, 1)
df = btc_cross.loc['2015-07':'2015-12']
df[['Adj Close', 'SMAL', 'SMAS']].rename(columns=columns_cross).plot(ax=ax[0])
df[['Position']].plot(ax=ax[1])
plt.suptitle('Golden Cross(200, 50) Crossovers and Positions in 2015H2')
plt.tight_layout()
plt.show()
```




```
(
    btc_cross
    [['Return', 'Strategy']]
    .dropna()
    .rename(columns=columns_cross)
    .add(1)
    .cumprod()
    .plot()
)
plt.semilogy()
plt.suptitle('Comparison of One-Dollar Investments in BTC-USD Strategies')
plt.ylabel('Value of One-Dollar Investments')
plt.show()
```

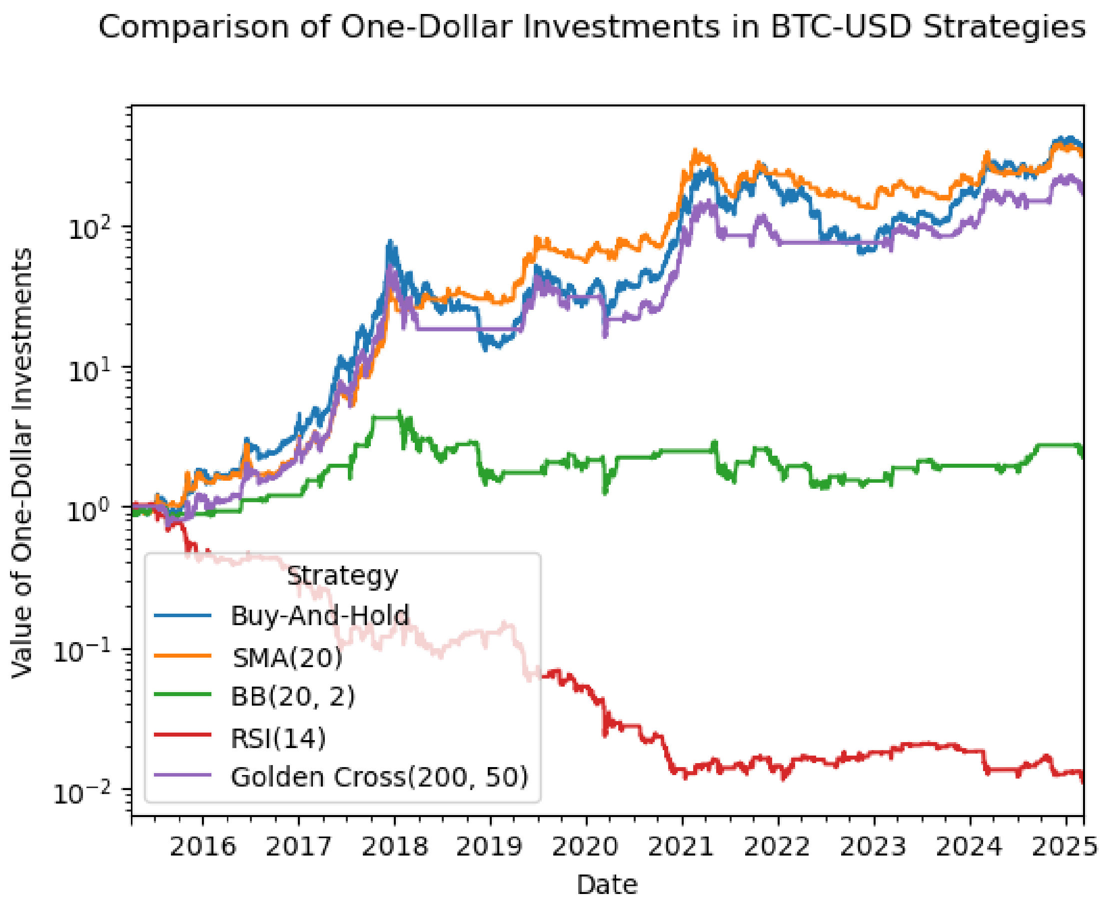


Compare all strategies

I added this comparison of all strategies after class.

```
df = (
    btc_sma[['Return', 'Strategy']]
    .join(btc_bb[['Strategy']], rsuffix='_bb')
    .join(btc_rsi[['Strategy']], rsuffix='_rsi')
    .join(btc_cross[['Strategy']], rsuffix='_cross')
    .dropna()
)


(
    df
```

```
    .add(1)
    .cumprod()
    .rename_axis(columns='Strategy')
    .rename(columns={
        'Return': 'Buy-And-Hold',
        'Strategy': 'SMA(20)',
        'Strategy_bb': 'BB(20, 2)',
        'Strategy_rsi': 'RSI(14)',
        'Strategy_cross': 'Golden Cross(200, 50)',
    })
    .plot()
)

plt.semilogy()
plt.suptitle('Comparison of One-Dollar Investments in BTC-USD Strategies')
plt.ylabel('Value of One-Dollar Investments')
plt.show()
```


Comparison of One-Dollar Investments in BTC-USD Strategies
Value of One-Dollar Investments
10^2
10^1
10^0
10^{-1}
10^{-2}
Strategy
Buy-And-Hold
SMA(20)
BB(20, 2)
RSI(14)
Golden Cross(200, 50)
2016
2017
2018
2019
2020
2021
2022
2023
2024
2025
Date

Herron Topic 2 - Practice - Sec 03

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import pandas_datareader as pdr
import seaborn as sns
import statsmodels.api as sm
import yfinance as yf
```

```
%precision 4
pd.options.display.float_format = '{:.4f}'.format
# %config InlineBackend.figure_format = 'retina'
```

Announcements

1. I am still grading your projects; I hope to finish them by next Tuesday
2. I posted 50 practice problems to prepare for the end-of-course Programming Assessment here: https://northeastern.instructure.com/courses/207607/discussion_topics/2727917
    1. I built 5 autograded notebooks to help you prepare for the assessment, and I will build more in the coming weeks
    2. To run these notebooks, install the `otter-grader` package: In the Anaconda command prompt (or Terminal on Mac) run `conda activate fina6333` then `conda install otter-grader`
    3. See the video at the link above for details

Five-Minute Review

1. Technical analysis (TA) is a method of evaluating trends in trading prices and volume (and open interest in the futures and options markets only)
2. The three tenants of TA are:
    1. Markets discount everything

 2. Prices move in trends
 3. History repeats itself, so these trends are recurring

3. Academics have not found much evidence that TA generate profits that exceed transaction costs, but we will spend a week on it for three reasons:
   1. You asked for it, and it will help build our data analytics skills
   2. It is a small part of the Chartered Financial Analyst (CFA) curriculum
   3. That TA still receives attention 60 years after the Efficient Markets Hypothesis (EMH) suggests that it has some value that academics have been unable to measure

Practice

Implement the SMA(20) strategy with BTC-USD from the lecture notebook

Try to create the `btc_sma` data frame from the `btc` data frame in one code cell with one assignment (i.e., one `=`).

```
btc = (
    yf.download(
        tickers='BTC-USD',
        auto_adjust=False,
        progress=False,
        multi_level_index=False
    )
    .iloc[:-1] # drop incomplete trading day
)
```

After class, I converted our code to a function. We might want to test different moving average parameters, and a function makes these tests easier.

The following `calc_sma()` function accepts:

1. A data frame `df` of daily values from `yfinance.download()`
2. An integer `window` that specifies the number of trading days in the SMA window

And returns the original data frame `df` plus the following columns:

1. `Return` with daily returns
2. `SMA` for the `window`-trading-day moving average
3. `Position` for the weight on the security each day
4. `Strategy` for the return on the strategy each day

```
def calc_sma(df, window=20):
    return (
        df
        .assign(
            Return=lambda x: x['Adj Close'].pct_change(),
            SMA=lambda x: x['Adj Close'].rolling(window=window).mean(),
            Position=lambda x: np.select(
                condlist=[
                    x['Adj Close'].shift(1) > x['SMA'].shift(1),
                    x['Adj Close'].shift(1) <= x['SMA'].shift(1)
                ],
                choicelist=[1, 0],
                default=np.nan
            ),
            Strategy=lambda x: x['Position'] * x['Return']
        )
    )
```

```
btc_sma = btc.pipe(calc_sma, window=20)
```

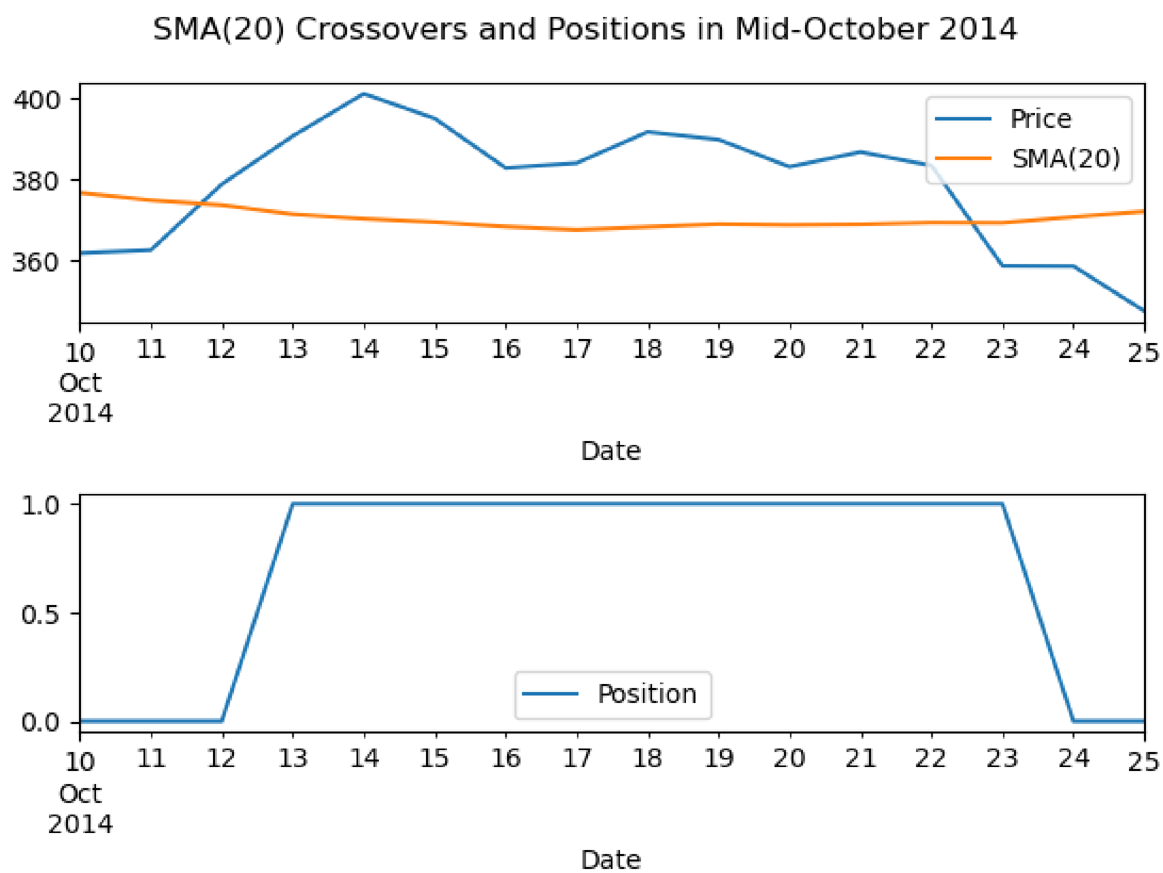
It can be helpful to visualize a few places where the price (here `Adj Close`) crosses the moving average (here SMA(20)). In class, we found that there are a few changes in position in the middle of October, 2014. The following code create two subplots in one figure, then puts one plot in each of the subplots.

For the first crossover on October 12, we *buy* at the close on October 12 and earn the security return on October 13. For the second crossover on October 23, we *sell* at the close on October 23 and earn zero return on October 24.

```
columns_sma20 = {
    'Adj Close': 'Price',
    'SMA': 'SMA(20)',
    'Return': 'Buy-And-Hold',
    'Strategy': 'SMA(20)'
}
```

```
fig, ax = plt.subplots(2, 1)
df = btc_sma.loc['2014-10-10':'2014-10-25']
df[['Adj Close', 'SMA']].rename(columns=columns_sma20).plot(ax=ax[0])
df[['Position']].plot(ax=ax[1])
plt.suptitle('SMA(20) Crossovers and Positions in Mid-October 2014')
```

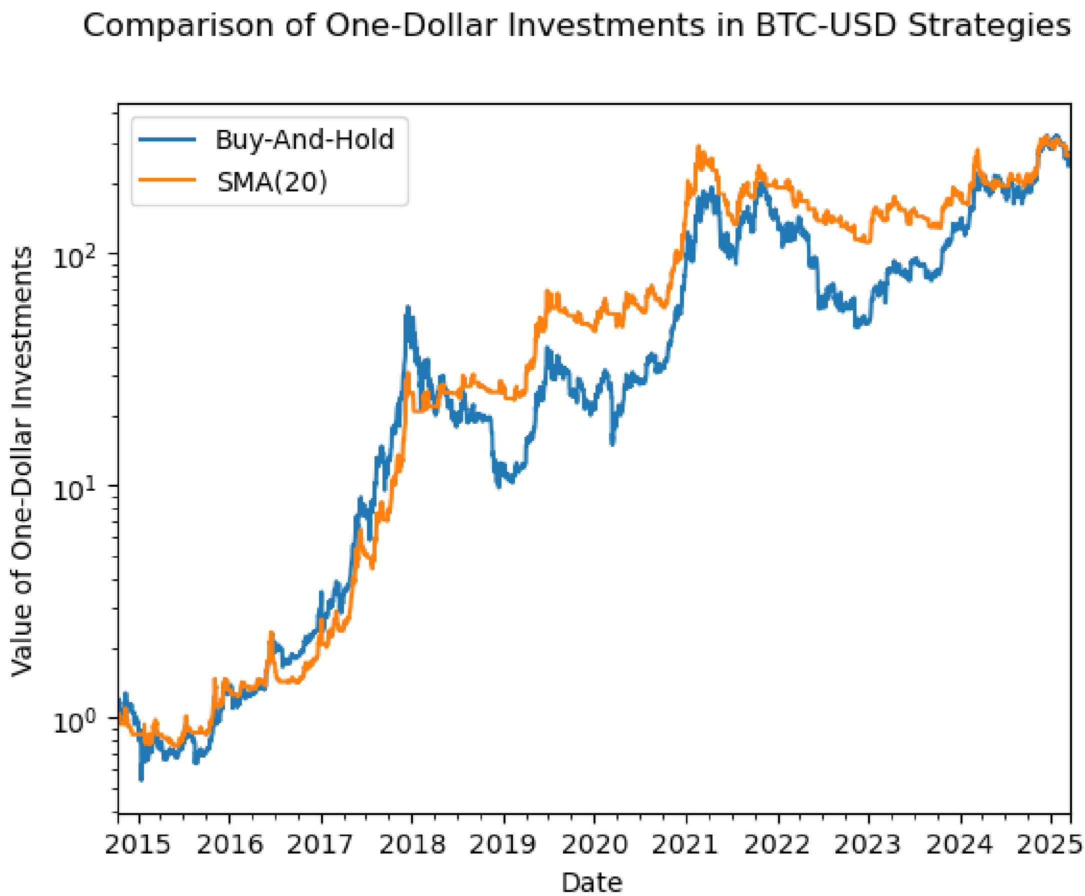
```
plt.tight_layout()
plt.show()
```



We can compare the total returns on BTC-USD and our SMA(20) strategy! We `.dropna()` first because the SMA(20) strategy needs 20 days of data to make its first investing decision.

```
(
    btc_sma
    [['Return', 'Strategy']]
    .dropna()
    .rename(columns=columns_sma20)
    .add(1)
    .cumprod()
    .plot()
)
plt.semilogy()
```

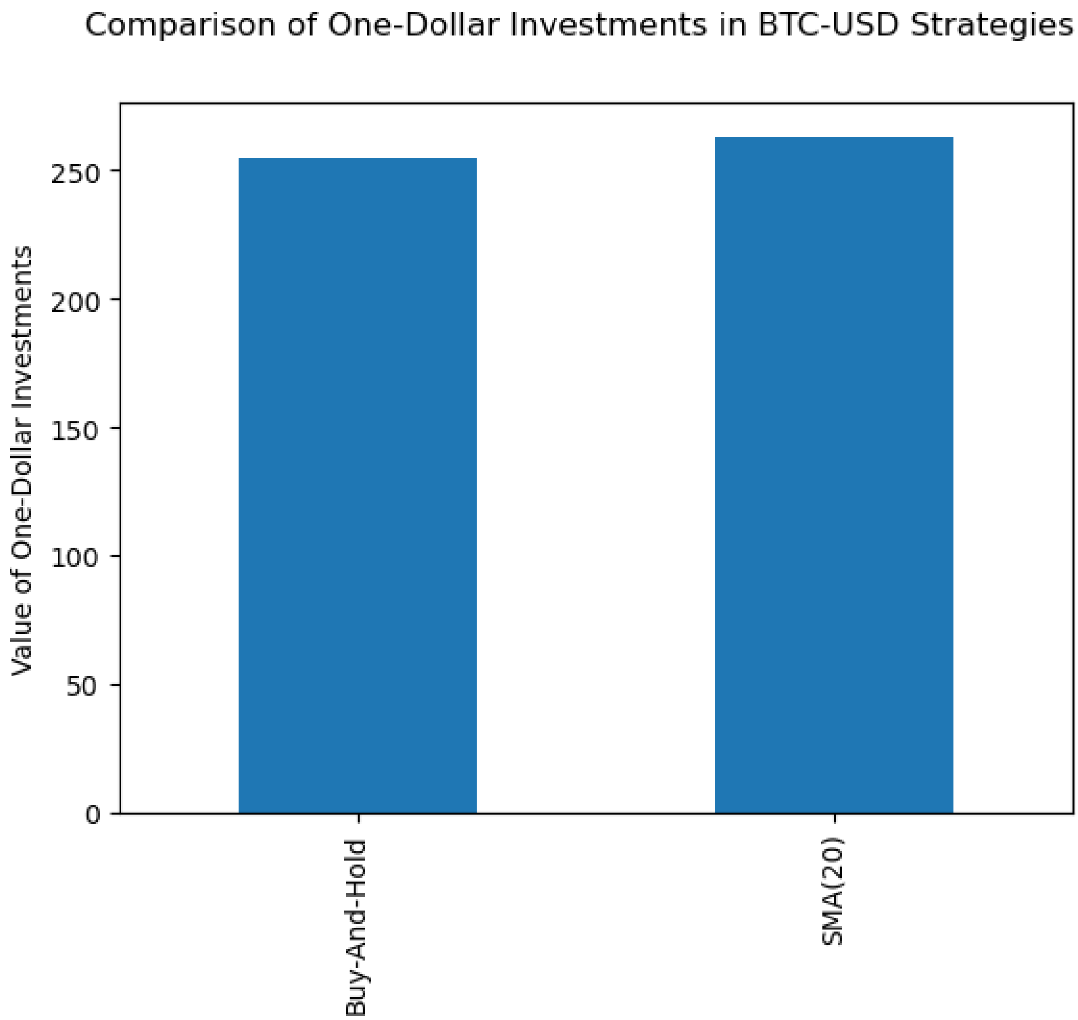
```
plt.suptitle('Comparison of One-Dollar Investments in BTC-USD Strategies')
plt.ylabel('Value of One-Dollar Investments')
plt.show()
```



These total returns, at least today, are similar!

```
(
    btc_sma
    [['Return', 'Strategy']]
    .dropna()
    .rename(columns=columns_sma20)
    .add(1)
    .prod()
    .plot(kind='bar')
)
plt.suptitle('Comparison of One-Dollar Investments in BTC-USD Strategies')
```

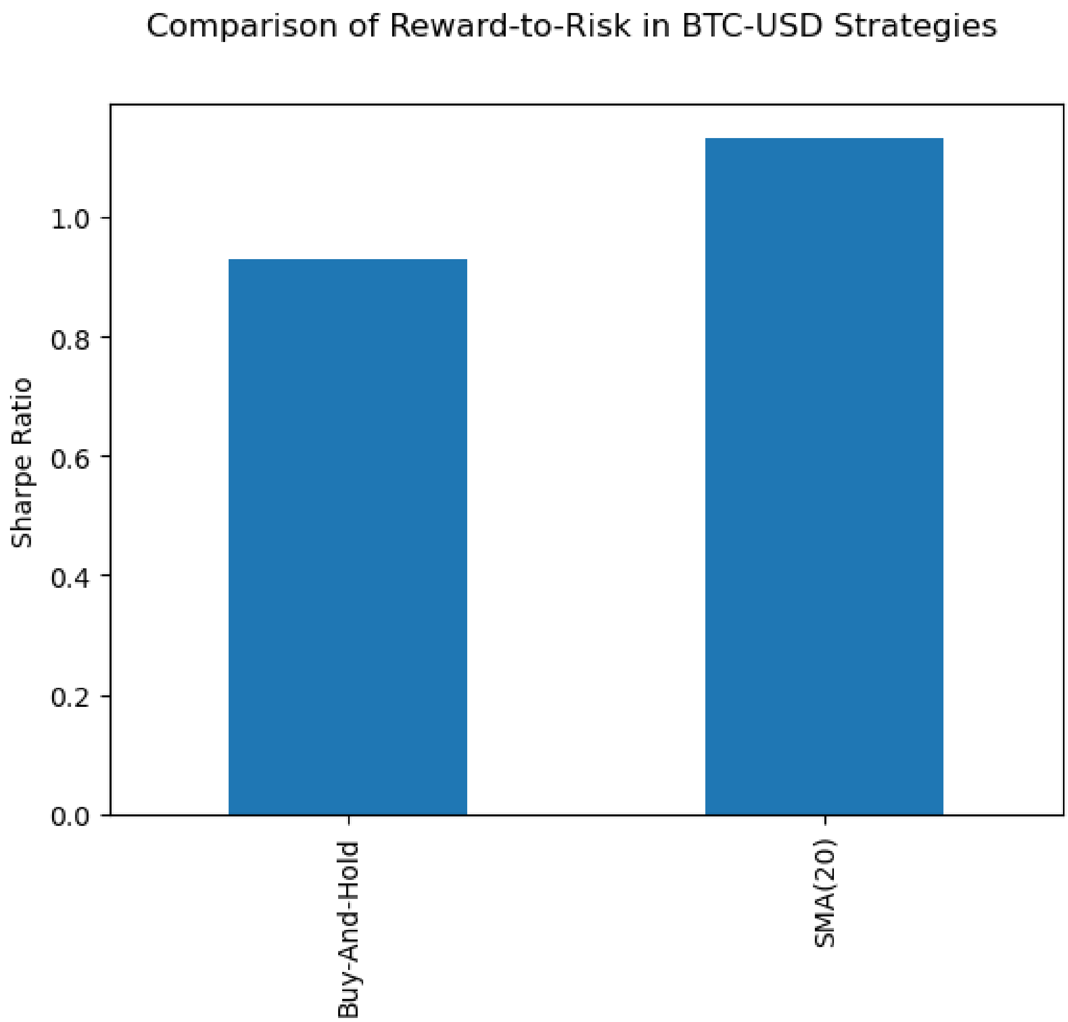
```
plt.ylabel('Value of One-Dollar Investments')
plt.show()
```



However, the Sharpe ratio of the SMA(20) is higher, because time out of the market when `Position=0` reduces risk. For simplicity, we will ignore the risk-free rate in the Sharpe ratio formula.

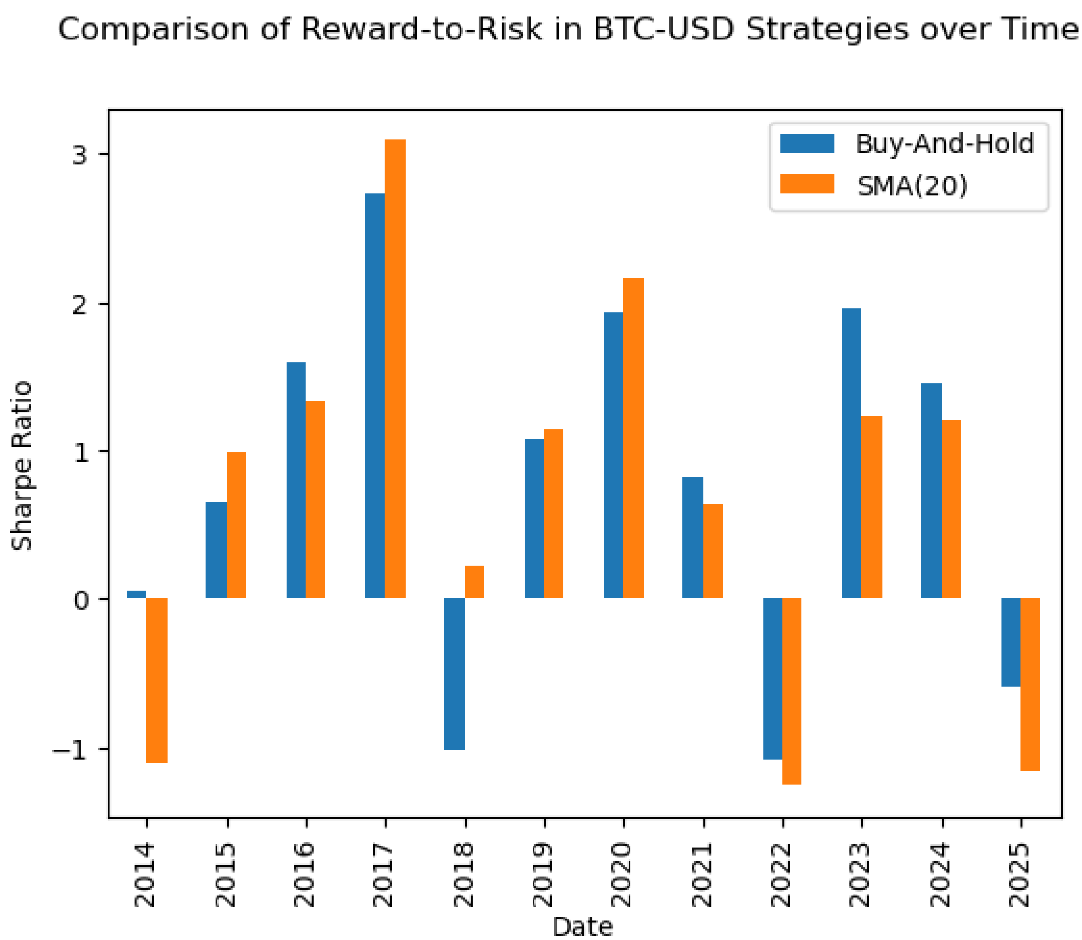
```
(
    btc_sma
    [['Return', 'Strategy']]
    .dropna()
    .rename(columns=columns_sma20)
```

```
    .apply(lambda x: np.sqrt(252) * x.mean() / x.std())
    .plot(kind='bar')
)
plt.suptitle('Comparison of Reward-to-Risk in BTC-USD Strategies')
plt.ylabel('Sharpe Ratio')
plt.show()
```



But how persistent is this outperformance in terms of reward-to-risk ratios? We can quickly modify the code above to calculate Sharpe ratios each year! We find that SMA(20) Sharpe ratio edge is not persistent.

```
df = (
    btc_sma
    [['Return', 'Strategy']]
    .dropna()
    .rename(columns=columns_sma20)
    .resample('YE')
    .apply(lambda x: np.sqrt(252) * x.mean() / x.std())
)
df.index = df.index.year
df.plot(kind='bar')
plt.suptitle('Comparison of Reward-to-Risk in BTC-USD Strategies over Time')
plt.ylabel('Sharpe Ratio')
plt.show()
```



We can use our list comprehension skills to easily try several window sizes!

```
def CAGR(x):
    T = x.count()
    return (x.add(1).prod() ** (252 / T)) - 1
```

```
def Sharpe(x, ann_fac=np.sqrt(252)):
    return ann_fac * x.mean() / x.std()
```

```
columns_sman = {
    'Adj Close': 'Price',
    'SMA': 'SMA(N)',
    'Return': 'Buy-And-Hold',
    'Strategy': 'SMA(N)'
}
```

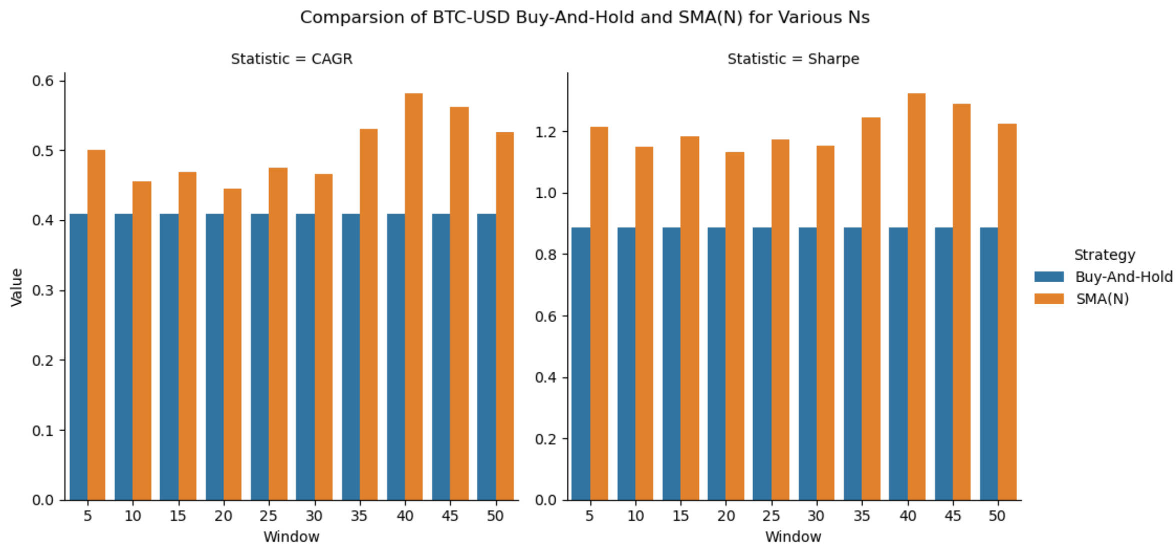
```
windows = list(range(5, 55, 5))

btc_smas = (
    pd.concat(
        objs=[
            btc.pipe(calc_sma, window=w)[['Return', 'Strategy']].agg([CAGR, Sharpe])
            for w in windows
        ],
        keys=windows,
        names=['Window', 'Statistic']
    )
    .rename(columns=columns_sman)
)
```

```
btc_smas
```

Window	Statistic	Buy-
5	CAGR	0.40
	Sharpe	0.88
10	CAGR	0.40
	Sharpe	0.88

Window	Statistic	Buy-
15	CAGR	0.40
	Sharpe	0.88
20	CAGR	0.40
	Sharpe	0.88
25	CAGR	0.40
	Sharpe	0.88
30	CAGR	0.40
	Sharpe	0.88
35	CAGR	0.40
	Sharpe	0.88
40	CAGR	0.40
	Sharpe	0.88
45	CAGR	0.40
	Sharpe	0.88
50	CAGR	0.40
	Sharpe	0.88

```
(
    btc_smas
    .reset_index()
    .melt(
        id_vars=['Window', 'Statistic'],
        value_vars=['Buy-And-Hold', 'SMA(N)'],
        var_name='Strategy',
        value_name='Value'
    )
    .pipe(
        sns.catplot,
        x='Window',
        col='Statistic',
        hue='Strategy',
        y='Value',
        kind='bar',
        sharey=False
    )
)
plt.suptitle('Comparsion of BTC-USD Buy-And-Hold and SMA(N) for Various Ns', y=1.05)
plt.show()
```



Investigate how SMA(20) generates returns

Consider the following:

1. Does SMA(20) avoid the worst performing days? How many of the worst 20 days does SMA(20) avoid? Try the `.nlargest()` method.
2. Does SMA(20) preferentially avoid low-return days? Try to combine the `.groupby()` method and `pd.qcut()` function.
3. Does SMA(20) preferentially avoid high-volatility days? Try to combine the `.groupby()` method and `pd.qcut()` function.

The SMA(20) does well here because it avoids 17 of the 20 worst days, without avoiding the best days.

```
btc_sma.loc[btc_sma['Return'].nsmallest(20).index, ['Position']].value_counts()
```

```
Position
0.0000      17
1.0000       3
Name: count, dtype: int64
```

```
btc_sma.loc[btc_sma['Return'].nlargest(20).index, ['Position']].value_counts()
```

```
Position
0.0000      10
1.0000      10
Name: count, dtype: int64
```

We can also look at the descriptive statistics by `Postiion` and strategy. Two observations:

1. The `min` column shows that SMA(20) misses the bad days, like we see above
2. The `mean` in `Position=1` is about 5 times higher than in `Position=0`

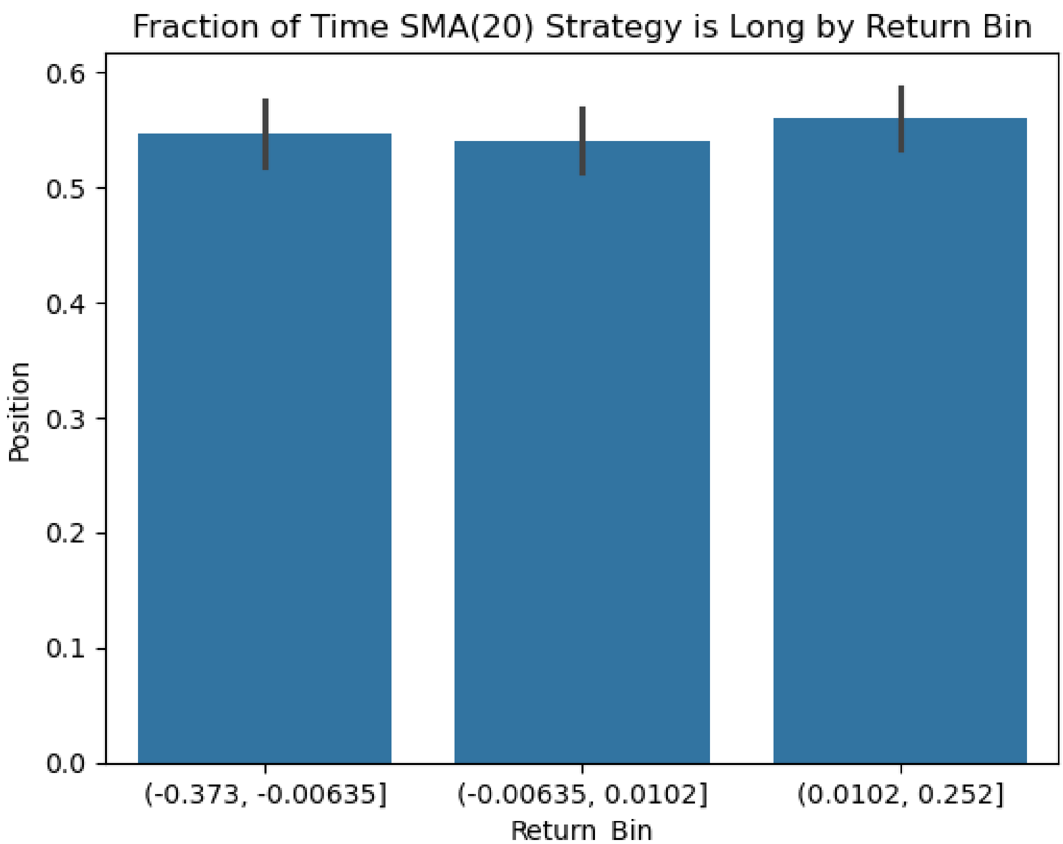
```
(
    btc_sma
    [['Position', 'Return', 'Strategy']]
    .dropna()
    .groupby('Position')
    .describe()
    .rename(columns=columns_sma20)
    .rename_axis(columns=['Strategy', 'Statistic'])
    .stack('Strategy', future_stack=True)
)
```

Position	Statistic Strategy
0.0000	Buy-And-Hold SMA(20)
1.0000	Buy-And-Hold SMA(20)

We can also use the seaborn package to visualize `Position` (i.e., the portfolio weight on Bitcoin) during periods of high and low Bitcoin returns and volatility. The SMA(20) strategy is long Bitcoin about 55% of the time, whether Bitcoin returns are high (bin 2) or low (bin 1).

```
(
    btc_sma
    [['Return', 'Position']]
    .dropna()
    .assign(Return_Bin=lambda x: pd.qcut(x['Return'], q=3))
```

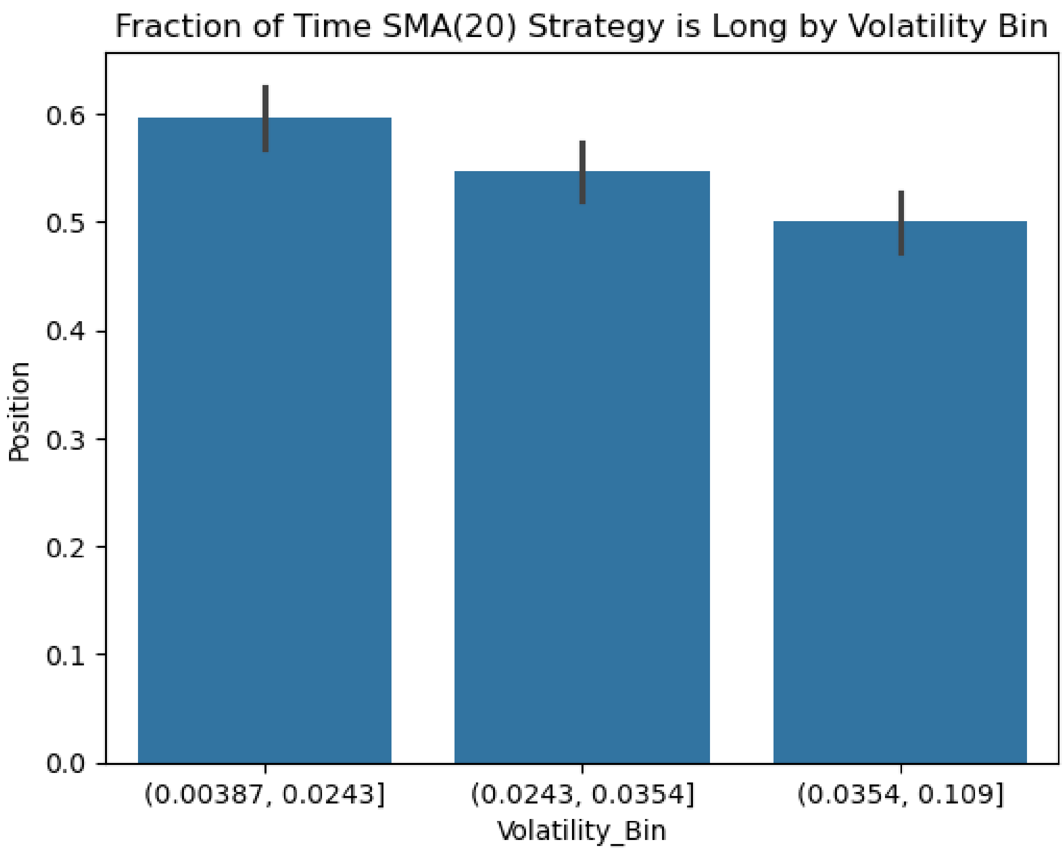
```
    .pipe(
        sns.barplot,
        x='Return_Bin',
        y='Position'
    )
)

plt.title('Fraction of Time SMA(20) Strategy is Long by Return Bin')
plt.show()
```



However, the SMA(20) stateegy, *for this security, sample, and window,* spends less times in Bitcoind during volatilit times.

```
(
    btc_sma
    [['Return', 'Position']]
    .dropna()
```

```
    .assign(Volatility_Bin=lambda x: pd.qcut(x['Return'].rolling(20).std(), q=3))
    .pipe(
        sns.barplot,
        x='Volatility_Bin',
        y='Position'
    )
)

plt.title('Fraction of Time SMA(20) Strategy is Long by Volatility Bin')
plt.show()
```



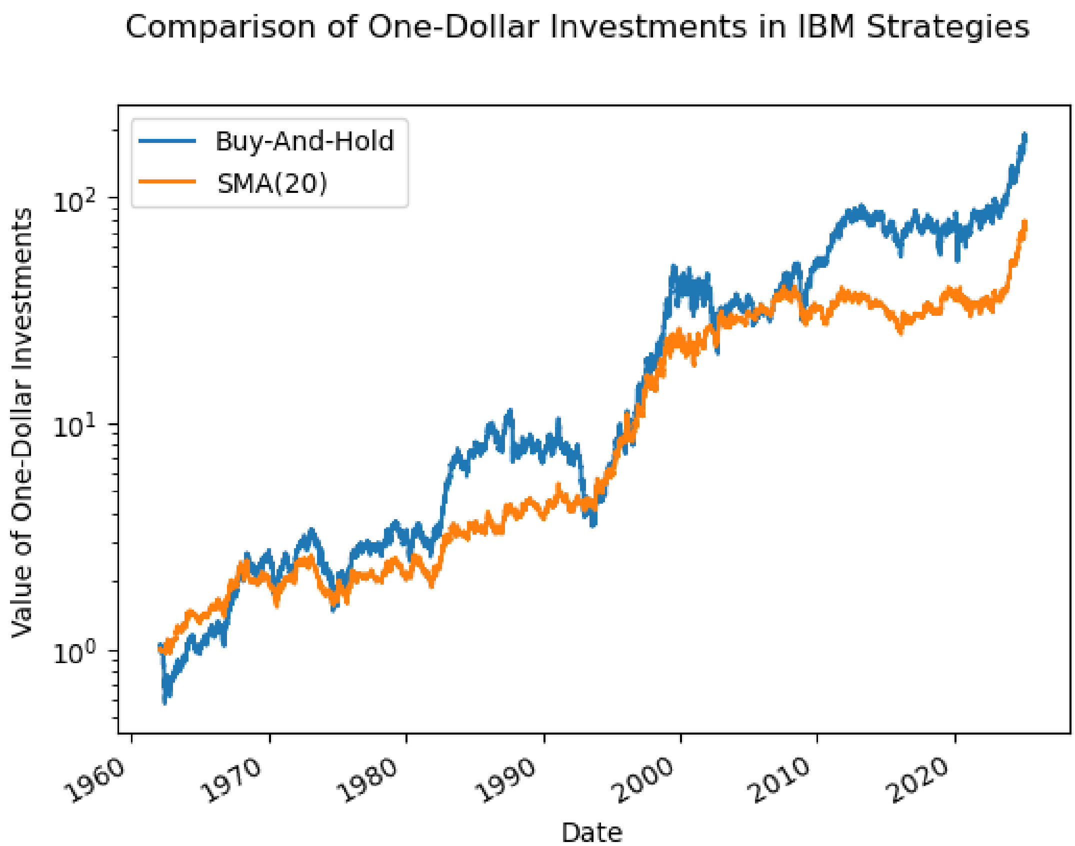
Implement the SMA(20) strategy with IBM

How often does SMA(20) outperform buy-and-hold with 10-year rolling windows?

```
ibm = (
    yf.download(
        tickers='IBM',
        auto_adjust=False,
        progress=False,
        multi_level_index=False
    )
    .iloc[:-1] # drop incomplete trading day
)
```

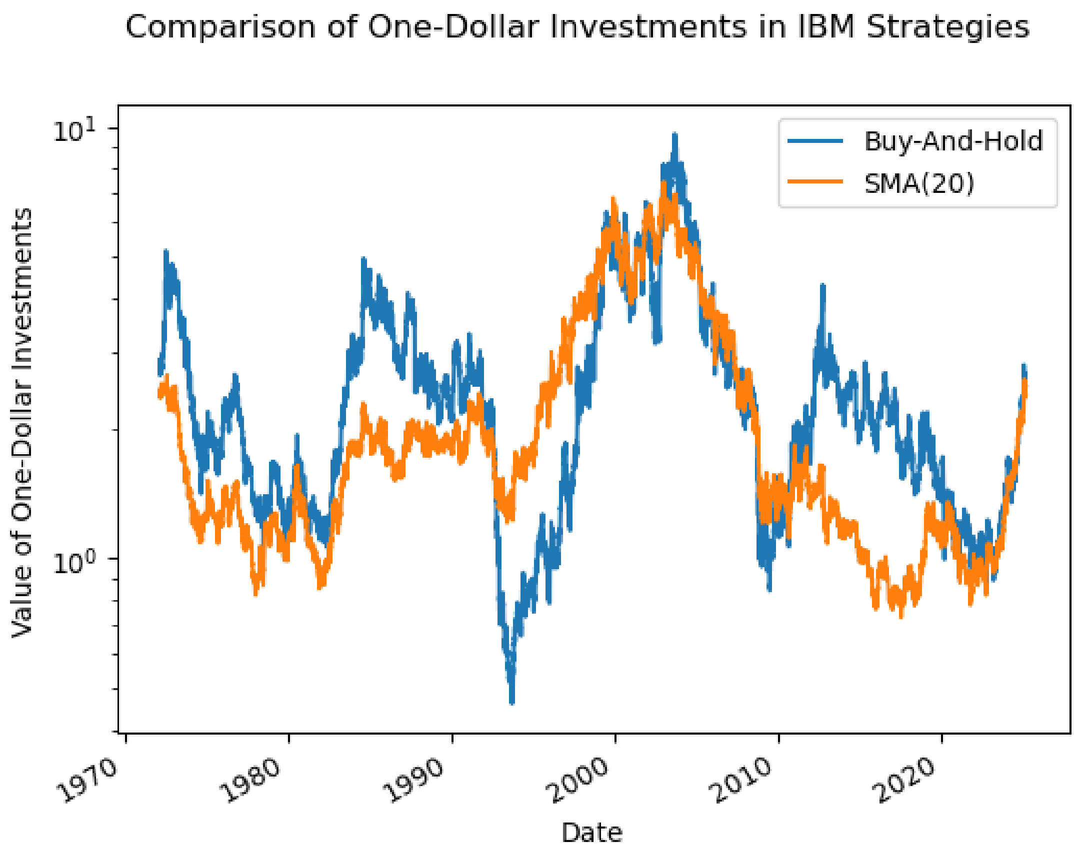
```
ibm_sma = ibm.pipe(calc_sma, window=20)
```

```
(
    ibm_sma
    [['Return', 'Strategy']]
    .dropna()
    .rename(columns=columns_sma20)
    .add(1)
    .cumprod()
    .plot()
)
plt.semilogy()
plt.suptitle('Comparison of One-Dollar Investments in IBM Strategies')
plt.ylabel('Value of One-Dollar Investments')
plt.show()
```



Over the full, 60-year sample, SMA(20) underperforms buy-and-hold. What about on rolling ten-year windows? The results look balanced, and neither clearly outperforms.

```
(
    ibm_sma
    [['Return', 'Strategy']]
    .dropna()
    .rename(columns=columns_sma20)
    .pipe(np.log1p)
    .rolling(window=10*252)
    .sum()
    .pipe(np.exp)
    .plot()
)
plt.semilogy()
plt.suptitle('Comparison of One-Dollar Investments in IBM Strategies')
plt.ylabel('Value of One-Dollar Investments')
plt.show()
```



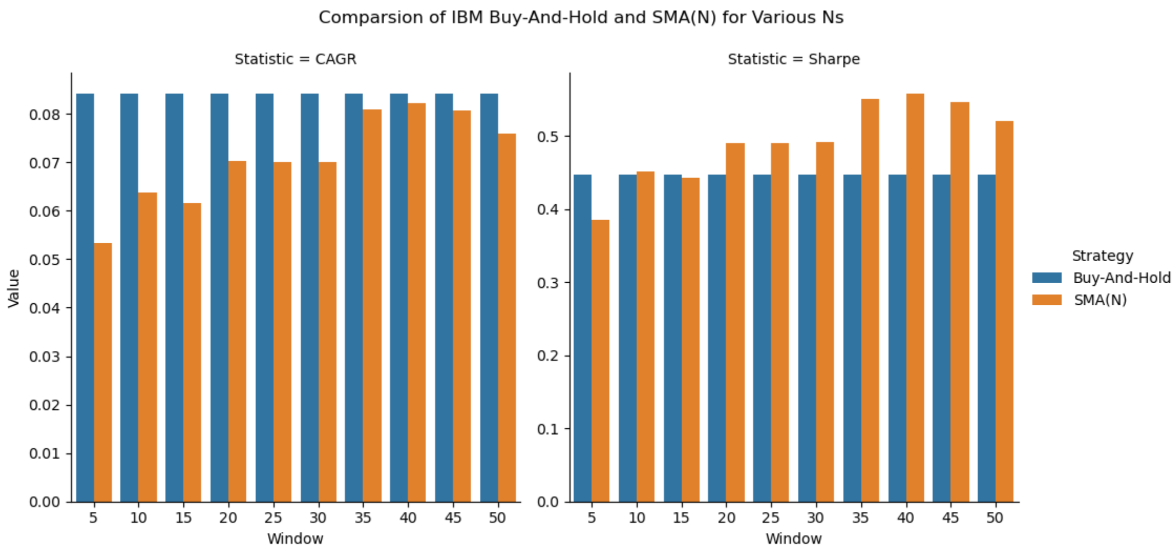
We can quantify how often SMA(20) outperform buy-and-hold. SMA(20) outperforms only 24% of the time!

```
(
    ibm_sma
    [['Return', 'Strategy']]
    .dropna()
    .pipe(np.log1p)
    .rolling(window=10*252)
    .sum()
    .pipe(np.exp)
    .pipe(lambda x: x['Strategy'] > x['Return'])
    .mean()
)
```

```
0.2315
```

```
windows = list(range(5, 55, 5))

ibm_smas = (
    pd.concat(
        objs=[
            ibm.pipe(calc_sma, window=w)[['Return', 'Strategy']].agg([CAGR, Sharpe])
            for w in windows
        ],
        keys=windows,
        names=['Window', 'Statistic']
    )
    .rename(columns=columns_sman)
)
```

```
(
    ibm_smas
    .reset_index()
    .melt(
        id_vars=['Window', 'Statistic'],
        value_vars=['Buy-And-Hold', 'SMA(N)'],
        var_name='Strategy',
        value_name='Value'
    )
    .pipe(
        sns.catplot,
        x='Window',
        col='Statistic',
        hue='Strategy',
        y='Value',
        kind='bar',
        sharey=False
    )
)
plt.suptitle('Comparsion of IBM Buy-And-Hold and SMA(N) for Various Ns', y=1.05)
plt.show()
```



Implement a long-only BB(20, 2) strategy with Bitcoin

Bollinger Bands are bands around a trend, typically defined in terms of simple moving averages and volatilities. A long-only BB(20, 2) strategy has upper and lower bands at 2 standard deviations above and below the SMA(20). It invests as follows:

1. Buy when the closing price crosses LB(20) from below
2. Sell when the closing price crosses UB(20) from above
3. No short-selling

The long-only BB(20, 2) is more difficult to implement than the long-only SMA(20) because we need to track buys and sells. For example, if the closing price is between LB(20) and BB(20), we need to know if our last trade was a buy or a sell. Further, if the closing price is below LB(20), we can still be long because we sell when the closing price crosses UB(20) from above.

More on Bollinger Bands here and here.

```
def calc_bb(df, m=20, n=2):
    return (
        df
        .assign(
            Return=lambda x: x['Adj Close'].pct_change(),
            SMA=lambda x: x['Adj Close'].rolling(window=m).mean(),
            SMV=lambda x: x['Adj Close'].rolling(window=m).std(),
            UB=lambda x: x['SMA'] + n*x['SMV'],
```

```
            LB=lambda x: x['SMA'] - n*x['SMV'],
            Position_w_nan=lambda x: np.select(
                condlist=[
                    (x['Adj Close'].shift(1) > x['LB'].shift(1)) & (x['Adj Close'].shift(2) <
                    (x['Adj Close'].shift(1) < x['UB'].shift(1)) & (x['Adj Close'].shift(2) >
                ],
                choicelist=[1, 0],
                default=np.nan
            ),
            Position=lambda x: x['Position_w_nan'].ffill(),
            Strategy=lambda x: x['Position'] * x['Return']
        )
    )
```

```
btc_bb = btc.pipe(calc_bb)
```

The BB(20, 2) only spends 40% of its time long BTC!

```
btc_bb['Position'].mean()
```

```
0.4013
```

And BTC-USD performance is worse when it long that when its neutral!

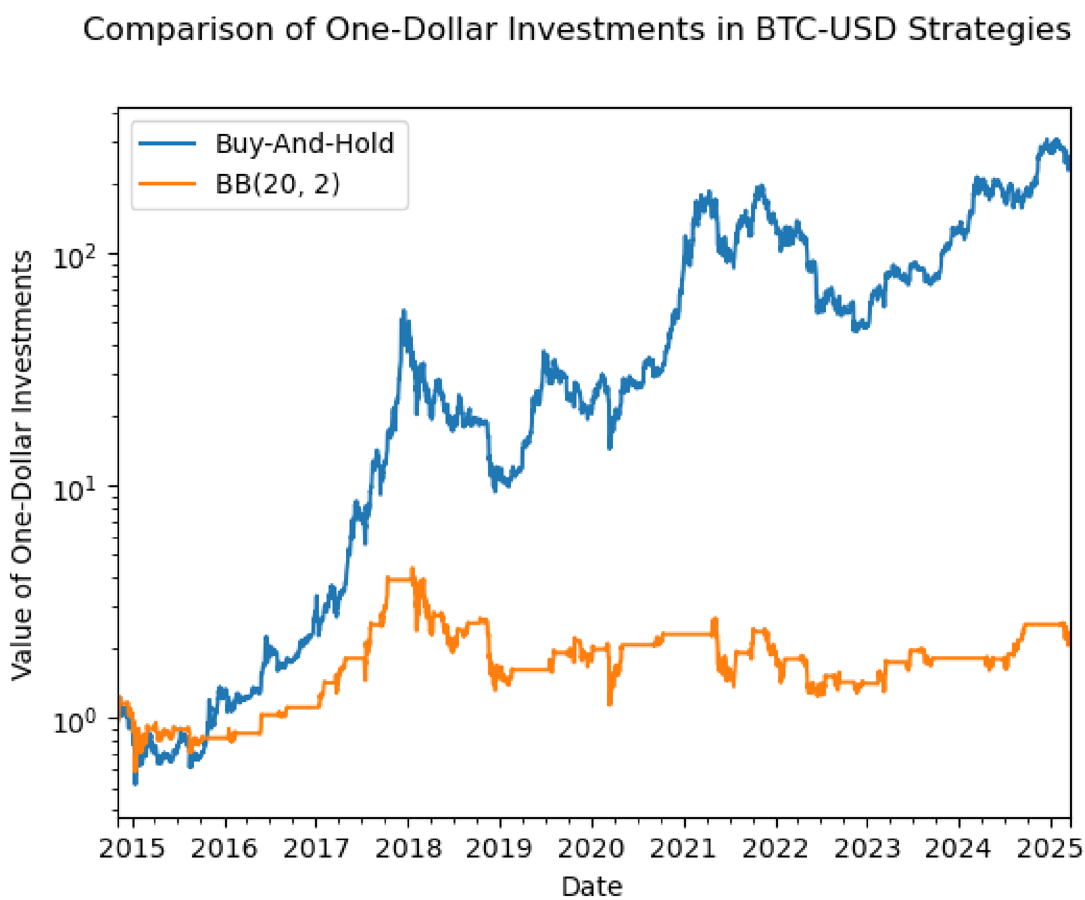
- Mean daily return is lower
- Volatility of daily returns is higher
- Every percentile of the distribution is worse

```
btc_bb.groupby('Position')['Return'].describe()
```

	count	mean	std	min	25%	50%	75%	max
Position								
0.0000	2268.0000	0.0027	0.0343	-0.1874	-0.0110	0.0019	0.0172	0.2525
1.0000	1520.0000	0.0013	0.0386	-0.3717	-0.0149	0.0007	0.0161	0.2394

```
columns_bb = {
    'Adj Close': 'Price',
    'Return': 'Buy-And-Hold',
    'Strategy': 'BB(20, 2)'
}
```

```
(
    btc_bb
    [['Return', 'Strategy']]
    .dropna()
    .rename(columns=columns_bb)
    .add(1)
    .cumprod()
    .plot()
)
plt.semilogy()
plt.suptitle('Comparison of One-Dollar Investments in BTC-USD Strategies')
plt.ylabel('Value of One-Dollar Investments')
plt.show()
```



Here are the final values of $1 investments, which are difficult to read on the log scale above.

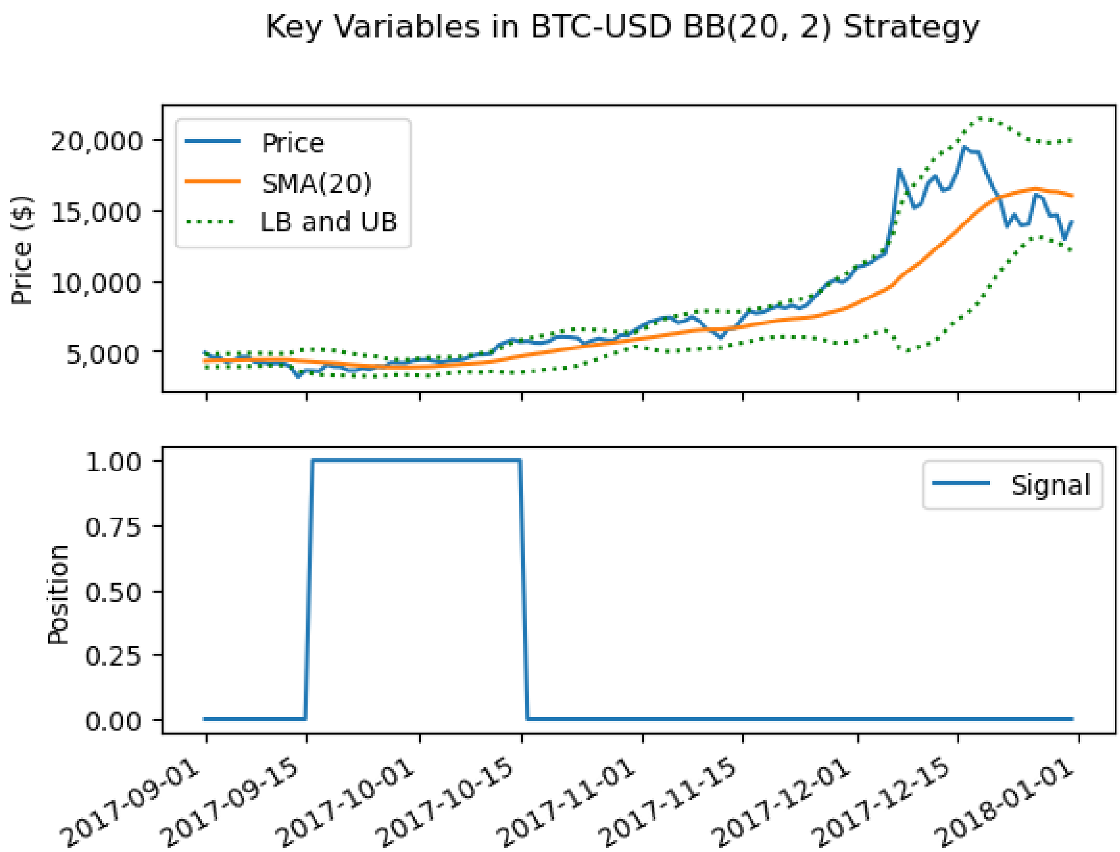
```
(
    btc_bb
    [['Return', 'Strategy']]
    .rename(columns=columns_bb)
    .add(1)
    .prod()
    .rename_axis('Strategy')
    .to_frame('Value of One-Dollar Investment')
)
```

	Value of One-Dollar Investment
Strategy	
Buy-And-Hold	183.6056
BB(20, 2)	2.1877

We need a more complex plot to better understand what is going on!

```
import matplotlib.ticker as ticker
```

```
fig, ax = plt.subplots(nrows=2, ncols=1, sharex=True)
df = btc_bb.loc['2017-09':'2017-12']

ax[0].plot(df[['Adj Close']], label='Price')
ax[0].plot(df[['SMA']], label='SMA(20)')
ax[0].plot(df[['UB']], label='LB and UB', color='green', linestyle=':')
ax[0].plot(df[['LB']], color='green', linestyle=':')
ax[0].legend()
ax[0].set_ylabel('Price ($)')

ax[1].plot(df[['Position']], label='Signal')
ax[1].legend()
ax[1].set_ylabel('Position')

ax[0].yaxis.set_major_formatter(ticker.StrMethodFormatter('{x:,.0f}'))
fig.autofmt_xdate()

plt.suptitle('Key Variables in BTC-USD BB(20, 2) Strategy')
plt.show()
```



Implement a long-short RSI(14) strategy with Bitcoin

From Fidelity:

> The Relative Strength Index (RSI), developed by J. Welles Wilder, is a momentum oscillator that measures the speed and change of price movements. The RSI oscillates between zero and 100. Traditionally the RSI is considered overbought when above 70 and oversold when below 30. Signals can be generated by looking for divergences and failure swings. RSI can also be used to identify the general trend.

The RSI formula: $RSI(n) = 100 - \frac{100}{1+RS(n)}$, where $RS(n) = \frac{SMA(U,n)}{SMA(D,n)}$. For "up days", $U = \Delta\text{Adj Close}$ and $D = 0$. For "down days", $U = 0$ and $D = -\Delta\text{Adj Close}$.

We will implement a long-short RSI(14) as follows:

1. Buy when the RSI crosses 30 from below, and sell when the RSI crosses 50 from below

2. Short when the RSI crosses 70 from above, and cover when the RSI crosses 50 from above

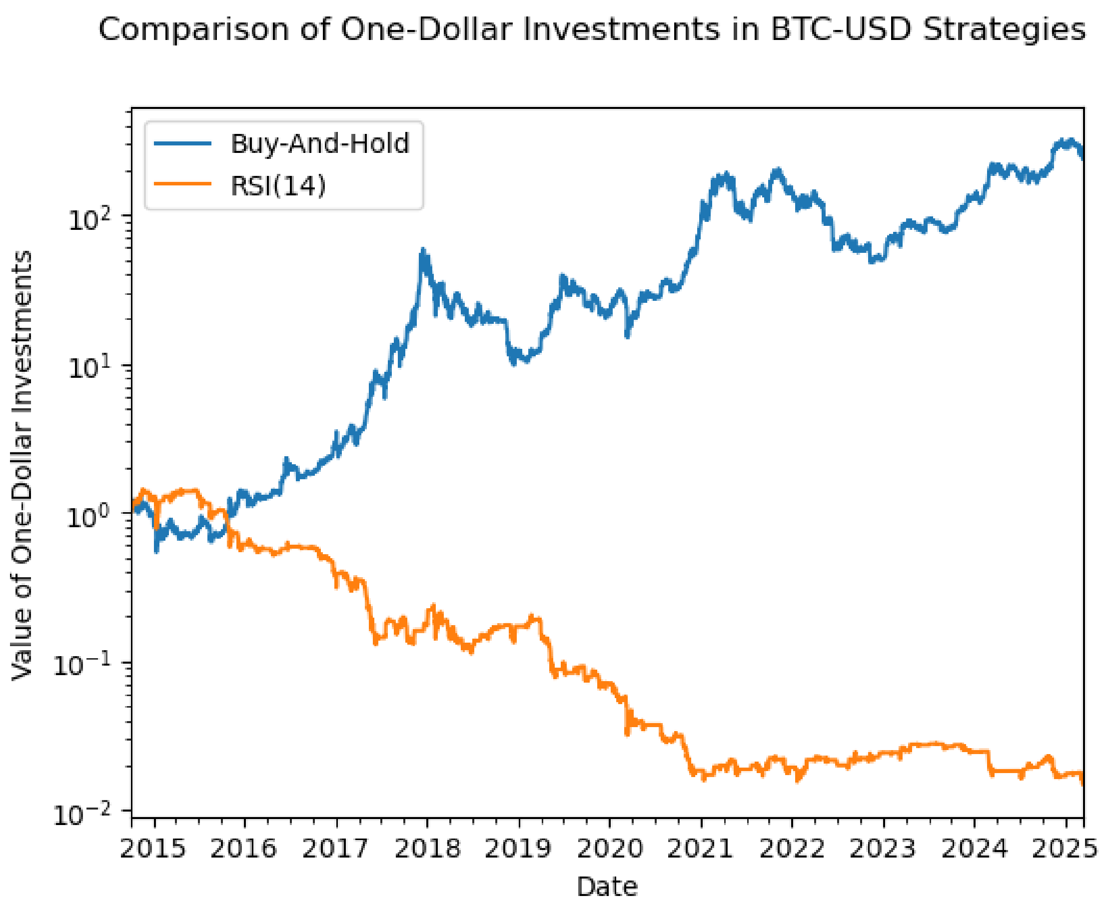
More about RSI here.

```
def calc_rsi(df, window=14, lo=30, mid=50, hi=70):
    return (
        df
        .assign(
            Return=lambda x: x['Adj Close'].pct_change(),
            # This approach with .max() and .min() handles NA values better than the in-class
            Diff=lambda x: x['Adj Close'].diff(),
            Zero=0,
            U=lambda x: x[['Diff', 'Zero']].max(axis=1, skipna=False),
            D=lambda x: -x[['Diff', 'Zero']].min(axis=1, skipna=False),
            SMAU=lambda x: x['U'].rolling(window=window).mean(),
            SMAD=lambda x: x['D'].rolling(window=window).mean(),
            RS=lambda x: x['SMAU'] / x['SMAD'],
            RSI=lambda x: 100 - 100 / (1 + x['RS']),
            Position_w_nan=lambda x: np.select(
                condlist=[
                    (x['RSI'].shift(1) > lo) & (x['RSI'].shift(2) <= lo),
                    (x['RSI'].shift(1) > mid) & (x['RSI'].shift(2) <= mid),
                    (x['RSI'].shift(1) < hi) & (x['RSI'].shift(2) >= hi),
                    (x['RSI'].shift(1) < mid) & (x['RSI'].shift(2) >= mid),
                ],
                choicelist=[1, 0, -1, 0],
                default=np.nan
            ),
            Position=lambda x: x['Position_w_nan'].ffill(),
            Strategy=lambda x: x['Position'] * x['Return']
        )
    )
```

```
btc_rsi = btc.pipe(calc_rsi)
```

```
columns_rsi = {
    'Adj Close': 'Price',
    'Return': 'Buy-And-Hold',
    'Strategy': 'RSI(14)'
}
```

```
(
    btc_rsi
    [['Return', 'Strategy']]
    .dropna()
    .rename(columns=columns_rsi)
    .add(1)
    .cumprod()
    .plot()
)
plt.semilogy()
plt.suptitle('Comparison of One-Dollar Investments in BTC-USD Strategies')
plt.ylabel('Value of One-Dollar Investments')
plt.show()
```



Implement a golden cross with Bitcoin

Someone in Section 04 mentioned two-moving average strategies, so I added this golden cross, where the 50-day SMA crosses the 200-day SMA, to every section.

From Grok:

> In technical analysis, a golden cross is a bullish chart pattern that occurs when a short-term moving average (typically the 50-day moving average) crosses above a long-term moving average (typically the 200-day moving average). This crossover is considered a signal that a stock, index, or other asset may be entering a sustained upward trend, suggesting potential buying opportunities for traders and investors.

More here.

```
def calc_cross(df, long=200, short=50):
    return (
        df
        .assign(
            Return=lambda x: x['Adj Close'].pct_change(),
            SMAL=lambda x: x['Adj Close'].rolling(window=long).mean(),
            SMAS=lambda x: x['Adj Close'].rolling(window=short).mean(),
            Position=lambda x: np.select(
                condlist=[
                    x['SMAS'].shift(1) > x['SMAL'].shift(1),
                    x['SMAS'].shift(1) <= x['SMAL'].shift(1)
                ],
                choicelist=[1, 0],
                default=np.nan
            ),
            Strategy=lambda x: x['Position'] * x['Return']
        )
    )
```

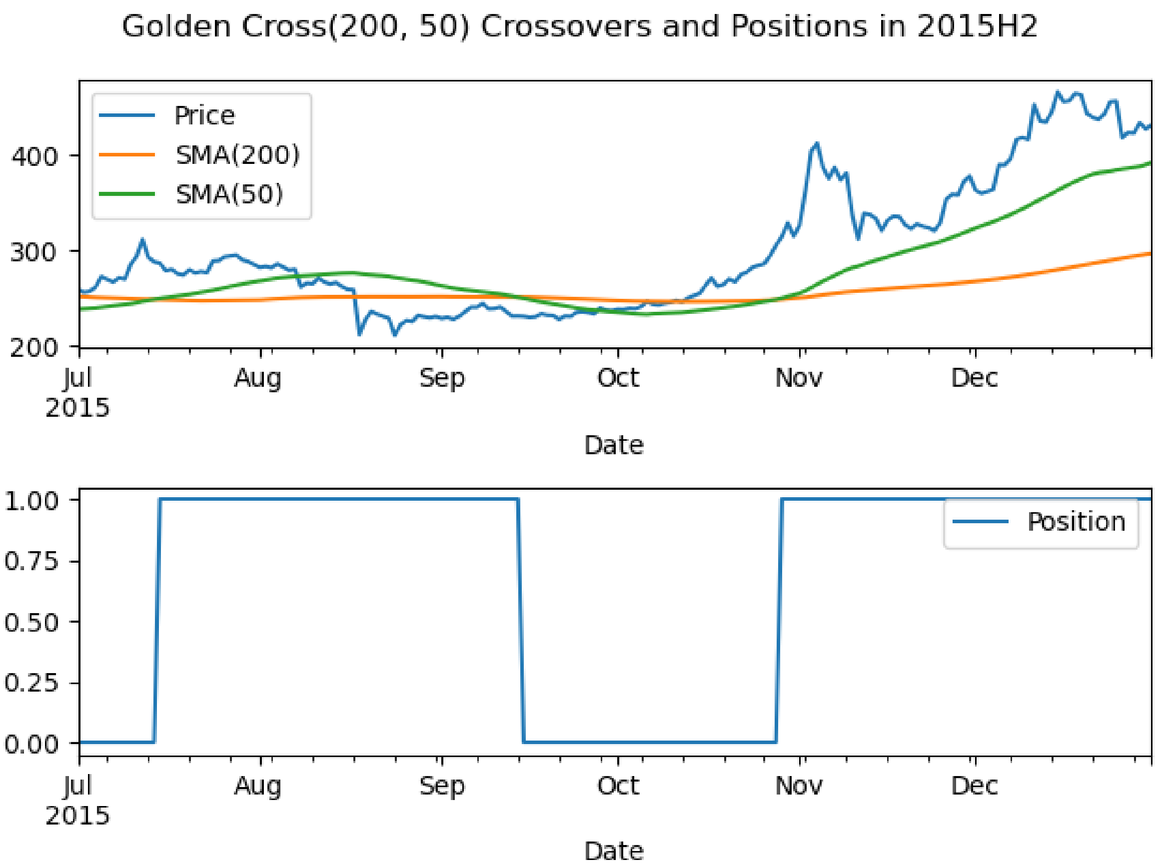
```
btc_cross = btc.pipe(calc_cross, long=200, short=50)
```

```
columns_cross = {
    'Adj Close': 'Price',
    'SMAL': 'SMA(200)',
    'SMAS': 'SMA(50)',
    'Return': 'Buy-And-Hold',
    'Strategy': 'Golden Cross(200, 50)'
}
```

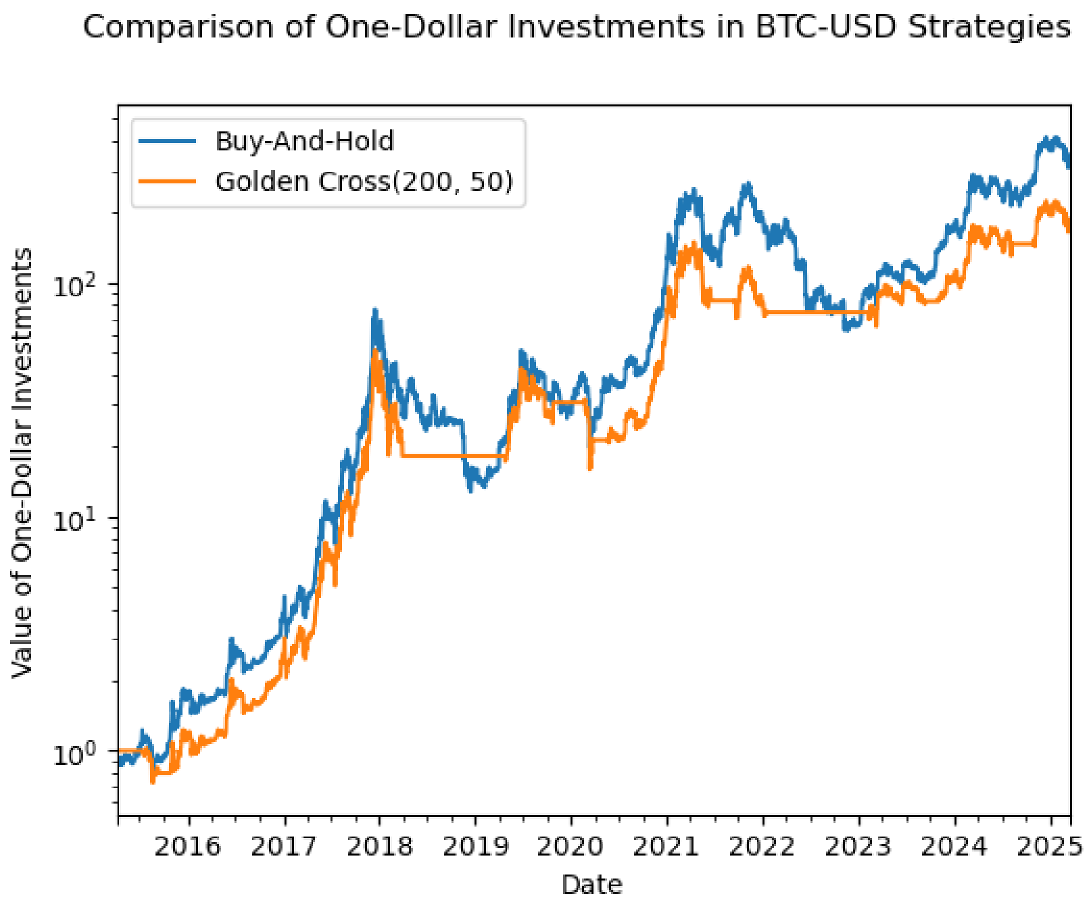
```python
btc_cross.query('Position == 1')
```

Date	Adj Close	Close	High	Low	Open	Volume	Return	SMAL
2015-07-15	285.8290	285.8290	293.2480	285.3670	288.0450	27486600	-0.0057	247.78
2015-07-16	278.0890	278.0890	291.1830	275.2400	286.0420	49482600	-0.0271	247.58
2015-07-17	279.4720	279.4720	280.2800	272.0430	278.0910	27591400	0.0050	247.42
2015-07-18	274.9010	274.9010	282.5270	274.0750	279.3310	25187100	-0.0164	247.24
2015-07-19	273.6140	273.6140	275.6700	272.5130	274.7670	15332500	-0.0047	247.00
...	...	...	...	...	...	...	...	...
2025-03-10	78532.0000	78532.0000	83955.9297	77420.5938	80597.1484	54061099422	-0.0257	83357
2025-03-11	82862.2109	82862.2109	83577.7578	76624.2500	78523.8750	54702837196	0.0551	83451
2025-03-12	83722.3594	83722.3594	84358.5781	80635.2500	82857.3750	40353484454	0.0104	83549
2025-03-13	81066.7031	81066.7031	84301.6953	79931.8516	83724.9219	31412940153	-0.0317	83633
2025-03-14	83969.1016	83969.1016	85263.2891	80797.5625	81066.9922	29588112414	0.0358	83738

```python
fig, ax = plt.subplots(2, 1)
df = btc_cross.loc['2015-07':'2015-12']
df[['Adj Close', 'SMAL', 'SMAS']].rename(columns=columns_cross).plot(ax=ax[0])
df[['Position']].plot(ax=ax[1])
plt.suptitle('Golden Cross(200, 50) Crossovers and Positions in 2015H2')
plt.tight_layout()
plt.show()
```




```
(
    btc_cross
    [['Return', 'Strategy']]
    .dropna()
    .rename(columns=columns_cross)
    .add(1)
    .cumprod()
    .plot()
)
plt.semilogy()
plt.suptitle('Comparison of One-Dollar Investments in BTC-USD Strategies')
plt.ylabel('Value of One-Dollar Investments')
plt.show()
```

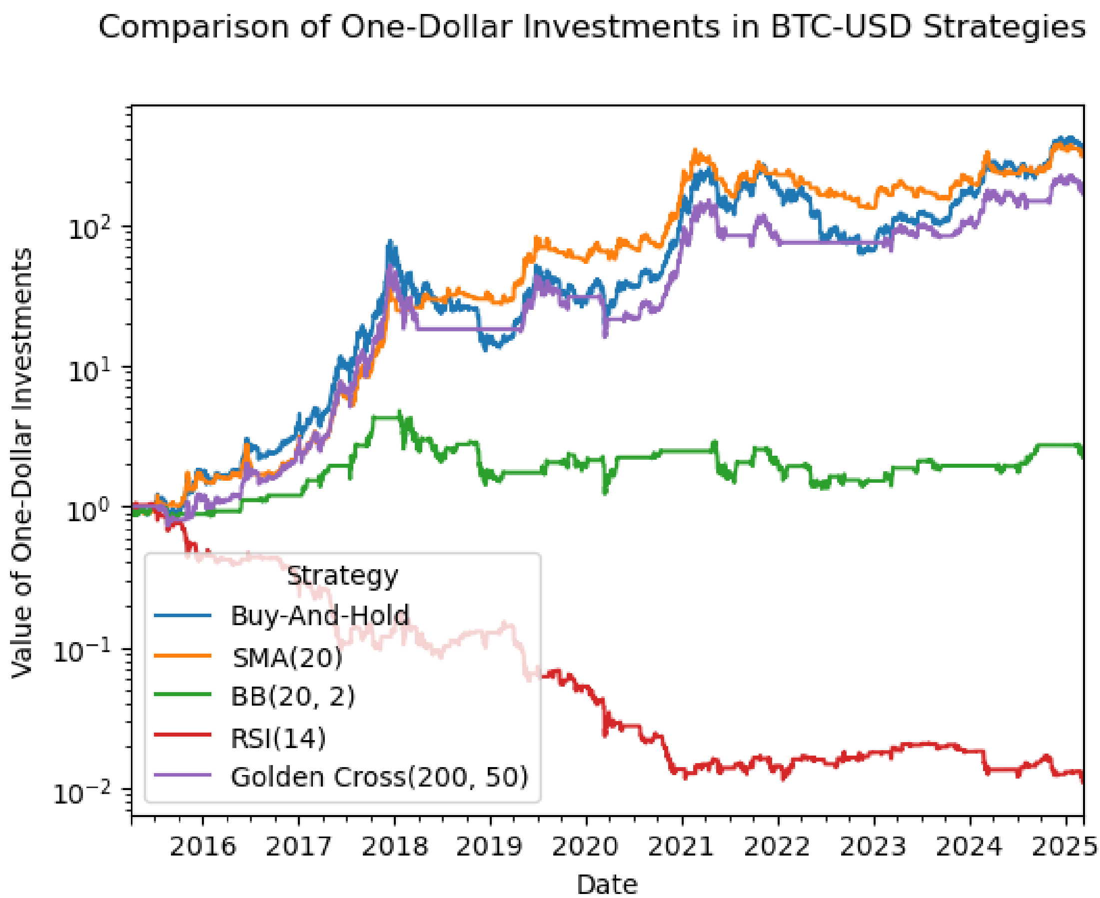


Compare all strategies

I added this comparison of all strategies after class.

```
df = (
    btc_sma[['Return', 'Strategy']]
    .join(btc_bb[['Strategy']], rsuffix='_bb')
    .join(btc_rsi[['Strategy']], rsuffix='_rsi')
    .join(btc_cross[['Strategy']], rsuffix='_cross')
    .dropna()
)


(
    df
```

```
    .add(1)
    .cumprod()
    .rename_axis(columns='Strategy')
    .rename(columns={
        'Return': 'Buy-And-Hold',
        'Strategy': 'SMA(20)',
        'Strategy_bb': 'BB(20, 2)',
        'Strategy_rsi': 'RSI(14)',
        'Strategy_cross': 'Golden Cross(200, 50)',
    })
    .plot()
)

plt.semilogy()
plt.suptitle('Comparison of One-Dollar Investments in BTC-USD Strategies')
plt.ylabel('Value of One-Dollar Investments')
plt.show()
```


Comparison of One-Dollar Investments in BTC-USD Strategies
Value of One-Dollar Investments
10^2
10^1
10^0
10^{-1}
10^{-2}
Strategy
Buy-And-Hold
SMA(20)
BB(20, 2)
RSI(14)
Golden Cross(200, 50)
2016
2017
2018
2019
2020
2021
2022
2023
2024
2025
Date

Herron Topic 2 - Practice - Sec 04

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import pandas_datareader as pdr
import seaborn as sns
import statsmodels.api as sm
import yfinance as yf
```

```
%precision 4
pd.options.display.float_format = '{:.4f}'.format
# %config InlineBackend.figure_format = 'retina'
```

Announcements

1. I am still grading your projects; I hope to finish them by next Tuesday
2. I posted 50 practice problems to prepare for the end-of-course Programming Assessment here: https://northeastern.instructure.com/courses/207607/discussion_topics/2727917
    1. I built 5 autograded notebooks to help you prepare for the assessment, and I will build more in the coming weeks
    2. To run these notebooks, install the `otter-grader` package: In the Anaconda command prompt (or Terminal on Mac) run `conda activate fina6333` then `conda install otter-grader`
    3. See the video at the link above for details

Five-Minute Review

1. Technical analysis (TA) is a method of evaluating trends in trading prices and volume (and open interest in the futures and options markets only)
2. The three tenants of TA are:
    1. Markets discount everything

   2. Prices move in trends
   3. History repeats itself, so these trends are recurring

3. Academics have not found much evidence that TA generate profits that exceed transaction costs, but we will spend a week on it for three reasons:

   1. You asked for it, and it will help build our data analytics skills
   2. It is a small part of the Chartered Financial Analyst (CFA) curriculum
   3. That TA still receives attention 60 years after the Efficient Markets Hypothesis (EMH) suggests that it has some value that academics have been unable to measure

Practice

Implement the SMA(20) strategy with BTC-USD from the lecture notebook

Try to create the `btc_sma` data frame from the `btc` data frame in one code cell with one assignment (i.e., one `=`).

```
btc = (
    yf.download(
        tickers='BTC-USD',
        auto_adjust=False,
        progress=False,
        multi_level_index=False
    )
    .iloc[:-1] # drop incomplete trading day
)
```

After class, I converted our code to a function. We might want to test different moving average parameters, and a function makes these tests easier.

The following `calc_sma()` function accepts:

1. A data frame `df` of daily values from `yfinance.download()`
2. An integer `window` that specifies the number of trading days in the SMA window

And returns the original data frame `df` plus the following columns:

1. `Return` with daily returns
2. `SMA` for the `window`-trading-day moving average
3. `Position` for the weight on the security each day
4. `Strategy` for the return on the strategy each day

```
def calc_sma(df, window=20):
    return (
        df
        .assign(
            Return=lambda x: x['Adj Close'].pct_change(),
            SMA=lambda x: x['Adj Close'].rolling(window=window).mean(),
            Position=lambda x: np.select(
                condlist=[
                    x['Adj Close'].shift(1) > x['SMA'].shift(1),
                    x['Adj Close'].shift(1) <= x['SMA'].shift(1)
                ],
                choicelist=[1, 0],
                default=np.nan
            ),
            Strategy=lambda x: x['Position'] * x['Return']
        )
    )
```

```
btc_sma = btc.pipe(calc_sma, window=20)
```

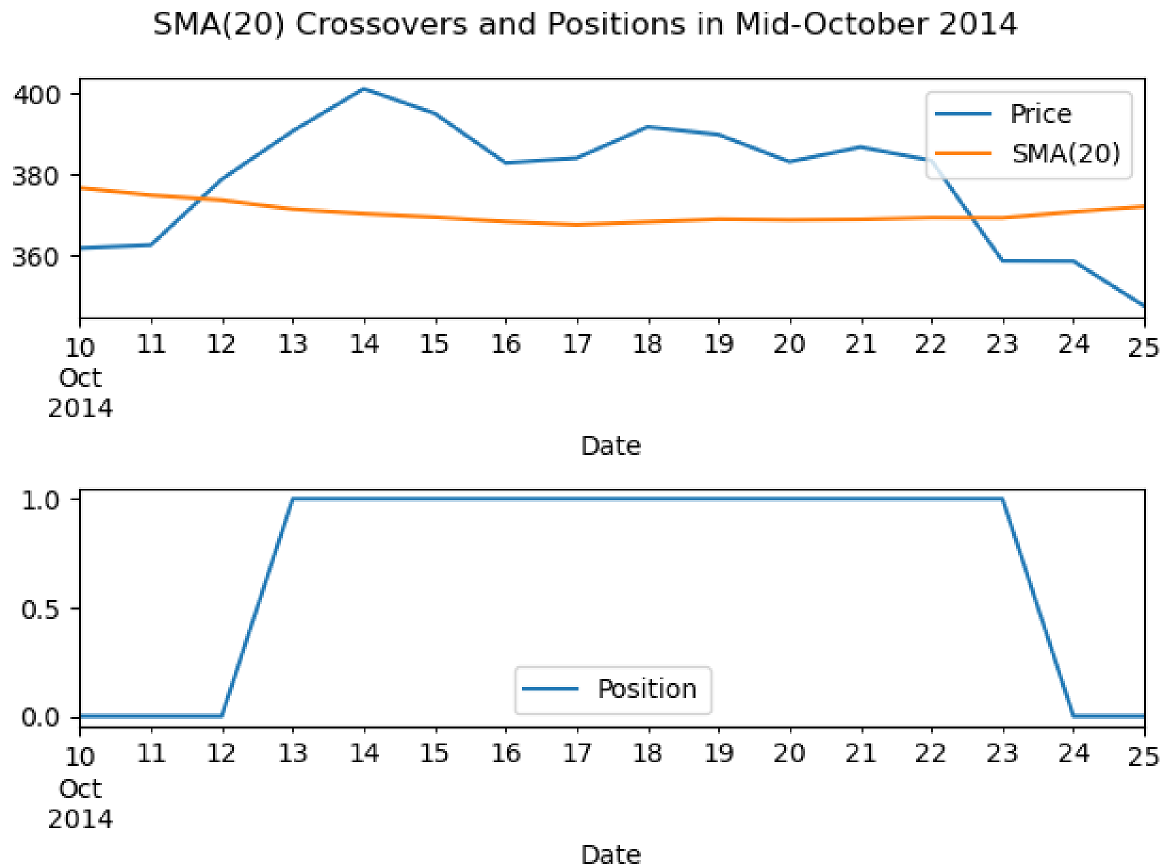
It can be helpful to visualize a few places where the price (here `Adj Close`) crosses the moving average (here SMA(20)). In class, we found that there are a few changes in position in the middle of October, 2014. The following code create two subplots in one figure, then puts one plot in each of the subplots.

For the first crossover on October 12, we *buy* at the close on October 12 and earn the security return on October 13. For the second crossover on October 23, we *sell* at the close on October 23 and earn zero return on October 24.

```
columns_sma20 = {
    'Adj Close': 'Price',
    'SMA': 'SMA(20)',
    'Return': 'Buy-And-Hold',
    'Strategy': 'SMA(20)'
}
```

```
fig, ax = plt.subplots(2, 1)
df = btc_sma.loc['2014-10-10':'2014-10-25']
df[['Adj Close', 'SMA']].rename(columns=columns_sma20).plot(ax=ax[0])
df[['Position']].plot(ax=ax[1])
plt.suptitle('SMA(20) Crossovers and Positions in Mid-October 2014')
```

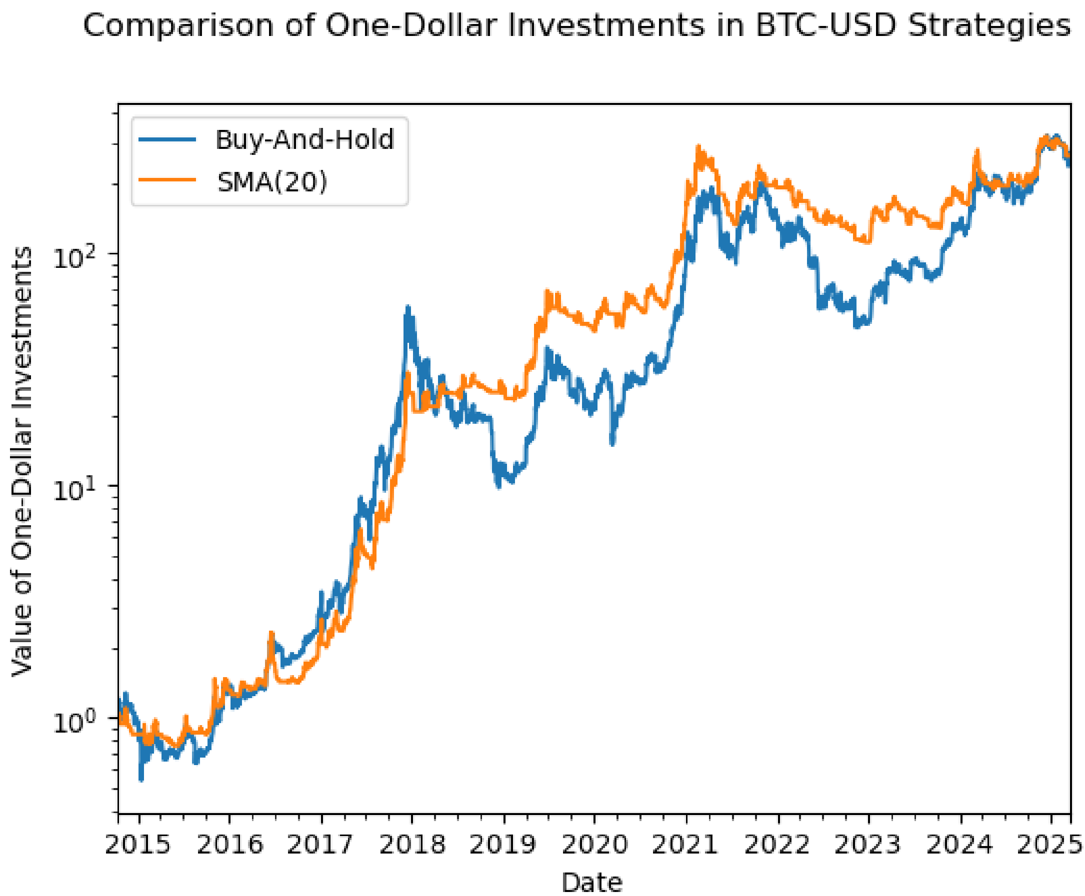
```
plt.tight_layout()
plt.show()
```



We can compare the total returns on BTC-USD and our SMA(20) strategy! We `.dropna()` first because the SMA(20) strategy needs 20 days of data to make its first investing decision.

```
(
    btc_sma
    [['Return', 'Strategy']]
    .dropna()
    .rename(columns=columns_sma20)
    .add(1)
    .cumprod()
    .plot()
)
plt.semilogy()
```

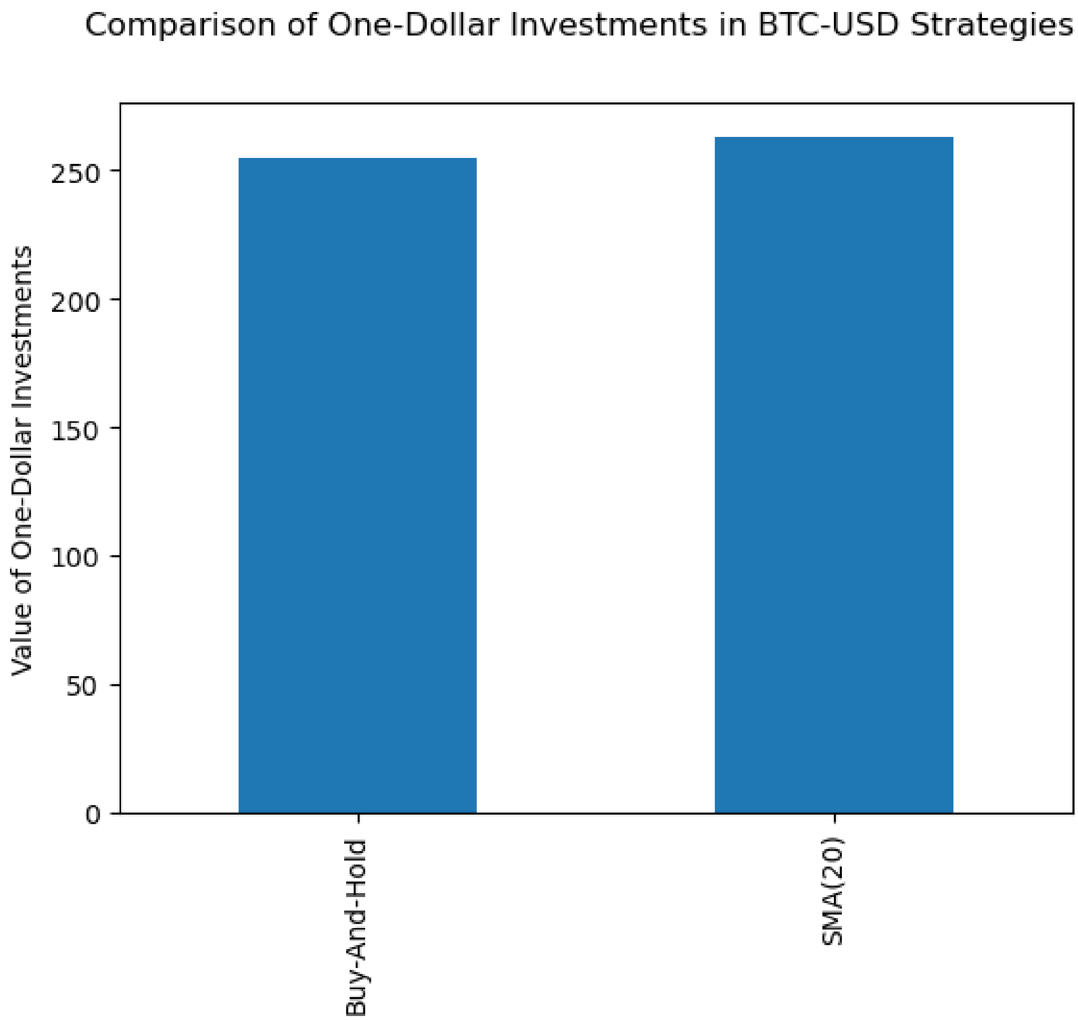
```
plt.suptitle('Comparison of One-Dollar Investments in BTC-USD Strategies')
plt.ylabel('Value of One-Dollar Investments')
plt.show()
```



These total returns, at least today, are similar!

```
(
    btc_sma
    [['Return', 'Strategy']]
    .dropna()
    .rename(columns=columns_sma20)
    .add(1)
    .prod()
    .plot(kind='bar')
)
plt.suptitle('Comparison of One-Dollar Investments in BTC-USD Strategies')
```

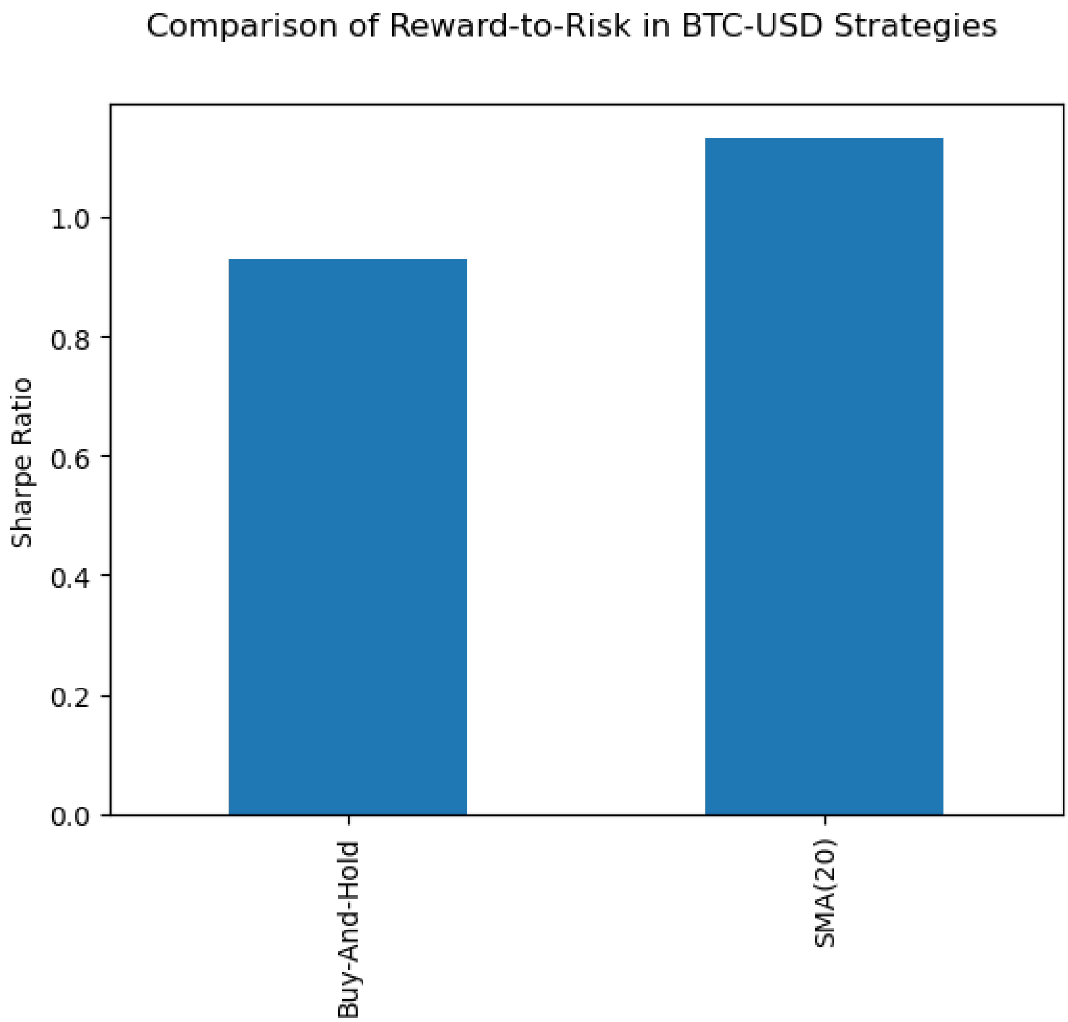
```
plt.ylabel('Value of One-Dollar Investments')
plt.show()
```



However, the Sharpe ratio of the SMA(20) is higher, because time out of the market when `Position=0` reduces risk. For simplicity, we will ignore the risk-free rate in the Sharpe ratio formula.

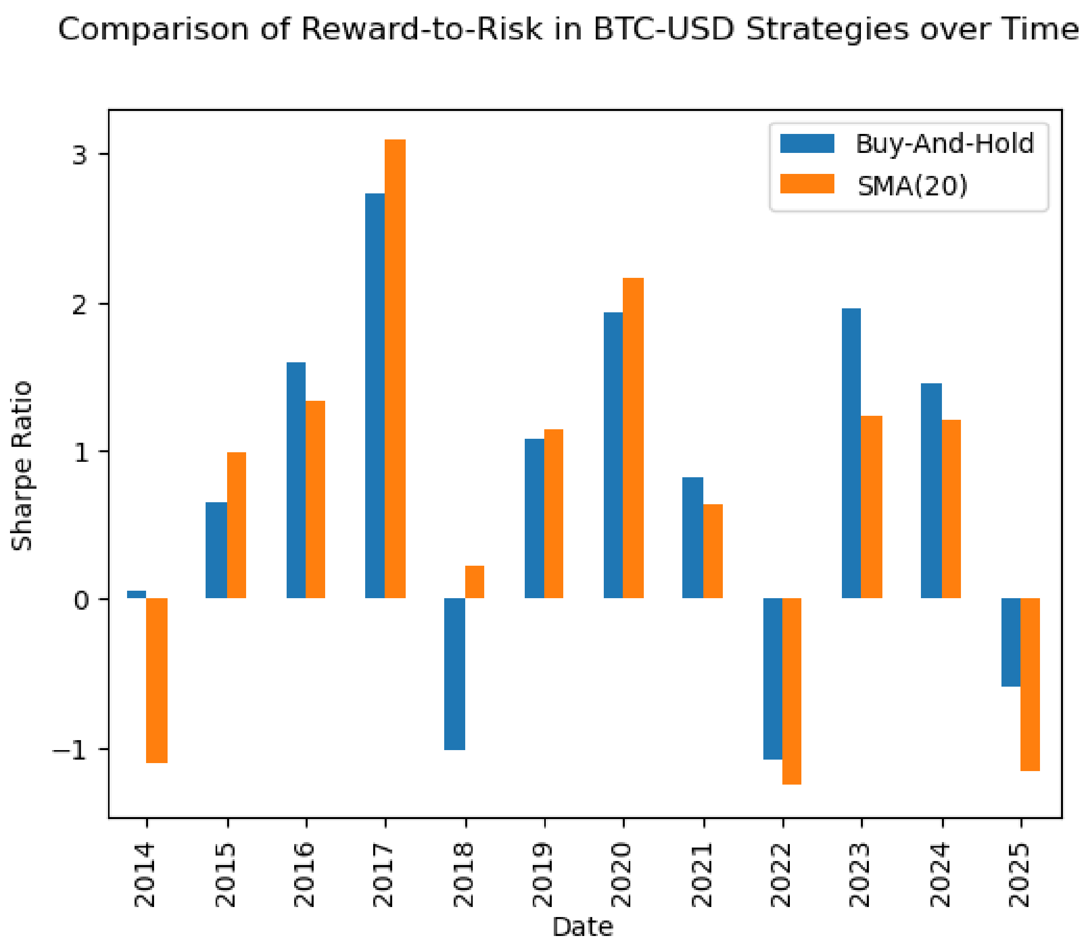
```
(
    btc_sma
    [['Return', 'Strategy']]
    .dropna()
    .rename(columns=columns_sma20)
```

```
    .apply(lambda x: np.sqrt(252) * x.mean() / x.std())
    .plot(kind='bar')
)
plt.suptitle('Comparison of Reward-to-Risk in BTC-USD Strategies')
plt.ylabel('Sharpe Ratio')
plt.show()
```



But how persistent is this outperformance in terms of reward-to-risk ratios? We can quickly modify the code above to calculate Sharpe ratios each year! We find that SMA(20) Sharpe ratio edge is not persistent.

```
df = (
    btc_sma
    [['Return', 'Strategy']]
    .dropna()
    .rename(columns=columns_sma20)
    .resample('YE')
    .apply(lambda x: np.sqrt(252) * x.mean() / x.std())
)
df.index = df.index.year
df.plot(kind='bar')
plt.suptitle('Comparison of Reward-to-Risk in BTC-USD Strategies over Time')
plt.ylabel('Sharpe Ratio')
plt.show()
```



We can use our list comprehension skills to easily try several window sizes!

```
def CAGR(x):
    T = x.count()
    return (x.add(1).prod() ** (252 / T)) - 1
```

```
def Sharpe(x, ann_fac=np.sqrt(252)):
    return ann_fac * x.mean() / x.std()
```

```
columns_sman = {
    'Adj Close': 'Price',
    'SMA': 'SMA(N)',
    'Return': 'Buy-And-Hold',
    'Strategy': 'SMA(N)'
}
```

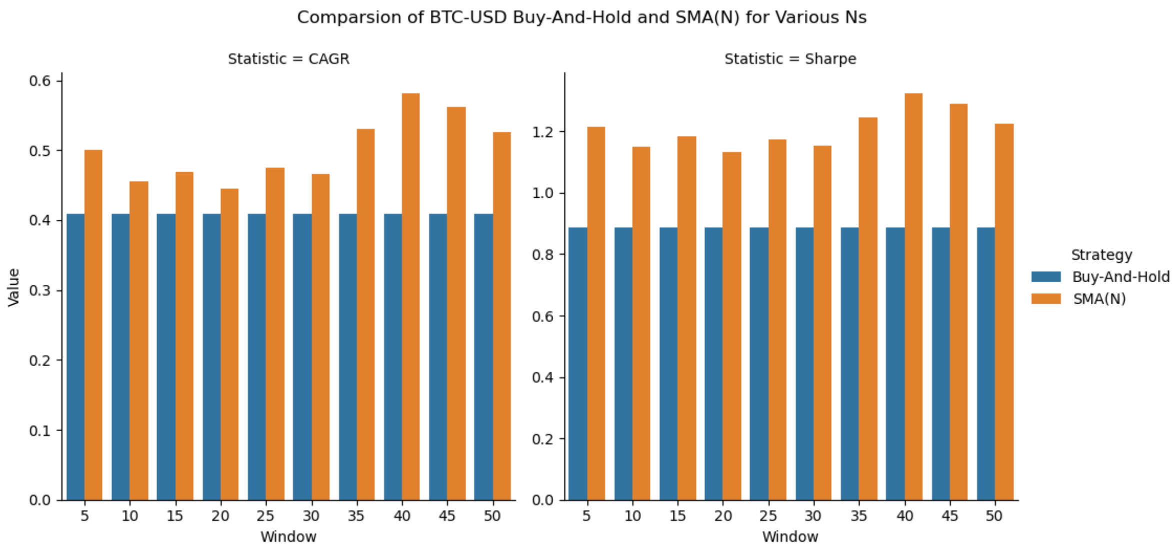
```
windows = list(range(5, 55, 5))

btc_smas = (
    pd.concat(
        objs=[
            btc.pipe(calc_sma, window=w)[['Return', 'Strategy']].agg([CAGR, Sharpe])
            for w in windows
        ],
        keys=windows,
        names=['Window', 'Statistic']
    )
    .rename(columns=columns_sman)
)
```

```
btc_smas
```

		Buy-
Window	Statistic	
5	CAGR	0.40
	Sharpe	0.88
10	CAGR	0.40
	Sharpe	0.88

		Buy-
Window	Statistic	
15	CAGR	0.40
	Sharpe	0.88
20	CAGR	0.40
	Sharpe	0.88
25	CAGR	0.40
	Sharpe	0.88
30	CAGR	0.40
	Sharpe	0.88
35	CAGR	0.40
	Sharpe	0.88
40	CAGR	0.40
	Sharpe	0.88
45	CAGR	0.40
	Sharpe	0.88
50	CAGR	0.40
	Sharpe	0.88

```
(
    btc_smas
    .reset_index()
    .melt(
        id_vars=['Window', 'Statistic'],
        value_vars=['Buy-And-Hold', 'SMA(N)'],
        var_name='Strategy',
        value_name='Value'
    )
    .pipe(
        sns.catplot,
        x='Window',
        col='Statistic',
        hue='Strategy',
        y='Value',
        kind='bar',
        sharey=False
    )
)
plt.suptitle('Comparsion of BTC-USD Buy-And-Hold and SMA(N) for Various Ns', y=1.05)
plt.show()
```



Investigate how SMA(20) generates returns

Consider the following:

1. Does SMA(20) avoid the worst performing days? How many of the worst 20 days does SMA(20) avoid? Try the `.nlargest()` method.
2. Does SMA(20) preferentially avoid low-return days? Try to combine the `.groupby()` method and `pd.qcut()` function.
3. Does SMA(20) preferentially avoid high-volatility days? Try to combine the `.groupby()` method and `pd.qcut()` function.

The SMA(20) does well here because it avoids 17 of the 20 worst days, without avoiding the best days.

```python
btc_sma.loc[btc_sma['Return'].nsmallest(20).index, ['Position']].value_counts()
```

```
Position
0.0000      17
1.0000       3
Name: count, dtype: int64
```

```
btc_sma.loc[btc_sma['Return'].nlargest(20).index, ['Position']].value_counts()
```

```
Position
0.0000      10
1.0000      10
Name: count, dtype: int64
```

We can also look at the descriptive statistics by `Postiion` and strategy. Two observations:

1. The `min` column shows that SMA(20) misses the bad days, like we see above
2. The `mean` in `Position=1` is about 5 times higher than in `Position=0`

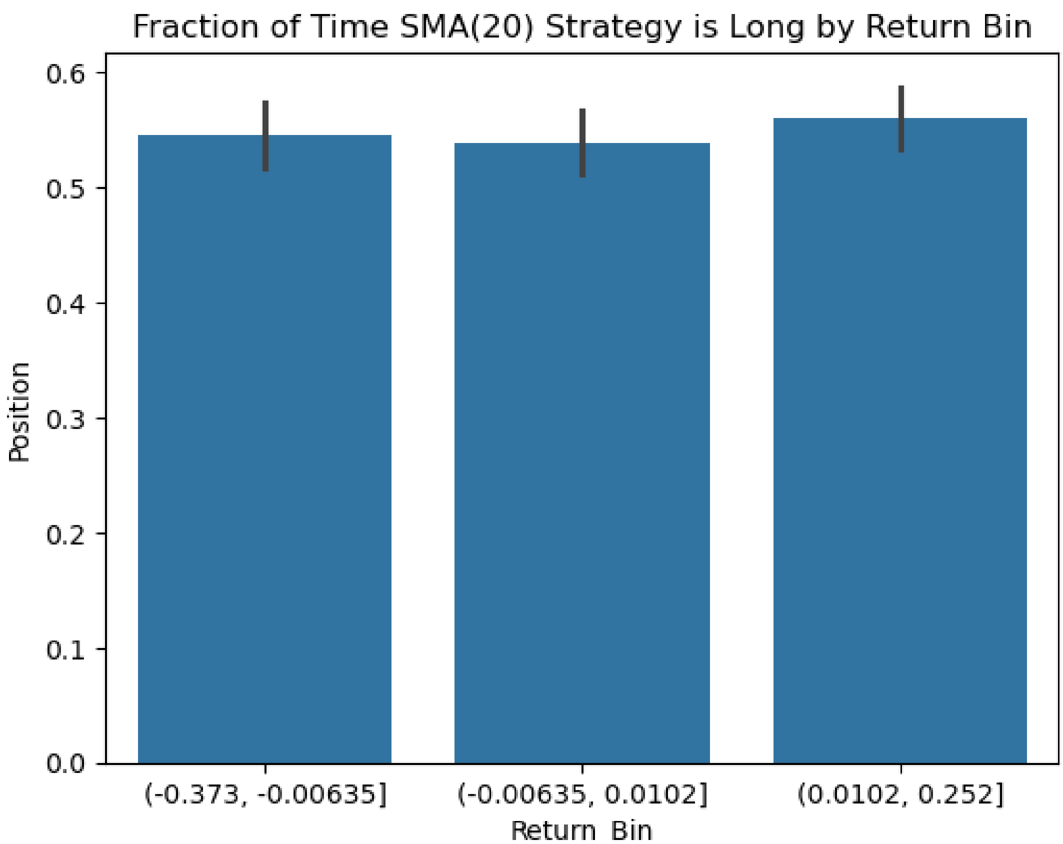
```
(
    btc_sma
    [['Position', 'Return', 'Strategy']]
    .dropna()
    .groupby('Position')
    .describe()
    .rename(columns=columns_sma20)
    .rename_axis(columns=['Strategy', 'Statistic'])
    .stack('Strategy', future_stack=True)
)
```

Position	Statistic Strategy
0.0000	Buy-And-Hold
	SMA(20)
1.0000	Buy-And-Hold
	SMA(20)

We can also use the seaborn package to visualize `Position` (i.e., the portfolio weight on Bitcoin) during periods of high and low Bitcoin returns and volatility. The SMA(20) strategy is long Bitcoin about 55% of the time, whether Bitcoin returns are high (bin 2) or low (bin 1).

```
(
    btc_sma
    [['Return', 'Position']]
    .dropna()
    .assign(Return_Bin=lambda x: pd.qcut(x['Return'], q=3))
```

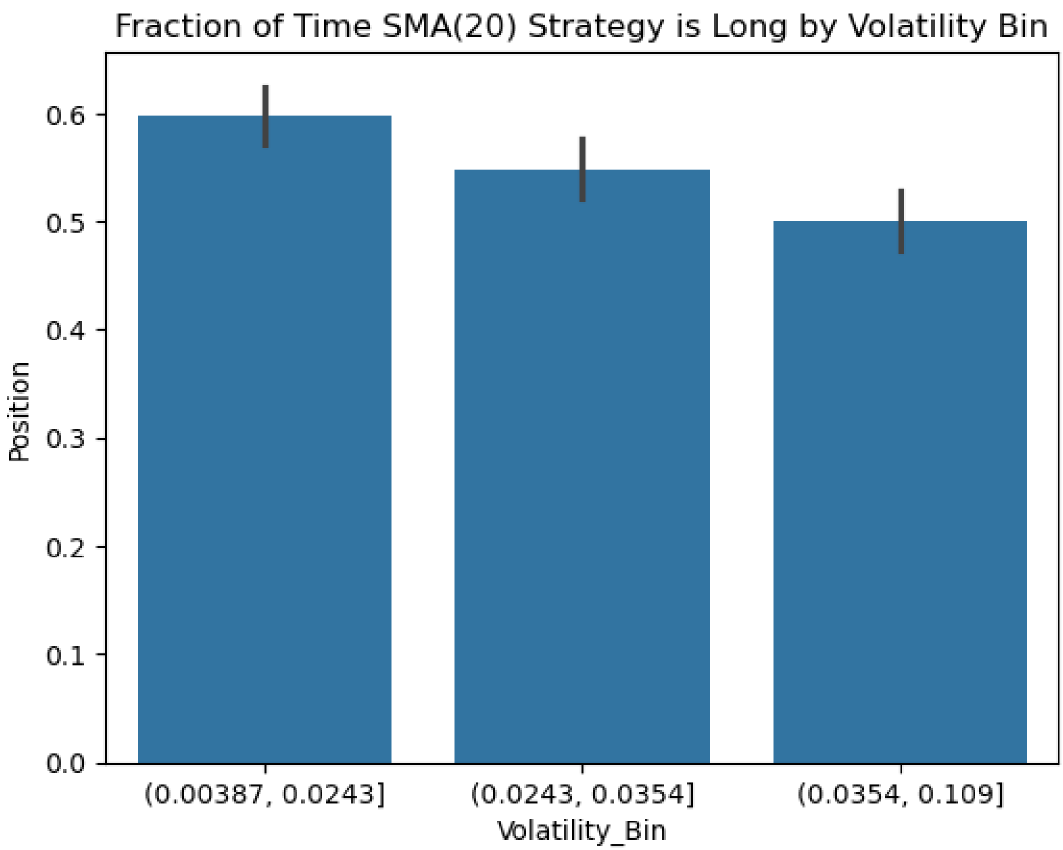
```
    .pipe(
        sns.barplot,
        x='Return_Bin',
        y='Position'
    )
)

plt.title('Fraction of Time SMA(20) Strategy is Long by Return Bin')
plt.show()
```



However, the SMA(20) stateegy, *for this security, sample, and window,* spends less times in Bitcoind during volatilit times.

```
(
    btc_sma
    [['Return', 'Position']]
    .dropna()
```

```
    .assign(Volatility_Bin=lambda x: pd.qcut(x['Return'].rolling(20).std(), q=3))
    .pipe(
        sns.barplot,
        x='Volatility_Bin',
        y='Position'
    )
)

plt.title('Fraction of Time SMA(20) Strategy is Long by Volatility Bin')
plt.show()
```



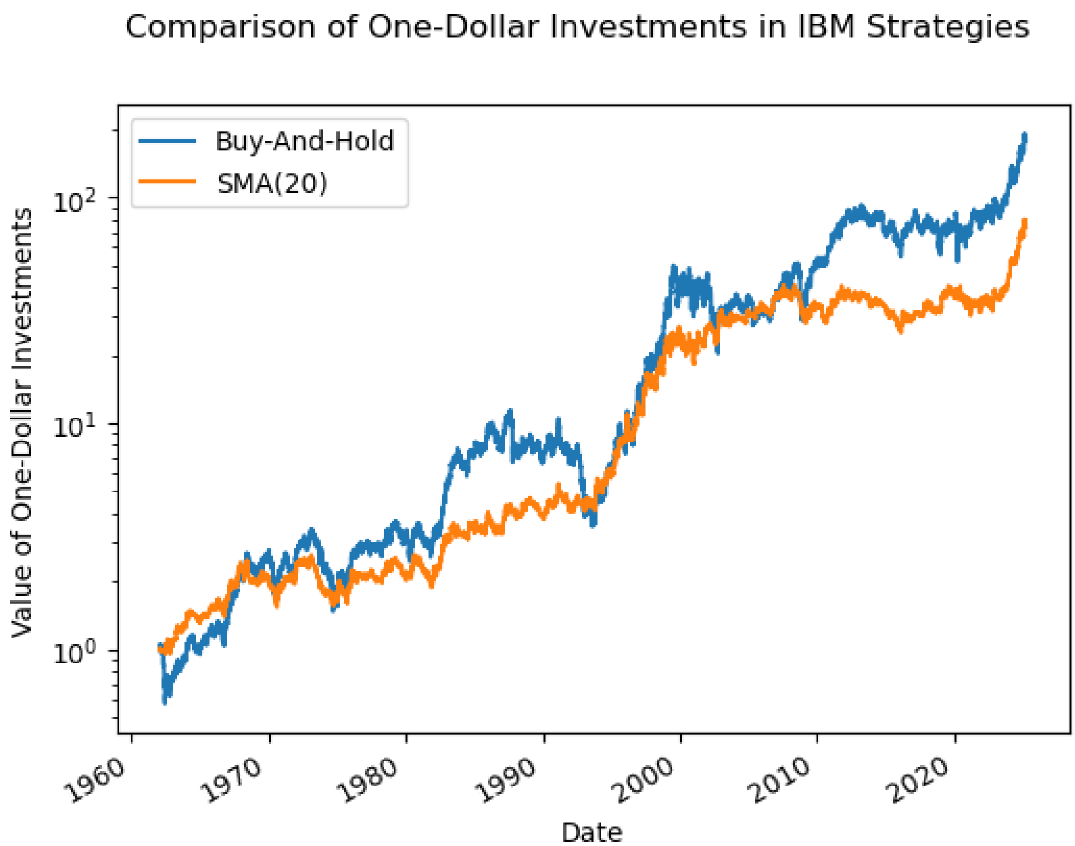
Implement the SMA(20) strategy with IBM

How often does SMA(20) outperform buy-and-hold with 10-year rolling windows?

```
ibm = (
    yf.download(
        tickers='IBM',
        auto_adjust=False,
        progress=False,
        multi_level_index=False
    )
    .iloc[:-1] # drop incomplete trading day
)
```

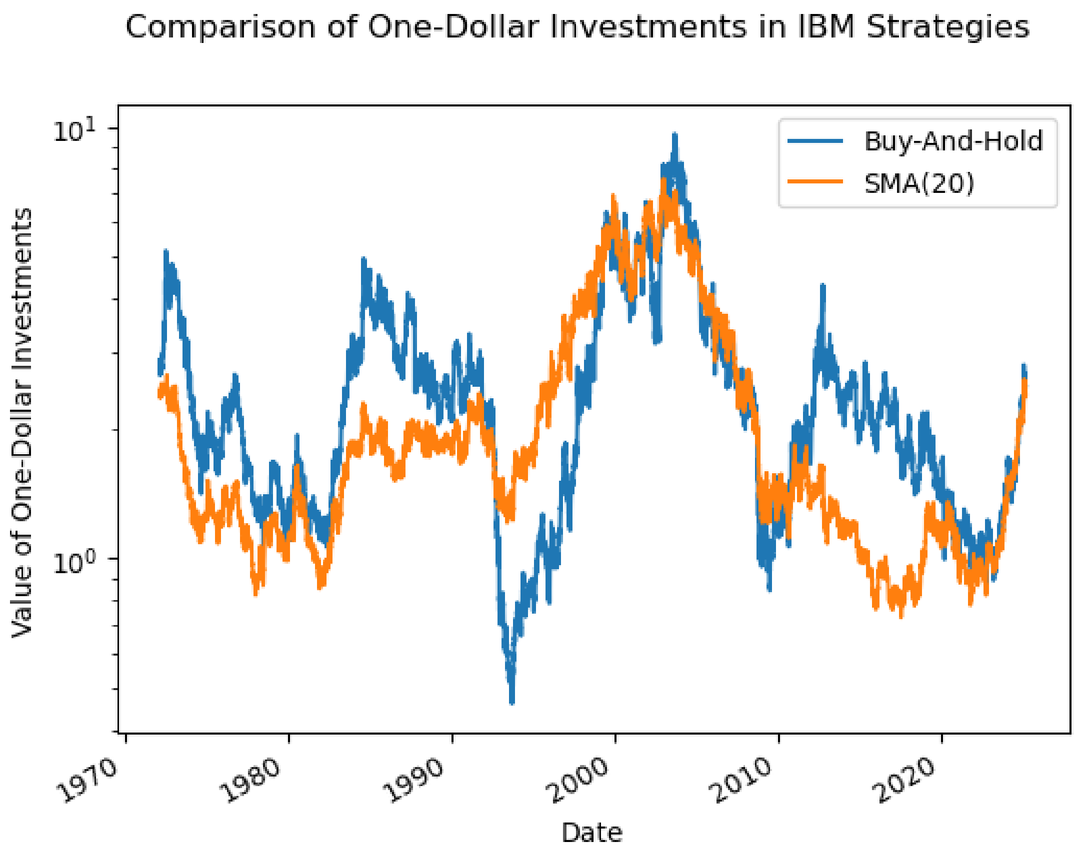
```
ibm_sma = ibm.pipe(calc_sma, window=20)
```

```
(
    ibm_sma
    [['Return', 'Strategy']]
    .dropna()
    .rename(columns=columns_sma20)
    .add(1)
    .cumprod()
    .plot()
)
plt.semilogy()
plt.suptitle('Comparison of One-Dollar Investments in IBM Strategies')
plt.ylabel('Value of One-Dollar Investments')
plt.show()
```



Over the full, 60-year sample, SMA(20) underperforms buy-and-hold. What about on rolling ten-year windows? The results look balanced, and neither clearly outperforms.

```
(
    ibm_sma
    [['Return', 'Strategy']]
    .dropna()
    .rename(columns=columns_sma20)
    .pipe(np.log1p)
    .rolling(window=10*252)
    .sum()
    .pipe(np.exp)
    .plot()
)
plt.semilogy()
plt.suptitle('Comparison of One-Dollar Investments in IBM Strategies')
plt.ylabel('Value of One-Dollar Investments')
plt.show()
```



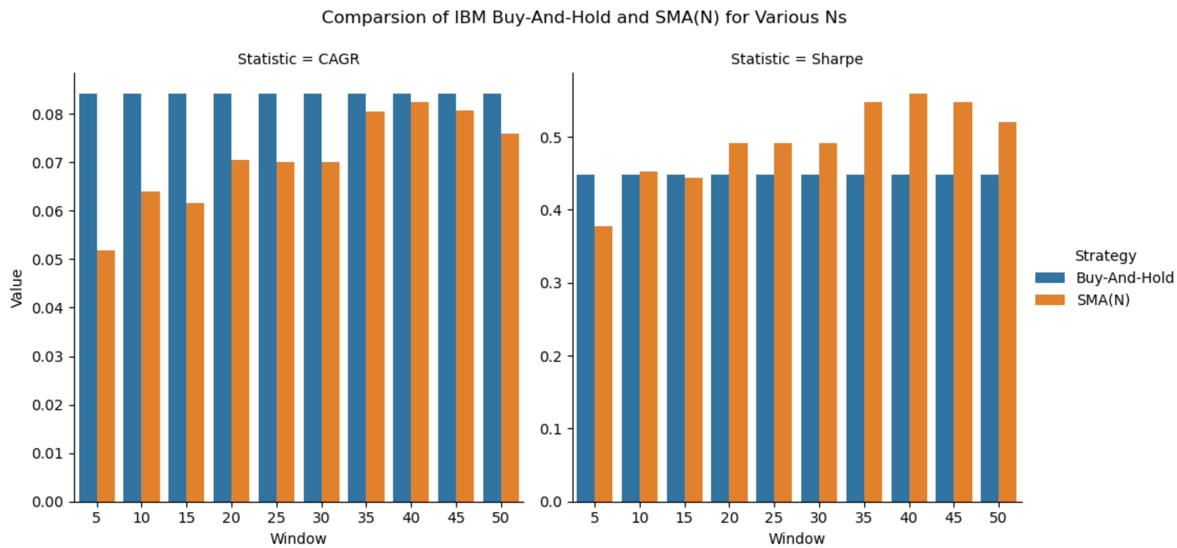
We can quantify how often SMA(20) outperform buy-and-hold. SMA(20) outperforms only 24% of the time!

```
(
    ibm_sma
    [['Return', 'Strategy']]
    .dropna()
    .pipe(np.log1p)
    .rolling(window=10*252)
    .sum()
    .pipe(np.exp)
    .pipe(lambda x: x['Strategy'] > x['Return'])
    .mean()
)
```

```
0.2378
```

```
windows = list(range(5, 55, 5))

ibm_smas = (
    pd.concat(
        objs=[
            ibm.pipe(calc_sma, window=w)[['Return', 'Strategy']].agg([CAGR, Sharpe])
            for w in windows
        ],
        keys=windows,
        names=['Window', 'Statistic']
    )
    .rename(columns=columns_sman)
)
```

```
(
    ibm_smas
    .reset_index()
    .melt(
        id_vars=['Window', 'Statistic'],
        value_vars=['Buy-And-Hold', 'SMA(N)'],
        var_name='Strategy',
        value_name='Value'
    )
    .pipe(
        sns.catplot,
        x='Window',
        col='Statistic',
        hue='Strategy',
        y='Value',
        kind='bar',
        sharey=False
    )
)
plt.suptitle('Comparsion of IBM Buy-And-Hold and SMA(N) for Various Ns', y=1.05)
plt.show()
```



Implement a long-only BB(20, 2) strategy with Bitcoin

Bollinger Bands are bands around a trend, typically defined in terms of simple moving averages and volatilities. A long-only BB(20, 2) strategy has upper and lower bands at 2 standard deviations above and below the SMA(20). It invests as follows:

1. Buy when the closing price crosses LB(20) from below
2. Sell when the closing price crosses UB(20) from above
3. No short-selling

The long-only BB(20, 2) is more difficult to implement than the long-only SMA(20) because we need to track buys and sells. For example, if the closing price is between LB(20) and BB(20), we need to know if our last trade was a buy or a sell. Further, if the closing price is below LB(20), we can still be long because we sell when the closing price crosses UB(20) from above.

More on Bollinger Bands here and here.

```
def calc_bb(df, m=20, n=2):
    return (
        df
        .assign(
            Return=lambda x: x['Adj Close'].pct_change(),
            SMA=lambda x: x['Adj Close'].rolling(window=m).mean(),
            SMV=lambda x: x['Adj Close'].rolling(window=m).std(),
            UB=lambda x: x['SMA'] + n*x['SMV'],
```

```
            LB=lambda x: x['SMA'] - n*x['SMV'],
            Position_w_nan=lambda x: np.select(
                condlist=[
                    (x['Adj Close'].shift(1) > x['LB'].shift(1)) & (x['Adj Close'].shift(2) <
                    (x['Adj Close'].shift(1) < x['UB'].shift(1)) & (x['Adj Close'].shift(2) >
                ],
                choicelist=[1, 0],
                default=np.nan
            ),
            Position=lambda x: x['Position_w_nan'].ffill(),
            Strategy=lambda x: x['Position'] * x['Return']
        )
    )
```

```
btc_bb = btc.pipe(calc_bb)
```

The BB(20, 2) only spends 40% of its time long BTC!

```
btc_bb['Position'].mean()
```

```
0.4013
```

And BTC-USD performance is worse when it long that when its neutral!

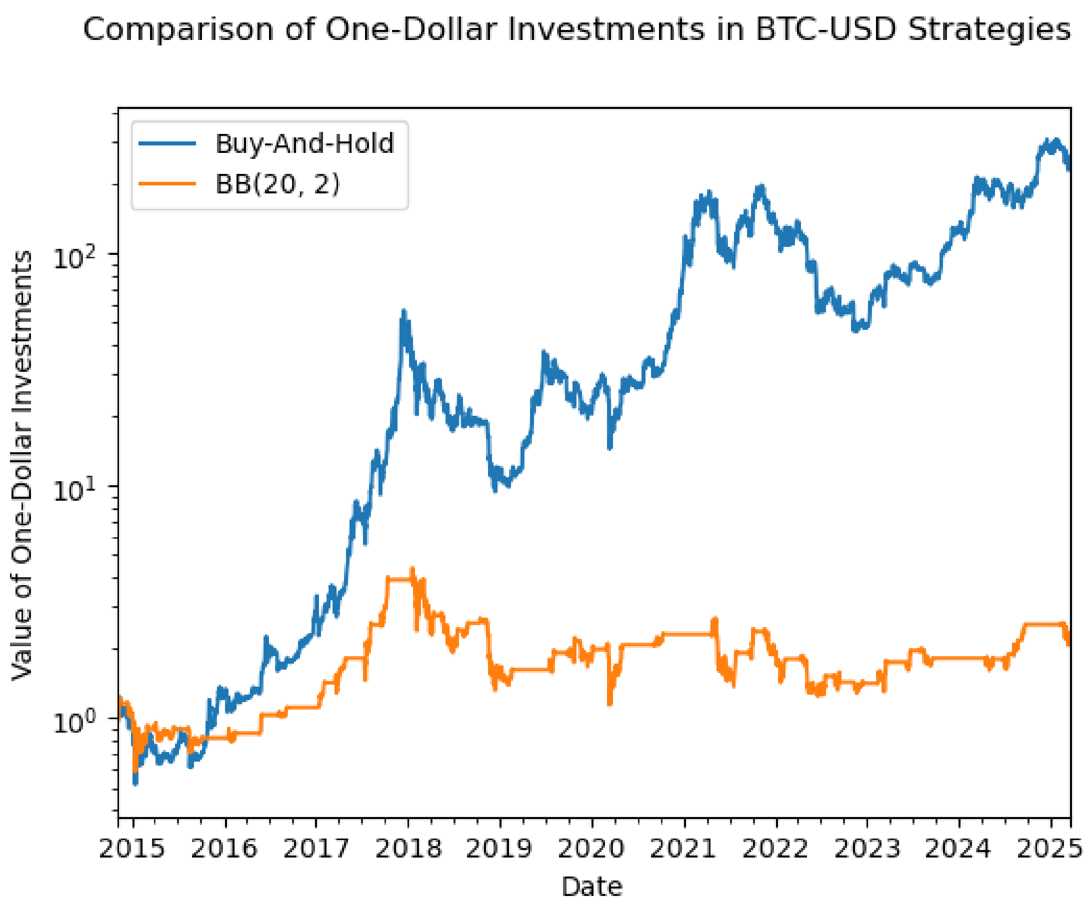
- Mean daily return is lower
- Volatility of daily returns is higher
- Every percentile of the distribution is worse

```
btc_bb.groupby('Position')['Return'].describe()
```

	count	mean	std	min	25%	50%	75%	max
Position								
0.0000	2268.0000	0.0027	0.0343	-0.1874	-0.0110	0.0019	0.0172	0.2525
1.0000	1520.0000	0.0013	0.0386	-0.3717	-0.0149	0.0007	0.0161	0.2394

```
columns_bb = {
    'Adj Close': 'Price',
    'Return': 'Buy-And-Hold',
    'Strategy': 'BB(20, 2)'
}
```

```
(
    btc_bb
    [['Return', 'Strategy']]
    .dropna()
    .rename(columns=columns_bb)
    .add(1)
    .cumprod()
    .plot()
)
plt.semilogy()
plt.suptitle('Comparison of One-Dollar Investments in BTC-USD Strategies')
plt.ylabel('Value of One-Dollar Investments')
plt.show()
```



Here are the final values of $1 investments, which are difficult to read on the log scale above.

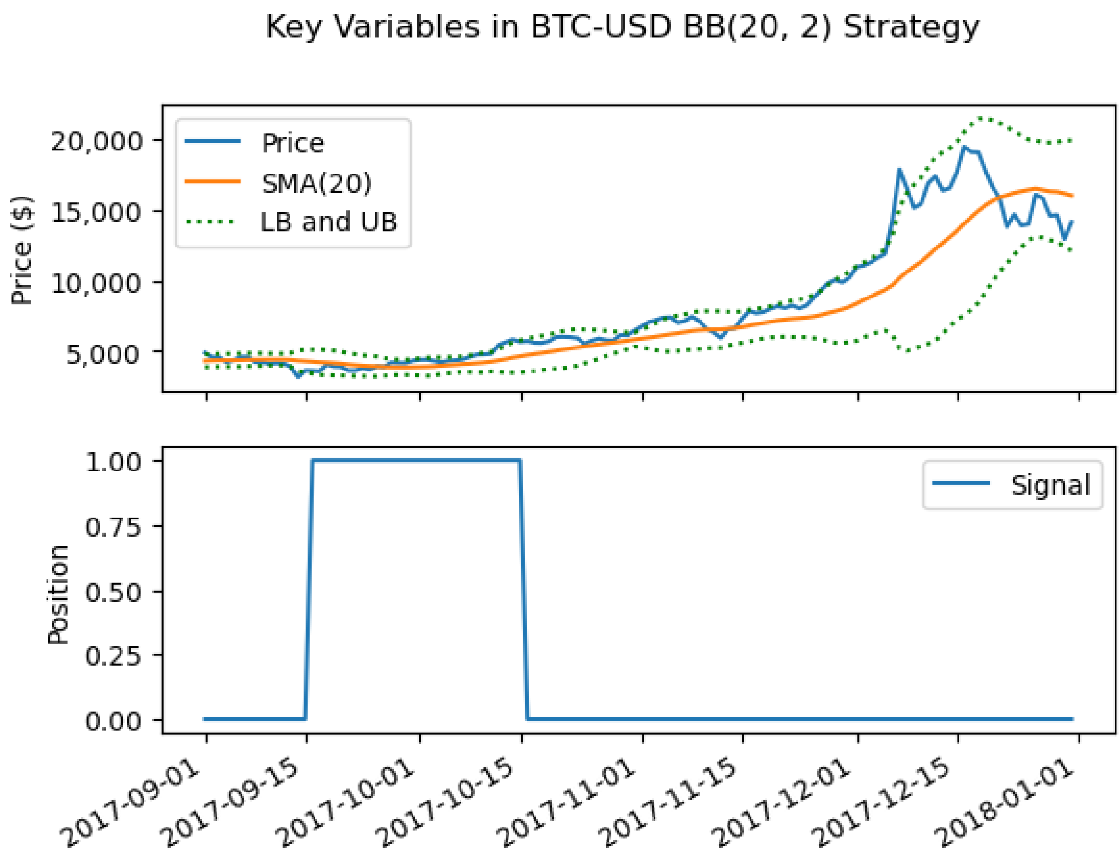
```
(
    btc_bb
    [['Return', 'Strategy']]
    .rename(columns=columns_bb)
    .add(1)
    .prod()
    .rename_axis('Strategy')
    .to_frame('Value of One-Dollar Investment')
)
```

Strategy	Value of One-Dollar Investment
Buy-And-Hold	183.6056
BB(20, 2)	2.1877

We need a more complex plot to better understand what is going on!

```
import matplotlib.ticker as ticker
```

```
fig, ax = plt.subplots(nrows=2, ncols=1, sharex=True)
df = btc_bb.loc['2017-09':'2017-12']

ax[0].plot(df[['Adj Close']], label='Price')
ax[0].plot(df[['SMA']], label='SMA(20)')
ax[0].plot(df[['UB']], label='LB and UB', color='green', linestyle=':')
ax[0].plot(df[['LB']], color='green', linestyle=':')
ax[0].legend()
ax[0].set_ylabel('Price ($)')

ax[1].plot(df[['Position']], label='Signal')
ax[1].legend()
ax[1].set_ylabel('Position')

ax[0].yaxis.set_major_formatter(ticker.StrMethodFormatter('{x:,.0f}'))
fig.autofmt_xdate()

plt.suptitle('Key Variables in BTC-USD BB(20, 2) Strategy')
plt.show()
```



Implement a long-short RSI(14) strategy with Bitcoin

From Fidelity:

> The Relative Strength Index (RSI), developed by J. Welles Wilder, is a momentum oscillator that measures the speed and change of price movements. The RSI oscillates between zero and 100. Traditionally the RSI is considered overbought when above 70 and oversold when below 30. Signals can be generated by looking for divergences and failure swings. RSI can also be used to identify the general trend.

The RSI formula: $RSI(n) = 100 - \frac{100}{1+RS(n)}$, where $RS(n) = \frac{SMA(U,n)}{SMA(D,n)}$. For "up days", $U = \Delta \text{Adj Close}$ and $D = 0$. For "down days", $U = 0$ and $D = -\Delta \text{Adj Close}$.

We will implement a long-short RSI(14) as follows:

1. Buy when the RSI crosses 30 from below, and sell when the RSI crosses 50 from below

2. Short when the RSI crosses 70 from above, and cover when the RSI crosses 50 from above

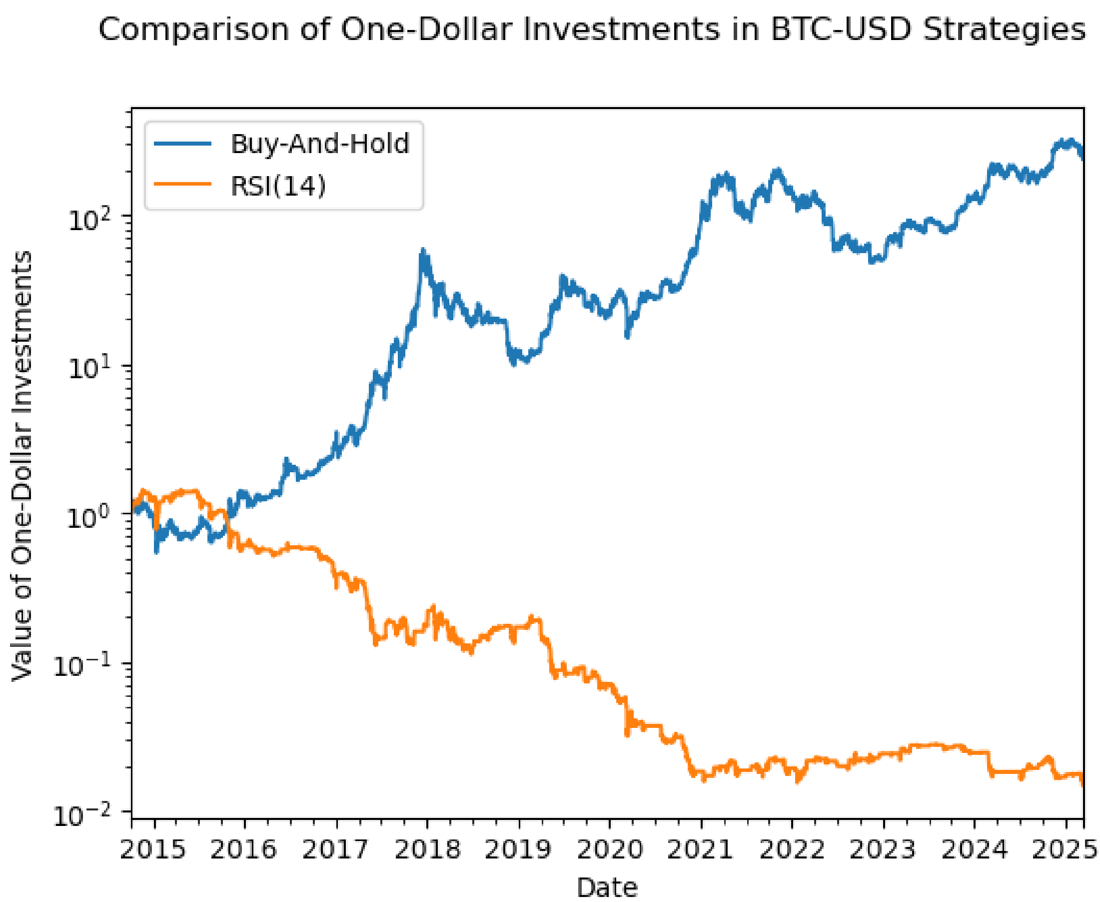
More about RSI here.

```
def calc_rsi(df, window=14, lo=30, mid=50, hi=70):
    return (
        df
        .assign(
            Return=lambda x: x['Adj Close'].pct_change(),
            # This approach with .max() and .min() handles NA values better than the in-clas
            Diff=lambda x: x['Adj Close'].diff(),
            Zero=0,
            U=lambda x: x[['Diff', 'Zero']].max(axis=1, skipna=False),
            D=lambda x: -x[['Diff', 'Zero']].min(axis=1, skipna=False),
            SMAU=lambda x: x['U'].rolling(window=window).mean(),
            SMAD=lambda x: x['D'].rolling(window=window).mean(),
            RS=lambda x: x['SMAU'] / x['SMAD'],
            RSI=lambda x: 100 - 100 / (1 + x['RS']),
            Position_w_nan=lambda x: np.select(
                condlist=[
                    (x['RSI'].shift(1) > lo) & (x['RSI'].shift(2) <= lo),
                    (x['RSI'].shift(1) > mid) & (x['RSI'].shift(2) <= mid),
                    (x['RSI'].shift(1) < hi) & (x['RSI'].shift(2) >= hi),
                    (x['RSI'].shift(1) < mid) & (x['RSI'].shift(2) >= mid),
                ],
                choicelist=[1, 0, -1, 0],
                default=np.nan
            ),
            Position=lambda x: x['Position_w_nan'].ffill(),
            Strategy=lambda x: x['Position'] * x['Return']
        )
    )
```

```
btc_rsi = btc.pipe(calc_rsi)
```

```
columns_rsi = {
    'Adj Close': 'Price',
    'Return': 'Buy-And-Hold',
    'Strategy': 'RSI(14)'
}
```

```
(
    btc_rsi
    [['Return', 'Strategy']]
    .dropna()
    .rename(columns=columns_rsi)
    .add(1)
    .cumprod()
    .plot()
)
plt.semilogy()
plt.suptitle('Comparison of One-Dollar Investments in BTC-USD Strategies')
plt.ylabel('Value of One-Dollar Investments')
plt.show()
```



Implement a golden cross with Bitcoin

Someone in Section 04 mentioned two-moving average strategies, so I added this golden cross, where the 50-day SMA crosses the 200-day SMA, to every section.

From Grok:

> In technical analysis, a golden cross is a bullish chart pattern that occurs when a short-term moving average (typically the 50-day moving average) crosses above a long-term moving average (typically the 200-day moving average). This crossover is considered a signal that a stock, index, or other asset may be entering a sustained upward trend, suggesting potential buying opportunities for traders and investors.

More here.

```
def calc_cross(df, long=200, short=50):
    return (
        df
        .assign(
            Return=lambda x: x['Adj Close'].pct_change(),
            SMAL=lambda x: x['Adj Close'].rolling(window=long).mean(),
            SMAS=lambda x: x['Adj Close'].rolling(window=short).mean(),
            Position=lambda x: np.select(
                condlist=[
                    x['SMAS'].shift(1) > x['SMAL'].shift(1),
                    x['SMAS'].shift(1) <= x['SMAL'].shift(1)
                ],
                choicelist=[1, 0],
                default=np.nan
            ),
            Strategy=lambda x: x['Position'] * x['Return']
        )
    )
```

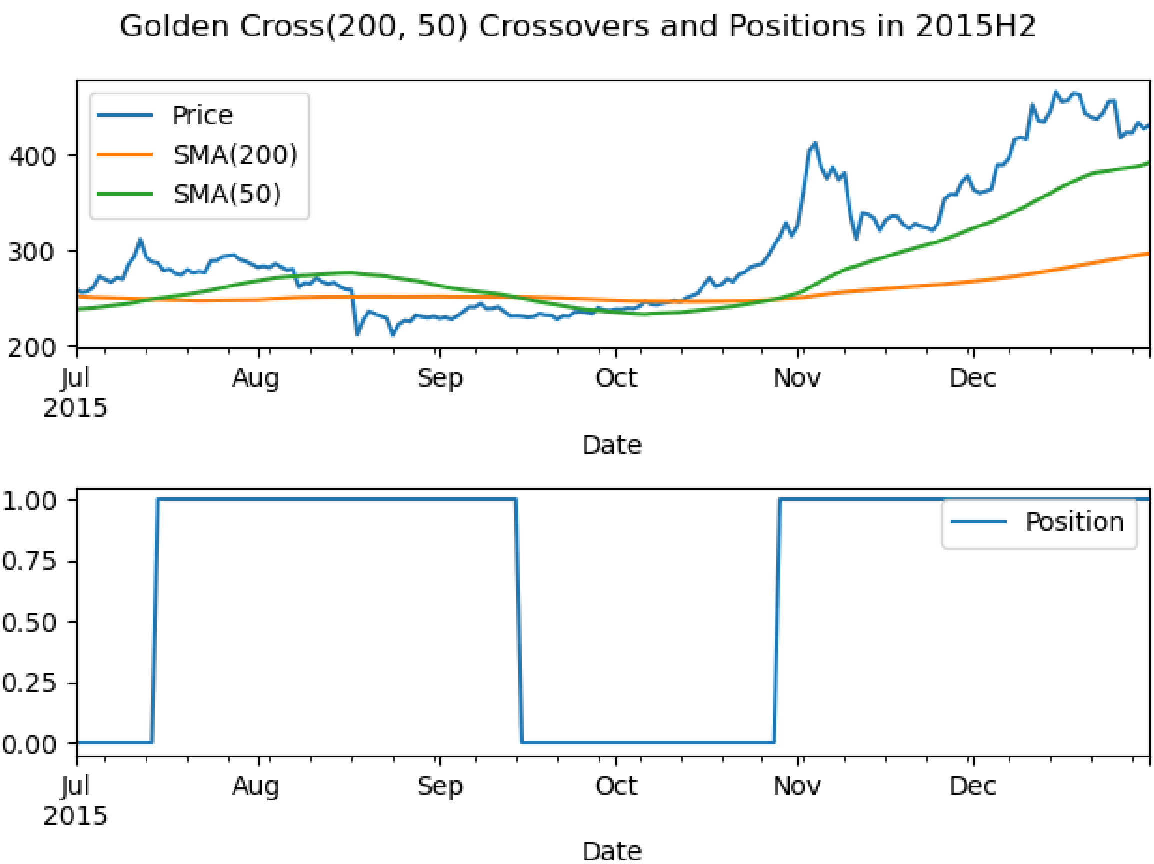
```
btc_cross = btc.pipe(calc_cross, long=200, short=50)
```

```
columns_cross = {
    'Adj Close': 'Price',
    'SMAL': 'SMA(200)',
    'SMAS': 'SMA(50)',
    'Return': 'Buy-And-Hold',
    'Strategy': 'Golden Cross(200, 50)'
}
```

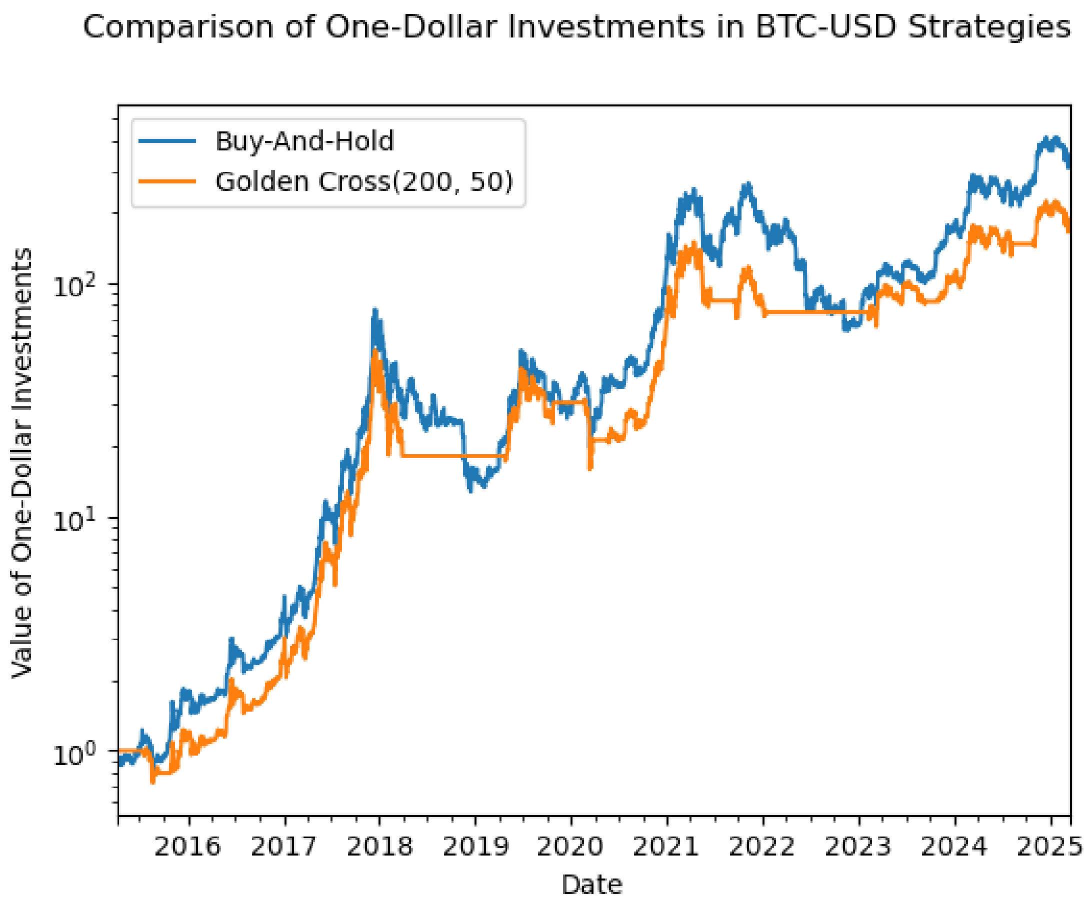
```
btc_cross.query('Position == 1')
```

Date	Adj Close	Close	High	Low	Open	Volume	Return	SMAI
2015-07-15	285.8290	285.8290	293.2480	285.3670	288.0450	27486600	-0.0057	247.78
2015-07-16	278.0890	278.0890	291.1830	275.2400	286.0420	49482600	-0.0271	247.58
2015-07-17	279.4720	279.4720	280.2800	272.0430	278.0910	27591400	0.0050	247.42
2015-07-18	274.9010	274.9010	282.5270	274.0750	279.3310	25187100	-0.0164	247.24
2015-07-19	273.6140	273.6140	275.6700	272.5130	274.7670	15332500	-0.0047	247.00
...	...	...	...	...	...	...	...	...
2025-03-10	78532.0000	78532.0000	83955.9297	77420.5938	80597.1484	54061099422	-0.0257	83357
2025-03-11	82862.2109	82862.2109	83577.7578	76624.2500	78523.8750	54702837196	0.0551	83451
2025-03-12	83722.3594	83722.3594	84358.5781	80635.2500	82857.3750	40353484454	0.0104	83549
2025-03-13	81066.7031	81066.7031	84301.6953	79931.8516	83724.9219	31412940153	-0.0317	83633
2025-03-14	83969.1016	83969.1016	85263.2891	80797.5625	81066.9922	29588112414	0.0358	83738

```
fig, ax = plt.subplots(2, 1)
df = btc_cross.loc['2015-07':'2015-12']
df[['Adj Close', 'SMAL', 'SMAS']].rename(columns=columns_cross).plot(ax=ax[0])
df[['Position']].plot(ax=ax[1])
plt.suptitle('Golden Cross(200, 50) Crossovers and Positions in 2015H2')
plt.tight_layout()
plt.show()
```




```
(
    btc_cross
    [['Return', 'Strategy']]
    .dropna()
    .rename(columns=columns_cross)
    .add(1)
    .cumprod()
    .plot()
)
plt.semilogy()
plt.suptitle('Comparison of One-Dollar Investments in BTC-USD Strategies')
plt.ylabel('Value of One-Dollar Investments')
plt.show()
```

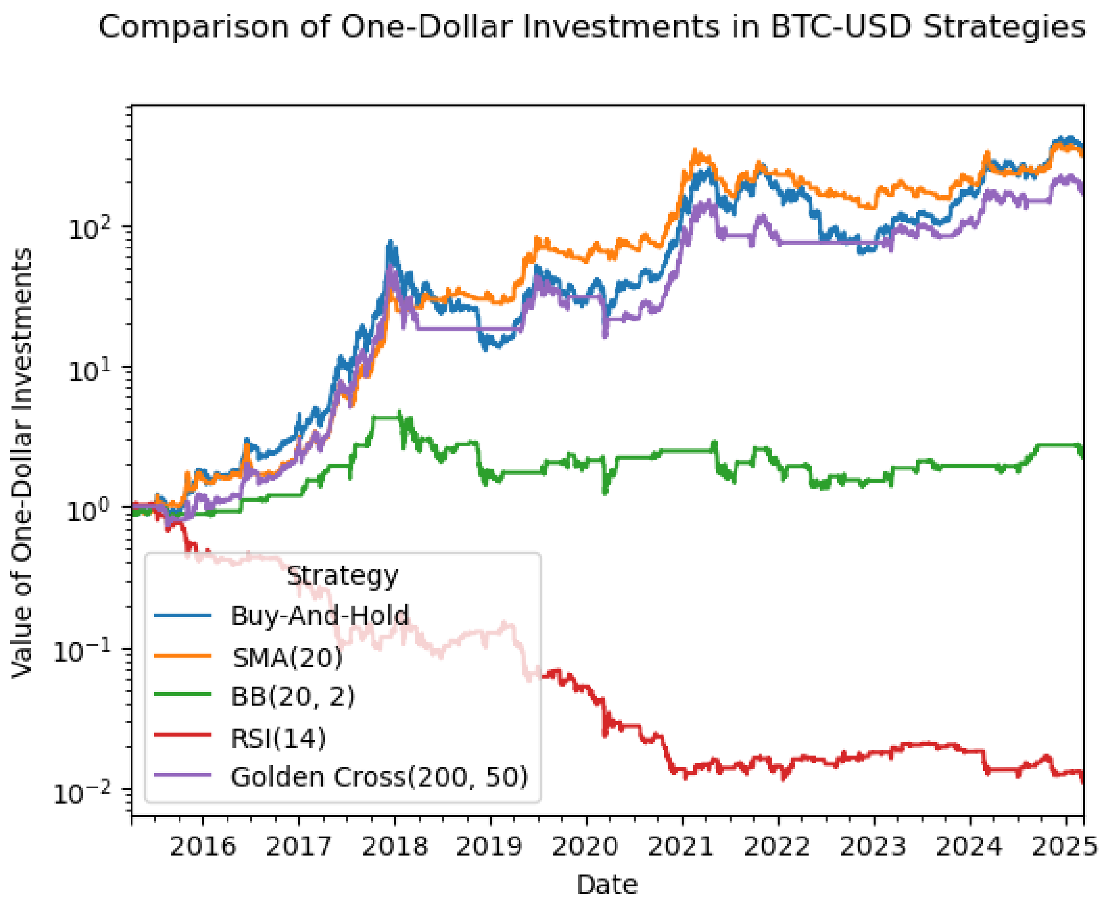


Compare all strategies

I added this comparison of all strategies after class.

```
df = (
    btc_sma[['Return', 'Strategy']]
    .join(btc_bb[['Strategy']], rsuffix='_bb')
    .join(btc_rsi[['Strategy']], rsuffix='_rsi')
    .join(btc_cross[['Strategy']], rsuffix='_cross')
    .dropna()
)



(
    df
```

```
    .add(1)
    .cumprod()
    .rename_axis(columns='Strategy')
    .rename(columns={
        'Return': 'Buy-And-Hold',
        'Strategy': 'SMA(20)',
        'Strategy_bb': 'BB(20, 2)',
        'Strategy_rsi': 'RSI(14)',
        'Strategy_cross': 'Golden Cross(200, 50)',
    })
    .plot()
)

plt.semilogy()
plt.suptitle('Comparison of One-Dollar Investments in BTC-USD Strategies')
plt.ylabel('Value of One-Dollar Investments')
plt.show()
```


Comparison of One-Dollar Investments in BTC-USD Strategies
Value of One-Dollar Investments
10^2
10^1
10^0
10^{-1}
10^{-2}
Strategy
Buy-And-Hold
SMA(20)
BB(20, 2)
RSI(14)
Golden Cross(200, 50)
2016
2017
2018
2019
2020
2021
2022
2023
2024
2025
Date

Week 10

Herron Topic 3 - Quantitative Value Investing

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import pandas_datareader as pdr
import statsmodels.api as sm
import yfinance as yf
```

```
%precision 4
pd.options.display.float_format = '{:.4f}'.format
# %config InlineBackend.figure_format = 'retina'
```

Introduction

This notebook adapts the portfolio formation approach from Fama and French (1993), who show that firms with high book-to-market equity ratios (value stocks) tend to outperform those with low ratios (growth stocks), forming the basis of the value factor in their three-factor model. This notebook relies on two large data files. You can find these files in our shared OneDrive folder or our Notebook and Syllabus folder on Canvas. These two files are:

1. Firm-level market data (e.g., stock prices and share counts) from the Center for Research in Security Prices (CRSP) in `crsp.csv`
2. Firm-level fundamentals (i.e., financial statement information) from Compustat in `compustat.csv`

This notebook walks you through the steps to test a quantitative value investing strategy based on the book-to-market equity ratio (B/M). I based this lecture notebook on Gray and Carlisle (2012, Chapters 1 and 7), which presents much more on quantitative value investing.

Import the data from CRSP and Compustat

CRSP

The CRSP data provide the following variables:

1. `PERMNO` is the stock identifier
2. `date` is the last trading day of the month
3. `SHRCD` indicates share class, and we can ignore it
4. `PRC` is the share price in dollars, and negative prices indicate the price is the midpoint of the bid-ask spread instead of a transaction
5. `SHROUT` is the number of shares outstanding in thousands

Read the CRSP data into a data frame named `crsp`. Be sure to parse dates. The codea `A`, `B`, and `C` indicate missing values.

```
crsp = (
    pd.read_csv(
        filepath_or_buffer='crsp.csv',
        parse_dates=['date'],
        na_values=list('ABC')
    )
    .query('date >= 1965')
    .sort_values(by=['PERMNO', 'date'])
    .assign(
        ME=lambda x: x['PRC'].abs() * x['SHROUT'] / 1000,
        date=lambda x: x['date'] + pd.offsets.MonthEnd(0)
    )
)
```

```
crsp.head()
```

	PERMNO	date	SHRCD	PRC	RET	SHROUT	ME
0	10000	1986-01-31	10	-4.3750	NaN	3680.0000	16.1000
1	10000	1986-02-28	10	-3.2500	-0.2571	3680.0000	11.9600
2	10000	1986-03-31	10	-4.4375	0.3654	3680.0000	16.3300
3	10000	1986-04-30	10	-4.0000	-0.0986	3793.0000	15.1720
4	10000	1986-05-31	10	-3.1094	-0.2227	3793.0000	11.7939

Compustat

The Compustat data provide the following variables:

1. `GVKEY` is the S&P firm identifier
2. `LPERMNO` is the link between CRSP and Compustat
3. `datadate` is the date of the filing, which is the last day of the fiscal year
4. `fyear` is the fiscal year
5. `ceq` is the book value of equity
6. `ib` is income before extraordinary items
7. We can ignore the other variables that Wharton Research Data Services (WRDS) adds to this data set

Read the Compustat data into a data frame named `compustat`. Be sure to parse dates.

```
compustat = (
    pd.read_csv(
        filepath_or_buffer='compustat.csv',
        parse_dates=['datadate']
    )
    .query('datadate >= 1965')
    .sort_values(['LPERMNO', 'fyear', 'datadate'])
    .drop_duplicates(subset=['LPERMNO', 'fyear'], keep='last')
)
```

```
compustat.head()
```

	GVKEY	LPERMNO	datadate	fyear	indfmt	consol	popsrc	datafmt	curcd	ceq
165681	13007	10000	1986-10-31	1986.0000	INDL	C	D	STD	USD	0.418
165586	12994	10001	1986-06-30	1986.0000	INDL	C	D	STD	USD	5.432
165587	12994	10001	1987-06-30	1987.0000	INDL	C	D	STD	USD	5.369
165588	12994	10001	1988-06-30	1988.0000	INDL	C	D	STD	USD	5.512
165589	12994	10001	1989-06-30	1989.0000	INDL	C	D	STD	USD	6.321

Create interim data frames

To test the book/market investing strategy, we need to match the following data:

1. Book value of equity for year $t-1$
2. Market value of equity from December of year $t-1$

3. Portfolios for year t based on book and market values of equity in year $t-1$
4. Returns from July of year t through June of year $t+1$

We will form portfolios once a year because the book value of equity updates once a year. Fama and French (1993) wait until June of year t to form portfolios, ensuring financial data from year $t-1$ is public, then track returns from July in year t through June in year $t+1$. The simplest and clearest way to match these data is to create one data frame for each of these three data sets.

Market value of equity from December of year $t-1$

Then create a new data frame `mve` that contains the latest market value of equity as of December.

```
mve = (
    crsp
    .query('ME > 0')
    .groupby(['PERMNO', pd.Grouper(key='date', freq='YE-DEC')])
    [['ME']]
    .last()
)
```

```
crsp.query('PERMNO == 10001').head(12)
```

	PERMNO	date	SHRCD	PRC	RET	SHROUT	ME
18	10001	1986-01-31	11	-6.1250	NaN	985.0000	6.0331
19	10001	1986-02-28	11	-6.2500	0.0204	985.0000	6.1562
20	10001	1986-03-31	11	-6.3125	0.0252	985.0000	6.2178
21	10001	1986-04-30	11	-6.3750	0.0099	985.0000	6.2794
22	10001	1986-05-31	11	-6.3125	-0.0098	985.0000	6.2178
23	10001	1986-06-30	11	-6.1250	-0.0131	985.0000	6.0331
24	10001	1986-07-31	11	-6.0625	-0.0102	985.0000	5.9716
25	10001	1986-08-31	11	-6.5000	0.0722	985.0000	6.4025
26	10001	1986-09-30	11	6.3750	-0.0031	991.0000	6.3176
27	10001	1986-10-31	11	6.6250	0.0392	991.0000	6.5654
28	10001	1986-11-30	11	7.0000	0.0566	991.0000	6.9370
29	10001	1986-12-31	11	7.0000	0.0150	991.0000	6.9370

```
mve.head()
```

PERMNO	date	M
10000	1986-12-31	1.
	1987-12-31	0.
10001	1986-12-31	6.
	1987-12-31	5.
	1988-12-31	6.

Book value of equity for year $t - 1$

Create a new data frame `bve` that contains the latest book value of equity as of December. Because some firms have fiscal years that do not end in December, use `.groupby()`, `pd.Grouper()` with `YE-DEC`, and `.last()` to roll forward earlier fiscal year ends.

```
bve = (
    compustat
    .query('ceq > 0')
    .groupby(by=['LPERMNO', pd.Grouper(key='datadate', freq='YE-DEC')])
    [['ceq']]
    .last()
)
```

```
compustat.query('LPERMNO == 10001').head()
```

	GVKEY	LPERMNO	datadate	fyear	indfmt	consol	popsrc	datafmt	curcd	ceq
165586	12994	10001	1986-06-30	1986.0000	INDL	C	D	STD	USD	5.432
165587	12994	10001	1987-06-30	1987.0000	INDL	C	D	STD	USD	5.369
165588	12994	10001	1988-06-30	1988.0000	INDL	C	D	STD	USD	5.512
165589	12994	10001	1989-06-30	1989.0000	INDL	C	D	STD	USD	6.321
165590	12994	10001	1990-06-30	1990.0000	INDL	C	D	STD	USD	7.179

```
bve.head()
```

LPERMNO	datadate	ce
10000	1986-12-31	0.
	1986-12-31	5.
10001		

LPERMNO	datadate	ce
	1987-12-31	5.
	1988-12-31	5.
	1989-12-31	6.

Portfolios for year t

```
1 + pd.qcut(x=np.arange(10), q=5, labels=False)
```

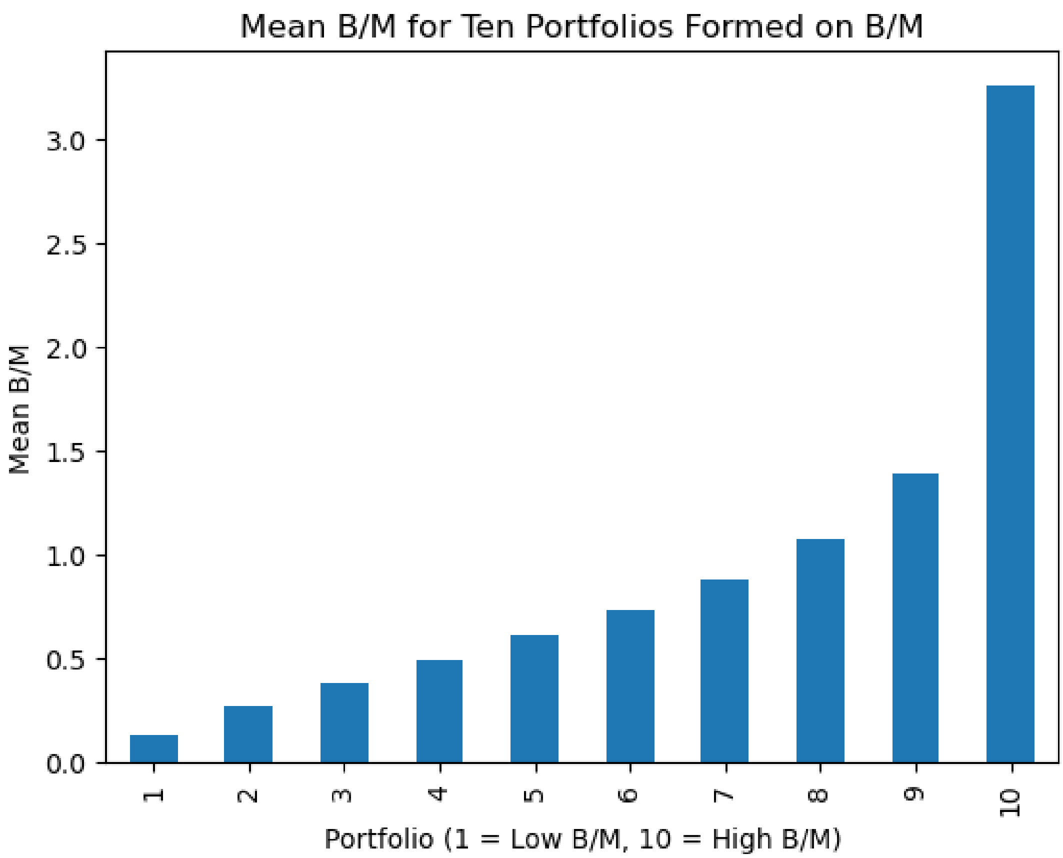
```
array([1, 1, 2, 2, 3, 3, 4, 4, 5, 5])
```

```
portfolios = (
    bve
    [['ceq']]
    .dropna()
    .rename_axis(index=['PERMNO', 'Date'])
    .join(
        other=(
            mve
            [['ME']]
            .dropna()
            .rename_axis(index=['PERMNO', 'Date'])
        ),
        how='inner'
    )
    .assign(BM=lambda x: x['ceq'] / x['ME'])
    .reset_index()
    .assign(
        Portfolio=lambda x: x.groupby('Date')['BM'].transform(lambda x: 1 + pd.qcut(x=x, q=1
        Date=lambda x: x['Date'] + pd.DateOffset(months=7)
    )
    [['PERMNO', 'Date', 'Portfolio', 'BM']]
)
```

```
portfolios.head()
```

	PERMNO	Date	Portfolio	BM
0	10000	1987-07-31	2	0.2109
1	10001	1987-07-31	7	0.7830
2	10001	1988-07-31	7	0.9212
3	10001	1989-07-31	7	0.8664
4	10001	1990-07-31	5	0.6109

```python
(
    portfolios
    .groupby(by=['Portfolio', 'Date'])
    ['BM']
    .mean()
    .groupby(by=['Portfolio'])
    .mean()
    .plot(kind='bar')
)
plt.ylabel('Mean B/M')
plt.xlabel('Portfolio (1 = Low B/M, 10 = High B/M)')
plt.title('Mean B/M for Ten Portfolios Formed on B/M')
plt.show()
```



Returns from July of year t to June of year $t + 1$

Now we assign returns to each of these ten portfolios.

```
returns = (
    pd.merge_asof(
        left=crsp.sort_values(by=['date', 'PERMNO']),
        right=portfolios.sort_values(by=['Date', 'PERMNO']),
        left_on='date',
        right_on='Date',
        by='PERMNO',
        tolerance=pd.Timedelta('366 days')
    )
    [['PERMNO', 'date', 'RET', 'Portfolio']]
    .dropna()
    .assign(Portfolio=lambda x: x['Portfolio'].astype(int))
```

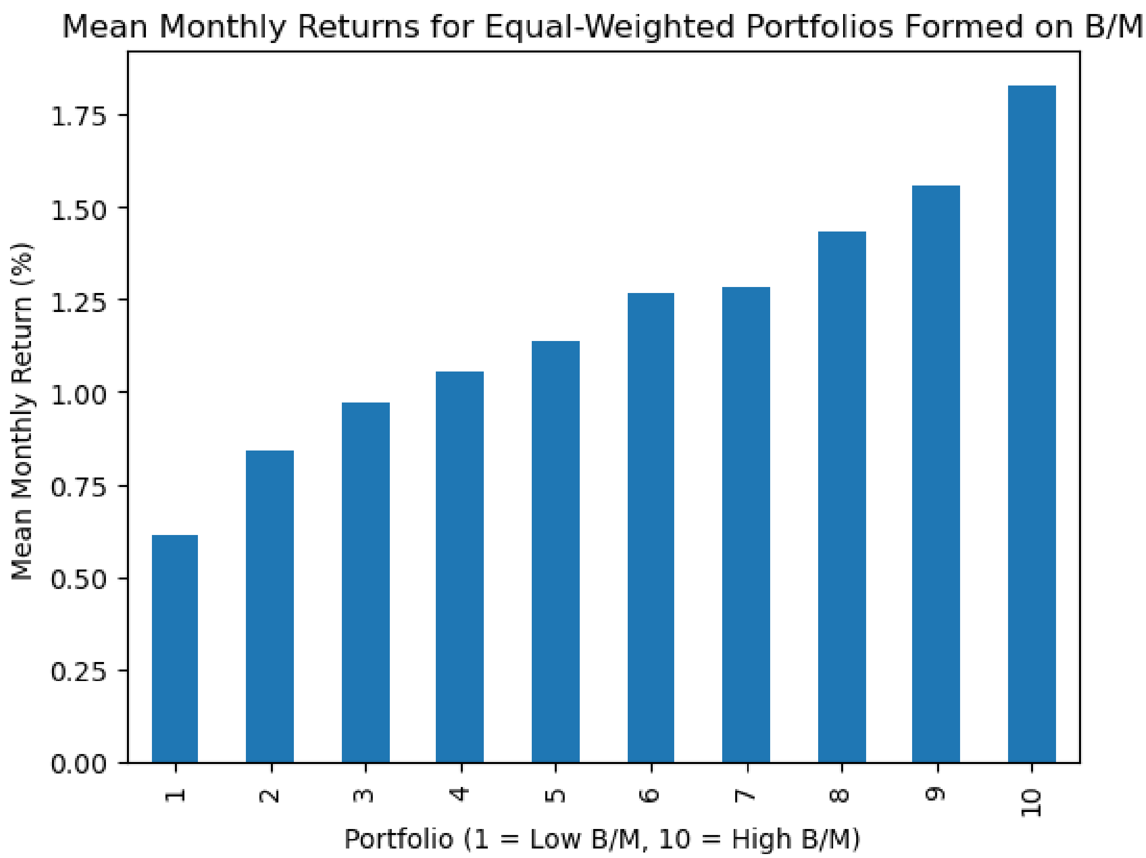
```
    .rename(columns={'date': 'Date'})
)
```

```
returns
```

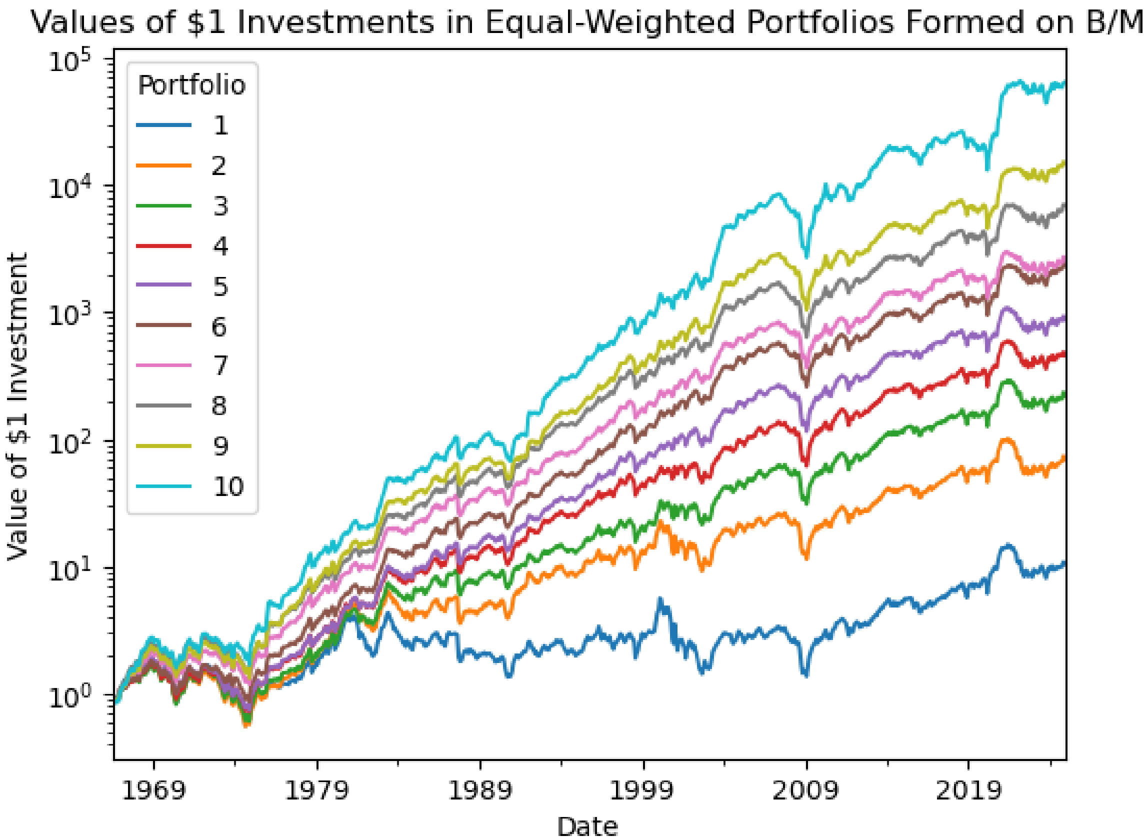
	PERMNO	Date	RET	Portfolio
37636	10006	1966-07-31	-0.0149	6
37640	10102	1966-07-31	-0.0545	6
37641	10137	1966-07-31	0.0105	4
37642	10145	1966-07-31	-0.0129	6
37643	10153	1966-07-31	-0.0788	9
...	...	...	...	...
3457680	93397	2024-12-31	-0.1174	5
3457681	93426	2024-12-31	0.0218	7
3457682	93429	2024-12-31	-0.0947	3
3457683	93434	2024-12-31	0.1333	10
3457684	93436	2024-12-31	0.1700	1

Performance of Portfolios Formed on B/M

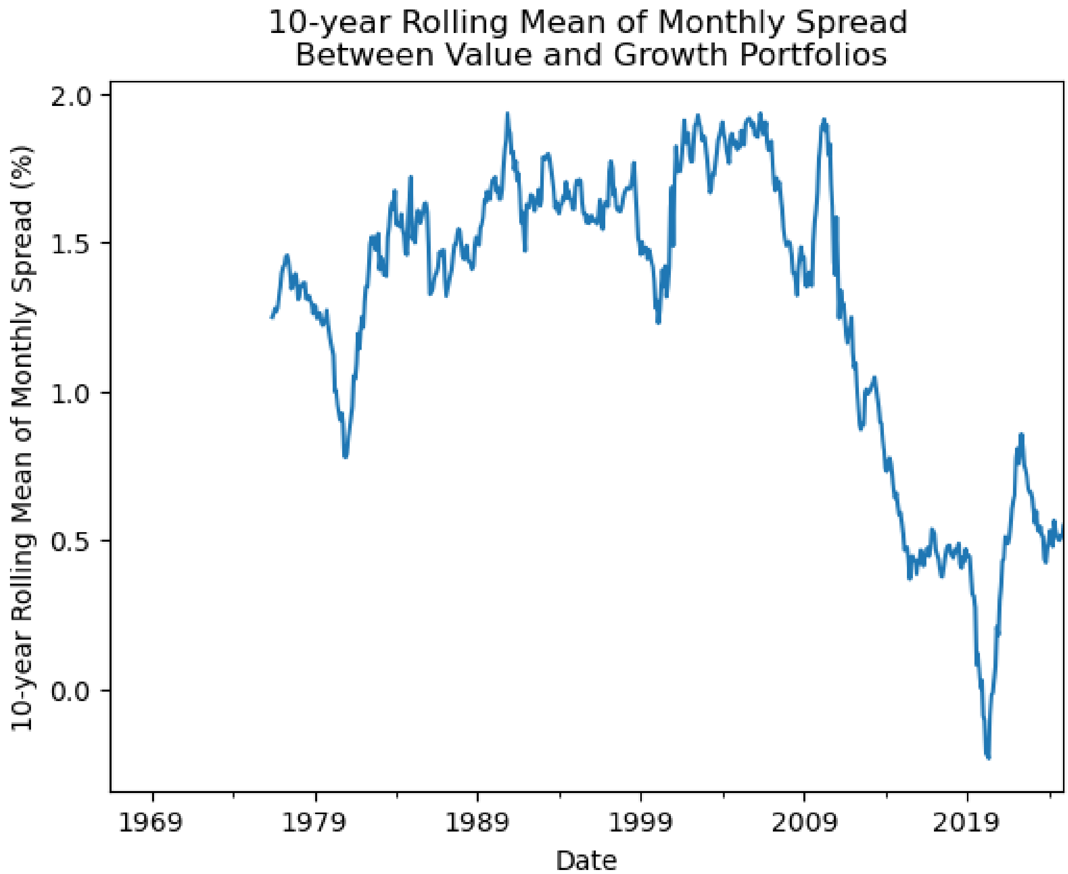
```
(
    returns
    .groupby(['Portfolio', 'Date'])
    ['RET']
    .mean()
    .groupby(['Portfolio'])
    .mean()
    .mul(100)
    .plot(kind='bar')
)
plt.ylabel('Mean Monthly Return (%)')
plt.xlabel('Portfolio (1 = Low B/M, 10 = High B/M)')
plt.title('Mean Monthly Returns for Equal-Weighted Portfolios Formed on B/M')
plt.show()
```




```
(
    returns
    .groupby(['Portfolio', 'Date'])
    ['RET']
    .mean()
    .unstack('Portfolio')
    .add(1)
    .cumprod()
    .plot()
)
plt.semilogy()
plt.ylabel('Value of $1 Investment')
plt.title('Values of $1 Investments in Equal-Weighted Portfolios Formed on B/M')
plt.show()
```




```
(
    returns
    .groupby(['Portfolio', 'Date'])
    ['RET']
    .mean()
    .unstack('Portfolio')
    .assign(Diff=lambda x: x[10] - x[1])
    ['Diff']
    .rolling(12*10)
    .mean()
    .mul(100)
    .plot()
)
plt.ylabel('10-year Rolling Mean of Monthly Spread (%)')
plt.title('10-year Rolling Mean of Monthly Spread\n Between Value and Growth Portfolios')
plt.show()
```


10-year Rolling Mean of Monthly Spread
Between Value and Growth Portfolios
10-year Rolling Mean of Monthly Spread (%)
2.0
1.5
1.0
0.5
0.0
1969
1979
1989
1999
2009
2019
Date

Herron Topic 3 - Practice - Blank

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import pandas_datareader as pdr
import statsmodels.api as sm
import yfinance as yf
```

```
%precision 4
pd.options.display.float_format = '{:.4f}'.format
# %config InlineBackend.figure_format = 'retina'
```

Announcements

Five-Minute Review

Practice

Re-implement the value strategy from the lecture notebook

Read the data

Create the interim data frames

Combine the data frames and form portfolios

Backtest the strategy

Re-implement the value strategy from the lecture notebook *with value-weighted portfolios*

Estimate the αs of the equal-weighted and value-weighted portfolios

Implement a momentum strategy

Form deciles on the 11-month returns from months $t - 12$ to month $t - 1$.

Herron Topic 3 - Practice - Sec 02

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import pandas_datareader as pdr
import statsmodels.api as sm
import yfinance as yf
```

```
%precision 4
pd.options.display.float_format = '{:.4f}'.format
# %config InlineBackend.figure_format = 'retina'
```

Announcements

1. Projects:
    1. Project 1:
        1. I am still grading
        2. I plan to finish grading by Friday
    2. Project 2:
        1. Due on Friday, 3/28, at 11:59 PM
        2. We will use class time next week for group work
        3. Ask your questions here: https://northeastern.instructure.com/courses/207607/discussion_topics/2738173
2. Assessments:
    1. ***Both assessments are in class on Tuesday, 4/15***
    2. Programming assessment: Should take 15 minutes and will be based on the questions here:
    3. MSFQ assessment (only for MSFQ students): Should take 45 minutes and based on the 5 required MSFQ courses (standby for a Canvas announcement with more details after I discuss it with the MSFQ program director)

Five-Minute Review

Quantitative value investing is a systematic investment strategy that combines the principles of value investing with data-driven, quantitative techniques. It involves identifying undervalued securities—those trading below their intrinsic value—using predefined, measurable criteria rather than subjective judgment. Common criteria are price/earnings (P/E) and price/book (P/B) ratios. Here, we will use the book-to-market equity ratio (B/M), which is common in the academic literature and quantitative investing.

The key concepts in this topic are creating metrics, forming portfolios, and backtesting the strategy.

Practice

Re-implement the value strategy from the lecture notebook

Read the data

This code reads stock market data from a file called `crsp.csv` and prepares it for analysis. It loads the data into a data frame, makes sure the dates are in the right format, and marks certain letters (`A`, `B`, and `C`) as missing values. Then, it filters the data to include only records from 1965 or later, sorts it by company and date, and adds two new columns: one for market equity (calculated as the number of shares times the stock price, converted to millions), and another to adjust all dates to the end of each month.

```
crsp = (
    pd.read_csv(
        filepath_or_buffer='crsp.csv',  # Read data from crsp.csv file
        parse_dates=['date'],           # Convert 'date' column to datetime format
        na_values=['A', 'B', 'C']       # Treat 'A', 'B', 'C' as missing values (NaN)
    )
    .query('date >= 1965')              # Filter to keep only data from 1965 onward
    .sort_values(['PERMNO', 'date'])    # Sort by company identifier (PERMNO) and date
    .assign(
        ME=lambda x: x['SHROUT'] * x['PRC'].abs() / 1_000,  # Calculate market equity (ME) as
        date=lambda x: x['date'] + pd.offsets.MonthEnd(0)   # Adjust dates to the last day of
    )
)
```

```
crsp.head()
```

	PERMNO	date	SHRCD	PRC	RET	SHROUT	ME
0	10000	1986-01-31	10	-4.3750	NaN	3680.0000	16.1000
1	10000	1986-02-28	10	-3.2500	-0.2571	3680.0000	11.9600
2	10000	1986-03-31	10	-4.4375	0.3654	3680.0000	16.3300
3	10000	1986-04-30	10	-4.0000	-0.0986	3793.0000	15.1720
4	10000	1986-05-31	10	-3.1094	-0.2227	3793.0000	11.7939

This code brings in financial data from a file named `compustat.csv` and prepares it for analysis. It loads the data into a data frame, formats dates, and filters out any records before 1965. Then, it sorts the data by company, fiscal year, and date. Finally, it removes any duplicate entries for the same company and fiscal year, keeping only the most recent record.

```
compustat = (
    pd.read_csv(
        filepath_or_buffer='compustat.csv',  # Read data from compustat.csv file
        parse_dates=['datadate']              # Convert 'datadate' column to datetime format
    )
    .query('datadate >= 1965')                # Filter to keep only data from 1965 onward
    .sort_values(['LPERMNO', 'fyear', 'datadate'])  # Sort by company identifier (LPERMNO), f
    .drop_duplicates(subset=['LPERMNO', 'fyear'], keep='last')  # Keep only the latest recor
)
```

```
compustat.head()
```

	GVKEY	LPERMNO	datadate	fyear	indfmt	consol	popsrc	datafmt	curcd	ceq
165681	13007	10000	1986-10-31	1986.0000	INDL	C	D	STD	USD	0.418
165586	12994	10001	1986-06-30	1986.0000	INDL	C	D	STD	USD	5.432
165587	12994	10001	1987-06-30	1987.0000	INDL	C	D	STD	USD	5.369
165588	12994	10001	1988-06-30	1988.0000	INDL	C	D	STD	USD	5.512
165589	12994	10001	1989-06-30	1989.0000	INDL	C	D	STD	USD	6.321

About 64 percent of firm-years have December fiscal-year ends.

```
compustat['datadate'].dt.month.value_counts() / compustat['datadate'].shape[0]
```

```
datadate
12   0.6401
6    0.0723
```

```
9     0.0609
3     0.0518
10    0.0341
1     0.0334
8     0.0203
7     0.0187
5     0.0187
4     0.0179
11    0.0173
2     0.0146
Name: count, dtype: float64
```

Create the interim data frames

This code finds the market value of equity (`ME`) as of December each year from the `crsp` data frame (to match with book value of equity `ceq` from the `compustat` data frame below). It filters out any rows where market equity is zero or negative, then sorts the data by company and date. Next, it groups the data by company and year, focusing on December values. Finally, it selects the last market equity value for each company in each year, giving us the December `ME`.

```
mve = (
    crsp
    .query('ME > 0')                              # Filter for positive market equity values
    .sort_values(['PERMNO', 'date'])              # Sort by company identifier (PERMNO) and da
    .groupby(by=['PERMNO', pd.Grouper(key='date', freq='YE-DEC')])  # Group by company and ye
    [['ME']]                                      # Select the market equity (ME) column
    .last()                                       # Take the last ME value for each group (Dec
)
```

```
mve.head()
```

		M
PERMNO	date	
10000	1986-12-31	1.
	1987-12-31	0.
10001	1986-12-31	6.
	1987-12-31	5.
	1988-12-31	6.

This code finds the book value of equity (`ceq`) as of December each year from the `compustat` data frame. It filters out any rows where common equity is zero or negative, then sorts the data by company and date. Next, it groups the data by company and year, focusing on December values. Finally, it selects the last common equity value for each company in each year, giving us the December `ceq`.

```
bve = (
    compustat
    .query('ceq > 0')                                # Filter for positive common equity values
    .sort_values(['LPERMNO', 'datadate'])            # Sort by company identifier (LPERMNO) and d
    .groupby(by=['LPERMNO', pd.Grouper(key='datadate', freq='YE-DEC')])  # Group by company a
    [['ceq']]                                        # Select the common equity (ceq) column
    .last()                                          # Take the last CEQ value for each group (De
)
```

```
bve.head()
```

		ce
LPERMNO	datadate	
10000	1986-12-31	0.
10001	1986-12-31	5.
	1987-12-31	5.
	1988-12-31	5.
	1989-12-31	6.

Combine the data frames and form portfolios

This code shows a simple example of how to form portfolios using the `pd.qcut()` function. It takes a sequence of numbers from 0 to 9 and splits them into 2 equal groups (quantiles), assigning them to either a low or high category. The `labels=False` argument gives us numeric group identifiers (0 or 1), and adding 1 shifts these to 1 or 2, which can represent portfolio numbers.

```
1 + pd.qcut(x=np.arange(10), q=2, labels=False)
```

```
array([1, 1, 1, 1, 1, 2, 2, 2, 2, 2])
```

> **Note**
>
> In class, in the following cell, I used multiple cursors to simultaneously type two `.rename_axis(index=['PERMNO', 'Date'])` methods. If you hold down the `CTRL` key, every mouse click will generate a new cursor. Each cursor will do the same action (e.g., typing, highlighting, and deleting). More here.

This code creates portfolios based on book-to-market (`BM`) ratios using the `bve` and `mve` data frames. It starts by aligning the company and date indexes of both datasets and merges them, keeping only rows with matching data. Next, it calculates the `BM` ratio by dividing book value of equity (`ceq`) by market value of equity (`ME`). After resetting the index, it shifts the dates forward by 7 months to time the portfolio formation correctly, giving 6 months between `BM` calculation and buying stocks at the end of June to receive July returns. Finally, for each date, it sorts the `BM` ratios into 10 equal groups (deciles) and assigns portfolio numbers from 1 to 10.

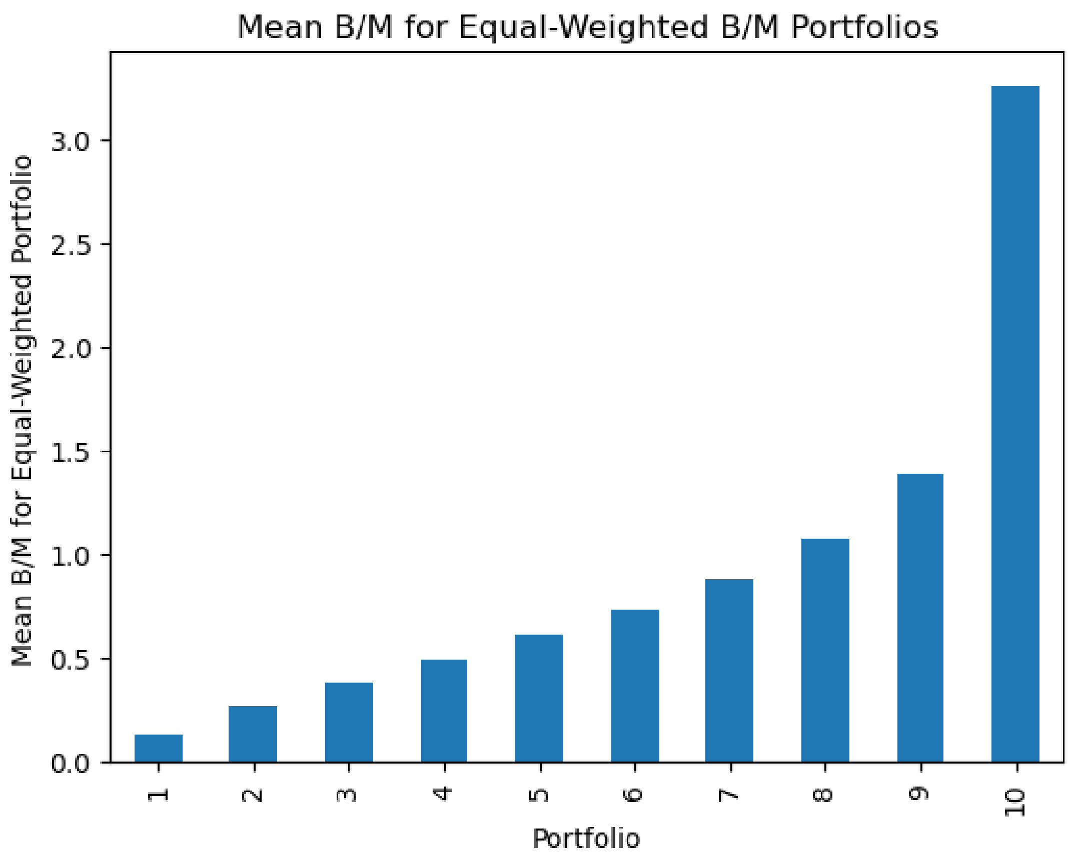
```
bm = (
    bve
    .rename_axis(index=['PERMNO', 'Date'])  # Standardize index names to PERMNO (company ID)
    .join(
        other=mve.rename_axis(index=['PERMNO', 'Date']),  # Combine with market equity data,
        how='inner'                                       # Use inner join to retain only records wi
    )
    .reset_index()                                        # Flatten the index into columns for easie
    .assign(
        Date=lambda x: x['Date'] + pd.offsets.MonthEnd(7),  # Shift dates forward 7 months (
        BM=lambda x: x['ceq'] / x['ME'],                  # Compute book-to-market ratio as
        Portfolio=lambda x: x.groupby('Date')['BM'].transform(  # Assign stocks to portfolios
            lambda x: 1 + pd.qcut(x=x, q=10, labels=False)  # Divide BM into 10 quantiles (0
        )
    )
)
```

```
bm.head()
```

	PERMNO	Date	ceq	ME	BM	Portfolio
0	10000	1987-07-31	0.4180	1.9816	0.2109	2
1	10001	1987-07-31	5.4320	6.9370	0.7830	7
2	10001	1988-07-31	5.3690	5.8280	0.9212	7
3	10001	1989-07-31	5.5120	6.3623	0.8664	7
4	10001	1990-07-31	6.3210	10.3477	0.6109	5

This code creates a bar chart showing the average book-to-market (`BM`) ratio for each portfolio using the `portfolios` data frame. It groups the data by portfolio number (1 to 10), calculates the mean `BM` ratio for each group, and then plots these averages as bars. As expected, we see that `BM` rises from portfolio 1 to 10.

```
(
    bm
    .groupby(by=['Portfolio', 'Date'])  # Group data by portfolio number and date
    ['BM']                              # Select the book-to-market (BM) column
    .mean()                             # Calculate the mean BM for each portfolio-date combi
    .groupby(by='Portfolio')            # Group again by portfolio to average across all date
    .mean()                             # Calculate the time-averaged mean BM for each portfo
    .plot(kind='bar')                   # Plot the results as a bar chart
)
plt.ylabel('Mean B/M for Equal-Weighted Portfolio')
plt.title('Mean B/M for Equal-Weighted B/M Portfolios')
plt.show()
```



This code combines monthly stock returns from the `crsp` data frame with the *most recent* portfolio assignments from the `bm` data frame. The `tolerance` argument tells the merge to only use portfolio assignments within 366 days to avoid stale data.

```
stocks = (
    pd.merge_asof(
        left=crsp.sort_values(['date', 'PERMNO']),    # Sort CRSP data by date and company II
        right=bm.sort_values(['Date', 'PERMNO']),     # Sort BM data by date and company ID f
        left_on='date',                               # Match CRSP dates to the closest prio
        right_on='Date',                              # Use BM dates as the reference for the
        by='PERMNO',                                  # Merge on company ID (PERMNO) to link
        tolerance=pd.Timedelta('366d')                # Limit matches to portfolio assignmen
    )
    [['PERMNO', 'date', 'Portfolio', 'RET']]          # Select key columns: company ID, date
    .dropna()                                         # Remove rows with missing values (e.g
    .assign(Portfolio=lambda x: x['Portfolio'].astype(int))  # Convert portfolio numbers to i
    .rename(columns={'date': 'Date'})                 # Rename 'date' column to 'Date' for co
)
```

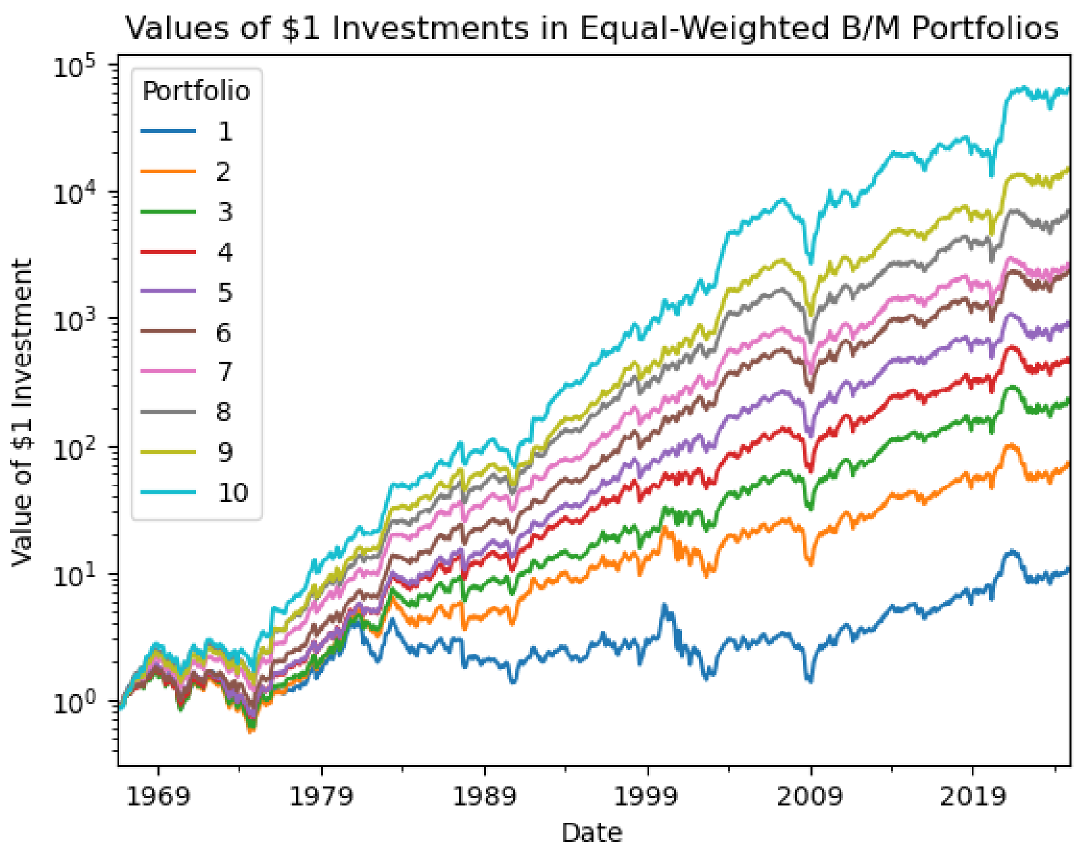
This code creates equal-weighted portfolios using the `stocks` data frame. It groups the data by portfolio number (`Portfolio`) and date (`Date`), then calculates the mean return (`RET`) for each group. This averaging process gives each stock within a portfolio the same weight, producing equal-weighted portfolio returns.

> **Note**
>
> Here I made `portfolios` a wide data frame (i.e., portfolio numbers in columns) to simplify plots and calculations. This is a difference from class

```
portfolios = (
    stocks
    .groupby(['Portfolio', 'Date'])  # Group stocks by portfolio number (1-10) and trading da
    [['RET']]                        # Isolate the returns column for portfolio-level aggrega
    .mean()                          # Average returns across stocks within each portfolio-da
    .unstack('Portfolio')            # Pivot the data to create columns for each portfolio's
    ['RET']                          # Extract the returns data, retaining portfolio numbers
)
```

Finally, we can see that the value portfolio 10 outperforms the growth portfolio 1!

```
portfolios.add(1).cumprod().plot()
plt.semilogy()
plt.ylabel('Value of $1 Investment')
plt.title('Values of $1 Investments in Equal-Weighted B/M Portfolios')
plt.show()
```



Backtest the strategy

We will need the Fama and French (1993) factors for the risk-free rate.

> **Note**
>
> Here I set the `Date` index to simplify merges/joins below. This is a difference from class

```
ff3 = (
    pdr.DataReader(
        name='F-F_Research_Data_Factors',
        data_source='famafrench',
        start='1900'
    )
    [0]
    .div(100)
    .reset_index()
    .assign(Date=lambda x: x['Date'].dt.to_timestamp(how='end').dt.normalize()) # convert mor
    .set_index('Date')
)
```

```
C:\Users\r.herron\AppData\Local\Temp\ipykernel_20756\1889697072.py:2: FutureWarning: The argu
  pdr.DataReader(
C:\Users\r.herron\AppData\Local\Temp\ipykernel_20756\1889697072.py:2: FutureWarning: The argu
  pdr.DataReader(
```

```
portfolios_rf = portfolios.sub(ff3['RF'], axis=0)
```

```
def CAGR(x, ann_fac=12):
    return (1 + x).prod() ** (ann_fac / x.count()) - 1
```

```
def Sharpe(x, ann_fac=np.sqrt(12)):
    return ann_fac * x.mean() / x.std()
```

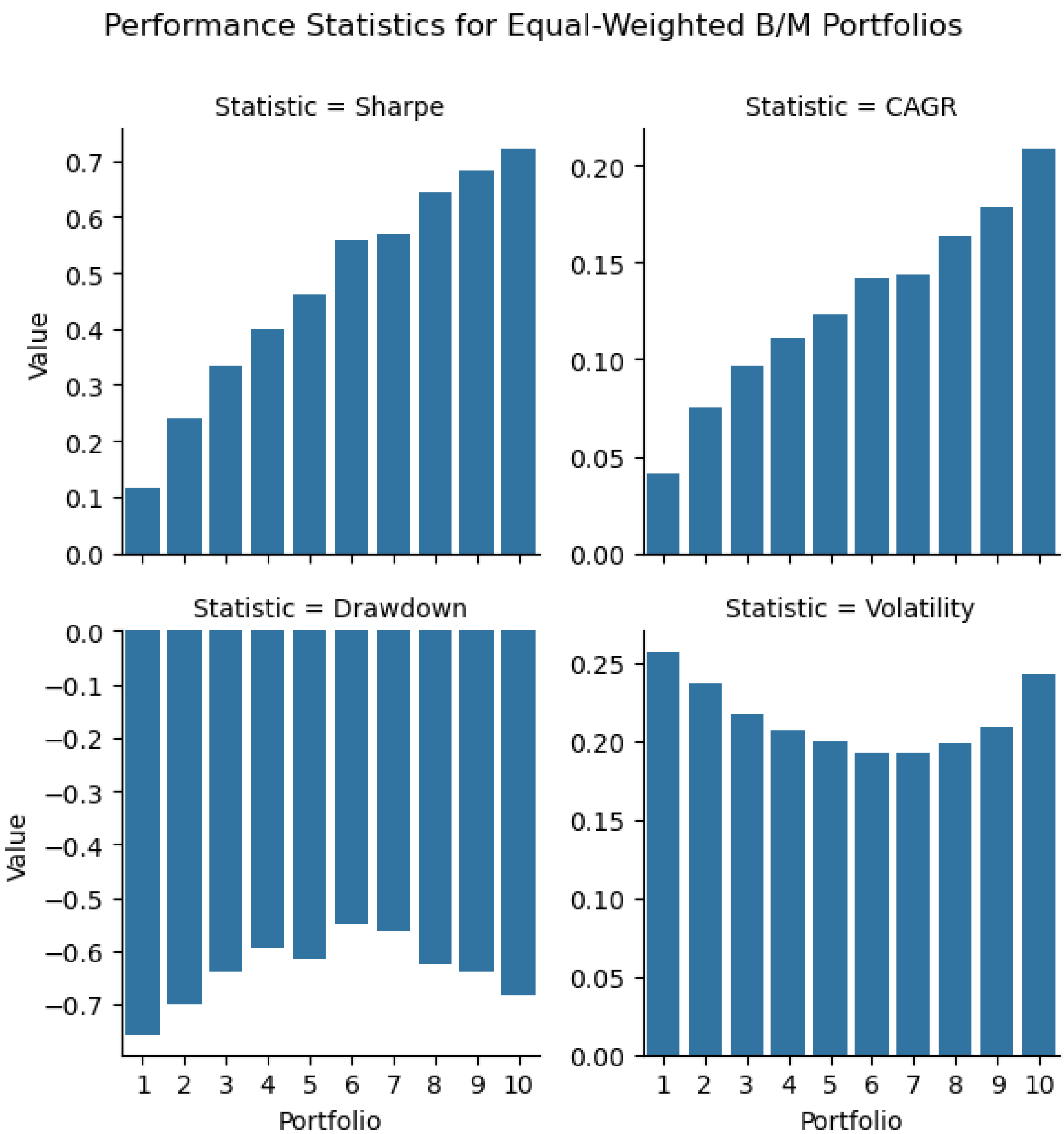
```
def Volatility(x, ann_fac=np.sqrt(12)):
    return ann_fac * x.std()
```

```
def Drawdown(x):
    price = x.add(1).cumprod()
    return (price / price.cummax()).min() - 1
```

The high B/M have higher Sharper ratios and CAGRs than low B/M portfolios. However, the risk relation is not monotonic.

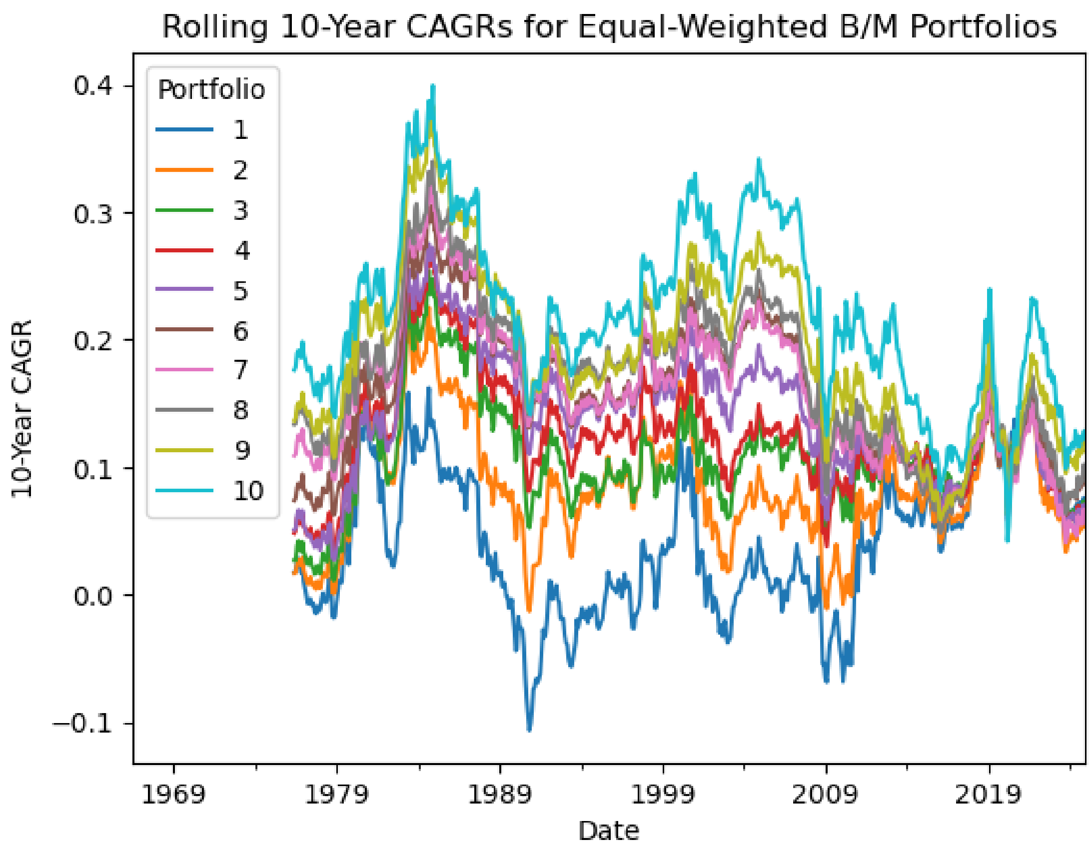
```
import seaborn as sns
```

```
stats = (
    portfolios_rf
    .agg(Sharpe)
    .to_frame('Sharpe')
    .join(
        portfolios
        .agg([CAGR, Drawdown, Volatility])
        .transpose()
    )
    .rename_axis(columns='Statistic')
)

df = stats.stack().to_frame('Value').reset_index()

sns.catplot(
    data=df,
    x='Portfolio',
    y='Value',
    col='Statistic',
    col_wrap=2,
    height=3,
    kind='bar',
    sharey=False
)

plt.suptitle('Performance Statistics for Equal-Weighted B/M Portfolios', y=1.05)
plt.show()
```



Also, this outperformance narrows over the past few decades.

```
portfolios.rolling(window=120).apply(CAGR).plot()
plt.ylabel('10-Year CAGR')
plt.title('Rolling 10-Year CAGRs for Equal-Weighted B/M Portfolios')
plt.show()
```



Re-implement the value strategy from the lecture notebook *with value-weighted portfolios*

We need to do two things:

1. Add the beginning of month market values of equity to use as portfolio weights
2. Replace `.mean()` with `np.average()` to calculate weighted means

This code prepares stock data from the `crsp` data frame by adding beginning-of-month market equity (`ME`) and merging it with portfolio assignments from the `bm` data frame. It first selects `PERMNO`, `date`, and `RET` from `crsp`, then merges it with `ME` data shifted forward by 1 month using an inner join to align current returns with next month's market equity. Next, it sorts the CRSP data by `date` and `PERMNO`, and merges it with a subset of `bm` containing only `Date`, `PERMNO`, and `Portfolio`, sorted by `Date` and `PERMNO`, matching each stock's date to the closest prior `Date` in `bm` within a 366-day tolerance. Finally, it keeps `PERMNO`, `date`, `Portfolio`, `RET`, and `ME`, drops missing values, converts `Portfolio` to integers, and renames `date` to `Date` for consistency.

```
# Add beginning-of-month market equity (ME) to CRSP data
crsp_w_bom_me = (
    crsp
    [['PERMNO', 'date', 'RET']]                  # Select stock ID, date, and returns from CR
    .merge(
        right=crsp[['PERMNO', 'date', 'ME']].assign(  # Select stock ID, date, and ME, shift
            date=lambda x: x['date'] + pd.offsets.MonthEnd(1)),
        how='inner'                              # Inner join to align current month with nex
    )
)

# Merge CRSP data with book-to-market portfolio assignments
stocks_vw = (
    pd.merge_asof(
        left=crsp_w_bom_me.sort_values(['date', 'PERMNO']),  # Sort CRSP with ME by date and
        right=bm.sort_values(['Date', 'PERMNO'])[['Date', 'PERMNO', 'Portfolio']],  # Select
        left_on='date',                                      # Match CRSP dates to the closes
        right_on='Date',                                     # Use BM Date as the reference f
        by='PERMNO',                                         # Join on stock ID (PERMNO) to e
        tolerance=pd.Timedelta('366d')                       # Limit matches to BM data withi
    )
    [['PERMNO', 'date', 'Portfolio', 'RET', 'ME']]           # Select essential columns: stoc
    .dropna()                                                # Remove rows with missing value
    .assign(Portfolio=lambda x: x['Portfolio'].astype(int))  # Convert portfolio numbers to i
    .rename(columns={'date': 'Date'})                        # Rename 'date' to 'Date' to ali
)
```

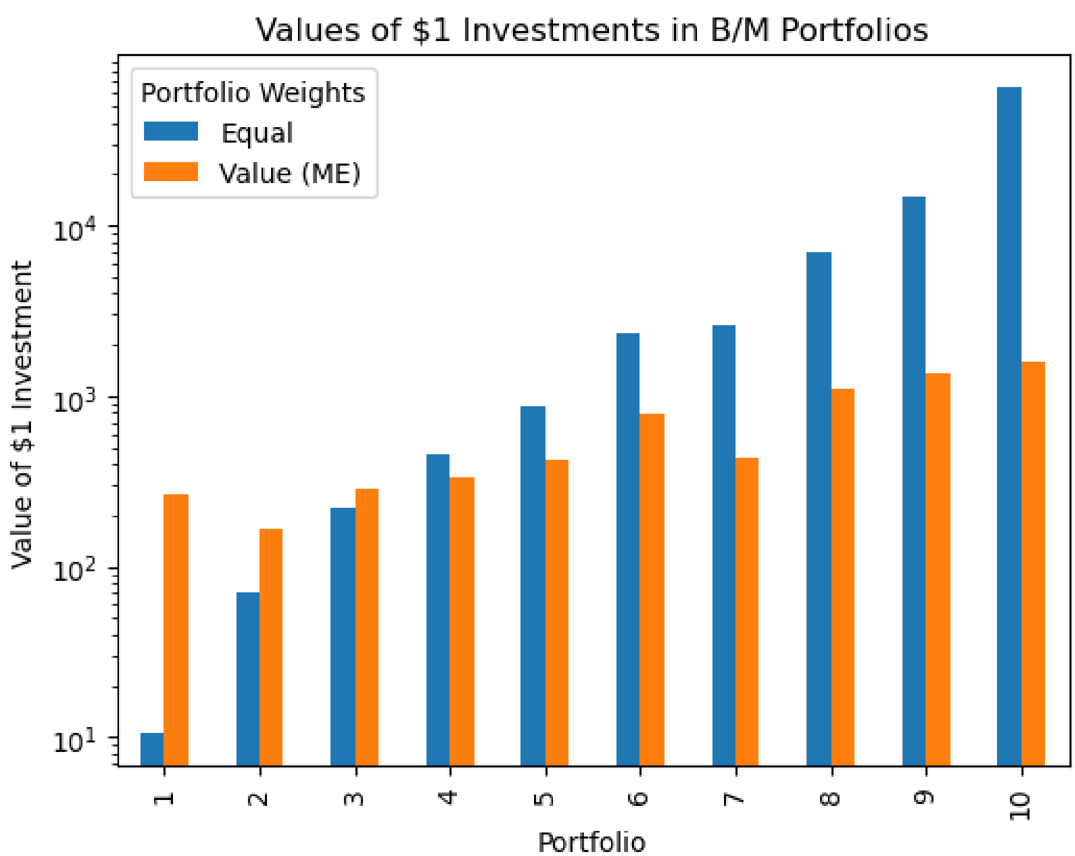
This code creates value-weighted portfolios using the `stocks` data frame and reshapes the results into a wide format. It groups the data by `Portfolio` (1 to 10) and `Date`, then calculates the value-weighted average return (`RET`) for each group, using market equity (`ME`) as weights. Next, it pivots the data so each portfolio's returns become separate columns, with `Date` as the index.

```
# Calculate value-weighted portfolio returns and reshape into wide format
portfolios_vw = (
    stocks_vw
    .groupby(['Portfolio', 'Date'])              # Group stocks by portfolio number (1-10) an
    .apply(lambda x: np.average(a=x['RET'],      # Compute value-weighted average returns usi
                                weights=x['ME']),
           include_groups=False)                 # Exclude grouping columns (Portfolio, Date)
    .unstack('Portfolio')                        # Pivot data so each portfolio's returns bec
)
```

We can briefly compare the equal- and value-weighted portfolios. *We see that the value effect is concentrated in small stocks because the equal-weighted portfolios returns are much larger than the value-weighted portfolio returns.*

```
df = pd.concat(
    objs=[portfolios.add(1).prod(), portfolios_vw.add(1).prod()],
    keys=['Equal', 'Value (ME)'],
    names=['Portfolio Weights'],
    axis=1
)
```

```
df.plot(kind='bar')
plt.semilogy()
plt.ylabel('Value of $1 Investment')
plt.title('Values of $1 Investments in B/M Portfolios')
plt.show()
```



Estimate the αs of the equal-weighted and value-weighted portfolios

> **Note**
>
> Here `portfolios` is wide, so I have to `.stack()` it before the join. This is a difference from class

```
portfolios_ff3 = (
    portfolios
    .stack(future_stack=True)
    .to_frame('RET')
    .join(ff3, how='inner')
)
```

```
import statsmodels.formula.api as smf
```

The value portfolio 10 has an α of 82 basis points! Annualized, this is $12 \times 0.0082 = 0.0984$ or almost 10% of returns not explained by market risk.

```
smf.ols(formula='I(RET-RF) ~ Q("Mkt-RF")', data=portfolios_ff3.query('(Portfolio == 10)')).f
```

Dep. Variable:	I(RET - RF)	**R-squared:**	0.506
Model:	OLS	**Adj. R-squared:**	0.505
Method:	Least Squares	**F-statistic:**	717.6
Date:	Sat, 22 Mar 2025	**Prob (F-statistic):**	2.34e-109
Time:	06:49:48	**Log-Likelihood:**	1116.4
No. Observations:	702	**AIC:**	-2229.
Df Residuals:	700	**BIC:**	-2220.
Df Model:	1		
Covariance Type:	nonrobust		

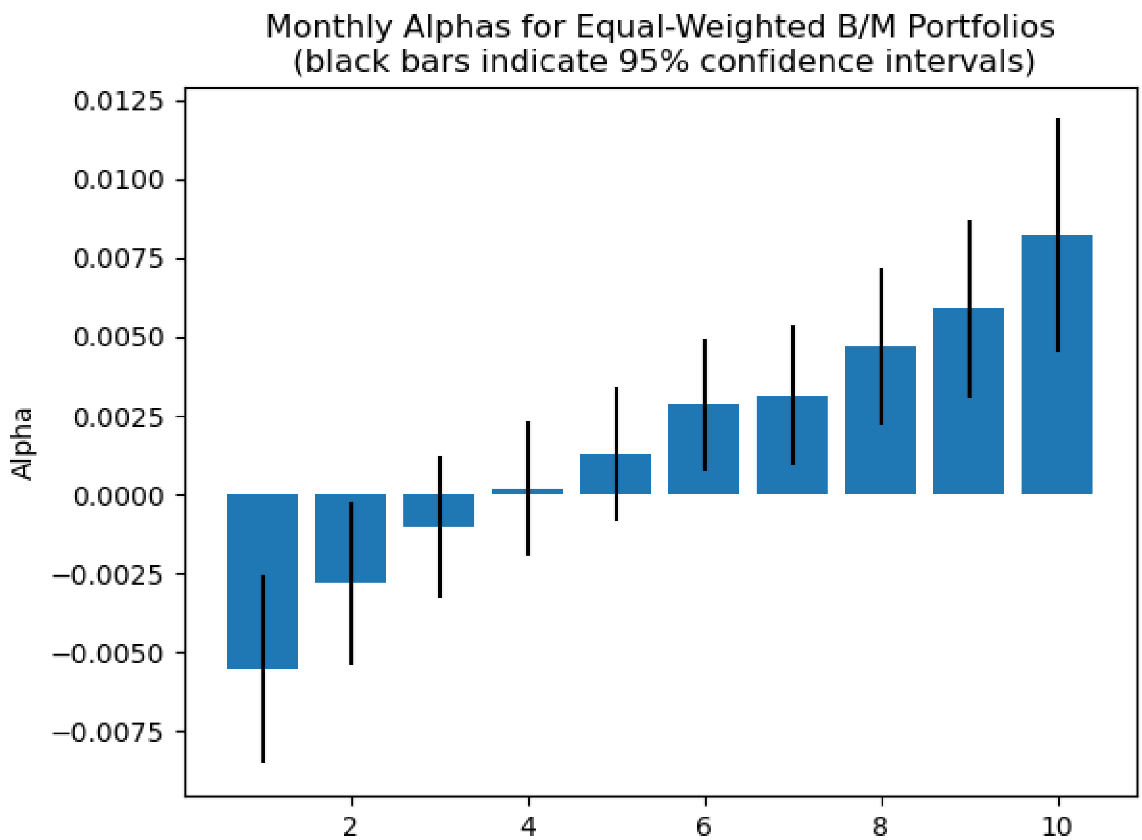
	coef	**std err**	**t**	**P> |t|**	**[0.025**	**0.975]**
Intercept	0.0082	0.002	4.370	0.000	0.005	0.012
Q("Mkt-RF")	1.0948	0.041	26.788	0.000	1.015	1.175

Omnibus:	268.363	**Durbin-Watson:**	1.740
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	1787.450
Skew:	1.560	**Prob(JB):**	0.00
Kurtosis:	10.167	**Cond. No.**	21.9

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

We can plot these αs (and confidence intervals) with list comprehensions!

```
fits = [
    smf.ols(formula='I(RET-RF) ~ Q("Mkt-RF")', data=portfolios_ff3.query(f'(Portfolio == {p}]
    for p in range(1, 11)
]

df = (
    pd.DataFrame(
        data={
            'Alpha': [f.params['Intercept'] for f in fits],
            'Sterr': [f.bse['Intercept'] for f in fits],
            'Portfolio': range(1, 11)
        }
    )
    .set_index('Portfolio')
)

plt.bar(
    x=df.index,
    height=df['Alpha'],
    yerr=df['Sterr'].mul(1.96)
)
plt.ylabel('Alpha')
plt.title('Monthly Alphas for Equal-Weighted B/M Portfolios\n (black bars indicate 95% confic
plt.show()
```



Implement a momentum strategy

Form deciles on the 11-month returns from months $t-12$ to month $t-2$.

This code calculates 11-month momentum returns (`RET_11`) from 2 to 12 months prior using the `crsp` data frame and assigns portfolios based on those returns. It sets a multi-index with `PERMNO` and `date`, selects the `RET` column, and pivots to a wide format with stocks as columns. After sorting by date, it computes compounded returns over an 11-month rolling window using log transformations (`log1p` and `expm1`), then reshapes back to long format. It drops rows with missing `RET_11` values, creates a DataFrame, and resets the index. Finally, it assigns stocks to decile portfolios (1 to 10) based on `RET_11` for each date and shifts the `date` forward by 2 months to reflect the momentum strategy lag. That is, we receive returns in month t based on returns from month $t-12$ through $t-2$.

```
# Calculate 2-12 month momentum returns and assign portfolios
mom_02_12 = (
    crsp
```

```
    .set_index(['PERMNO', 'date'])                # Set multi-index with stock ID (PERMNO) and
    ['RET']                                       # Select the monthly returns column
    .unstack('PERMNO')                            # Pivot to wide format, with PERMNO as column
    .sort_index()                                 # Sort dates chronologically for consistent r
    .pipe(np.log1p)                               # Convert returns to log scale (log(1 + r)) f
    .rolling(11)                                  # Apply an 11-month rolling window to prior r
    .sum()                                        # Sum log returns over the 11-month window
    .pipe(np.expm1)                               # Convert summed log returns back to compound
    .stack(future_stack=True)                     # Reshape to long format, aligning with panda
    .dropna()                                     # Remove rows with NaN from incomplete window
    .to_frame('RET_11')                           # Create a DataFrame, naming the momentum ret
    .reset_index()                                # Move PERMNO and date back to columns from t
    .assign(
        Portfolio=lambda x: x.groupby('date')['RET_11'].transform(  # Assign stocks to decile
            lambda x: 1 + pd.qcut(x=x, q=10, labels=False)),       # Split RET_11 into 10 qua
        date=lambda x: x['date'] + pd.offsets.MonthEnd(2)          # Shift dates forward 2 mo
    )
)
```

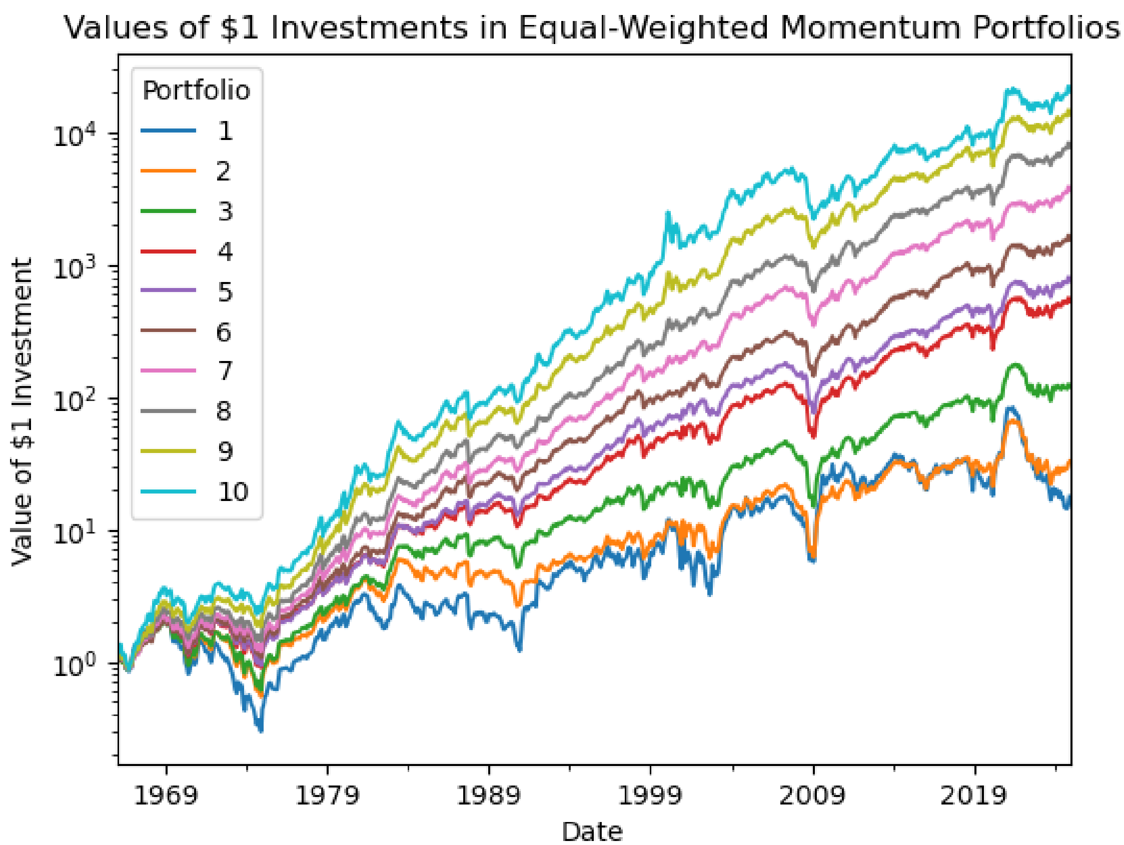
```
mom_02_12.head()
```

	date	PERMNO	RET_11	Portfolio
0	1966-01-31	10006	0.1508	5
1	1966-01-31	10014	0.2000	6
2	1966-01-31	10030	0.1489	5
3	1966-01-31	10057	0.4666	8
4	1966-01-31	10102	0.3503	7

```
stocks = (
    pd.merge(
        left=crsp,
        right=mom_02_12,
        on=['PERMNO', 'date'],
        how='inner'
    )
    [['PERMNO', 'date', 'Portfolio', 'RET']]
    .dropna()
    .rename(columns={'date': 'Date'})
)
```

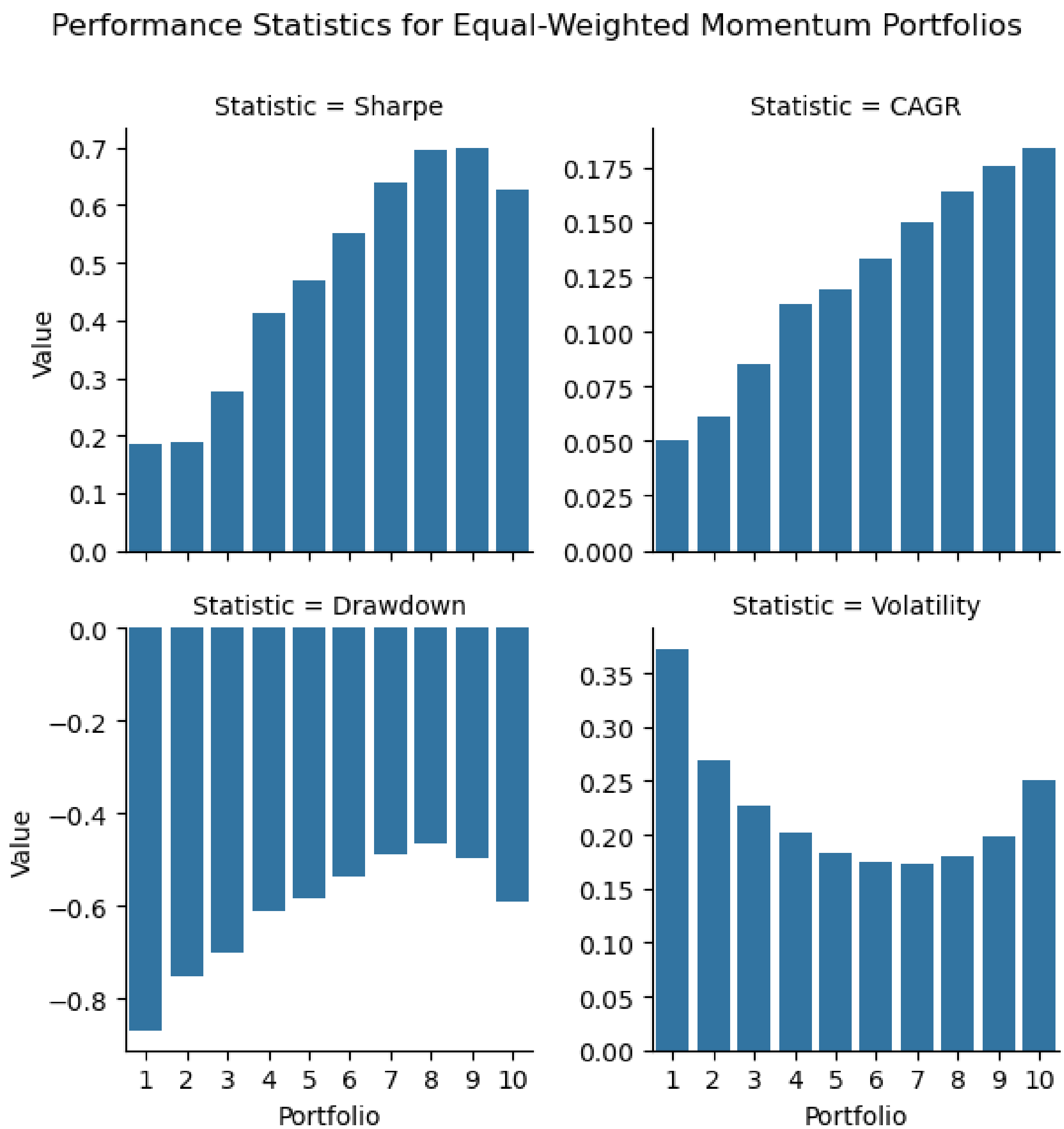
```
portfolios = (
    stocks
    .groupby(['Portfolio', 'Date'])  # Group stocks by portfolio number (1-10) and trading da
    [['RET']]                        # Isolate the returns column for portfolio-level aggrega
    .mean()                          # Average returns across stocks within each portfolio-da
    .unstack('Portfolio')            # Pivot the data to create columns for each portfolio's
    ['RET']                          # Extract the returns data, retaining portfolio numbers
)
```

```
portfolios.add(1).cumprod().plot()
plt.semilogy()
plt.ylabel('Value of $1 Investment')
plt.title('Values of $1 Investments in Equal-Weighted Momentum Portfolios')
plt.show()
```



```
portfolios_rf = portfolios.sub(ff3['RF'], axis=0)
```

```
stats = (
    portfolios_rf
    .agg(Sharpe)
    .to_frame('Sharpe')
    .join(
        portfolios
        .agg([CAGR, Drawdown, Volatility])
        .transpose()
    )
    .rename_axis(columns='Statistic')
)

df = stats.stack().to_frame('Value').reset_index()

sns.catplot(
    data=df,
    x='Portfolio',
    y='Value',
    col='Statistic',
    col_wrap=2,
    height=3,
    kind='bar',
    sharey=False
)

plt.suptitle('Performance Statistics for Equal-Weighted Momentum Portfolios', y=1.05)
plt.show()
```


Performance Statistics for Equal-Weighted Momentum Portfolios
Statistic = Sharpe
Statistic = CAGR
Statistic = Drawdown
Statistic = Volatility
Value
Value
Portfolio
Portfolio
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.000
0.025
0.050
0.075
0.100
0.125
0.150
0.175
0.0
−0.2
−0.4
−0.6
−0.8
0.00
0.05
0.10
0.15
0.20
0.25
0.30
0.35
1
2
3
4
5
6
7
8
9
10

Herron Topic 3 - Practice - Sec 03

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import pandas_datareader as pdr
import statsmodels.api as sm
import yfinance as yf
```

```
%precision 4
pd.options.display.float_format = '{:.4f}'.format
# %config InlineBackend.figure_format = 'retina'
```

Announcements

1. Projects:
    1. Project 1:
        1. I am still grading
        2. I plan to finish grading by Friday
    2. Project 2:
        1. Due on Friday, 3/28, at 11:59 PM
        2. We will use class time next week for group work
        3. Ask your questions here: https://northeastern.instructure.com/courses/207607/discussion_topics/2738173
2. Assessments:
    1. ***Both assessments are in class on Tuesday, 4/15***
    2. Programming assessment: Should take 15 minutes and will be based on the questions here:
    3. MSFQ assessment (only for MSFQ students): Should take 45 minutes and based on the 5 required MSFQ courses (standby for a Canvas announcement with more details after I discuss it with the MSFQ program director)

Five-Minute Review

Quantitative value investing is a systematic investment strategy that combines the principles of value investing with data-driven, quantitative techniques. It involves identifying undervalued securities—those trading below their intrinsic value—using predefined, measurable criteria rather than subjective judgment. Common criteria are price/earnings (P/E) and price/book (P/B) ratios. Here, we will use the book-to-market equity ratio (B/M), which is common in the academic literature and quantitative investing.

The key concepts in this topic are creating metrics, forming portfolios, and backtesting the strategy.

Practice

Re-implement the value strategy from the lecture notebook

Read the data

This code reads stock market data from a file called `crsp.csv` and prepares it for analysis. It loads the data into a data frame, makes sure the dates are in the right format, and marks certain letters (`A`, `B`, and `C`) as missing values. Then, it filters the data to include only records from 1965 or later, sorts it by company and date, and adds two new columns: one for market equity (calculated as the number of shares times the stock price, converted to millions), and another to adjust all dates to the end of each month.

```
crsp = (
    pd.read_csv(
        filepath_or_buffer='crsp.csv',  # Read data from crsp.csv file
        parse_dates=['date'],           # Convert 'date' column to datetime format
        na_values=['A', 'B', 'C']       # Treat 'A', 'B', 'C' as missing values (NaN)
    )
    .query('date >= 1965')              # Filter to keep only data from 1965 onward
    .sort_values(['PERMNO', 'date'])    # Sort by company identifier (PERMNO) and date
    .assign(
        ME=lambda x: x['SHROUT'] * x['PRC'].abs() / 1_000,  # Calculate market equity (ME) a
        date=lambda x: x['date'] + pd.offsets.MonthEnd(0)   # Adjust dates to the last day o
    )
)
```

```
crsp.head()
```

	PERMNO	date	SHRCD	PRC	RET	SHROUT	ME
0	10000	1986-01-31	10	-4.3750	NaN	3680.0000	16.1000
1	10000	1986-02-28	10	-3.2500	-0.2571	3680.0000	11.9600
2	10000	1986-03-31	10	-4.4375	0.3654	3680.0000	16.3300
3	10000	1986-04-30	10	-4.0000	-0.0986	3793.0000	15.1720
4	10000	1986-05-31	10	-3.1094	-0.2227	3793.0000	11.7939

This code brings in financial data from a file named `compustat.csv` and prepares it for analysis. It loads the data into a data frame, formats dates, and filters out any records before 1965. Then, it sorts the data by company, fiscal year, and date. Finally, it removes any duplicate entries for the same company and fiscal year, keeping only the most recent record.

```
compustat = (
    pd.read_csv(
        filepath_or_buffer='compustat.csv',  # Read data from compustat.csv file
        parse_dates=['datadate']              # Convert 'datadate' column to datetime format
    )
    .query('datadate >= 1965')                # Filter to keep only data from 1965 onward
    .sort_values(['LPERMNO', 'fyear', 'datadate'])  # Sort by company identifier (LPERMNO), f
    .drop_duplicates(subset=['LPERMNO', 'fyear'], keep='last')  # Keep only the latest recor
)
```

```
compustat.head()
```

	GVKEY	LPERMNO	datadate	fyear	indfmt	consol	popsrc	datafmt	curcd	ceq
165681	13007	10000	1986-10-31	1986.0000	INDL	C	D	STD	USD	0.418
165586	12994	10001	1986-06-30	1986.0000	INDL	C	D	STD	USD	5.432
165587	12994	10001	1987-06-30	1987.0000	INDL	C	D	STD	USD	5.369
165588	12994	10001	1988-06-30	1988.0000	INDL	C	D	STD	USD	5.512
165589	12994	10001	1989-06-30	1989.0000	INDL	C	D	STD	USD	6.321

About 64 percent of firm-years have December fiscal-year ends.

```
compustat['datadate'].dt.month.value_counts() / compustat['datadate'].shape[0]
```

```
datadate
12    0.6401
6     0.0723
```

```
9     0.0609
3     0.0518
10    0.0341
1     0.0334
8     0.0203
7     0.0187
5     0.0187
4     0.0179
11    0.0173
2     0.0146
Name: count, dtype: float64
```

Create the interim data frames

This code finds the market value of equity (`ME`) as of December each year from the `crsp` data frame (to match with book value of equity `ceq` from the `compustat` data frame below). It filters out any rows where market equity is zero or negative, then sorts the data by company and date. Next, it groups the data by company and year, focusing on December values. Finally, it selects the last market equity value for each company in each year, giving us the December `ME`.

```
mve = (
    crsp
    .query('ME > 0')                            # Filter for positive market equity values
    .sort_values(['PERMNO', 'date'])            # Sort by company identifier (PERMNO) and da
    .groupby(by=['PERMNO', pd.Grouper(key='date', freq='YE-DEC')])  # Group by company and ye
    [['ME']]                                    # Select the market equity (ME) column
    .last()                                     # Take the last ME value for each group (Dec
)
```

```
mve.head()
```

		M
PERMNO	date	
10000	1986-12-31	1.
	1987-12-31	0.
10001	1986-12-31	6.
	1987-12-31	5.
	1988-12-31	6.

This code finds the book value of equity (`ceq`) as of December each year from the `compustat` data frame. It filters out any rows where common equity is zero or negative, then sorts the data by company and date. Next, it groups the data by company and year, focusing on December values. Finally, it selects the last common equity value for each company in each year, giving us the December `ceq`.

```
bve = (
    compustat
    .query('ceq > 0')                              # Filter for positive common equity values
    .sort_values(['LPERMNO', 'datadate'])          # Sort by company identifier (LPERMNO) and d
    .groupby(by=['LPERMNO', pd.Grouper(key='datadate', freq='YE-DEC')])  # Group by company a
    [['ceq']]                                      # Select the common equity (ceq) column
    .last()                                        # Take the last CEQ value for each group (De
)
```

```
bve.head()
```

		ce
LPERMNO	datadate	
10000	1986-12-31	0.
10001	1986-12-31	5.
	1987-12-31	5.
	1988-12-31	5.
	1989-12-31	6.

Combine the data frames and form portfolios

This code shows a simple example of how to form portfolios using the `pd.qcut()` function. It takes a sequence of numbers from 0 to 9 and splits them into 2 equal groups (quantiles), assigning them to either a low or high category. The `labels=False` argument gives us numeric group identifiers (0 or 1), and adding 1 shifts these to 1 or 2, which can represent portfolio numbers.

```
1 + pd.qcut(x=np.arange(10), q=2, labels=False)
```

```
array([1, 1, 1, 1, 1, 2, 2, 2, 2, 2])
```

> **Note**
>
> In class, in the following cell, I used multiple cursors to simultaneously type two `.rename_axis(index=['PERMNO', 'Date'])` methods. If you hold down the `CTRL` key, every mouse click will generate a new cursor. Each cursor will do the same action (e.g., typing, highlighting, and deleting). More here.

This code creates portfolios based on book-to-market (`BM`) ratios using the `bve` and `mve` data frames. It starts by aligning the company and date indexes of both datasets and merges them, keeping only rows with matching data. Next, it calculates the `BM` ratio by dividing book value of equity (`ceq`) by market value of equity (`ME`). After resetting the index, it shifts the dates forward by 7 months to time the portfolio formation correctly, giving 6 months between `BM` calculation and buying stocks at the end of June to receive July returns. Finally, for each date, it sorts the `BM` ratios into 10 equal groups (deciles) and assigns portfolio numbers from 1 to 10.

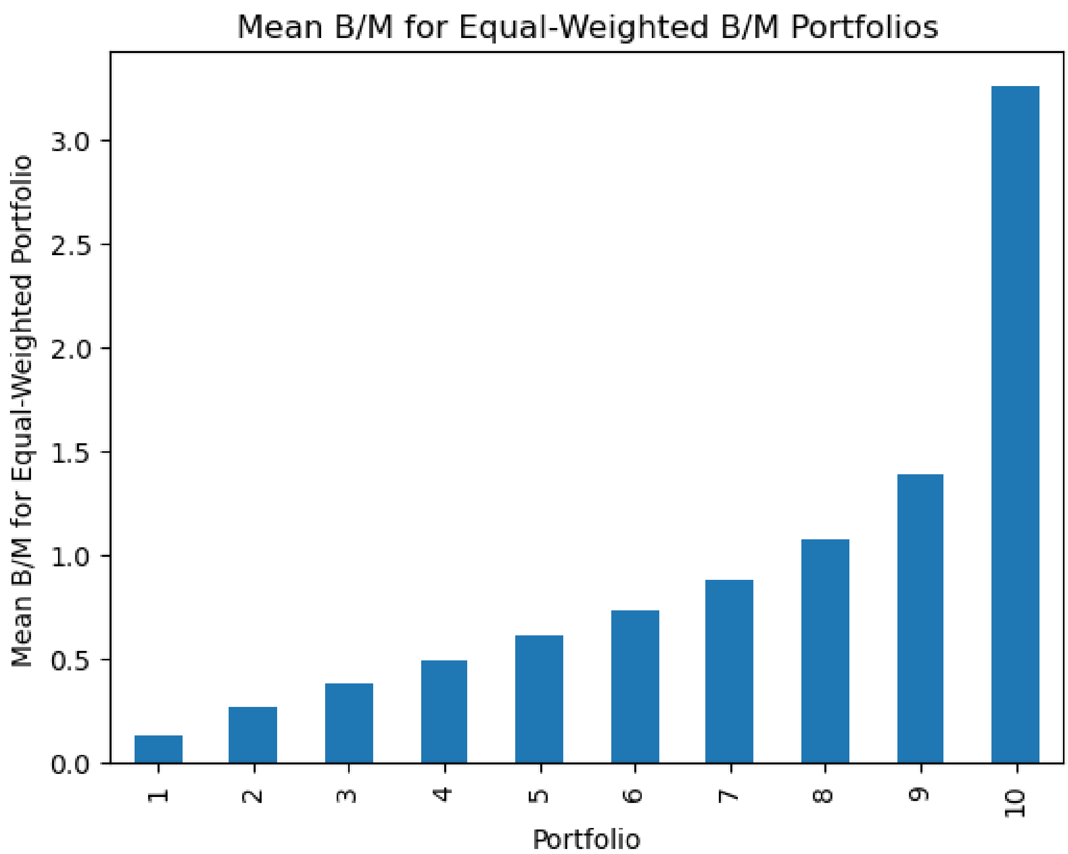
```
bm = (
    bve
    .rename_axis(index=['PERMNO', 'Date'])  # Standardize index names to PERMNO (company ID)
    .join(
        other=mve.rename_axis(index=['PERMNO', 'Date']),  # Combine with market equity data,
        how='inner'                                      # Use inner join to retain only records wi
    )
    .reset_index()                                       # Flatten the index into columns for easie
    .assign(
        Date=lambda x: x['Date'] + pd.offsets.MonthEnd(7),  # Shift dates forward 7 months (
        BM=lambda x: x['ceq'] / x['ME'],                   # Compute book-to-market ratio as 
        Portfolio=lambda x: x.groupby('Date')['BM'].transform(  # Assign stocks to portfolios
            lambda x: 1 + pd.qcut(x=x, q=10, labels=False)  # Divide BM into 10 quantiles (0-
        )
    )
)
```

```
bm.head()
```

	PERMNO	Date	ceq	ME	BM	Portfolio
0	10000	1987-07-31	0.4180	1.9816	0.2109	2
1	10001	1987-07-31	5.4320	6.9370	0.7830	7
2	10001	1988-07-31	5.3690	5.8280	0.9212	7
3	10001	1989-07-31	5.5120	6.3623	0.8664	7
4	10001	1990-07-31	6.3210	10.3477	0.6109	5

This code creates a bar chart showing the average book-to-market (`BM`) ratio for each portfolio using the `portfolios` data frame. It groups the data by portfolio number (1 to 10), calculates the mean `BM` ratio for each group, and then plots these averages as bars. As expected, we see that `BM` rises from portfolio 1 to 10.

```
(
    bm
    .groupby(by=['Portfolio', 'Date'])  # Group data by portfolio number and date
    ['BM']                              # Select the book-to-market (BM) column
    .mean()                             # Calculate the mean BM for each portfolio-date combi
    .groupby(by='Portfolio')            # Group again by portfolio to average across all date
    .mean()                             # Calculate the time-averaged mean BM for each portfo
    .plot(kind='bar')                   # Plot the results as a bar chart
)
plt.ylabel('Mean B/M for Equal-Weighted Portfolio')
plt.title('Mean B/M for Equal-Weighted B/M Portfolios')
plt.show()
```



This code combines monthly stock returns from the `crsp` data frame with the *most recent* portfolio assignments from the `bm` data frame. The `tolerance` argument tells the merge to only use portfolio assignments within 366 days to avoid stale data.

```
stocks = (
    pd.merge_asof(
        left=crsp.sort_values(['date', 'PERMNO']),    # Sort CRSP data by date and company II
        right=bm.sort_values(['Date', 'PERMNO']),     # Sort BM data by date and company ID
        left_on='date',                               # Match CRSP dates to the closest prio
        right_on='Date',                              # Use BM dates as the reference for the
        by='PERMNO',                                  # Merge on company ID (PERMNO) to link
        tolerance=pd.Timedelta('366d')                # Limit matches to portfolio assignmen
    )
    [['PERMNO', 'date', 'Portfolio', 'RET']]          # Select key columns: company ID, date
    .dropna()                                         # Remove rows with missing values (e.g
    .assign(Portfolio=lambda x: x['Portfolio'].astype(int))  # Convert portfolio numbers to i
    .rename(columns={'date': 'Date'})                 # Rename 'date' column to 'Date' for co
)
```

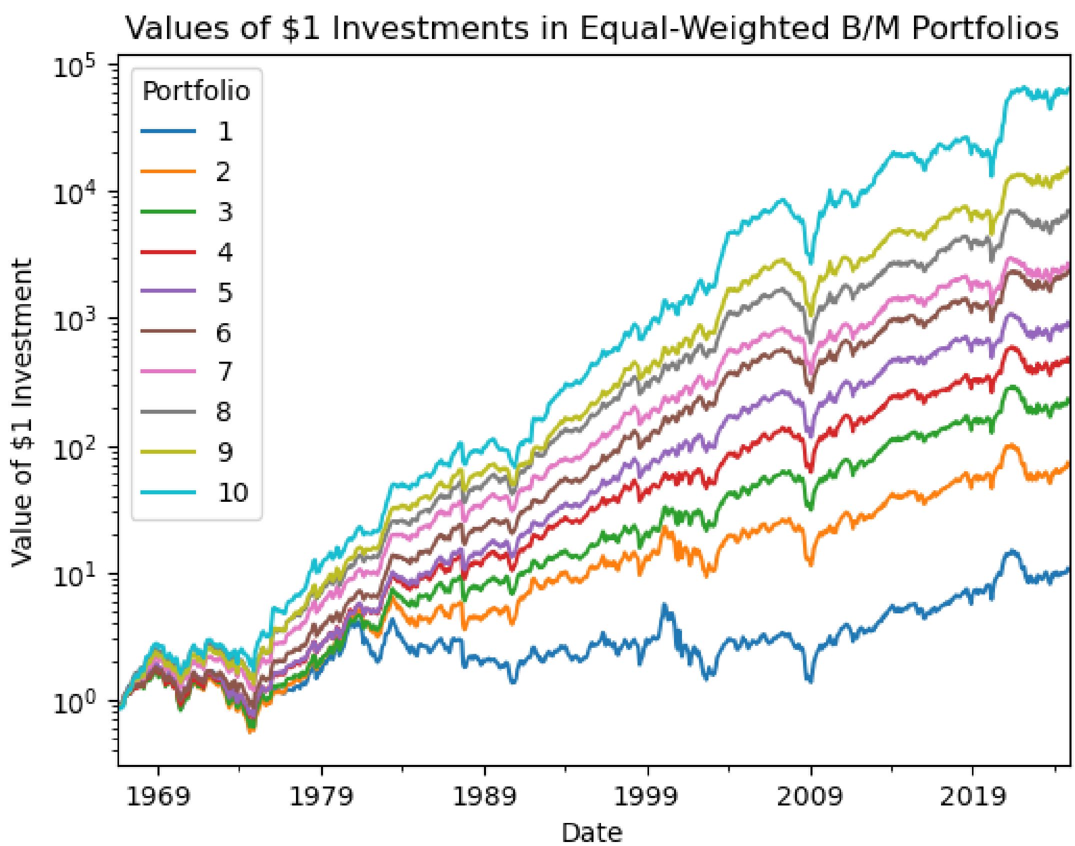
This code creates equal-weighted portfolios using the `stocks` data frame. It groups the data by portfolio number (`Portfolio`) and date (`Date`), then calculates the mean return (`RET`) for each group. This averaging process gives each stock within a portfolio the same weight, producing equal-weighted portfolio returns.

> **Note**
>
> Here I made `portfolios` a wide data frame (i.e., portfolio numbers in columns) to simplify plots and calculations. This is a difference from class

```
portfolios = (
    stocks
    .groupby(['Portfolio', 'Date'])  # Group stocks by portfolio number (1-10) and trading da
    [['RET']]                        # Isolate the returns column for portfolio-level aggrega
    .mean()                          # Average returns across stocks within each portfolio-da
    .unstack('Portfolio')            # Pivot the data to create columns for each portfolio's
    ['RET']                          # Extract the returns data, retaining portfolio numbers
)
```

Finally, we can see that the value portfolio 10 outperforms the growth portfolio 1!

```
portfolios.add(1).cumprod().plot()
plt.semilogy()
plt.ylabel('Value of $1 Investment')
plt.title('Values of $1 Investments in Equal-Weighted B/M Portfolios')
plt.show()
```



Backtest the strategy

We will need the Fama and French (1993) factors for the risk-free rate.

> **Note**
>
> Here I set the `Date` index to simplify merges/joins below. This is a difference from class

```
ff3 = (
    pdr.DataReader(
        name='F-F_Research_Data_Factors',
        data_source='famafrench',
        start='1900'
    )
    [0]
    .div(100)
    .reset_index()
    .assign(Date=lambda x: x['Date'].dt.to_timestamp(how='end').dt.normalize()) # convert mon
    .set_index('Date')
)
```

```
C:\Users\r.herron\AppData\Local\Temp\ipykernel_20756\1889697072.py:2: FutureWarning: The argu
  pdr.DataReader(
C:\Users\r.herron\AppData\Local\Temp\ipykernel_20756\1889697072.py:2: FutureWarning: The argu
  pdr.DataReader(
```

```
portfolios_rf = portfolios.sub(ff3['RF'], axis=0)
```

```
def CAGR(x, ann_fac=12):
    return (1 + x).prod() ** (ann_fac / x.count()) - 1
```

```
def Sharpe(x, ann_fac=np.sqrt(12)):
    return ann_fac * x.mean() / x.std()
```

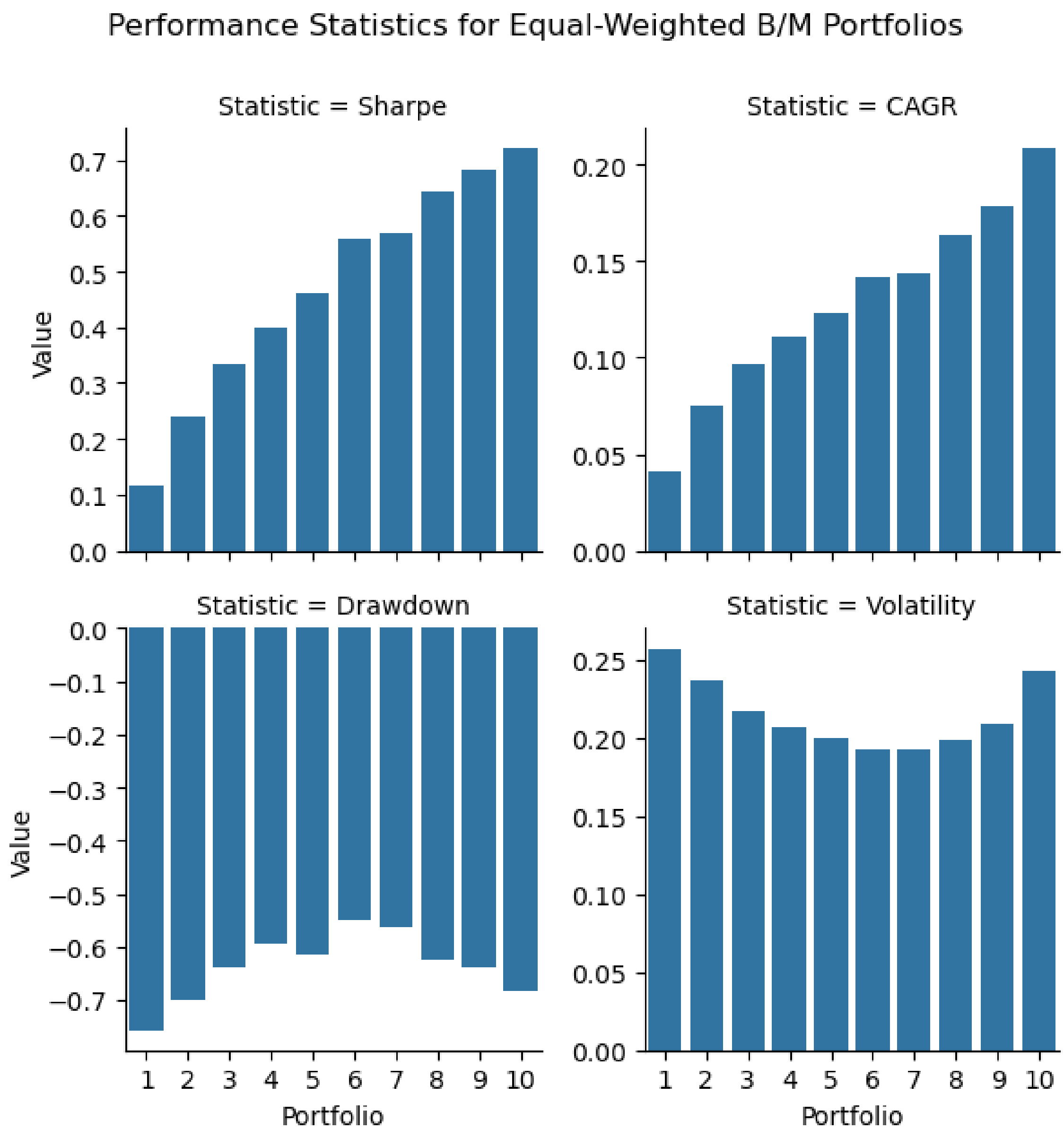
```
def Volatility(x, ann_fac=np.sqrt(12)):
    return ann_fac * x.std()
```

```
def Drawdown(x):
    price = x.add(1).cumprod()
    return (price / price.cummax()).min() - 1
```

The high B/M have higher Sharper ratios and CAGRs than low B/M portfolios. However, the risk relation is not monotonic.

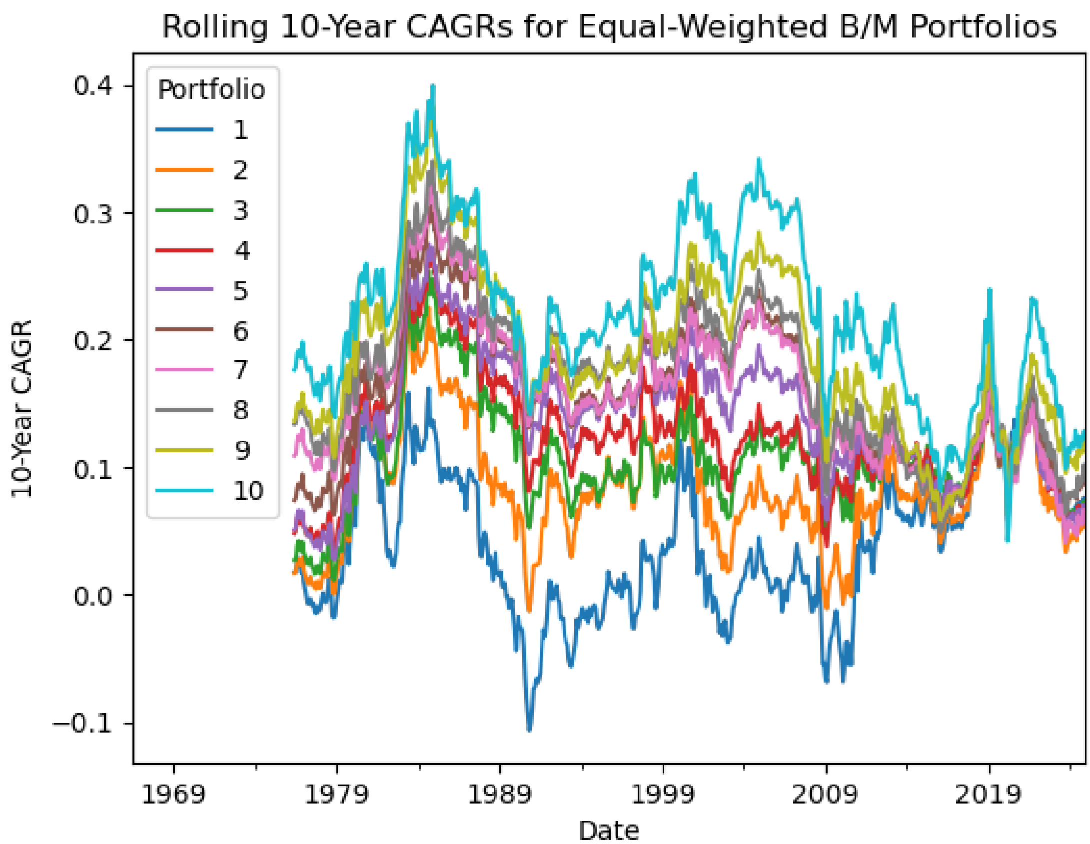
```
import seaborn as sns
```

```
stats = (
    portfolios_rf
    .agg(Sharpe)
    .to_frame('Sharpe')
    .join(
        portfolios
        .agg([CAGR, Drawdown, Volatility])
        .transpose()
    )
    .rename_axis(columns='Statistic')
)

df = stats.stack().to_frame('Value').reset_index()

sns.catplot(
    data=df,
    x='Portfolio',
    y='Value',
    col='Statistic',
    col_wrap=2,
    height=3,
    kind='bar',
    sharey=False
)

plt.suptitle('Performance Statistics for Equal-Weighted B/M Portfolios', y=1.05)
plt.show()
```



Also, this outperformance narrows over the past few decades.

```
portfolios.rolling(window=120).apply(CAGR).plot()
plt.ylabel('10-Year CAGR')
plt.title('Rolling 10-Year CAGRs for Equal-Weighted B/M Portfolios')
plt.show()
```



Re-implement the value strategy from the lecture notebook *with value-weighted portfolios*

We need to do two things:

1. Add the beginning of month market values of equity to use as portfolio weights
2. Replace `.mean()` with `np.average()` to calculate weighted means

This code prepares stock data from the `crsp` data frame by adding beginning-of-month market equity (`ME`) and merging it with portfolio assignments from the `bm` data frame. It first selects `PERMNO`, `date`, and `RET` from `crsp`, then merges it with `ME` data shifted forward by 1 month using an inner join to align current returns with next month's market equity. Next, it sorts the CRSP data by `date` and `PERMNO`, and merges it with a subset of `bm` containing only `Date`, `PERMNO`, and `Portfolio`, sorted by `Date` and `PERMNO`, matching each stock's date to the closest prior `Date` in `bm` within a 366-day tolerance. Finally, it keeps `PERMNO`, `date`, `Portfolio`, `RET`, and `ME`, drops missing values, converts `Portfolio` to integers, and renames `date` to `Date` for consistency.

```
# Add beginning-of-month market equity (ME) to CRSP data
crsp_w_bom_me = (
    crsp
    [['PERMNO', 'date', 'RET']]                  # Select stock ID, date, and returns from CR
    .merge(
        right=crsp[['PERMNO', 'date', 'ME']].assign(  # Select stock ID, date, and ME, shift
            date=lambda x: x['date'] + pd.offsets.MonthEnd(1)),
        how='inner'                              # Inner join to align current month with nex
    )
)

# Merge CRSP data with book-to-market portfolio assignments
stocks_vw = (
    pd.merge_asof(
        left=crsp_w_bom_me.sort_values(['date', 'PERMNO']),  # Sort CRSP with ME by date and
        right=bm.sort_values(['Date', 'PERMNO'])[['Date', 'PERMNO', 'Portfolio']],  # Select
        left_on='date',                                      # Match CRSP dates to the closes
        right_on='Date',                                     # Use BM Date as the reference f
        by='PERMNO',                                         # Join on stock ID (PERMNO) to l
        tolerance=pd.Timedelta('366d')                       # Limit matches to BM data withi
    )
    [['PERMNO', 'date', 'Portfolio', 'RET', 'ME']]           # Select essential columns: stoc
    .dropna()                                                # Remove rows with missing value
    .assign(Portfolio=lambda x: x['Portfolio'].astype(int))  # Convert portfolio numbers to i
    .rename(columns={'date': 'Date'})                        # Rename 'date' to 'Date' to ali
)
```

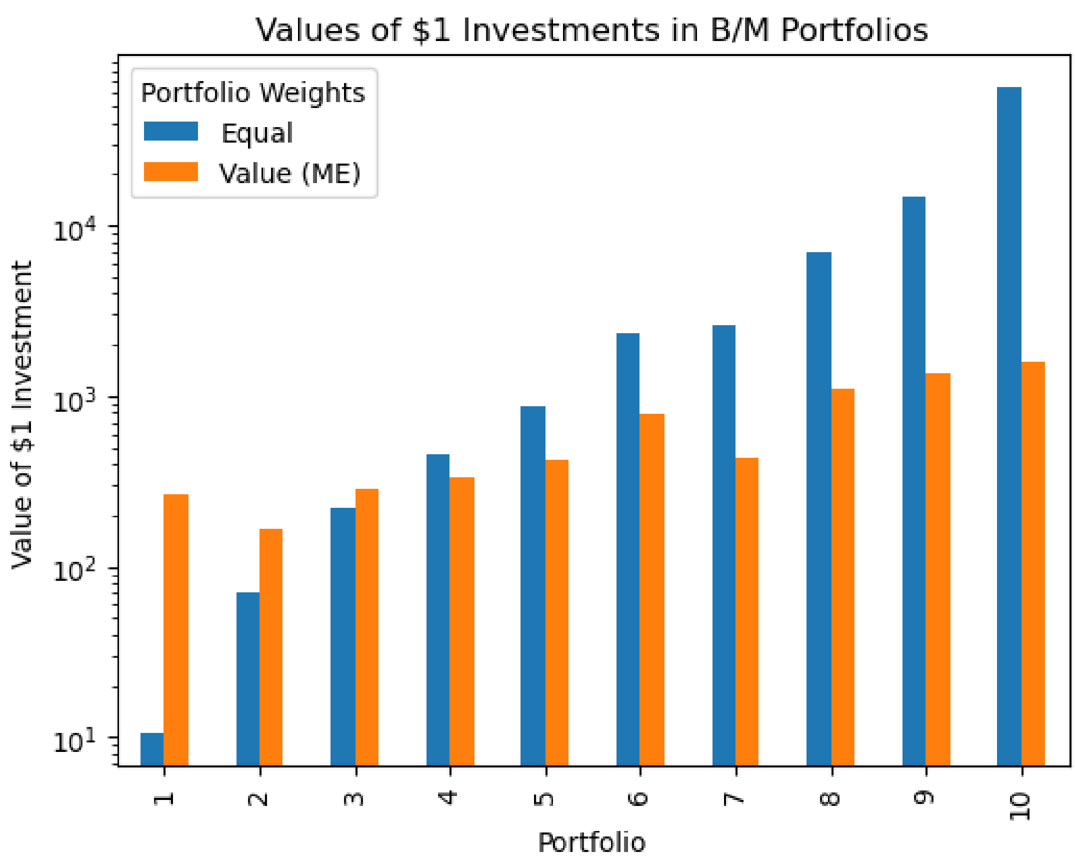
This code creates value-weighted portfolios using the `stocks` data frame and reshapes the results into a wide format. It groups the data by `Portfolio` (1 to 10) and `Date`, then calculates the value-weighted average return (`RET`) for each group, using market equity (`ME`) as weights. Next, it pivots the data so each portfolio's returns become separate columns, with `Date` as the index.

```
# Calculate value-weighted portfolio returns and reshape into wide format
portfolios_vw = (
    stocks_vw
    .groupby(['Portfolio', 'Date'])              # Group stocks by portfolio number (1-10) an
    .apply(lambda x: np.average(a=x['RET'],      # Compute value-weighted average returns usi
                            weights=x['ME']),
           include_groups=False)                 # Exclude grouping columns (Portfolio, Date)
    .unstack('Portfolio')                        # Pivot data so each portfolio's returns bec
)
```

We can briefly compare the equal- and value-weighted portfolios. *We see that the value effect is concentrated in small stocks because the equal-weighted portfolios returns are much larger than the value-weighted portfolio returns.*

```
df = pd.concat(
    objs=[portfolios.add(1).prod(), portfolios_vw.add(1).prod()],
    keys=['Equal', 'Value (ME)'],
    names=['Portfolio Weights'],
    axis=1
)
```

```
df.plot(kind='bar')
plt.semilogy()
plt.ylabel('Value of $1 Investment')
plt.title('Values of $1 Investments in B/M Portfolios')
plt.show()
```



Estimate the αs of the equal-weighted and value-weighted portfolios

> **Note**
>
> Here `portfolios` is wide, so I have to `.stack()` it before the join. This is a difference from class

```
portfolios_ff3 = (
    portfolios
    .stack(future_stack=True)
    .to_frame('RET')
    .join(ff3, how='inner')
)
```

```
import statsmodels.formula.api as smf
```

The value portfolio 10 has an α of 82 basis points! Annualized, this is $12 \times 0.0082 = 0.0984$ or almost 10% of returns not explained by market risk.

```
smf.ols(formula='I(RET-RF) ~ Q("Mkt-RF")', data=portfolios_ff3.query('(Portfolio == 10)')).f
```

Dep. Variable:	I(RET - RF)	**R-squared:**	0.506
Model:	OLS	**Adj. R-squared:**	0.505
Method:	Least Squares	**F-statistic:**	717.6
Date:	Sat, 22 Mar 2025	**Prob (F-statistic):**	2.34e-109
Time:	06:49:48	**Log-Likelihood:**	1116.4
No. Observations:	702	**AIC:**	-2229.
Df Residuals:	700	**BIC:**	-2220.
Df Model:	1		
Covariance Type:	nonrobust		

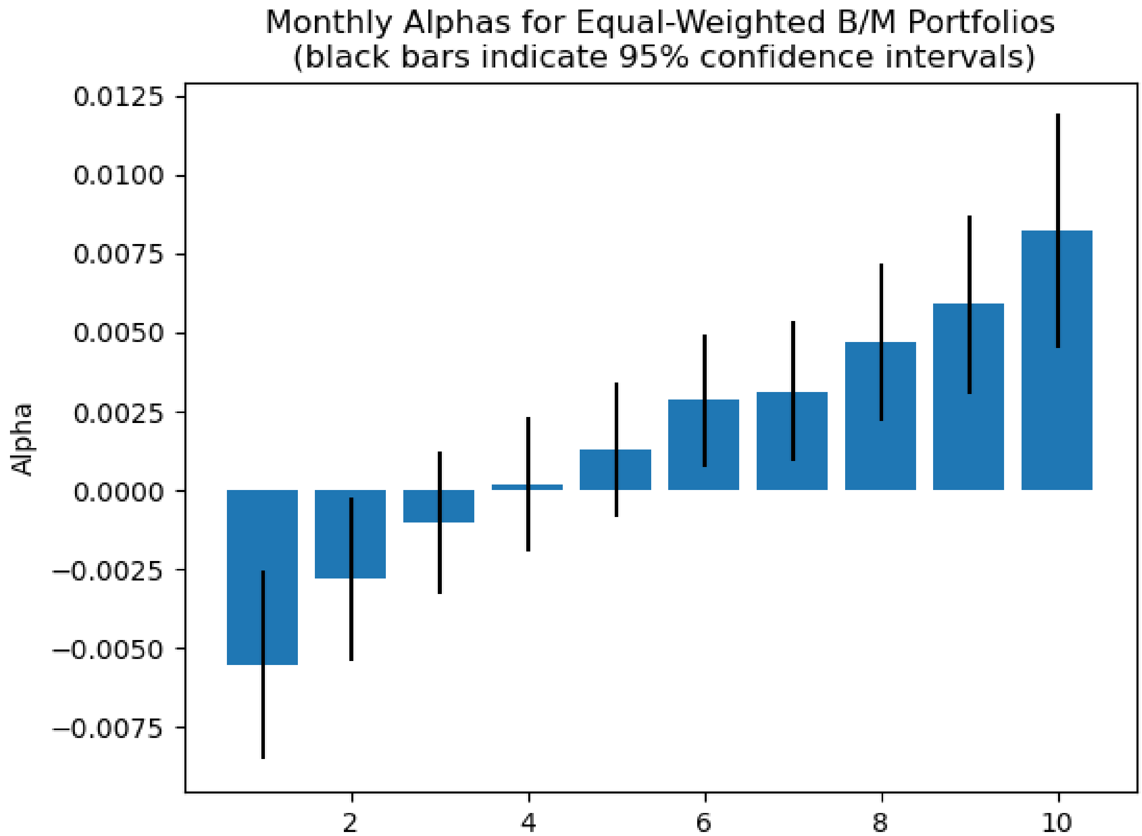
	coef	**std err**	**t**	**P> |t|**	**[0.025**	**0.975]**
Intercept	0.0082	0.002	4.370	0.000	0.005	0.012
Q(”Mkt-RF”)	1.0948	0.041	26.788	0.000	1.015	1.175

Omnibus:	268.363	**Durbin-Watson:**	1.740
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	1787.450
Skew:	1.560	**Prob(JB):**	0.00
Kurtosis:	10.167	**Cond. No.**	21.9

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

We can plot these αs (and confidence intervals) with list comprehensions!

```
fits = [
    smf.ols(formula='I(RET-RF) ~ Q("Mkt-RF")', data=portfolios_ff3.query(f'(Portfolio == {p}
    for p in range(1, 11)
]

df = (
    pd.DataFrame(
        data={
            'Alpha': [f.params['Intercept'] for f in fits],
            'Sterr': [f.bse['Intercept'] for f in fits],
            'Portfolio': range(1, 11)
        }
    )
    .set_index('Portfolio')
)

plt.bar(
    x=df.index,
    height=df['Alpha'],
    yerr=df['Sterr'].mul(1.96)
)
plt.ylabel('Alpha')
plt.title('Monthly Alphas for Equal-Weighted B/M Portfolios\n (black bars indicate 95% confid
plt.show()
```



Implement a momentum strategy

Form deciles on the 11-month returns from months $t-12$ to month $t-2$.

This code calculates 11-month momentum returns (`RET_11`) from 2 to 12 months prior using the `crsp` data frame and assigns portfolios based on those returns. It sets a multi-index with `PERMNO` and `date`, selects the `RET` column, and pivots to a wide format with stocks as columns. After sorting by date, it computes compounded returns over an 11-month rolling window using log transformations (`log1p` and `expm1`), then reshapes back to long format. It drops rows with missing `RET_11` values, creates a DataFrame, and resets the index. Finally, it assigns stocks to decile portfolios (1 to 10) based on `RET_11` for each date and shifts the `date` forward by 2 months to reflect the momentum strategy lag. That is, we receive returns in month t based on returns from month $t-12$ through $t-2$.

```
# Calculate 2-12 month momentum returns and assign portfolios
mom_02_12 = (
    crsp
```

```
    .set_index(['PERMNO', 'date'])                    # Set multi-index with stock ID (PERMNO) and
    ['RET']                                           # Select the monthly returns column
    .unstack('PERMNO')                                # Pivot to wide format, with PERMNO as colum
    .sort_index()                                     # Sort dates chronologically for consistent 
    .pipe(np.log1p)                                   # Convert returns to log scale (log(1 + r)) 
    .rolling(11)                                      # Apply an 11-month rolling window to prior 
    .sum()                                            # Sum log returns over the 11-month window
    .pipe(np.expm1)                                   # Convert summed log returns back to compoun
    .stack(future_stack=True)                         # Reshape to long format, aligning with pand
    .dropna()                                         # Remove rows with NaN from incomplete windo
    .to_frame('RET_11')                               # Create a DataFrame, naming the momentum re
    .reset_index()                                    # Move PERMNO and date back to columns from 
    .assign(
        Portfolio=lambda x: x.groupby('date')['RET_11'].transform(  # Assign stocks to decile
            lambda x: 1 + pd.qcut(x=x, q=10, labels=False)),       # Split RET_11 into 10 qua
        date=lambda x: x['date'] + pd.offsets.MonthEnd(2)          # Shift dates forward 2 mo
    )
)
```

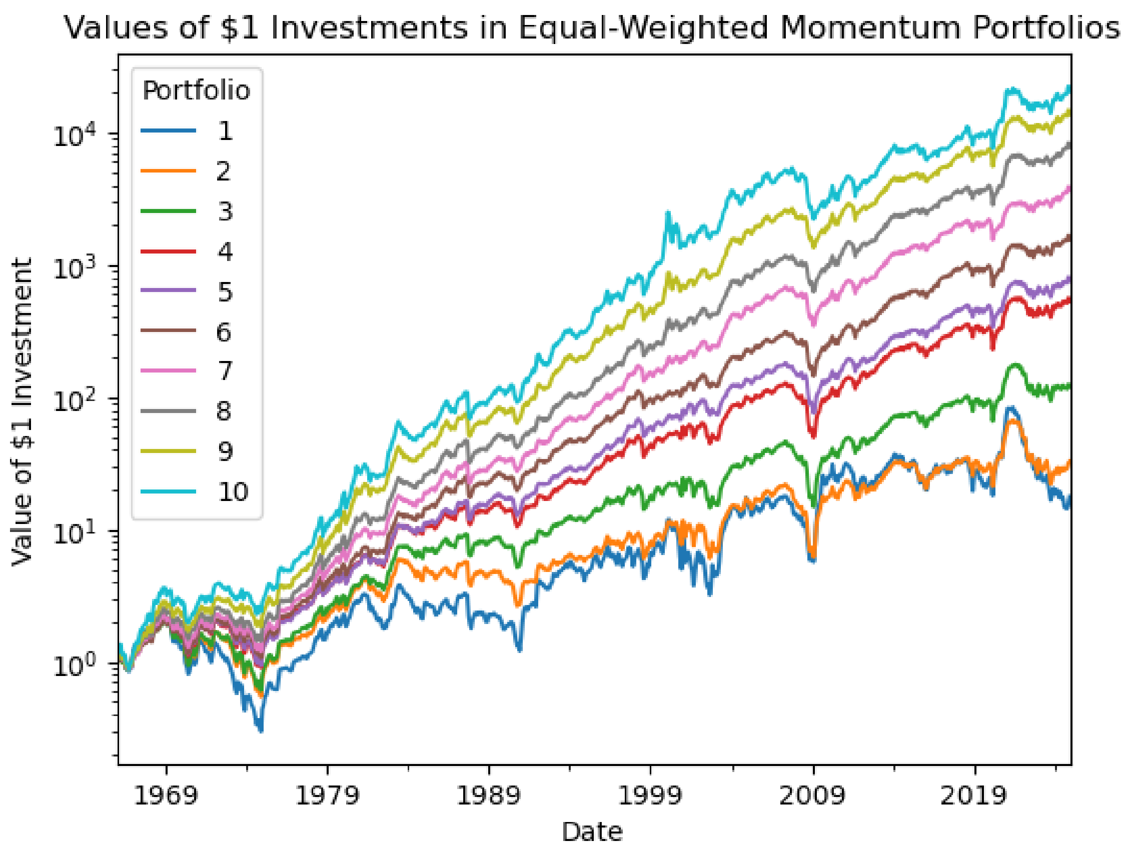
```
mom_02_12.head()
```

	date	PERMNO	RET_11	Portfolio
0	1966-01-31	10006	0.1508	5
1	1966-01-31	10014	0.2000	6
2	1966-01-31	10030	0.1489	5
3	1966-01-31	10057	0.4666	8
4	1966-01-31	10102	0.3503	7

```
stocks = (
    pd.merge(
        left=crsp,
        right=mom_02_12,
        on=['PERMNO', 'date'],
        how='inner'
    )
    [['PERMNO', 'date', 'Portfolio', 'RET']]
    .dropna()
    .rename(columns={'date': 'Date'})
)
```

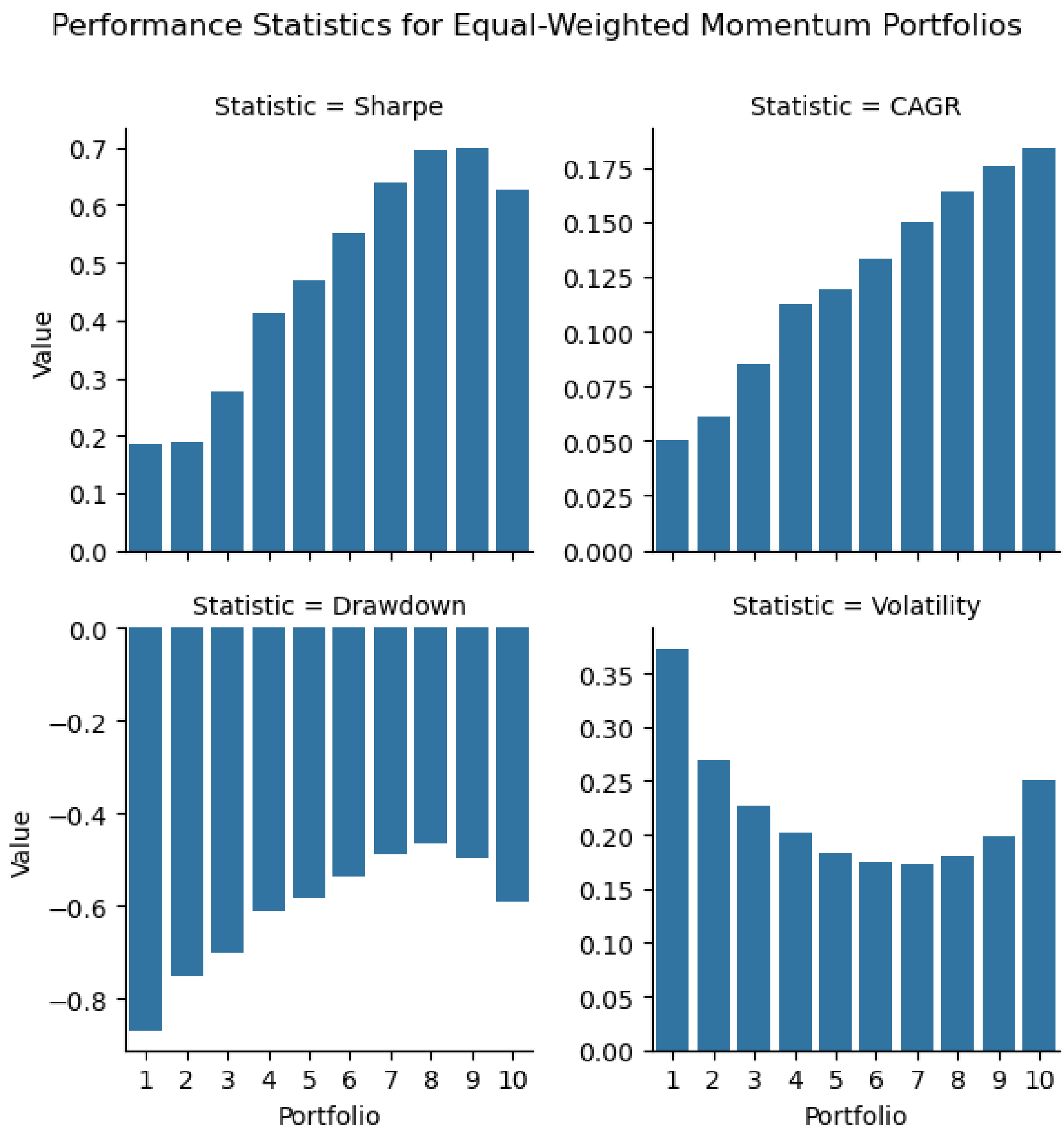
```
portfolios = (
    stocks
    .groupby(['Portfolio', 'Date'])  # Group stocks by portfolio number (1-10) and trading da
    [['RET']]                        # Isolate the returns column for portfolio-level aggrega
    .mean()                          # Average returns across stocks within each portfolio-da
    .unstack('Portfolio')            # Pivot the data to create columns for each portfolio's
    ['RET']                          # Extract the returns data, retaining portfolio numbers
)
```

```
portfolios.add(1).cumprod().plot()
plt.semilogy()
plt.ylabel('Value of $1 Investment')
plt.title('Values of $1 Investments in Equal-Weighted Momentum Portfolios')
plt.show()
```



```
portfolios_rf = portfolios.sub(ff3['RF'], axis=0)
```

```
stats = (
    portfolios_rf
    .agg(Sharpe)
    .to_frame('Sharpe')
    .join(
        portfolios
        .agg([CAGR, Drawdown, Volatility])
        .transpose()
    )
    .rename_axis(columns='Statistic')
)

df = stats.stack().to_frame('Value').reset_index()

sns.catplot(
    data=df,
    x='Portfolio',
    y='Value',
    col='Statistic',
    col_wrap=2,
    height=3,
    kind='bar',
    sharey=False
)

plt.suptitle('Performance Statistics for Equal-Weighted Momentum Portfolios', y=1.05)
plt.show()
```


Performance Statistics for Equal-Weighted Momentum Portfolios
Statistic = Sharpe
Statistic = CAGR
Statistic = Drawdown
Statistic = Volatility
Value
Value
Portfolio
Portfolio
0.0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.000
0.025
0.050
0.075
0.100
0.125
0.150
0.175
0.0
−0.2
−0.4
−0.6
−0.8
0.00
0.05
0.10
0.15
0.20
0.25
0.30
0.35
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10

Herron Topic 3 - Practice - Sec 04

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import pandas_datareader as pdr
import statsmodels.api as sm
import yfinance as yf
```

```
%precision 4
pd.options.display.float_format = '{:.4f}'.format
# %config InlineBackend.figure_format = 'retina'
```

Announcements

1. Projects:
    1. Project 1:
        1. I am still grading
        2. I plan to finish grading by Friday
    2. Project 2:
        1. Due on Friday, 3/28, at 11:59 PM
        2. We will use class time next week for group work
        3. Ask your questions here: https://northeastern.instructure.com/courses/207607/discussion_topics/2738173
2. Assessments:
    1. ***Both assessments are in class on Tuesday, 4/15***
    2. Programming assessment: Should take 15 minutes and will be based on the questions here:
    3. MSFQ assessment (only for MSFQ students): Should take 45 minutes and based on the 5 required MSFQ courses (standby for a Canvas announcement with more details after I discuss it with the MSFQ program director)

Five-Minute Review

Quantitative value investing is a systematic investment strategy that combines the principles of value investing with data-driven, quantitative techniques. It involves identifying undervalued securities—those trading below their intrinsic value—using predefined, measurable criteria rather than subjective judgment. Common criteria are price/earnings (P/E) and price/book (P/B) ratios. Here, we will use the book-to-market equity ratio (B/M), which is common in the academic literature and quantitative investing.

The key concepts in this topic are creating metrics, forming portfolios, and backtesting the strategy.

Practice

Re-implement the value strategy from the lecture notebook

Read the data

This code reads stock market data from a file called `crsp.csv` and prepares it for analysis. It loads the data into a data frame, makes sure the dates are in the right format, and marks certain letters (`A`, `B`, and `C`) as missing values. Then, it filters the data to include only records from 1965 or later, sorts it by company and date, and adds two new columns: one for market equity (calculated as the number of shares times the stock price, converted to millions), and another to adjust all dates to the end of each month.

```
crsp = (
    pd.read_csv(
        filepath_or_buffer='crsp.csv',  # Read data from crsp.csv file
        parse_dates=['date'],           # Convert 'date' column to datetime format
        na_values=['A', 'B', 'C']       # Treat 'A', 'B', 'C' as missing values (NaN)
    )
    .query('date >= 1965')              # Filter to keep only data from 1965 onward
    .sort_values(['PERMNO', 'date'])    # Sort by company identifier (PERMNO) and date
    .assign(
        ME=lambda x: x['SHROUT'] * x['PRC'].abs() / 1_000,  # Calculate market equity (ME) as
        date=lambda x: x['date'] + pd.offsets.MonthEnd(0)   # Adjust dates to the last day of
    )
)
```

```
crsp.head()
```

	PERMNO	date	SHRCD	PRC	RET	SHROUT	ME
0	10000	1986-01-31	10	-4.3750	NaN	3680.0000	16.1000
1	10000	1986-02-28	10	-3.2500	-0.2571	3680.0000	11.9600
2	10000	1986-03-31	10	-4.4375	0.3654	3680.0000	16.3300
3	10000	1986-04-30	10	-4.0000	-0.0986	3793.0000	15.1720
4	10000	1986-05-31	10	-3.1094	-0.2227	3793.0000	11.7939

This code brings in financial data from a file named `compustat.csv` and prepares it for analysis. It loads the data into a data frame, formats dates, and filters out any records before 1965. Then, it sorts the data by company, fiscal year, and date. Finally, it removes any duplicate entries for the same company and fiscal year, keeping only the most recent record.

```
compustat = (
    pd.read_csv(
        filepath_or_buffer='compustat.csv',  # Read data from compustat.csv file
        parse_dates=['datadate']              # Convert 'datadate' column to datetime format
    )
    .query('datadate >= 1965')                # Filter to keep only data from 1965 onward
    .sort_values(['LPERMNO', 'fyear', 'datadate'])  # Sort by company identifier (LPERMNO), f
    .drop_duplicates(subset=['LPERMNO', 'fyear'], keep='last')  # Keep only the latest recor
)
```

```
compustat.head()
```

	GVKEY	LPERMNO	datadate	fyear	indfmt	consol	popsrc	datafmt	curcd	ceq
165681	13007	10000	1986-10-31	1986.0000	INDL	C	D	STD	USD	0.418
165586	12994	10001	1986-06-30	1986.0000	INDL	C	D	STD	USD	5.432
165587	12994	10001	1987-06-30	1987.0000	INDL	C	D	STD	USD	5.369
165588	12994	10001	1988-06-30	1988.0000	INDL	C	D	STD	USD	5.512
165589	12994	10001	1989-06-30	1989.0000	INDL	C	D	STD	USD	6.321

About 64 percent of firm-years have December fiscal-year ends.

```
compustat['datadate'].dt.month.value_counts() / compustat['datadate'].shape[0]
```

```
datadate
12   0.6401
6    0.0723
```

```
9     0.0609
3     0.0518
10    0.0341
1     0.0334
8     0.0203
7     0.0187
5     0.0187
4     0.0179
11    0.0173
2     0.0146
Name: count, dtype: float64
```

Create the interim data frames

This code finds the market value of equity (`ME`) as of December each year from the `crsp` data frame (to match with book value of equity `ceq` from the `compustat` data frame below). It filters out any rows where market equity is zero or negative, then sorts the data by company and date. Next, it groups the data by company and year, focusing on December values. Finally, it selects the last market equity value for each company in each year, giving us the December `ME`.

```
mve = (
    crsp
    .query('ME > 0')                            # Filter for positive market equity values
    .sort_values(['PERMNO', 'date'])            # Sort by company identifier (PERMNO) and da
    .groupby(by=['PERMNO', pd.Grouper(key='date', freq='YE-DEC')])  # Group by company and ye
    [['ME']]                                    # Select the market equity (ME) column
    .last()                                     # Take the last ME value for each group (Dec
)
```

```
mve.head()
```

		M
PERMNO	date	
10000	1986-12-31	1.
	1987-12-31	0.
10001	1986-12-31	6.
	1987-12-31	5.
	1988-12-31	6.

This code finds the book value of equity (`ceq`) as of December each year from the `compustat` data frame. It filters out any rows where common equity is zero or negative, then sorts the data by company and date. Next, it groups the data by company and year, focusing on December values. Finally, it selects the last common equity value for each company in each year, giving us the December `ceq`.

```
bve = (
    compustat
    .query('ceq > 0')                           # Filter for positive common equity values
    .sort_values(['LPERMNO', 'datadate'])       # Sort by company identifier (LPERMNO) and d
    .groupby(by=['LPERMNO', pd.Grouper(key='datadate', freq='YE-DEC')])  # Group by company a
    [['ceq']]                                   # Select the common equity (ceq) column
    .last()                                     # Take the last CEQ value for each group (De
)
```

```
bve.head()
```

		ce
LPERMNO	datadate	
10000	1986-12-31	0.
10001	1986-12-31	5.
	1987-12-31	5.
	1988-12-31	5.
	1989-12-31	6.

Combine the data frames and form portfolios

This code shows a simple example of how to form portfolios using the `pd.qcut()` function. It takes a sequence of numbers from 0 to 9 and splits them into 2 equal groups (quantiles), assigning them to either a low or high category. The `labels=False` argument gives us numeric group identifiers (0 or 1), and adding 1 shifts these to 1 or 2, which can represent portfolio numbers.

```
1 + pd.qcut(x=np.arange(10), q=2, labels=False)
```

```
array([1, 1, 1, 1, 1, 2, 2, 2, 2, 2])
```

> **Note**
>
> In class, in the following cell, I used multiple cursors to simultaneously type two `.rename_axis(index=['PERMNO', 'Date'])` methods. If you hold down the `CTRL` key, every mouse click will generate a new cursor. Each cursor will do the same action (e.g., typing, highlighting, and deleting). More here.

This code creates portfolios based on book-to-market (`BM`) ratios using the `bve` and `mve` data frames. It starts by aligning the company and date indexes of both datasets and merges them, keeping only rows with matching data. Next, it calculates the `BM` ratio by dividing book value of equity (`ceq`) by market value of equity (`ME`). After resetting the index, it shifts the dates forward by 7 months to time the portfolio formation correctly, giving 6 months between `BM` calculation and buying stocks at the end of June to receive July returns. Finally, for each date, it sorts the `BM` ratios into 10 equal groups (deciles) and assigns portfolio numbers from 1 to 10.

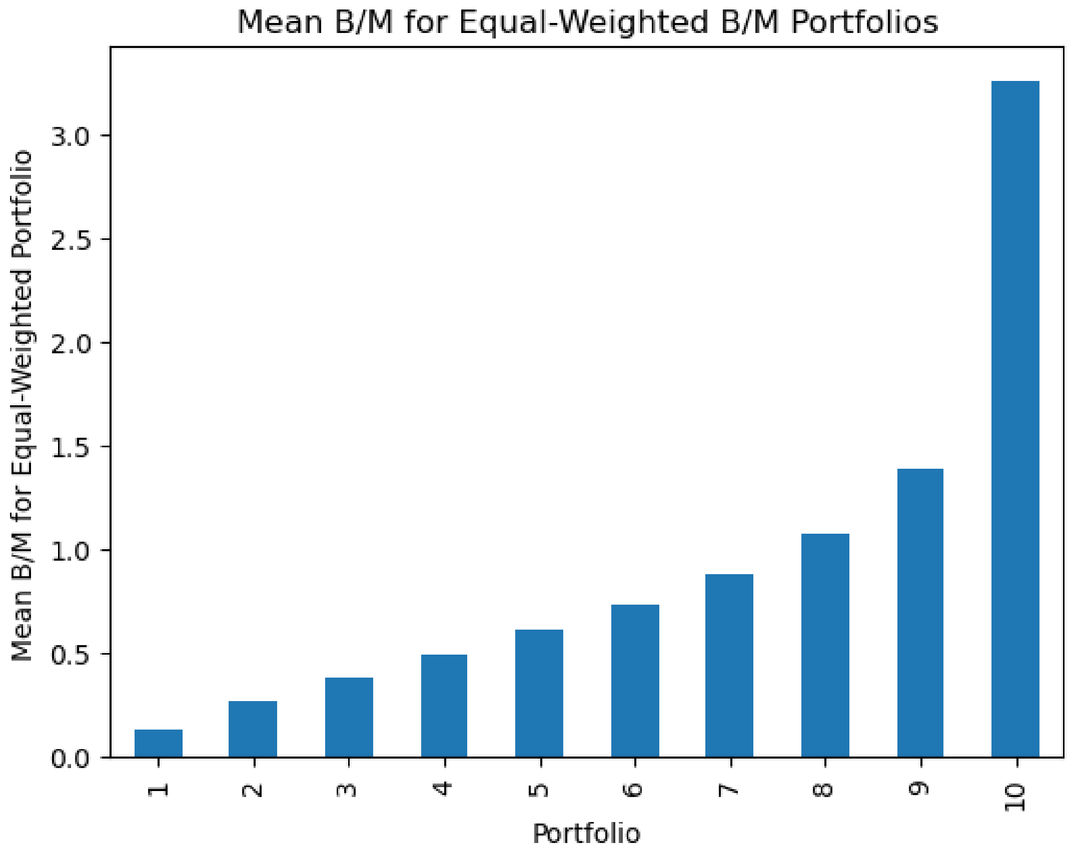
```
bm = (
    bve
    .rename_axis(index=['PERMNO', 'Date'])  # Standardize index names to PERMNO (company ID)
    .join(
        other=mve.rename_axis(index=['PERMNO', 'Date']),  # Combine with market equity data,
        how='inner'                                     # Use inner join to retain only records wi
    )
    .reset_index()                                      # Flatten the index into columns for easie
    .assign(
        Date=lambda x: x['Date'] + pd.offsets.MonthEnd(7),  # Shift dates forward 7 months (
        BM=lambda x: x['ceq'] / x['ME'],                    # Compute book-to-market ratio as
        Portfolio=lambda x: x.groupby('Date')['BM'].transform(  # Assign stocks to portfolio
            lambda x: 1 + pd.qcut(x=x, q=10, labels=False)  # Divide BM into 10 quantiles (0
        )
    )
)
```

```
bm.head()
```

	PERMNO	Date	ceq	ME	BM	Portfolio
0	10000	1987-07-31	0.4180	1.9816	0.2109	2
1	10001	1987-07-31	5.4320	6.9370	0.7830	7
2	10001	1988-07-31	5.3690	5.8280	0.9212	7
3	10001	1989-07-31	5.5120	6.3623	0.8664	7
4	10001	1990-07-31	6.3210	10.3477	0.6109	5

This code creates a bar chart showing the average book-to-market (`BM`) ratio for each portfolio using the `portfolios` data frame. It groups the data by portfolio number (1 to 10), calculates the mean `BM` ratio for each group, and then plots these averages as bars. As expected, we see that `BM` rises from portfolio 1 to 10.

```
(
    bm
    .groupby(by=['Portfolio', 'Date'])  # Group data by portfolio number and date
    ['BM']                              # Select the book-to-market (BM) column
    .mean()                             # Calculate the mean BM for each portfolio-date combi
    .groupby(by='Portfolio')            # Group again by portfolio to average across all date
    .mean()                             # Calculate the time-averaged mean BM for each portfo
    .plot(kind='bar')                   # Plot the results as a bar chart
)
plt.ylabel('Mean B/M for Equal-Weighted Portfolio')
plt.title('Mean B/M for Equal-Weighted B/M Portfolios')
plt.show()
```



This code combines monthly stock returns from the `crsp` data frame with the *most recent* portfolio assignments from the `bm` data frame. The `tolerance` argument tells the merge to only use portfolio assignments within 366 days to avoid stale data.

```
stocks = (
    pd.merge_asof(
        left=crsp.sort_values(['date', 'PERMNO']),    # Sort CRSP data by date and company II
        right=bm.sort_values(['Date', 'PERMNO']),     # Sort BM data by date and company ID f
        left_on='date',                               # Match CRSP dates to the closest prio
        right_on='Date',                              # Use BM dates as the reference for the
        by='PERMNO',                                  # Merge on company ID (PERMNO) to link
        tolerance=pd.Timedelta('366d')                # Limit matches to portfolio assignmen
    )
    [['PERMNO', 'date', 'Portfolio', 'RET']]          # Select key columns: company ID, date
    .dropna()                                         # Remove rows with missing values (e.g
    .assign(Portfolio=lambda x: x['Portfolio'].astype(int))  # Convert portfolio numbers to i
    .rename(columns={'date': 'Date'})                 # Rename 'date' column to 'Date' for co
)
```

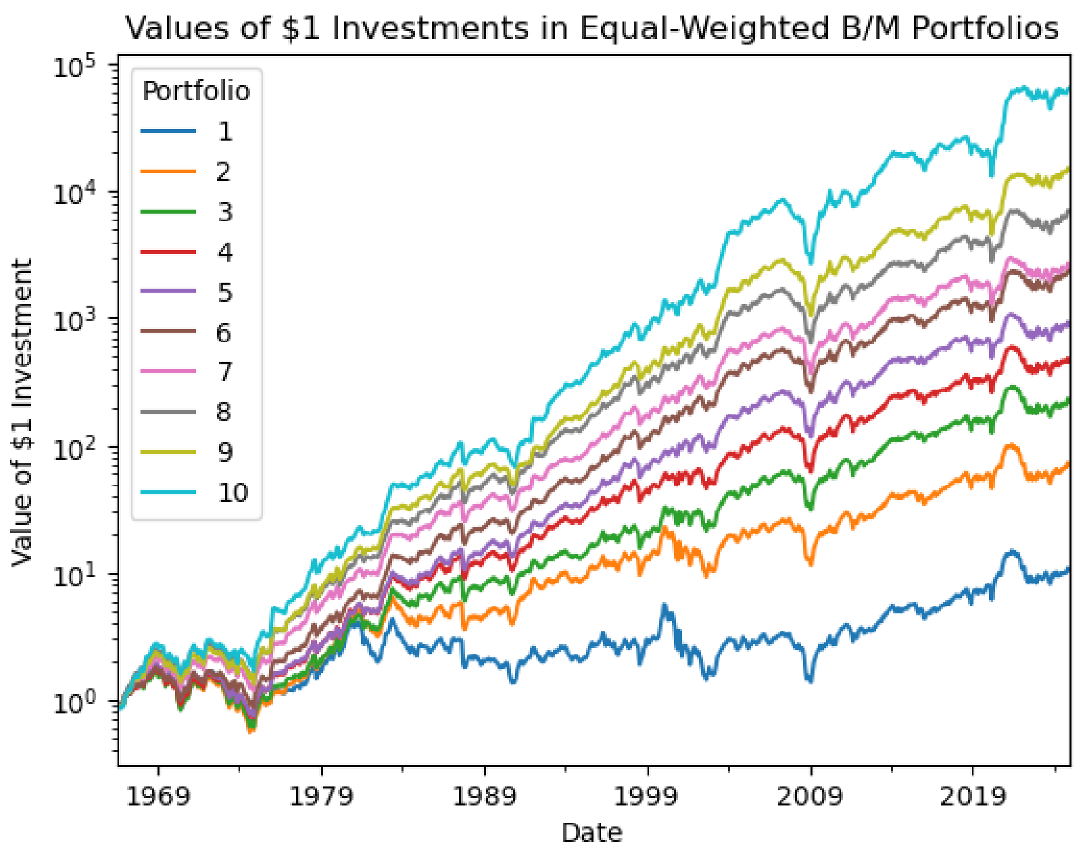
This code creates equal-weighted portfolios using the `stocks` data frame. It groups the data by portfolio number (`Portfolio`) and date (`Date`), then calculates the mean return (`RET`) for each group. This averaging process gives each stock within a portfolio the same weight, producing equal-weighted portfolio returns.

> **Note**
>
> Here I made `portfolios` a wide data frame (i.e., portfolio numbers in columns) to simplify plots and calculations. This is a difference from class

```
portfolios = (
    stocks
    .groupby(['Portfolio', 'Date'])  # Group stocks by portfolio number (1-10) and trading da
    [['RET']]                        # Isolate the returns column for portfolio-level aggrega
    .mean()                          # Average returns across stocks within each portfolio-da
    .unstack('Portfolio')            # Pivot the data to create columns for each portfolio's
    ['RET']                          # Extract the returns data, retaining portfolio numbers
)
```

Finally, we can see that the value portfolio 10 outperforms the growth portfolio 1!

```
portfolios.add(1).cumprod().plot()
plt.semilogy()
plt.ylabel('Value of $1 Investment')
plt.title('Values of $1 Investments in Equal-Weighted B/M Portfolios')
plt.show()
```



Backtest the strategy

We will need the Fama and French (1993) factors for the risk-free rate.

> **Note**
>
> Here I set the `Date` index to simplify merges/joins below. This is a difference from class

```
ff3 = (
    pdr.DataReader(
        name='F-F_Research_Data_Factors',
        data_source='famafrench',
        start='1900'
    )
    [0]
    .div(100)
    .reset_index()
    .assign(Date=lambda x: x['Date'].dt.to_timestamp(how='end').dt.normalize()) # convert mon
    .set_index('Date')
)
```

```
C:\Users\r.herron\AppData\Local\Temp\ipykernel_20756\1889697072.py:2: FutureWarning: The argu
  pdr.DataReader(
C:\Users\r.herron\AppData\Local\Temp\ipykernel_20756\1889697072.py:2: FutureWarning: The argu
  pdr.DataReader(
```

```
portfolios_rf = portfolios.sub(ff3['RF'], axis=0)
```

```
def CAGR(x, ann_fac=12):
    return (1 + x).prod() ** (ann_fac / x.count()) - 1
```

```
def Sharpe(x, ann_fac=np.sqrt(12)):
    return ann_fac * x.mean() / x.std()
```

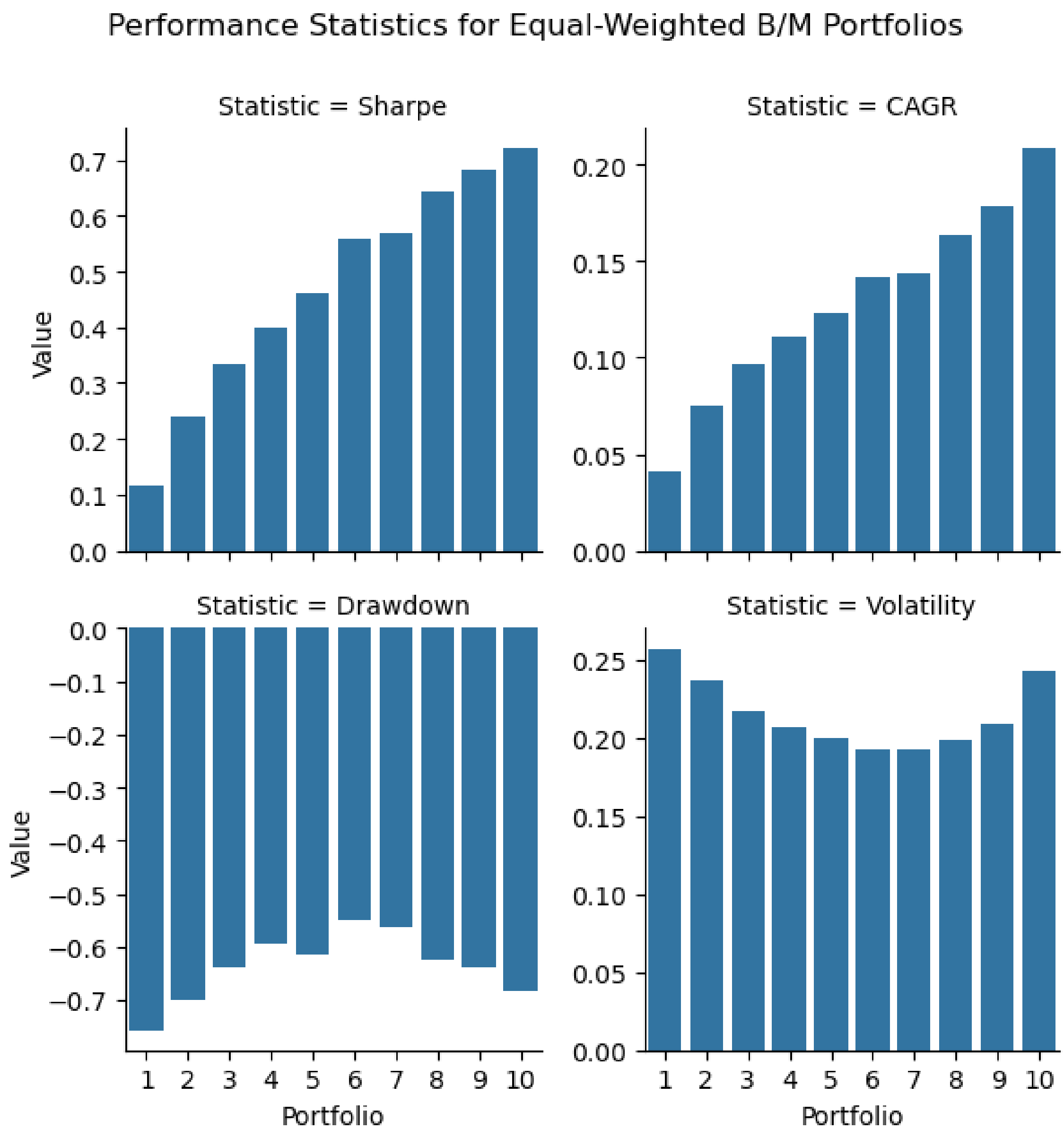
```
def Volatility(x, ann_fac=np.sqrt(12)):
    return ann_fac * x.std()
```

```
def Drawdown(x):
    price = x.add(1).cumprod()
    return (price / price.cummax()).min() - 1
```

The high B/M have higher Sharper ratios and CAGRs than low B/M portfolios. However, the risk relation is not monotonic.

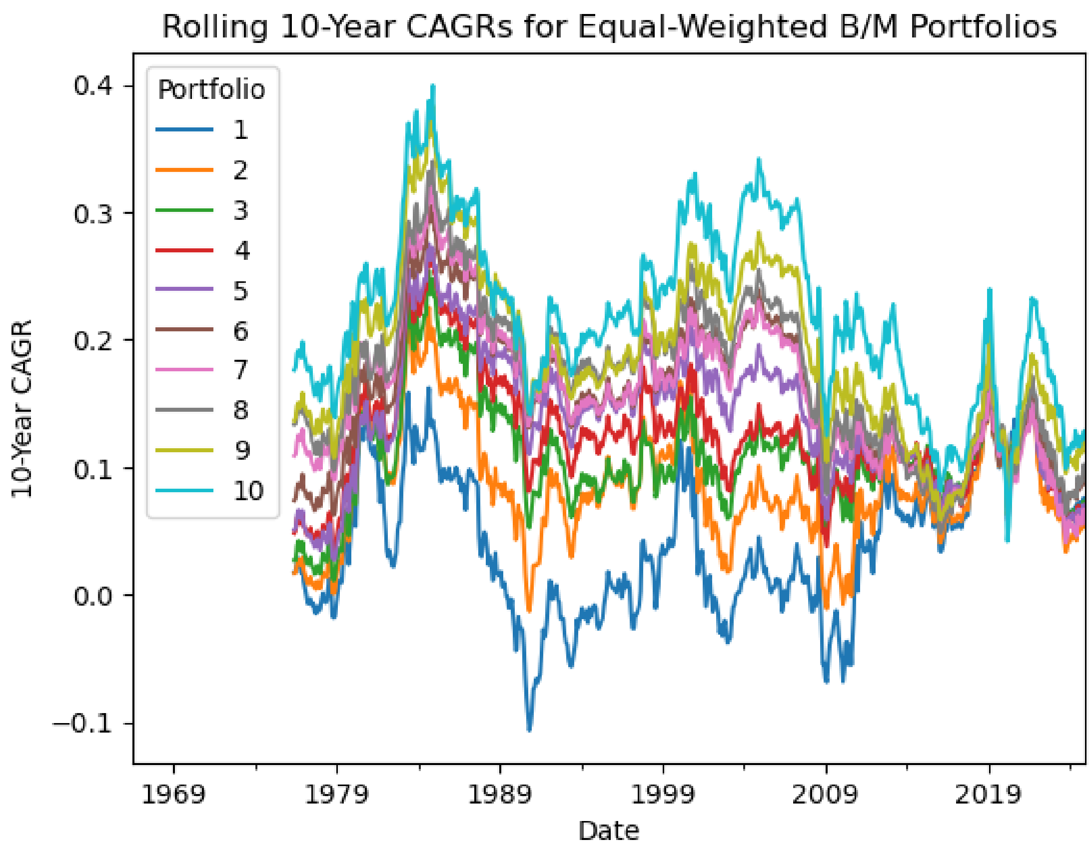
```
import seaborn as sns
```

```
stats = (
    portfolios_rf
    .agg(Sharpe)
    .to_frame('Sharpe')
    .join(
        portfolios
        .agg([CAGR, Drawdown, Volatility])
        .transpose()
    )
    .rename_axis(columns='Statistic')
)

df = stats.stack().to_frame('Value').reset_index()

sns.catplot(
    data=df,
    x='Portfolio',
    y='Value',
    col='Statistic',
    col_wrap=2,
    height=3,
    kind='bar',
    sharey=False
)

plt.suptitle('Performance Statistics for Equal-Weighted B/M Portfolios', y=1.05)
plt.show()
```



Also, this outperformance narrows over the past few decades.

```
portfolios.rolling(window=120).apply(CAGR).plot()
plt.ylabel('10-Year CAGR')
plt.title('Rolling 10-Year CAGRs for Equal-Weighted B/M Portfolios')
plt.show()
```



Re-implement the value strategy from the lecture notebook *with value-weighted portfolios*

We need to do two things:

1. Add the beginning of month market values of equity to use as portfolio weights
2. Replace `.mean()` with `np.average()` to calculate weighted means

This code prepares stock data from the `crsp` data frame by adding beginning-of-month market equity (`ME`) and merging it with portfolio assignments from the `bm` data frame. It first selects `PERMNO`, `date`, and `RET` from `crsp`, then merges it with `ME` data shifted forward by 1 month using an inner join to align current returns with next month's market equity. Next, it sorts the CRSP data by `date` and `PERMNO`, and merges it with a subset of `bm` containing only `Date`, `PERMNO`, and `Portfolio`, sorted by `Date` and `PERMNO`, matching each stock's date to the closest prior `Date` in `bm` within a 366-day tolerance. Finally, it keeps `PERMNO`, `date`, `Portfolio`, `RET`, and `ME`, drops missing values, converts `Portfolio` to integers, and renames `date` to `Date` for consistency.

```
# Add beginning-of-month market equity (ME) to CRSP data
crsp_w_bom_me = (
    crsp
    [['PERMNO', 'date', 'RET']]                  # Select stock ID, date, and returns from CR
    .merge(
        right=crsp[['PERMNO', 'date', 'ME']].assign(  # Select stock ID, date, and ME, shift
            date=lambda x: x['date'] + pd.offsets.MonthEnd(1)),
        how='inner'                              # Inner join to align current month with nex
    )
)

# Merge CRSP data with book-to-market portfolio assignments
stocks_vw = (
    pd.merge_asof(
        left=crsp_w_bom_me.sort_values(['date', 'PERMNO']),  # Sort CRSP with ME by date and
        right=bm.sort_values(['Date', 'PERMNO'])[['Date', 'PERMNO', 'Portfolio']],  # Select
        left_on='date',                                      # Match CRSP dates to the closes
        right_on='Date',                                     # Use BM Date as the reference i
        by='PERMNO',                                         # Join on stock ID (PERMNO) to l
        tolerance=pd.Timedelta('366d')                       # Limit matches to BM data withi
    )
    [['PERMNO', 'date', 'Portfolio', 'RET', 'ME']]           # Select essential columns: stoc
    .dropna()                                                # Remove rows with missing value
    .assign(Portfolio=lambda x: x['Portfolio'].astype(int))  # Convert portfolio numbers to i
    .rename(columns={'date': 'Date'})                        # Rename 'date' to 'Date' to ali
)
```

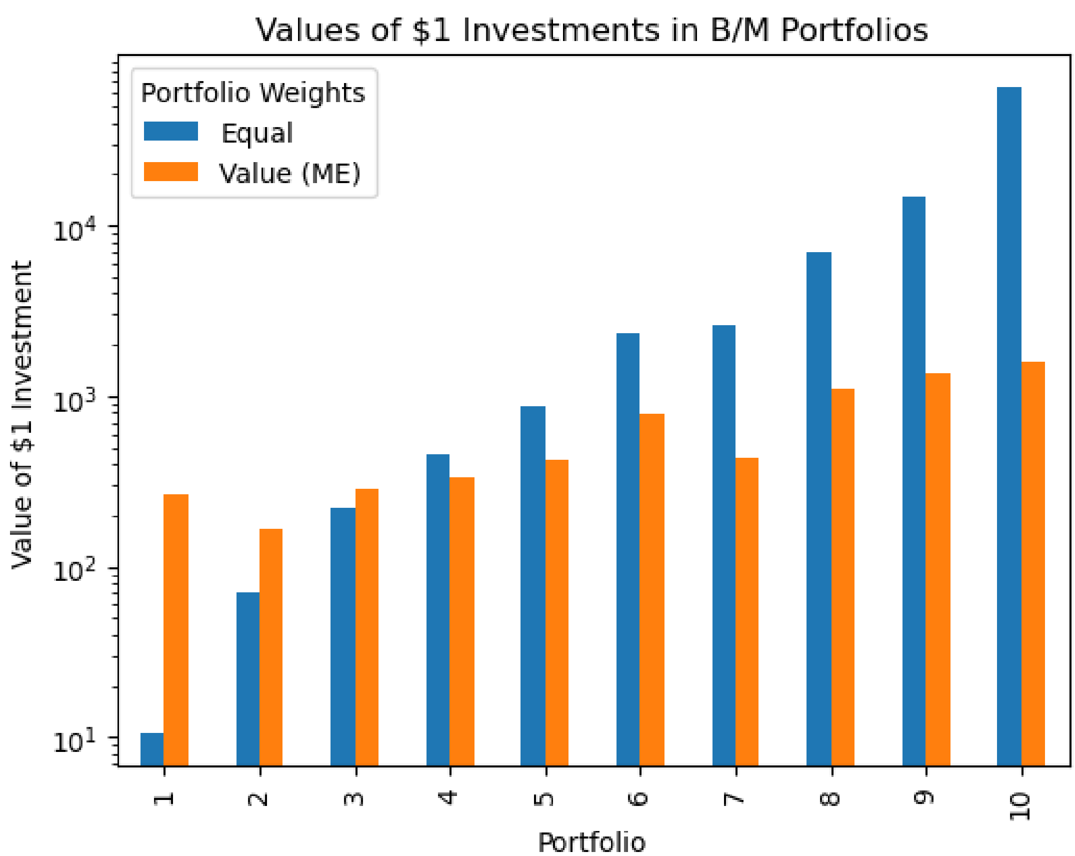
This code creates value-weighted portfolios using the `stocks` data frame and reshapes the results into a wide format. It groups the data by `Portfolio` (1 to 10) and `Date`, then calculates the value-weighted average return (`RET`) for each group, using market equity (`ME`) as weights. Next, it pivots the data so each portfolio's returns become separate columns, with `Date` as the index.

```
# Calculate value-weighted portfolio returns and reshape into wide format
portfolios_vw = (
    stocks_vw
    .groupby(['Portfolio', 'Date'])              # Group stocks by portfolio number (1-10) an
    .apply(lambda x: np.average(a=x['RET'],      # Compute value-weighted average returns usi
                                weights=x['ME']),
           include_groups=False)                 # Exclude grouping columns (Portfolio, Date)
    .unstack('Portfolio')                        # Pivot data so each portfolio's returns bec
)
```

We can briefly compare the equal- and value-weighted portfolios. *We see that the value effect is concentrated in small stocks because the equal-weighted portfolios returns are much larger than the value-weighted portfolio returns.*

```
df = pd.concat(
    objs=[portfolios.add(1).prod(), portfolios_vw.add(1).prod()],
    keys=['Equal', 'Value (ME)'],
    names=['Portfolio Weights'],
    axis=1
)
```

```
df.plot(kind='bar')
plt.semilogy()
plt.ylabel('Value of $1 Investment')
plt.title('Values of $1 Investments in B/M Portfolios')
plt.show()
```



Estimate the αs of the equal-weighted and value-weighted portfolios

> **Note**
>
> Here `portfolios` is wide, so I have to `.stack()` it before the join. This is a difference from class

```
portfolios_ff3 = (
    portfolios
    .stack(future_stack=True)
    .to_frame('RET')
    .join(ff3, how='inner')
)
```

```
import statsmodels.formula.api as smf
```

The value portfolio 10 has an α of 82 basis points! Annualized, this is $12 \times 0.0082 = 0.0984$ or almost 10% of returns not explained by market risk.

```
smf.ols(formula='I(RET-RF) ~ Q("Mkt-RF")', data=portfolios_ff3.query('(Portfolio == 10)')).fi
```

Dep. Variable:	I(RET - RF)	**R-squared:**	0.506
Model:	OLS	**Adj. R-squared:**	0.505
Method:	Least Squares	**F-statistic:**	717.6
Date:	Sat, 22 Mar 2025	**Prob (F-statistic):**	2.34e-109
Time:	06:49:48	**Log-Likelihood:**	1116.4
No. Observations:	702	**AIC:**	-2229.
Df Residuals:	700	**BIC:**	-2220.
Df Model:	1		
Covariance Type:	nonrobust		

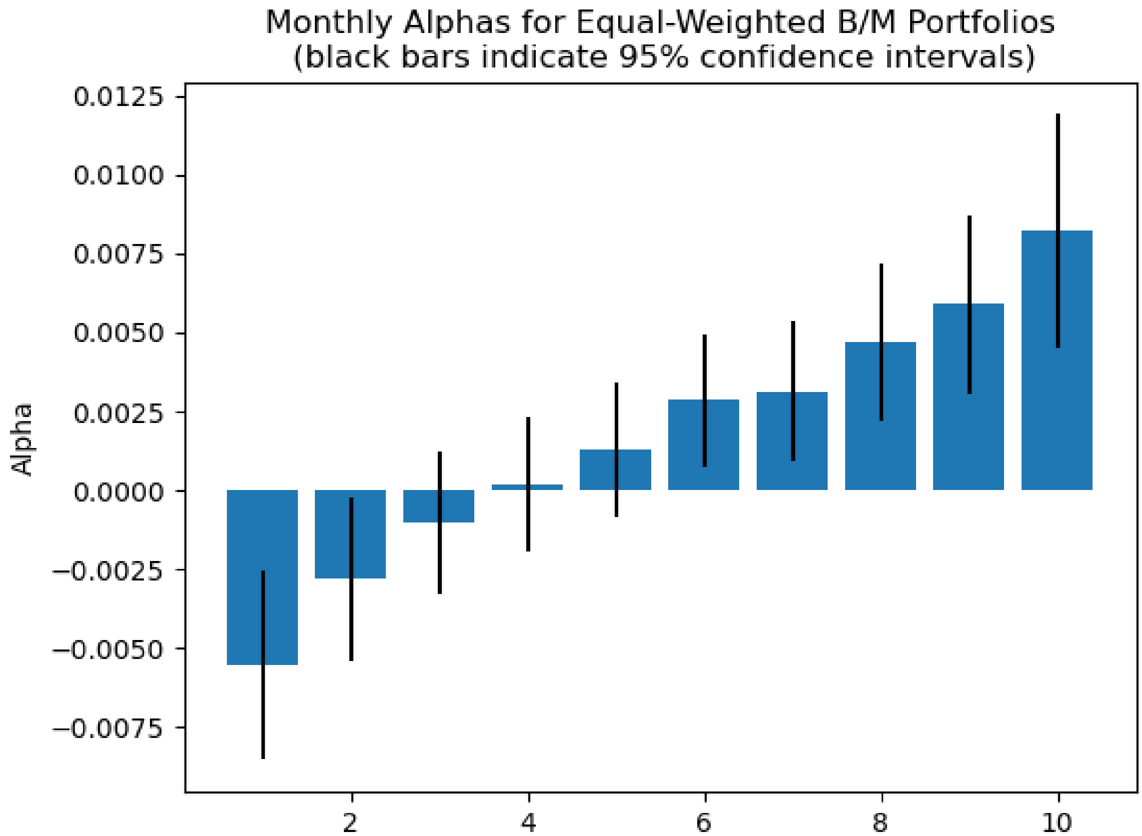
	coef	**std err**	**t**	**P>|t|**	**[0.025**	**0.975]**
Intercept	0.0082	0.002	4.370	0.000	0.005	0.012
Q("Mkt-RF")	1.0948	0.041	26.788	0.000	1.015	1.175

Omnibus:	268.363	**Durbin-Watson:**	1.740
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	1787.450
Skew:	1.560	**Prob(JB):**	0.00
Kurtosis:	10.167	**Cond. No.**	21.9

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

We can plot these αs (and confidence intervals) with list comprehensions!

```
fits = [
    smf.ols(formula='I(RET-RF) ~ Q("Mkt-RF")', data=portfolios_ff3.query(f'(Portfolio == {p})
    for p in range(1, 11)
]

df = (
    pd.DataFrame(
        data={
            'Alpha': [f.params['Intercept'] for f in fits],
            'Sterr': [f.bse['Intercept'] for f in fits],
            'Portfolio': range(1, 11)
        }
    )
    .set_index('Portfolio')
)

plt.bar(
    x=df.index,
    height=df['Alpha'],
    yerr=df['Sterr'].mul(1.96)
)
plt.ylabel('Alpha')
plt.title('Monthly Alphas for Equal-Weighted B/M Portfolios\n (black bars indicate 95% confid
plt.show()
```



Implement a momentum strategy

Form deciles on the 11-month returns from months $t-12$ to month $t-2$.

This code calculates 11-month momentum returns (`RET_11`) from 2 to 12 months prior using the `crsp` data frame and assigns portfolios based on those returns. It sets a multi-index with `PERMNO` and `date`, selects the `RET` column, and pivots to a wide format with stocks as columns. After sorting by date, it computes compounded returns over an 11-month rolling window using log transformations (`log1p` and `expm1`), then reshapes back to long format. It drops rows with missing `RET_11` values, creates a DataFrame, and resets the index. Finally, it assigns stocks to decile portfolios (1 to 10) based on `RET_11` for each date and shifts the `date` forward by 2 months to reflect the momentum strategy lag. That is, we receive returns in month t based on returns from month $t-12$ through $t-2$.

```
# Calculate 2-12 month momentum returns and assign portfolios
mom_02_12 = (
    crsp
```

```
    .set_index(['PERMNO', 'date'])                # Set multi-index with stock ID (PERMNO) and
    ['RET']                                       # Select the monthly returns column
    .unstack('PERMNO')                            # Pivot to wide format, with PERMNO as colum
    .sort_index()                                 # Sort dates chronologically for consistent 
    .pipe(np.log1p)                               # Convert returns to log scale (log(1 + r)) 
    .rolling(11)                                  # Apply an 11-month rolling window to prior 
    .sum()                                        # Sum log returns over the 11-month window
    .pipe(np.expm1)                               # Convert summed log returns back to compoun
    .stack(future_stack=True)                     # Reshape to long format, aligning with pand
    .dropna()                                     # Remove rows with NaN from incomplete windo
    .to_frame('RET_11')                           # Create a DataFrame, naming the momentum re
    .reset_index()                                # Move PERMNO and date back to columns from 
    .assign(
        Portfolio=lambda x: x.groupby('date')['RET_11'].transform(  # Assign stocks to decile
            lambda x: 1 + pd.qcut(x=x, q=10, labels=False)),       # Split RET_11 into 10 qua
        date=lambda x: x['date'] + pd.offsets.MonthEnd(2)          # Shift dates forward 2 mo
    )
)
```

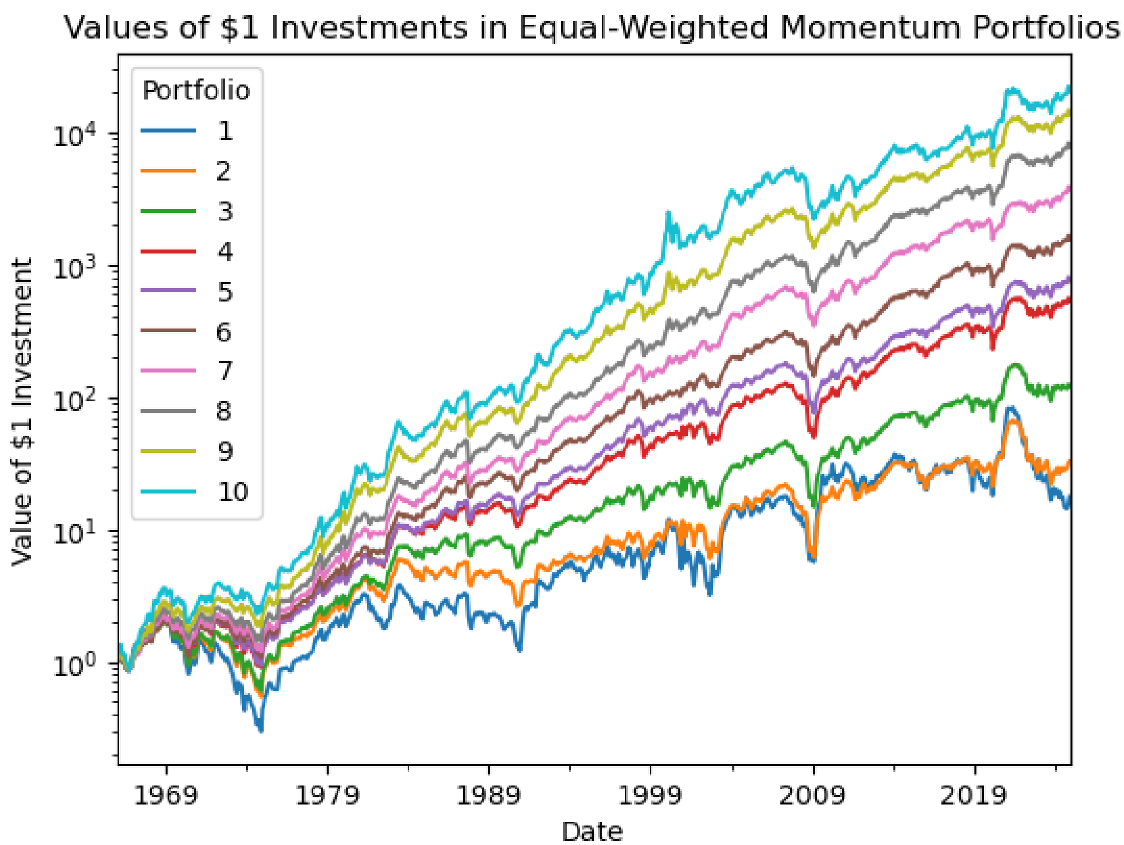
```
mom_02_12.head()
```

	date	PERMNO	RET_11	Portfolio
0	1966-01-31	10006	0.1508	5
1	1966-01-31	10014	0.2000	6
2	1966-01-31	10030	0.1489	5
3	1966-01-31	10057	0.4666	8
4	1966-01-31	10102	0.3503	7

```
stocks = (
    pd.merge(
        left=crsp,
        right=mom_02_12,
        on=['PERMNO', 'date'],
        how='inner'
    )
    [['PERMNO', 'date', 'Portfolio', 'RET']]
    .dropna()
    .rename(columns={'date': 'Date'})
)
```

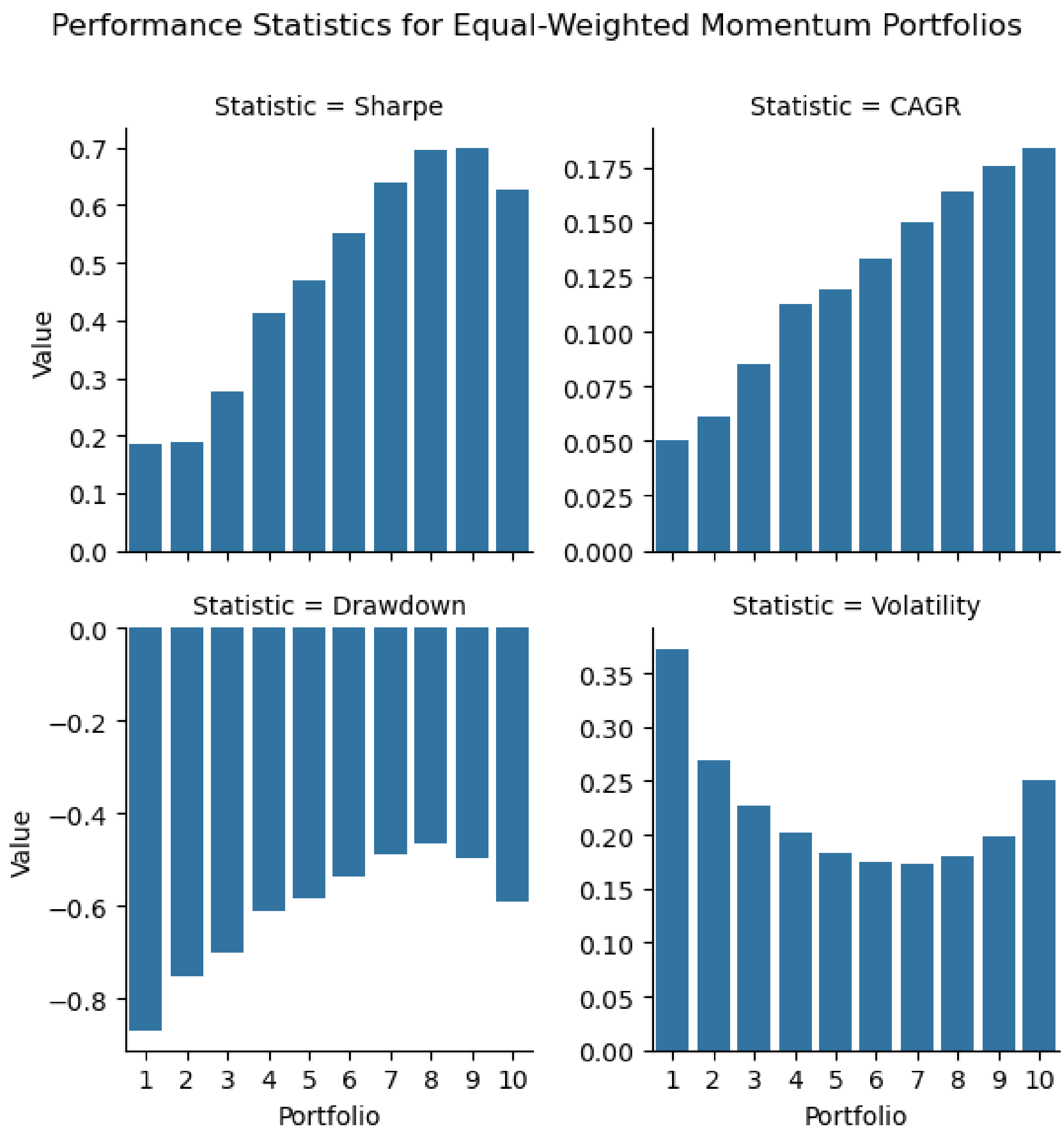
```
portfolios = (
    stocks
    .groupby(['Portfolio', 'Date'])  # Group stocks by portfolio number (1-10) and trading da
    [['RET']]                        # Isolate the returns column for portfolio-level aggrega
    .mean()                          # Average returns across stocks within each portfolio-da
    .unstack('Portfolio')            # Pivot the data to create columns for each portfolio's
    ['RET']                          # Extract the returns data, retaining portfolio numbers
)
```

```
portfolios.add(1).cumprod().plot()
plt.semilogy()
plt.ylabel('Value of $1 Investment')
plt.title('Values of $1 Investments in Equal-Weighted Momentum Portfolios')
plt.show()
```



```
portfolios_rf = portfolios.sub(ff3['RF'], axis=0)
```

```
stats = (
    portfolios_rf
    .agg(Sharpe)
    .to_frame('Sharpe')
    .join(
        portfolios
        .agg([CAGR, Drawdown, Volatility])
        .transpose()
    )
    .rename_axis(columns='Statistic')
)

df = stats.stack().to_frame('Value').reset_index()

sns.catplot(
    data=df,
    x='Portfolio',
    y='Value',
    col='Statistic',
    col_wrap=2,
    height=3,
    kind='bar',
    sharey=False
)

plt.suptitle('Performance Statistics for Equal-Weighted Momentum Portfolios', y=1.05)
plt.show()
```


Performance Statistics for Equal-Weighted Momentum Portfolios
Statistic = Sharpe
Statistic = CAGR
Statistic = Drawdown
Statistic = Volatility
Value
Portfolio
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
0.175
0.150
0.125
0.100
0.075
0.050
0.025
0.000
−0.2
−0.4
−0.6
−0.8
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00
1 2 3 4 5 6 7 8 9 10

Week 11

Project 2

Purpose

I have two goals for this project:

1. Implement and backtest a 52-week high/low breakout strategy in Python
2. Compare and contrast performance, *explaining any differences*

Assignment

Evaluate a 52-week high/low breakout strategy against a passive buy-and-hold approach. Compare and contrast the strategy on the Dow-Jones Industrial Average (DJIA) ETF, an equal-weighted portfolio of the strategy on the DJIA constituents,[1] and buying-and-holding the DJIA ETF (ticker `DIA`). *Explain any differences among the three, and recommend one.*

Here is the strategy:

- *Long Position:*
  - *Entry:* Price at the 52-week high and volume more than 1.5 times the 20-day simple moving average
  - *Exit:* Price falls 5% below the entry price or 20 days, whichever comes first
- *Short:*
  - *Entry:* Price at the 52-week low and volume more than 1.5 times the 20-day simple moving average
  - *Exit:* Price rises 5% above the entry price or 20 days, whichever comes first
- *Otherwise:*
  - Hold cash and earn the risk-free rate of return

Here are ideas to consider:

[1] Here equal-weighted portfolio means that you find the position for all the DJIA constituents. If you are long stocks 1, 2, and 3, and you are short stock 4. then your portfolio return would be: $r_p = \frac{r_1+r_2+r_3-r_4}{4}$.

1. Use the provided dataset for the current DJIA constituents, a DJIA ETF (ticker `DIA`), and the Fama-French factors
2. At a minimum, compare total returns and Sharpe Ratios
3. You might also explore maximum drawdowns, subsamples, rolling windows, trade frequencies, the volume confirmation, *plus any other compelling insights*
4. Title, label, and caption your figures and tables, referencing them in your summary

Criteria

Table 1 provides the project grading rubric. The project is worth 200 points. The peer reviews are worth 100 points, and students will receive their median score. Almost all students earn perfect peer review scores, so I will factor that into project scores. For example, a project score *without peer review scores* of 77.5% converts to a project score *with perfect peer review scores* of 85% because $\frac{0.775 \times 200 + 1.00 \times 100}{300} = 0.85 = 85\%$.

Table 1: This table provides the project grading rubric

Topic	Points
Clarity, correctness, and completeness of calculations	60
Clarity, correctness, and completeness of visualizations	60
Clarity, correctness, and completeness of discussions	60
Correctness of submission according to the deliverables section	20
Total	200

Deliverables

Upload the following as unzipped files to Canvas by 11:59 PM on 3/28:

1. One Jupyter notebook that contains your report and performs *all* your analysis
    1. Name this file `project_2.ipynb` for me to run your code
    2. Your notebook must run on my computer; I will place the data files in the same folder as your notebook
    3. You may not edit the Word documents after you create them
2. One Quarto-generated Word document *without code* for our partner to review
    1. Name this file `project_2_without_code.docx`
    2. Typing `echo: false` in the first cell of this Jupyter notebook hides code in your Word document

3. One Quarto-generated Word document *with code* for me to grade
    1. Name this file `project_2_with_code.docx`
    2. Typing `echo: true` in the first cell of this Jupyter notebook displays code in your Word document

Here is some additional guidance:

1. Provide an up to three-page executive summary at the start of your report, *which is the only writing I will read*
2. Your Word document *without code* must not exceed 15 pages in length
3. Your submission must not include your name

Data

This project requires two data files. Save these data files in the same folder as your `project_2.ipynb` notebook file.

1. `data_djia.csv` provides long-formatted data from Yahoo! Finance for the current DJIA companies, plus a DJIA ETF (ticker `DIA`), from 1998-01-20 through 2024-12-31
2. `data_ff3` provides factor data from Kenneth French's data library (name `F-F_Research_Data_Factors_da` for the same period

You can read these data files as follows.

```
import matplotlib.pyplot as plt
import matplotlib.ticker as ticker
import numpy as np
import pandas as pd
```

```
djia = (
    pd.read_csv(
        filepath_or_buffer='data_djia.csv',
        index_col=['Ticker', 'Date'],
        parse_dates=['Date']
    )
    .sort_index()
    .rename_axis(columns=['Variable'])
)
```

```python
ff3 = (
    pd.read_csv(
        filepath_or_buffer='data_ff3.csv',
        index_col=['Date'],
        parse_dates=['Date']
    )
    .sort_index()
    .rename_axis(columns=['Variable'])
)
```

Quarto

Basics

1. Use Quarto to generate your Word document from your notebook
2. Use `#` to create a title and `##` to create sections
3. Use `-` or `1.` to create lists
4. Use the first cell in this notebook to hide or display code with `echo=false` or `echo=true`, respectively
5. This first cell must be a `raw` cell instead of a `code` or `markdown` cell
6. Use `quarto render project_1.ipynb` in the same folder as your notebook to render it to a Word document
7. Use the `cd` command in the terminal to change the working directory to the directory with your notebook

Examples

This section provides a sample analysis highlighting how code and formatting work with Quarto. Figure 1 provides a line plot of the value of a $10,000 investment in DIA, the DJIA ETF. Note that `#| label:` and `#| fig-cap:` comments at the top of the figure cell create the figure reference/link and the figure caption, respectively. You can learn more about cross-referencing figures and tables here.

```python
(
    djia
    .loc['DIA']
    ['Adj Close']
    .pct_change()
    .add(1)
    .cumprod()
```

```
    .mul(10_000)
    .plot()
)
plt.ylabel('Value ($)')
plt.title(f'Value of $10,000 investment in DIA at close on {djia.loc['DIA'].index[0]: %b %d,
plt.gca().yaxis.set_major_formatter(ticker.FuncFormatter(lambda x, p: format(int(x), ',')))

plt.show()
```

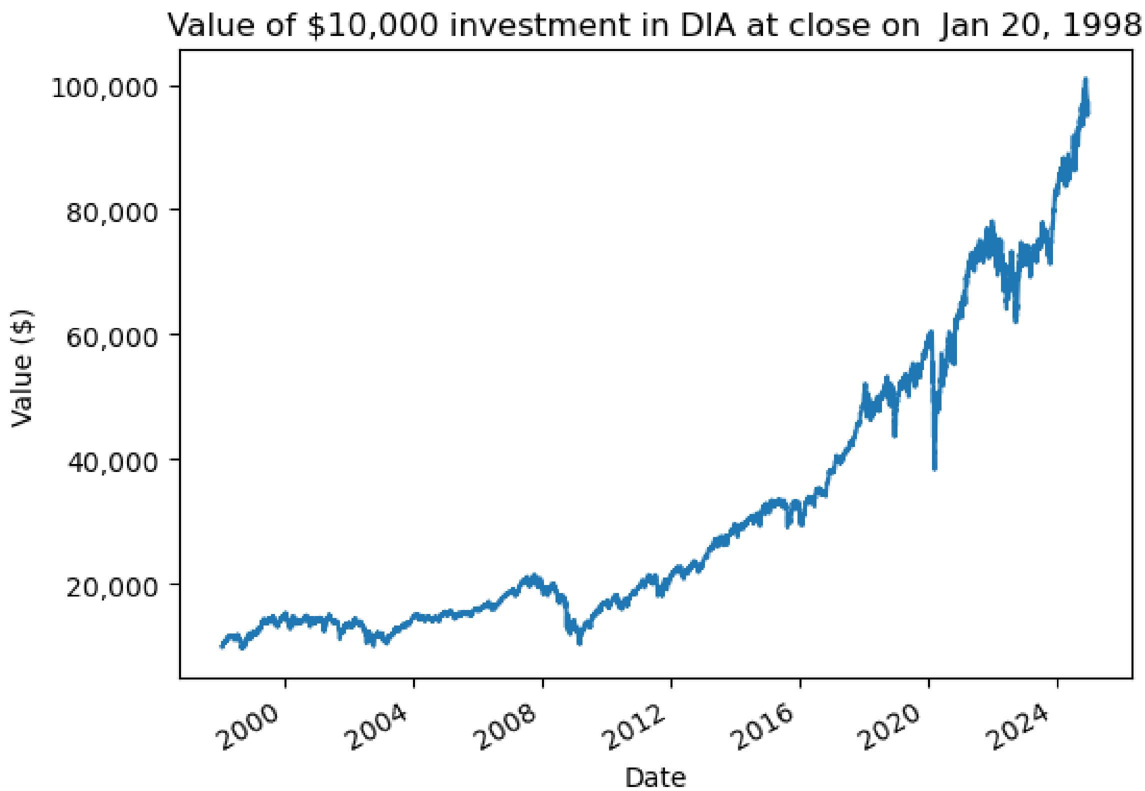


Figure 1: This line plot shows the value of a $10,000 investment in the DJIA ETF at the close of its first day of trading

Artificial Intelligence (AI)

You may use AI (e.g., ChatGPT) to *help* you prepare your analysis and discussion. However:

1. AI will not do very well on this project without significant input from your team

2. AI will not be a defense against plagiarism because AI should not *write* your code and slides; If you plagiarize an AI that plagiarizes other sources, you are responsible for plagiarizing the AI and its sources

Week 12

Herron Topic 4 - Portfolio Optimization

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import pandas_datareader as pdr
import scipy.optimize as sco # new addition for portfolio optimization
import statsmodels.api as sm
import yfinance as yf
```

```
%precision 4
pd.options.display.float_format = '{:.4f}'.format
# %config InlineBackend.figure_format = 'retina'
```

Introduction

This notebook covers portfolio optimization. In this notebook, we will:

1. Review the 1/N portfolio (or equal-weighted portfolio) from the Herron topic 1 notebook
2. Use SciPy's `minimize()` function to:
    1. Find the minimum-variance portfolio
    2. Find the minimum-variance frontier

Ivo Welch provides a clear discussion of the theory and practice of portfolio optimization in Chapter 12 of his free investments textbook.

The 1/N Portfolio

We first saw the 1/N portfolio (or equal-weighted portfolio) in the Herron topic 1 notebook. The 1/N portfolio gives each of N assets an equal portfolio weight of 1/N. While the 1/N strategy seems too simple to be useful, DeMiguel, Garlappi, and Uppal (2009) show the 1/N portfolio typically has a higher mean-variance efficiency than advanced portfolio optimizations.

We will investigate the 1/N portfolio and portfolio optimization with the most recent three years of daily returns for the Mag 7 stocks (i.e., GOOGL, AMZN, AAPL, META, MSFT, NVDA, and TSLA).

```
mag7 = (
    yf.download(
        tickers='GOOGL AMZN AAPL META MSFT NVDA TSLA',
        auto_adjust=False,
        progress=False
    )
    .iloc[:-1] # drop incomplete trading day
)
```

```
returns = (
    mag7
    ['Adj Close']
    .pct_change()
    .iloc[(-3 * 252):]
)
```

We can manually calculate 1/N portfolio returns three ways.

The first calculation is a literal interpretation of $r_P = \frac{1}{N} \sum_i^N r_i$. The pandas code for this first calculation is `returns.sum(axis=1).div(n)`, which sums across the rows, and then divides by `n`. Recall that with constant weights, we rebalance our portfolio every return period. If we have daily data, we rebalance daily. If we have monthly data, we rebalance monthly, and so on.

```
n = returns.shape[1]
r_p1 = returns.sum(axis=1).div(n)
```

```
r_p1.describe()
```

```
count    756.0000
mean       0.0010
std        0.0196
min       -0.0674
25%       -0.0099
50%        0.0007
75%        0.0127
max        0.0984
dtype: float64
```

We learned the last two calculations in the Herron topic 1. The pandas code for the second calculation is `returns.mean(axis=1)`, which calculates the mean of each row.

```
r_p2 = returns.mean(axis=1)
```

```
r_p2.describe()
```

```
count   756.0000
mean      0.0010
std       0.0196
min      -0.0674
25%      -0.0099
50%       0.0007
75%       0.0127
max       0.0984
dtype: float64
```

The pandas code for the third calculations is `returns.dot(weights)`, where `weights` is a NumPy array or a pandas series of portfolio weights. This calculation allows different weights for each asset. We can make 1/N portfolio weights with `np.ones(n) / n`.

```
weights = np.ones(n) / n
```

```
weights
```

```
array([0.1429, 0.1429, 0.1429, 0.1429, 0.1429, 0.1429, 0.1429])
```

```
r_p3 = returns.dot(weights)
```

```
r_p3.describe()
```

```
count   756.0000
mean      0.0010
std       0.0196
min      -0.0674
25%      -0.0099
50%       0.0007
75%       0.0127
max       0.0984
dtype: float64
```

We can use `np.allclose()` to show that `r_p1`, `r_p2`, and `r_p3` are similar.

```
np.allclose(r_p1, r_p2)
```

```
True
```

```
np.allclose(r_p2, r_p3)
```

```
True
```

```
np.allclose(r_p3, r_p1)
```

```
True
```

Here is a brief detour to show how the `.dot()` method works.

```
silly_n = 3
silly_w = np.ones(silly_n) / silly_n
silly_r = pd.DataFrame(np.arange(2*silly_n).reshape(2, silly_n))
```

```
print(
    f'silly_n:\n{silly_n}',
    f'silly_w:\n{silly_w}',
    f'silly_r:\n{silly_r}',
    sep='\n\n'
)
```

```
silly_n:
3

silly_w:
[0.3333 0.3333 0.3333]

silly_r:
   0  1  2
0  0  1  2
1  3  4  5
```

```
silly_r.dot(silly_w)
```

```
0    1.0000
1    4.0000
dtype: float64
```

Under the hood, the `.dot()` method efficiently calculates the following:

```
for i, row in silly_r.iterrows():
    print(
        f'Row {i}: ',
        ' + '.join([f'{w:0.2f} * {y}' for w, y in zip(silly_w, row)]),
        ' = ',
        f'{silly_r.dot(silly_w).iloc[i]:0.2f}'
    )
```

```
Row 0:  0.33 * 0 + 0.33 * 1 + 0.33 * 2  =  1.00
Row 1:  0.33 * 3 + 0.33 * 4 + 0.33 * 5  =  4.00
```

SciPy's `minimize()` Function

SciPy's `minimize()` function, from the `optimize` module, finds the input array `x` that minimizes the output of a given function `fun`. The `minimize()` function uses optimization techniques that are beyond the scope of this course. We can consider these optimization techniques as sophisticated trial-and-error (or guess-and-check).

Here are the common arguments for the the `minimize()` function:

1. `fun=` is the name of the function who's output we want to minimize
2. `x0=` is our first guess for the input to the function `fun`
3. `args=` is a tuple of additional arguments to the function `fun`
4. `bounds=` is a tuple of tuples with lower and upper bounds for each element in `x`
5. `contraints=` is a tuple of dictionaries that limit `x` by constraining the output of a given function to zero or non-negative values

Here is a simple example that minimizes the function `quadratic()`, which has arguments x and a and returns $(x-a)^2$.

```python
def quadratic(x, a=5):
    return (x - a) ** 2
```

```python
quadratic(x=0, a=5)
```

```
25
```

```python
quadratic(x=5, a=5)
```

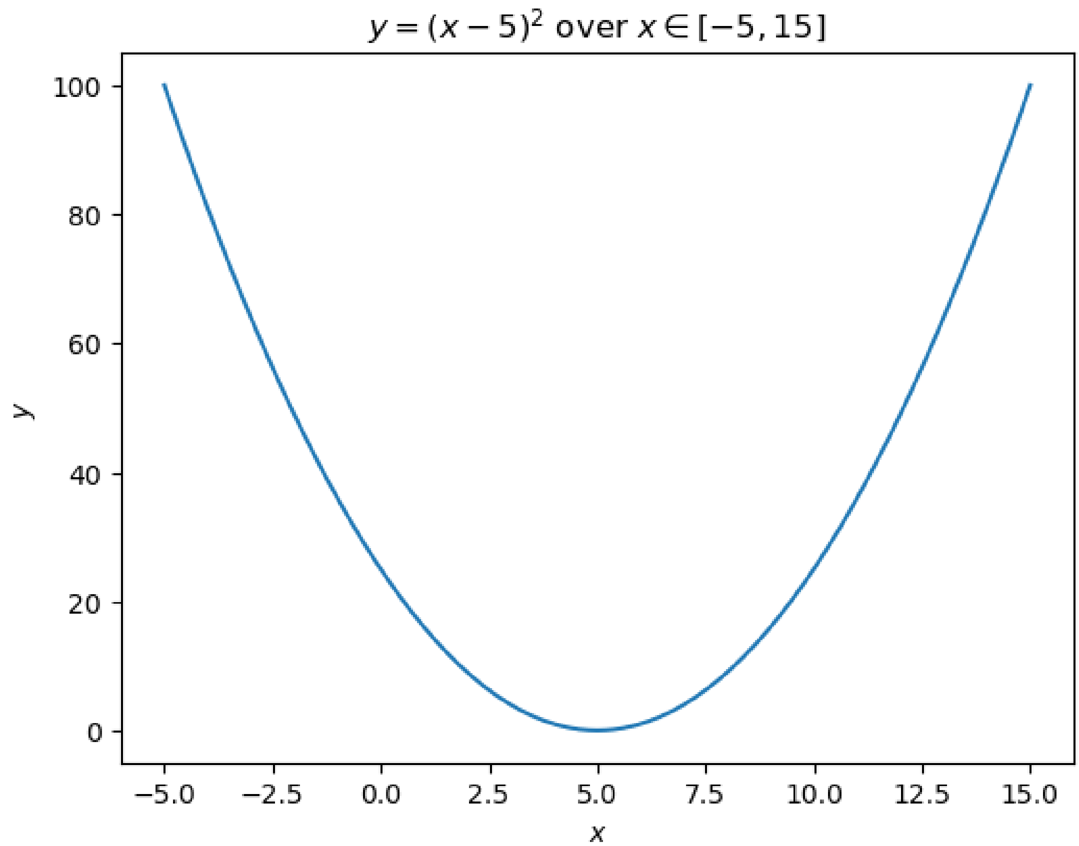
```
0
```

```python
quadratic(x=10, a=5)
```

```
25
```

It might be helpful to plot $y = (x - a)^2$.

```python
x = np.linspace(-5, 15, 101)
y = quadratic(x=x)
plt.plot(x, y)
plt.xlabel(r'$x$')
plt.ylabel(r'$y$')
plt.title(r'$y = (x - 5)^2$ over $x \in [-5, 15]$')
plt.show()
```



The minimum output of `quadratic()` occurs at $x = 5$ if we do not use bounds or constraints. For this function, we can even start far away from $x = 5$.

```
sco.minimize(
    fun=quadratic,
    x0=np.array([2001])
)
```

```
  message: Optimization terminated successfully.
  success: True
   status: 0
      fun: 2.0392713450495178e-16
        x: [ 5.000e+00]
      nit: 4
      jac: [-1.366e-08]
 hess_inv: [[ 5.000e-01]]
     nfev: 18
```

```
    njev: 9
```

The minimum output of `quadratic()` occurs at $x = 6$ if we bound $6 \leq x \leq 10$ with `bounds=((6, 10),)`. `bounds=` requires a tuple of tuples, so we need the trailing comma if we only have one set of bounds.

```
sco.minimize(
    fun=quadratic,
    x0=np.array([2001]),
    bounds=((6, 10),)
)
```

```
  message: CONVERGENCE: NORM OF PROJECTED GRADIENT <= PGTOL
  success: True
   status: 0
      fun: 1.0
        x: [ 6.000e+00]
      nit: 1
      jac: [ 2.000e+00]
     nfev: 4
     njev: 2
 hess_inv: <1x1 LbfgsInvHessProduct with dtype=float64>
```

The minimum output of `quadratic()` also occurs at $x = 6$ if we constrain $x - 6 \geq 0$ with `constraints=({'type': 'ineq', 'fun': lambda x: x - 6})`. For inequality constraints, the given function must evaluate to a non-negative value. We use bounds to limit the search space directly, and we use constraints to limit the search space based on a given function.

```
sco.minimize(
    fun=quadratic,
    x0=np.array([2001]),
    constraints=({'type': 'ineq', 'fun': lambda x: x - 6})
)
```

```
 message: Optimization terminated successfully
 success: True
  status: 0
     fun: 1.0000000000000018
       x: [ 6.000e+00]
     nit: 3
     jac: [ 2.000e+00]
```

```
    nfev: 6
    njev: 3
```

We use the `args=` argument to pass additional arguments to the function `fun`. For example, we change the `a=` argument in `quadratic()` from the default of `a=5` to `a=20` with `args=(20,)`. Like `bounds=`, `args=` expects a tuple, so we need a trailing comma if we have only one argument.

```
sco.minimize(
    fun=quadratic,
    args=(20,),
    x0=np.array([2001]),
)
```

```
  message: Optimization terminated successfully.
  success: True
   status: 0
      fun: 7.090392030754976e-17
        x: [ 2.000e+01]
      nit: 4
      jac: [-1.940e-09]
 hess_inv: [[ 5.000e-01]]
     nfev: 18
     njev: 9
```

The Minimum-Variance Portfolio

Now we apply `minimize()` to our Mag-7 returns to find the the minimum-variance portfolio. Recall, the `minimize()` function varies an input array `x`, starting from argument `x0=`, to minimize the output of a given function `fun=` subject to the bounds and constraints in `bounds=` and `constraints=`, respectively.

We can define a function `calc_sigmap()` to calculate portfolio volatility. The first argument to `calc_sigmap()` must be the input array `x` that `minimize()` searches over. We call this first argument `w` because it has portfolio weights. Recall that portfolio variance is $\sigma_p^2 = w'\Sigma w$, so portfolio volatility is $\sigma_p = \sqrt{w'\Sigma w}$. Here Σ (and `Sigma`) is the returns covariance matrix, and w (and `w`) is the portfolio weights array.

```
def calc_sigmap(w, Sigma, ppy=252):
    return np.sqrt(ppy * w.T @ Sigma @ w)
```

```
Sigma = returns.cov()
```

```
Sigma
```

Ticker Ticker	AAPL	AMZN	GOOGL	META	MSFT	NVDA	TSLA
AAPL	0.0003	0.0002	0.0002	0.0002	0.0002	0.0003	0.0003
AMZN	0.0002	0.0005	0.0003	0.0004	0.0003	0.0004	0.0004
GOOGL	0.0002	0.0003	0.0004	0.0003	0.0002	0.0004	0.0003
META	0.0002	0.0004	0.0003	0.0008	0.0003	0.0005	0.0004
MSFT	0.0002	0.0003	0.0002	0.0003	0.0003	0.0004	0.0003
NVDA	0.0003	0.0004	0.0004	0.0005	0.0004	0.0012	0.0006
TSLA	0.0003	0.0004	0.0003	0.0004	0.0003	0.0006	0.0015

We can also define a function `calc_mup()` that calculates a mean portfolio return. Recall that mean portfolio return is $\mu_p = w'\mu$. Here μ (and `mu`) is the mean returns array, and w (and `w`) is the portfolio weights array.

```
def calc_mup(w, mu, ppy=252):
    return ppy * w.T @ mu
```

```
mu = returns.mean()
```

```
mu
```

```
Ticker
AAPL    0.0005
AMZN    0.0005
GOOGL   0.0004
META    0.0018
MSFT    0.0005
NVDA    0.0025
TSLA    0.0005
dtype: float64
```

Finally, we can use `minimize()` to find the portfolio weights that minimize portfolio volatility. Below, the equality constraint is met when `x.sum() - 1` is zero, which is the Python equivalent of $\sum_i w_i = 1$.

```
[(0,1) for _ in Sigma]
```

```
[(0, 1), (0, 1), (0, 1), (0, 1), (0, 1), (0, 1), (0, 1)]
```

```
res_mv = sco.minimize(
    fun=calc_sigmap,
    x0=np.ones(Sigma.shape[0]) / Sigma.shape[0],
    args=(Sigma, 252),
    bounds=[(0,1) for _ in Sigma],
    constraints=(
        {'type': 'eq', 'fun': lambda x: x.sum() - 1},
    )
)
```

```
print(res_mv)
```

```
 message: Optimization terminated successfully
 success: True
  status: 0
     fun: 0.2434752847371044
       x: [ 4.614e-01  0.000e+00  9.627e-02  2.819e-18  4.423e-01
            4.207e-17  9.975e-18]
     nit: 10
     jac: [ 2.435e-01  2.573e-01  2.435e-01  2.770e-01  2.435e-01
            3.424e-01  3.046e-01]
    nfev: 80
    njev: 10
```

What are the attributes of the minimum-variance portfolio? We can write a helper function `print_port_res()` to simplify our lives!

```
def print_port_res(w, mu, Sigma, title, ppy=252):
    width = max(len(title), 32)
    mup = ppy * w.T @ mu
    sigmap = np.sqrt(ppy * w.T @ Sigma @ w)

    return print(
        title,
        '=' * width,
        '',
```

```
        'Performance',
        '-' * width,
        'Annualized Mean Return:'.ljust(width - 6) + f'{mup:0.4f}',
        'Annualized Volatility:'.ljust(width - 6) + f'{sigmap:0.4f}',
        '',
        'Weights',
        '-' * width,
        '\n'.join([
            f'{_r}:'.ljust(width - 6) + f'{_w:0.4f}'
            for _r, _w in zip(Sigma.index, w)
        ]),
        '=' * width,
        sep='\n',
    )
```

```
print_port_res(
    w=res_mv['x'],
    mu=returns.mean(),
    Sigma=returns.cov(),
    title='minimum-variance portfolio'
)
```

```
minimum-variance portfolio
================================

Performance
--------------------------------
Annualized Mean Return:   0.1292
Annualized Volatility:    0.2435

Weights
--------------------------------
AAPL:                     0.4614
AMZN:                     0.0000
GOOGL:                    0.0963
META:                     0.0000
MSFT:                     0.4423
NVDA:                     0.0000
TSLA:                     0.0000
================================
```

The Minimum-Variance Frontier

We can use `minimize()` to find the minimum-variance portfolio for a *target return*, then use a `for` loop to repeat this calculation for a wide range of target returns. This calculation will map out the minimum-variance frontier. Here are the steps:

1. Create a NumPy array `returns_target` of target returns
2. Create an empty list `res_ef` to hold `minimize()` results
3. Loop over `returns_target`, passing each target return as a constraint to `minimize()`
4. Append each `minimize()` result to `res_ef`

```
returns_target = 252 * np.linspace(
    start=mu.min(),
    stop=mu.max(),
    num=50
)
```

```
returns_target[:5]
```

```
array([0.1035, 0.1143, 0.1251, 0.1359, 0.1467])
```

Below, the first equality constraint is met when `x.sum() - 1` is zero, which is the Python equivalent of $\sum_i w_i = 1$. The second equality constraint is met when `calc_mup(w=x, mu=mu, ppy=252) - r` is zero, which is the Python equivbalent of $w'\mu =$ target return.

```
res_ef = []

for r in returns_target:
    _ = sco.minimize(
        fun=calc_sigmap,
        x0=np.ones(Sigma.shape[0]) / Sigma.shape[0],
        args=(Sigma, 252),
        bounds=[(0, 1) for c in Sigma.index],
        constraints=(
            {'type': 'eq', 'fun': lambda x: x.sum() - 1},
            {'type': 'eq', 'fun': lambda x: calc_mup(w=x, mu=mu, ppy=252) - r}
        )
    )
    res_ef.append(_)
```

The list `res_ef` contains the results of all 50 minimum-variance portfolios. For example, `res_ef[0]` is the minimum-variance portfolio for the lowest target return.

```python
res_ef[0]
```

```
 message: Optimization terminated successfully
 success: True
  status: 0
     fun: 0.3237995797768805
       x: [ 0.000e+00  1.110e-16  1.000e+00  2.776e-16  5.171e-08
            0.000e+00  0.000e+00]
     nit: 2
     jac: [ 1.592e-01  2.418e-01  3.238e-01  2.692e-01  1.773e-01
            2.893e-01  2.533e-01]
    nfev: 16
    njev: 2
```

```python
print_port_res(
    w=res_ef[0]['x'],
    mu=mu,
    Sigma=Sigma,
    title='minimum-variance portfolio for lowest tgt. ret.'
)
```

```
minimum-variance portfolio for lowest tgt. ret.
===============================================

Performance
-----------------------------------------------
Annualized Mean Return:                  0.1035
Annualized Volatility:                   0.3238

Weights
-----------------------------------------------
AAPL:                                    0.0000
AMZN:                                    0.0000
GOOGL:                                   1.0000
META:                                    0.0000
MSFT:                                    0.0000
NVDA:                                    0.0000
TSLA:                                    0.0000
===============================================
```

We should check that all minimizations suceeded. If a portfolio volatility minimization fails, we should check our function, bounds, and constraints. Portfolio optimization is typically a simple problem for `minimize()`, so we typically have an error if `minimize()` does not succeed.

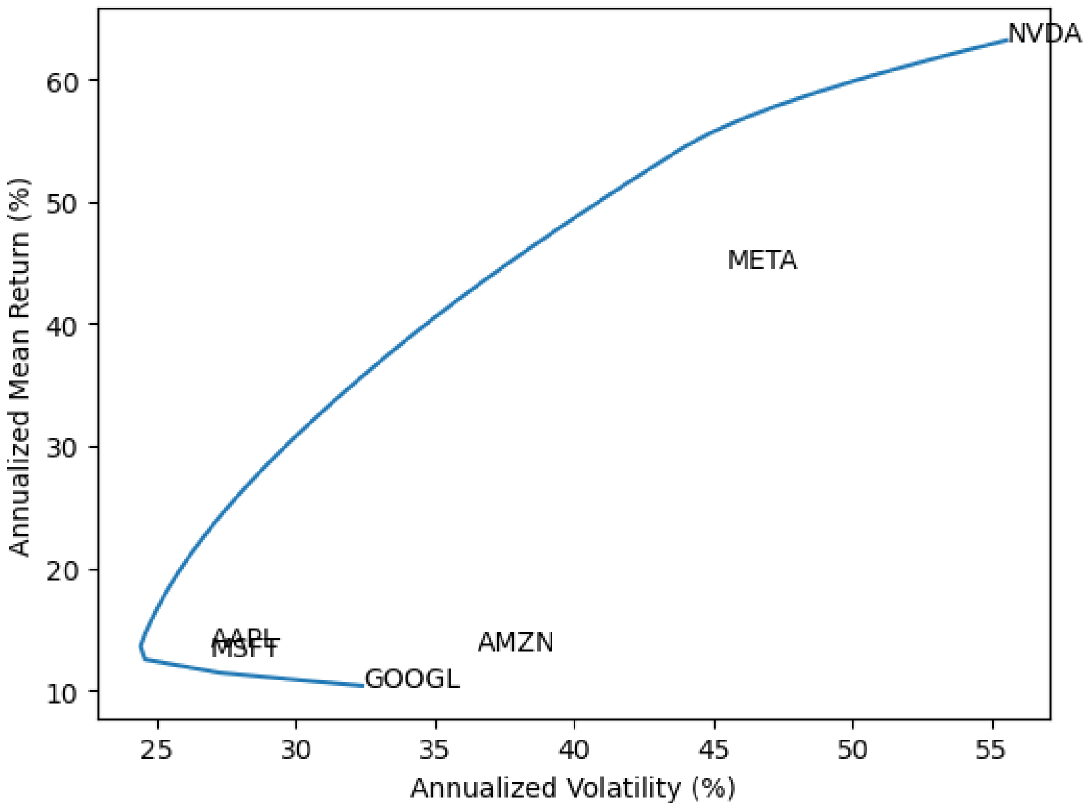
```
for r in res_ef:
    assert r['success']
```

We can combine the target returns and volatilities into a data frame `mv_frontier`.

```
mv_frontier = pd.DataFrame({
    'returns_target': returns_target,
    'volatility': np.array([r['fun'] if r['success'] else np.nan for r in res_ef])
})
```

Finally, we can plot the minimum-variance frontier.

```
(
    mv_frontier
    .mul(100)
    .plot(x='volatility', y='returns_target', legend=False)
)
plt.ylabel('Annualized Mean Return (%)')
plt.xlabel('Annualized Volatility (%)')

for t in Sigma.index:
    x = 100 * np.sqrt(252 * Sigma.loc[t, t])
    y = 100 * 252 * mu.loc[t]
    plt.annotate(text=t, xy=(x, y))

plt.show()
```


NVDA
META
AAPL
MSFT
AMZN
GOOGL
Annualized Mean Return (%)
Annualized Volatility (%)
10
20
30
40
50
60
25
30
35
40
45
50
55

Herron Topic 4 - Practice

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import pandas_datareader as pdr
import scipy.optimize as sco # new addition for portfolio optimization
import yfinance as yf
```

```
%precision 4
pd.options.display.float_format = '{:.4f}'.format
# %config InlineBackend.figure_format = 'retina'
```

Announcements

Five-Minute Recap

Practice

Find the maximum Sharpe Ratio portfolio of Mag 7 stocks over the last three years

Note that `sco.minimize()` finds *minimums*, so you need to minimize the *negative* Sharpe Ratio.

Find the maximum Sharpe Ratio portfolio of Mag 7 stocks over the last three years, but allow short weights up to 10% on each stock

Find the maximum Sharpe Ratio portfolio of Mag 7 stocks over the last three years, but allow total short weights of up to 30%

Find the maximum Sharpe Ratio portfolio of Mag 7 stocks over the last three years, but do not allow any weight to exceed 30% in magnitude

Find the minimum 95% Value at Risk (Var) portfolio of Mag 7 stocks over the last three years

More on VaR here.

Find the minimum draw down portfolio of Mag 7 stocks over the last three years

Find the minimum draw down portfolio for the sample with complete data for the current Dow-Jones Industrial Average (DJIA) stocks

You can find the DJIA tickers on Wikipedia.

Plot the minimum-variance frontier for the sample with complete data for the current the DJIA stocks

Find the maximum Sharpe Ratio portfolio for the sample with complete data for the current the DJIA stocks

Herron Topic 4 - Practice - Sec 02

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import pandas_datareader as pdr
import scipy.optimize as sco # new addition for portfolio optimization
import statsmodels.api as sm
import yfinance as yf
```

```
%precision 4
pd.options.display.float_format = '{:.4f}'.format
# %config InlineBackend.figure_format = 'retina'
```

Announcements

1. Please complete TRACE! I use TRACE to improve my treaching and courses, and I value your feedback. More here: https://northeastern.instructure.com/courses/207607/discussion_topics/2753925
2. Please plan for the in-class programming and MSFQ assessments on Tuesday, 4/15

Five-Minute Recap

Please see the lecture notebook for an in-depth explanation of how we will use `sco.minimize()` for portfolio optimization. Here are the key arguments to `sco.minimize()`:

1. `fun`: Name of function whose output we want to *minimize*
2. `x0`: First guess at inputs that *minimize* the output of the function in `fun`
3. `args`: A tuple of additional arguments to the function in `fun`
4. `bounds`: A list or tuple of tuples; For example, `((0, 1), (0, 1))` bounds inputs to fall between 0 and 1
5. `constraints`: A tuple of dictionaries with functions to constrain our inputs; For example, `{'type': 'eq', 'fun': lambda w: w.sum() - 1}` constraints te sum or our inputs to 1

Practice

Find the maximum Sharpe Ratio portfolio of Mag 7 stocks over the last three years

Note that `sco.minimize()` finds *minimums*, so you need to minimize the *negative* Sharpe Ratio.

```
mag7 = (
    yf.download(
        tickers='GOOGL AMZN AAPL META MSFT NVDA TSLA',
        auto_adjust=False,
        progress=False
    )
    .iloc[:-1] # drop incomplete trading day
)
```

```
returns = mag7['Adj Close'].pct_change().iloc[-756:]
```

We need the risk-free rate of return to calculate Sharpe ratios. French provides the risk-free rate of return as `RF` in most of his data sets.

```
ff3 = (
    pdr.DataReader(
        name='F-F_Research_Data_Factors_daily',
        data_source='famafrench',
        start='1900'
    )
    [0]
    .div(100)
)
```

```
C:\Users\r.herron\AppData\Local\Temp\ipykernel_23028\582763811.py:2: FutureWarning: The argu
  pdr.DataReader(
```

The Sharpe ratio is the ratio of the mean portfolio *excess* return to the volatility of portfolio *excess* returns.

$$S_p = \frac{\overline{r_p - r_f}}{\sigma(r_p - r_f)}$$

We can simplify this calculation if we calculate a data frame of *excess* returns. Then, we can use this data frame of excess returns to define the covariance matrix and mean returns.

```
returns_excess = returns.sub(ff3['RF'], axis=0)
Sigma_excess = returns_excess.cov()
mu_excess = returns_excess.mean()
```

The maximize the Sharpe ratio, we need to *minimize* the *negative* Sharpe ratio because there is no `maximize()` function. To simplify our code, we can define `Sharpe_neg()` and `Sharpe()` functions. We will use the former with `minimize()` and the latter everwhere else. The `Sharpe()` function uses matrix math, where `@` is the Numpy and pandas symbol for matrix multiplication.

1. `w.T @ mu_excess` is code for $w'\mu_{\text{excess}}$
2. `w.T @ Sigma_excess @ w` is code for $w'\Sigma_{\text{excess}}\mu$

```
def Sharpe(w, Sigma_excess, mu_excess, ppy=252):
    return (ppy * w.T @ mu_excess) / np.sqrt(ppy * w.T @ Sigma_excess @ w)
```

```
def Sharpe_neg(w, Sigma_excess, mu_excess, ppy=252):
    return -1 * Sharpe(w=w, Sigma_excess=Sigma_excess, mu_excess=mu_excess, ppy=252)
```

```
def equal_weights(n):
    return np.ones(n) / n
```

```
equal_weights(Sigma_excess.shape[1])
```

```
array([0.1429, 0.1429, 0.1429, 0.1429, 0.1429, 0.1429, 0.1429])
```

```
[(0, 1) for _ in range(Sigma_excess.shape[1])]
```

```
[(0, 1), (0, 1), (0, 1), (0, 1), (0, 1), (0, 1), (0, 1)]
```

We can store our `minimize()` results in `res_X` and increment `X` for each of the following practices.

```
res_1 = sco.minimize(
    fun=Sharpe_neg,
    x0=equal_weights(Sigma_excess.shape[1]),
    args=(Sigma_excess, mu_excess),
    bounds=[(0, 1) for _ in range(7)],
    constraints=(
```

```
        {'type': 'eq', 'fun': lambda x: x.sum() - 1}
    )
)
```

We want to make sure that `minimize()` finds a solution (i.e., `res_1['success']` is `True`). The *negative* Sharpe ratio is the value for the `fun` key, and the portfolio weights are the value for the `x` key.

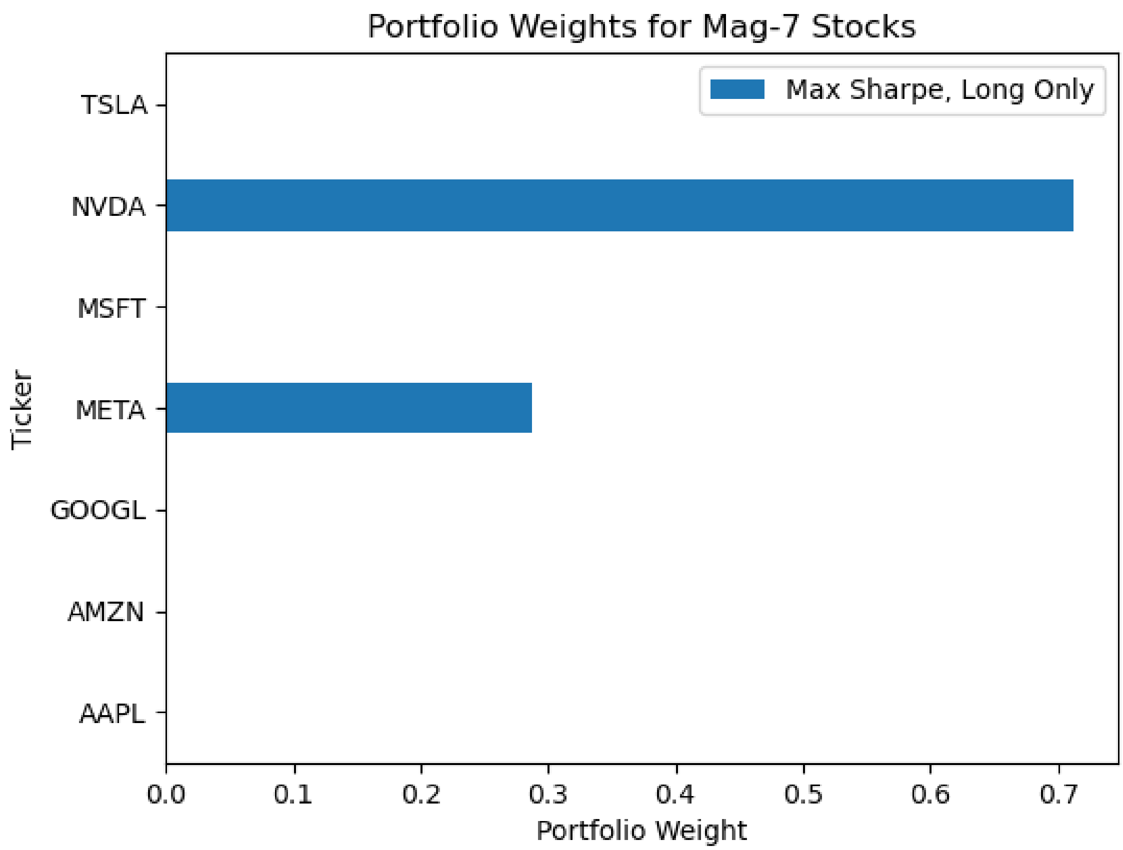
```
res_1
```

```
 message: Optimization terminated successfully
 success: True
  status: 0
     fun: -1.2695739772405898
       x: [ 3.451e-17  8.810e-17  0.000e+00  2.880e-01  0.000e+00
            7.120e-01  0.000e+00]
     nit: 5
     jac: [ 1.624e-01  3.758e-01  2.860e-01 -4.549e-04  2.635e-01
            1.840e-04  3.562e-01]
    nfev: 40
    njev: 5
```

We can save these results to a data frame for easy updating and plotting.

```
res_df = pd.DataFrame(
    data={'Max Sharpe, Long Only': res_1['x']},
    index=Sigma_excess.columns
)
```

```
res_df.plot(kind='barh')
plt.title('Portfolio Weights for Mag-7 Stocks')
plt.xlabel('Portfolio Weight')
plt.show()
```



What if we want the actual, positive Sharpe ratio? We can use the `Sharpe()` function. If we want to update the results in `res_1`, we could write another helper function. However, I generally give in and learn to live with the output of commonly used functions.

```
res_1
```

```
 message: Optimization terminated successfully
 success: True
  status: 0
     fun: -1.2695739772405898
       x: [ 3.451e-17  8.810e-17  0.000e+00  2.880e-01  0.000e+00
            7.120e-01  0.000e+00]
     nit: 5
     jac: [ 1.624e-01  3.758e-01  2.860e-01 -4.549e-04  2.635e-01
```

```
            1.840e-04  3.562e-01]
    nfev: 40
    njev: 5
```

```
Sharpe(w=res_1['x'], Sigma_excess=Sigma_excess, mu_excess=mu_excess)
```

```
1.2696
```

Find the maximum Sharpe Ratio portfolio of Mag 7 stocks over the last three years, but allow short weights up to 10% on each stock

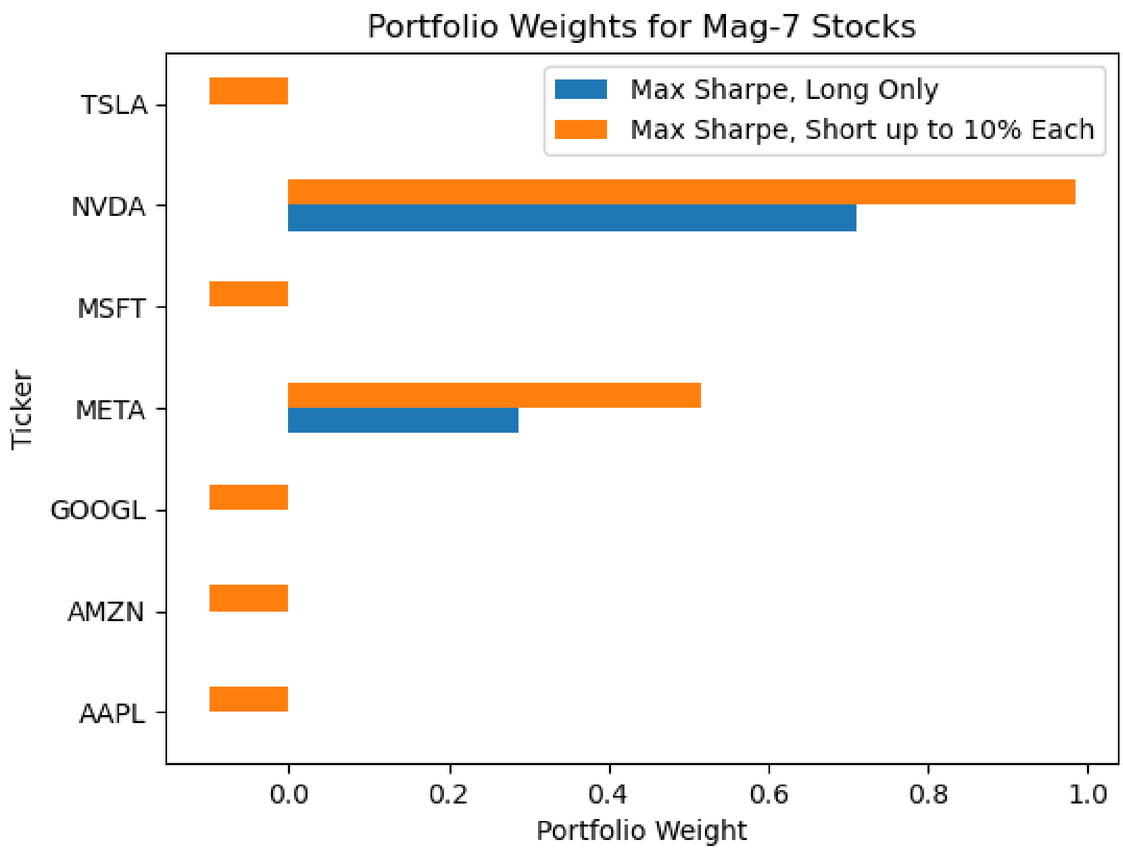
We can short 6 of 7 stocks up to -0.1 for a total of -0.6. Therefore, the maximum possible long weight is 1.6, and `bounds` becomes:

```
    bounds=[(-0.1, 1.6) for _ in range(7)],
```

```
res_2 = sco.minimize(
    fun=Sharpe_neg,
    x0=equal_weights(Sigma_excess.shape[1]),
    args=(Sigma_excess, mu_excess),
    bounds=[(-0.1, 1.6) for _ in range(7)],
    constraints=(
        {'type': 'eq', 'fun': lambda x: x.sum() - 1}
    )
)
```

```
res_df['Max Sharpe, Short up to 10% Each'] = res_2['x']
```

```
res_df.plot(kind='barh')
plt.title('Portfolio Weights for Mag-7 Stocks')
plt.xlabel('Portfolio Weight')
plt.show()
```



Find the maximum Sharpe Ratio portfolio of Mag 7 stocks over the last three years, but allow total short weights of up to 30%

We need to use an unequality constraints to make sure the *sum* of the negative portfolios weights is greater than -0.3. We express inequality constraints with functions with non-negative outputs, so we use $\sum_i w_i[w_i < 0] + 0.3 \geq 0$.

We see that once we allow shorts greater than 30%, we spend almost all of this short budget on AMZN.

```python
toy = np.arange(-2, 3)
```

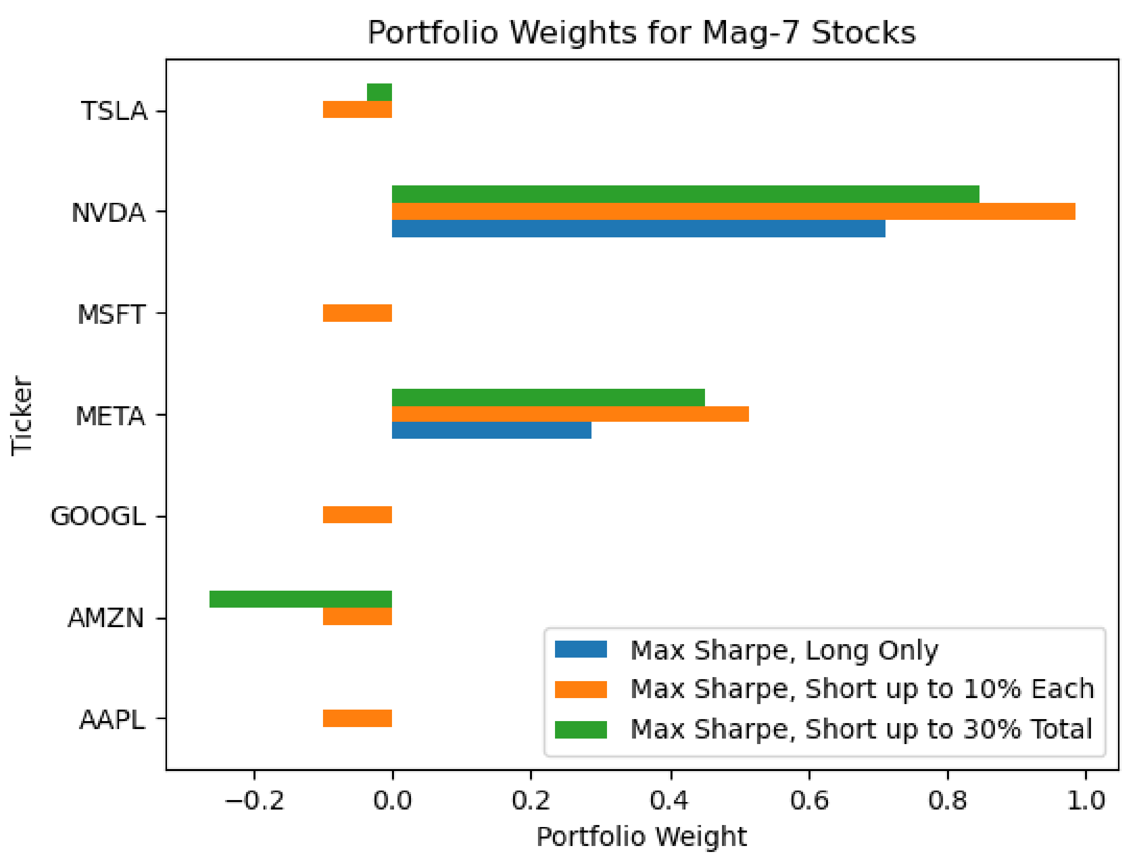
```python
toy[toy < 0].sum()
```

```
np.int64(-3)
```

```
res_3 = sco.minimize(
    fun=Sharpe_neg,
    x0=equal_weights(Sigma_excess.shape[1]),
    args=(Sigma_excess, mu_excess),
    bounds=[(-0.3, 1.3) for _ in range(7)],
    constraints=(
        {'type': 'eq', 'fun': lambda x: x.sum() - 1},
        {'type': 'ineq', 'fun': lambda x: x[x < 0].sum() + 0.3}
    )
)
```

```
res_df['Max Sharpe, Short up to 30% Total'] = res_3['x']
```

```
res_df.plot(kind='barh')
plt.title('Portfolio Weights for Mag-7 Stocks')
plt.xlabel('Portfolio Weight')
plt.show()
```



Find the maximum Sharpe Ratio portfolio of Mag 7 stocks over the last three years, but do not allow any weight to exceed 30% in magnitude

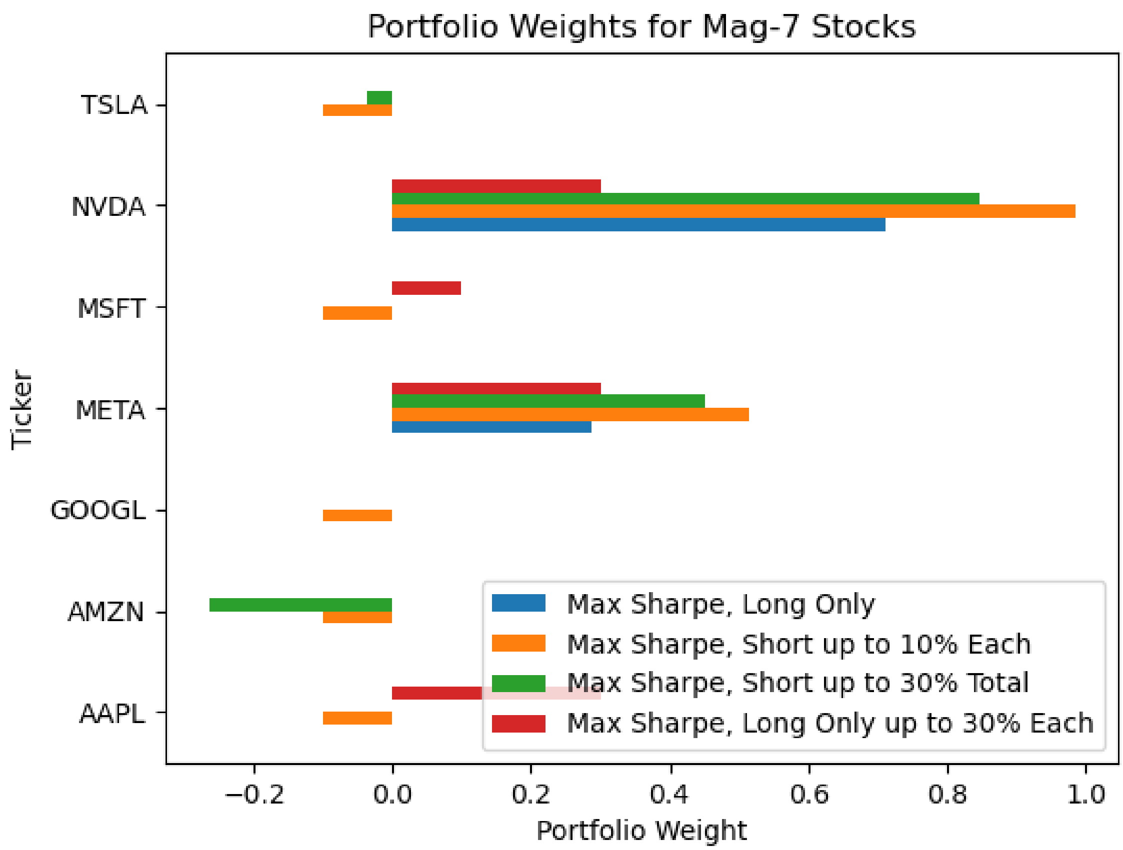
We can bound all portfolios weights on $[0, 0.3]$ with:

```
bounds=[(0, 0.3) for _ in range(7)]
```

```
res_4 = sco.minimize(
    fun=Sharpe_neg,
    x0=equal_weights(Sigma_excess.shape[1]),
    args=(Sigma_excess, mu_excess),
    bounds=[(0, 0.3) for _ in range(7)],
    constraints=(
        {'type': 'eq', 'fun': lambda x: x.sum() - 1}
    )
)
```

```
res_df['Max Sharpe, Long Only up to 30% Each'] = res_4['x']
```

```
res_df.plot(kind='barh')
plt.title('Portfolio Weights for Mag-7 Stocks')
plt.xlabel('Portfolio Weight')
plt.show()
```



Find the minimum 95% Value at Risk (Var) portfolio of Mag 7 stocks over the last three years

More on VaR here.

```
def VaR(weights, returns, percent):
    return returns.dot(weights).quantile(1 - percent)
```

```
VaR(
    weights=equal_weights(returns.shape[1]),
    returns=returns,
    percent=0.95
)
```

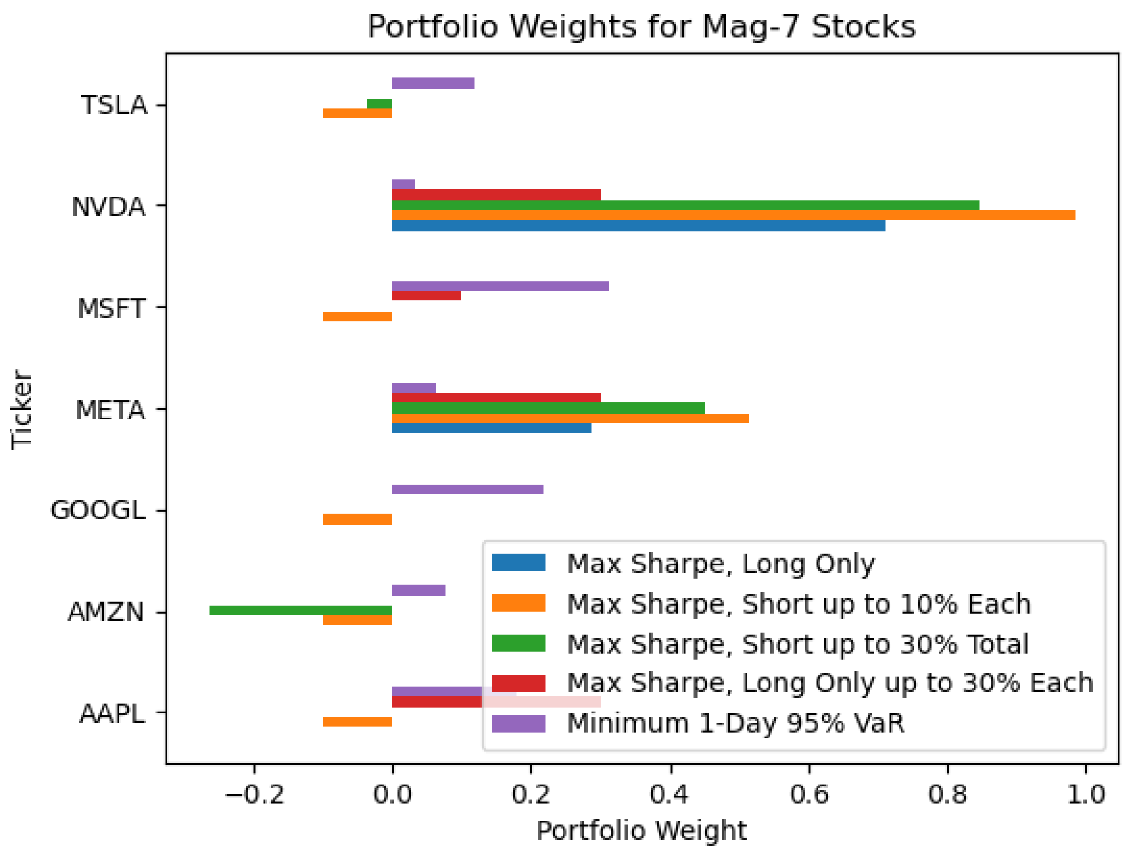
```
-0.0336
```

```
def VaR_neg(weights, returns, percent):
    return -1 * VaR(weights=weights, returns=returns, percent=percent)
```

```
res_var = sco.minimize(
    fun=VaR_neg,
    x0=equal_weights(returns.shape[1]), # np.ones(7) / 7
    args=(returns, 0.95),
    bounds=[(0, 1) for _ in range(returns.shape[1])],
    constraints=(
        {'type': 'eq', 'fun': lambda w: w.sum() - 1}
    )
)
```

```
res_df['Minimum 1-Day 95% VaR'] = res_var['x']
```

```
res_df.plot(kind='barh')
plt.title('Portfolio Weights for Mag-7 Stocks')
plt.xlabel('Portfolio Weight')
plt.show()
```



Find the minimum draw down portfolio of Mag 7 stocks over the last three years

```
def Max_Drawdown(w, returns):
    price = returns.dot(w).add(1).cumprod()
    return (price / price.cummax() - 1).min()
```

```
Max_Drawdown(equal_weights(7), returns)
```

```
-0.4524
```

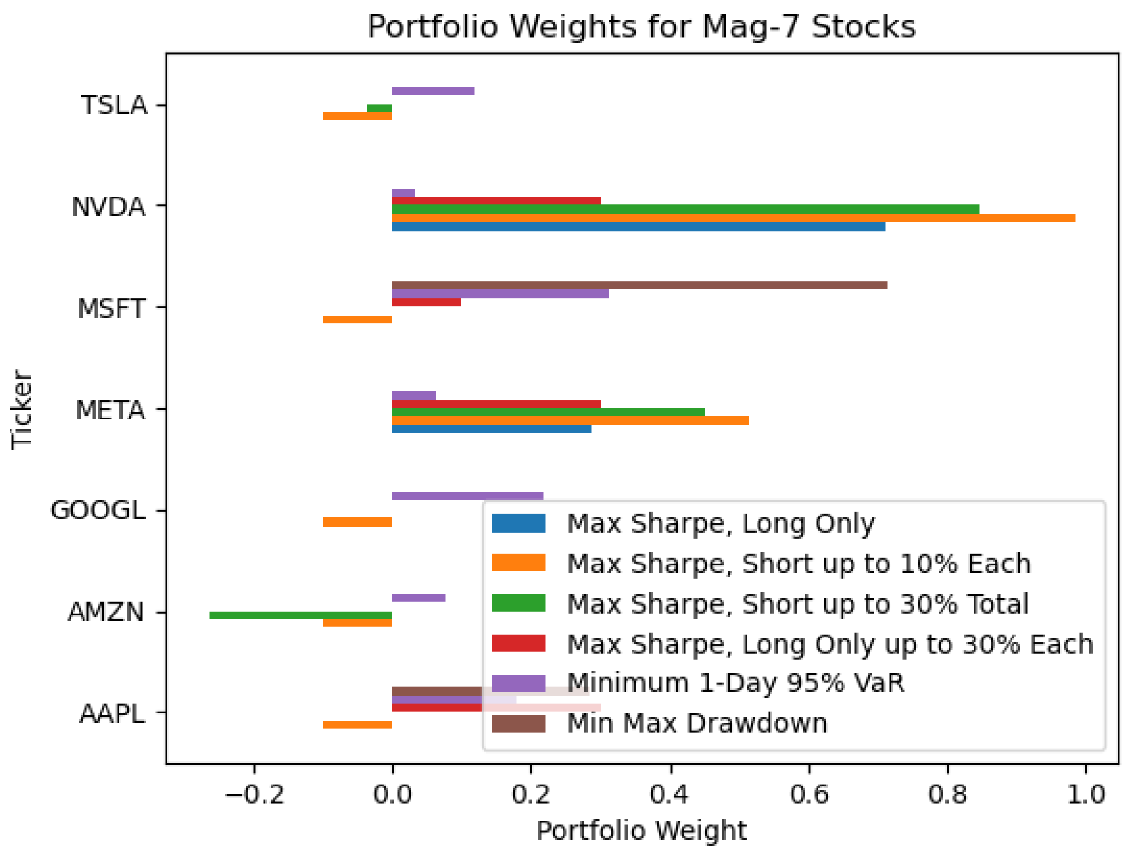
The maximum drawdown is a negative number, but we want the *smallest* negative number, so we minimize negative one times the maximum drawdown.

```python
def Max_Drawdown_neg(w, returns):
    return -1 * Max_Drawdown(w=w, returns=returns)
```

```python
res_mdd = sco.minimize(
    fun=Max_Drawdown_neg,
    x0=equal_weights(returns.shape[1]), # np.ones(7) / 7
    args=(returns,),
    bounds=[(0, 1) for _ in range(returns.shape[1])],
    constraints=(
        {'type': 'eq', 'fun': lambda w: w.sum() - 1}
    )
)
```

```python
res_df['Min Max Drawdown'] = res_mdd['x']
```

```python
res_df.plot(kind='barh')
plt.title('Portfolio Weights for Mag-7 Stocks')
plt.xlabel('Portfolio Weight')
plt.show()
```



Find the minimum draw down portfolio for the sample with complete data for the current Dow-Jones Industrial Average (DJIA) stocks

You can find the DJIA tickers on Wikipedia.

```
tickers = (
    pd.read_html(io='https://en.wikipedia.org/wiki/Dow_Jones_Industrial_Average')
    [2]
    ['Symbol']
    .to_list()
)
```

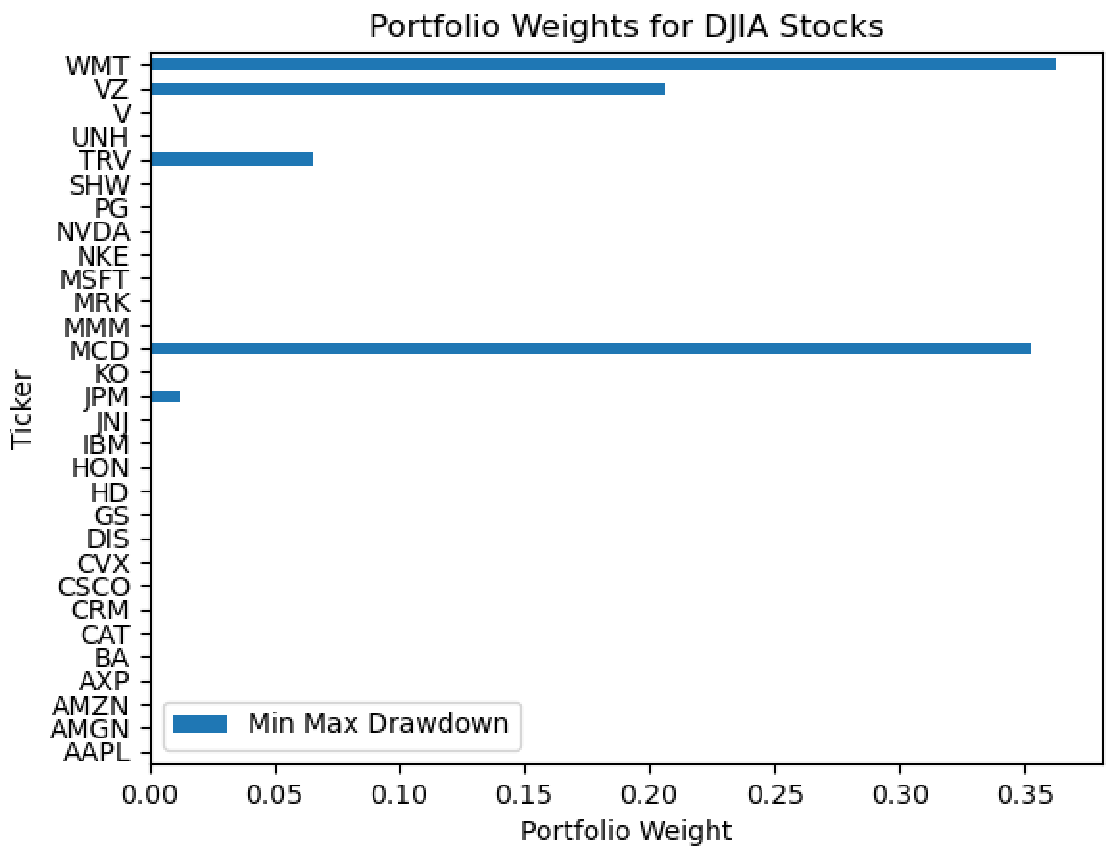
```
djia = (
    yf.download(
        tickers=tickers,
        auto_adjust=False,
```

```
        progress=False
    )
    .iloc[:-1]
    ['Adj Close']
    .pct_change()
    .dropna()
)
```

```
res_mdd = sco.minimize(
    fun=Max_Drawdown_neg,
    x0=equal_weights(djia.shape[1]),
    args=(djia,),
    bounds=[(0, 1) for _ in range(djia.shape[1])],
    constraints=(
        {'type': 'eq', 'fun': lambda w: w.sum() - 1}
    )
)
```

```
res_djia = pd.DataFrame(
    data={'Min Max Drawdown': res_mdd['x']},
    index=djia.columns
)
```

```
res_djia.plot(kind='barh')
plt.title('Portfolio Weights for DJIA Stocks')
plt.xlabel('Portfolio Weight')
plt.show()
```



Plot the minimum-variance frontier for the sample with complete data for the current the DJIA stocks

See the lecture notebook for a similar exercise with the Mag-7 stocks.

```
def calc_sigmap(w, Sigma, ppy=252):
    return np.sqrt(ppy * w.T @ Sigma @ w)
```

```
def calc_mup(w, mu, ppy=252):
    return ppy * w.T @ mu
```

```
Sigma = djia.cov()
```

```
mu = djia.mean()
```

```
djia_target = 252 * np.linspace(
    start=mu.min(),
    stop=mu.max(),
    num=50
)
```

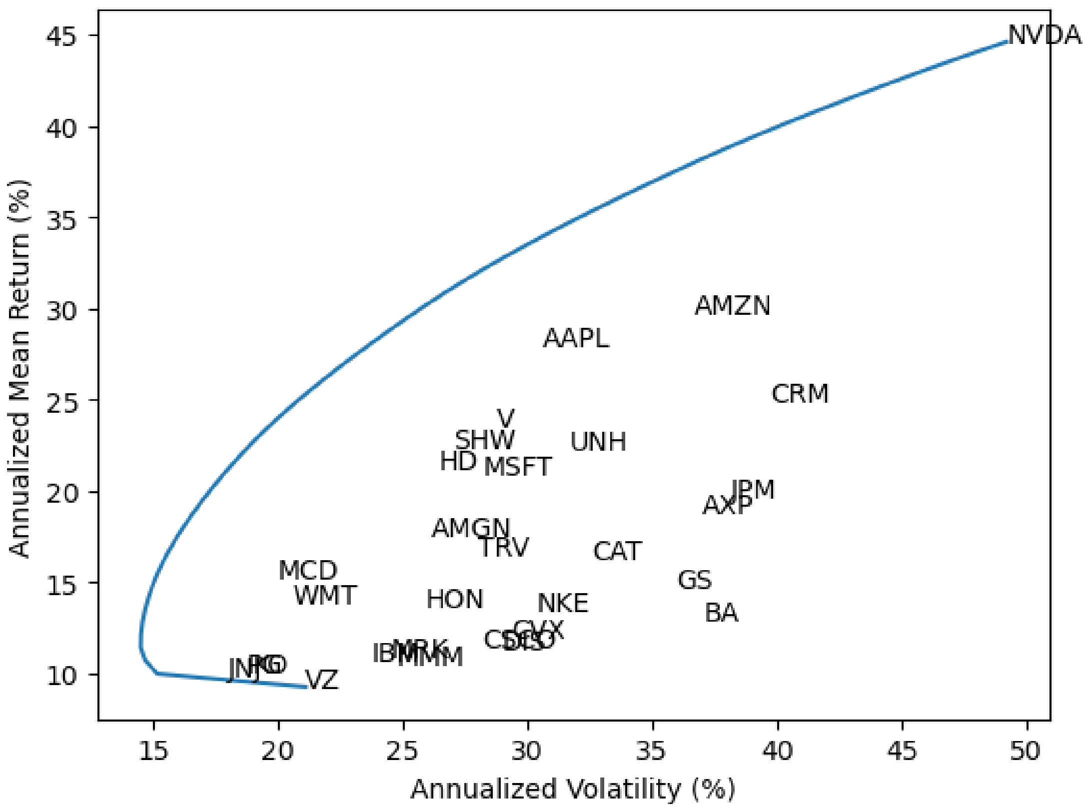
```
res_ef = []

for r in djia_target:
    _ = sco.minimize(
        fun=calc_sigmap,
        x0=np.ones(Sigma.shape[1]) / Sigma.shape[1],
        args=(Sigma, 252),
        bounds=[(0, 1) for c in Sigma.index],
        constraints=(
            {'type': 'eq', 'fun': lambda x: x.sum() - 1},
            {'type': 'eq', 'fun': lambda x: calc_mup(w=x, mu=mu, ppy=252) - r}
        )
    )
    res_ef.append(_)
```

```
for r in res_ef:
    assert r['success']
```

```
mv_frontier = pd.DataFrame({
    'returns_target': djia_target,
    'volatility': np.array([r['fun'] if r['success'] else np.nan for r in res_ef])
})
```

```
(
    mv_frontier
    .mul(100)
    .plot(x='volatility', y='returns_target', legend=False)
)
plt.ylabel('Annualized Mean Return (%)')
plt.xlabel('Annualized Volatility (%)')

for t in Sigma.index:
    x = 100 * np.sqrt(252 * Sigma.loc[t, t])
    y = 100 * 252 * mu.loc[t]
    plt.annotate(text=t, xy=(x, y))
```

```
plt.show()
```



Find the maximum Sharpe Ratio portfolio for the sample with complete data for the current the DJIA stocks excluding the last three years, so we can compare to the 1/n portfolio to the maximum Sharpe ratio portfolio

This exercise is less dramatic withe NVDA this year. We will see three things:

1. The maximum Sharpe ratio portfolio has a high Sharpe ratio *in sample* (or what we called `_before` in class)
2. But this same portfolio has a much lower Sharpe ratio *out of sample* (or what we called `_after` in class)
3. The 1/N or equal-weighted portfolio has about the same Sharpe ratio as the maximum Sharpe ratio portfolio *out of sample*

```
djia_excess = djia.sub(ff3['RF'], axis=0)
Sigma_excess_before = djia_excess.iloc[:-756].cov()
mu_excess_before = djia_excess.iloc[:-756].mean()
```

```
Sigma_excess_after = djia_excess.iloc[-756:].cov()
mu_excess_after = djia_excess.iloc[-756:].mean()
```

```
res_before = sco.minimize(
    fun=Sharpe_neg,
    x0=equal_weights(Sigma_excess_before.shape[1]),
    args=(Sigma_excess_before, mu_excess_before),
    bounds=[(0, 1) for _ in range(Sigma_excess_before.shape[1])],
    constraints=(
        {'type': 'eq', 'fun': lambda x: x.sum() - 1}
    )
)
```

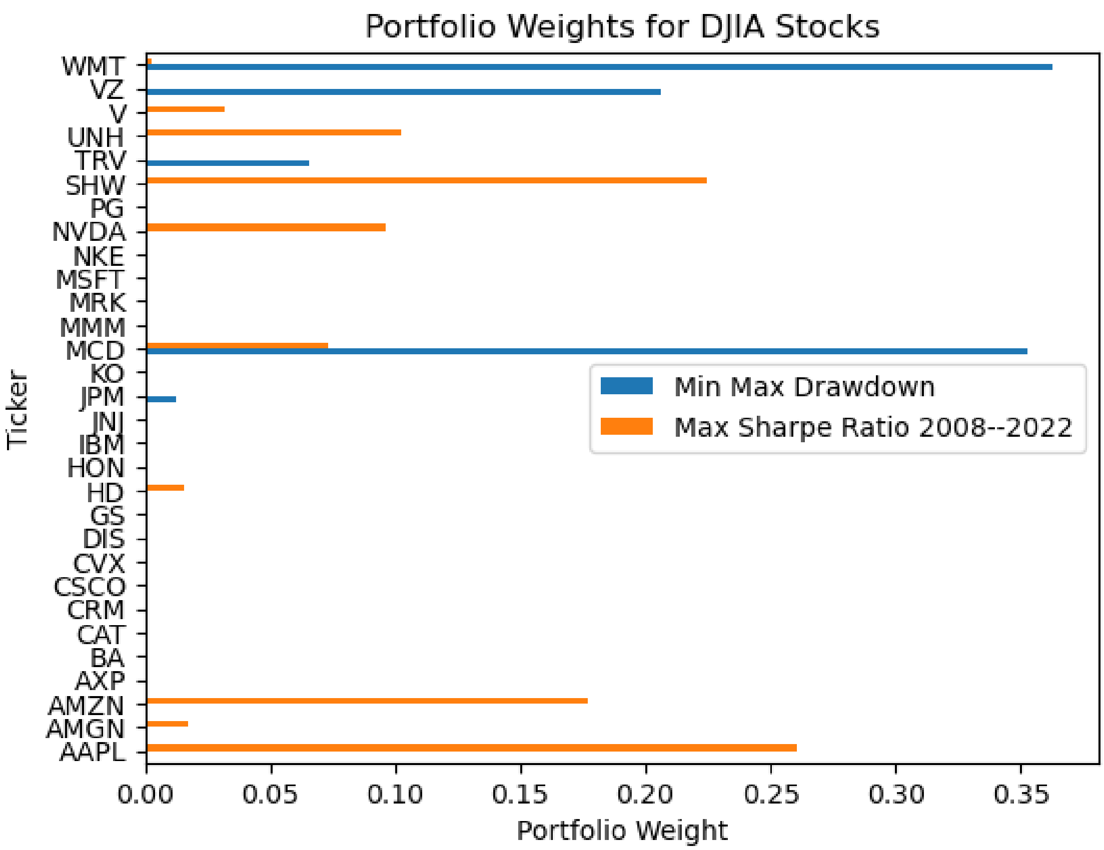
The maximum Sharpe ratio portfolio has a high Sharpe ratio *in sample*!

```
Sharpe(
    w=res_before['x'],
    Sigma_excess=Sigma_excess_after,
    mu_excess=mu_excess_after,
    ppy=252
)
```

```
0.7175
```

```
res_djia['Max Sharpe Ratio 2008--2022'] = res_before['x']
```

```
res_djia.plot(kind='barh')
plt.title('Portfolio Weights for DJIA Stocks')
plt.xlabel('Portfolio Weight')
plt.show()
```



The maximum Sharpe ratio portfolio has a much lower Sharpe ratio *out of sample*!

```
Sharpe(
    w=res_before['x'],
    Sigma_excess=Sigma_excess_after,
    mu_excess=mu_excess_after,
    ppy=252
)
```

```
0.7175
```

Furthermore, the 1/N or equal-weighted portfolio does about as well *out of sample*!

```
Sharpe(
    w=equal_weights(Sigma_excess_after.shape[1]),
    Sigma_excess=Sigma_excess_after,
    mu_excess=mu_excess_after,
```

```
    ppy=252
)
```

```
0.6134
```

Herron Topic 4 - Practice - Sec 03

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import pandas_datareader as pdr
import scipy.optimize as sco # new addition for portfolio optimization
import statsmodels.api as sm
import yfinance as yf
```

```
%precision 4
pd.options.display.float_format = '{:.4f}'.format
# %config InlineBackend.figure_format = 'retina'
```

Announcements

1. Please complete TRACE! I use TRACE to improve my treaching and courses, and I value your feedback. More here: https://northeastern.instructure.com/courses/207607/discussion_topics/2753925
2. Please plan for the in-class programming and MSFQ assessments on Tuesday, 4/15

Five-Minute Recap

Please see the lecture notebook for an in-depth explanation of how we will use `sco.minimize()` for portfolio optimization. Here are the key arguments to `sco.minimize()`:

1. `fun`: Name of function whose output we want to *minimize*
2. `x0`: First guess at inputs that *minimize* the output of the function in `fun`
3. `args`: A tuple of additional arguments to the function in `fun`
4. `bounds`: A list or tuple of tuples; For example, `((0, 1), (0, 1))` bounds inputs to fall between 0 and 1
5. `constraints`: A tuple of dictionaries with functions to constrain our inputs; For example, `{'type': 'eq', 'fun': lambda w: w.sum() - 1}` constraints te sum or our inputs to 1

Practice

Find the maximum Sharpe Ratio portfolio of Mag 7 stocks over the last three years

Note that `sco.minimize()` finds *minimums*, so you need to minimize the *negative* Sharpe Ratio.

```
mag7 = (
    yf.download(
        tickers='GOOGL AMZN AAPL META MSFT NVDA TSLA',
        auto_adjust=False,
        progress=False
    )
    .iloc[:-1] # drop incomplete trading day
)
```

```
returns = mag7['Adj Close'].pct_change().iloc[-756:]
```

We need the risk-free rate of return to calculate Sharpe ratios. French provides the risk-free rate of return as `RF` in most of his data sets.

```
ff3 = (
    pdr.DataReader(
        name='F-F_Research_Data_Factors_daily',
        data_source='famafrench',
        start='1900'
    )
    [0]
    .div(100)
)
```

```
C:\Users\r.herron\AppData\Local\Temp\ipykernel_19476\582763811.py:2: FutureWarning: The argu
  pdr.DataReader(
```

The Sharpe ratio is the ratio of the mean portfolio *excess* return to the volatility of portfolio *excess* returns.

$$S_p = \frac{\overline{r_p - r_f}}{\sigma(r_p - r_f)}$$

We can simplify this calculation if we calculate a data frame of *excess* returns. Then, we can use this data frame of excess returns to define the covariance matrix and mean returns.

```
returns_excess = returns.sub(ff3['RF'], axis=0)
Sigma_excess = returns_excess.cov()
mu_excess = returns_excess.mean()
```

The maximize the Sharpe ratio, we need to *minimize* the *negative* Sharpe ratio because there is no `maximize()` function. To simplify our code, we can define `Sharpe_neg()` and `Sharpe()` functions. We will use the former with `minimize()` and the latter everwhere else. The `Sharpe()` function uses matrix math, where `@` is the Numpy and pandas symbol for matrix multiplication.

1. `w.T @ mu_excess` is code for $w'\mu_{\text{excess}}$
2. `w.T @ Sigma_excess @ w` is code for $w'\Sigma_{\text{excess}}\mu$

```
def Sharpe(w, Sigma_excess, mu_excess, ppy=252):
    return (ppy * w.T @ mu_excess) / np.sqrt(ppy * w.T @ Sigma_excess @ w)
```

```
def Sharpe_neg(w, Sigma_excess, mu_excess, ppy=252):
    return -1 * Sharpe(w=w, Sigma_excess=Sigma_excess, mu_excess=mu_excess, ppy=252)
```

```
def equal_weights(n):
    return np.ones(n) / n
```

```
equal_weights(Sigma_excess.shape[1])
```

```
array([0.1429, 0.1429, 0.1429, 0.1429, 0.1429, 0.1429, 0.1429])
```

```
[(0, 1) for _ in range(Sigma_excess.shape[1])]
```

```
[(0, 1), (0, 1), (0, 1), (0, 1), (0, 1), (0, 1), (0, 1)]
```

We can store our `minimize()` results in `res_X` and increment `X` for each of the following practices.

```
res_1 = sco.minimize(
    fun=Sharpe_neg,
    x0=equal_weights(Sigma_excess.shape[1]),
    args=(Sigma_excess, mu_excess),
    bounds=[(0, 1) for _ in range(7)],
    constraints=(
```

```
        {'type': 'eq', 'fun': lambda x: x.sum() - 1}
    )
)
```

We want to make sure that `minimize()` finds a solution (i.e., `res_1['success']` is `True`). The *negative* Sharpe ratio is the value for the `fun` key, and the portfolio weights are the value for the `x` key.

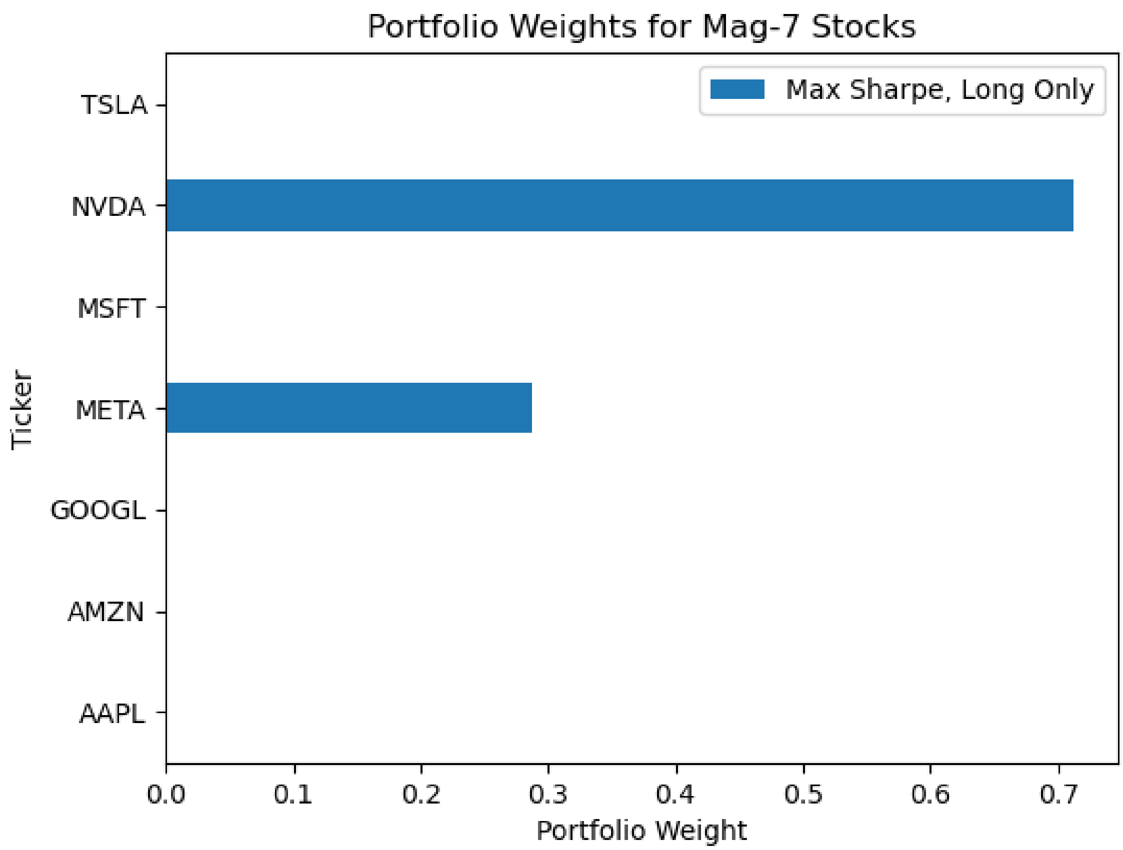
```
res_1
```

```
 message: Optimization terminated successfully
 success: True
  status: 0
     fun: -1.269574051296866
       x: [ 4.888e-17  0.000e+00  0.000e+00  2.880e-01  1.373e-17
            7.120e-01  5.408e-17]
     nit: 5
     jac: [ 1.624e-01  3.758e-01  2.860e-01 -4.549e-04  2.635e-01
            1.840e-04  3.562e-01]
    nfev: 40
    njev: 5
```

We can save these results to a data frame for easy updating and plotting.

```
res_df = pd.DataFrame(
    data={'Max Sharpe, Long Only': res_1['x']},
    index=Sigma_excess.columns
)
```

```
res_df.plot(kind='barh')
plt.title('Portfolio Weights for Mag-7 Stocks')
plt.xlabel('Portfolio Weight')
plt.show()
```



What if we want the actual, positive Sharpe ratio? We can use the `Sharpe()` function. If we want to update the results in `res_1`, we could write another helper function. However, I generally give in and learn to live with the output of commonly used functions.

```
res_1
```

```
 message: Optimization terminated successfully
 success: True
  status: 0
     fun: -1.269574051296866
       x: [ 4.888e-17  0.000e+00  0.000e+00  2.880e-01  1.373e-17
            7.120e-01  5.408e-17]
     nit: 5
     jac: [ 1.624e-01  3.758e-01  2.860e-01 -4.549e-04  2.635e-01
```

```
            1.840e-04  3.562e-01]
    nfev: 40
    njev: 5
```

```
Sharpe(w=res_1['x'], Sigma_excess=Sigma_excess, mu_excess=mu_excess)
```

```
1.2696
```

Find the maximum Sharpe Ratio portfolio of Mag 7 stocks over the last three years, but allow short weights up to 10% on each stock

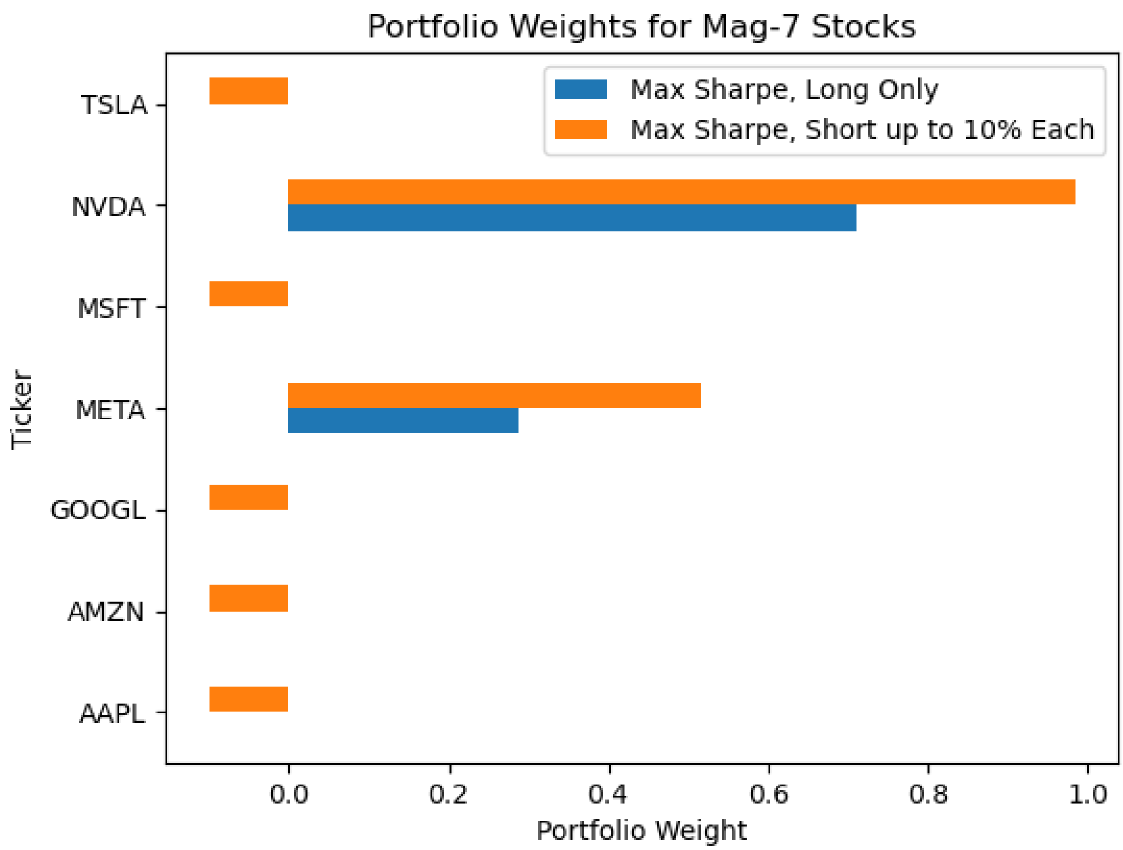
We can short 6 of 7 stocks up to -0.1 for a total of -0.6. Therefore, the maximum possible long weight is 1.6, and `bounds` becomes:

```
    bounds=[(-0.1, 1.6) for _ in range(7)],
```

```
res_2 = sco.minimize(
    fun=Sharpe_neg,
    x0=equal_weights(Sigma_excess.shape[1]),
    args=(Sigma_excess, mu_excess),
    bounds=[(-0.1, 1.6) for _ in range(7)],
    constraints=(
        {'type': 'eq', 'fun': lambda x: x.sum() - 1}
    )
)
```

```
res_df['Max Sharpe, Short up to 10% Each'] = res_2['x']
```

```
res_df.plot(kind='barh')
plt.title('Portfolio Weights for Mag-7 Stocks')
plt.xlabel('Portfolio Weight')
plt.show()
```



Find the maximum Sharpe Ratio portfolio of Mag 7 stocks over the last three years, but allow total short weights of up to 30%

We need to use an unequality constraints to make sure the *sum* of the negative portfolios weights is greater than -0.3. We express inequality constraints with functions with non-negative outputs, so we use $\sum_i w_i[w_i < 0] + 0.3 \geq 0$.

We see that once we allow shorts greater than 30%, we spend almost all of this short budget on AMZN.

```
toy = np.arange(-2, 3)
```

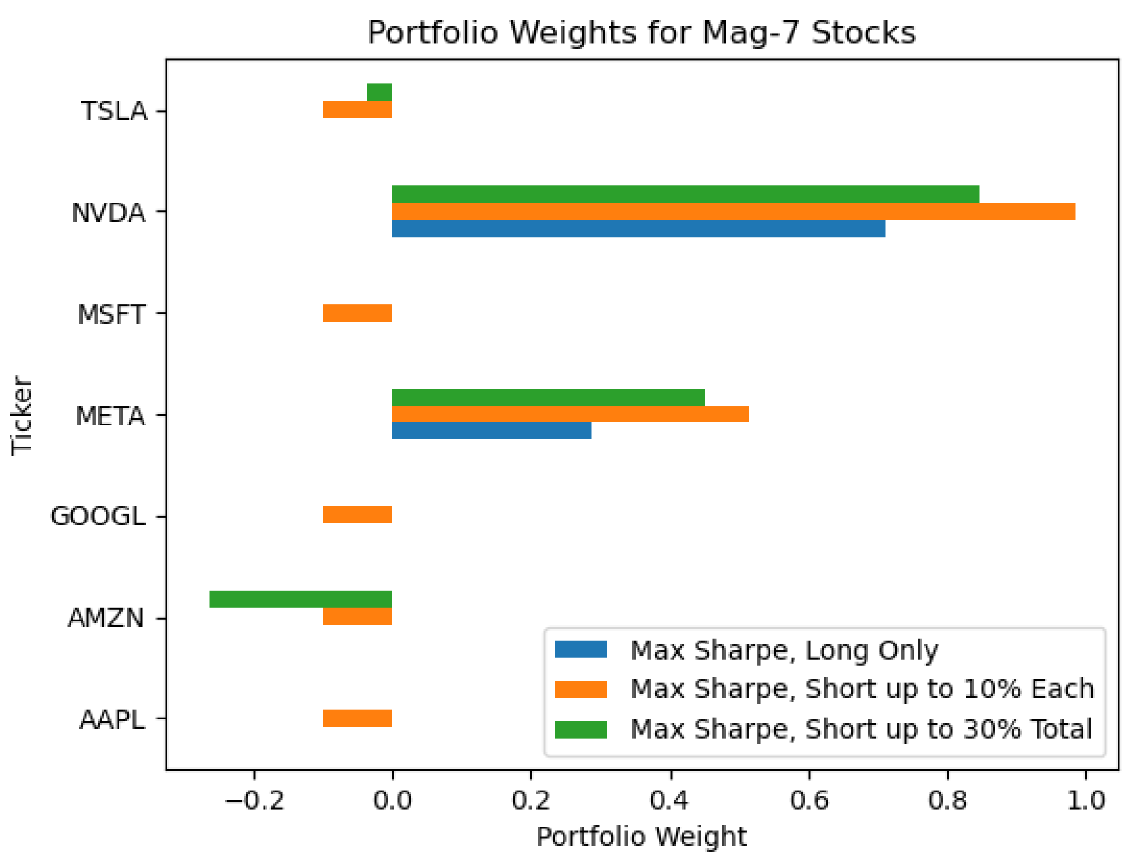
```
toy[toy < 0].sum()
```

```
np.int64(-3)
```

```
res_3 = sco.minimize(
    fun=Sharpe_neg,
    x0=equal_weights(Sigma_excess.shape[1]),
    args=(Sigma_excess, mu_excess),
    bounds=[(-0.3, 1.3) for _ in range(7)],
    constraints=(
        {'type': 'eq', 'fun': lambda x: x.sum() - 1},
        {'type': 'ineq', 'fun': lambda x: x[x < 0].sum() + 0.3}
    )
)
```

```
res_df['Max Sharpe, Short up to 30% Total'] = res_3['x']
```

```
res_df.plot(kind='barh')
plt.title('Portfolio Weights for Mag-7 Stocks')
plt.xlabel('Portfolio Weight')
plt.show()
```



Find the maximum Sharpe Ratio portfolio of Mag 7 stocks over the last three years, but do not allow any weight to exceed 30% in magnitude

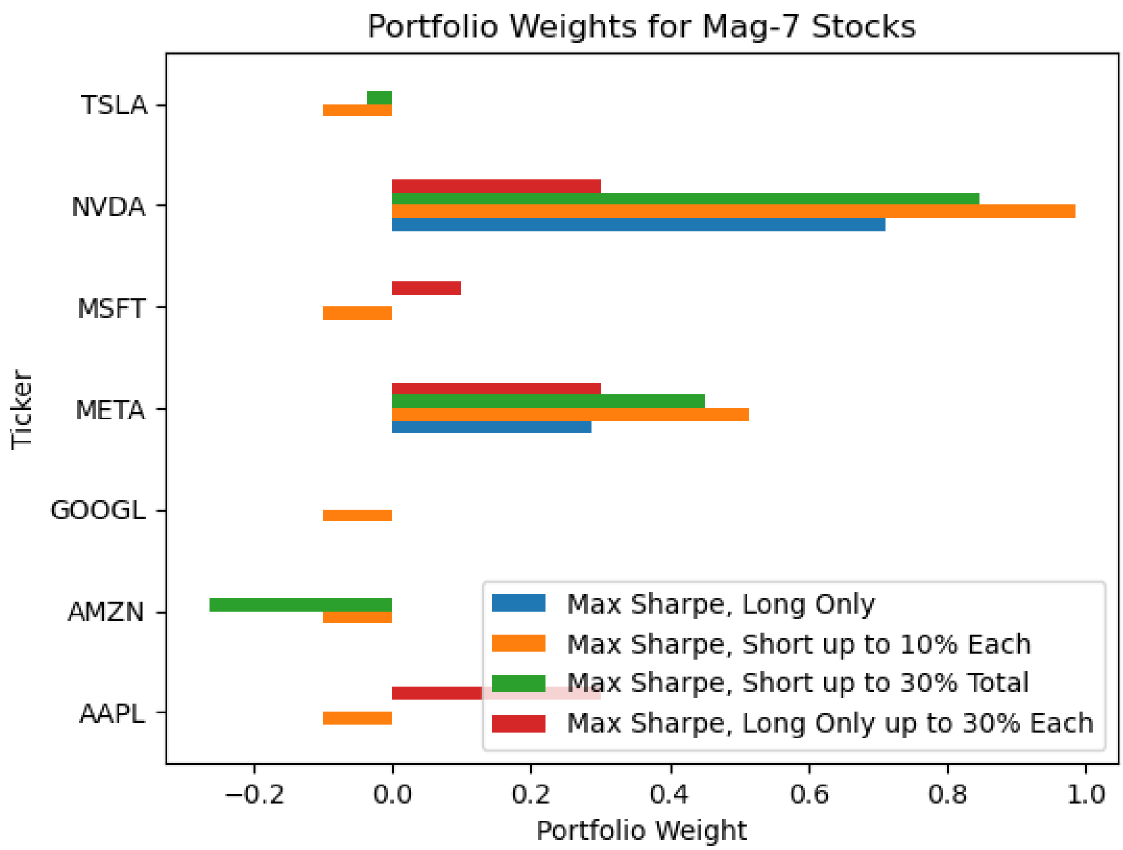
We can bound all portfolios weights on $[0, 0.3]$ with:

```
bounds=[(0, 0.3) for _ in range(7)]
```

```
res_4 = sco.minimize(
    fun=Sharpe_neg,
    x0=equal_weights(Sigma_excess.shape[1]),
    args=(Sigma_excess, mu_excess),
    bounds=[(0, 0.3) for _ in range(7)],
    constraints=(
        {'type': 'eq', 'fun': lambda x: x.sum() - 1}
    )
)
```

```
res_df['Max Sharpe, Long Only up to 30% Each'] = res_4['x']
```

```
res_df.plot(kind='barh')
plt.title('Portfolio Weights for Mag-7 Stocks')
plt.xlabel('Portfolio Weight')
plt.show()
```



Find the minimum 95% Value at Risk (Var) portfolio of Mag 7 stocks over the last three years

More on VaR here.

```
def VaR(weights, returns, percent):
    return returns.dot(weights).quantile(1 - percent)
```

```
VaR(
    weights=equal_weights(returns.shape[1]),
    returns=returns,
    percent=0.95
)
```

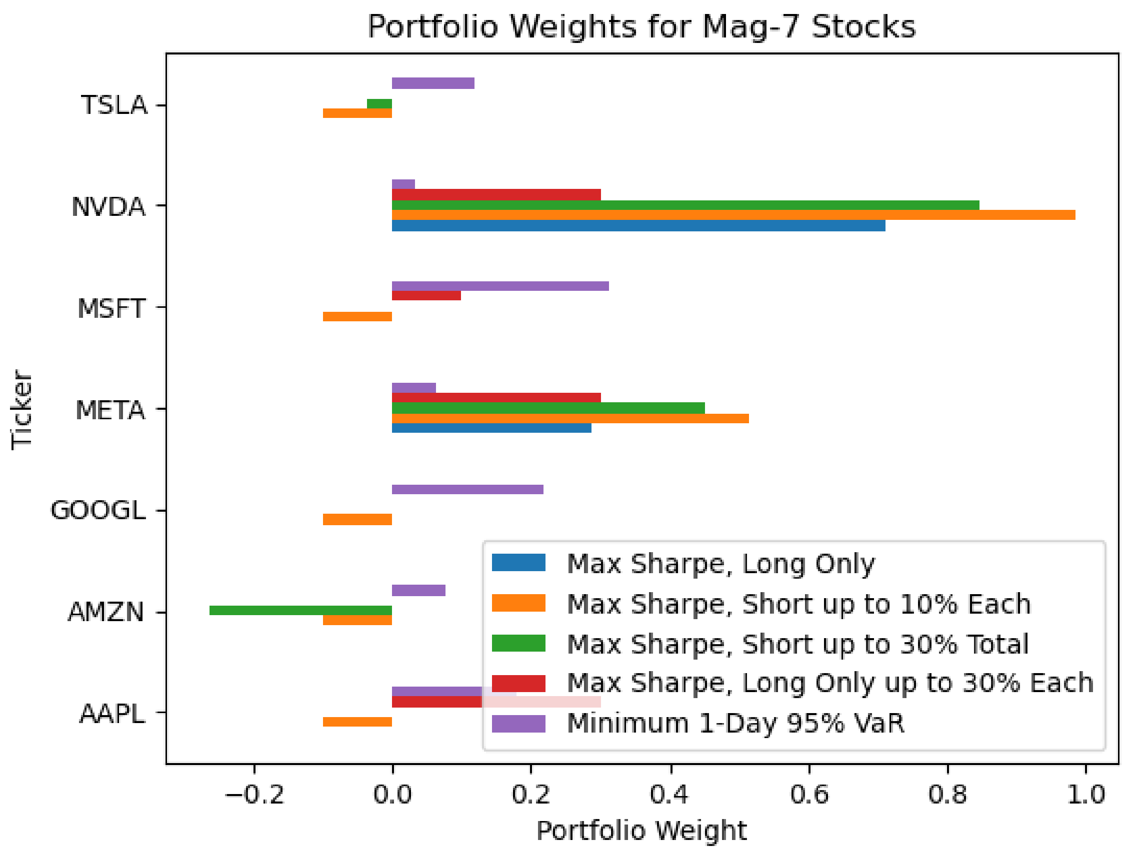
```
-0.0336
```

```
def VaR_neg(weights, returns, percent):
    return -1 * VaR(weights=weights, returns=returns, percent=percent)
```

```
res_var = sco.minimize(
    fun=VaR_neg,
    x0=equal_weights(returns.shape[1]), # np.ones(7) / 7
    args=(returns, 0.95),
    bounds=[(0, 1) for _ in range(returns.shape[1])],
    constraints=(
        {'type': 'eq', 'fun': lambda w: w.sum() - 1}
    )
)
```

```
res_df['Minimum 1-Day 95% VaR'] = res_var['x']
```

```
res_df.plot(kind='barh')
plt.title('Portfolio Weights for Mag-7 Stocks')
plt.xlabel('Portfolio Weight')
plt.show()
```



Find the minimum draw down portfolio of Mag 7 stocks over the last three years

```
def Max_Drawdown(w, returns):
    price = returns.dot(w).add(1).cumprod()
    return (price / price.cummax() - 1).min()
```

```
Max_Drawdown(equal_weights(7), returns)
```

```
-0.4524
```

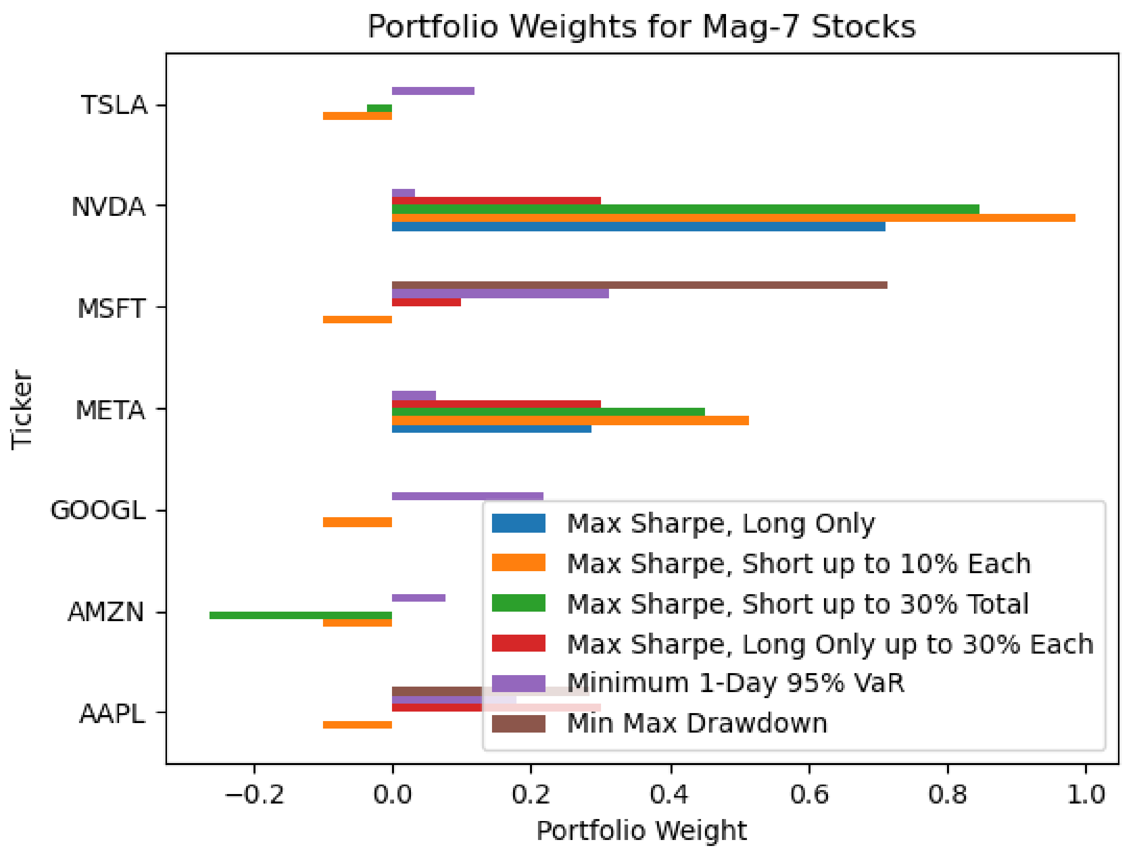
The maximum drawdown is a negative number, but we want the *smallest* negative number, so we minimize negative one times the maximum drawdown.

```
def Max_Drawdown_neg(w, returns):
    return -1 * Max_Drawdown(w=w, returns=returns)
```

```
res_mdd = sco.minimize(
    fun=Max_Drawdown_neg,
    x0=equal_weights(returns.shape[1]), # np.ones(7) / 7
    args=(returns,),
    bounds=[(0, 1) for _ in range(returns.shape[1])],
    constraints=(
        {'type': 'eq', 'fun': lambda w: w.sum() - 1}
    )
)
```

```
res_df['Min Max Drawdown'] = res_mdd['x']
```

```
res_df.plot(kind='barh')
plt.title('Portfolio Weights for Mag-7 Stocks')
plt.xlabel('Portfolio Weight')
plt.show()
```



Find the minimum draw down portfolio for the sample with complete data for the current Dow-Jones Industrial Average (DJIA) stocks

You can find the DJIA tickers on Wikipedia.

```
tickers = (
    pd.read_html(io='https://en.wikipedia.org/wiki/Dow_Jones_Industrial_Average')
    [2]
    ['Symbol']
    .to_list()
)
```

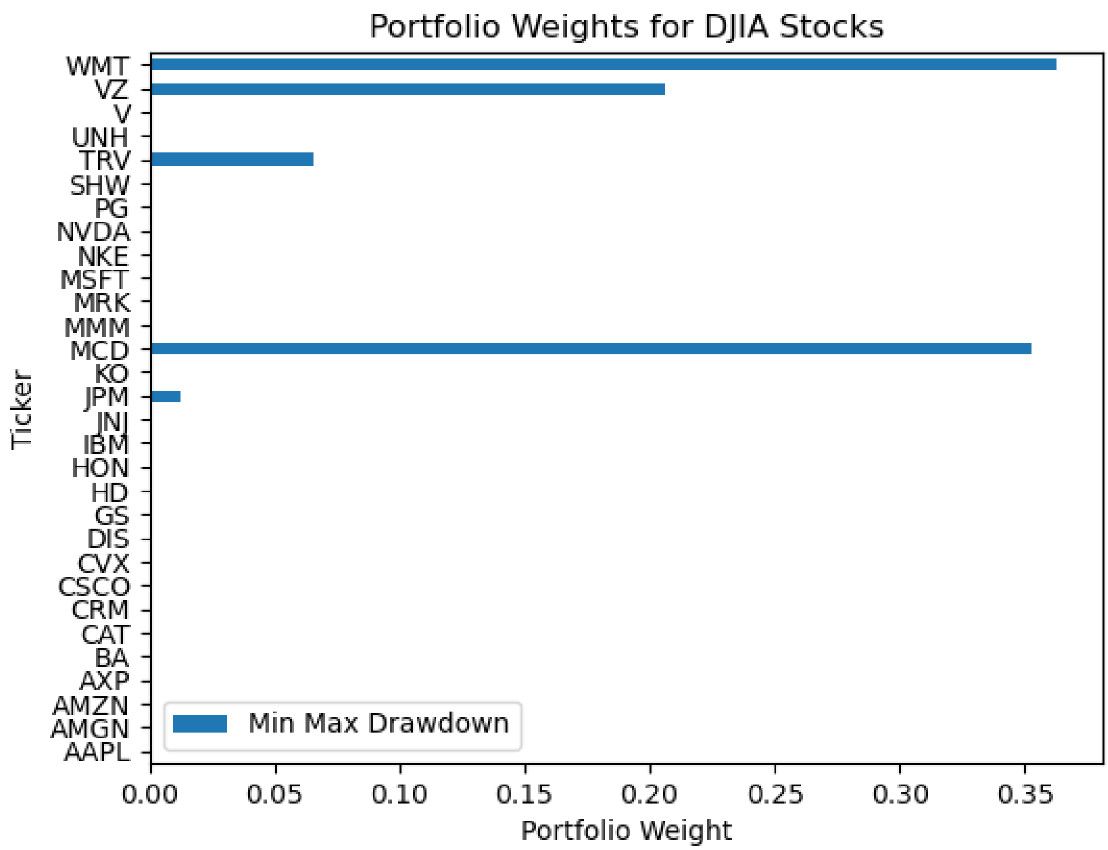
```
djia = (
    yf.download(
        tickers=tickers,
        auto_adjust=False,
```

```
        progress=False
    )
    .iloc[:-1]
    ['Adj Close']
    .pct_change()
    .dropna()
)
```

```
res_mdd = sco.minimize(
    fun=Max_Drawdown_neg,
    x0=equal_weights(djia.shape[1]),
    args=(djia,),
    bounds=[(0, 1) for _ in range(djia.shape[1])],
    constraints=(
        {'type': 'eq', 'fun': lambda w: w.sum() - 1}
    )
)
```

```
res_djia = pd.DataFrame(
    data={'Min Max Drawdown': res_mdd['x']},
    index=djia.columns
)
```

```
res_djia.plot(kind='barh')
plt.title('Portfolio Weights for DJIA Stocks')
plt.xlabel('Portfolio Weight')
plt.show()
```



Plot the minimum-variance frontier for the sample with complete data for the current the DJIA stocks

See the lecture notebook for a similar exercise with the Mag-7 stocks.

```
def calc_sigmap(w, Sigma, ppy=252):
    return np.sqrt(ppy * w.T @ Sigma @ w)
```

```
def calc_mup(w, mu, ppy=252):
    return ppy * w.T @ mu
```

```
Sigma = djia.cov()
```

```
mu = djia.mean()
```

```
djia_target = 252 * np.linspace(
    start=mu.min(),
    stop=mu.max(),
    num=50
)
```

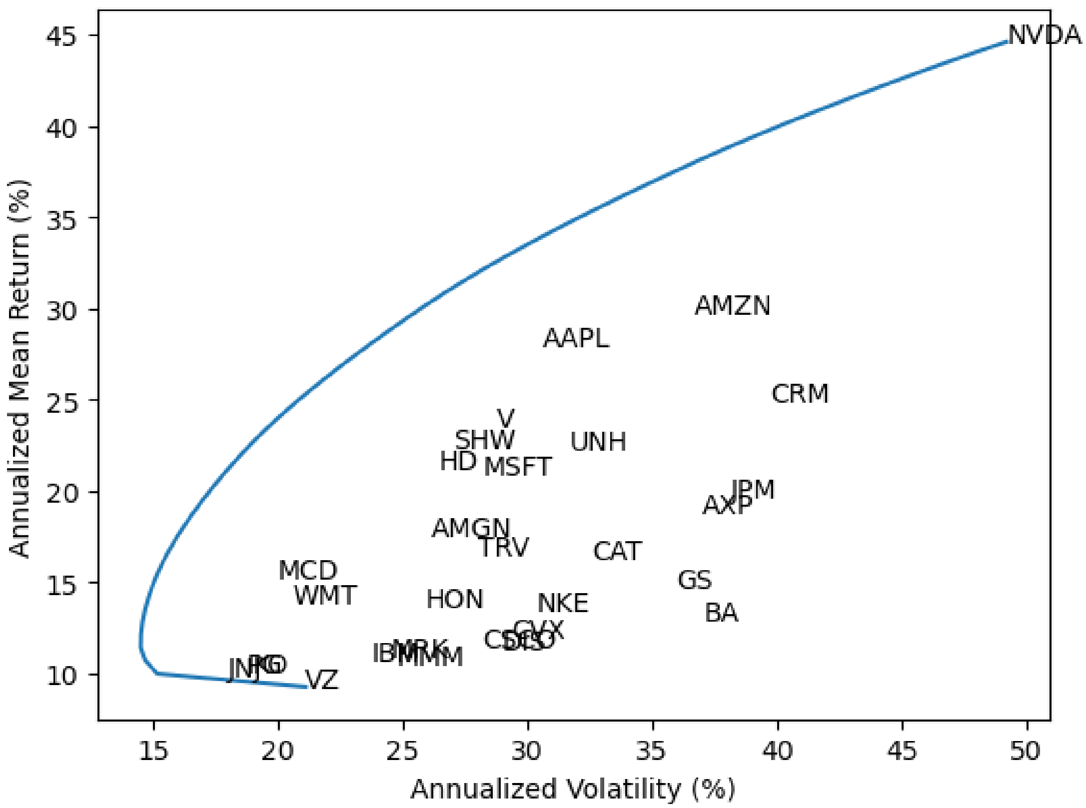
```
res_ef = []

for r in djia_target:
    _ = sco.minimize(
        fun=calc_sigmap,
        x0=np.ones(Sigma.shape[1]) / Sigma.shape[1],
        args=(Sigma, 252),
        bounds=[(0, 1) for c in Sigma.index],
        constraints=(
            {'type': 'eq', 'fun': lambda x: x.sum() - 1},
            {'type': 'eq', 'fun': lambda x: calc_mup(w=x, mu=mu, ppy=252) - r}
        )
    )
    res_ef.append(_)
```

```
for r in res_ef:
    assert r['success']
```

```
mv_frontier = pd.DataFrame({
    'returns_target': djia_target,
    'volatility': np.array([r['fun'] if r['success'] else np.nan for r in res_ef])
})
```

```
(
    mv_frontier
    .mul(100)
    .plot(x='volatility', y='returns_target', legend=False)
)
plt.ylabel('Annualized Mean Return (%)')
plt.xlabel('Annualized Volatility (%)')

for t in Sigma.index:
    x = 100 * np.sqrt(252 * Sigma.loc[t, t])
    y = 100 * 252 * mu.loc[t]
    plt.annotate(text=t, xy=(x, y))
```

```
plt.show()
```



Find the maximum Sharpe Ratio portfolio for the sample with complete data for the current the DJIA stocks excluding the last three years, so we can compare to the 1/n portfolio to the maximum Sharpe ratio portfolio

This exercise is less dramatic withe NVDA this year. We will see three things:

1. The maximum Sharpe ratio portfolio has a high Sharpe ratio *in sample* (or what we called `_before` in class)
2. But this same portfolio has a much lower Sharpe ratio *out of sample* (or what we called `_after` in class)
3. The 1/N or equal-weighted portfolio has about the same Sharpe ratio as the maximum Sharpe ratio portfolio *out of sample*

```
djia_excess = djia.sub(ff3['RF'], axis=0)
Sigma_excess_before = djia_excess.iloc[:-756].cov()
mu_excess_before = djia_excess.iloc[:-756].mean()
```

```
Sigma_excess_after = djia_excess.iloc[-756:].cov()
mu_excess_after = djia_excess.iloc[-756:].mean()
```

```
res_before = sco.minimize(
    fun=Sharpe_neg,
    x0=equal_weights(Sigma_excess_before.shape[1]),
    args=(Sigma_excess_before, mu_excess_before),
    bounds=[(0, 1) for _ in range(Sigma_excess_before.shape[1])],
    constraints=(
        {'type': 'eq', 'fun': lambda x: x.sum() - 1}
    )
)
```

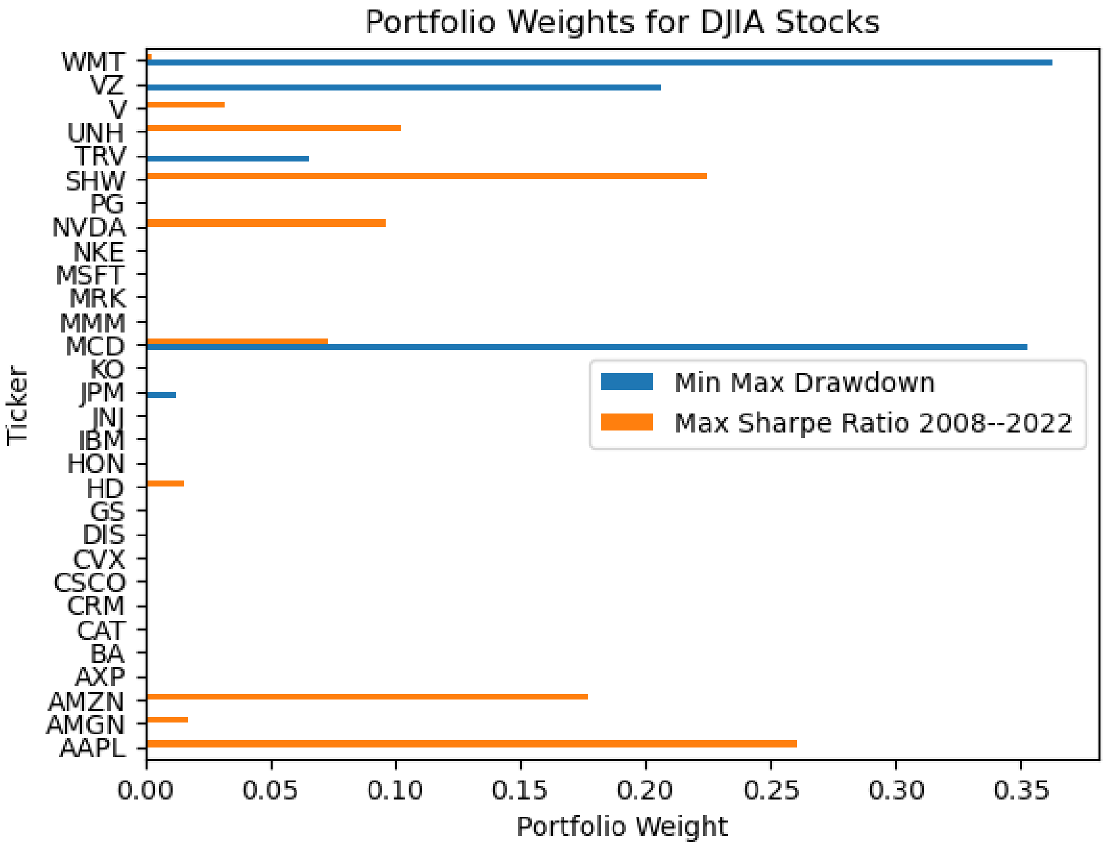
The maximum Sharpe ratio portfolio has a high Sharpe ratio *in sample*!

```
Sharpe(
    w=res_before['x'],
    Sigma_excess=Sigma_excess_after,
    mu_excess=mu_excess_after,
    ppy=252
)
```

```
0.7175
```

```
res_djia['Max Sharpe Ratio 2008--2022'] = res_before['x']
```

```
res_djia.plot(kind='barh')
plt.title('Portfolio Weights for DJIA Stocks')
plt.xlabel('Portfolio Weight')
plt.show()
```



The maximum Sharpe ratio portfolio has a much lower Sharpe ratio *out of sample*!

```
Sharpe(
    w=res_before['x'],
    Sigma_excess=Sigma_excess_after,
    mu_excess=mu_excess_after,
    ppy=252
)
```

```
0.7175
```

Furthermore, the 1/N or equal-weighted portfolio does about as well *out of sample*!

```
Sharpe(
    w=equal_weights(Sigma_excess_after.shape[1]),
    Sigma_excess=Sigma_excess_after,
    mu_excess=mu_excess_after,
```

```
    ppy=252
)
```

```
0.6134
```

Herron Topic 4 - Practice - Sec 04

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import pandas_datareader as pdr
import scipy.optimize as sco # new addition for portfolio optimization
import statsmodels.api as sm
import yfinance as yf
```

```
%precision 4
pd.options.display.float_format = '{:.4f}'.format
# %config InlineBackend.figure_format = 'retina'
```

Announcements

1. Please complete TRACE! I use TRACE to improve my treaching and courses, and I value your feedback. More here: https://northeastern.instructure.com/courses/207607/discussion_topics/2753925
2. Please plan for the in-class programming and MSFQ assessments on Tuesday, 4/15

Five-Minute Recap

Please see the lecture notebook for an in-depth explanation of how we will use `sco.minimize()` for portfolio optimization. Here are the key arguments to `sco.minimize()`:

1. `fun`: Name of function whose output we want to *minimize*
2. `x0`: First guess at inputs that *minimize* the output of the function in `fun`
3. `args`: A tuple of additional arguments to the function in `fun`
4. `bounds`: A list or tuple of tuples; For example, `((0, 1), (0, 1))` bounds inputs to fall between 0 and 1
5. `constraints`: A tuple of dictionaries with functions to constrain our inputs; For example, `{'type': 'eq', 'fun': lambda w: w.sum() - 1}` constraints te sum or our inputs to 1

Practice

Find the maximum Sharpe Ratio portfolio of Mag 7 stocks over the last three years

Note that `sco.minimize()` finds *minimums*, so you need to minimize the *negative* Sharpe Ratio.

```
mag7 = (
    yf.download(
        tickers='GOOGL AMZN AAPL META MSFT NVDA TSLA',
        auto_adjust=False,
        progress=False
    )
    .iloc[:-1] # drop incomplete trading day
)
```

```
returns = mag7['Adj Close'].pct_change().iloc[-756:]
```

We need the risk-free rate of return to calculate Sharpe ratios. French provides the risk-free rate of return as `RF` in most of his data sets.

```
ff3 = (
    pdr.DataReader(
        name='F-F_Research_Data_Factors_daily',
        data_source='famafrench',
        start='1900'
    )
    [0]
    .div(100)
)
```

```
C:\Users\r.herron\AppData\Local\Temp\ipykernel_19752\582763811.py:2: FutureWarning: The argu
  pdr.DataReader(
```

The Sharpe ratio is the ratio of the mean portfolio *excess* return to the volatility of portfolio *excess* returns.

$$S_p = \frac{\overline{r_p - r_f}}{\sigma(r_p - r_f)}$$

We can simplify this calculation if we calculate a data frame of *excess* returns. Then, we can use this data frame of excess returns to define the covariance matrix and mean returns.

```python
returns_excess = returns.sub(ff3['RF'], axis=0)
Sigma_excess = returns_excess.cov()
mu_excess = returns_excess.mean()
```

The maximize the Sharpe ratio, we need to *minimize* the *negative* Sharpe ratio because there is no `maximize()` function. To simplify our code, we can define `Sharpe_neg()` and `Sharpe()` functions. We will use the former with `minimize()` and the latter everwhere else. The `Sharpe()` function uses matrix math, where `@` is the Numpy and pandas symbol for matrix multiplication.

1. `w.T @ mu_excess` is code for $w'\mu_{\text{excess}}$
2. `w.T @ Sigma_excess @ w` is code for $w'\Sigma_{\text{excess}}\mu$

```python
def Sharpe(w, Sigma_excess, mu_excess, ppy=252):
    return (ppy * w.T @ mu_excess) / np.sqrt(ppy * w.T @ Sigma_excess @ w)
```

```python
def Sharpe_neg(w, Sigma_excess, mu_excess, ppy=252):
    return -1 * Sharpe(w=w, Sigma_excess=Sigma_excess, mu_excess=mu_excess, ppy=252)
```

```python
def equal_weights(n):
    return np.ones(n) / n
```

```python
equal_weights(Sigma_excess.shape[1])
```

```
array([0.1429, 0.1429, 0.1429, 0.1429, 0.1429, 0.1429, 0.1429])
```

```python
[(0, 1) for _ in range(Sigma_excess.shape[1])]
```

```
[(0, 1), (0, 1), (0, 1), (0, 1), (0, 1), (0, 1), (0, 1)]
```

We can store our `minimize()` results in `res_X` and increment `X` for each of the following practices.

```python
res_1 = sco.minimize(
    fun=Sharpe_neg,
    x0=equal_weights(Sigma_excess.shape[1]),
    args=(Sigma_excess, mu_excess),
    bounds=[(0, 1) for _ in range(7)],
    constraints=(
```

```
        {'type': 'eq', 'fun': lambda x: x.sum() - 1}
    )
)
```

We want to make sure that `minimize()` finds a solution (i.e., `res_1['success']` is `True`). The *negative* Sharpe ratio is the value for the `fun` key, and the portfolio weights are the value for the `x` key.

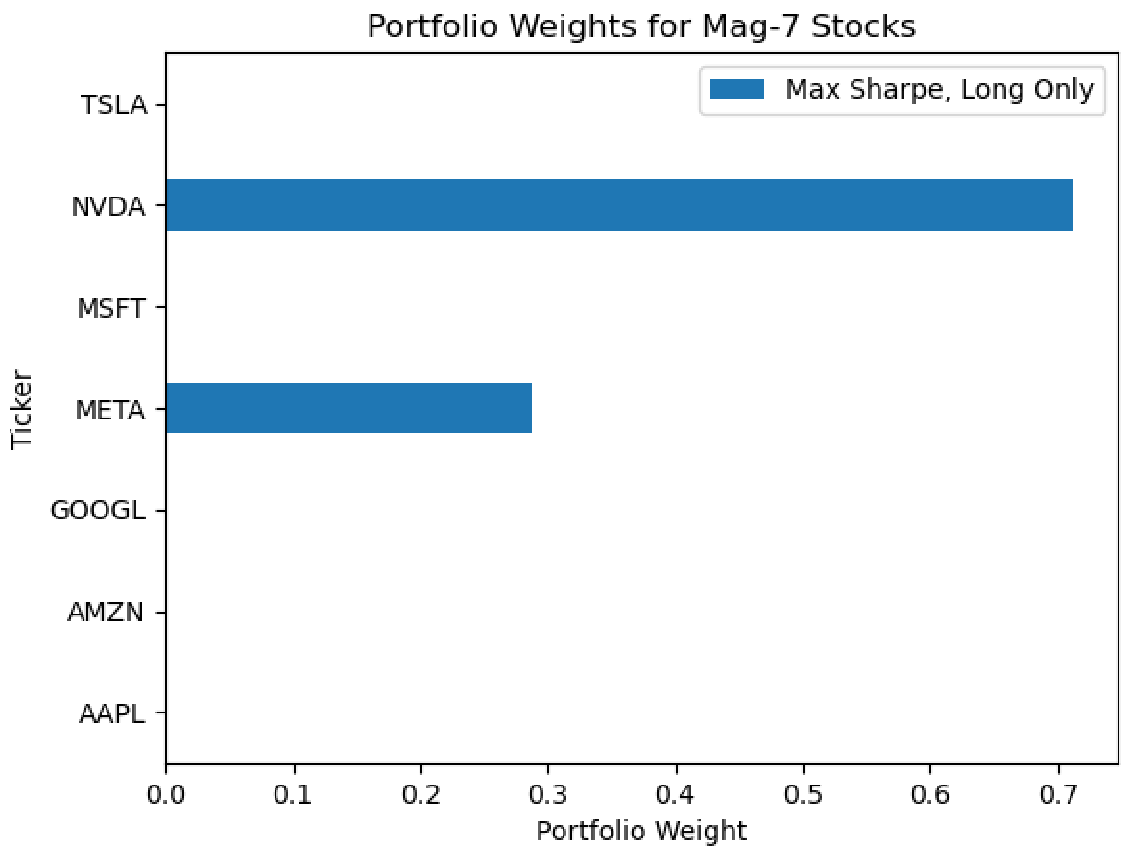
```
res_1
```

```
 message: Optimization terminated successfully
 success: True
  status: 0
     fun: -1.2695740617740525
       x: [ 1.292e-17  6.889e-18  0.000e+00  2.880e-01  5.088e-17
            7.120e-01  0.000e+00]
     nit: 5
     jac: [ 1.624e-01  3.758e-01  2.860e-01 -4.549e-04  2.635e-01
            1.840e-04  3.562e-01]
    nfev: 40
    njev: 5
```

We can save these results to a data frame for easy updating and plotting.

```
res_df = pd.DataFrame(
    data={'Max Sharpe, Long Only': res_1['x']},
    index=Sigma_excess.columns
)
```

```
res_df.plot(kind='barh')
plt.title('Portfolio Weights for Mag-7 Stocks')
plt.xlabel('Portfolio Weight')
plt.show()
```



What if we want the actual, positive Sharpe ratio? We can use the `Sharpe()` function. If we want to update the results in `res_1`, we could write another helper function. However, I generally give in and learn to live with the output of commonly used functions.

```
res_1
```

```
 message: Optimization terminated successfully
 success: True
  status: 0
     fun: -1.2695740617740525
       x: [ 1.292e-17  6.889e-18  0.000e+00  2.880e-01  5.088e-17
            7.120e-01  0.000e+00]
     nit: 5
     jac: [ 1.624e-01  3.758e-01  2.860e-01 -4.549e-04  2.635e-01
```

```
             1.840e-04  3.562e-01]
    nfev: 40
    njev: 5
```

```
Sharpe(w=res_1['x'], Sigma_excess=Sigma_excess, mu_excess=mu_excess)
```

```
1.2696
```

Find the maximum Sharpe Ratio portfolio of Mag 7 stocks over the last three years, but allow short weights up to 10% on each stock

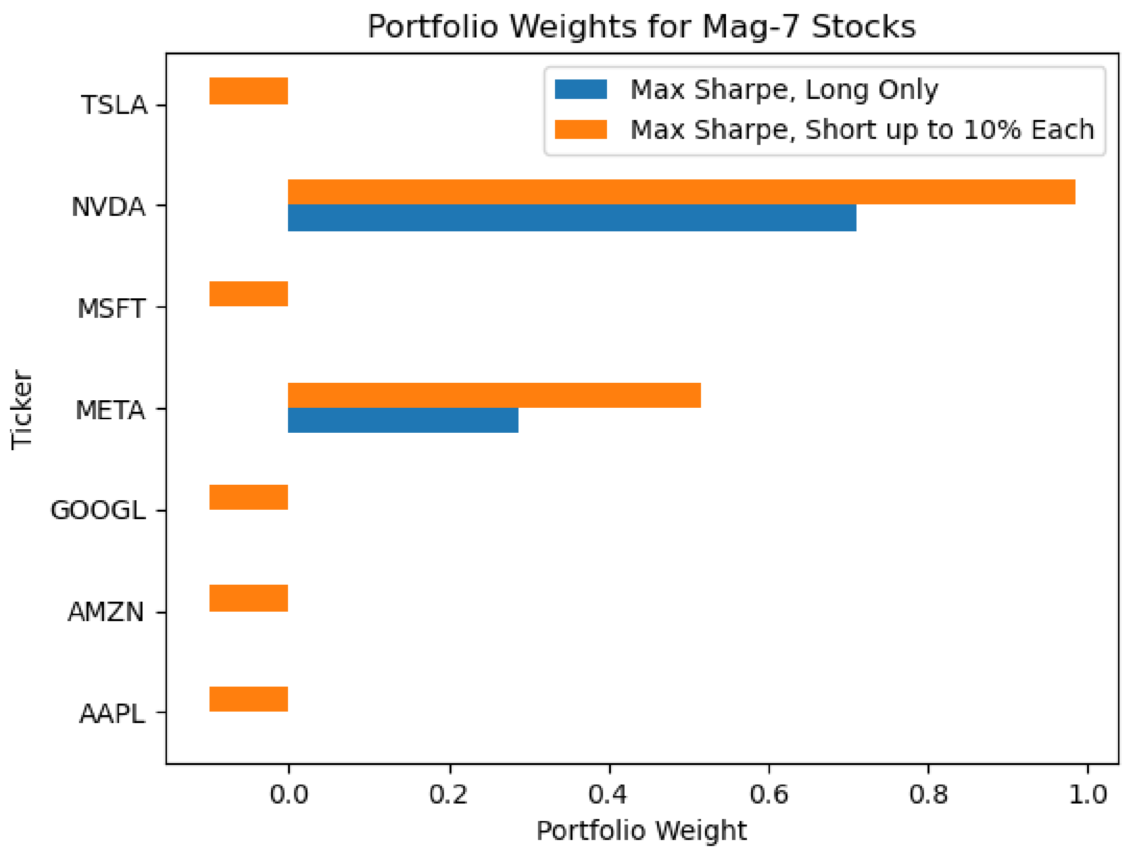
We can short 6 of 7 stocks up to -0.1 for a total of -0.6. Therefore, the maximum possible long weight is 1.6, and `bounds` becomes:

```
    bounds=[(-0.1, 1.6) for _ in range(7)],
```

```
res_2 = sco.minimize(
    fun=Sharpe_neg,
    x0=equal_weights(Sigma_excess.shape[1]),
    args=(Sigma_excess, mu_excess),
    bounds=[(-0.1, 1.6) for _ in range(7)],
    constraints=(
        {'type': 'eq', 'fun': lambda x: x.sum() - 1}
    )
)
```

```
res_df['Max Sharpe, Short up to 10% Each'] = res_2['x']
```

```
res_df.plot(kind='barh')
plt.title('Portfolio Weights for Mag-7 Stocks')
plt.xlabel('Portfolio Weight')
plt.show()
```



Find the maximum Sharpe Ratio portfolio of Mag 7 stocks over the last three years, but allow total short weights of up to 30%

We need to use an unequality constraints to make sure the *sum* of the negative portfolios weights is greater than -0.3. We express inequality constraints with functions with non-negative outputs, so we use $\sum_i w_i [w_i < 0] + 0.3 \geq 0$.

We see that once we allow shorts greater than 30%, we spend almost all of this short budget on AMZN.

```
toy = np.arange(-2, 3)
```

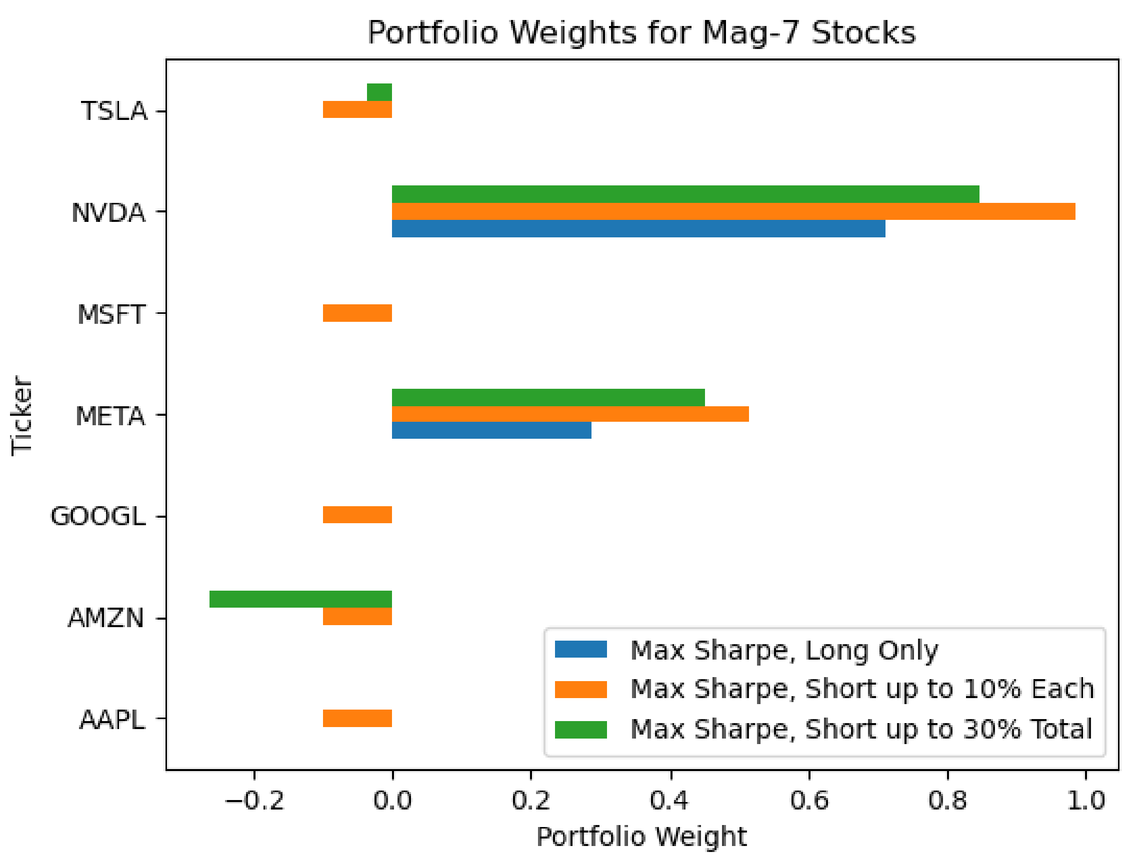
```
toy[toy < 0].sum()
```

```
np.int64(-3)
```

```
res_3 = sco.minimize(
    fun=Sharpe_neg,
    x0=equal_weights(Sigma_excess.shape[1]),
    args=(Sigma_excess, mu_excess),
    bounds=[(-0.3, 1.3) for _ in range(7)],
    constraints=(
        {'type': 'eq', 'fun': lambda x: x.sum() - 1},
        {'type': 'ineq', 'fun': lambda x: x[x < 0].sum() + 0.3}
    )
)
```

```
res_df['Max Sharpe, Short up to 30% Total'] = res_3['x']
```

```
res_df.plot(kind='barh')
plt.title('Portfolio Weights for Mag-7 Stocks')
plt.xlabel('Portfolio Weight')
plt.show()
```



Find the maximum Sharpe Ratio portfolio of Mag 7 stocks over the last three years, but do not allow any weight to exceed 30% in magnitude

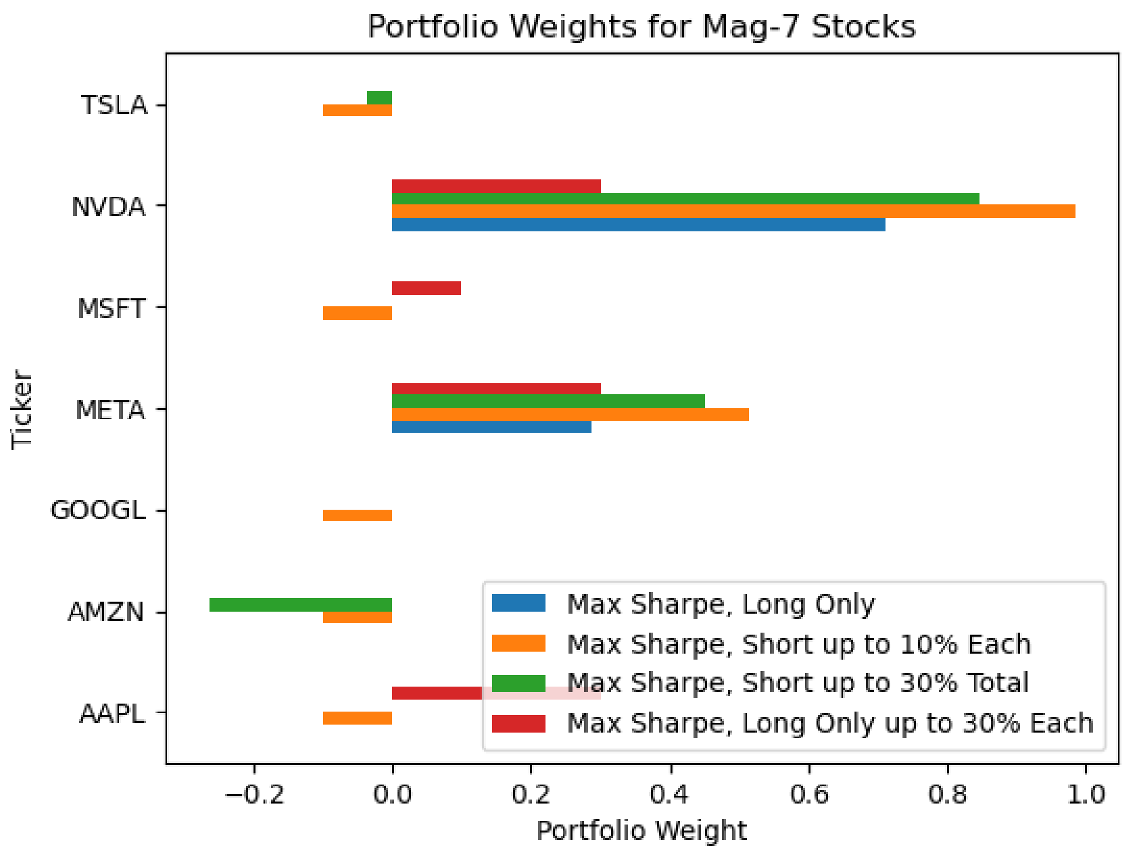
We can bound all portfolios weights on $[0, 0.3]$ with:

```
bounds=[(0, 0.3) for _ in range(7)]
```

```
res_4 = sco.minimize(
    fun=Sharpe_neg,
    x0=equal_weights(Sigma_excess.shape[1]),
    args=(Sigma_excess, mu_excess),
    bounds=[(0, 0.3) for _ in range(7)],
    constraints=(
        {'type': 'eq', 'fun': lambda x: x.sum() - 1}
    )
)
```

```
res_df['Max Sharpe, Long Only up to 30% Each'] = res_4['x']
```

```
res_df.plot(kind='barh')
plt.title('Portfolio Weights for Mag-7 Stocks')
plt.xlabel('Portfolio Weight')
plt.show()
```



Find the minimum 95% Value at Risk (Var) portfolio of Mag 7 stocks over the last three years

More on VaR here.

```python
def VaR(weights, returns, percent):
    return returns.dot(weights).quantile(1 - percent)
```

```python
VaR(
    weights=equal_weights(returns.shape[1]),
    returns=returns,
    percent=0.95
)
```

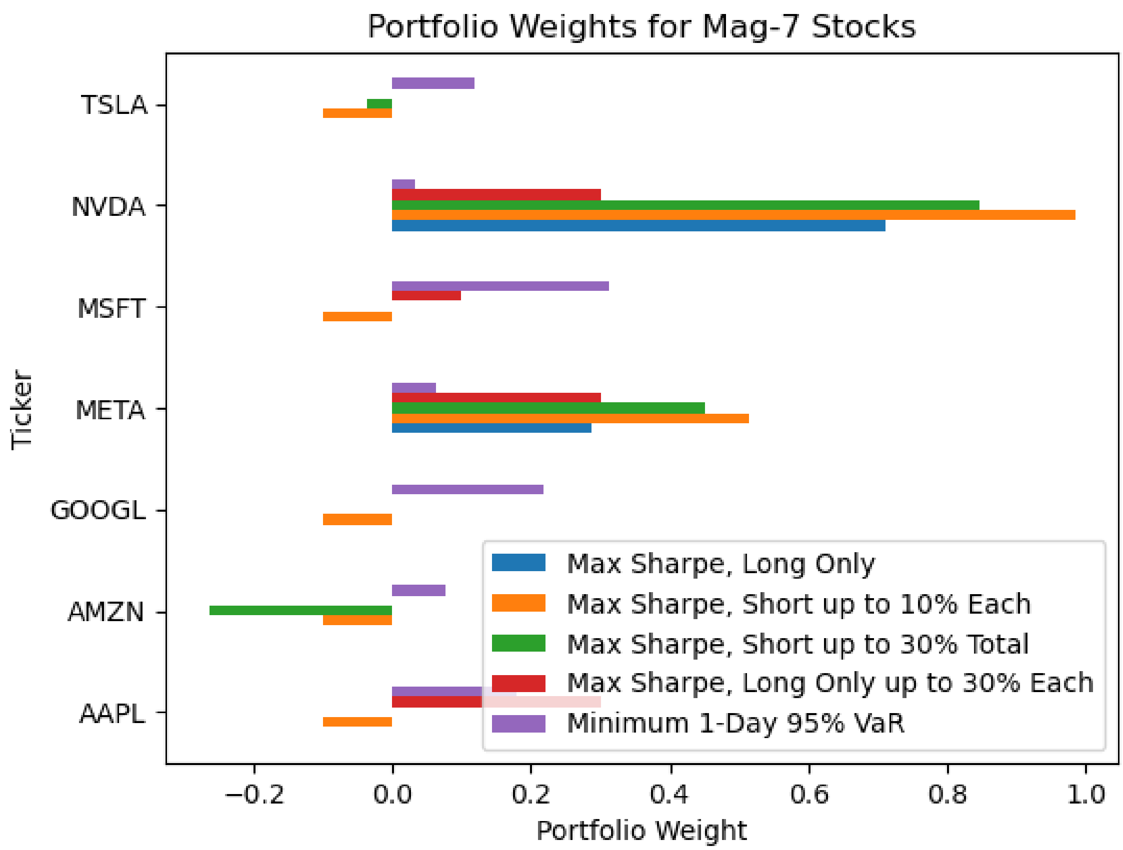
```
-0.0336
```

```
def VaR_neg(weights, returns, percent):
    return -1 * VaR(weights=weights, returns=returns, percent=percent)
```

```
res_var = sco.minimize(
    fun=VaR_neg,
    x0=equal_weights(returns.shape[1]), # np.ones(7) / 7
    args=(returns, 0.95),
    bounds=[(0, 1) for _ in range(returns.shape[1])],
    constraints=(
        {'type': 'eq', 'fun': lambda w: w.sum() - 1}
    )
)
```

```
res_df['Minimum 1-Day 95% VaR'] = res_var['x']
```

```
res_df.plot(kind='barh')
plt.title('Portfolio Weights for Mag-7 Stocks')
plt.xlabel('Portfolio Weight')
plt.show()
```



Find the minimum draw down portfolio of Mag 7 stocks over the last three years

```
def Max_Drawdown(w, returns):
    price = returns.dot(w).add(1).cumprod()
    return (price / price.cummax() - 1).min()
```

```
Max_Drawdown(equal_weights(7), returns)
```

```
-0.4524
```

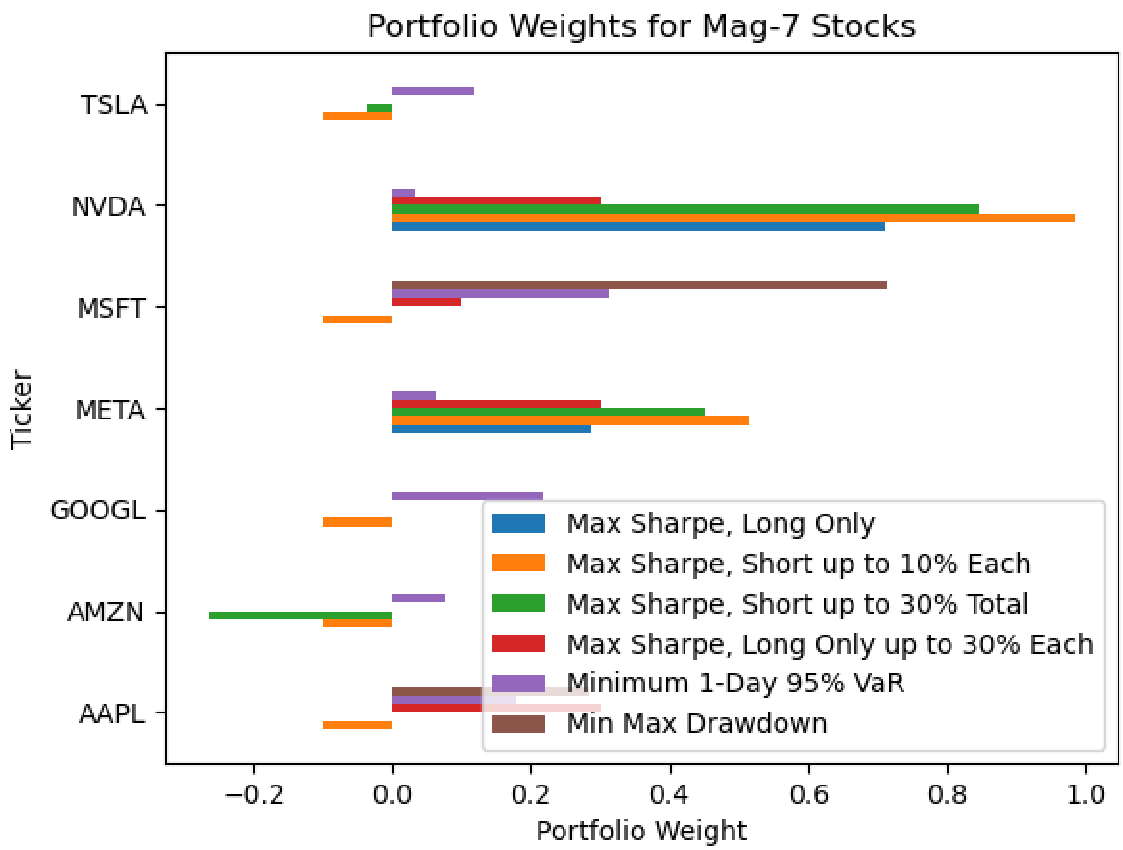
The maximum drawdown is a negative number, but we want the *smallest* negative number, so we minimize negative one times the maximum drawdown.

```python
def Max_Drawdown_neg(w, returns):
    return -1 * Max_Drawdown(w=w, returns=returns)
```

```python
res_mdd = sco.minimize(
    fun=Max_Drawdown_neg,
    x0=equal_weights(returns.shape[1]), # np.ones(7) / 7
    args=(returns,),
    bounds=[(0, 1) for _ in range(returns.shape[1])],
    constraints=(
        {'type': 'eq', 'fun': lambda w: w.sum() - 1}
    )
)
```

```python
res_df['Min Max Drawdown'] = res_mdd['x']
```

```python
res_df.plot(kind='barh')
plt.title('Portfolio Weights for Mag-7 Stocks')
plt.xlabel('Portfolio Weight')
plt.show()
```



Find the minimum draw down portfolio for the sample with complete data for the current Dow-Jones Industrial Average (DJIA) stocks

You can find the DJIA tickers on Wikipedia.

```
tickers = (
    pd.read_html(io='https://en.wikipedia.org/wiki/Dow_Jones_Industrial_Average')
    [2]
    ['Symbol']
    .to_list()
)
```

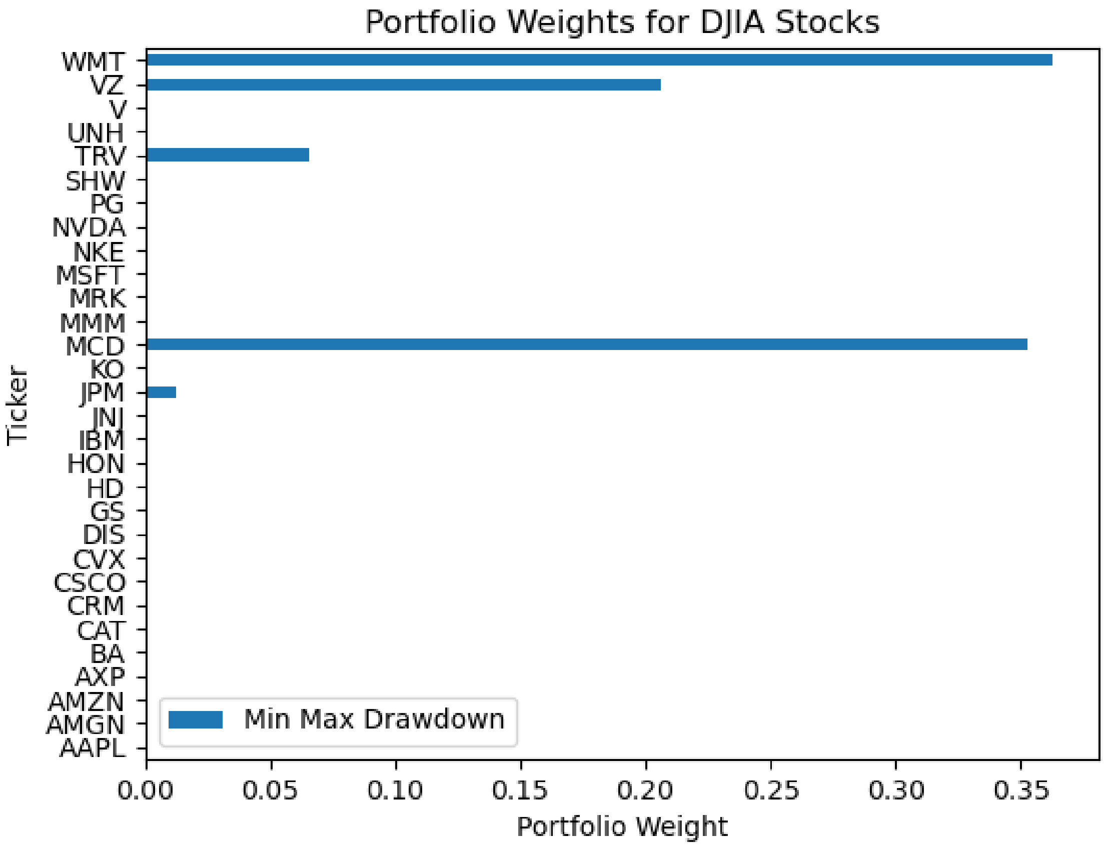
```
djia = (
    yf.download(
        tickers=tickers,
        auto_adjust=False,
```

```
        progress=False
    )
    .iloc[:-1]
    ['Adj Close']
    .pct_change()
    .dropna()
)
```

```
res_mdd = sco.minimize(
    fun=Max_Drawdown_neg,
    x0=equal_weights(djia.shape[1]),
    args=(djia,),
    bounds=[(0, 1) for _ in range(djia.shape[1])],
    constraints=(
        {'type': 'eq', 'fun': lambda w: w.sum() - 1}
    )
)
```

```
res_djia = pd.DataFrame(
    data={'Min Max Drawdown': res_mdd['x']},
    index=djia.columns
)
```

```
res_djia.plot(kind='barh')
plt.title('Portfolio Weights for DJIA Stocks')
plt.xlabel('Portfolio Weight')
plt.show()
```



Plot the minimum-variance frontier for the sample with complete data for the current the DJIA stocks

See the lecture notebook for a similar exercise with the Mag-7 stocks.

```
def calc_sigmap(w, Sigma, ppy=252):
    return np.sqrt(ppy * w.T @ Sigma @ w)
```

```
def calc_mup(w, mu, ppy=252):
    return ppy * w.T @ mu
```

```
Sigma = djia.cov()
```

```
mu = djia.mean()
```

```
djia_target = 252 * np.linspace(
    start=mu.min(),
    stop=mu.max(),
    num=50
)
```

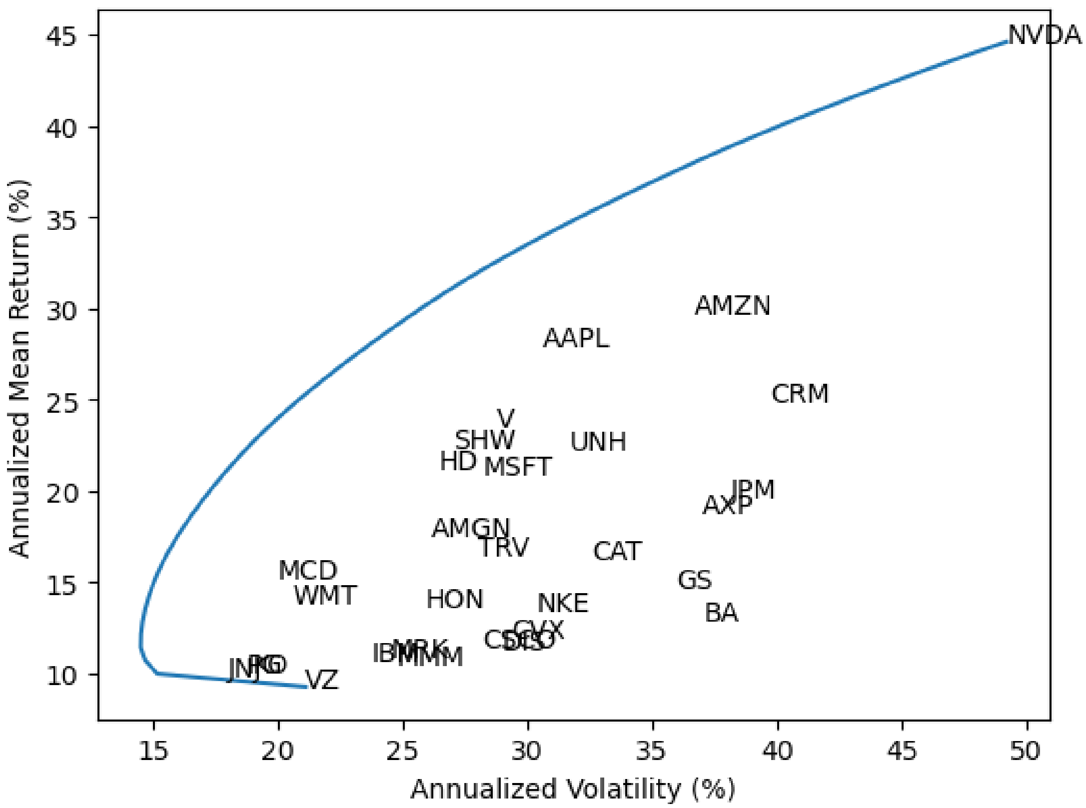
```
res_ef = []

for r in djia_target:
    _ = sco.minimize(
        fun=calc_sigmap,
        x0=np.ones(Sigma.shape[1]) / Sigma.shape[1],
        args=(Sigma, 252),
        bounds=[(0, 1) for c in Sigma.index],
        constraints=(
            {'type': 'eq', 'fun': lambda x: x.sum() - 1},
            {'type': 'eq', 'fun': lambda x: calc_mup(w=x, mu=mu, ppy=252) - r}
        )
    )
    res_ef.append(_)
```

```
for r in res_ef:
    assert r['success']
```

```
mv_frontier = pd.DataFrame({
    'returns_target': djia_target,
    'volatility': np.array([r['fun'] if r['success'] else np.nan for r in res_ef])
})
```

```
(
    mv_frontier
    .mul(100)
    .plot(x='volatility', y='returns_target', legend=False)
)
plt.ylabel('Annualized Mean Return (%)')
plt.xlabel('Annualized Volatility (%)')

for t in Sigma.index:
    x = 100 * np.sqrt(252 * Sigma.loc[t, t])
    y = 100 * 252 * mu.loc[t]
    plt.annotate(text=t, xy=(x, y))
```

```python
plt.show()
```



Find the maximum Sharpe Ratio portfolio for the sample with complete data for the current the DJIA stocks excluding the last three years, so we can compare to the 1/n portfolio to the maximum Sharpe ratio portfolio

This exercise is less dramatic withe NVDA this year. We will see three things:

1. The maximum Sharpe ratio portfolio has a high Sharpe ratio *in sample* (or what we called `_before` in class)
2. But this same portfolio has a much lower Sharpe ratio *out of sample* (or what we called `_after` in class)
3. The 1/N or equal-weighted portfolio has about the same Sharpe ratio as the maximum Sharpe ratio portfolio *out of sample*

```python
djia_excess = djia.sub(ff3['RF'], axis=0)
Sigma_excess_before = djia_excess.iloc[:-756].cov()
mu_excess_before = djia_excess.iloc[:-756].mean()
```

```python
Sigma_excess_after = djia_excess.iloc[-756:].cov()
mu_excess_after = djia_excess.iloc[-756:].mean()
```

```python
res_before = sco.minimize(
    fun=Sharpe_neg,
    x0=equal_weights(Sigma_excess_before.shape[1]),
    args=(Sigma_excess_before, mu_excess_before),
    bounds=[(0, 1) for _ in range(Sigma_excess_before.shape[1])],
    constraints=(
        {'type': 'eq', 'fun': lambda x: x.sum() - 1}
    )
)
```

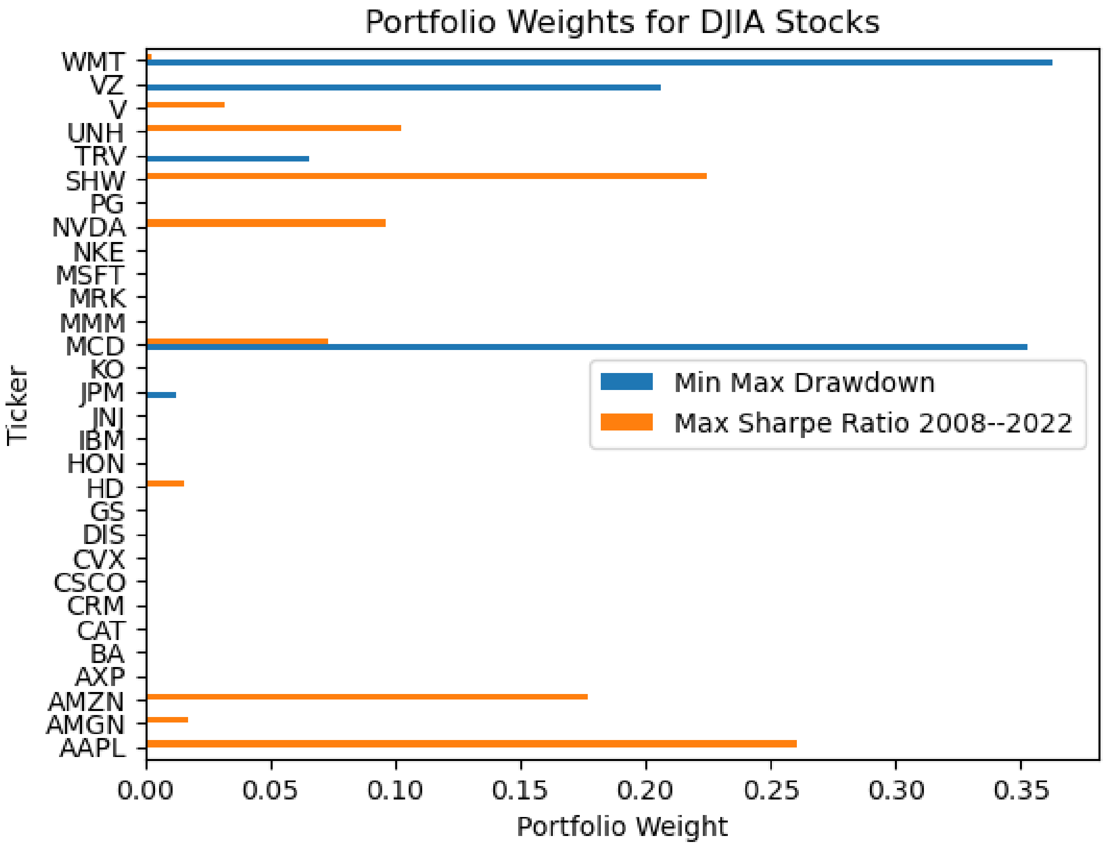
The maximum Sharpe ratio portfolio has a high Sharpe ratio *in sample*!

```python
Sharpe(
    w=res_before['x'],
    Sigma_excess=Sigma_excess_after,
    mu_excess=mu_excess_after,
    ppy=252
)
```

```
0.7175
```

```python
res_djia['Max Sharpe Ratio 2008--2022'] = res_before['x']
```

```python
res_djia.plot(kind='barh')
plt.title('Portfolio Weights for DJIA Stocks')
plt.xlabel('Portfolio Weight')
plt.show()
```



The maximum Sharpe ratio portfolio has a much lower Sharpe ratio *out of sample*!

```
Sharpe(
    w=res_before['x'],
    Sigma_excess=Sigma_excess_after,
    mu_excess=mu_excess_after,
    ppy=252
)
```

```
0.7175
```

Furthermore, the 1/N or equal-weighted portfolio does about as well *out of sample*!

```
Sharpe(
    w=equal_weights(Sigma_excess_after.shape[1]),
    Sigma_excess=Sigma_excess_after,
    mu_excess=mu_excess_after,
```

```
    ppy=252
)
```

```
0.6134
```

These out of sample drop in performance become more extreme if we remove the NVDA and AAPL rocketships.

```
Sigma_excess_before_wo = djia_excess.drop(columns=['AAPL', 'NVDA']).iloc[:-756].cov()
mu_excess_before_wo = djia_excess.drop(columns=['AAPL', 'NVDA']).iloc[:-756].mean()
```

```
Sigma_excess_after_wo = djia_excess.drop(columns=['AAPL', 'NVDA']).iloc[-756:].cov()
mu_excess_after_wo = djia_excess.drop(columns=['AAPL', 'NVDA']).iloc[-756:].mean()
```

```
res_before_wo = sco.minimize(
    fun=Sharpe_neg,
    x0=equal_weights(Sigma_excess_before_wo.shape[0]),
    args=(Sigma_excess_before_wo, mu_excess_before_wo),
    bounds=[(0, 1) for _ in range(Sigma_excess_before_wo.shape[0])],
    constraints=(
        {'type': 'eq', 'fun': lambda w: w.sum() - 1}
    )
)
```

In sample Sharpe ratio of maximum Sharpe Ratio portfolio:

```
Sharpe(
    w=res_before_wo['x'],
    Sigma_excess=Sigma_excess_before_wo,
    mu_excess=mu_excess_before_wo
)
```

```
1.1618
```

Out of sample Sharpe ratio of maximum Sharpe Ratio portfolio:

```
Sharpe(
    w=res_before_wo['x'],
    Sigma_excess=Sigma_excess_after_wo,
    mu_excess=mu_excess_after_wo
)
```

```
0.4699
```

Out of sample Sharpe ratio of of 1/N or equal-weighted portfolio:

```
Sharpe(
    w=equal_weights(Sigma_excess_after_wo.shape[1]),
    Sigma_excess=Sigma_excess_after_wo,
    mu_excess=mu_excess_after_wo
)
```

```
0.4891
```

Week 13

Herron Topic 5 - Simulations

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import pandas_datareader as pdr
from scipy.stats import norm # new addition for Monte Carlo methods
import yfinance as yf
```

```
%precision 2
pd.options.display.float_format = '{:.2f}'.format
# %config InlineBackend.figure_format = 'retina'
```

In finance, we use two main simulation methods: Monte Carlo and bootstrap. Both help analyze financial data, but work differently.

Monte Carlo methods create many possible paths of financial variables (e.g., stock prices, interest rates) based on theoretical probability distributions:

- Generates synthetic data from assumed models
- Creates data that may not exist in historical records
- Based on random sampling from probability distributions

Bootstrap methods take a different approach:

- Resamples from actual historical data (with or without replacement)
- Makes minimal assumptions about distributions
- Maintains statistical properties of original data

Check Monte Carlo methods, Bootstrapping (statistics), and Monte Carlo methods in finance for details.

In this lecture notebook, we focus on: 1. *Option pricing:* Monte Carlo methods can estimate derivative values by simulating future asset paths and calculating discounted expected payoffs. 2. *Value at Risk (VaR):* Both methods help compute this risk metric to estimate potential portfolio losses at specific confidence levels.

Specifically, we will simulate AAPL price paths and price a call option, then compute VaR for a portfolio of GOOG and META.

Estimating Option Prices with Monte Carlo Methods

We can use Monte Carlo methods to value stock options. First, we simulate several thousand random price paths for the underlying stock. Then, we calculate the payoff for each path. Along some paths, the option will expire "in the money" with $S_T > K$ and pay $S_T - K$. Along other paths, the option will expire "out of the money" with $S_T < K$ and pay 0. We average these payoffs and discount them to today. The present value of this expected payoffs is the option price. This is a simple example, and there is a lot of depth to Monte Carlo methods for option pricing.

Simulating Stock Prices

We can simulate stock prices with the following stochastic differential equation (SDE) for Geometric Brownian Motion (GBM):

$$dS = \mu S dt + \sigma S dW_t.$$

GBM does not account for mean-reversion and time-varying volatility. So GBM is often used for stocks and not for bond prices, which tend to revert to face value in the long-term. In the SDE for GBM:

1. S is the stock price
2. μ is the drift coefficient (i.e., instantaneous expected return)
3. σ is the diffusion coefficient (i.e., volatility of the drift)
4. W_t is the Wiener Process or Brownian Motion

The GBM SDE has a closed-form solution:

$$S(t) = S_0 \exp\left(\left(\mu - \frac{1}{2}\sigma^2\right) t + \sigma W_t\right).$$

We can iteratively apply this closed form solution:

$$S(t_{i+1}) = S(t_i) \exp\left(\left(\mu - \frac{1}{2}\sigma^2\right)(t_{i+1} - t_i) + \sigma \sqrt{t_{i+1} - t_i} Z_{i+1}\right).$$

Here, Z_i is a Standard Normal random variable (becuase dW_t are independent and normally distributed) and $i = 0, \dots, T - 1$ is the time index.

We can use this closed form solution to simulate stock prices for AAPL.

```
aapl = (
    yf.download(
        tickers='AAPL',
        auto_adjust=False,
```

```
        progress=False,
        multi_level_index=False
    )
    .iloc[:-1]
    .assign(Return=lambda x: x['Adj Close'].pct_change())
)
```

We can use returns from 2021 to predict price paths in 2022 (i.e., *train* in 2021, and *test* in 2022).

```
train = aapl.loc['2021']
test = aapl.loc['2022']
```

We can use the following `simulate_gbm()` function to simulate one price path. Throughout this lecture notebook, we will use one-trading-day steps (i.e., `dt=1`).

```
def simulate_gbm(S_0, mu, sigma, n_steps, dt=1, seed=42):
    '''
    Function to simulate stock prices following Geometric Brownian Motion (GBM).

    Parameters
    ------------
    S_0 : float
        Initial stock price
    mu : float
        Drift coefficient
    sigma : float
        Diffusion coefficient
    n_steps : int
        Length of the forecast horizon in time increments, so T = n_steps * dt
    dt : int
        Time increment, typically one day
    seed : int
        Random seed for reproducibility

    Returns
    -----------
    S_t : np.ndarray
        Array (length: n_steps + 1) of simulated prices
    '''
```

```
    np.random.seed(seed)
    dW = np.random.normal(scale=np.sqrt(dt), size=n_steps)
    W = dW.cumsum()

    t = np.linspace(dt, n_steps * dt, n_steps)

    S_t = S_0 * np.exp((mu - 0.5 * sigma**2) * t + sigma * W)
    S_t = np.insert(S_t, 0, S_0)

    return S_t
```

Here is one simulated price path:

```
S_0 = train['Adj Close'].iloc[-1]
mu = train['Return'].pipe(np.log1p).mean()
sigma = train['Return'].pipe(np.log1p).std()
n_steps = test.shape[0]

simulate_gbm(S_0=S_0, mu=mu, sigma=sigma, n_steps=n_steps)
```

```
array([174.52, 176.08, 175.88, 177.88, 182.4 , 181.92, 181.44, 186.22,
       188.69, 187.5 , 189.31, 188.13, 186.95, 187.86, 182.46, 177.74,
       176.35, 173.73, 174.78, 172.47, 168.84, 172.98, 172.55, 172.92,
       169.24, 167.97, 168.44, 165.58, 166.74, 165.34, 164.76, 163.37,
       168.4 , 168.54, 165.92, 168.27, 165.23, 165.95, 161.06, 157.88,
       158.54, 160.57, 161.18, 161.05, 160.46, 156.92, 155.3 , 154.34,
       157.11, 158.13, 153.95, 154.9 , 154.12, 152.64, 154.29, 156.99,
       159.49, 157.55, 156.95, 157.94, 160.56, 159.52, 159.22, 156.62,
       153.85, 156.01, 159.56, 159.54, 162.26, 163.37, 161.88, 162.98,
       167.17, 167.25, 171.62, 164.83, 167.16, 167.57, 166.95, 167.37,
       162.37, 161.98, 163.07, 167.1 , 165.91, 163.97, 162.85, 165.4 ,
       166.44, 165.22, 166.74, 167.18, 169.94, 168.24, 167.54, 166.69,
       163.05, 163.98, 164.84, 165.02, 164.59, 161.11, 160.22, 159.52,
       157.68, 157.44, 158.62, 163.59, 164.21, 165.06, 165.04, 160.28,
       160.38, 160.7 , 167.26, 166.93, 167.9 , 167.99, 165.09, 168.27,
       170.46, 172.79, 170.51, 174.51, 170.87, 172.64, 178.91, 176.32,
       174.93, 175.39, 174.19, 170.15, 170.51, 167.85, 169.29, 167.03,
       171.35, 169.42, 168.74, 171.1 , 167.98, 168.77, 172.47, 168.32,
       168.99, 169.87, 172.16, 169.  , 165.69, 167.24, 168.2 , 169.05,
       170.16, 168.51, 169.31, 170.28, 168.54, 173.77, 175.26, 172.18,
       174.16, 171.67, 174.01, 177.41, 175.31, 178.19, 179.54, 182.08,
```

```
187.82, 187.29, 185.27, 182.88, 180.72, 180.69, 181.86, 182.85,
185.46, 185.69, 190.21, 189.61, 198.15, 200.33, 197.85, 194.73,
196.43, 195.94, 198.38, 200.08, 200.06, 197.61, 193.13, 191.98,
194.8 , 195.67, 192.05, 192.78, 194.17, 191.67, 192.34, 192.72,
189.47, 190.75, 192.65, 196.18, 199.69, 195.6 , 192.92, 194.7 ,
196.5 , 198.31, 210.99, 213.13, 217.22, 220.75, 223.27, 222.39,
225.32, 222.81, 222.22, 220.75, 221.27, 229.76, 223.31, 225.99,
220.53, 219.12, 223.16, 223.63, 220.08, 217.84, 220.42, 218.12,
219.1 , 219.49, 217.47, 225.2 , 227.71, 220.77, 221.66, 219.58,
222.79, 220.25, 220.09, 222.08, 225.38, 221.38, 220.45, 219.03,
217.01, 223.38, 225.06, 220.85])
```

We can combine `simulate_gbm()` with a list comprehension and `pd.concat()` to simulate and collect many price paths.

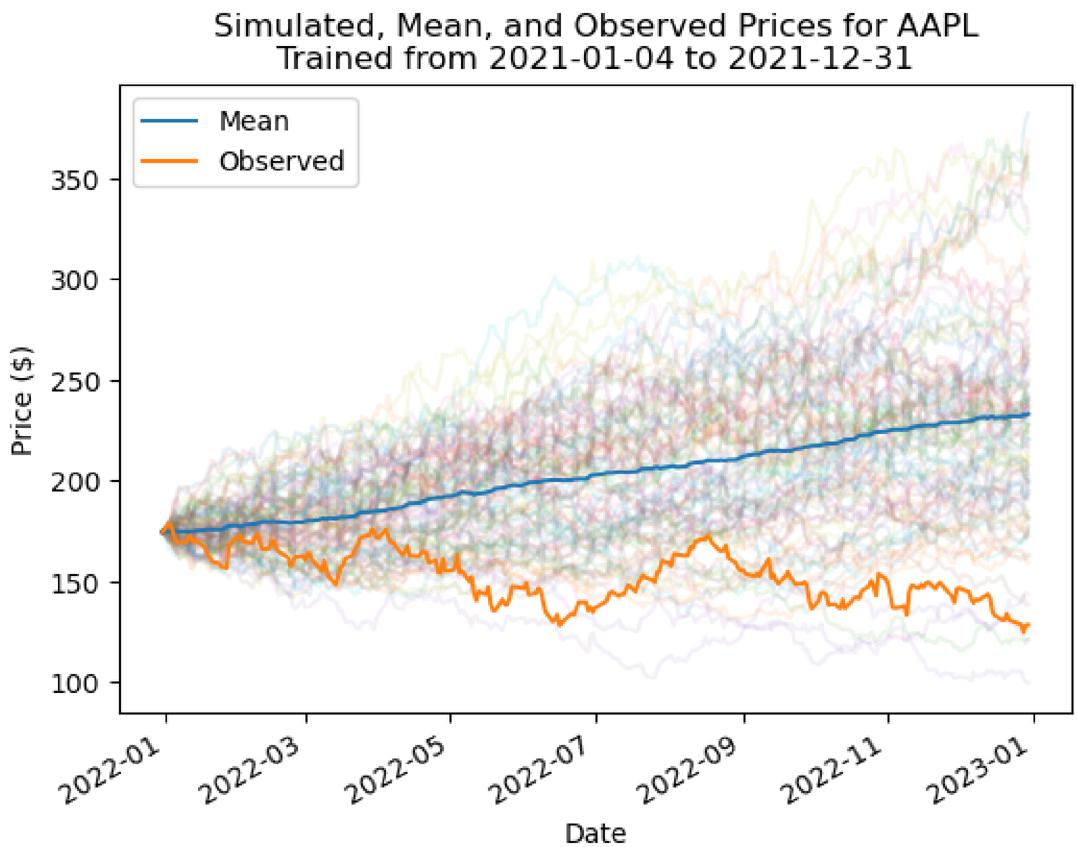
```
n = 100
S_t = pd.concat(
    objs=[
        pd.Series(
            data=simulate_gbm(S_0=S_0, mu=mu, sigma=sigma, n_steps=n_steps, seed=seed),
            index=test.index.insert(0, train.index[-1])
        )
        for seed in range(n)
    ],
    axis=1,
    keys=range(n),
    names=['Simulation']
)
```

```
S_t.iloc[:5, :5]
```

Simulation Date	0	1	2	3	4
2021-12-31	174.52	174.52	174.52	174.52	174.52
2022-01-03	179.64	179.24	173.55	179.71	174.84
2022-01-04	180.97	177.71	173.58	181.15	176.41
2022-01-05	183.99	176.42	168.00	181.61	173.84
2022-01-06	190.82	173.63	172.59	176.53	175.94

We can prefix the simulated price path column names with `_` to hide them from the legend.

```
fig, ax = plt.subplots(1,1)
S_t.add_prefix('_').plot(alpha=0.1, ax=ax)
S_t.mean(axis=1).plot(label='Mean', ax=ax)
aapl.loc[S_t.index, 'Adj Close'].plot(label='Observed', ax=ax)
plt.legend()
plt.ylabel('Price ($)')
plt.title(
    'Simulated, Mean, and Observed Prices for AAPL' +
    f'\nTrained from {train.index[0]:%Y-%m-%d} to {train.index[-1]:%Y-%m-%d}'
)
plt.show()
```



Simulating Option Prices

We can use these simulated price paths to price options! We can use the Black and Scholes (1973) formula as a benchmark. Black and Scholes (1973) provide an analytic or closed form solution to price European options.

```
def price_bs(S_0, K, T, r, sigma, type='call'):
    '''
    Function used for calculating the price of European options using the analytical form of

    Parameters
    ------------
    S_0 : float
        Initial stock price
    K : float
        Strike price
    T : float
        Time to expiration in days
    r : float
        Daily risk-free rate
    sigma : float
        Standard deviation of daily stock returns
    type : str
        Type of the option. Allowable: ['call', 'put']

    Returns
    -----------
    option_premium : float
        The premium on the option calculated using the Black-Scholes model
    '''

    d1 = (np.log(S_0 / K) + (r + 0.5 * sigma ** 2) * T) / (sigma * np.sqrt(T))
    d2 = d1 - sigma * np.sqrt(T)

    if type == 'call':
        val = (norm.cdf(d1, 0, 1) * S_0) - (norm.cdf(d2, 0, 1) * K * np.exp(-r * T))
    elif type == 'put':
        val = (norm.cdf(-d2, 0, 1) * K * np.exp(-r * T)) - (norm.cdf(-d1, 0, 1) * S_0)
    else:
        raise ValueError('Argument type must be "call" or "put"')

    return val
```

We can use the AAPL parameters above to price a European call option on AAPL stock. We will calculate its price at the end of 2021 with an end of 2022 expiration, a $100 strike price, and a 5% risk-free rate.

```
S_0 = train['Adj Close'].iloc[-1]
K = 100
T = test.shape[0]
r = 0.05/252
sigma = train['Return'].pipe(np.log1p).std()
```

```
call = price_bs(S_0=S_0, K=K, T=T, r=r, sigma=sigma)
print(f'Black and Scholes (1973) call option price: {call:0.2f}')
```

```
Black and Scholes (1973) call option price: 79.46
```

To simulate the Black and Scholes (1973) option price, we must simulate AAPL price paths with the same drift as the 5% risk-free rate. That is, we enter **mu=r** as an argument to the **simulate_gbm()** function. We can simulate 10,000 price paths to increase the precision of our call option price.

```
n = 10_000
S_t = pd.concat(
    objs=[
        pd.Series(
            data=simulate_gbm(S_0=S_0, mu=r, sigma=sigma, n_steps=n_steps, seed=seed),
            index=test.index.insert(0, train.index[-1])
        )
        for seed in range(n)
    ],
    axis=1,
    keys=range(n),
    names=['Simulation']
)
```

```
S_t.iloc[:5, :5]
```

Simulation Date	0	1	2	3	4
2021-12-31	174.52	174.52	174.52	174.52	174.52
2022-01-03	179.46	179.07	173.38	179.53	174.67
2022-01-04	180.62	177.36	173.24	180.79	176.07
2022-01-05	183.45	175.90	167.50	181.08	173.33
2022-01-06	190.07	172.95	171.91	175.84	175.25

The payoff of the call option is $\max(S_T - K, 0)$. The price of the call option is the present value of its mean payoff, discounted at the risk-free rate.

```
payoff = np.maximum(S_t.iloc[-1] - K, 0)
print(f'Simulated option price: {payoff.mean() * np.exp(-r * T):0.2f}')
```

```
Simulated option price: 80.16
```

The simulated call option price does not exactly match the analytical solution. We could simulate more price paths to drive the simulated call option price closer to the analytical solution.

Estimating Value-at-Risk using Monte Carlo

Value-at-Risk (VaR) estimates the risk associated with a portfolio. VaR reports the worst expected loss, at a given level of confidence, over a given horizon, under normal market conditions. For example, say the 1-day 95% VaR of our portfolio is 100 dollars. This implies that that 95% of the time, under normal market conditions, we should not lose more than 100 dollars over 1 day. We typically present VaR as a positive value, so a VaR of 100 dollars implies a loss of *less than 100 dollars*.

We can calculate VaR several ways, including:

- Parametric Approach (covariance)
- Bootstrap simulations
- Monte Carlo simulations

Here we use a Monte Carlo simulation to calculate the 1-day 95% VaR of an portfolio of 20 shares each of META and GOOG.

```
tickers = ['GOOG', 'META']
shares = np.array([20, 20])
T = 1
n = 10_000
```

We can download all data from Yahoo! Finance and subset our data later.

```
df = (
    yf.download(
        tickers=tickers,
        auto_adjust=False,
        progress=False
    )
    .iloc[:-1]
)
```

Next, we calculate daily returns during 2022. Choosing the window to define "normal market conditions" is part art, part science, and beyod the scope of this lecture notebook.

```
returns = df['Adj Close'].pct_change().loc['2022']
```

Next, we calculate the covariance matrix.

```
cov_mat = returns.cov()
```

Covariances and variances on daily decimal returns are small, so we display `cov_mat` in "percent squared".

```
cov_mat * 100**2
```

Ticker Ticker	GOOG	META
GOOG	5.96	6.74
META	6.74	16.38

Next, we use the covariance matrix to calculate the Cholesky decomposition. The Cholesky decomposition helps us generate random variables with the same covariances as the observed data.

```
chol_mat = np.linalg.cholesky(cov_mat)
```

```
chol_mat
```

```
array([[0.02, 0.  ],
       [0.03, 0.03]])
```

```
rv = np.random.normal(size=(n, len(tickers)))
```

```
correlated_rv = (chol_mat @ rv.T).T
```

```
correlated_rv
```

```
array([[ 0.04,  0.05],
       [ 0.02,  0.02],
       [-0.02, -0.04],
       ...,
       [-0.04,  0.01],
       [-0.  , -0.09],
       [-0.02,  0.  ]], shape=(10000, 2))
```

These random variables have a covariance matrix similar to the real data.

```
np.cov(correlated_rv.T) * 100**2
```

```
array([[ 5.97,  6.85],
       [ 6.85, 16.47]])
```

Here are the parameters for the simulated price paths:

```
mu = returns.mean().values
sigma = returns.std().values
S_0 = df.loc['2021', 'Adj Close'].iloc[-1].values
P_0 = S_0.dot(shares)
```

Calculate terminal prices using the GBM formula above:

```
S_T = S_0 * np.exp((r - 0.5 * sigma ** 2) * T + sigma * np.sqrt(T) * correlated_rv)
```

```
S_T
```

```
array([[144.11, 335.24],
       [144.06, 334.8 ],
       [143.93, 334.05],
       ...,
       [143.85, 334.72],
       [143.97, 333.38],
       [143.92, 334.62]], shape=(10000, 2))
```

Calculate terminal portfolio values and returns. Note that these are dollar values, since VaR is typically expressed in dollar values.

```
V_T = S_T.dot(shares)
```

```
V_T
```

```
array([9586.98, 9577.31, 9559.53, ..., 9571.3 , 9546.99, 9570.8 ],
      shape=(10000,))
```

```
V_diff = V_T - P_0
```

```
V_diff
```

```
array([ 11.59,   1.91, -15.86, ...,  -4.09, -28.4 ,  -4.59],
      shape=(10000,))
```

Next, we calculate VaR.

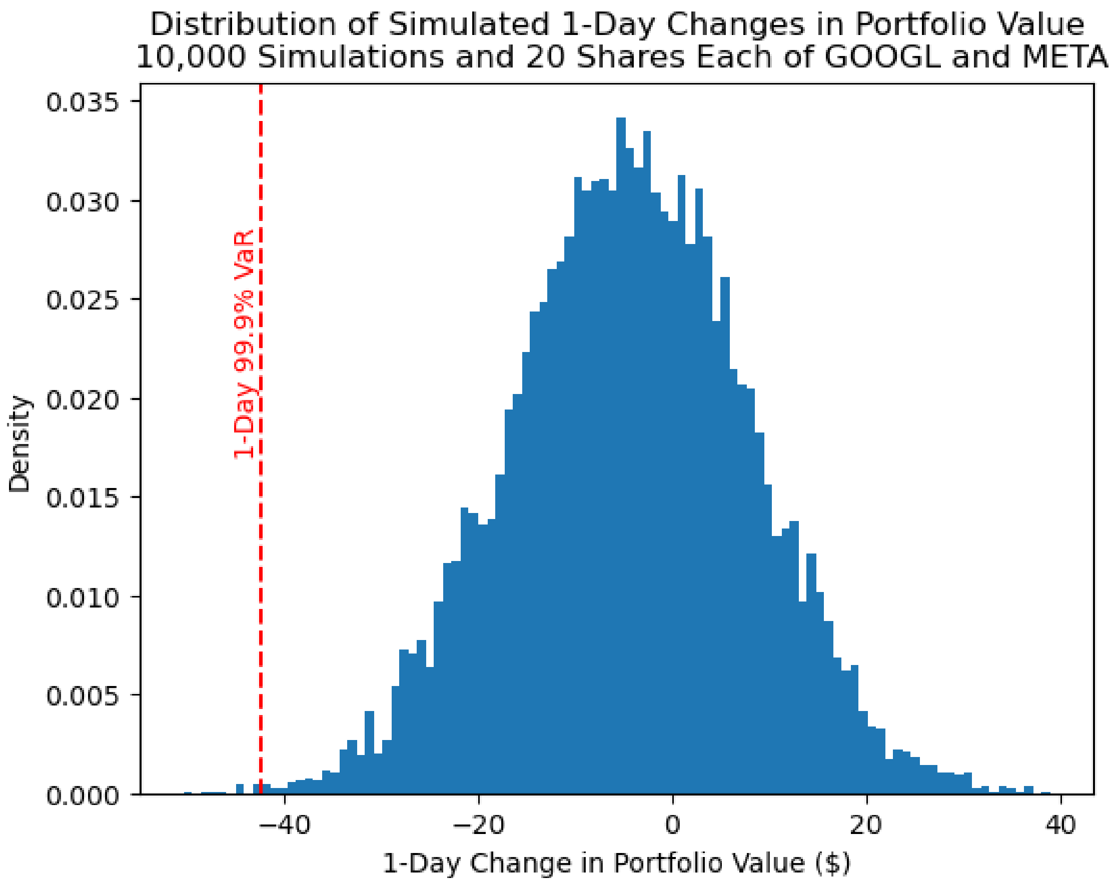
```
percentiles = [5, 1, 0.1]
var = np.percentile(V_diff, percentiles)

for x, y in zip(percentiles, var):
    print(f'1-day {100-x:0.1f}% VaR: ${-y:.2f}')
```

```
1-day 95.0% VaR: $24.53
1-day 99.0% VaR: $32.75
1-day 99.9% VaR: $42.43
```

Finally, we plot VaR:

```
fig, ax = plt.subplots()
ax.hist(V_diff, bins=100, density=True)
ax.axvline(x=var[2], color='red', ls='--')
ax.text(
    x=var[2],
    y=0.8,
    s=f'1-Day {100-percentiles[2]:0.1f}% VaR',
    color='red',
```

```
    ha='right',
    va='top',
    rotation=90,
    transform=ax.get_xaxis_transform()
)
ax.set_xlabel('1-Day Change in Portfolio Value ($)')
ax.set_ylabel('Density')
ax.set_title(f'Distribution of Simulated 1-Day Changes in Portfolio Value\n {n:,.0f} Simulati
plt.show()
```



Herron Topic 5 - Practice - Sec 02

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import pandas_datareader as pdr
import seaborn as sns
from scipy.stats import norm # new addition for Monte Carlo methods
import warnings # to suppress the pandas_datareader warning
import yfinance as yf
```

```
%precision 2
pd.options.display.float_format = '{:.2f}'.format
# %config InlineBackend.figure_format = 'retina'
```

Announcements

1. Programming assessment and MSFQ assessment will be *in class and in person on Tuesday, 4/15, in your scheduled section*
    1. Details here: https://northeastern.instructure.com/courses/207607/discussion_topics/2765433
    2. And here: https://northeastern.instructure.com/courses/207607/discussion_topics/2727917
2. Our plan for the rest of the semester
    1. Friday, 4/11, Project 2 solution, envelopes solution, and Project 3 group work
    2. Tuesday, 4/15, programming and MSFQ assessment *in class and in person, in your scheduled section*
    3. Friday, 4/18, Project 3 group work
    4. Tuesday, 4/22, Project 3 group work
    5. Wednesday, 4/23, Project 3 is due by 11:59 PM
    6. Friday, 4/25, office hours during regular class time
3. Please complete TRACE! I value and use your feedback.

1. Details here: https://northeastern.instructure.com/courses/207607/discussion_topics/2753925

Five-Minute Recap

In finance, we use two main simulation methods: Monte Carlo and bootstrap. Both help analyze financial data, but work differently.

Monte Carlo methods create many possible paths of financial variables (e.g., stock prices, interest rates) based on theoretical probability distributions:

- Generates synthetic data from assumed models
- Creates data that may not exist in historical records
- Based on random sampling from probability distributions
- Think `np.random.uniform()` or `np.random.normal()` to create new random variables

Bootstrap methods take a different approach:

- Resamples from actual historical data (with or without replacement)
- Makes minimal assumptions about distributions
- Maintains statistical properties of original data
- Think `.sample()` to resample past obervations

Practice

Estimate π by simulating darts thrown at a dart board

Hints: Select random xs and ys such that $-r \leq x \leq +r$ and $-r \leq x \leq +r$. Darts are on the board if $x^2 + y^2 \leq r^2$. The area of the circlular board is πr^2, and the area of square around the board is $(2r)^2 = 4r^2$. The fraction f of darts on the board is the same as the ratio of circle area to square area, so $f = \frac{\pi r^2}{4r^2}$.

First we throw darts at the board. Darts with $x^2 + y^2 \leq r^2$ are on the board.

```
def throw_darts(r=1, n=1_000, seed=42):
    np.random.seed(seed)
    return (
        pd.DataFrame(
            data=np.random.uniform(low=-r, high=r, size=2*n).reshape(n, 2),
            columns=['X', 'Y']
        )
```

```
        .assign(On_Board=lambda x: x['X']**2 + x['Y']**2 <= r**2)
    )
```

```
throw_darts()
```

	X	Y	On_Board
0	-0.25	0.90	True
1	0.46	0.20	True
2	-0.69	-0.69	True
3	-0.88	0.73	False
4	0.20	0.42	True
...	...	...	...
995	0.53	-0.68	True
996	0.22	-0.73	True
997	0.50	0.31	True
998	0.91	-0.86	False
999	-0.89	-0.44	True

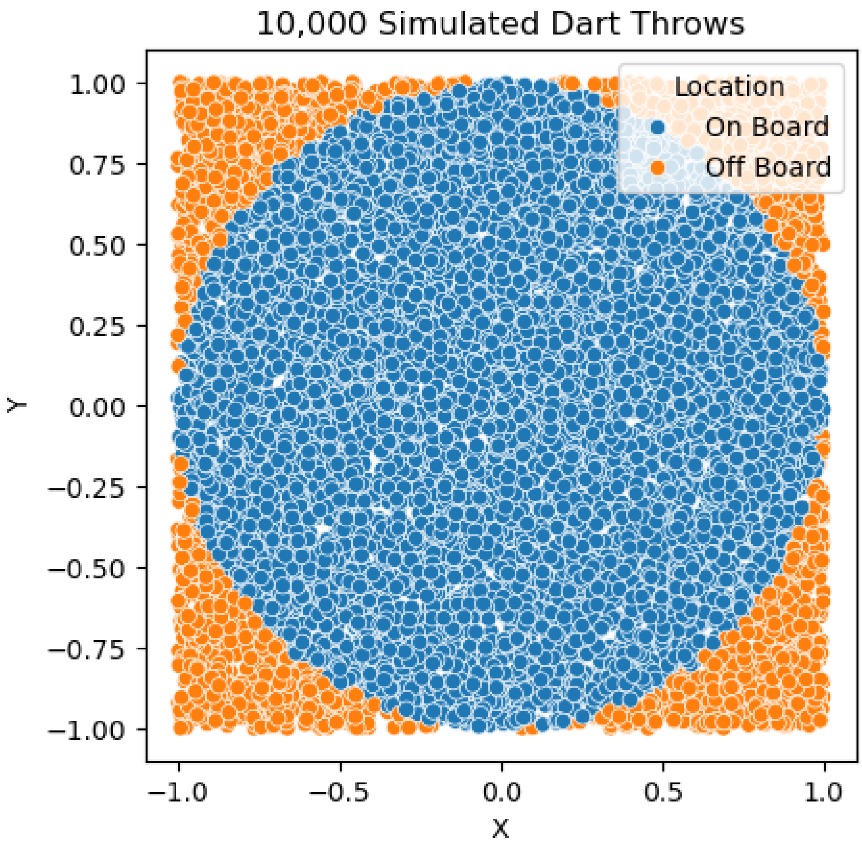
```
4 * throw_darts(n=1_000_000)['On_Board'].mean()
```

```
3.14
```

Next, we visualize these darts with a scatter plot. Seaborn's `scatterplot()` helps color darts by location (i.e., on or off board). The `.pipe()` method lets us send the output of the `.assign()` method to `sns.scatterplot()` without assigning a temporary data frame.

```
n = 10_000

(
    throw_darts(n=n)
    .assign(Location=lambda x: np.where(x['On_Board'], 'On Board', 'Off Board'))
    .pipe(
        sns.scatterplot,
        x='X',
        y='Y',
        hue='Location'
    )
)
plt.gca().set_aspect('equal')
```

```
plt.title(f'{n:,.0f} Simulated Dart Throws')
plt.show()
```



Finally, we use the hint above to estimate π. The hint above says $f = \frac{\pi r^2}{4r^2}$, where f is the fraction of darts on the board. Therefore, $\pi \approx \frac{4fr^2}{r^2} = 4f$. We increase the precision of our π estimate by increasing the number of simulated darts n.

```
for n in 10**np.arange(7):
    print(f'With {n:>9,.0f} dart throws, our estimate of pi is: {4 * throw_darts(n=n)['On_Boa
```

```
With         1 dart throws, our estimate of pi is: 4.0000
With        10 dart throws, our estimate of pi is: 3.2000
With       100 dart throws, our estimate of pi is: 3.0400
With     1,000 dart throws, our estimate of pi is: 3.1040
With    10,000 dart throws, our estimate of pi is: 3.1544
With   100,000 dart throws, our estimate of pi is: 3.1468
With 1,000,000 dart throws, our estimate of pi is: 3.1420
```

Simulate your wealth W_T by randomly sampling market returns

Use monthly market returns from the French Data Library. Only invest one cash flow W_0, and plot the distribution of W_T.

First, we download data from the French Data Library. We convert these returns from percent to decimal to simplify compounding.

```
with warnings.catch_warnings():
    warnings.filterwarnings("ignore", category=FutureWarning)
    ff3 = (
        pdr.DataReader(
            name='F-F_Research_Data_Factors',
            data_source='famafrench',
            start='1900'
        )
        [0]
        .assign(Mkt=lambda x: x['Mkt-RF'] + x['RF'])
        .div(100)
    )
```

1. Use `.sample()` to simulate one alternate history
2. Add `pd.concat()` and a list comprehension to simulate `N` alternate histories
3. Examine/plot the distributoin of `W_T`

Here is one example `.sample()`:

```
ff3[['Mkt']].sample(n=5, ignore_index=True, random_state=42)
```

	Mkt
0	0.03
1	0.03
2	0.04
3	-0.11
4	-0.02

We can combine many samples of many months with a list comprehension and `pd.concat()`.

```
W_0 = 50_000
T = 12 * 50
N = 1_000
```

```
r_t = pd.concat(
    objs=[ff3['Mkt'].sample(n=T, ignore_index=True, random_state=i) for i in range(N)],
    axis=1,
    keys=range(N),
    names='Simulation'
)
```

```
r_t.iloc[:5, :5]
```

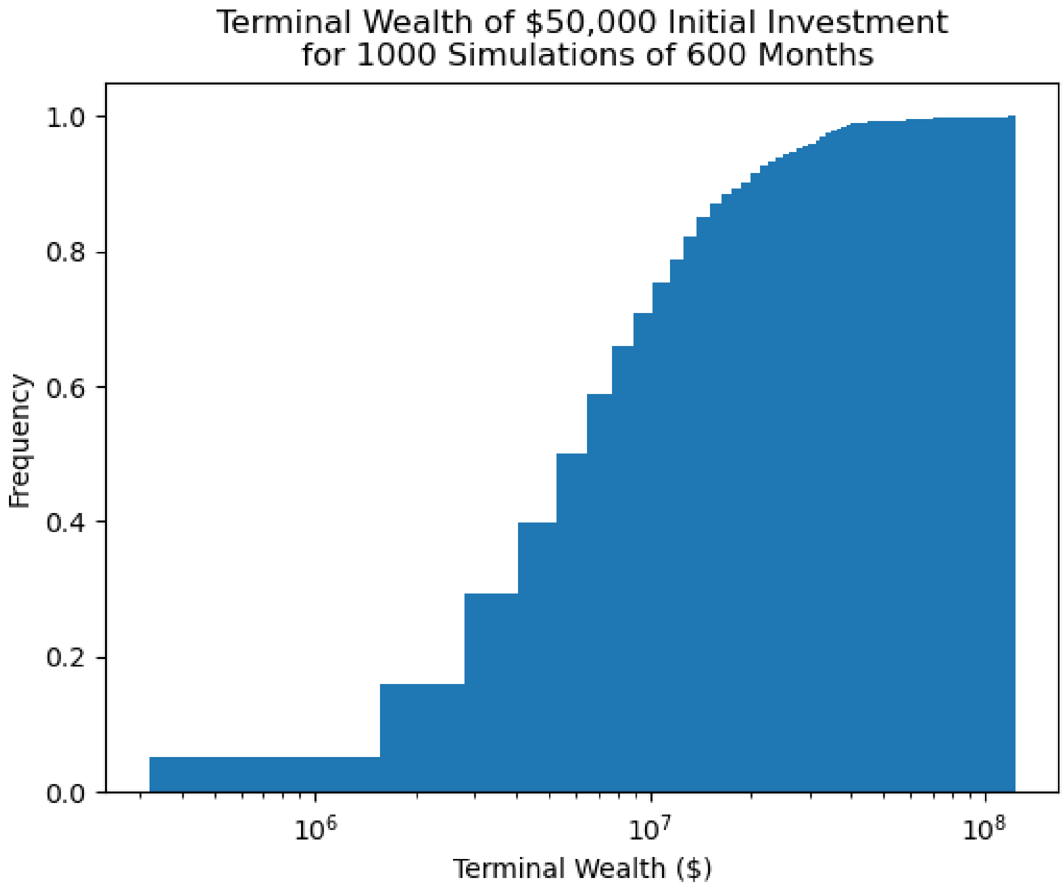
Simulation	0	1	2	3	4
0	0.04	0.01	0.02	0.03	0.08
1	-0.03	-0.02	0.04	0.01	-0.02
2	-0.00	0.06	0.06	0.02	0.00
3	-0.05	-0.02	0.04	0.02	-0.05
4	0.02	-0.04	-0.01	-0.02	0.00

We compound daily returns `r_t` to find future wealth `W_T`.

```
W_T = W_0 * r_t.add(1).prod()
```

We visualize terminal wealth W_T with a cumulative distribution plot.

```
W_T.plot(kind='hist', cumulative=True, density=True, bins=100, logx=True)
plt.xlabel('Terminal Wealth ($)')
plt.title(f'Terminal Wealth of ${W_0:,.0f} Initial Investment\n for {N} Simulations of {T} M
plt.show()
```



The plot above may be difficult to read and interpret because W_T has large outliers. We can use a `for` to calculate several levels of confidence.

```
for i in [0.01, 0.02, 0.03, 0.04, 0.05]:
    print(f'In {100-100*i:,.0f}% of outcomes, an initial investment of ${W_0:,.0f} for {T} mo
```

```
In 99% of outcomes, an initial investment of $50,000 for 600 months has a terminal wealth gre
In 98% of outcomes, an initial investment of $50,000 for 600 months has a terminal wealth gre
In 97% of outcomes, an initial investment of $50,000 for 600 months has a terminal wealth gre
In 96% of outcomes, an initial investment of $50,000 for 600 months has a terminal wealth gre
In 95% of outcomes, an initial investment of $50,000 for 600 months has a terminal wealth gre
```

Repeat the exercise above but add end-of-month investments C_t

We can use the same data frame `r_t` of simulated market returns. However, need to consider end-of-month investments of `C_t`. The easiest approach is to write a `for` that starts with

`W_last = W_0`, then each month updates `W_last = W_last * (1 + r) + C_t`. Finally, it `pd.concat()`s these monthly values of `W_last` into a data frame of monthly wealths `W_t`.

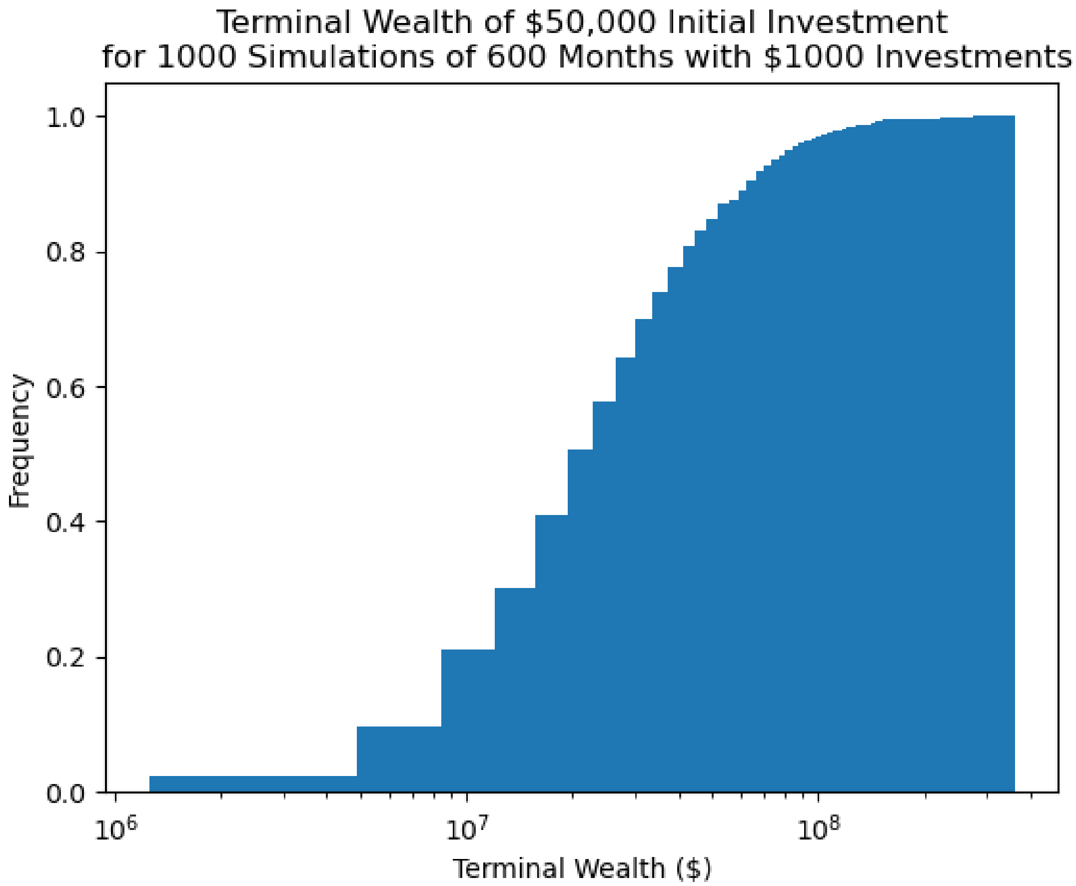
```
C_t = 1_000
W_last = W_0
W_t = []

for i, r in r_t.iterrows():
    W_last = (1 + r) * W_last + C_t
    W_t.append(W_last)

W_t = pd.concat(objs=W_t, axis=1, keys=r_t.index).transpose()
```

```
W_t.iloc[:5, :5]
```

Simulation	0	1	2	3	4
0	53190.00	51595.00	51805.00	52455.00	54830.00
1	52860.25	51449.59	54726.97	54178.88	54996.58
2	53743.96	55613.74	58829.98	56051.16	56106.58
3	52051.39	55662.75	62400.85	57919.95	54318.08
4	54087.21	54514.16	62864.21	57645.71	55328.94

```
W_T = W_t.iloc[-1]
```

```
W_T.plot(kind='hist', cumulative=True, density=True, bins=100, logx=True)
plt.xlabel('Terminal Wealth ($)')
plt.title(f'Terminal Wealth of ${W_0:,.0f} Initial Investment\n for {N} Simulations of {T} Mo
plt.show()
```




```
for i in [0.01, 0.02, 0.03, 0.04, 0.05]:
    print(f'In {100-100*i:,.0f}% of outcomes, an initial investment of ${W_0:,.0f} for {T} mo
```

```
In 99% of outcomes, an initial investment of $50,000 for 600 months with $1000 investments ha
In 98% of outcomes, an initial investment of $50,000 for 600 months with $1000 investments ha
In 97% of outcomes, an initial investment of $50,000 for 600 months with $1000 investments ha
In 96% of outcomes, an initial investment of $50,000 for 600 months with $1000 investments ha
In 95% of outcomes, an initial investment of $50,000 for 600 months with $1000 investments ha
```

Herron Topic 5 - Practice - Sec 03

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import pandas_datareader as pdr
import seaborn as sns
from scipy.stats import norm # new addition for Monte Carlo methods
import warnings # to suppress the pandas_datareader warning
import yfinance as yf
```

```
%precision 2
pd.options.display.float_format = '{:.2f}'.format
# %config InlineBackend.figure_format = 'retina'
```

Announcements

1. Programming assessment and MSFQ assessment will be *in class and in person on Tuesday, 4/15, in your scheduled section*
    1. Details here: https://northeastern.instructure.com/courses/207607/discussion_topics/2765433
    2. And here: https://northeastern.instructure.com/courses/207607/discussion_topics/2727917
2. Our plan for the rest of the semester
    1. Friday, 4/11, Project 2 solution, envelopes solution, and Project 3 group work
    2. Tuesday, 4/15, programming and MSFQ assessment *in class and in person, in your scheduled section*
    3. Friday, 4/18, Project 3 group work
    4. Tuesday, 4/22, Project 3 group work
    5. Wednesday, 4/23, Project 3 is due by 11:59 PM
    6. Friday, 4/25, office hours during regular class time
3. Please complete TRACE! I value and use your feedback.

1. Details here: https://northeastern.instructure.com/courses/207607/discussion_topics/2753925

Five-Minute Recap

In finance, we use two main simulation methods: Monte Carlo and bootstrap. Both help analyze financial data, but work differently.

Monte Carlo methods create many possible paths of financial variables (e.g., stock prices, interest rates) based on theoretical probability distributions:

- Generates synthetic data from assumed models
- Creates data that may not exist in historical records
- Based on random sampling from probability distributions
- Think `np.random.uniform()` or `np.random.normal()` to create new random variables

Bootstrap methods take a different approach:

- Resamples from actual historical data (with or without replacement)
- Makes minimal assumptions about distributions
- Maintains statistical properties of original data
- Think `.sample()` to resample past obervations

Practice

Estimate π by simulating darts thrown at a dart board

Hints: Select random xs and ys such that $-r \leq x \leq +r$ and $-r \leq x \leq +r$. Darts are on the board if $x^2 + y^2 \leq r^2$. The area of the circlular board is πr^2, and the area of square around the board is $(2r)^2 = 4r^2$. The fraction f of darts on the board is the same as the ratio of circle area to square area, so $f = \frac{\pi r^2}{4r^2}$.

First we throw darts at the board. Darts with $x^2 + y^2 \leq r^2$ are on the board.

```
def throw_darts(r=1, n=1_000, seed=42):
    np.random.seed(seed)
    return (
        pd.DataFrame(
            data=np.random.uniform(low=-r, high=r, size=2*n).reshape(n, 2),
            columns=['X', 'Y']
        )
```

```
        .assign(On_Board=lambda x: x['X']**2 + x['Y']**2 <= r**2)
    )
```

```
throw_darts()
```

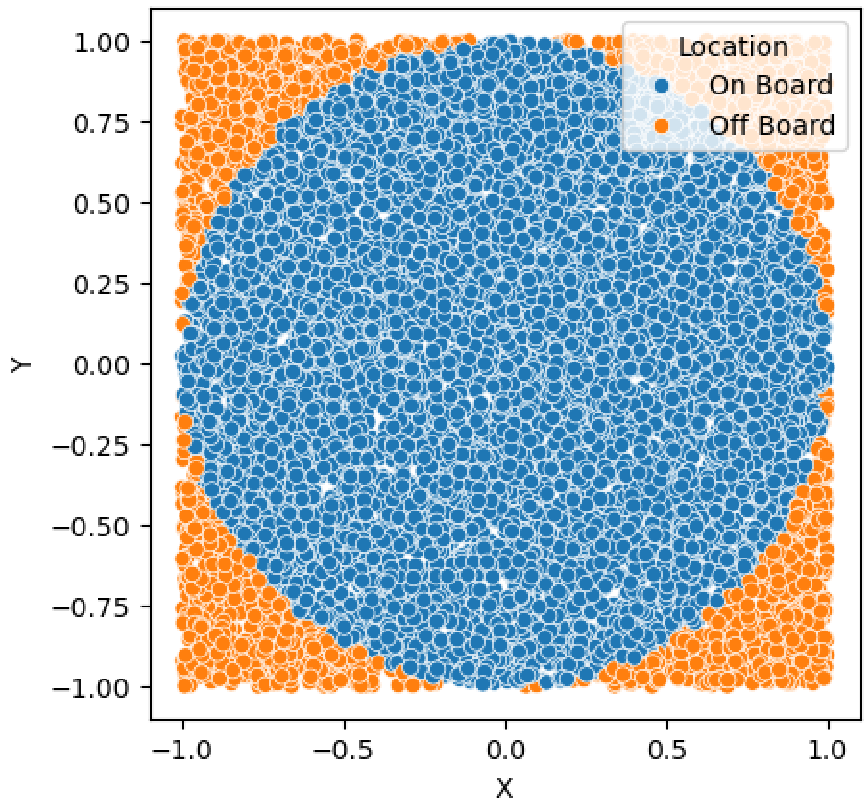
	X	Y	On_Board
0	-0.25	0.90	True
1	0.46	0.20	True
2	-0.69	-0.69	True
3	-0.88	0.73	False
4	0.20	0.42	True
...	...	...	...
995	0.53	-0.68	True
996	0.22	-0.73	True
997	0.50	0.31	True
998	0.91	-0.86	False
999	-0.89	-0.44	True

```
4 * throw_darts(n=1_000_000)['On_Board'].mean()
```

```
3.14
```

Next, we visualize these darts with a scatter plot. Seaborn's `scatterplot()` helps color darts by location (i.e., on or off board). The `.pipe()` method lets us send the output of the `.assign()` method to `sns.scatterplot()` without assigning a temporary data frame.

```
(
    throw_darts(n=10_000)
    .assign(Location=lambda x: np.where(x['On_Board'], 'On Board', 'Off Board'))
    .pipe(
        sns.scatterplot,
        x='X',
        y='Y',
        hue='Location'
    )
)
plt.gca().set_aspect('equal')
```



Finally, we use the hint above to estimate π. The hint above says $f = \frac{\pi r^2}{4r^2}$, where f is the fraction of darts on the board. Therefore, $\pi \approx \frac{4fr^2}{r^2} = 4f$. We increase the precision of our π estimate by increasing the number of simulated darts n.

```
for n in 10**np.arange(7):
    print(f'With {n:>9,.0f} dart throws, our estimate of pi is: {4 * throw_darts(n=n)['On_Boa
```

```
With         1 dart throws, our estimate of pi is: 4.0000
With        10 dart throws, our estimate of pi is: 3.2000
With       100 dart throws, our estimate of pi is: 3.0400
With     1,000 dart throws, our estimate of pi is: 3.1040
With    10,000 dart throws, our estimate of pi is: 3.1544
With   100,000 dart throws, our estimate of pi is: 3.1468
With 1,000,000 dart throws, our estimate of pi is: 3.1420
```

Simulate your wealth W_T by randomly sampling market returns

Use monthly market returns from the French Data Library. Only invest one cash flow W_0, and plot the distribution of W_T.

First, we download data from the French Data Library. We convert these returns from percent to decimal to simplify compounding.

```
with warnings.catch_warnings():
    warnings.filterwarnings("ignore", category=FutureWarning)
    ff3 = (
        pdr.DataReader(
            name='F-F_Research_Data_Factors',
            data_source='famafrench',
            start='1900'
        )
        [0]
        .assign(Mkt=lambda x: x['Mkt-RF'] + x['RF'])
        .div(100)
    )
```

1. Use `.sample()` to simulate one alternate history
2. Add `pd.concat()` and a list comprehension to simulate `N` alternate histories
3. Examine/plot the distributoin of `W_T`

Here is one example `.sample()`:

```
ff3[['Mkt']].sample(n=5, ignore_index=True, random_state=42)
```

	Mkt
0	0.03
1	0.03
2	0.04
3	-0.11
4	-0.02

We can combine many samples of many months with a list comprehension and `pd.concat()`.

```
W_0 = 50_000
T = 12 * 30
N = 1_000
```

```
r_t = pd.concat(
    objs=[ff3['Mkt'].sample(n=T, ignore_index=True, random_state=i) for i in range(N)],
    axis=1,
    keys=range(N),
    names=['Simulation']
)
```

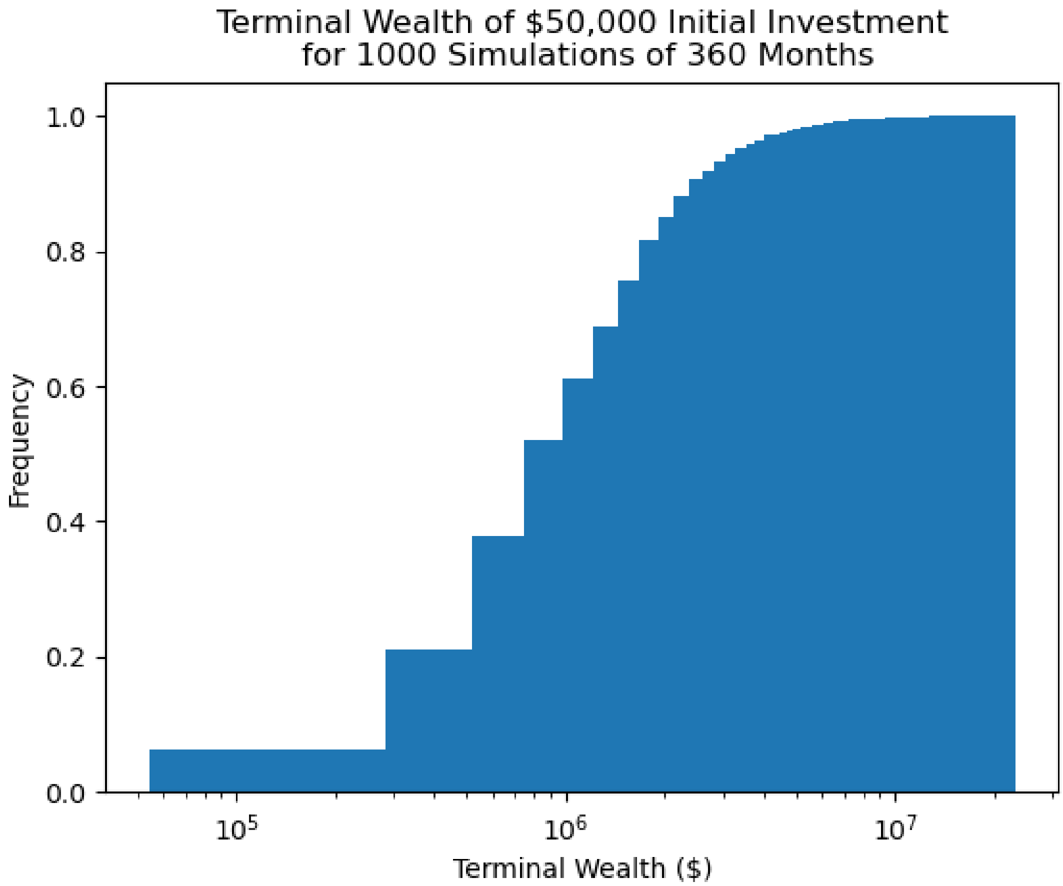
```
r_t.iloc[:5, :5]
```

Simulation	0	1	2	3	4
0	0.04	0.01	0.02	0.03	0.08
1	-0.03	-0.02	0.04	0.01	-0.02
2	-0.00	0.06	0.06	0.02	0.00
3	-0.05	-0.02	0.04	0.02	-0.05
4	0.02	-0.04	-0.01	-0.02	0.00

```
W_T = W_0 * r_t.add(1).prod()
```

We visualize terminal wealth W_T with a cumulative distribution plot.

```
W_T.plot(kind='hist', cumulative=True, density=True, bins=100, logx=True)
plt.xlabel('Terminal Wealth ($)')
plt.title(f'Terminal Wealth of ${W_0:,.0f} Initial Investment\n for {N} Simulations of {T} Mo
plt.show()
```



The plot above may be difficult to read and interpret because W_T has large outliers. We can use a `for` to calculate several levels of confidence.

```
for i in [0.01, 0.02, 0.03, 0.04, 0.05]:
    print(f'In {100-100*i:,.0f}% of outcomes, an initial investment of ${W_0:,.0f} for {T} mo
```

```
In 99% of outcomes, an initial investment of $50,000 for 360 months has a terminal wealth gre
In 98% of outcomes, an initial investment of $50,000 for 360 months has a terminal wealth gre
In 97% of outcomes, an initial investment of $50,000 for 360 months has a terminal wealth gre
In 96% of outcomes, an initial investment of $50,000 for 360 months has a terminal wealth gre
In 95% of outcomes, an initial investment of $50,000 for 360 months has a terminal wealth gre
```

Repeat the exercise above but add end-of-month investments C_t

We can use the same data frame `r_t` of simulated market returns. However, need to consider end-of-month investments of `C_t`. The easiest approach is to write a `for` that starts with

`W_last = W_0`, then each month updates `W_last = W_last * (1 + r) + C_t`. Finally, it `pd.concat()`s these monthly values of `W_last` into a data frame of monthly wealths `W_t`.

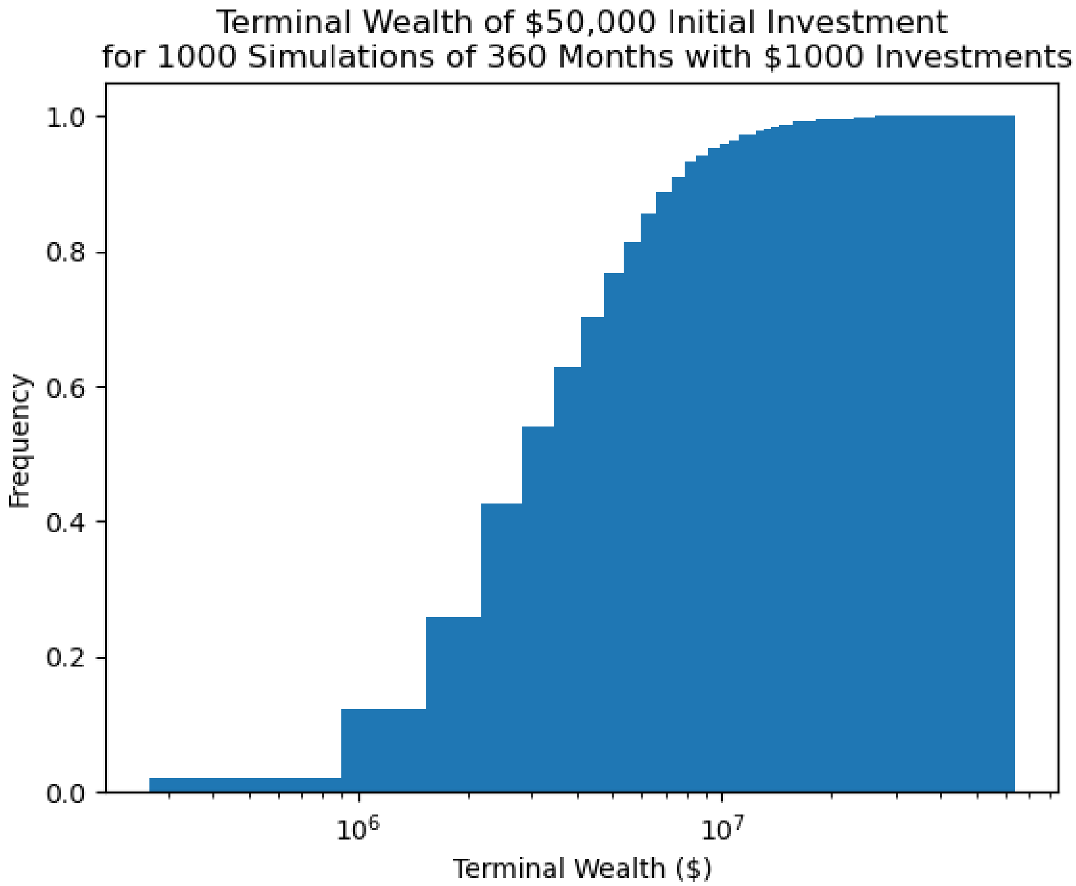
```
C_t = 1_000
W_last = W_0
W_t = []

for i, r in r_t.iterrows():
    W_last = (1 + r) * W_last + C_t
    W_t.append(W_last)

W_t = pd.concat(objs=W_t, axis=1, keys=r_t.index).transpose()
```

```
W_t.iloc[:5, :5]
```

Simulation	0	1	2	3	4
0	53190.00	51595.00	51805.00	52455.00	54830.00
1	52860.25	51449.59	54726.97	54178.88	54996.58
2	53743.96	55613.74	58829.98	56051.16	56106.58
3	52051.39	55662.75	62400.85	57919.95	54318.08
4	54087.21	54514.16	62864.21	57645.71	55328.94

```
W_T = W_t.iloc[-1]
```

```
W_T.plot(kind='hist', cumulative=True, density=True, bins=100, logx=True)
plt.xlabel('Terminal Wealth ($)')
plt.title(f'Terminal Wealth of ${W_0:,.0f} Initial Investment\n for {N} Simulations of {T} Mo
plt.show()
```




```
for i in [0.01, 0.02, 0.03, 0.04, 0.05]:
    print(f'In {100-100*i:,.0f}% of outcomes, an initial investment of ${W_0:,.0f} for {T} mo
```

```
In 99% of outcomes, an initial investment of $50,000 for 360 months with $1000 investments ha
In 98% of outcomes, an initial investment of $50,000 for 360 months with $1000 investments ha
In 97% of outcomes, an initial investment of $50,000 for 360 months with $1000 investments ha
In 96% of outcomes, an initial investment of $50,000 for 360 months with $1000 investments ha
In 95% of outcomes, an initial investment of $50,000 for 360 months with $1000 investments ha
```

Herron Topic 5 - Practice - Sec 03

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import pandas_datareader as pdr
import seaborn as sns
from scipy.stats import norm # new addition for Monte Carlo methods
import warnings # to suppress the pandas_datareader warning
import yfinance as yf
```

```
%precision 2
pd.options.display.float_format = '{:.2f}'.format
# %config InlineBackend.figure_format = 'retina'
```

Announcements

1. Programming assessment and MSFQ assessment will be *in class and in person on Tuesday, 4/15, in your scheduled section*
    1. Details here: https://northeastern.instructure.com/courses/207607/discussion_topics/2765433
    2. And here: https://northeastern.instructure.com/courses/207607/discussion_topics/2727917
2. Our plan for the rest of the semester
    1. Friday, 4/11, Project 2 solution, envelopes solution, and Project 3 group work
    2. Tuesday, 4/15, programming and MSFQ assessment *in class and in person, in your scheduled section*
    3. Friday, 4/18, Project 3 group work
    4. Tuesday, 4/22, Project 3 group work
    5. Wednesday, 4/23, Project 3 is due by 11:59 PM
    6. Friday, 4/25, office hours during regular class time
3. Please complete TRACE! I value and use your feedback.

1. Details here: https://northeastern.instructure.com/courses/207607/discussion_topics/2753925

Five-Minute Recap

In finance, we use two main simulation methods: Monte Carlo and bootstrap. Both help analyze financial data, but work differently.

Monte Carlo methods create many possible paths of financial variables (e.g., stock prices, interest rates) based on theoretical probability distributions:

- Generates synthetic data from assumed models
- Creates data that may not exist in historical records
- Based on random sampling from probability distributions
- Think `np.random.uniform()` or `np.random.normal()` to create new random variables

Bootstrap methods take a different approach:

- Resamples from actual historical data (with or without replacement)
- Makes minimal assumptions about distributions
- Maintains statistical properties of original data
- Think `.sample()` to resample past obervations

Practice

Estimate π by simulating darts thrown at a dart board

Hints: Select random xs and ys such that $-r \leq x \leq +r$ and $-r \leq x \leq +r$. Darts are on the board if $x^2 + y^2 \leq r^2$. The area of the circlular board is πr^2, and the area of square around the board is $(2r)^2 = 4r^2$. The fraction f of darts on the board is the same as the ratio of circle area to square area, so $f = \frac{\pi r^2}{4r^2}$.

First we throw darts at the board. Darts with $x^2 + y^2 \leq r^2$ are on the board.

```
def throw_darts(n=1_000, r=1, seed=42):
    np.random.seed(seed)
    return (
        pd.DataFrame(
            data=np.random.uniform(low=-r, high=r, size=2*n).reshape(n, 2),
            columns=['X', 'Y']
        )
```

```
        .assign(On_Board=lambda x: x['X']**2 + x['Y']**2 <= r**2)
    )
```

```
throw_darts()
```

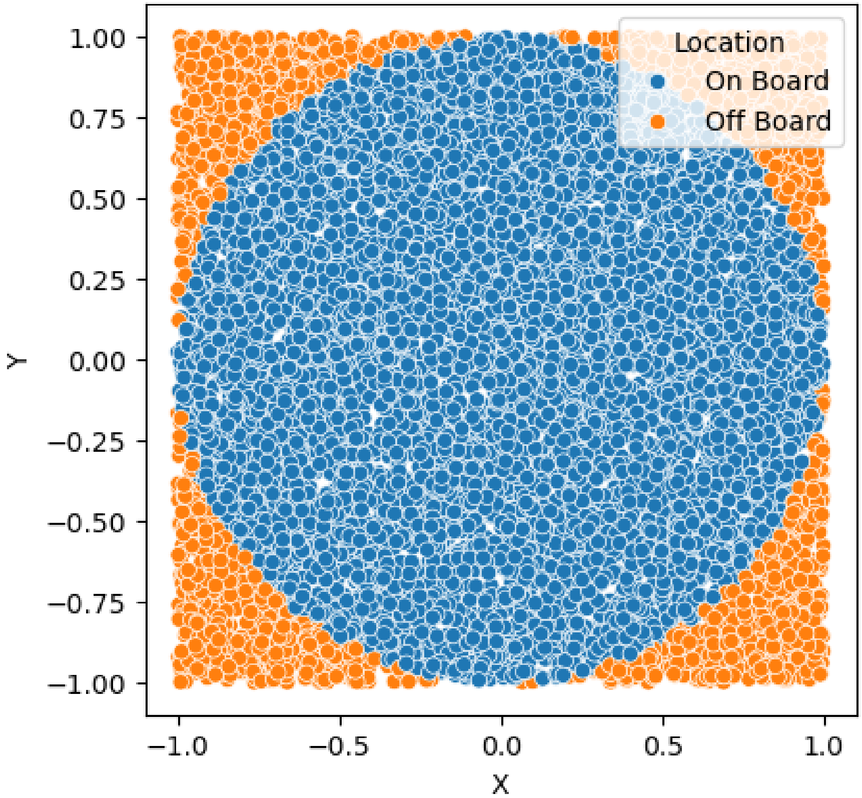
	X	Y	On_Board
0	-0.25	0.90	True
1	0.46	0.20	True
2	-0.69	-0.69	True
3	-0.88	0.73	False
4	0.20	0.42	True
...	...	...	...
995	0.53	-0.68	True
996	0.22	-0.73	True
997	0.50	0.31	True
998	0.91	-0.86	False
999	-0.89	-0.44	True

```
4 * throw_darts(n=1_000_000)['On_Board'].mean()
```

```
3.14
```

Next, we visualize these darts with a scatter plot. Seaborn's `scatterplot()` helps color darts by location (i.e., on or off board). The `.pipe()` method lets us send the output of the `.assign()` method to `sns.scatterplot()` without assigning a temporary data frame.

```
(
    throw_darts(n=10_000)
    .assign(Location=lambda x: np.where(x['On_Board'], 'On Board', 'Off Board'))
    .pipe(
        sns.scatterplot,
        x='X',
        y='Y',
        hue='Location'
    )
)
plt.gca().set_aspect('equal')
```



Finally, we use the hint above to estimate π. The hint above says $f = \frac{\pi r^2}{4r^2}$, where f is the fraction of darts on the board. Therefore, $\pi \approx \frac{4fr^2}{r^2} = 4f$. We increase the precision of our π estimate by increasing the number of simulated darts n.

```
for n in 10**np.arange(7):
    print(f'With {n:>9,.0f} dart throws, our estimate of pi is: {4 * throw_darts(n=n)['On_Boa
```

```
With         1 dart throws, our estimate of pi is: 4.0000
With        10 dart throws, our estimate of pi is: 3.2000
With       100 dart throws, our estimate of pi is: 3.0400
With     1,000 dart throws, our estimate of pi is: 3.1040
With    10,000 dart throws, our estimate of pi is: 3.1544
With   100,000 dart throws, our estimate of pi is: 3.1468
With 1,000,000 dart throws, our estimate of pi is: 3.1420
```

Simulate your wealth W_T by randomly sampling market returns

Use monthly market returns from the French Data Library. Only invest one cash flow W_0, and plot the distribution of W_T.

First, we download data from the French Data Library. We convert these returns from percent to decimal to simplify compounding.

```
with warnings.catch_warnings():
    warnings.filterwarnings("ignore", category=FutureWarning)
    ff3 = (
        pdr.DataReader(
            name='F-F_Research_Data_Factors',
            data_source='famafrench',
            start='1900'
        )
        [0]
        .assign(Mkt=lambda x: x['Mkt-RF'] + x['RF'])
        .div(100)
    )
```

1. Use `.sample()` to simulate one alternate history
2. Add `pd.concat()` and a list comprehension to simulate `N` alternate histories
3. Examine/plot the distributoin of `W_T`

```
ff3['Mkt'].sample(n=5, random_state=42, ignore_index=True)
```

```
0    0.03
1    0.03
2    0.04
3   -0.11
4   -0.02
Name: Mkt, dtype: float64
```

We can combine many samples of many months with a list comprehension and `pd.concat()`.

```
W_0 = 50_000
T = 30 * 12
N = 1_000

r_t = pd.concat(
    objs=[ff3['Mkt'].sample(n=T, random_state=i, ignore_index=True) for i in range(N)],
```

```
    keys=range(N),
    names=['Simulation'],
    axis=1
)
```

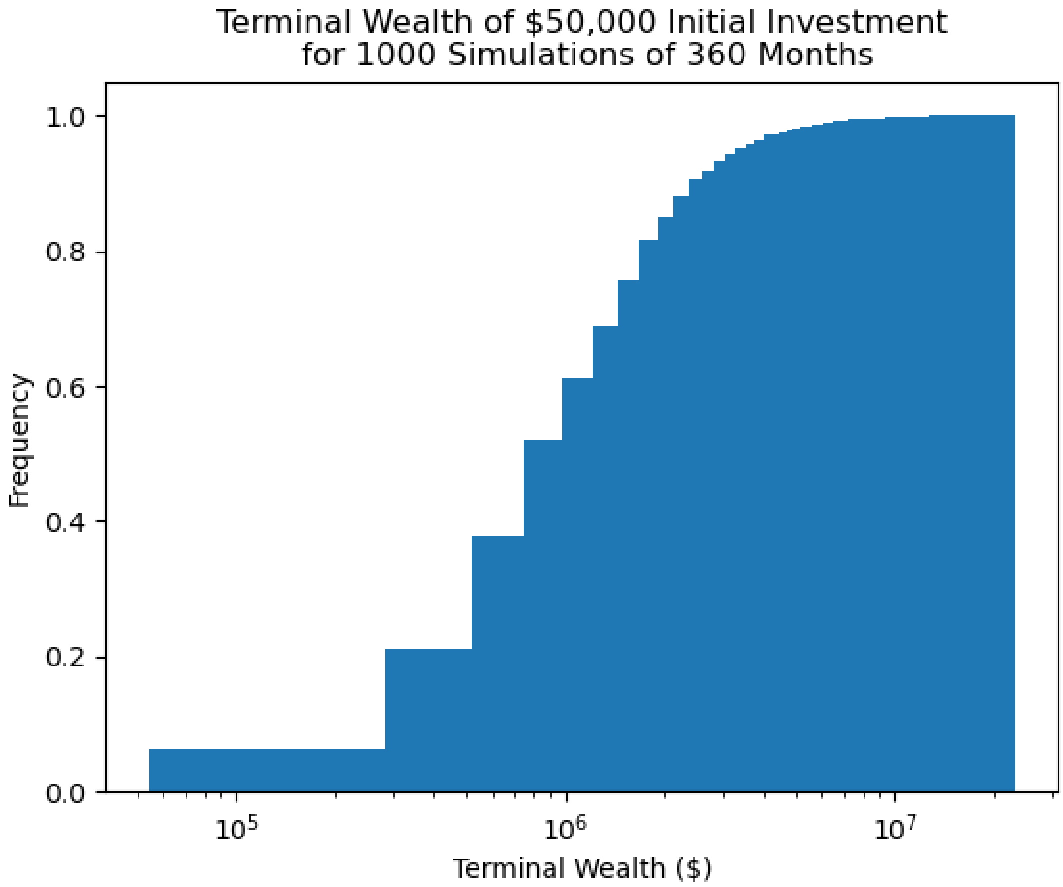
```
r_t.iloc[:5, :5]
```

Simulation	0	1	2	3	4
0	0.04	0.01	0.02	0.03	0.08
1	-0.03	-0.02	0.04	0.01	-0.02
2	-0.00	0.06	0.06	0.02	0.00
3	-0.05	-0.02	0.04	0.02	-0.05
4	0.02	-0.04	-0.01	-0.02	0.00

```
W_T = W_0 * r_t.add(1).prod()
```

We visualize terminal wealth W_T with a cumulative distribution plot.

```
W_T.plot(kind='hist', cumulative=True, density=True, bins=100, logx=True)
plt.xlabel('Terminal Wealth ($)')
plt.title(f'Terminal Wealth of ${W_0:,.0f} Initial Investment\n for {N} Simulations of {T} Mo
plt.show()
```



The plot above may be difficult to read and interpret because W_T has large outliers. We can use a `for` to calculate several levels of confidence.

```
for i in [0.01, 0.02, 0.03, 0.04, 0.05]:
    print(f'In {100-100*i:,.0f}% of outcomes, an initial investment of ${W_0:,.0f} for {T} mo
```

```
In 99% of outcomes, an initial investment of $50,000 for 360 months has a terminal wealth gre
In 98% of outcomes, an initial investment of $50,000 for 360 months has a terminal wealth gre
In 97% of outcomes, an initial investment of $50,000 for 360 months has a terminal wealth gre
In 96% of outcomes, an initial investment of $50,000 for 360 months has a terminal wealth gre
In 95% of outcomes, an initial investment of $50,000 for 360 months has a terminal wealth gre
```

Repeat the exercise above but add end-of-month investments C_t

We can use the same data frame `r_t` of simulated market returns. However, need to consider end-of-month investments of `C_t`. The easiest approach is to write a `for` that starts with

`W_last = W_0`, then each month updates `W_last = W_last * (1 + r) + C_t`. Finally, it `pd.concat()`s these monthly values of `W_last` into a data frame of monthly wealths `W_t`.

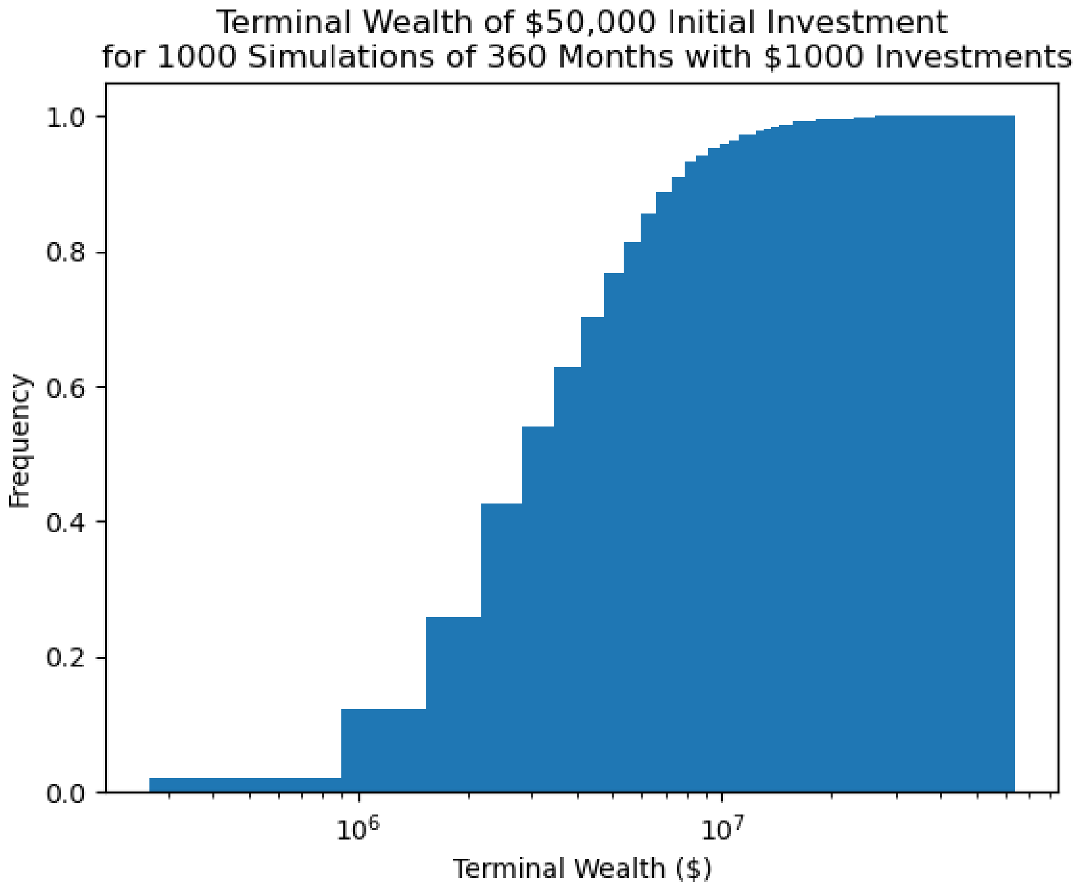
```
C_t = 1_000
W_last = W_0
W_t = []

for i, r in r_t.iterrows():
    W_last = (1 + r) * W_last + C_t
    W_t.append(W_last)

W_t = pd.concat(objs=W_t, axis=1, keys=r_t.index).transpose()
```

```
W_t.iloc[:5, :5]
```

Simulation	0	1	2	3	4
0	53190.00	51595.00	51805.00	52455.00	54830.00
1	52860.25	51449.59	54726.97	54178.88	54996.58
2	53743.96	55613.74	58829.98	56051.16	56106.58
3	52051.39	55662.75	62400.85	57919.95	54318.08
4	54087.21	54514.16	62864.21	57645.71	55328.94

```
W_T = W_t.iloc[-1]
```

```
W_T.plot(kind='hist', cumulative=True, density=True, bins=100, logx=True)
plt.xlabel('Terminal Wealth ($)')
plt.title(f'Terminal Wealth of ${W_0:,.0f} Initial Investment\n for {N} Simulations of {T} Mo
plt.show()
```




```
for i in [0.01, 0.02, 0.03, 0.04, 0.05]:
    print(f'In {100-100*i:,.0f}% of outcomes, an initial investment of ${W_0:,.0f} for {T} mo
```

```
In 99% of outcomes, an initial investment of $50,000 for 360 months with $1000 investments ha
In 98% of outcomes, an initial investment of $50,000 for 360 months with $1000 investments ha
In 97% of outcomes, an initial investment of $50,000 for 360 months with $1000 investments ha
In 96% of outcomes, an initial investment of $50,000 for 360 months with $1000 investments ha
In 95% of outcomes, an initial investment of $50,000 for 360 months with $1000 investments ha
```

The Monty Hall Problem - Sec 02

```
import io
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import random
import seaborn as sns
```

Introduction

The Monty Hall problem is a probability puzzle from *Let's Make a Deal*. You pick one of three doors (car behind one, goats behind two), the host reveals a goat, and you decide to switch or stay. We played this in class with envelopes. This problem is often used in quantitative finance interviews because it tests your ability to think about probabilities (see Crack (2024)). This notebook explores the Monty Hall problem four ways:

1. **Intuition:** We will use simple logic to understand why switching might be better.
2. **Bayes' Theorem:** We will use a mathematical rule to calculate the exact probabilities.
3. **Class Data:** We will analyze the results from our envelope game in class.
4. **Simulated Data:** We will use Python to simulate the game thousands of times to see the results.

We will show that switching wins $\frac{2}{3}$ of the time, staying wins $\frac{1}{3}$. I based this notebook on Crack (2024), Kritzman (2000), and the Monty Hall problem Wikipedia page.

Intuition

Let us solve the Monty Hall problem with simple logic. We will use the same setup as in the introduction: You pick Door 1, the host opens Door 3 (revealing a goat), and you decide whether to switch to Door 2. Here are the steps in the intuitive solution:

1. **Initial Choice:** There are three doors, and the car is equally likely to be behind any one of them. So, when you pick Door 1, the probability that the car is behind Door 1 is $\frac{1}{3}$. That means there is a $\frac{2}{3}$ chance the car is behind one of the other two doors (Door 2 or Door 3 combined).
2. **Host Opens a Door:** The host, who knows where the car is, opens Door 3 and shows it has a goat. Now we know Door 3 does not have the car.
3. **Probability Transfer:** Before the host opened Door 3, there was a $\frac{2}{3}$ chance the car was behind Door 2 or Door 3. Since Door 3 has a goat, that entire $\frac{2}{3}$ probability now belongs to Door 2 (the door you can switch to). The probability that the car is behind Door 1 (your original choice) is still $\frac{1}{3}$, because nothing has changed about Door 1.

In conclusion:

- If you **switch** to Door 2, your probability of winning is $\frac{2}{3}$.
- If you **stay** with Door 1, your probability of winning is $\frac{1}{3}$.

So, you should switch to Door 2 to have a better chance of winning the car!

Bayes' Theorem

We can also solve the Monty Hall problem using a mathematical rule called Bayes' Theorem. Bayes' Theorem helps us calculate the probability of something happening based on new information. Here, we will use it to find the probability that the car is behind Door 2, given that you picked Door 1 and the host opened Door 3 (revealing a goat).

Step 1: Define the Events

- Let B_2 be the event that the car is behind Door 2 (the door you can switch to).
- Let H_3 be the event that the host opens Door 3 (and shows a goat).

We want to find $P(B_2|H_3)$, which is the probability that the car is behind Door 2, given that the host opened Door 3.

Step 2: Bayes' Theorem Formula

Bayes' Theorem says:

$$P(B_2|H_3) = \frac{P(H_3|B_2) \times P(B_2)}{P(H_3)}$$

Let us break down each part:

- $P(H_3|B_2)$: The probability that the host opens Door 3, given that the car is behind Door 2. You picked Door 1, so the host must open a door that's not Door 2 (the car) and not Door 1 (your choice). The only option is Door 3, so $P(H_3|B_2) = 1$.
- $P(B_2)$: The probability that the car is behind Door 2 before we know anything else. Since the car is equally likely to be behind any of the three doors, $P(B_2) = \frac{1}{3}$.
- $P(H_3)$: The total probability that the host opens Door 3. This is trickier, so let's calculate it carefully.

Step 3: Calculate $P(H_3)$

We can define $P(H_3)$ as follows:

$$P(H_3) = [P(H_3|B_1) \times P(B_1)] + [P(H_3|B_2) \times P(B_2)] + [P(H_3|B_3) \times P(B_3)]$$

The host can open Door 3 in three scenarios, depending on where the car is:

- **Car behind Door 1 (B_1):** Probability $P(B_1) = \frac{1}{3}$. You picked Door 1, so the host must open a door that's not Door 1 (your choice). The host randomly picks a door, so $P(H_2|B_1) = P(H_3|B_1) = \frac{1}{2}$.
- **Car behind Door 2 (B_2):** Probability $P(B_2) = \frac{1}{3}$. We calculate above that $P(H_3|B_2) = 1$.
- **Car behind Door 3 (B_3):** Probability $P(B_3) = \frac{1}{3}$. But if Door 3 has the car, the host cannot open Door 3, and $P(H_3|B_3) = 0$.

Therefore, we can calculate $P(H_3)$ as follows:

$$P(H_3) = (\frac{1}{2} \times \frac{1}{3}) + (1 \times \frac{1}{3}) + (0 \times \frac{1}{3}) = \frac{1}{6} + \frac{1}{3} + 0 = \frac{1}{2}$$

Step 4: Apply Bayes' Theorem

Now we can plug everything into the formula:

$$P(B_2|H_3) = \frac{P(H_3|B_2) \times P(B_2)}{P(H_3)} = \frac{1 \times \frac{1}{3}}{\frac{1}{2}} = \frac{\frac{1}{3}}{\frac{1}{2}} = \frac{1}{3} \times \frac{2}{1} = \frac{2}{3}$$

This confirms that switching doubles your odds.

Conclusion

The probability that the car is behind Door 2, given that the host opened Door 3, is $\frac{2}{3}$. This matches our intuition! You should switch to Door 2 to have a $\frac{2}{3}$ chance of winning, compared to a $\frac{1}{3}$ chance if you stay with Door 1.

Class Data

Let us look at the results from our envelope game in class. Here are the data:

```
data_class = """
Date,Section,Switched,Won
2025-02-07,2,0,1
2025-02-07,3,1,0
2025-02-07,4,0,0
2025-02-11,2,0,1
2025-02-11,3,0,0
2025-02-11,4,0,1
2025-02-18,2,0,0
2025-02-18,3,1,0
2025-02-18,4,0,0
2025-02-21,2,1,1
2025-02-21,3,1,1
2025-02-21,4,1,0
2025-03-14,2,0,0
2025-03-14,3,1,1
2025-03-14,4,1,1
2025-03-21,2,1,1
2025-03-21,3,1,0
2025-03-22,4,0,0
2025-04-01,2,0,0
2025-04-01,3,1,1
2025-04-01,4,0,0
2025-04-04,2,1,0
2025-04-04,3,1,0
2025-04-08,2,1,1
2025-04-08,3,1,1
2025-04-08,4,0,0
"""
```

```
df_class = (
    pd.read_csv(
        filepath_or_buffer=io.StringIO(data_class),
        parse_dates=['Date'],
        index_col=['Date']
    )
    .assign(Choice=lambda x: np.where(x['Switched']==1, 'Switch', 'Stay'))
)
```

We can calculate the fraction of times students won when they switched versus when they stayed. Let us use Python to do this.

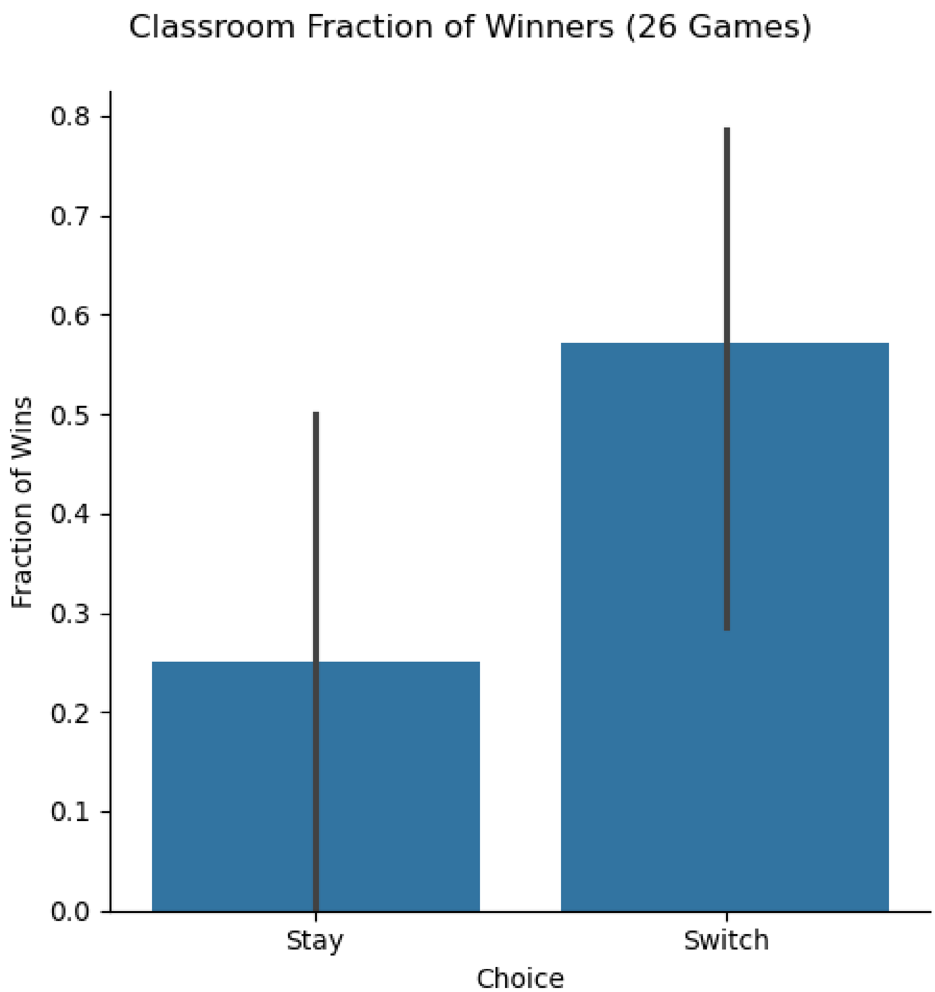
```
(
    df_class
    .groupby(by='Choice')
    [['Won']]
    .mean()
    .round(4)
)
```

	Won
Choice	
Stay	0.2500
Switch	0.5714

Our class data supports the idea that switching is better! Furthermore, these results are close to the theoretical probabilities we calculated: $\frac{1}{3}$ for staying and $\frac{2}{3}$ for switching.

Let us plot the win rates with a bar chart. The black lines on each bar show the 95% confidence interval.

```
sns.catplot(
    data=df_class,
    x='Choice',
    y='Won',
    kind='bar',
    errorbar=('ci', 95)
)
plt.suptitle(f'Classroom Fraction of Winners ({df_class.shape[0]} Games)', y=1.05)
plt.ylabel('Fraction of Wins')
plt.show()
```



Simulated Data

We can simulate as many plays of the game as we want! The code below simulates 10,000 plays of the game. For each play, we calculate whether switching or staying wins. So, for every play, we have both stay and switch outcomes

```
trials = 10_000
doors = [1, 2, 3]
doors_set = set(doors)

random.seed(42)
```

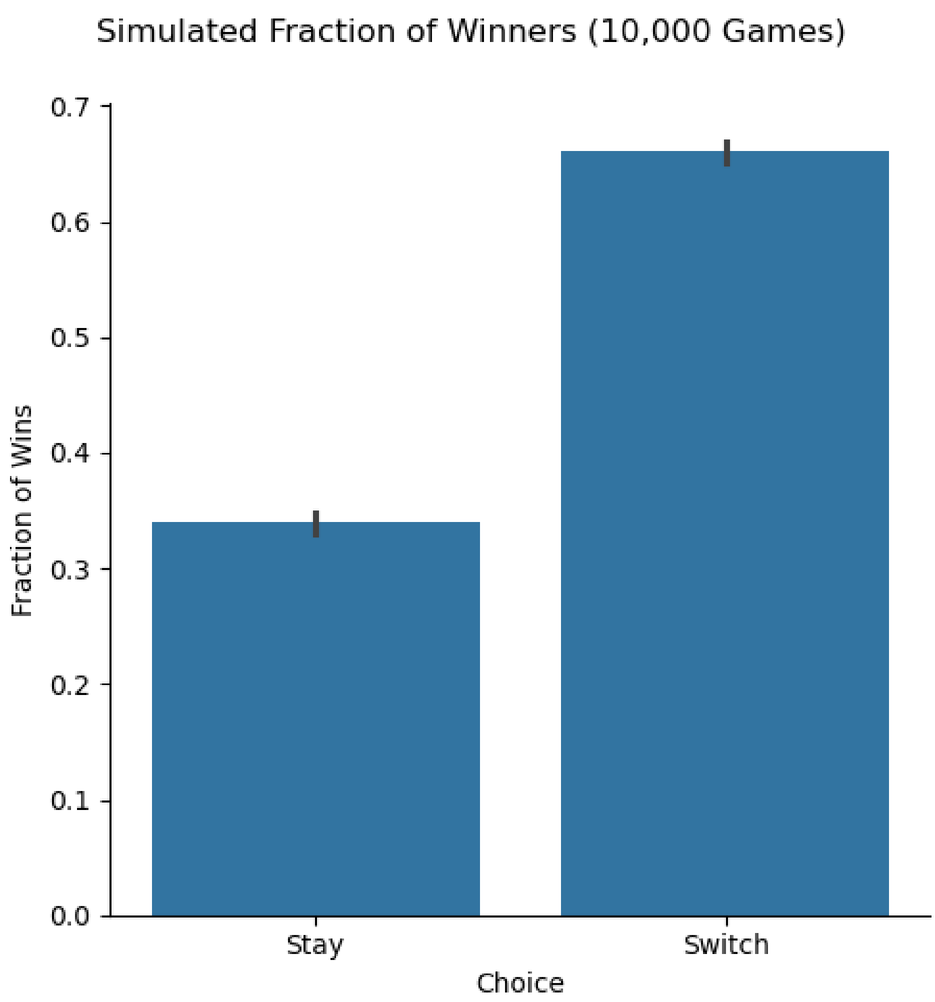
```
df_sim = (
    pd.DataFrame(
        data={
            'Winner': [random.choice(doors) for _ in range(trials)],
            'Choice_1': [random.choice(doors) for _ in range(trials)],
            'Open': np.nan,
            'Choice_2': np.nan,
            'Stay': np.nan,
            'Switch': np.nan
        },
        index=pd.RangeIndex(stop=trials, name='Simulation')
    )
    .assign(
        Open=lambda x: [(doors_set - {w, c}).pop() for w, c in x[['Winner', 'Choice_1']].ite
        Choice_2=lambda x: [(doors_set - {c, o}).pop() for c, o in x[['Choice_1', 'Open']].i
        Stay=lambda x: x['Choice_1'] == x['Winner'],
        Switch=lambda x: x['Choice_2'] == x['Winner']
    )
    .melt(
        value_vars=['Switch', 'Stay'],
        var_name='Choice',
        value_name='Won')
)
```

```
(
    df_sim
    .groupby(by='Choice')
    [['Won']]
    .mean()
    .round(4)
)
```

	Won
Choice	
Stay	0.3396
Switch	0.6604

With 10,000 simulations, these results are much more reliable than our class data. They confirm that switching is the better strategy!

```
sns.catplot(
    data=df_sim.sort_values(by='Choice'),
    x='Choice',
    y='Won',
    kind='bar',
    errorbar=('ci', 95)
)
plt.suptitle(f'Simulated Fraction of Winners ({trials:,.0f} Games)', y=1.05)
plt.ylabel('Fraction of Wins')
plt.show()
```



Conclusion

We explored the Monty Hall problem in four different ways, and they all point to the same conclusion:

- **Intuition:** Switching transfers the $\frac{2}{3}$ probability from the other two doors to the remaining door.
- **Bayes' Theorem:** The probability of winning by switching is $\frac{2}{3}$.
- **Class Data:** Our 26 games showed that switchers won 57% of the time, compared to 25% for stayers.
- **Simulation:** Over 10,000 simulated games, switchers won 67% of the time, compared to 33% for stayers.

All these methods agree: you should always switch in the Monty Hall problem to maximize your chance of winning the car, giving you a $\frac{2}{3}$ probability of success. Staying with your original choice only gives you a $\frac{1}{3}$ chance. So, next time you're on a game show (or playing an envelope game in class), remember to switch!

The Monty Hall Problem - Sec 03

```
import io
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import random
import seaborn as sns
```

Introduction

The Monty Hall problem is a probability puzzle from *Let's Make a Deal*. You pick one of three doors (car behind one, goats behind two), the host reveals a goat, and you decide to switch or stay. We played this in class with envelopes. This problem is often used in quantitative finance interviews because it tests your ability to think about probabilities (see Crack (2024)). This notebook explores the Monty Hall problem four ways:

1. **Intuition:** We will use simple logic to understand why switching might be better.
2. **Bayes' Theorem:** We will use a mathematical rule to calculate the exact probabilities.
3. **Class Data:** We will analyze the results from our envelope game in class.
4. **Simulated Data:** We will use Python to simulate the game thousands of times to see the results.

We will show that switching wins $\frac{2}{3}$ of the time, staying wins $\frac{1}{3}$. I based this notebook on Crack (2024), Kritzman (2000), and the Monty Hall problem Wikipedia page.

Intuition

Let us solve the Monty Hall problem with simple logic. We will use the same setup as in the introduction: You pick Door 1, the host opens Door 3 (revealing a goat), and you decide whether to switch to Door 2. Here are the steps in the intuitive solution:

1. **Initial Choice:** There are three doors, and the car is equally likely to be behind any one of them. So, when you pick Door 1, the probability that the car is behind Door 1 is $\frac{1}{3}$. That means there is a $\frac{2}{3}$ chance the car is behind one of the other two doors (Door 2 or Door 3 combined).
2. **Host Opens a Door:** The host, who knows where the car is, opens Door 3 and shows it has a goat. Now we know Door 3 does not have the car.
3. **Probability Transfer:** Before the host opened Door 3, there was a $\frac{2}{3}$ chance the car was behind Door 2 or Door 3. Since Door 3 has a goat, that entire $\frac{2}{3}$ probability now belongs to Door 2 (the door you can switch to). The probability that the car is behind Door 1 (your original choice) is still $\frac{1}{3}$, because nothing has changed about Door 1.

In conclusion:

- If you **switch** to Door 2, your probability of winning is $\frac{2}{3}$.
- If you **stay** with Door 1, your probability of winning is $\frac{1}{3}$.

So, you should switch to Door 2 to have a better chance of winning the car!

Bayes' Theorem

We can also solve the Monty Hall problem using a mathematical rule called Bayes' Theorem. Bayes' Theorem helps us calculate the probability of something happening based on new information. Here, we will use it to find the probability that the car is behind Door 2, given that you picked Door 1 and the host opened Door 3 (revealing a goat).

Step 1: Define the Events

- Let B_2 be the event that the car is behind Door 2 (the door you can switch to).
- Let H_3 be the event that the host opens Door 3 (and shows a goat).

We want to find $P(B_2|H_3)$, which is the probability that the car is behind Door 2, given that the host opened Door 3.

Step 2: Bayes' Theorem Formula

Bayes' Theorem says:

$$P(B_2|H_3) = \frac{P(H_3|B_2) \times P(B_2)}{P(H_3)}$$

Let us break down each part:

- $P(H_3|B_2)$: The probability that the host opens Door 3, given that the car is behind Door 2. You picked Door 1, so the host must open a door that's not Door 2 (the car) and not Door 1 (your choice). The only option is Door 3, so $P(H_3|B_2) = 1$.
- $P(B_2)$: The probability that the car is behind Door 2 before we know anything else. Since the car is equally likely to be behind any of the three doors, $P(B_2) = \frac{1}{3}$.
- $P(H_3)$: The total probability that the host opens Door 3. This is trickier, so let's calculate it carefully.

Step 3: Calculate $P(H_3)$

We can define $P(H_3)$ as follows:

$$P(H_3) = [P(H_3|B_1) \times P(B_1)] + [P(H_3|B_2) \times P(B_2)] + [P(H_3|B_3) \times P(B_3)]$$

The host can open Door 3 in three scenarios, depending on where the car is:

- **Car behind Door 1 (B_1):** Probability $P(B_1) = \frac{1}{3}$. You picked Door 1, so the host must open a door that's not Door 1 (your choice). The host randomly picks a door, so $P(H_2|B_1) = P(H_3|B_1) = \frac{1}{2}$.
- **Car behind Door 2 (B_2):** Probability $P(B_2) = \frac{1}{3}$. We calculate above that $P(H_3|B_2) = 1$.
- **Car behind Door 3 (B_3):** Probability $P(B_3) = \frac{1}{3}$. But if Door 3 has the car, the host cannot open Door 3, and $P(H_3|B_3) = 0$.

Therefore, we can calculate $P(H_3)$ as follows:

$$P(H_3) = (\frac{1}{2} \times \frac{1}{3}) + (1 \times \frac{1}{3}) + (0 \times \frac{1}{3}) = \frac{1}{6} + \frac{1}{3} + 0 = \frac{1}{2}$$

Step 4: Apply Bayes' Theorem

Now we can plug everything into the formula:

$$P(B_2|H_3) = \frac{P(H_3|B_2) \times P(B_2)}{P(H_3)} = \frac{1 \times \frac{1}{3}}{\frac{1}{2}} = \frac{\frac{1}{3}}{\frac{1}{2}} = \frac{1}{3} \times \frac{2}{1} = \frac{2}{3}$$

This confirms that switching doubles your odds.

Conclusion

The probability that the car is behind Door 2, given that the host opened Door 3, is $\frac{2}{3}$. This matches our intuition! You should switch to Door 2 to have a $\frac{2}{3}$ chance of winning, compared to a $\frac{1}{3}$ chance if you stay with Door 1.

Class Data

Let us look at the results from our envelope game in class. Here are the data:

```
data_class = """
Date,Section,Switched,Won
2025-02-07,2,0,1
2025-02-07,3,1,0
2025-02-07,4,0,0
2025-02-11,2,0,1
2025-02-11,3,0,0
2025-02-11,4,0,1
2025-02-18,2,0,0
2025-02-18,3,1,0
2025-02-18,4,0,0
2025-02-21,2,1,1
2025-02-21,3,1,1
2025-02-21,4,1,0
2025-03-14,2,0,0
2025-03-14,3,1,1
2025-03-14,4,1,1
2025-03-21,2,1,1
2025-03-21,3,1,0
2025-03-22,4,0,0
2025-04-01,2,0,0
2025-04-01,3,1,1
2025-04-01,4,0,0
2025-04-04,2,1,0
2025-04-04,3,1,0
2025-04-08,2,1,1
2025-04-08,3,1,1
2025-04-08,4,0,0
"""
```

```
df_class = (
    pd.read_csv(
        filepath_or_buffer=io.StringIO(data_class),
        parse_dates=['Date'],
        index_col=['Date']
    )
    .assign(Choice=lambda x: np.where(x['Switched']==1, 'Switch', 'Stay'))
)
```

We can calculate the fraction of times students won when they switched versus when they stayed. Let us use Python to do this.

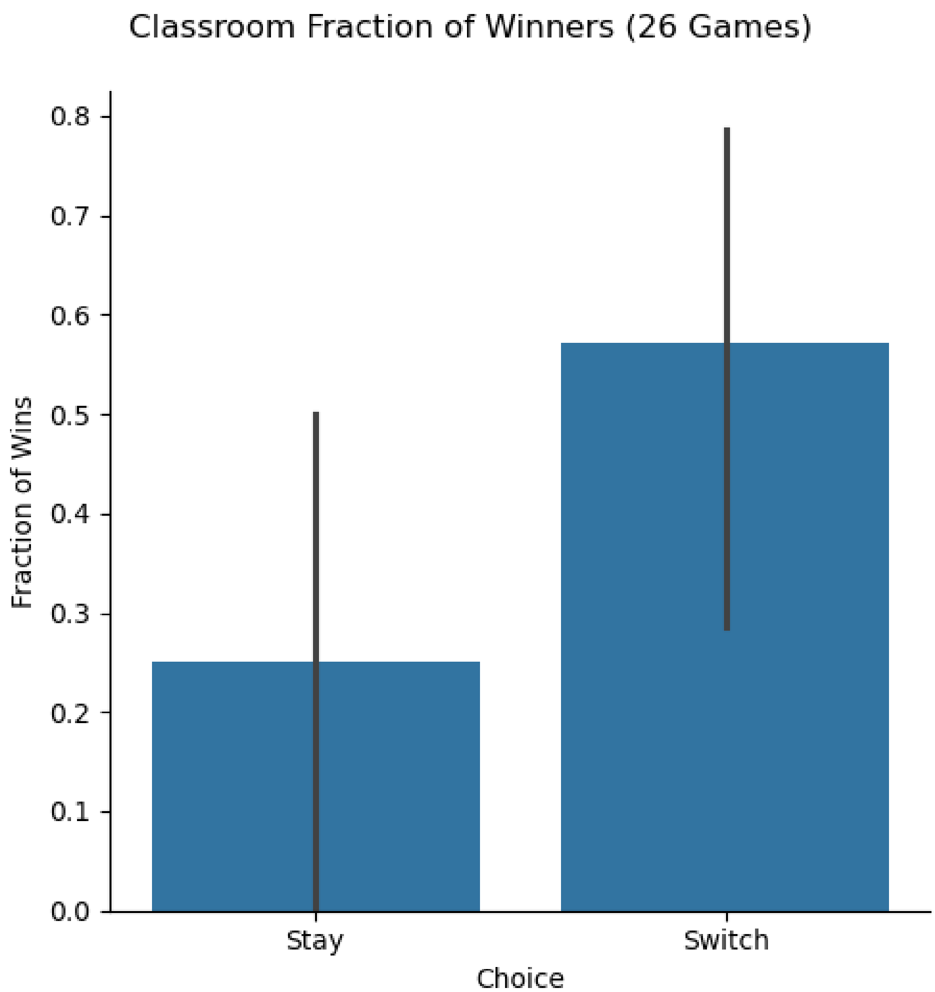
```
(
    df_class
    .groupby(by='Choice')
    [['Won']]
    .mean()
    .round(4)
)
```

	Won
Choice	
Stay	0.2500
Switch	0.5714

Our class data supports the idea that switching is better! Furthermore, these results are close to the theoretical probabilities we calculated: $\frac{1}{3}$ for staying and $\frac{2}{3}$ for switching.

Let us plot the win rates with a bar chart. The black lines on each bar show the 95% confidence interval.

```
sns.catplot(
    data=df_class,
    x='Choice',
    y='Won',
    kind='bar',
    errorbar=('ci', 95)
)
plt.suptitle(f'Classroom Fraction of Winners ({df_class.shape[0]} Games)', y=1.05)
plt.ylabel('Fraction of Wins')
plt.show()
```



Simulated Data

We can simulate as many plays of the game as we want! The code below simulates 10,000 plays of the game. For each play, we calculate whether switching or staying wins. So, for every play, we have both stay and switch outcomes

```
trials = 10_000
doors = [1, 2, 3]
doors_set = set(doors)

random.seed(42)
```

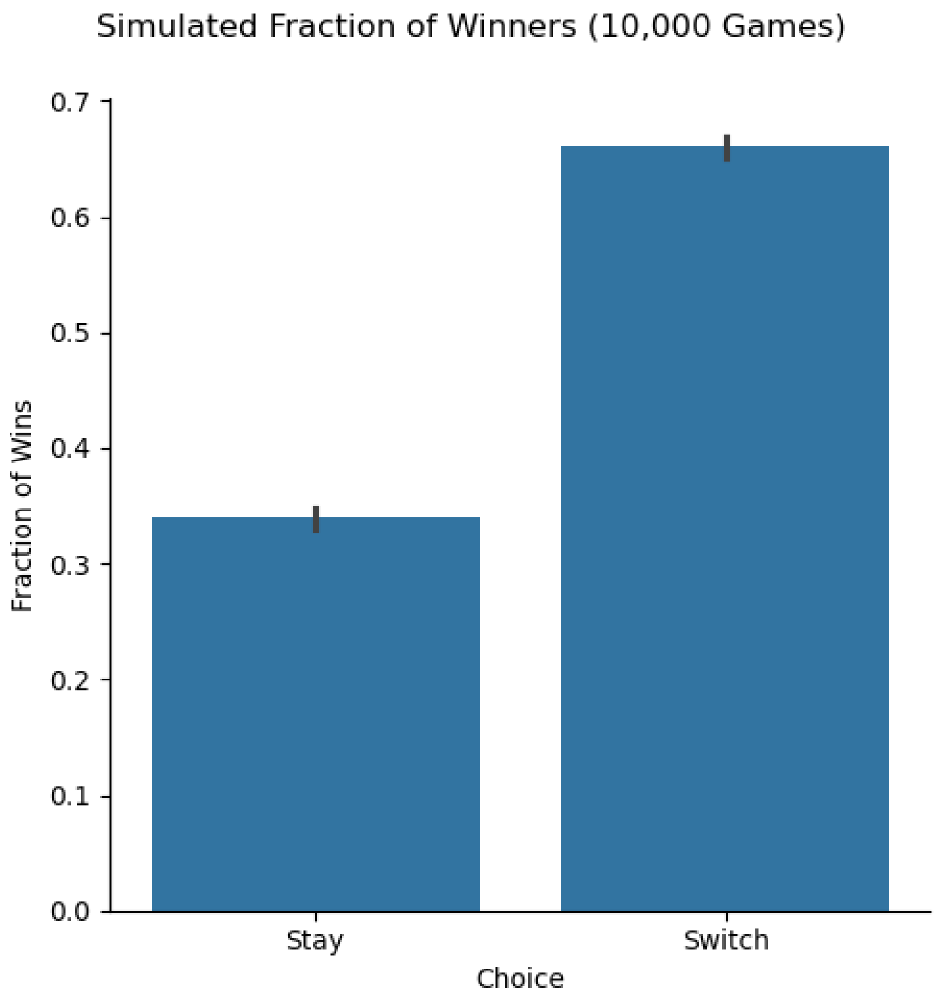
```
df_sim = (
    pd.DataFrame(
        data={
            'Winner': [random.choice(doors) for _ in range(trials)],
            'Choice_1': [random.choice(doors) for _ in range(trials)],
            'Open': np.nan,
            'Choice_2': np.nan,
            'Stay': np.nan,
            'Switch': np.nan
        },
        index=pd.RangeIndex(stop=trials, name='Simulation')
    )
    .assign(
        Open=lambda x: [(doors_set - {w, c}).pop() for w, c in x[['Winner', 'Choice_1']].ite
        Choice_2=lambda x: [(doors_set - {c, o}).pop() for c, o in x[['Choice_1', 'Open']].i
        Stay=lambda x: x['Choice_1'] == x['Winner'],
        Switch=lambda x: x['Choice_2'] == x['Winner']
    )
    .melt(
        value_vars=['Switch', 'Stay'],
        var_name='Choice',
        value_name='Won')
)
```

```
(
    df_sim
    .groupby(by='Choice')
    [['Won']]
    .mean()
    .round(4)
)
```

Choice	Won
Stay	0.3396
Switch	0.6604

With 10,000 simulations, these results are much more reliable than our class data. They confirm that switching is the better strategy!

```
sns.catplot(
    data=df_sim.sort_values(by='Choice'),
    x='Choice',
    y='Won',
    kind='bar',
    errorbar=('ci', 95)
)
plt.suptitle(f'Simulated Fraction of Winners ({trials:,.0f} Games)', y=1.05)
plt.ylabel('Fraction of Wins')
plt.show()
```



Conclusion

We explored the Monty Hall problem in four different ways, and they all point to the same conclusion:

- **Intuition:** Switching transfers the $\frac{2}{3}$ probability from the other two doors to the remaining door.
- **Bayes' Theorem:** The probability of winning by switching is $\frac{2}{3}$.
- **Class Data:** Our 26 games showed that switchers won 57% of the time, compared to 25% for stayers.
- **Simulation:** Over 10,000 simulated games, switchers won 67% of the time, compared to 33% for stayers.

All these methods agree: you should always switch in the Monty Hall problem to maximize your chance of winning the car, giving you a $\frac{2}{3}$ probability of success. Staying with your original choice only gives you a $\frac{1}{3}$ chance. So, next time you're on a game show (or playing an envelope game in class), remember to switch!

The Monty Hall Problem - Sec 04

```
import io
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import random
import seaborn as sns
```

Introduction

The Monty Hall problem is a probability puzzle from *Let's Make a Deal*. You pick one of three doors (car behind one, goats behind two), the host reveals a goat, and you decide to switch or stay. We played this in class with envelopes. This problem is often used in quantitative finance interviews because it tests your ability to think about probabilities (see Crack (2024)). This notebook explores the Monty Hall problem four ways:

1. **Intuition:** We will use simple logic to understand why switching might be better.
2. **Bayes' Theorem:** We will use a mathematical rule to calculate the exact probabilities.
3. **Class Data:** We will analyze the results from our envelope game in class.
4. **Simulated Data:** We will use Python to simulate the game thousands of times to see the results.

We will show that switching wins $\frac{2}{3}$ of the time, staying wins $\frac{1}{3}$. I based this notebook on Crack (2024), Kritzman (2000), and the Monty Hall problem Wikipedia page.

Intuition

Let us solve the Monty Hall problem with simple logic. We will use the same setup as in the introduction: You pick Door 1, the host opens Door 3 (revealing a goat), and you decide whether to switch to Door 2. Here are the steps in the intuitive solution:

1. **Initial Choice:** There are three doors, and the car is equally likely to be behind any one of them. So, when you pick Door 1, the probability that the car is behind Door 1 is $\frac{1}{3}$. That means there is a $\frac{2}{3}$ chance the car is behind one of the other two doors (Door 2 or Door 3 combined).
2. **Host Opens a Door:** The host, who knows where the car is, opens Door 3 and shows it has a goat. Now we know Door 3 does not have the car.
3. **Probability Transfer:** Before the host opened Door 3, there was a $\frac{2}{3}$ chance the car was behind Door 2 or Door 3. Since Door 3 has a goat, that entire $\frac{2}{3}$ probability now belongs to Door 2 (the door you can switch to). The probability that the car is behind Door 1 (your original choice) is still $\frac{1}{3}$, because nothing has changed about Door 1.

In conclusion:

- If you **switch** to Door 2, your probability of winning is $\frac{2}{3}$.
- If you **stay** with Door 1, your probability of winning is $\frac{1}{3}$.

So, you should switch to Door 2 to have a better chance of winning the car!

Bayes' Theorem

We can also solve the Monty Hall problem using a mathematical rule called Bayes' Theorem. Bayes' Theorem helps us calculate the probability of something happening based on new information. Here, we will use it to find the probability that the car is behind Door 2, given that you picked Door 1 and the host opened Door 3 (revealing a goat).

Step 1: Define the Events

- Let B_2 be the event that the car is behind Door 2 (the door you can switch to).
- Let H_3 be the event that the host opens Door 3 (and shows a goat).

We want to find $P(B_2|H_3)$, which is the probability that the car is behind Door 2, given that the host opened Door 3.

Step 2: Bayes' Theorem Formula

Bayes' Theorem says:

$$P(B_2|H_3) = \frac{P(H_3|B_2) \times P(B_2)}{P(H_3)}$$

Let us break down each part:

- $P(H_3|B_2)$: The probability that the host opens Door 3, given that the car is behind Door 2. You picked Door 1, so the host must open a door that's not Door 2 (the car) and not Door 1 (your choice). The only option is Door 3, so $P(H_3|B_2) = 1$.
- $P(B_2)$: The probability that the car is behind Door 2 before we know anything else. Since the car is equally likely to be behind any of the three doors, $P(B_2) = \frac{1}{3}$.
- $P(H_3)$: The total probability that the host opens Door 3. This is trickier, so let's calculate it carefully.

Step 3: Calculate $P(H_3)$

We can define $P(H_3)$ as follows:

$$P(H_3) = [P(H_3|B_1) \times P(B_1)] + [P(H_3|B_2) \times P(B_2)] + [P(H_3|B_3) \times P(B_3)]$$

The host can open Door 3 in three scenarios, depending on where the car is:

- **Car behind Door 1 (B_1):** Probability $P(B_1) = \frac{1}{3}$. You picked Door 1, so the host must open a door that's not Door 1 (your choice). The host randomly picks a door, so $P(H_2|B_1) = P(H_3|B_1) = \frac{1}{2}$.
- **Car behind Door 2 (B_2):** Probability $P(B_2) = \frac{1}{3}$. We calculate above that $P(H_3|B_2) = 1$.
- **Car behind Door 3 (B_3):** Probability $P(B_3) = \frac{1}{3}$. But if Door 3 has the car, the host cannot open Door 3, and $P(H_3|B_3) = 0$.

Therefore, we can calculate $P(H_3)$ as follows:

$$P(H_3) = (\frac{1}{2} \times \frac{1}{3}) + (1 \times \frac{1}{3}) + (0 \times \frac{1}{3}) = \frac{1}{6} + \frac{1}{3} + 0 = \frac{1}{2}$$

Step 4: Apply Bayes' Theorem

Now we can plug everything into the formula:

$$P(B_2|H_3) = \frac{P(H_3|B_2) \times P(B_2)}{P(H_3)} = \frac{1 \times \frac{1}{3}}{\frac{1}{2}} = \frac{\frac{1}{3}}{\frac{1}{2}} = \frac{1}{3} \times \frac{2}{1} = \frac{2}{3}$$

This confirms that switching doubles your odds.

Conclusion

The probability that the car is behind Door 2, given that the host opened Door 3, is $\frac{2}{3}$. This matches our intuition! You should switch to Door 2 to have a $\frac{2}{3}$ chance of winning, compared to a $\frac{1}{3}$ chance if you stay with Door 1.

Class Data

Let us look at the results from our envelope game in class. Here are the data:

```
data_class = """
Date,Section,Switched,Won
2025-02-07,2,0,1
2025-02-07,3,1,0
2025-02-07,4,0,0
2025-02-11,2,0,1
2025-02-11,3,0,0
2025-02-11,4,0,1
2025-02-18,2,0,0
2025-02-18,3,1,0
2025-02-18,4,0,0
2025-02-21,2,1,1
2025-02-21,3,1,1
2025-02-21,4,1,0
2025-03-14,2,0,0
2025-03-14,3,1,1
2025-03-14,4,1,1
2025-03-21,2,1,1
2025-03-21,3,1,0
2025-03-22,4,0,0
2025-04-01,2,0,0
2025-04-01,3,1,1
2025-04-01,4,0,0
2025-04-04,2,1,0
2025-04-04,3,1,0
2025-04-08,2,1,1
2025-04-08,3,1,1
2025-04-08,4,0,0
"""
```

```
df_class = (
    pd.read_csv(
        filepath_or_buffer=io.StringIO(data_class),
        parse_dates=['Date'],
        index_col=['Date']
    )
    .assign(Choice=lambda x: np.where(x['Switched']==1, 'Switch', 'Stay'))
)
```

We can calculate the fraction of times students won when they switched versus when they stayed. Let us use Python to do this.

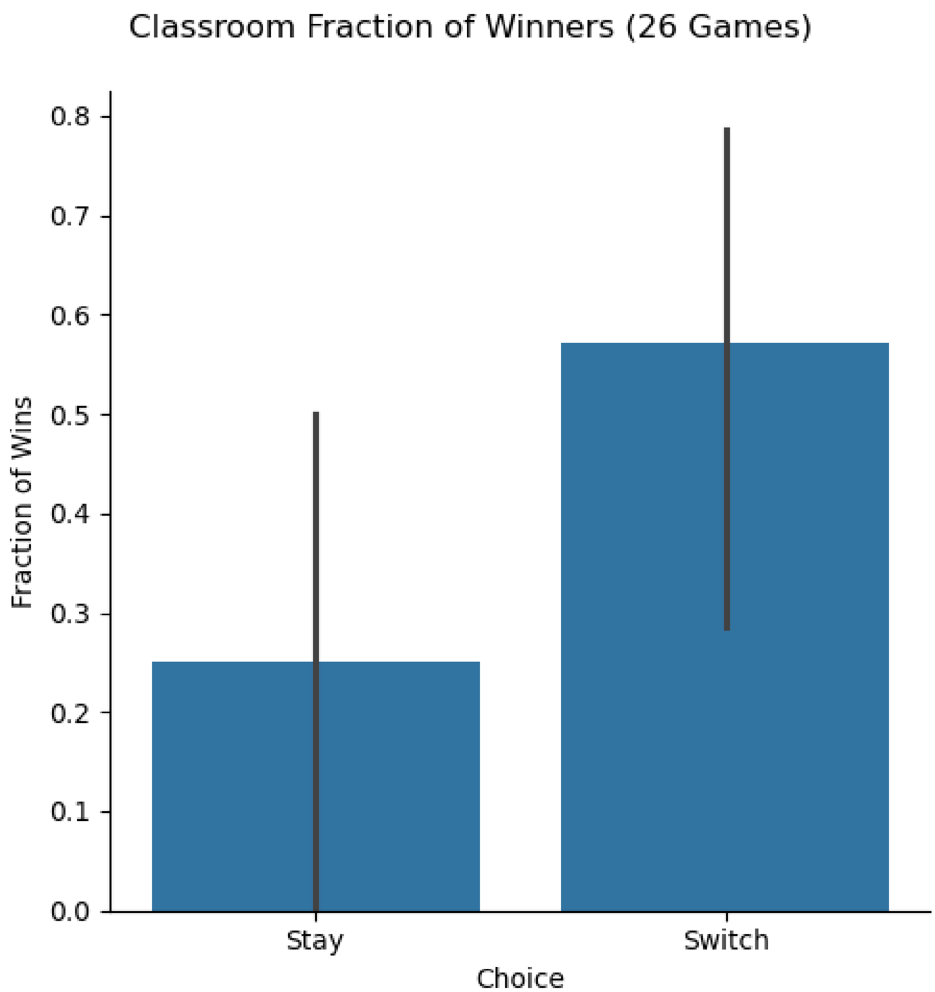
```
(
    df_class
    .groupby(by='Choice')
    [['Won']]
    .mean()
    .round(4)
)
```

	Won
Choice	
Stay	0.2500
Switch	0.5714

Our class data supports the idea that switching is better! Furthermore, these results are close to the theoretical probabilities we calculated: $\frac{1}{3}$ for staying and $\frac{2}{3}$ for switching.

Let us plot the win rates with a bar chart. The black lines on each bar show the 95% confidence interval.

```
sns.catplot(
    data=df_class,
    x='Choice',
    y='Won',
    kind='bar',
    errorbar=('ci', 95)
)
plt.suptitle(f'Classroom Fraction of Winners ({df_class.shape[0]} Games)', y=1.05)
plt.ylabel('Fraction of Wins')
plt.show()
```



Simulated Data

We can simulate as many plays of the game as we want! The code below simulates 10,000 plays of the game. For each play, we calculate whether switching or staying wins. So, for every play, we have both stay and switch outcomes

```
trials = 10_000
doors = [1, 2, 3]
doors_set = set(doors)

random.seed(42)
```

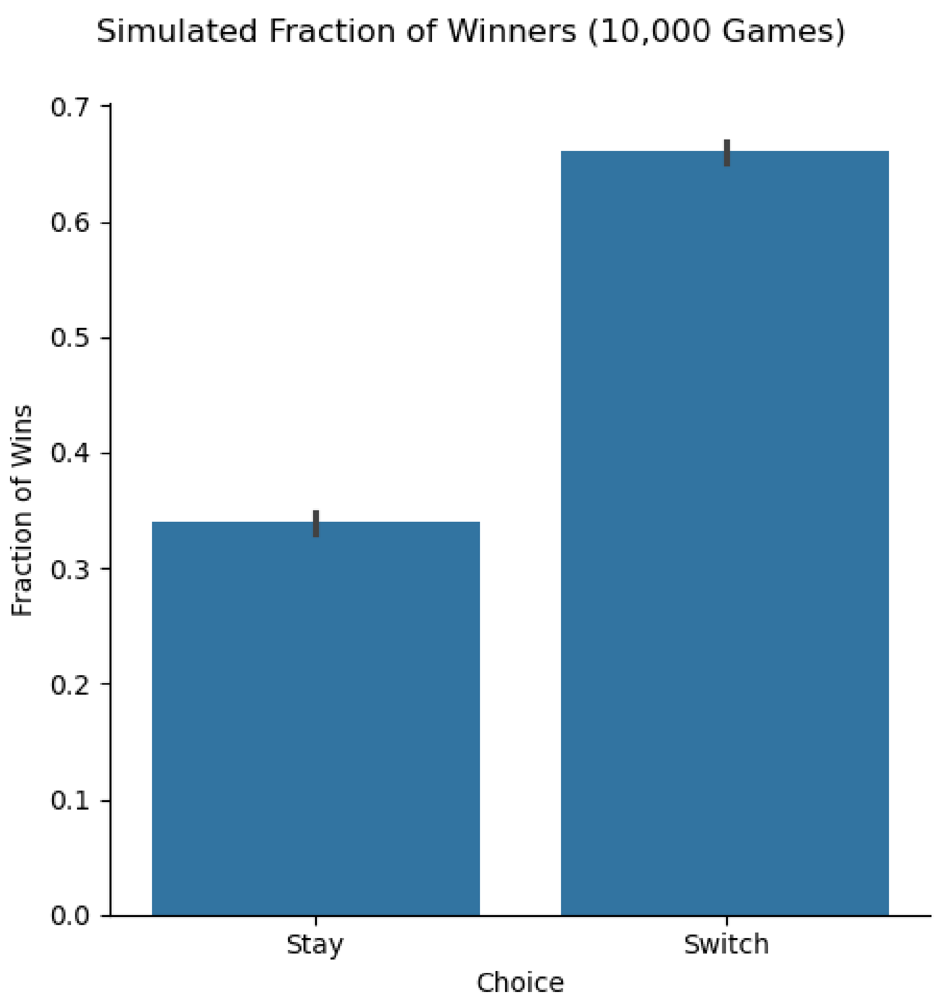
```
df_sim = (
    pd.DataFrame(
        data={
            'Winner': [random.choice(doors) for _ in range(trials)],
            'Choice_1': [random.choice(doors) for _ in range(trials)],
            'Open': np.nan,
            'Choice_2': np.nan,
            'Stay': np.nan,
            'Switch': np.nan
        },
        index=pd.RangeIndex(stop=trials, name='Simulation')
    )
    .assign(
        Open=lambda x: [(doors_set - {w, c}).pop() for w, c in x[['Winner', 'Choice_1']].ite
        Choice_2=lambda x: [(doors_set - {c, o}).pop() for c, o in x[['Choice_1', 'Open']].i
        Stay=lambda x: x['Choice_1'] == x['Winner'],
        Switch=lambda x: x['Choice_2'] == x['Winner']
    )
    .melt(
        value_vars=['Switch', 'Stay'],
        var_name='Choice',
        value_name='Won')
)
```

```
(
    df_sim
    .groupby(by='Choice')
    [['Won']]
    .mean()
    .round(4)
)
```

	Won
Choice	
Stay	0.3396
Switch	0.6604

With 10,000 simulations, these results are much more reliable than our class data. They confirm that switching is the better strategy!

```
sns.catplot(
    data=df_sim.sort_values(by='Choice'),
    x='Choice',
    y='Won',
    kind='bar',
    errorbar=('ci', 95)
)
plt.suptitle(f'Simulated Fraction of Winners ({trials:,.0f} Games)', y=1.05)
plt.ylabel('Fraction of Wins')
plt.show()
```



Conclusion

We explored the Monty Hall problem in four different ways, and they all point to the same conclusion:

- **Intuition:** Switching transfers the $\frac{2}{3}$ probability from the other two doors to the remaining door.
- **Bayes' Theorem:** The probability of winning by switching is $\frac{2}{3}$.
- **Class Data:** Our 26 games showed that switchers won 57% of the time, compared to 25% for stayers.
- **Simulation:** Over 10,000 simulated games, switchers won 67% of the time, compared to 33% for stayers.

All these methods agree: you should always switch in the Monty Hall problem to maximize your chance of winning the car, giving you a $\frac{2}{3}$ probability of success. Staying with your original choice only gives you a $\frac{1}{3}$ chance. So, next time you're on a game show (or playing an envelope game in class), remember to switch!

Week 14

MSFQ Assessment Exam

We will take the assessment exam in class on Tuesday of this week.

Week 15

Project 3

```
import matplotlib.pyplot as plt
import matplotlib.ticker as ticker
import numpy as np
import pandas as pd
```

Purpose

I have two goals for you for this project:

1. Implement, backtest, describe, and *explain* an investing strategy of your choice in Python
2. Investigate industry returns; several students have come to me interested in "sector rotation strategies"

Assignment

Use the 49 industry returns to build an investing strategy that maximizes the Sharpe ratio and/or total return. You may use any strategy, such as portfolio optimization, momentum, equal weights, or inverse volatility weights, among others. You may use short weights of up to 30% of your portfolio, including borrowing at the risk-free rate (i.e., `RF` from the Fama-French factors). ***You may not use future returns to assemble your current portfolio.*** That is, for each month, use only past data to set portfolio weights, not future returns.

However, you may backtest your strategy on the full sample, even though this backtesting creates a bias.[1] To reduce the impact of this bias, I will backtest all groups on the same random five-year sample.

For my backtest, your notebook must create monthly portfolio returns. Start these returns no later than five years into the sample to allow lookback periods for your strategy. Your

[1] To avoid this bias, I would give you anonymized data, you would automate your strategy into one function, and I would backtest your function output on a holdout sample. This process would be beyond the scope of this course.

notebook must export these returns to a CSV file named `Group_XX.csv`, where `XX` is your two-digit group number (e.g., `Group_01.csv` for Group 01). This CSV file must have only two columns:

1. `Date` with dates in the same format as the Fama-French files
2. `Group_XX` (e.g., `Group_01`) with your portfolio returns as decimals

Here is an example row in `Group_01.csv`: `Date,Group_01` with `1931-07,0.0025`.

> **Note**
>
> On Friday, 4/11, I added a function to the end of this notebook that you can use export your data for the competition.

I will pick a random five-year period to backtest your strategy and evaluate it based on its Sharpe ratio and total return. The highest Sharpe ratio will earn 15 points, and the highest total return will earn 15 points. Other groups get fewer points based on their Sharpe ratios and total returns relative to the top performers. The other 170 points come from your calculations, visualizations, discussions, and deliverables. See Section and Table 1 for details.

Other important considerations:

1. Use only the provided CSV files for industry and factor returns
2. Write up to a three-page summary of your strategy, including:
   1. A description of your strategy
   2. An explanation of why your strategy might have the highest Sharpe ratio and/or total return
   3. Figures and tables to support your description and explanation
3. Title, label, and caption your figures and tables, referencing them in your summary

Criteria

Table 1 provides the project grading rubric. The project is worth 200 points. The peer reviews are worth 100 points, and students will receive their median score. Almost all students earn perfect peer review scores, so I will factor that into project scores. For example, a project score *without peer review scores* of 77.5% converts to a project score *with perfect peer review scores* of 85% because $\frac{0.775 \times 200 + 1.00 \times 100}{300} = 0.85 = 85\%$.

Table 1: This table provides the project grading rubric

Topic	Points
Clarity, correctness, and completeness of calculations	50
Clarity, correctness, and completeness of visualizations	50
Clarity, correctness, and completeness of discussions	50
Highest Sharpe ratio in the five-year sample that I select	15
Highest total return in the five-year sample that I select	15
Correctness of submission according to the deliverables section	20
Total	200

Deliverables

Upload the following as unzipped files to Canvas by 11:59 PM on 4/23:

1. One Jupyter notebook that contains your report, performs *all* your analysis, and exports a CSV file with your strategy returns
    1. Name this notebook `project_3.ipynb` for me to run your code
    2. This notebook must run on my computer; I will place the data files in the same folder as your notebook
    3. Name your exported CSV file `Group_XX.csv`, where `XX` is your two-digit group number; details are in Section
2. One Quarto-generated Word document *with code* for me to grade
    1. *Unlike prior projects, submit only one Word document with code*
    2. Name this document `project_3.docx`
    3. Use the first cell in this notebook, with `echo: true`, to create this Word document with code
    4. You may not edit the Word documents after you create them

Here is some additional guidance:

1. Write up to a three-page summary at the start of your report, *which is the only writing I will read*
2. Your submission must not include your name

Data

This project requires two data files. Save these data files as-is in the same folder as your `project_3.ipynb` notebook file.

1. `data_ind49.csv` provides all available monthly returns from the 49 industries from Kenneth French's data library
2. `data_ff3.csv` provides all available monthly returns on the three factors from Kenneth French's data library

See `data_3.ipynb` if you want to see how I create these data files. You can read these data files as follows to remove missing values, recreate the `PeriodIndex` that `pdr.DataReader()` returns, and convert percent returns to decimal returns.

> **Note**
>
> Several students requested the other Fama-French factors to use a signals. I updated this notebook on Friday, 4/11, to provide two more data files:
>
> 1. `data_mom.csv` with the value-weighted monthly returns on the momentum factor
> 2. `data_ff5.csv` with the monthly Fama-French *five* factors, which only begin in 1963-07 because they require accounting data

```
ind49 = (
    pd.read_csv(
        filepath_or_buffer='data_ind49.csv',
        parse_dates=['Date'],
        na_values=[-99.99, -999]
    )
    .assign(Date=lambda x: x['Date'].dt.to_period('M'))
    .set_index('Date')
    .sort_index()
    .rename_axis(columns=['Industry'])
    .div(100)
)
```

```
ind49.iloc[:5, :5]
```

Industry Date	Agric	Food	Soda	Beer	Smoke
1926-07	0.0237	0.0012	NaN	-0.0519	0.0129
1926-08	0.0223	0.0268	NaN	0.2703	0.0650
1926-09	-0.0057	0.0158	NaN	0.0402	0.0126
1926-10	-0.0046	-0.0368	NaN	-0.0331	0.0106
1926-11	0.0675	0.0626	NaN	0.0729	0.0455

```
ff3 = (
    pd.read_csv(
        filepath_or_buffer='data_ff3.csv',
        parse_dates=['Date'],
        na_values=[-99.99, -999]
    )
    .assign(Date=lambda x: x['Date'].dt.to_period('M'))
    .set_index('Date')
    .rename_axis(columns=['Factor'])
    .div(100)
)
```

```
ff3.head()
```

Factor Date	Mkt-RF	SMB	HML	RF
1926-07	0.0296	-0.0256	-0.0243	0.0022
1926-08	0.0264	-0.0117	0.0382	0.0025
1926-09	0.0036	-0.0140	0.0013	0.0023
1926-10	-0.0324	-0.0009	0.0070	0.0032
1926-11	0.0253	-0.0010	-0.0051	0.0031

```
mom = (
    pd.read_csv(
        filepath_or_buffer='data_mom.csv',
        parse_dates=['Date'],
        na_values=[-99.99, -999]
    )
    .assign(Date=lambda x: x['Date'].dt.to_period('M'))
    .set_index('Date')
    .rename_axis(columns=['Factor'])
    .div(100)
)
```

```
mom.head()
```

Factor Date	Mom
1927-01	0.0036
1927-02	-0.0214
1927-03	0.0361
1927-04	0.0430
1927-05	0.0300

```
ff5 = (
    pd.read_csv(
        filepath_or_buffer='data_ff5.csv',
        parse_dates=['Date'],
        na_values=[-99.99, -999]
    )
    .assign(Date=lambda x: x['Date'].dt.to_period('M'))
    .set_index('Date')
    .rename_axis(columns=['Factor'])
    .div(100)
)
```

```
ff5.head()
```

Factor Date	Mkt-RF	SMB	HML	RMW	CMA	RF
1963-07	-0.0039	-0.0041	-0.0097	0.0068	-0.0118	0.0027
1963-08	0.0507	-0.0080	0.0180	0.0036	-0.0035	0.0025
1963-09	-0.0157	-0.0052	0.0013	-0.0071	0.0029	0.0027
1963-10	0.0253	-0.0139	-0.0010	0.0280	-0.0201	0.0029
1963-11	-0.0085	-0.0088	0.0175	-0.0051	0.0224	0.0027

Quarto

Basics

1. Use Quarto to generate your Word document from your notebook
2. Use `#` to create a title and `##` to create sections
3. Use `-` or `1.` to create lists

4. Use the first cell in this notebook to hide or display code with `echo=false` or `echo=true`, respectively
5. This first cell must be a `raw` cell instead of a `code` or `markdown` cell
6. Use `quarto render project_1.ipynb` in the same folder as your notebook to render it to a Word document
7. Use the `cd` command in the terminal to change the working directory to the directory with your notebook

Examples

This section provides a sample analysis highlighting how code and formatting work with Quarto. Figure 1 provides a line plot of the value of a $10,000 investment in the Smoking industry. Note that `#| label:` and `#| fig-cap:` comments at the top of the figure cell create the figure reference/link and the figure caption, respectively. You can learn more about cross-referencing figures and tables here.

```
(
    ind49
    ['Smoke']
    .add(1)
    .cumprod()
    .mul(10_000)
    .plot()
)
plt.semilogy()
plt.ylabel('Value ($)')
plt.title(f'Value of a $10,000 investment in the smoking industry\n at the start of the Fama
plt.gca().yaxis.set_major_formatter(ticker.FuncFormatter(lambda x, p: format(int(x), ',')))

plt.show()
```

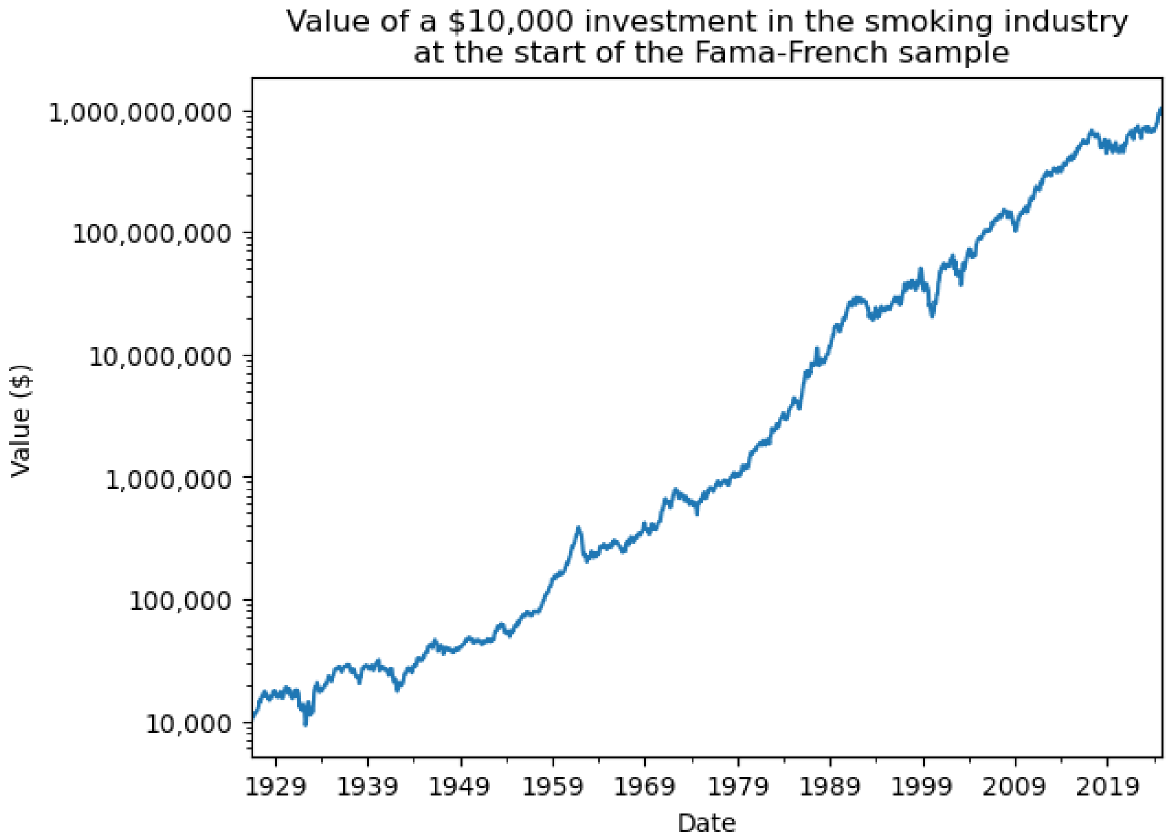


Figure 1: This line plot shows the value of a $10,000 investment in the smoking industry at the start of the Fama-French sample

Artificial Intelligence (AI)

You may use AI (e.g., ChatGPT) to *help* you prepare your analysis and discussion. However:

1. AI will not do very well on this project without significant input from your team
2. AI will not be a defense against plagiarism because AI should not *write* your code and slides; If you plagiarize an AI that plagiarizes other sources, you are responsible for plagiarizing the AI and its sources

Function to Prepare Data File

Use this function to export your data file for the competition.

```
def write_csv(portfolio_returns, group_number):
    """
    Write portfolio returns to a CSV file named Group_XX.csv with Date as YYYY-MM.

    Parameters:
    - portfolio_returns: pandas Series or DataFrame with monthly returns (as decimals) and Da
    - group_number: int, your two-digit group number (e.g., 1 for Group_01)

    Output:
    - Saves Group_XX.csv with two columns: Date (YYYY-MM) and Group_XX (returns as decimals)
    """

    # Make two-digit group number
    group_str = f"{group_number:02d}"
    filename = f"Group_{group_str}.csv"

    # Convert returns to DataFrame if Series
    if isinstance(portfolio_returns, pd.Series):
        df = portfolio_returns.to_frame(name=f"Group_{group_str}")
    else:
        df = portfolio_returns.rename(columns={portfolio_returns.columns[0]: f"Group_{group_s

    # Convert index to PeriodIndex with monthly frequency and format as YYYY-MM
    if isinstance(df.index, pd.PeriodIndex):
        if df.index.freq != 'M':
            df.index = df.index.to_timestamp().to_period('M')
    elif isinstance(df.index, pd.DatetimeIndex):
        df.index = df.index.to_period('M')
    else:
        raise ValueError("Index must be a datetime or PeriodIndex")

    # Rename index to Date
    df.index.name = "Date"

    # Write to CSV
    df.to_csv(filename, index=True)

    print(f"Saved portfolio returns to {filename}")
```

```
# Example usage:
dates = pd.date_range(start="1931-07-01", end="1936-06-01", freq="MS")
returns = pd.Series([0.0025, 0.01, -0.005] + [0.001] * (len(dates) - 3), index=dates)
```

```
write_csv(returns, 1)
```

```
Saved portfolio returns to Group_01.csv
```

```
group_01 = (
    pd.read_csv(
        filepath_or_buffer='Group_01.csv',
        parse_dates=['Date'],
        na_values=[-99.99, -999]
    )
    .assign(Date=lambda x: x['Date'].dt.to_period('M'))
    .set_index('Date')
    .rename_axis(columns=['Team'])
    .div(100)
)
```

```
group_01.head()
```

Team Date	Group_01
1931-07	0.000025
1931-08	0.000100
1931-09	-0.000050
1931-10	0.000010
1931-11	0.000010

References

Black, Fischer and Myron Scholes (1973). “The pricing of options and corporate liabilities”. In: *Journal of Political Economy* 81.3, pp. 637–654.

Crack, Timothy Falcon (2024). *Heard on the street: quantitative questions from Wall Street job interviews*. Timothy Crack.

DeMiguel, Victor, Lorenzo Garlappi, and Raman Uppal (2009). “Optimal versus naive diversification: How inefficient is the 1/N portfolio strategy?” In: *The Review of Financial Studies* 22.5, pp. 1915–1953.

Fama, Eugene F. and Kenneth R. French (1993). “Common risk factors in the returns on stocks and bonds”. In: *Journal of Financial Economics* 33.1, pp. 3–56.

Gray, Wesley R. and Tobias E. Carlisle (2012). *Quantitative value: a practitioner’s guide to automating intelligent investment and eliminating behavioral errors*. John Wiley & Sons.

Kritzman, Mark P. (2000). *Puzzles of Finance: Six Practical Problems and Their Remarkable Solutions*. John Wiley & Sons.

Lewinson, Eryk (2020). *Python for Finance Cookbook: Over 50 recipes for applying modern Python libraries to financial data analysis*. Packt Publishing Ltd.

McKinney, Wes (2022). *Python for Data Analysis*. 3rd ed. O’Reilly Media, Inc.

Murphy, John J. (1999). *Technical analysis of the financial markets: A comprehensive guide to trading methods and applications*. Penguin.

Welch, Ivo (2022). *Corporate Finance*. 5th ed. Ivo Welch.