Data Analytics in Finance

FINA 6333 for 2024 Spring

Richard Herron

Table of contents

1

Welcome to FINA 6333 for Spring 2024 at the D'Amore-McKim School of Business at Northeastern University!

For each course topic, we will have one notebook for its lecture and one for its in-class practice exercises. I will maintain these notebooks here because Canvas does not render notebooks. I will maintain everything else on Canvas (e.g., announcements, discussions, grades, etc.).

You have three choices to download or run these notebooks:

1. Download them from our shared folder on OneDrive (or click the cloud icon in the left sidebar)
2. Download them from our Notebooks folder on GitHub (or click the octo-cat icon in the left sidebar)
3. Open them on Google Colab (or click the Google icon in the left sidebar)

Part I

Week 1

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

2.1 Introduction

We must understand the basics of Python before we can use it to analyze financial data. Chapter 2 of Wes McKinney's *Python for Data Analysis* 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 unless otherwise indicated. The section numbers here differ from McKinney because we will only discuss some topics.

2.2 Language Semantics

2.2.1 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.

So, spaces are more than cosmetic in Python. For non-Python programmers, white space is often Python's defining feature. Here is a for loop with an if block that shows how Python uses white space.

```
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]
```

Note: We will use f-string print statements wherever we can. These f-string print statements are easy to use, and I do not want to teach old approaches when the new ones are better.

2.2.2 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.

```
# 5 + 5
```

2.2.3 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.

We can write a function that adds two numbers.

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

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

```
10
```

We can write a function that adds two strings separated by a space.

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

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

```
'5 5'
```

What is the difference between `print()` and `return`? `print()` returns its arguments to the console or "standard output", whereas `return` returns its argument as an output we can assign to variables. In the example below, we use the `return` line to assign the output of `add_string_2()` to the variable `return_from_add_strings_2`. The `print()` line prints to the console or "standard output", but its output is not assigned or captured.

```
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)'
```

2.2.4 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]
```

```
a
```

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

If we assign `a` to a new variable `b`, both `a` and `b` refer to the *same* object. This same object is the list `[1, 2, 3]`. If we change `a`, we also change `b`, because these variables or names refer to the *same* object.

```
b = a
```

```
b
```

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

Variables `a` and `b` refer to the same object, a list `[1, 2, 3]`. We will learn more about lists (and tuples and dictionaries) in chapter 3 of McKinney.

```
a is b
```

```
True
```

If we modify `a` by appending a 4, we change `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 a 5, we change `a`, too!

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

```
a
```

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

```
b
```

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

The behavior is useful but a double-edged sword! Here is a deeper discussion of this behavior.

2.2.5 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.

In Python,

1. We do not declare variables and their types

2. We can change variables' types because variables are only names that refer to objects

Dynamic references mean we can reassign a variable to a new object in Python. For example, we can reassign `a` from a list to an integer to a string.

```
a
```

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

```
type(a)
```

```
list
```

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

```
int
```

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

```
str
```

Strong types mean Python typically will not convert object types. For example, the code returns either `'55'` as a string or `10` as an integer in many programming languages. However, `'5' + 5` returns an error in Python.

```
# '5' + 5
```

However, Python implicitly converts integers to floats.

```
a = 4.5
b = 2
print(f'a is {type(a)}, b is {type(b)}')
a / b
```

```
a is <class 'float'>, b is <class 'int'>
```

```
2.25
```

In the previous code cell:

1. The 'a is ...' output prints because of the explicit `print()` function call
2. The output of `a / b` prints (or displays) because it is the last line in the code cell

If we want integer division (or floor division), we have to use `//`.

```
5 // 2
```

```
2
```

```
5 / 2
```

```
2.5
```

2.2.6 Attributes and methods

We can use tab completion to list attributes (characteristics stored inside objects) and methods (functions associated with objects).

```
a = 'foo'
```

```
a.capitalize()
```

```
'Foo'
```

2.2.7 Imports

In Python a module is simply a file with the .py extension containing Python code.

We can import with `import` statements, which have several syntaxes. The basic syntax uses the module name as the prefix to separate module items from our current namespace.

```
import pandas
```

The `import as` syntax lets us define an abbreviated prefix.

```
import pandas as pd
```

We can also import one or more items from a package into our namespace with the following syntaxes.

```
from pandas import DataFrame
```

```
from pandas import DataFrame as df
```

2.2.8 Binary operators and comparisons

Binary operators work like Excel.

```
5 - 7
```

```
-2
```

```
12 + 21.5
```

```
33.5
```

```
5 <= 2
```

```
False
```

We can operate during an assignment to avoid two names referring to the same object.

```
a = [1, 2, 3]
b = a
c = list(a)
```

```
a is b
```

```
True
```

```
a is c
```

```
False
```

Here `a` and `c` have the same *values* but are not the same object!

```
a == c
```

```
True
```

```
a is not c
```

```
True
```

In Python, `=` is the assignment operator, `==` tests equality, and `!=` tests inequality.

```
a == c
```

```
True
```

```
a != c
```

```
False
```

`a` and `c` have the same values but reference different objects in memory.

Table 2-1 from McKinney 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

2.2.9 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.

Lists are mutable, so we can modify them.

```
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)
```

Tuples are *immutable*, so we cannot modify them.

```
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'
```

Note: Tuples do not require `()`, but `()` improve readability.

```
test = 1, 2, 3
```

```
type(test)
```

```
tuple
```

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

2.3 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 lists 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

2.3.1 Numeric types

In Python, integers are unbounded, and `**` raises numbers to a power. So, `ival ** 6` is 17239781^6.

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

```
26254519291092456596965462913230729701102721
```

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 have to use `//` if we want C-style integer division (i.e., $3/2 = 1$).

```
3 // 2
```

```
1
```

2.3.2 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.

Python is case sensitive, so we must type Booleans as `True` and `False`.

```
True and True
```

```
True
```

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

```
True
```

```
False or True
```

```
True
```

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

```
True
```

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

```
True & True
```

```
True
```

```
False | True
```

```
True
```

2.3.3 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
```

Zero is Boolean `False`, and all other values are Boolean `True`.

```
bool(0)
```

```
False
```

```
bool(1)
```

```
True
```

```
bool(-1)
```

```
True
```

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

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

```
10
```

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

```
'55'
```

2.3.4 None

In Python, `None` is null. `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
```

2.4 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`.

2.4.1 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 previous code cell, we use double quotes to differentiate between the enclosing quotes and the apostrophe in `"It's"`.

Python's `elif` avoids Excel's nested `if()`s. `elif` continues an `if` block, and `else` runs if the other conditions are not 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`.

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

```
Made it
```

2.4.2 for loops

We use `for` loops to loop over a collection, like a list or tuple. The `continue` keyword skips the remainder of the `if` block for that loop iteration.

The following example assigns values with `+=`, where `a += 5` is an abbreviation for `a = a + 5`. There are equivalent abbreviations for subtraction, multiplication, and division (`-=`, `*=`, and `/=`).

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

```
total
```

```
12
```

The `break` keyword exits the loop altogether.

```
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
```

2.4.3 range

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

- With one argument, `range()` creates an iterator from 0 to that number *but excludes that number* (so `range(10)` is an interator with a length of 10 that starts at 0)
- With two arguments, the first argument is the *inclusive* starting value, and the second argument is the *exclusive* ending 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]
```

```
list(range(1, 10))
```

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

```
list(range(1, 10, 1))
```

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

```
list(range(0, 20, 2))
```

```
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

Python intervals are "closed" (inclusive) on the left and "open" (exclusive) 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]
```

For loops have the following syntax in many other programming languages.

```
seq = [1, 2, 3, 4]
for i in range(len(seq)):
    val = seq[i]
```

However, in Python, we can directly loop over the list `seq`. The following code cell is equivalent to the previous code cell and more "Pythonic".

```
for i in seq:
    val = i
```

2.4.4 Ternary expressions

We said above that Python `if` and `else` is cumbersome relative to Excel's `if()`. We can complete simple comparisons on one line in Python.

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

```
'Non-negative'
```

3 McKinney Chapter 2 - Practice for Section 04

3.1 Announcements

3.2 Practice

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

Try 20080915.

3.2.2 Use your answer above to write a function `date` that accepts an integer 8-digit date argument and returns a tuple of the year, month, and date (e.g., `return (year, month, date)`).

3.2.3 Rewrite `date` to accept an 8-digit date as an integer or string.

3.2.4 Finally, rewrite `date` to accept a list of 8-digit dates as integers or strings.

Return a list of tuples of year, month, and date.

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

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

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

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

3.2.9 FizzBuzz

Write a for loop that prints the numbers from 1 to 100. For multiples of three print "Fizz" instead of the number. For multiples of five print "Buzz". For numbers that are multiples of both three and five print "FizzBuzz". More here.

3.2.10 Use ternary expressions to make your FizzBuzz code more compact.

3.2.11 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
```

3.2.12 Two Sum

Write a function `two_sum` that does the following.

Given an array 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.

3.2.13 Best Time

Write a function `best_time` that solves the following.

You are given an array `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.

Part II

Week 2

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

4.1 Introduction

We must understand Python's core functionality to fully use NumPy and pandas. Chapter 3 of Wes McKinney's *Python for Data Analysis* discusses Python's core functionality. We will focus on the following:

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

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

4.2 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.

4.2.1 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 the parentheses are not always required.

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

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

```
tup[0]
```

```
4
```

```
tup[1]
```

```
5
```

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

Python is zero-indexed!

```
nested_tup[0]
```

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

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

```
4
```

```
tup = tuple('string')
```

```
tup
```

```
('s', 't', 'r', 'i', 'n', 'g')
```

```
tup[0]
```

```
's'
```

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

```
tup
```

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

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

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

```
tup
```

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

```
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 the output of `2 * tup`.

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

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

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

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

4.2.1.1 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
```

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

We can unpack nested tuples!

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

4.2.1.2 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
```

4.2.2 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)
```

Pyhon is zero-indexed!

```
a_list[0]
```

```
2
```

4.2.2.1 Adding and removing elements

Elements can be appended to the end of the list with the append method.

The `.append()` method appends an element to the list *in place* without reassigning the list.

```
b_list.append('dwarf')
```

Using insert you can insert an element at a specific location in the list. The insertion index must be between 0 and the length of the list, inclusive.

```
b_list.insert(1, 'red')
```

```
b_list.index('red')
```

```
1
```

```
b_list[b_list.index('red')] = 'blue'
```

The inverse operation to insert is pop, which removes and returns an element at a particular index.

```
b_list.pop(2)
```

```
'bar'
```

```
b_list
```

```
['foo', 'blue', 'baz', 'dwarf']
```

Note that `.pop(2)` removes the 2 element. If we do not want to remove the 2 element, we should use `[2]` to access an element without removing it.

> Elements can be removed by value with remove, which locates the first such value and removes it from the list.

```
b_list.append('foo')
```

```
b_list.remove('foo')
```

```
'dwarf' in b_list
```

```
True
```

```
'dwarf' not in b_list
```

```
False
```

4.2.2.2 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)])
```

> 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)])
```

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

4.2.2.3 Sorting

You can sort a list in-place (without creating a new object) by calling its sort function.

```
a = [7, 2, 5, 1, 3]
a.sort()
```

sort has a few options that will occasionally come in handy. One is the ability to pass a secondary sort key—that is, a function that produces a value to use to sort the objects. For example, we could sort a collection of strings by their lengths.

Before you write your own solution to a problem, read the docstring (help file) of the built-in function. The built-in function may already solve your problem faster with fewer bugs.

```
b = ['saw', 'small', 'He', 'foxes', 'six']
b.sort()
```

Python is case sensitive, so "He" sorts before "foxes"!

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

4.2.2.4 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. The necessary 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
```

```
seq[:5]
```

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

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

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

```
seq[3:5]
```

```
[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.

```
seq[:5]
```

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

```
seq[3:]
```

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

Negative indices slice the sequence relative to the end.

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

```
[1]
```

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

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

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

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

```
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]
```

I remember the trick above as `:2` is "count by 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 it is worth a few minutes to understand the examples above.

4.2.3 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 names, 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'
```

```
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.

```
d1['b']
```

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

```
'b' in d1
```

```
True
```

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'])
```

You can merge one dict into another using the update method.

```
d1
```

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

```
d1.update({'b': 'foo', 'c': 12})
```

```
d1
```

```
{'a': 'some value', 'b': 'foo', 7: 'an integer', 'c': 12}
```

4.3 List, Set, and Dict Comprehensions

We will focus on list comprehensions.

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.

List comprehensions are very Pythonic.

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

We could use a for loop to capitalize the strings in `strings` and keep only strings with lengths greater than two.

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

caps
```

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

A list comprehension is a more Pythonic solution and replaces four lines of code with one. The general format for 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. Write a for-loop and the 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]
```

4.4 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.

Here is the basic syntax for a function:

```
def mult_by_two(x):
    return 2*x
```

4.4.1 Returning Multiple Values

We can write Python functions that return multiple objects. In reality, the function `f()` below returns one object, a 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
```

4.4.2 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 very Pythonic and let us to write simple functions on the fly. For example, we could use a lambda function to sort `strings` by the number of unique letters.

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

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

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

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

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

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

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

How can I sort by the *second* letter in each string?

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

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

5 McKinney Chapter 3 - Practice for Section 04

5.1 Announcements

5.2 Practice

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

```
a = 1
```

```
b = 2
```

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

```
a = 1
```

```
b = 2
```

5.2.3 What is the output of the following code and why?

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

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

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

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

Name this list `l2`.

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

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

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

5.2.9 Use a lambda function to sort `strings` by the last letter.

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

5.2.10 Given an integer array `nums` and an integer `k`, 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.

5.2.11 Given an integer array `nums` and an integer `k`, 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.

5.2.12 Test whether the given strings are palindromes.

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

5.2.13 Write a function `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]
```

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

5.2.15 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]`

```
numbers = [18.5, 17.0, 18.0, 19.0, 18.0]
```

Part III

Week 3

6 McKinney Chapter 4 - NumPy Basics: Arrays and Vectorized Computation

6.1 Introduction

Chapter 4 of Wes McKinney's *Python for Data Analysis* 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 unless otherwise indicated. The section numbers here differ from McKinney because we will only discuss some topics.

The typical abbreviation for NumPy is `np`.

```
import numpy as np
```

The "magic" function `%precision 4` tells JupyterLab to print NumPy arrays to 4 decimals. This magic function only changes the printed precision and does not change the stored precision of the underlying values.

```
%precision 4
```

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

McKinney thoroughly discuesses the history on NumPy, as well as its technical advantages. But here is a simple illustration of the speed and syntax advantages of NumPy of Python's built-in data structures. First, we create a list and 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])
```

If we want to double each value in `my_list` we have to use a for loop or a list comprehension.

```
len(my_list * 2) # concatenates two copies of my_list
```

```
2000000
```

```
# [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.

```
my_arr * 2
```

```
array([      0,       2,       4, ..., 1999994, 1999996, 1999998])
```

We can use the "magic" function `%timeit` to time these two calculations.

```
%timeit [x * 2 for x in my_list]
```

```
56.1 ms ± 5.42 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

```
%timeit my_arr * 2
```

```
872 µs ± 64.5 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

The NumPy version is a hundred times faster than the list version. The NumPy version is also faster to type, read, and troubleshoot, which are typically more important. Our time is more valuable than the computer time!

6.2 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.

We generate random data to explore NumPy arrays. Whenever we generate random data, we should set the random number seed with `np.random.seed(42)`, which makes our random numbers repeatable. If we use the same random number seed, our random numbers will be the same.

```
np.random.seed(42)
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). To achieve this common-sense behavior, NumPy arrays must contain homogeneous data types (e.g., all floats or all integers).

```
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 Jupyter Notebooks provides tab completion.

```
data.ndim
```

```
2
```

```
data.shape
```

```
(2, 3)
```

```
data.dtype
```

```
dtype('float64')
```

We access or slice elements in a NumPy array using `[]`, the same as we slice lists and tuples.

```
data[0]
```

```
array([ 0.4967, -0.1383,  0.6477])
```

As with list and tuples, we can chain `[]`s.

```
data[0][0]
```

```
0.4967
```

However, with NumPy arrays. we can replace n chained `[]`s with one pair of `[]`s containing n values separated by commas. For example, `[i][j]` becomes `[i, j]`, `[i][j][k]` becomes `[i, j, k]`. This abbreviated notation is similar to what you see in your math and econometrics courses.

```
data[0, 0] # zero row, zero column
```

```
0.4967
```

```
data[0][0] == data[0, 0]
```

```
True
```

6.2.1 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()` casts the values in `data1` to floats becuase NumPy arrays must have homogenous data types. We could coerce these values to integers but would lose information.

```
np.array(data1, dtype=np.int64)
```

```
array([6, 7, 8, 0, 1], dtype=int64)
```

We can coerce or 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.ndim
```

```
2
```

```
arr2.shape
```

```
(2, 4)
```

```
arr2.dtype
```

```
dtype('int32')
```

There are several other ways to create NumPy arrays.

```
np.zeros(10)
```

```
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

```
np.zeros((3, 6))
```

```
array([[0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0.]])
```

The `np.arange()` function is similar to Python's built-in `range()` but creates an array directly.

```
list(range(15))
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
```

```
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 lists some 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)

6.2.2 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.]])
```

```
arr.shape
```

```
(2, 3)
```

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 an array!

```
2 ** arr
```

```
array([[ 2.,  4.,  8.],
       [16., 32., 64.]])
```

6.2.3 Basic Indexing and Slicing

One-dimensional array index and slice the same as lists.

```
arr = np.arange(10)
arr
```

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

```
arr[5]
```

```
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 have to jump through some hoops if we want to replace elements 5, 6, and 7 in `equiv_list` with the value 12.

```
# 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 the NumPy array `arr`!

```
arr[5:8] = 12
arr
```

```
array([ 0,  1,  2,  3,  4, 12, 12, 12,  8,  9])
```

"Broadcasting" is the name for this behavior.

> 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])
```

```
x = arr_slice
x
```

```
array([12, 12, 12])
```

```
x is arr_slice
```

```
True
```

```
y = x.copy()
```

```
y is arr_slice
```

```
False
```

```
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[:] = 2001
arr_slice_2
```

```
array([2001, 2001, 2001])
```

```
arr
```

```
array([ 0,  1,  2,  3,  4, 64, 64, 64,  8,  9])
```

6.2.4 Indexing with slices

We can slice across two or more dimensions, including 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]])
```

Slicing multi-dimension arrays is tricky! ***Always check your output!***

6.2.5 Boolean Indexing

We can use Booleans (`True`s and `False`s) 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 < 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]])
```

```
  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]])
```

6.3 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.

```
arr = np.arange(10)
arr
```

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

```
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.

```
2**arr
```

```
array([  1,   2,   4,   8,  16,  32,  64, 128, 256, 512], dtype=int32)
```

`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])
```

The functions above accept one argument. 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])
```

Be careful! Function names are not the whole story. Check your output and read the docstring! For example, `np.max()` returns the maximum of an array, instead of the elementwise maximum of two arrays for `np.maximum()`.

```
np.max(x)
```

```
1.5792
```

Table 4-4 from McKinney lists some fast, element-wise unary functions:

- `abs`, `fabs`: Compute the absolute value element-wise for integer, oating-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 thatnumber)
- `floor`: Compute the oor 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).

Table 4-5 from McKinney lists some 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 & |, ^)

6.4 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.

6.4.1 Expressing Conditional Logic as Array Operations

> The numpy.where function is a vectorized version of the ternary expression x if condition else y.

NumPy's `where()` is an if-else statement that operates 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 the list comprehension is 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])
```

We could also use `np.select()` here, but it is overkill to test one condition. `np.select()` lets us test more more than one condition and provides 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])
```

6.4.2 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.

We will use these aggregations extensively in pandas.

```
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.mean(axis=0)
```

```
array([-0.1956, -0.2858, -0.1739, -0.03  ])
```

```
arr[:, 0].mean()
```

```
-0.1956
```

The `.cumsum()` method returns the sum of all previous elements.

```
arr = np.array([0, 1, 2, 3, 4, 5, 6, 7])
arr.cumsum()
```

```
array([ 0,  1,  3,  6, 10, 15, 21, 28])
```

We can use the `.cumsum()` method along the axis of a multi-dimension array, too.

```
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 lists some 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

6.4.3 Methods for Boolean Arrays

```
np.random.seed(42)
arr = np.random.randn(100)
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,  1.4656,
       -0.2258,  0.0675, -1.4247, -0.5444,  0.1109, -1.151 ,  0.3757,
       -0.6006, -0.2917, -0.6017,  1.8523, -0.0135, -1.0577,  0.8225,
       -1.2208,  0.2089, -1.9597, -1.3282,  0.1969,  0.7385,  0.1714,
       -0.1156, -0.3011, -1.4785, -0.7198, -0.4606,  1.0571,  0.3436,
       -1.763 ,  0.3241, -0.3851, -0.6769,  0.6117,  1.031 ,  0.9313,
       -0.8392, -0.3092,  0.3313,  0.9755, -0.4792, -0.1857, -1.1063,
       -1.1962,  0.8125,  1.3562, -0.072 ,  1.0035,  0.3616, -0.6451,
        0.3614,  1.538 , -0.0358,  1.5646, -2.6197,  0.8219,  0.087 ,
       -0.299 ,  0.0918, -1.9876, -0.2197,  0.3571,  1.4779, -0.5183,
       -0.8085, -0.5018,  0.9154,  0.3288, -0.5298,  0.5133,  0.0971,
        0.9686, -0.7021, -0.3277, -0.3921, -1.4635,  0.2961,  0.2611,
        0.0051, -0.2346])
```

```
arr > 0
```

```
array([ True, False,  True,  True, False, False,  True,  True, False,
        True, False, False,  True, False, False, False, False,  True,
       False, False,  True, False,  True, False, False,  True, False,
        True, False, False, False,  True, False, False,  True, False,
        True, False, False,  True,  True,  True, False, False, False,
       False, False,  True,  True, False,  True, False, False,  True,
        True,  True, False, False,  True,  True, False, False, False,
       False,  True,  True, False,  True,  True, False,  True,  True,
       False,  True, False,  True,  True, False,  True, False, False,
        True,  True, False, False, False,  True,  True, False,  True,
        True,  True, False, False, False, False,  True,  True,  True,
       False])
```

```
(arr > 0).sum() # Number of positive values
```

```
46
```

```
(arr > 0).mean() # percentage of positive values
```

```
0.4600
```

```
bools = np.array([False, False, True, False])
bools
```

```
array([False, False,  True, False])
```

```
bools.any()
```

```
True
```

```
bools.all()
```

```
False
```

7 McKinney Chapter 4 - Practice for Section 04

7.1 Announcements

7.2 Practice

```
import numpy as np
%precision 4
```

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

7.2.1 Create a 1-dimensional array named `a1` that counts from 0 to 24 by 1.

7.2.2 Create a 1-dimentional array named `a2` that counts from 0 to 24 by 3.

7.2.3 Create a 1-dimentional array named `a3` that counts from 0 to 100 by multiples of 3 and 5.

7.2.4 Create a 1-dimensional array `a3` that contains the squares of the even integers through 100,000.

How much faster is the NumPy version than the list comprehension version?

7.2.5 Write a function that mimic Excel's `pv` function.

7.2.6 Write a function that mimic Excel's `fv` function.

7.2.7 Replace the negative values in `data` with -1 and positive values with +1.

```
np.random.seed(42)
data = np.random.randn(7, 7)
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],
       [-0.6006, -0.2917, -0.6017,  1.8523, -0.0135, -1.0577,  0.8225],
       [-1.2208,  0.2089, -1.9597, -1.3282,  0.1969,  0.7385,  0.1714],
       [-0.1156, -0.3011, -1.4785, -0.7198, -0.4606,  1.0571,  0.3436]])
```

7.2.8 Write a function `npmts()` that calculates the number of payments that generate $x\%$ of the present value of a perpetuity.

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 $PV = \frac{C_1}{r-g}\left[1 - \left(\frac{1+g}{1+r}\right)^t\right]$.

7.2.9 Write a function that calculates the internal rate of return given a NumPy array of cash flows.

7.2.10 Write a function `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])
```

7.2.11 Rewrite the function `returns()` so it returns *NumPy arrays* of returns, capital gains yields, and dividend yields.

7.2.12 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])`

```
numbers = np.array([18.5, 17.0, 18.0, 19.0, 18.0])
```

7.2.13 Write functions `var()` and `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.

Both function should have an argument `sample` that is `True` by default so both functions return sample statistics by default.

Use the output of `returns()` to compare your functions with NumPy's `.var()` and `.std()` methods.

Part IV

Week 4

8 McKinney Chapter 5 - Getting Started with pandas

8.1 Introduction

Chapter 5 of Wes McKinney's *Python for Data Analysis* discusses the fundamentals of pandas, which will be our main tool for the rest of the semester. pandas is an abbrviation for *pan*el *da*ta, which provide time-stamped data for multiple individuals or firms.

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

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
```

```
%precision 4
pd.options.display.float_format = '{:.4f}'.format
%config InlineBackend.figure_format = 'retina'
```

> 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.

We will use pandas—a wrapper for NumPy that helps us manipulate and combine data—every day for the rest of the course.

8.2 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.

8.2.1 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 a NumPy array equivalent:

```
np.array([4, 7, -5, 3])
```

```
array([ 4,  7, -5,  3])
```

```
obj.values
```

```
array([ 4,  7, -5,  3], dtype=int64)
```

```
obj.index  # similar to range(4)
```

```
RangeIndex(start=0, stop=4, step=1)
```

We did not explicitly assign an index to `obj`, so `obj` has an integer index that starts at 0. We can explicitly assign an index with the `index=` argument.

```
obj2 = pd.Series([4, 7, -5, 3], index=['d', 'b', 'a', 'c'])
obj2
```

```
d    4
b    7
a   -5
c    3
dtype: int64
```

```
obj2.index
```

```
Index(['d', 'b', 'a', 'c'], dtype='object')
```

```
obj2['a']
```

```
-5
```

```
obj2.iloc[2]
```

```
-5
```

```
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
```

A pandas series behaves like a NumPy array. 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 * 2
```

```
d    12
b    14
a   -10
c     6
dtype: int64
```

```
'b' in obj2
```

```
True
```

```
'e' in obj2
```

```
False
```

We can create a pandas series from a dictionary. The dictionary labels 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
```

We can create a pandas series from a list, too. Note that pandas respects the order of the assigned index. Also, pandas keeps California with `NaN` (not a number or missing value) and drops 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
```

Indices are one of pandas' super powers. When we perform mathematical operations, pandas aligns series by their indices. Here `NaN` is "not a number", which indicates missing values. `NaN` is considered 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
```

8.2.2 DataFrame

A pandas data frame is like a worksheet in an Excel workbook with row and columns 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, via either `df.column` or `df['column']`, the result is a series. We can use either the `df.colname` or the `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 a 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
```

Similarly, if we extract one row. via either `df.loc['rowlabel']` or `df.iloc[rownumber]`, the result is a series.

```
frame2.loc['one']
```

```
year      2000
state     Ohio
pop     1.5000
debt       NaN
Name: one, dtype: object
```

Data frame have two dimensions, so we have to 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 indices

```
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]` syntanx with `.loc[]` and `.iloc[]`.

```
frame2.loc['three', 'state'] # row, column
```

```
'Ohio'
```

```
frame2.loc['three', ['state', 'pop']] # 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 missing values `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
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

8.2.3 Index Objects

```
obj = pd.Series(range(3), index=['a', 'b', 'c'])
index = obj.index
```

```
index[1:]
```

```
Index(['b', 'c'], dtype='object')
```

Index objects are immutable!

```
# index[1] = 'd'  # TypeError: Index does not support mutable operations
```

Indices can contain duplicates, so an index does not guarantee our data are duplicate-free.

```
dup_labels = pd.Index(['foo', 'foo', 'bar', 'bar'])
```

8.3 Essential Functionality

This section provides the most import pandas operations. It is difficult to provide an exhaustive reference, but this section provides a head start on the core pandas functionality.

8.3.1 Dropping Entries from an Axis

> Dropping one or more entries from an axis is easy if you already have an index array or list without those entries. As that can require a bit of munging and set logic, the drop method will return a new object with the indicated value or values deleted from an axis.

```
obj = pd.Series(np.arange(5.), index=['a', 'b', 'c', 'd', 'e'])
obj
```

```
a   0.0000
b   1.0000
c   2.0000
d   3.0000
e   4.0000
dtype: float64
```

```
obj_without_d_and_c = obj.drop(['d', 'c'])
obj_without_d_and_c
```

```
a   0.0000
b   1.0000
e   4.0000
dtype: float64
```

The `.drop()` method works on data frames, too.

```
data = pd.DataFrame(
    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

```
data.drop(['Colorado', 'Ohio']) # implied ", axis=0"
```

	one	two	three	four
Utah	8	9	10	11
New York	12	13	14	15

```
data.drop(['Colorado', 'Ohio'], axis=0)
```

	one	two	three	four
Utah	8	9	10	11
New York	12	13	14	15

```
data.drop(index=['Colorado', 'Ohio'])
```

	one	two	three	four
Utah	8	9	10	11
New York	12	13	14	15

The `.drop()` method accepts an `axis` argument and the default is `axis=0` to drop rows based on labels. To drop columns, we use `axis=1` or `axis='columns'`.

```
data.drop('two', axis=1)
```

	one	three	four
Ohio	0	2	3
Colorado	4	6	7
Utah	8	10	11
New York	12	14	15

```
data.drop(columns='two')
```

	one	three	four
Ohio	0	2	3
Colorado	4	6	7
Utah	8	10	11
New York	12	14	15

8.3.2 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.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['b':'c']
```

```
b   1.0000
c   2.0000
dtype: float64
```

```
obj['b':'c'] = 5
obj
```

```
a   0.0000
b   5.0000
c   5.0000
d   3.0000
dtype: float64
```

```
data = pd.DataFrame(
    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: int32
```

Indexing two or more columns returns a data frame.

```
data[['three', 'one']]
```

	three	one
Ohio	2	0
Colorado	6	4
Utah	10	8
New York	14	12

If we want a one-column data frame, we can use `[[]]`:

```
data['three']
```

```
Ohio          2
Colorado      6
Utah         10
New York     14
Name: three, dtype: int32
```

```
data[['three']]
```

	three
Ohio	2
Colorado	6
Utah	10
New York	14

```
data.iloc[:2]
```

	one	two	three	four
Ohio	0	1	2	3
Colorado	4	5	6	7

We can index a data frame with Booleans, as we did with NumPy arrays.

```
data < 5
```

	one	two	three	four
Ohio	True	True	True	True
Colorado	True	False	False	False
Utah	False	False	False	False
New York	False	False	False	False

```
data[data < 5] = 0
data
```

	one	two	three	four
Ohio	0	0	0	0

	one	two	three	four
Colorado	0	5	6	7
Utah	8	9	10	11
New York	12	13	14	15

Finally, we can chain slices.

```
data.iloc[:, :3][data.three > 5]
```

	one	two	three
Colorado	0	5	6
Utah	8	9	10
New York	12	13	14

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.

8.3.3 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([7.3, -2.5, 3.4, 1.5], index=['a', 'c', 'd', 'e'])
s2 = pd.Series([-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(np.arange(9.).reshape((3, 3)), columns=list('bcd'), index=['Ohio', 'Tex
df2 = pd.DataFrame(np.arange(12.).reshape((4, 3)), columns=list('bde'), index=['Utah', 'Oh
```

```
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

```
df1 = pd.DataFrame({'A': [1, 2]})
df2 = pd.DataFrame({'B': [3, 4]})
```

```
df1
```

	A
0	1
1	2

	A

```
df2
```

	B
0	3
1	4

```
df1 - df2
```

	A	B
0	NaN	NaN
1	NaN	NaN

8.3.3.1 Arithmetic methods with fill values

```
df1 = pd.DataFrame(np.arange(12.).reshape((3, 4)), columns=list('abcd'))
df2 = pd.DataFrame(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

	a	b	c	d	e
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. Note that 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

8.3.3.2 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 the example above.

```
frame = pd.DataFrame(
    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

	b	d	e
Texas	6.0000	6.0000	6.0000
Oregon	9.0000	9.0000	9.0000

```
series2 = pd.Series(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

```
series3 = frame['d']
```

```
frame.sub(series3, axis='index')
```

	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

8.3.4 Function Application and Mapping

```
np.random.seed(42)
frame = pd.DataFrame(
    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:

Note that we can use anonymous (lambda) functions "on the fly":

```
frame.apply(lambda x: x.max() - x.min())
```

```
b   1.0825
d   1.2309
e   1.1172
dtype: float64
```

```
frame.apply(lambda x: x.max() - x.min(), axis=1)
```

```
Utah     0.7860
Ohio     1.7572
Texas    2.0487
Oregon   1.0083
dtype: float64
```

However, under the hood, the `.apply()` is basically a `for` loop and much slowly than optimized, built-in methods. Here is an example of the speed costs of `.apply()`:

```
%timeit frame['e'].abs()
```

```
11.5 µs ± 1.34 µs per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
```

```
%timeit frame['e'].apply(np.abs)
```

```
17.1 µs ± 2.03 µs per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
```

8.4 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()
```

```
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.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
```

8.4.1 Correlation and Covariance

```
import yfinance as yf
```

```
tickers = yf.Tickers('AAPL IBM MSFT GOOG')
```

```
prices = tickers.history(period='max', auto_adjust=False)
```

```
[**********100%%**********]  4 of 4 completed
```

```
prices['Adj Close']
```

Date	AAPL	GOOG	IBM	MSFT
1962-01-02	NaN	NaN	1.5558	NaN
1962-01-03	NaN	NaN	1.5694	NaN
1962-01-04	NaN	NaN	1.5537	NaN
1962-01-05	NaN	NaN	1.5231	NaN
1962-01-08	NaN	NaN	1.4946	NaN
...	...	...	...	...
2023-12-18	195.8900	137.1900	162.7400	372.6500
2023-12-19	196.9400	138.1000	161.5600	373.2600
2023-12-20	194.8300	139.6600	160.0500	370.6200
2023-12-21	194.6800	141.8000	160.7800	373.5400
2023-12-22	195.2550	143.1900	161.8766	374.1400

The `prices` data frames contains daily data for AAPL, IBM, MSFT, and GOOG. The `Adj Close` column provides a reverse-engineered daily closing price that accounts for dividends paid and stock splits (and reverse splits). As a result, the `.pct_change()` in `Adj Close` considers both price changes (i.e., capital gains) and dividends, 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 = prices['Adj Close'].pct_change().dropna()
returns
```

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
2004-08-25	0.0344	0.0108	0.0042	0.0114
2004-08-26	0.0487	0.0180	-0.0045	-0.0040
...	...	...	...	...
2023-12-18	-0.0085	0.0250	0.0031	0.0052
2023-12-19	0.0054	0.0066	-0.0073	0.0016
2023-12-20	-0.0107	0.0113	-0.0093	-0.0071
2023-12-21	-0.0008	0.0153	0.0046	0.0079
2023-12-22	0.0030	0.0098	0.0068	0.0016

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)
```

```
AAPL   0.3665
GOOG   0.2565
IBM    0.0908
MSFT   0.1972
dtype: float64
```

We multiply by $\sqrt{252}$ to annualize the standard deviation of daily returns because variances grow linearly with time, there are (about) 252 trading days per year, and the standard deviation is the square root of the variance.

```
returns.std().mul(np.sqrt(252))
```

```
AAPL   0.3281
GOOG   0.3076
IBM    0.2264
MSFT   0.2727
dtype: float64
```

The best explanation I have found on why stock return volatility (the standard deviation of stocks returns) grows with the square root of time is at the bottom of page 7 of chapter 8 of Ivo Welch's free corporate finance textbook.

We can calculate pairwise correlations.

```
returns['MSFT'].corr(returns['IBM'])
```

```
0.4943
```

We can also calculate correlation matrices.

```
returns.corr()
```

	AAPL	GOOG	IBM	MSFT
AAPL	1.0000	0.5188	0.4295	0.5234
GOOG	0.5188	1.0000	0.3956	0.5623
IBM	0.4295	0.3956	1.0000	0.4943

	AAPL	GOOG	IBM	MSFT
MSFT	0.5234	0.5623	0.4943	1.0000

```
returns.corr().loc['MSFT', 'IBM']
```

```
0.4943
```

9 McKinney Chapter 5 - Practice for Section 04

9.1 Announcements

9.2 Practice

```
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'
```

```
tickers = yf.Tickers('AAPL IBM MSFT GOOG')
prices = tickers.history(period='max', auto_adjust=False)
returns = prices['Adj Close'].pct_change().dropna()
returns
```

```
[**********100%%**********]  4 of 4 completed
```

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
2004-08-25	0.0344	0.0108	0.0042	0.0114
2004-08-26	0.0487	0.0180	-0.0045	-0.0040
...	...	...	...	...

Date	AAPL	GOOG	IBM	MSFT
2023-12-18	-0.0085	0.0250	0.0031	0.0052
2023-12-19	0.0054	0.0066	-0.0073	0.0016
2023-12-20	-0.0107	0.0113	-0.0093	-0.0071
2023-12-21	-0.0008	0.0153	0.0046	0.0079
2023-12-22	0.0026	0.0076	0.0072	0.0012

9.2.1 What are the mean daily returns for these four stocks?

9.2.2 What are the standard deviations of daily returns for these four stocks?

9.2.3 What are the *annualized* means and standard deviations of daily returns for these four stocks?

9.2.4 Plot *annualized* means versus standard deviations of daily returns for these four stocks

Use `plt.scatter()`, which expects arguments as `x` (standard deviations) then `y` (means).

9.2.5 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 will need to download new data, into `tickers2`, `prices2`, and `returns2`.

9.2.6 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.

9.2.7 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.

9.2.8 Which stocks have the minimum and maximum total returns?

9.2.9 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.

9.2.10 Repeat the plot above with only the minimum and maximum total returns

Part V

Week 5

10 Herron Topic 1 - Web Data, Log and Simple Returns, and Portfolio Math

This notebook covers three topics:

1. How to download web data with the yfinance and pandas-datareader packages
2. How to calculate log and simple returns
3. How to calculate portfolio returns

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
```

```
%precision 4
pd.options.display.float_format = '{:.4f}'.format
%config InlineBackend.figure_format = 'retina'
```

10.1 Web Data

We will typically use the yfinance and pandas-datarader packages to download data from the web. If you followed my instructions to install Miniconda on your computer, you have already installed these packages.

10.1.1 The yfinance Package

The yfinance package provides "a reliable, threaded, and Pythonic way to download historical market data from Yahoo! finance." Other packages provide similar functionality, but yfinance is best.

```
import yfinance as yf
```

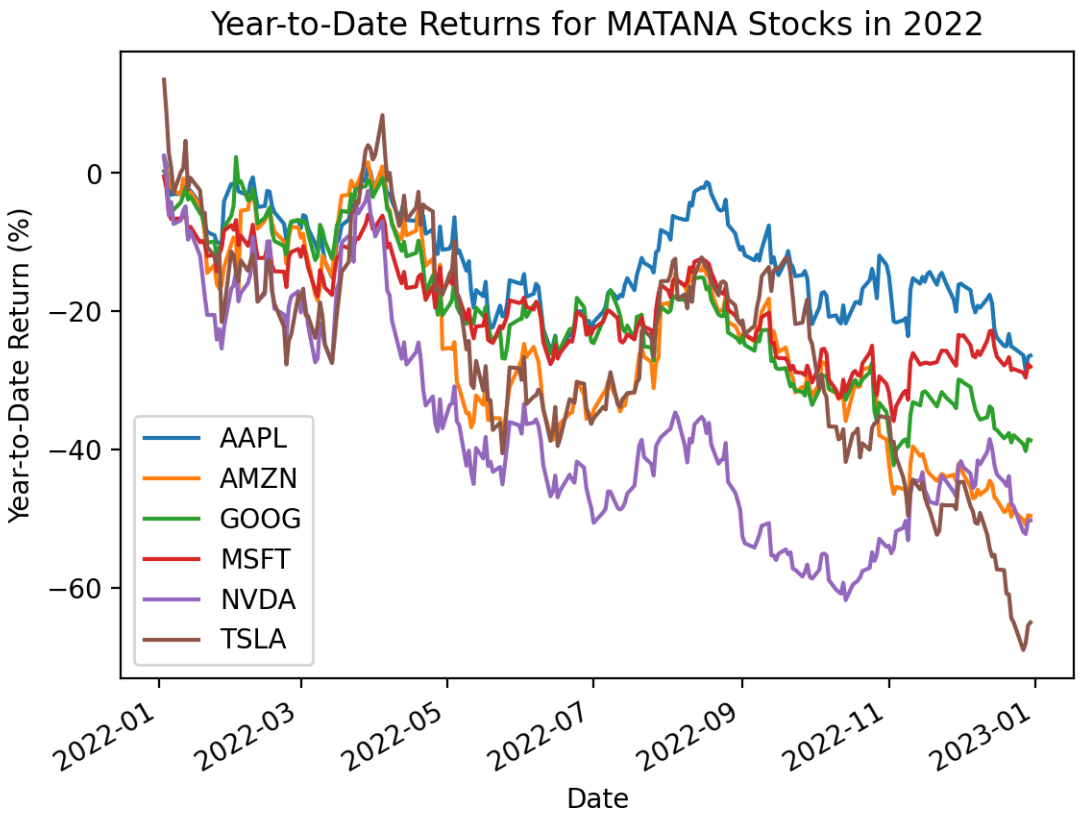
We can download data for the MATANA stocks (Microsoft, Alphabet, Tesla, Amazon, Nvidia, and Apple). We can pass tickers as either a space-delimited string or a list of strings.

```
df = yf.download(tickers='MSFT GOOG TSLA AMZN NVDA AAPL')
df
```

```
[**********************100%%**********************]  6 of 6 completed
```

	Adj Close						Close		
Date	AAPL	AMZN	GOOG	MSFT	NVDA	TSLA	AAPL	AMZN	GOOG
1980-12-12	0.0993	NaN	NaN	NaN	NaN	NaN	0.1283	NaN	NaN
1980-12-15	0.0941	NaN	NaN	NaN	NaN	NaN	0.1217	NaN	NaN
1980-12-16	0.0872	NaN	NaN	NaN	NaN	NaN	0.1127	NaN	NaN
1980-12-17	0.0894	NaN	NaN	NaN	NaN	NaN	0.1155	NaN	NaN
1980-12-18	0.0920	NaN	NaN	NaN	NaN	NaN	0.1189	NaN	NaN
...	...	...	...	...	...	...	...	...	...
2023-12-18	195.8900	154.0700	137.1900	372.6500	500.7700	252.0800	195.8900	154.0700	137.1900
2023-12-19	196.9400	153.7900	138.1000	373.2600	496.0400	257.2200	196.9400	153.7900	138.1000
2023-12-20	194.8300	152.1200	139.6600	370.6200	481.1100	247.1400	194.8300	152.1200	139.6600
2023-12-21	194.6800	153.8400	141.8000	373.5400	489.9000	254.5000	194.6800	153.8400	141.8000
2023-12-22	194.8899	153.7950	142.3800	374.0350	490.8500	257.1700	194.8899	153.7950	142.3800

```
(
    df
    ['Adj Close']
    .pct_change()
    .loc['2022']
    .add(1)
    .cumprod()
    .sub(1)
    .mul(100)
    .plot()
)
plt.ylabel('Year-to-Date Return (%)')
plt.title('Year-to-Date Returns for MATANA Stocks in 2022')
plt.show()
```



10.1.2 The pandas-datareader package

The pandas-datareader package provides easy access to various data sources, including the Kenneth French Data Library and the Federal Reserve Economic Data (FRED). The pandas-datareader package also downloads Yahoo! Finance data, but the yfinance package has better documentation. We will use `pdr` as the abbreviated prefix for pandas-datareader.

```
import pandas_datareader as pdr
```

Here we download the daily benchmark factors from Ken 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']
```

For Fama and French data, pandas-datareader returns the most recent five years of data unless we specify a `start` date. French typically provides data back through the second half of 1926. pandas-datareader returns dictionaries of data frames, and the `'DESCR'` value describes these data frames.

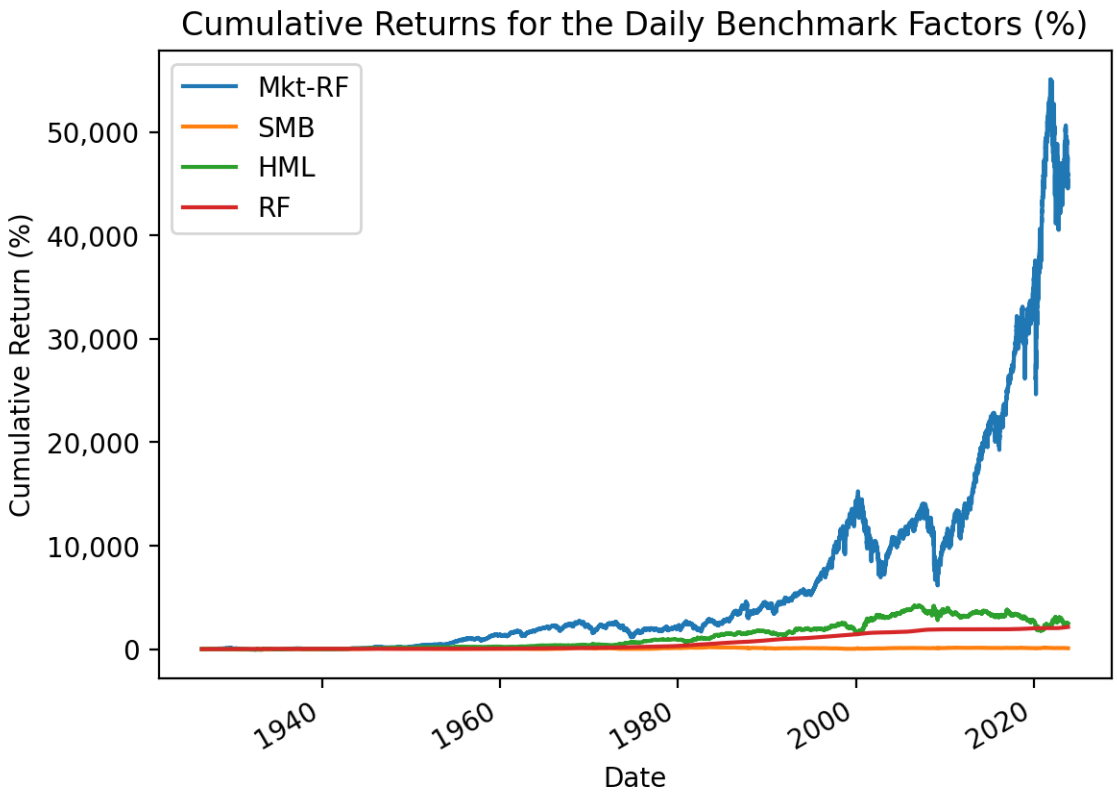
```
ff_all = pdr.DataReader(
    name='F-F_Research_Data_Factors_daily',
    data_source='famafrench',
    start='1900'
)
```

```
C:\Users\r.herron\AppData\Local\Temp\ipykernel_19752\2526882917.py:1: FutureWarning: The argu
  ff_all = pdr.DataReader(
```

```
print(ff_all['DESCR'])
```

```
F-F Research Data Factors daily
-------------------------------

This file was created by CMPT_ME_BEME_RETS_DAILY using the 202310 CRSP database. The Tbill re

  0 : (25608 rows x 4 cols)
```

```
(
    ff_all[0]
    .div(100)
    .add(1)
    .cumprod()
    .sub(1)
    .mul(100)
    .plot()
)
plt.ylabel('Cumulative Return (%)')
plt.title('Cumulative Returns for the Daily Benchmark Factors (%)')
plt.gca().yaxis.set_major_formatter(plt.matplotlib.ticker.StrMethodFormatter('{x:,.0f}'))
plt.show()
```



10.2 Log and Simple Returns

We will typically use *simple* returns, calculated as $R_{simple,t} = \frac{P_t+D_t-P_{t-1}}{P_{t-1}} = \frac{P_t+D_t}{P_{t-1}} - 1$. The simple return is the return that investors receive on invested dollars. 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:

$$R_{log,t} = \log(1+R_{simple,t}) = \log\left(1 + \frac{P_t + D_t}{P_{t-1}} - 1\right) = \log\left(\frac{P_t + D_t}{P_{t-1}}\right) = \log(P_t+D_t) - \log(P_{t-1})$$

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.

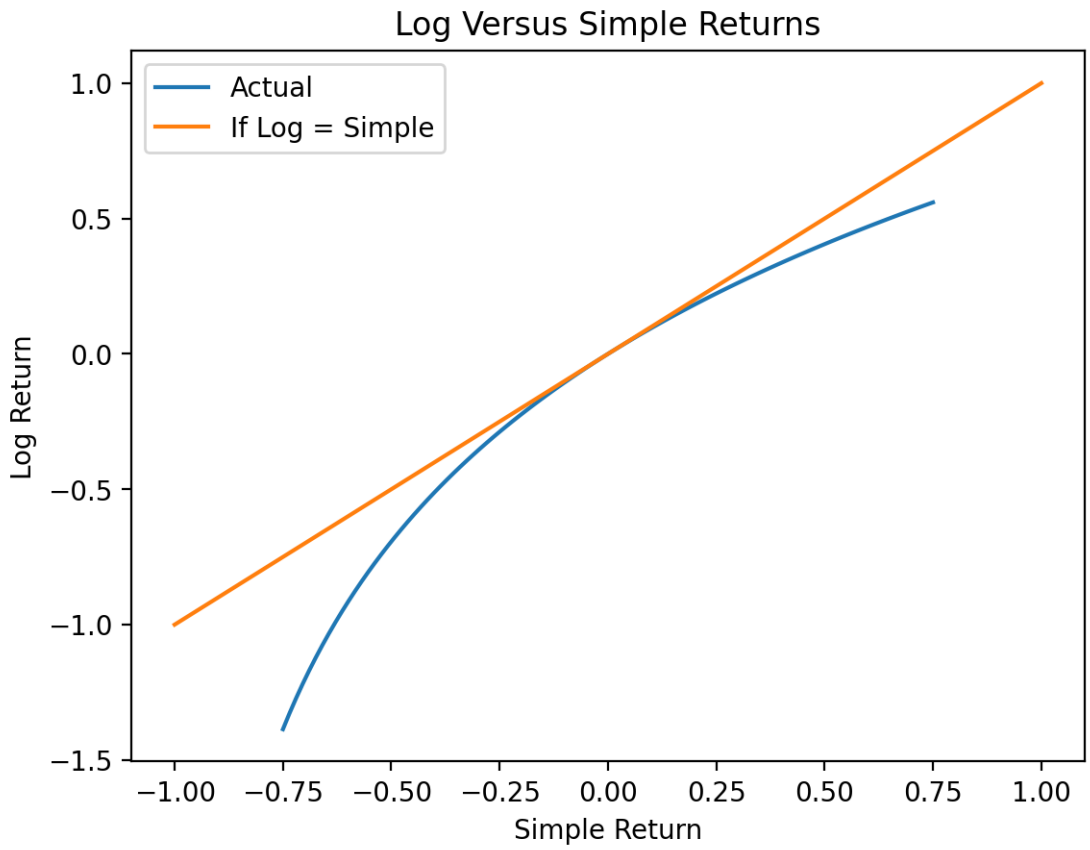
We will typically use *simple* returns instead of *log* returns. However, this section explains the differences between simple and log returns and where each is appropriate.

10.2.1 Simple and Log Returns are Similar for Small Returns

$\log(1 + x) \approx x$ 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 these horizons. The following figure shows $R_{simple,t} \approx R_{log,t}$ for small Rs.

```
R = np.linspace(-0.75, 0.75, 100)
logR = np.log(1 + R)
```

```
plt.plot(R, logR)
plt.plot([-1, 1], [-1, 1])
plt.xlabel('Simple Return')
plt.ylabel('Log Return')
plt.title('Log Versus Simple Returns')
plt.legend(['Actual', 'If Log = Simple'])
plt.show()
```



10.2.2 Simple Return Advantage: Portfolio Calculations

We can only perform portfolio calculations with simple returns. 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 two stocks with portfolio weights of 50%, our portfolio return is $R_{portfolio} = 0.5R_1 + 0.5R_2 = \frac{R_1+R_2}{2}$. However, we cannot calculate portfolio returns with log returns because the sum of logs is the log of products.

We cannot calculate portfolio returns as the weighted average of log returns.

10.2.3 Log Return Advantage: Log Returns are Additive

The advantage of log returns is that we can compound log returns with addition. The additive property of log returns makes code simple, computations fast, and proofs easy when we compound returns over multiple periods.

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.

The following code generates 10,000 random log returns. The `np.random.randn()` call generates normally distributed random numbers. To generate equivalent simple returns, we exponentiate these log returns, then subtract one.

```
np.random.seed(42)
df2 = pd.DataFrame(data={'R': np.exp(np.random.randn(10000)) - 1})
```

```
df2.describe()
```

	R
count	10000.0000
mean	0.6529
std	2.1918
min	-0.9802
25%	-0.4896

	R
50%	-0.0026
75%	0.9564
max	49.7158

We can time the calculation of 12-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 2,000. ***We will learn about `.rolling()` and `.apply()` in a few weeks, but they provide the best example of when to use log returns.***

```
%%timeit
df2['R12_via_simple'] = (
    df2['R']
    .add(1)
    .rolling(12)
    .apply(lambda x: x.prod())
    .sub(1)
)
```

```
316 ms ± 114 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
%%timeit
df2['R12_via_log'] = (
    df2['R']
    .add(1)
    .pipe(np.log)
    .rolling(12)
    .sum()
    .pipe(np.exp)
    .sub(1)
)
```

```
742 µs ± 91.2 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

```
np.allclose(df2['R12_via_simple'], df2['R12_via_log'], equal_nan=True)
```

```
True
```

These two approaches calculate the same return, but the simple-return approach is 1,000 times slower than the log-return approach!

We can use log returns to calculate total returns very quickly!

10.3 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, and w_i is the weight on asset i.

10.3.1 The 1/N Portfolio

The $\frac{1}{N}$ portfolio equally weights portfolio assets, so $w_1 = w_2 = \cdots = w_N = \frac{1}{N}$. We typically rebalance the $\frac{1}{N}$ portfolio every period. 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()` to calculate $\frac{1}{N}$ portfolio returns.

```
returns = df['Adj Close'].pct_change().loc['2022']

returns
```

Date	AAPL	AMZN	GOOG	MSFT	NVDA	TSLA
2022-01-03	0.0250	0.0221	0.0027	-0.0047	0.0241	0.1353
2022-01-04	-0.0127	-0.0169	-0.0045	-0.0171	-0.0276	-0.0418
2022-01-05	-0.0266	-0.0189	-0.0468	-0.0384	-0.0576	-0.0535
2022-01-06	-0.0167	-0.0067	-0.0007	-0.0079	0.0208	-0.0215
2022-01-07	0.0010	-0.0043	-0.0040	0.0005	-0.0330	-0.0354
...	...	...	...	...	...	...
2022-12-23	-0.0028	0.0174	0.0176	0.0023	-0.0087	-0.0176
2022-12-27	-0.0139	-0.0259	-0.0209	-0.0074	-0.0714	-0.1141
2022-12-28	-0.0307	-0.0147	-0.0167	-0.0103	-0.0060	0.0331
2022-12-29	0.0283	0.0288	0.0288	0.0276	0.0404	0.0808
2022-12-30	0.0025	-0.0021	-0.0025	-0.0049	0.0008	0.0112

```
returns.mean()
```

```
AAPL   -0.0010
AMZN   -0.0022
```

```
GOOG   -0.0016
MSFT   -0.0011
NVDA   -0.0020
TSLA   -0.0033
dtype: float64
```

```
rp_1 = returns.mean(axis=1)
rp_1
```

```
Date
2022-01-03    0.0341
2022-01-04   -0.0201
2022-01-05   -0.0403
2022-01-06   -0.0055
2022-01-07   -0.0125
               ...
2022-12-23    0.0014
2022-12-27   -0.0423
2022-12-28   -0.0075
2022-12-29    0.0391
2022-12-30    0.0008
Length: 251, 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.

10.3.2 A More General Solution

If we combine 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 of portfolio weights. We can use this approach to calculate $\frac{1}{N}$ portfolio returns, too.

```
weights = np.ones(returns.shape[1]) / returns.shape[1]
weights
```

```
array([0.1667, 0.1667, 0.1667, 0.1667, 0.1667, 0.1667])
```

```
rp_2 = returns.dot(weights)
rp_2
```

```
Date
2022-01-03    0.0341
2022-01-04   -0.0201
2022-01-05   -0.0403
2022-01-06   -0.0055
2022-01-07   -0.0125
               ...
2022-12-23    0.0014
2022-12-27   -0.0423
2022-12-28   -0.0075
2022-12-29    0.0391
2022-12-30    0.0008
Length: 251, dtype: float64
```

Both approaches give the same answer!

```
np.allclose(rp_1, rp_2, equal_nan=True)
```

```
True
```

11 Herron Topic 1 - Practice for Section 04

11.1 Announcements

11.2 Practice

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
```

```
%precision 4
pd.options.display.float_format = '{:.4f}'.format
%config InlineBackend.figure_format = 'retina'
```

```
import yfinance as yf
import pandas_datareader as pdr
```

11.2.1 Download all available daily price data for tickers TSLA, F, AAPL, AMZN, and META to data frame `histories`

Remove time zone information from the index and use `histories.columns.names` to label the variables and tickers as `Variable` and `Ticker`.

11.2.2 Calculate all available daily returns and save to data frame `returns`

11.2.3 Slices returns for the 2020s and assign to `returns_2020s`

11.2.4 Download all available data for the Fama and French daily benchmark factors to dictionary `ff_all`

I often use the following code snippet to find the exact name for the the daily benchmark factors file.

```
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']
```

11.2.5 Slice the daily benchmark factors, convert them to decimal returns, and assign to `ff`

11.2.6 Use the `.cumprod()` method to plot cumulative returns for these stocks in the 2020s

11.2.7 Use the `.cumsum()` method with log returns to plot cumulative returns for these stocks in the 2020s

11.2.8 Use price data only to plot cumulative returns for these stocks in the 2020s

11.2.9 Calculate the Sharpe Ratio for TSLA

Calculate the Sharpe Ratio with all available returns and 2020s returns. Recall the Sharpe Ratio is $\frac{\overline{R_i - R_f}}{\sigma_i}$, where σ_i is the volatility of *excess* returns.

I suggest you write a function named `sharpe()` to use for the rest of this notebook.

11.2.10 Calculate the market beta for TSLA

Calculate the market beta with all available returns and 2020s returns. Recall we estimate market beta with the ordinary least squares (OLS) regression $R_i - R_f = \alpha + \beta(R_m - R_f) + \epsilon$. We can estimate market beta with the covariance formula above for a univariate regression if we do not need goodness of fit statistics.

I suggest you write a function named `beta()` to use for the rest of this notebook.

11.2.11 Guess the Sharpe Ratios for these stocks in the 2020s

11.2.12 Guess the market betas for these stocks in the 2020s

11.2.13 Calculate the Sharpe Ratios for these stocks in the 2020s

How good were your guesses?

11.2.14 Calculate the market betas for these stocks in the 2020s

How good were your guesses?

11.2.15 Calculate the Sharpe Ratio for an *equally weighted* portfolio of these stocks in the 2020s

What do you notice?

11.2.16 Calculate the market beta for an *equally weighted* portfolio of these stocks in the 2020s

What do you notice?

11.2.17 Calculate the market betas for these stocks every calendar year for every possible year

Save these market betas to data frame `betas`. Our current Python knowledge limits us to a for-loop, but we will learn easier and faster approaches soon!

11.2.18 Plot the time series of market betas

Part VI

Week 6

12 McKinney Chapter 8 - Data Wrangling: Join, Combine, and Reshape

12.1 Introduction

Chapter 8 of Wes McKinney's *Python for Data Analysis* 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 data frames so it has fewer columns and more rows (wide to long) or more columns and fewer rows (long to wide); we can also reshape a series to a data frame and vice versa

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

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
```

```
%precision 4
pd.options.display.float_format = '{:.4f}'.format
%config InlineBackend.figure_format = 'retina'
```

```
import yfinance as yf
import pandas_datareader as pdr
```

12.2 Hierarchical Indexing

We need to learn about hierarchical indexing before we learn about combining and reshaping data. A hierarchical index gives two or more index levels to an axis. 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 partially index this series to concisely subset data.

```
data['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', 'd']]
```

```
b  1    1.5230
   3   -0.2342
d  2    0.7674
   3   -0.4695
dtype: float64
```

We can subset on the index inner level, too. Here the first `:` slices all values in the outer index.

```
data.loc[:, 2]
```

```
a   -0.1383
c    1.5792
d    0.7674
dtype: float64
```

Here `data` has a stacked format. For each outer index level (the letters), we have multiple observations based on the inner index level (the numbers). We can un-stack `data` to convert the inner index level to columns.

```
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

```
data.unstack().stack()
```

```
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 create a data frame with hieracrhical indexes or multi-indexes on rows *and* columns.

```
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		Colorado
		Green	Red	Green
a	1	0	1	2
	2	3	4	5
b	1	6	7	8
	2	9	10	11

We can name these multi-indexes but names are not required.

```
frame.index.names = ['key1', 'key2']
frame.columns.names = ['state', 'color']
frame
```

	state	Ohio		Colorado
	color	Green	Red	Green
key1	key2			
a	1	0	1	2
	2	3	4	5
b	1	6	7	8
	2	9	10	11

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 level.

```
frame['Ohio']
```

key1	color key2	Green	Red
a	1	0	1
	2	3	4
b	1	6	7
	2	9	10

We can pass a tuple if we only want one column.

```
frame[[('Ohio', 'Green')]]
```

key1	state color key2	Ohio Green
a	1	0
	2	3
b	1	6
	2	9

We have to do a more work to slice the inner level of the column index.

```
frame.loc[:, (slice(None), 'Green')]
```

key1	state color key2	Ohio Green	Colorado Green
a	1	0	2
	2	3	5
b	1	6	8
	2	9	11

We can use `pd.IndexSlice[:, 'Green']` an alternative to `(slice(None), 'Green')`.

```
frame.loc[:, pd.IndexSlice[:, 'Green']]
```

	state	Ohio	Colorado
	color	Green	Green
key1	key2		
a	1	0	2
	2	3	5
b	1	6	8
	2	9	11

12.2.1 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.swaplevel()
```

	state	Ohio		Colorado
	color	Green	Red	Green
key2	key1			
1	a	0	1	2
2	a	3	4	5
1	b	6	7	8
2	b	9	10	11

We can use index *names*, too.

```
frame.swaplevel('key1', 'key2')
```

	state	Ohio		Colorado
	color	Green	Red	Green
key2	key1			
1	a	0	1	2
2	a	3	4	5
1	b	6	7	8
2	b	9	10	11

We can also sort on an index (or list of indexes). After we swap levels, we may want to sort our data.

```
frame
```

	state	Ohio		Colorado
	color	Green	Red	Green
key1	key2			
a	1	0	1	2
	2	3	4	5
b	1	6	7	8
	2	9	10	11

```
frame.sort_index(level=1)
```

	state	Ohio		Colorado
	color	Green	Red	Green
key1	key2			
a	1	0	1	2
b	1	6	7	8
a	2	3	4	5
b	2	9	10	11

Again, we can give index *names*, too.

```
frame.sort_index(level='key2')
```

	state	Ohio		Colorado
	color	Green	Red	Green
key1	key2			
a	1	0	1	2
b	1	6	7	8
a	2	3	4	5
b	2	9	10	11

We can sort by two or more index levels by passing a list of index levels or names.

```
frame.sort_index(level=[0, 1])
```

	state	Ohio		Colorado
	color	Green	Red	Green
key1	key2			
a	1	0	1	2
	2	3	4	5
b	1	6	7	8
	2	9	10	11

We can chain these methods, too.

```
frame.swaplevel(0, 1).sort_index(level=0)
```

	state	Ohio		Colorado
	color	Green	Red	Green
key2	key1			
1	a	0	1	2
	b	6	7	8
2	a	3	4	5
	b	9	10	11

12.2.2 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

	a	b	c	d
4	4	3	two	1
5	5	2	two	2
6	6	1	two	3

The `.set_index()` method converts columns to indexes, and removes the columns from the data frame 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 removes the indexes, adds them as columns, and sets in 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

12.3 Combining and Merging Datasets

pandas provides several methods and functions to combine and merge data. We can typically create the same output with any of these methods or functions, but one may be more efficient than the others. If I want to combine data frames with similar indexes, I try the `.join()` method first. The `.join()` also lets use can combine more than two data frames at once. Otherwise, I try the `.merge()` method, which has a function `pd.merge()`, too. The `pd.merge()` function is more general than the `.join()` method, so we will start with `pd.merge()`.

The pandas website provides helpful visualizations.

12.3.1 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.

We will start with the `pd.merge()` syntax, but pandas also has `.merge()` and `.join()` methods. Learning these other syntaxes is easy once we understand the `pd.merge()` syntax.

```
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

	key	data2
2	d	2

```
pd.merge(df1, df2)
```

	key	data1	data2
0	b	0	1
1	b	1	1
2	b	6	1
3	a	2	0
4	a	4	0
5	a	5	0

The default `how` 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	b	0.0000	1.0000
1	b	1.0000	1.0000
2	b	6.0000	1.0000
3	a	2.0000	0.0000
4	a	4.0000	0.0000
5	a	5.0000	0.0000
6	c	3.0000	NaN
7	d	NaN	2.0000

A 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

	key	data1	data2
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 rights 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
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 all 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` to avoid unexpected results.

```
pd.merge(df1, df2, on='key')
```

	key	data1	data2
0	b	0	1
1	b	1	1
2	b	6	1
3	a	2	0
4	a	4	0
5	a	5	0

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
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
```

```
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	b	6	b	1
3	a	2	a	0
4	a	4	a	0
5	a	5	a	0

Here `pd.merge()` dropped row `c` from `df3` and row `d` from `df4`. Rows `c` and `d` dropped because

`pd.merge()` *inner* joins be default. An inner join keeps the intersection of the left and right data frame keys. Further, rows `a` and `b` from `df4` appear three times to match `df3`. If we want to keep rows `c` and `d`, we can *outer* join `df3` and `df4` with `how='outer'`.

```
pd.merge(df1, df2, how='outer')
```

	key	data1	data2
0	b	0.0000	1.0000
1	b	1.0000	1.0000
2	b	6.0000	1.0000
3	a	2.0000	0.0000
4	a	4.0000	0.0000
5	a	5.0000	0.0000
6	c	3.0000	NaN
7	d	NaN	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

	key	data2
4	d	4

```
pd.merge(df1, df2, on='key')
```

	key	data1	data2
0	b	0	1
1	b	0	3
2	b	1	1
3	b	1	3
4	b	5	1
5	b	5	3
6	a	2	0
7	a	2	2
8	a	4	0
9	a	4	2

> 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.***

We can merge on more than one key. For example, we may 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	foo	one	1.0000	4.0000
1	foo	one	1.0000	5.0000
2	foo	two	2.0000	NaN
3	bar	one	3.0000	6.0000
4	bar	two	NaN	7.0000

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 suffixes to avoid later 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 whenever I am in doubt. ***Table 8-2*** lists the most 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.

12.3.2 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
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
3	Nevada	2001	3.0000	0.0000	1.0000
4	Nevada	2002	4.0000	NaN	NaN
4	Nevada	2000	NaN	2.0000	3.0000

```
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 left and right 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.

If we have matching indexes on left and right, we can use `.join()`.

```
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. The `.join()` method uses indexes, so it requires few arguments and accepts 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

	New York	Oregon
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

12.3.3 Concatenating Along an Axis

The `pd.concat()` function provides a flexible way to combine data frames and series along either 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

The first is handy if we have to read and combine a directory of .csv files.

```
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'])
```

```
pd.concat([s1, s2, s3])
```

```
a    0
b    1
c    2
d    3
e    4
f    5
g    6
dtype: int64
```

```
pd.concat([s1, s2, s3], 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
f	NaN	NaN	5.0000
g	NaN	NaN	6.0000

```
result = pd.concat([s1, s2, s3], keys=['one', 'two', 'three'])

result
```

```
one    a    0
       b    1
two    c    2
       d    3
       e    4
three  f    5
       g    6
dtype: int64
```

```
result.unstack()
```

	a	b	c	d	e	f	g
one	0.0000	1.0000	NaN	NaN	NaN	NaN	NaN

	a	b	c	d	e	f	g
two	NaN	NaN	2.0000	3.0000	4.0000	NaN	NaN
three	NaN	NaN	NaN	NaN	NaN	5.0000	6.0000

```
pd.concat([s1, s2, s3], axis=1, keys=['one', 'two', 'three'])
```

	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(np.arange(6).reshape(3, 2), index=['a', 'b', 'c'], columns=['one', 'two
df2 = pd.DataFrame(5 + np.arange(4).reshape(2, 2), index=['a', 'c'], columns=['three', 'fo
```

```
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

12.4 Reshaping and Pivoting

Above, we briefly explore reshaping data with `.stack()` and `.unstack()`. Here we explore reshaping data more deeply.

12.4.1 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: int32
```

```
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
```

```
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

Un-stacking may introduce missing values because data frames are rectangular. By default, stacking drops these missing values.

```
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
```

However, we can keep missing values with `dropna=False`.

```
data2.unstack().stack(dropna=False)
```

```
one  a   0.0000
     b   1.0000
     c   2.0000
     d   3.0000
     e      NaN
two  a      NaN
     b      NaN
     c   4.0000
     d   5.0000
     e   6.0000
dtype: float64
```

```
df = pd.DataFrame({
    'left': result,
    'right': result + 5
    },
    columns=pd.Index(['left', 'right'], name='side')
)

df
```

state	side number	left	right
Ohio	one	0	5
	two	1	6
	three	2	7
Colorado	one	3	8
	two	4	9
	three	5	10

If we un-stack a data frame, the un-stacked level becomes the innermost level in the resulting index.

```
df.unstack('state')
```

side	left		right	
state	Ohio	Colorado	Ohio	Colorado
number				
one	0	3	5	8
two	1	4	6	9
three	2	5	7	10

We can chain `.stack()` and `.unstack()` to rearrange our data.

```
df.unstack('state').stack('side')
```

	state	Ohio	Colorado
number	side		
one	left	0	3
	right	5	8
two	left	1	4
	right	6	9
three	left	2	5
	right	7	10

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.

13 McKinney Chapter 8 - Practice for Section 04

13.1 Announcements

13.2 Practice

13.2.1 Download data from Yahoo! Finance for BAC, C, GS, JPM, MS, and PNC and assign to data frame `stocks`.

Use `stocks.columns.names` to assign the names `Variable` and `Ticker` to the column multi index.

13.2.2 Reshape `stocks` from wide to long with dates and tickers as row indexes and assign to data frame `stocks_long`.

13.2.3 Add daily returns for each stock to data frames `stocks` and `stocks_long`.

Name the returns variable `Returns`, and maintain all multi indexes. *Hint:* Use `pd.MultiIndex()` to create a multi index for the the wide data frame `stocks`.

13.2.4 Download the daily benchmark return factors from Ken French's data library.

13.2.5 Add the daily benchmark return factors to `stocks` and `stocks_long`.

For the wide data frame `stocks`, use the outer index name `Factors`.

13.2.6 Write a function `download()` that accepts tickers and returns a wide data frame of returns with the daily benchmark return factors.

13.2.7 Download earnings per share for the stocks in `stocks` 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`.

13.2.8 Combine `earnings` with the returns from `stocks_long`.

It is easier to leave `stocks` and `stocks_long` as-is and work with slices `returns` and `returns_long`. Use the `tz_localize('America/New_York')` method add time zone information back to `returns.index` and use `pd.to_timedelta(16, unit='h')` to set time to the market close in New York City. Use `pd.merge_asof()` to match earnings announcement dates and times to appropriate return periods. For example, if a firm announces earnings after the close at 5 PM on February 7, we want to match the return period from 4 PM on February 7 to 4 PM on February 8.

13.2.9 Plot the relation between daily returns and earnings surprises

Three options in increasing difficulty:

1. Scatter plot
2. Scatter plot with a best-fit line using `regplot()` from the seaborn package
3. Bar plot using `barplot()` from the seaborn package after using `pd.qcut()` to form five groups on earnings surprises

13.2.10 Repeat the earnings exercise with the S&P 100 stocks

13.2.11 Repeat the earnings exercise with *excess returns* of the S&P 100 Stocks

Excess returns are returns minus market returns. We need to add a timezone and the closing time to the market return from Fama and French.

13.2.12 Improve your `download()` function from above

Modify `download()` to accept one or more than one ticker. Since we will not use the advanced functionality of the tickers object that `yf.Tickers()` creates, we will use `yf.download()`. The current version of `yf.download()` does not accept a `session=` argument.

Part VII

Week 7

14 Project 1

FINA 6333 – Spring 2023

To be determined

Part VIII

Week 8

15 McKinney Chapter 10 - Data Aggregation and Group Operations

15.1 Introduction

Chapter 10 of Wes McKinney's *Python for Data Analysis* discusses groupby operations, which are the pandas equivalent of Excel pivot tables. Pivot tables help us calculate statistics (e.g., sum, mean, and median) for one set of variables by groups of other variables (e.g., weekday or ticker). For example, we could use a pivot table to calculate mean daily stock returns by weekday.

We will focus on:

1. Using `.groupby()` to group by columns, indexes, and functions
2. Using `.agg()` to aggregate multiple functions
3. Using pivot tables as an alternative to `.groupby()`

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

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
```

```
%precision 4
pd.options.display.float_format = '{:.4f}'.format
%config InlineBackend.figure_format = 'retina'
```

```
import yfinance as yf
import pandas_datareader as pdr
```

15.2 GroupBy Mechanics

"Split-apply-combine" is an excellent way to describe and visualize 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 describe this operation as "sum the `data` column by groups formed on the `key` column."

```
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 one way to calculate the means of `data1` by groups formed on `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 quickly!

1. Use the `.groupby()` method to group by `key1`
2. Use the `.mean()` method to sum `data1` within each value of `key1`

Note that without the `.mean()` method, pandas only sets up the grouped object, which can accept the `.mean()` method.

```
grouped = df['data1'].groupby(df['key1'])
grouped
```

```
<pandas.core.groupby.generic.SeriesGroupBy object at 0x00000266A8DE9550>
```

```
grouped.mean()
```

```
key1
a   0.0414
b   1.0854
Name: data1, dtype: float64
```

We can can chain the `.groupby()` and `.mean()` methods!

```
df['data1'].groupby(df['key1']).mean()
```

```
key1
a   0.0414
b   1.0854
Name: data1, dtype: float64
```

If we prefer our result as a dataframe instead of a series, we can wrap `data1` with two sets of square brackets.

```
df[['data1']].groupby(df['key1']).mean()
```

key1	data1
a	0.0414
b	1.0854

We can group by more than one variable. We get a hierarchical row index (or row multi-index) when we 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 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

The grouping variables can be columns in the data frame we want to group with the `.groupby()` method. Our grouping variables are typically columns in the data frame we want to group, so this syntax is more compact and easier to understand.

```
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').mean() # TypeError: agg function failed [how->mean,dtype->object]
```

```
df.groupby('key1')[['data1', 'data2']].mean()
```

key1	data1	data2
a	0.0414	0.6292
b	1.0854	0.1490

```
df.groupby(['key1', 'key2']).mean()
```

key1	key2	data1	data2
a	one	0.1313	0.1542
	two	-0.1383	1.5792
b	one	0.6477	0.7674
	two	1.5230	-0.4695

We can use tab completion to reminder ourselves of methods we can apply to grouped series and data frames.

15.2.1 Iterating Over Groups

We can iterate over groups, too, because the `.groupby()` method generates a sequence of tuples. Each tuples contains the value(s) of the grouping variable(s) and its chunk of the dataframe. McKinney provides two loops to show how to iterate over groups.

```
for k1, group in df.groupby('key1'):
    print(k1, group, sep='\n')
```

```
a
  key1 key2   data1   data2
0    a  one  0.4967 -0.2341
1    a  two -0.1383  1.5792
4    a  one -0.2342  0.5426
b
  key1 key2  data1   data2
2    b  one 0.6477  0.7674
3    b  two 1.5230 -0.4695
```

```
for (k1, k2), group in df.groupby(['key1', 'key2']):
    print((k1, k2), group, sep='\n')
```

```
('a', 'one')
  key1 key2   data1   data2
0    a  one  0.4967 -0.2341
4    a  one -0.2342  0.5426
('a', 'two')
  key1 key2   data1  data2
1    a  two -0.1383 1.5792
('b', 'one')
  key1 key2  data1  data2
2    b  one 0.6477 0.7674
('b', 'two')
  key1 key2  data1   data2
3    b  two 1.5230 -0.4695
```

15.2.2 Grouping with Functions

We can also group with functions. Below, we group with the `len` function, which calculates the length of the first names in the row index. We could instead add a helper column to `people`, but it is easier to pass a function to `.groupby()`.

```
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. that we pass to `.groupby()`.

```
key_list = ['one', 'one', 'one', 'two', 'two']
people.groupby([len, key_list]).min()
```

		a	b	c	d	e
3	one	-0.4634	-0.4657	0.2420	-1.9133	-1.7249
	two	-0.5623	-1.0128	0.3142	-0.9080	-1.4123
5	one	-0.2341	1.5792	0.7674	-0.4695	0.5426
6	two	1.4656	-0.2258	0.0675	-1.4247	-0.5444

```
d = {'Joe': 'a', 'Jim': 'b'}
people.groupby([len, d]).min()
```

		a	b	c	d	e
3	a	0.4967	-0.1383	0.6477	1.5230	-0.2342
	b	-0.5623	-1.0128	0.3142	-0.9080	-1.4123

```
d_2 = {'Joe': 'Cool', 'Jim': 'Nerd', 'Travis': 'Cool'}
people.groupby([len, d_2]).min()
```

		a	b	c	d	e
3	Cool	0.4967	-0.1383	0.6477	1.5230	-0.2342
	Nerd	-0.5623	-1.0128	0.3142	-0.9080	-1.4123
6	Cool	1.4656	-0.2258	0.0675	-1.4247	-0.5444

15.2.3 Grouping by Index Levels

We can also group by index levels. We can specify index levels by either level number or name.

```
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)

hier_df
```

cty	US			JP	
tenor	1	3	5	1	3
0	0.1109	-1.1510	0.3757	-0.6006	-0.2917
1	-0.6017	1.8523	-0.0135	-1.0577	0.8225
2	-1.2208	0.2089	-1.9597	-1.3282	0.1969
3	0.7385	0.1714	-0.1156	-0.3011	-1.4785

```
hier_df.T.groupby(level='cty').count()
```

	0	1	2	3
cty				
JP	2	2	2	2
US	3	3	3	3

```
hier_df.T.groupby(level='tenor').count()
```

	0	1	2	3
tenor				
1	2	2	2	2
3	2	2	2	2
5	1	1	1	1

15.3 Data Aggregation

Table 10-1 provides 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, but pandas lets us use other, non-optimized methods. First, any series method is available.

```
df.groupby('key1')['data1'].quantile(0.9)
```

```
key1
a   0.3697
b   1.4355
Name: data1, dtype: float64
```

Second, we can write our own functions and pass them to the `.agg()` method. These functions should accept an array and returns 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
```

Some other methods work, too, even if they are do not aggregate an array to a single value.

```
df.groupby('key1')['data1'].describe()
```

key1	count	mean	std	min	25%	50%	75%	max
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

15.3.1 Column-Wise and Multiple Function Application

The `.agg()` methods provides two more handy features:

1. We can pass multiple functions to operate on all of the columns
2. We can pass specific functions to operate on specific columns

Here is an example with multiple functions:

```
df.groupby('key1')['data1'].agg(['mean', 'median', 'min', 'max'])
```

key1	mean	median	min	max
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

What if I wanted to calculate the mean of `data1` and the median of `data2` by `key1`?

```
df.groupby('key1').agg({'data1': 'mean', 'data2': 'median'})
```

key1	data1	data2
a	0.0414	0.5426
b	1.0854	0.1490

What if I wanted to 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

15.4 Apply: General split-apply-combine

The `.agg()` method aggrates an array to a single value. We can use the `.apply()` method for more general calculations.

We can combine the `.groupby()` and `.apply()` methods to:

1. Split a dataframe by grouping variables
2. Call the applied function on each chunk of the original dataframe
3. Recombine the output of the applied function

```
def top(x, col, n=1):
    return x.sort_values(col).head(n)
```

```
df.groupby('key1').apply(top, col='data1')
```

key1		key1	key2	data1	data2
a	4	a	one	-0.2342	0.5426
b	2	b	one	0.6477	0.7674

```
df.groupby('key1').apply(top, col='data1', n=2)
```

key1		key1	key2	data1	data2
a	4	a	one	-0.2342	0.5426
	1	a	two	-0.1383	1.5792
b	2	b	one	0.6477	0.7674
	3	b	two	1.5230	-0.4695

15.5 Pivot Tables and Cross-Tabulation

Above we manually made pivot tables with the `groupby()`, `.agg()`, `.apply()` and `.unstack()` methods. pandas provides a literal interpreation of Excel-style pivot tables with the `.pivot_table()` method and the `pandas.pivot_table()` function. These also provide row and column totals via "margins". It is worthwhile to read-through the `.pivot_table()` docstring several times.

```
ind = (
    yf.download(tickers='^GSPC ^DJI ^IXIC ^FTSE ^N225 ^HSI')
    .rename_axis(columns=['Variable', 'Index'])
    .stack()
)

ind.head()
```

```
[**********100%%**********]  6 of 6 completed
```

Date	Variable Index	Adj Close	Close	High	Low	Open	Volume
1927-12-30	^GSPC	17.6600	17.6600	17.6600	17.6600	17.6600	0.0000
1928-01-03	^GSPC	17.7600	17.7600	17.7600	17.7600	17.7600	0.0000
1928-01-04	^GSPC	17.7200	17.7200	17.7200	17.7200	17.7200	0.0000
1928-01-05	^GSPC	17.5500	17.5500	17.5500	17.5500	17.5500	0.0000
1928-01-06	^GSPC	17.6600	17.6600	17.6600	17.6600	17.6600	0.0000

The default aggregation function for `.pivot_table()` is `mean`.

```
ind.loc['2015':].pivot_table(index='Index')
```

Variable Index	Adj Close	Close	High	Low	Open	Volume
^DJI	26268.7993	26268.7993	26409.2622	26112.8527	26265.2096	290352366.5425
^FTSE	7035.2159	7035.2159	7076.6397	6992.6821	7034.7444	802505272.4427
^GSPC	3122.5674	3122.5674	3139.2737	3103.6004	3122.0834	4017528407.9796
^HSI	24483.5675	24483.5675	24647.8922	24313.7685	24500.2319	2021151100.2261
^IXIC	9021.3379	9021.3379	9081.9083	8952.0787	9019.8093	3235066862.1951
^N225	23298.2397	23298.2397	23420.5926	23166.4854	23298.8808	94902460.1367

```
ind.loc['2015':].pivot_table(index='Index', aggfunc='median')
```

Variable Index	Adj Close	Close	High	Low	Open	Volume
^DJI	25979.4648	25979.4648	26113.8896	25817.3203	25992.4355	296605000.0000

Variable Index	Adj Close	Close	High	Low	Open	Volume
^FTSE	7183.5000	7183.5000	7217.0000	7141.5999	7182.8499	755440000.0000
^GSPC	2887.9149	2887.9149	2898.5200	2876.2100	2890.2450	3819130000.0000
^HSI	24698.4805	24698.4805	24855.4707	24540.6309	24697.9805	1864023100.0000
^IXIC	7963.3198	7963.3198	8005.2749	7912.0701	7972.7300	2413820000.0000
^N225	22380.0098	22380.0098	22509.3594	22269.5293	22374.2109	82200000.0000

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='A'),
        columns='Index',
        aggfunc=['min', 'max']
    )
)
```

	min						max	
Index Date	^DJI	^FTSE	^GSPC	^HSI	^IXIC	^N225	^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
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	37557.9219	8014.2

16 McKinney Chapter 10 - Practice for Section 04

16.1 Announcements

16.2 Practice

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
```

```
%precision 4
pd.options.display.float_format = '{:.4f}'.format
%config InlineBackend.figure_format = 'retina'
```

```
import yfinance as yf
import pandas_datareader as pdr
```

16.2.1 Replicate the following `.pivot_table()` output with `.groupby()`

```
ind = (
    yf.download(tickers='^GSPC ^DJI ^IXIC ^FTSE ^N225 ^HSI')
    .rename_axis(columns=['Variable', 'Index'])
    .stack()
)
```

```
[**********100%%**********]  6 of 6 completed
```

```
(
    ind
```

```
    .loc['2015':]
    .reset_index()
    .pivot_table(
        values='Close',
        index=pd.Grouper(key='Date', freq='A'),
        columns='Index',
        aggfunc=['min', 'max']
    )
)
```

	min						max	
Index	^DJI	^FTSE	^GSPC	^HSI	^IXIC	^N225	^DJI	^FTS
Date								
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
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	37557.9219	8014.2

16.2.2 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.

16.2.3 Calculate the mean and standard deviation of returns and the maximum of closing prices by ticker for the MATANA stocks

Again, consider only dates with complete returns data. Try this calculation with wide and long data frames, and confirm your results are the same.

16.2.4 Calculate monthly means and volatilities for SPY and GOOG returns

16.2.5 Plot the monthly means and volatilities from the previous exercise

16.2.6 Assign the Dow Jones stocks to five portfolios based on their monthly volatility

First, we need to download Dow Jones stock data and calculate daily returns. Use data from 2019 through today.

16.2.7 Plot the time-series volatilities of these five portfolios

How do these portfolio volatilies compare to (1) each other and (2) the mean volatility of their constituent stocks?

16.2.8 Calculate the *mean* monthly correlation between the Dow Jones stocks

Drop duplicate correlations and self correlations (i.e., correlation between AAPL and AAPL), which are 1, by definition.

16.2.9 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

Part IX

Week 9

17 McKinney Chapter 11 - Time Series

17.1 Introduction

Chapter 11 of Wes McKinney's *Python for Data Analysis* discusses time series and panel data, which is where pandas ***shines***. 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 rolling windows

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

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
```

```
%precision 4
pd.options.display.float_format = '{:.4f}'.format
%config InlineBackend.figure_format = 'retina'
```

```
import yfinance as yf
import pandas_datareader as pdr
```

McKinney provides an excellent introduction to the concept of time series and panel data:

> Time series data is an important form of structured data in many different fields, such as finance, economics, ecology, neuroscience, and physics. Anything that is observed or measured at many points in time forms a time series. Many time series are fixed frequency, which is to say that data points occur at regular intervals according to some rule, such as every 15 seconds, every 5 minutes, or once per month. Time series can also be irregular without a fixed unit of time or offset

between units. How you mark and refer to time series data depends on the application, and you may have one of the following: - Timestamps, specific instants in time - Fixed periods, such as the month January 2007 or the full year 2010 - Intervals of time, indicated by a start and end timestamp. Periods can be thought of as special cases of intervals - Experiment or elapsed time; each timestamp is a measure of time relative to a particular start time (e.g., the diameter of a cookie baking each second since being placed in the oven)

In this chapter, I am mainly concerned with time series in the first three categories, though many of the techniques can be applied to experimental time series where the index may be an integer or floating-point number indicating elapsed time from the start of the experiment. The simplest and most widely used kind of time series are those indexed by timestamp. 323

pandas provides many built-in time series tools and data algorithms. You can effi- ciently work with very large time series and easily slice and dice, aggregate, and resample irregular- and fixed-frequency time series. Some of these tools are especially useful for financial and economics applications, but you could certainly use them to analyze server log data, too.

17.2 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 arithmetic operations between pandas objects automatically align on indexes.

```
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
```

17.2.1 Indexing, Selection, Subsetting

We can use date and time labels to select data.

```
stamp = ts.index[2]
stamp
```

```
Timestamp('2011-01-07 00:00:00')
```

```
ts[stamp]
```

```
0.6477
```

pandas uses unambiguous date strings to select data.

```
ts['1/10/2011'] # M/D/YYYY
```

```
-0.2342
```

```
ts['20110110'] # YYYYMMDD
```

```
-0.2342
```

```
ts['2011-01-10'] # YYYY-MM-DD
```

```
-0.2342
```

```
ts['10-Jan-2011'] # D-Mon-YYYY
```

```
-0.2342
```

```
ts['Jan-10-2011'] # Mon-D-YYYY
```

```
-0.2342
```

But do not use U.K.-style dates.

```
# ts['10/1/2011'] # D/M/YYYY # KeyError: '10/1/2011'
```

Here is a longer time series for longer slices.

```
np.random.seed(42)
longer_ts = pd.Series(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 pass a year-month to slice all of the observations in May of 2001.

```
longer_ts['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 pass a year to slice all observations in 2001.

```
longer_ts['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['1/6/2011':'1/10/2011']
```

```
2011-01-07    0.6477
```

```
2011-01-08    1.5230
2011-01-10   -0.2342
dtype: float64
```

To use date slices, our data should be sorted by the date index, as above. The following code works as though our data were sorted, but raises a warning that it will not work in future versions.

```
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 is ambiguous and fails.

```
# ts2['1/6/2011':'1/11/2011'] # KeyError: 'Value based partial slicing on non-monotonic Da
```

```
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
```

To be clear, a range of date strings is inclusive on both ends.

```
longer_ts['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, if we modify a slice, we modify the original series or dataframe.

> 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.

17.2.2 Time Series with Duplicate Indices

Most data in this course will be well-formed with one observation per datetime for series or one observation per individual per datetime for dataframes. However, you may later receive poorly-formed data with duplicate observations. The toy data in 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(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: int32
```

The `.is_unique` property tells us if an index is unique.

```
dup_ts.index.is_unique
```

```
False
```

```
dup_ts['1/3/2000']  # not duplicated
```

```
4
```

```
dup_ts['1/2/2000']  # duplicated
```

```
2000-01-02    1
2000-01-02    2
2000-01-02    3
dtype: int32
```

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.

```
grouped = dup_ts.groupby(level=0)
```

```
grouped.mean()
```

```
2000-01-01   0.0000
2000-01-02   2.0000
2000-01-03   4.0000
dtype: float64
```

```
grouped.last()
```

```
2000-01-01    0
2000-01-02    3
2000-01-03    4
dtype: int32
```

Or we may want the number of observations on each date.

```
grouped.count()
```

```
2000-01-01    1
2000-01-02    3
2000-01-03    1
dtype: int64
```

17.3 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.

We will skip the sections on creating date ranges or different frequencies so we can focus on shifting data.

17.3.1 Shifting (Leading and Lagging) Data

Shifting is an important feature! Shifting is moving data backward (or forward) through time.

```
np.random.seed(42)
ts = pd.Series(np.random.randn(4), index=pd.date_range('1/1/2000', periods=4, freq='M'))

ts
```

```
2000-01-31    0.4967
2000-02-29   -0.1383
2000-03-31    0.6477
2000-04-30    1.5230
Freq: M, dtype: float64
```

If we pass a positive integer N to the `.shift()` method:

1. The date index remains the same
2. Values are shifted down N observations

"Lag" might be a better name than "shift" since a postive 2 makes the value at any timestamp the value from 2 timestamps above (earlier, since most time-series data are chronological).

```
ts.shift() # if we do not specify "periods", pandas assumes 1
```

```
2000-01-31       NaN
2000-02-29    0.4967
2000-03-31   -0.1383
```

```
2000-04-30    0.6477
Freq: M, dtype: float64
```

```
ts.shift(2)
```

```
2000-01-31       NaN
2000-02-29       NaN
2000-03-31    0.4967
2000-04-30   -0.1383
Freq: M, dtype: float64
```

If we pass a *negative* integer N to the `.shift()` method, values are shifted *up* N observations.

```
ts.shift(-2)
```

```
2000-01-31   0.6477
2000-02-29   1.5230
2000-03-31      NaN
2000-04-30      NaN
Freq: M, dtype: float64
```

We will almost never shift with negative values (i.e., we will almost never bring forward values from the future) to prevent look-ahead bias. We do not want to assume that financial market participants have access to future data. Our most common shift will be to compute the percent change from one period to the next. We can calculate the percent change two ways.

```
ts.pct_change()
```

```
2000-01-31       NaN
2000-02-29   -1.2784
2000-03-31   -5.6844
2000-04-30    1.3515
Freq: M, dtype: float64
```

```
(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: M, dtype: float64
```

We can use `np.allclose()` to test that the two return calculations above are the same.

```
np.allclose(
    a=ts.pct_change(),
    b=(ts - ts.shift()) / ts.shift(),
    equal_nan=True
)
```

```
True
```

Two observations on the percent change calculations above:

1. The first percent change is NaN (missing) because there is no previous value to change from
2. The default `periods` argument for `.shift()` and `.pct_change()` is 1

The naive shift examples above shift by a number of observations, without considering timestamps or their frequencies. As a result, timestamps are unchanged and values shift down (positive `periods` argument) or up (negative `periods` argument). However, we can also pass the `freq` argument to respect the timestamps. With the `freq` argument, timestamps shift by a multiple (specified by the `periods` argument) of datetime intervals (specified by the `freq` argument). Note that the examples below generate new datetime indexes.

```
ts
```

```
2000-01-31    0.4967
2000-02-29   -0.1383
2000-03-31    0.6477
2000-04-30    1.5230
Freq: M, dtype: float64
```

```
ts.shift(2, freq='M')
```

```
2000-03-31    0.4967
2000-04-30   -0.1383
2000-05-31    0.6477
2000-06-30    1.5230
Freq: M, dtype: float64
```

```
ts.shift(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 `T` is minutes.

```
ts.shift(1, freq='90T')
```

```
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
```

17.3.2 Shifting dates with offsets

We can also shift timestamps to the beginning or end of a period or interval.

```
from pandas.tseries.offsets import Day, MonthEnd
now = datetime(2011, 11, 17)
now + 3 * Day()
```

```
Timestamp('2011-11-20 00:00:00')
```

```
now + MonthEnd(0) # 0 is for move to the end of the month, but never leave the month
```

```
Timestamp('2011-11-30 00:00:00')
```

```
now + MonthEnd(1) # 1 is for move to the end of the month, if already at end, move to the
```

```
Timestamp('2011-11-30 00:00:00')
```

```
now + MonthEnd(1) + MonthEnd(1) # 1 is for move to the end of the month, if already at end
```

```
Timestamp('2011-12-31 00:00:00')
```

```
now + MonthEnd(2)
```

```
Timestamp('2011-12-31 00:00:00')
```

Date offsets can help us align data for presentation or merging. ***But, be careful!*** The default argument is 1, but we typically want 0.

```
datetime(2021, 10, 30) + MonthEnd(0)
```

```
Timestamp('2021-10-31 00:00:00')
```

```
datetime(2021, 10, 30) + MonthEnd(1)
```

```
Timestamp('2021-10-31 00:00:00')
```

```
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')
```

17.4 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 the `.groupby()` method syntax. This similarity is because `.resample()` is syntactic sugar for `.groupby()`.

17.4.1 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('2000-01-01', periods=12, freq='T')
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: T, dtype: int32
```

We can aggregate the one-minute frequency data above to a five-minute frequency. Resampling requires and aggregation method, and here McKinney chooses the `.sum()` method.

```
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: 5T, dtype: int32
```

Two observations about the previous resampling example:

1. For minute-frequency resampling, the default is that the new data are labeled by the left edge of the resampling interval
2. For minute-frequency resampling, the default is that the left edge is closed (included) and the right edge is open (excluded)

As a result, the first value of 10 at midnight is the sum of values at midnight and to the right of midnight, not including the value at 00:05 (i.e., $10 = 0 + 1 + 2 + 3 + 4$ at 00:00 and $35 = 5 + 6 + 7 + 8 + 9$ at 00:05). We can use the `closed` and `label` arguments to change this behavior.

In finance, we prefer `closed='right'` and `label='right'`.

```
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: 5T, dtype: int32
```

Mixed combinations of `closed` and `label` are possible but confusing.

```
ts.resample('5min', closed='right', label='left').sum()
```

```
1999-12-31 23:55:00     0
2000-01-01 00:00:00    15
2000-01-01 00:05:00    40
2000-01-01 00:10:00    11
Freq: 5T, dtype: int32
```

These defaults for minute-frequency data may seem odd, but any choice is arbitrary. I suggest you do the following when you use the `.resample()` method:

1. Read the docstring
2. Check your output

pandas (and the `.resample()` method) are mature and widely used, so the defaults are typically reasonable.

17.4.2 Upsampling and Interpolation

To downsample (i.e., resample from higher frequency to lower frequency), we have to choose an aggregation method (e.g., `.mean()`, `.sum()`, `.first()`, or `.last()`). To upsample (i.e., resample from lower frequency to higher frequency), we do not have to choose an aggregation method.

```
np.random.seed(42)
frame = pd.DataFrame(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 can use the `.asfreq()` method to convert to the new frequency "as is".

```
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 do not *have* to choose an aggregation (disaggregation?) method, but we may want to choose a method to fill in the 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

17.5 Moving Window Functions

Moving window (or rolling window) functions are one of the neatest features of pandas, and we will frequently use moving window functions. We will use data similar, but not identical, to the book data. ***We must remove the timezone if we want to merge with Fama-French data.***

```
df = (
    yf.download(tickers=['AAPL', 'MSFT', 'SPY'])
    .assign(Date = lambda x: x.index.tz_localize(None))
    .set_index('Date')
    .rename_axis(columns=['Variable', 'Ticker'])
)

df.head()
```

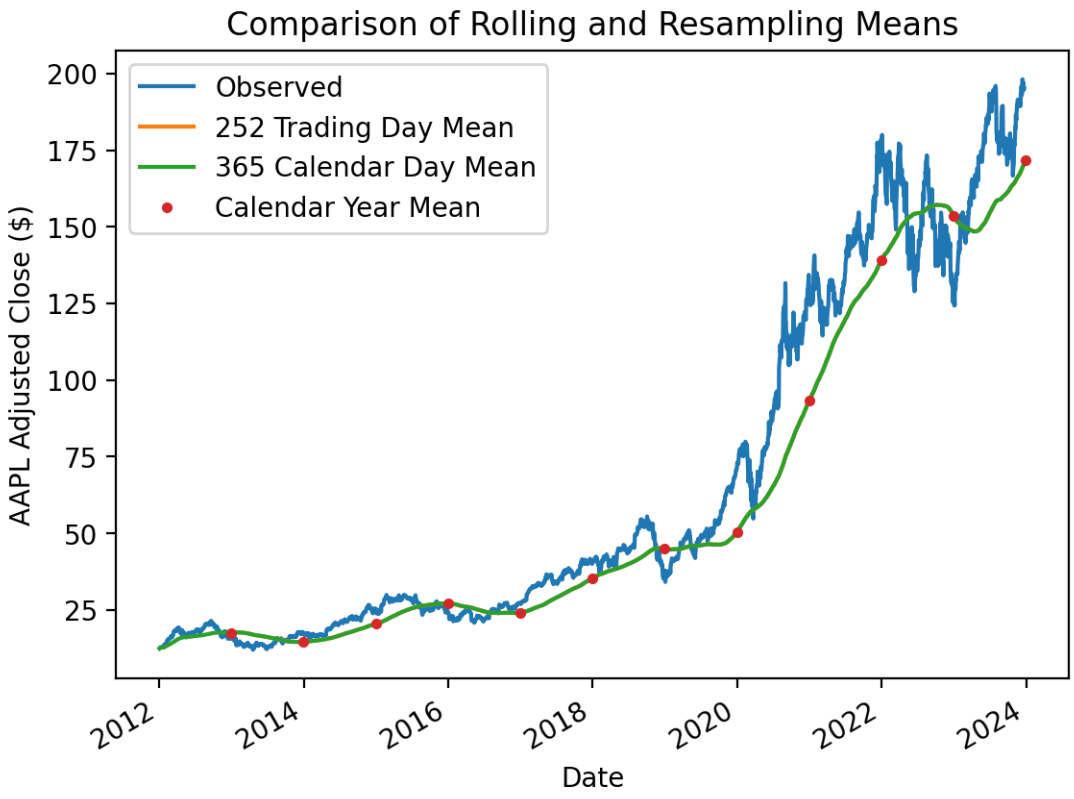
```
[**********100%%**********]  3 of 3 completed
```

Variable	Adj Close			Close			High			Low		
Ticker	AAPL	MSFT	SPY	AAPL	MSFT	SPY	AAPL	MSFT	SPY	AAPL	MSFT	SPY
Date												
1980-12-12	0.0993	NaN	NaN	0.1283	NaN	NaN	0.1289	NaN	NaN	0.1283	NaN	NaN
1980-12-15	0.0941	NaN	NaN	0.1217	NaN	NaN	0.1222	NaN	NaN	0.1217	NaN	NaN
1980-12-16	0.0872	NaN	NaN	0.1127	NaN	NaN	0.1133	NaN	NaN	0.1127	NaN	NaN
1980-12-17	0.0894	NaN	NaN	0.1155	NaN	NaN	0.1161	NaN	NaN	0.1155	NaN	NaN
1980-12-18	0.0920	NaN	NaN	0.1189	NaN	NaN	0.1194	NaN	NaN	0.1189	NaN	NaN

The `.rolling()` method is similar to the `.groupby()` and `.resample()` methods. The `.rolling()` method accepts a window-width and requires an aggregation method. The next

example calculates and plots the 252-trading day moving average of AAPL's price alongside the daily price.

```
aapl = df.loc['2012':, ('Adj Close', 'AAPL')]
aapl.plot(label='Observed')
aapl.rolling(252).mean().plot(label='252 Trading Day Mean') # min_periods defaults to 252
aapl.rolling('365D').mean().plot(label='365 Calendar Day Mean') # min_periods defaults to
aapl.resample('A').mean().plot(style='.', label='Calendar Year Mean')
plt.legend()
plt.ylabel('AAPL Adjusted Close ($)')
plt.title('Comparison of Rolling and Resampling Means')
plt.show()
```



Two observations:

1. If we pass the window-width as an integer, the window-width is based on the number of observations and ignores time stamps
2. If we pass the window-width as an integer, the `.rolling()` method requires that number

of observations for all windows (i.e., note that the moving average starts 251 trading days after the first daily price

We can use the `min_periods` argument 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`.

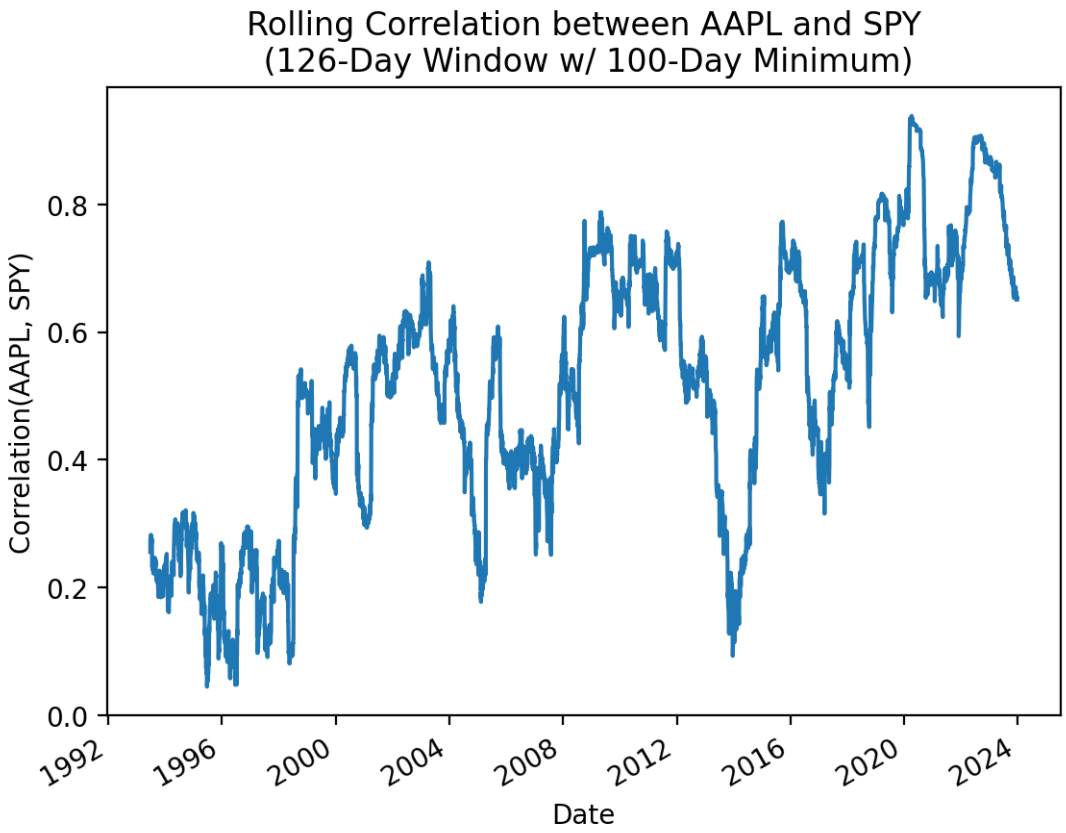
17.5.1 Binary Moving Window Functions

Binary moving window functions accept two inputs. The most common example is the rolling correlation between two returns series.

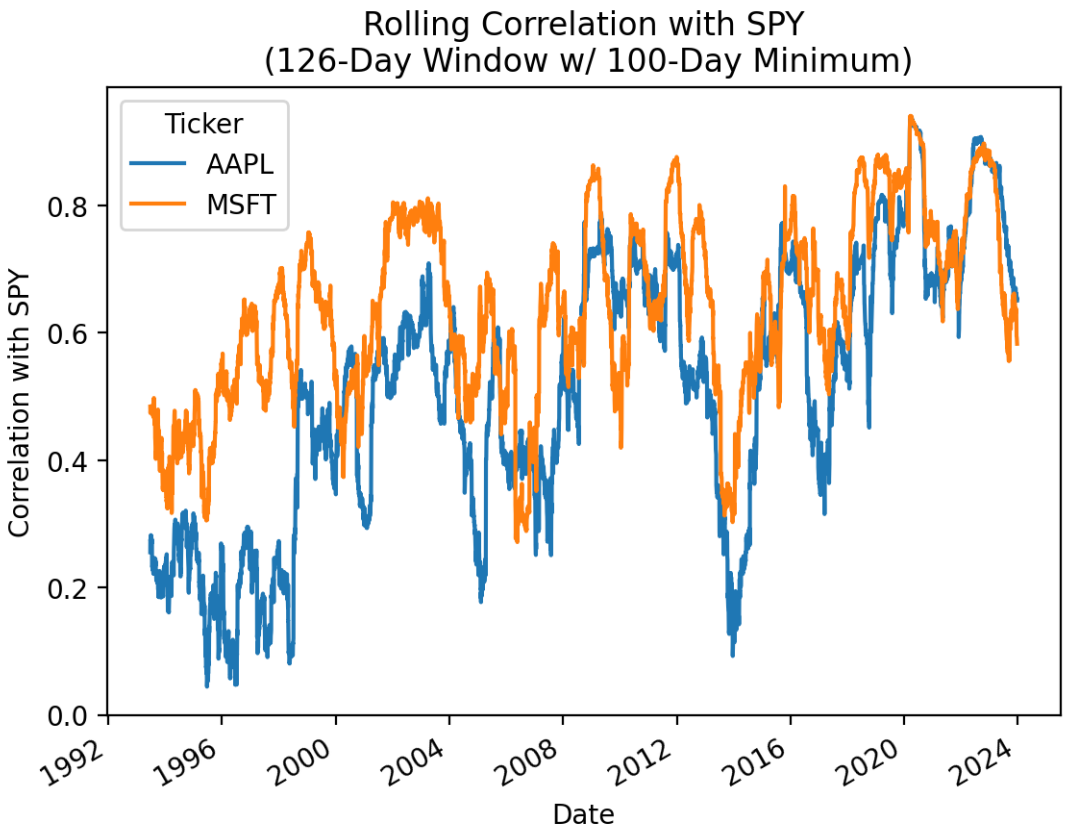
```
returns = df['Adj Close'].pct_change()

returns.head()
```

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

```
returns['AAPL'].rolling(126, min_periods=100).corr(returns['SPY']).plot()
plt.ylabel('Correlation(AAPL, 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()
```

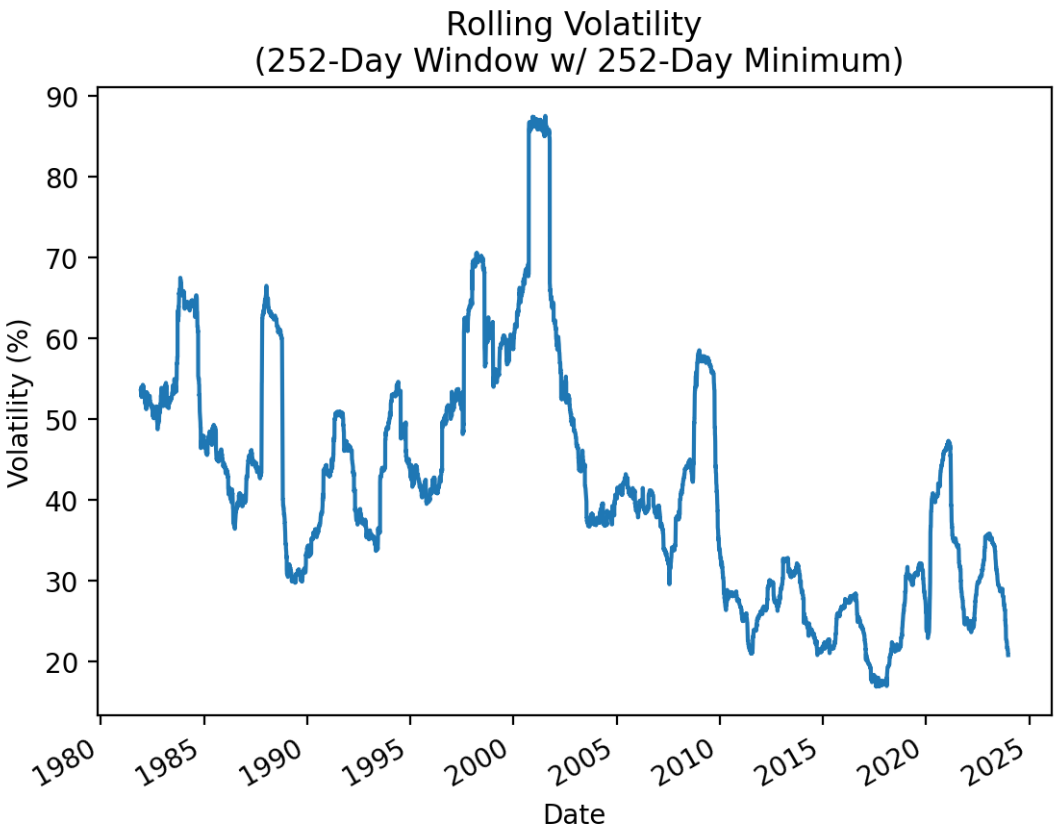


17.5.2 User-Defined Moving Window Functions

Finally, we can define our own moving window functions and use the `.apply()` method to apply them However, note that `.apply()` will be much slower than the the optimized moving window functions (e.g., `.mean()`, `.std()`, etc.).

McKinney provides an abstract example here, but we will discuss a simpler example that calculates rolling volatility. Also, calculating rolling volatility with the `.apply()` method provides us a chance to benchmark it against the optimized version.

```
returns['AAPL'].rolling(252).apply(np.std).mul(np.sqrt(252) * 100).plot() # annualize and
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 typically 1000-times slower than the pre-built method.

```
%timeit returns['AAPL'].rolling(252).apply(np.std)
```

```
936 ms ± 160 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
%timeit returns['AAPL'].rolling(252).std()
```

```
337 µs ± 58 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

18 McKinney Chapter 11 - Practice for Section 04

18.1 Announcements

18.2 Practice

18.2.1 Cumulative returns with all available data

18.2.2 Total returns for each calendar year

18.2.3 Total returns over rolling 252-trading-day windows

18.2.4 Total returns over rolling 12-months windows after calculating monthly returns

18.2.5 Sharpe Ratios for each calendar year

18.2.6 Calculate rolling betas

Calculate rolling capital asset pricing model (CAPM) betas for the MATANA stocks.

The CAPM says the risk premium on a stock depends on the risk-free rate, beta, and the risk premium on the market: $E(R_{stock}) = R_f + \beta_{stock} \times (E(R_{market}) - R_f)$. We can calculate CAPM betas as: $\beta_{stock} = \frac{Cov(R_{stock} - R_f, R_{market} - R_f)}{Var(R_{market} - R_f)}$.

18.2.7 Calculate rolling Sharpe Ratios

Calculate rolling Sharpe Ratios for the MATANA stocks.

The Sharpe Ratio is often used to evaluate fund managers. The Sharpe Ratio is $SR_i = \frac{\overline{R_i - R_f}}{\sigma}$, where $\overline{R_i - R_f}$ is mean fund return relative to the risk-free rate over some period and σ is the standard deviation of $R_i - R_f$ over the same period. While the Sharpe Ratio is typically used for funds, we can apply it to a single stock to test our knowledge of the `.rolling()`

method. Calculate and plot the one-year rolling Sharpe Ratio for the MATANA stocks using all available daily data.

Part X

Week 10

19 Herron Topic 2 - Trading Strategies

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 chapter 12 of Ivo Welch's *Corporate Finance* textbook and chapter 2 of Eryk Lewinson's *Python for Finance Cookbook*. The practice notebook will cover several other trading strategies based on technical analysis.

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
```

```
%precision 4
pd.options.display.float_format = '{:.4f}'.format
%config InlineBackend.figure_format = 'retina'
```

19.1 What is technical analysis?

Technical analysis is a methodology that analyzes past market data (e.g., past prices and volume) in an attempt to forecast future price movements. If technical analysis can predict future price movements, the market is not weak-form efficient. Ivo Welch provides the three degrees of market efficiency in section 12.2 of chapter 12 of his (free) *Corporate Finance* textbook:

> 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 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.

Section 12.2 goes on to provide Welch's 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. We will focus on technical analysis in this notebook, but chapter 12 is excellent.

19.2 Why might trading strategies based on technical analysis work or not?

19.2.1 ...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.

19.2.2 ...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.

19.2.2.1 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 $P_t = \rho P_{t-1} + mP_{t-1} + \varepsilon$, where m is a small drift term and $E[\varepsilon] = 0$. If $\rho > 1$, prices would quickly increase, and, if $\rho < 1$, prices would quickly decrease. Let us examine the historical record.

```
import pandas_datareader as pdr
```

```
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)
)

ff.head()
```

```
C:\Users\r.herron\AppData\Local\Temp\ipykernel_14600\3181934010.py:2: FutureWarning: The argu
  pdr.DataReader(
```

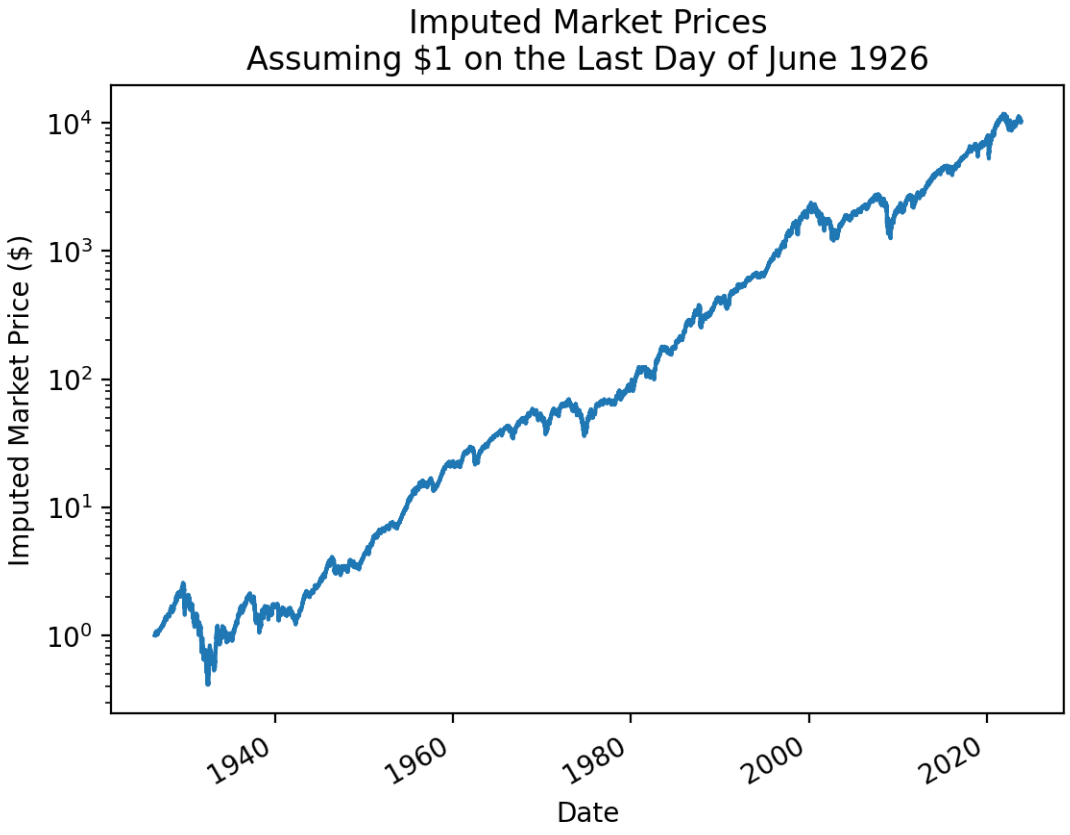
Date	Mkt-RF	SMB	HML	RF	Mkt
1926-07-01	0.0010	-0.0025	-0.0027	0.0001	0.0011
1926-07-02	0.0045	-0.0033	-0.0006	0.0001	0.0046
1926-07-06	0.0017	0.0030	-0.0039	0.0001	0.0018
1926-07-07	0.0009	-0.0058	0.0002	0.0001	0.0010
1926-07-08	0.0021	-0.0038	0.0019	0.0001	0.0022

We can use market returns to impute market prices relative to the last day of June 1926.

```
price = ff['Mkt'].add(1).cumprod()

price.tail()
```

```
Date
2023-10-25   10168.8352
2023-10-26   10054.0290
2023-10-27   10002.8540
2023-10-30   10119.9874
2023-10-31   10185.8686
Name: Mkt, dtype: float64
```

```
price.plot()
plt.title('Imputed Market Prices\nAssuming $1 on the Last Day of June 1926')
plt.ylabel('Imputed Market Price ($)')
plt.semilogy()
plt.show()
```

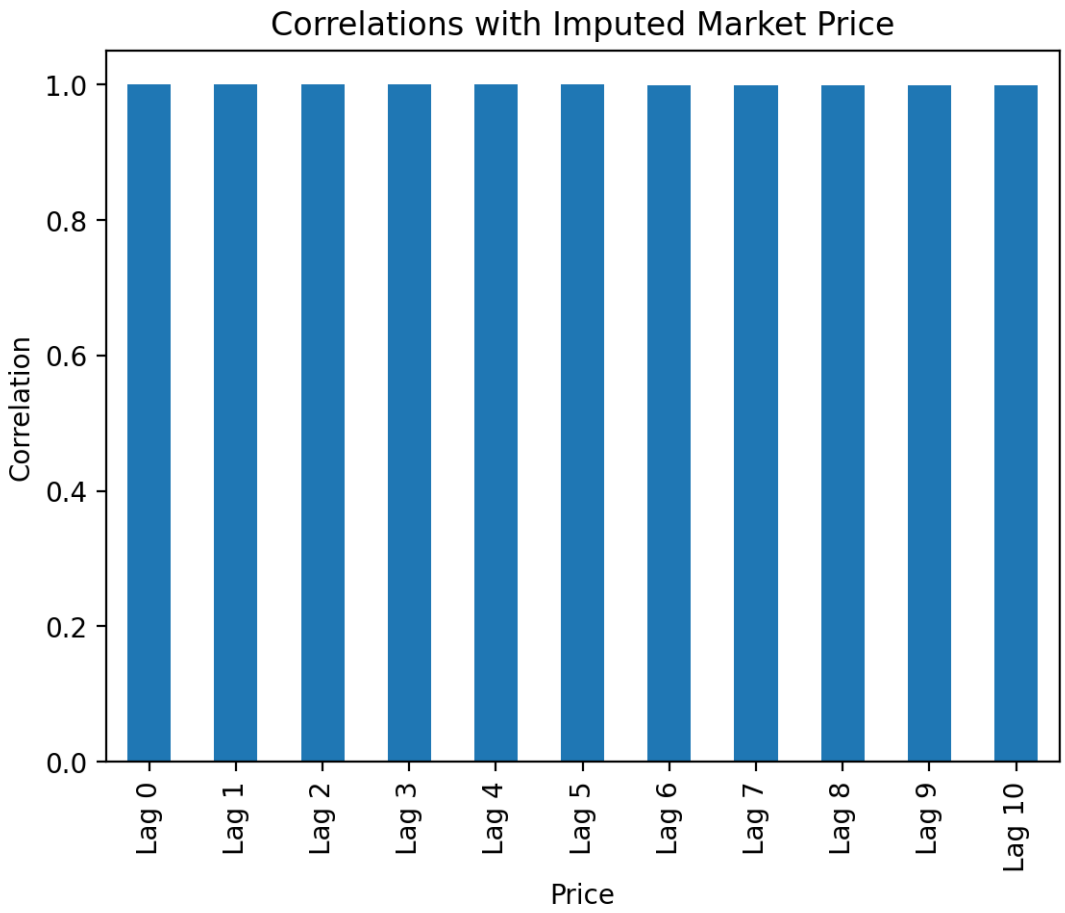


We need lagged prices to estimate ρ. We will add 10 lags of P to help us understand the relation between past and future prices.

```
price_lags = (
    pd.concat(
        objs=[price.shift(t) for t in range(11)],
        keys=[f'Lag {t}' for t in range(11)],
        names=['Price'],
        axis=1,
    )
)

price_lags.head()
```

Price Date	Lag 0	Lag 1	Lag 2	Lag 3	Lag 4	Lag 5	Lag 6	Lag 7	Lag 8	Lag 9	Lag 10
1926-07-01	1.0011	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
1926-07-02	1.0057	1.0011	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
1926-07-06	1.0075	1.0057	1.0011	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
1926-07-07	1.0085	1.0075	1.0057	1.0011	NaN	NaN	NaN	NaN	NaN	NaN	NaN
1926-07-08	1.0107	1.0085	1.0075	1.0057	1.0011	NaN	NaN	NaN	NaN	NaN	NaN

```
(
    price_lags
    .dropna()
    .corr()
    .loc['Lag 0']
    .plot(kind='bar')
)
plt.title('Correlations with Imputed Market Price')
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!

```
import statsmodels.api as sm
```

```
_ = price_lags.dropna()
y = _['Lag 0']
X = _.drop('Lag 0', axis=1).pipe(sm.add_constant)
model = sm.OLS(endog=y, exog=X)
fit = model.fit(cov_type='HAC', cov_kwds={'maxlags': 10})
fit.summary()
```

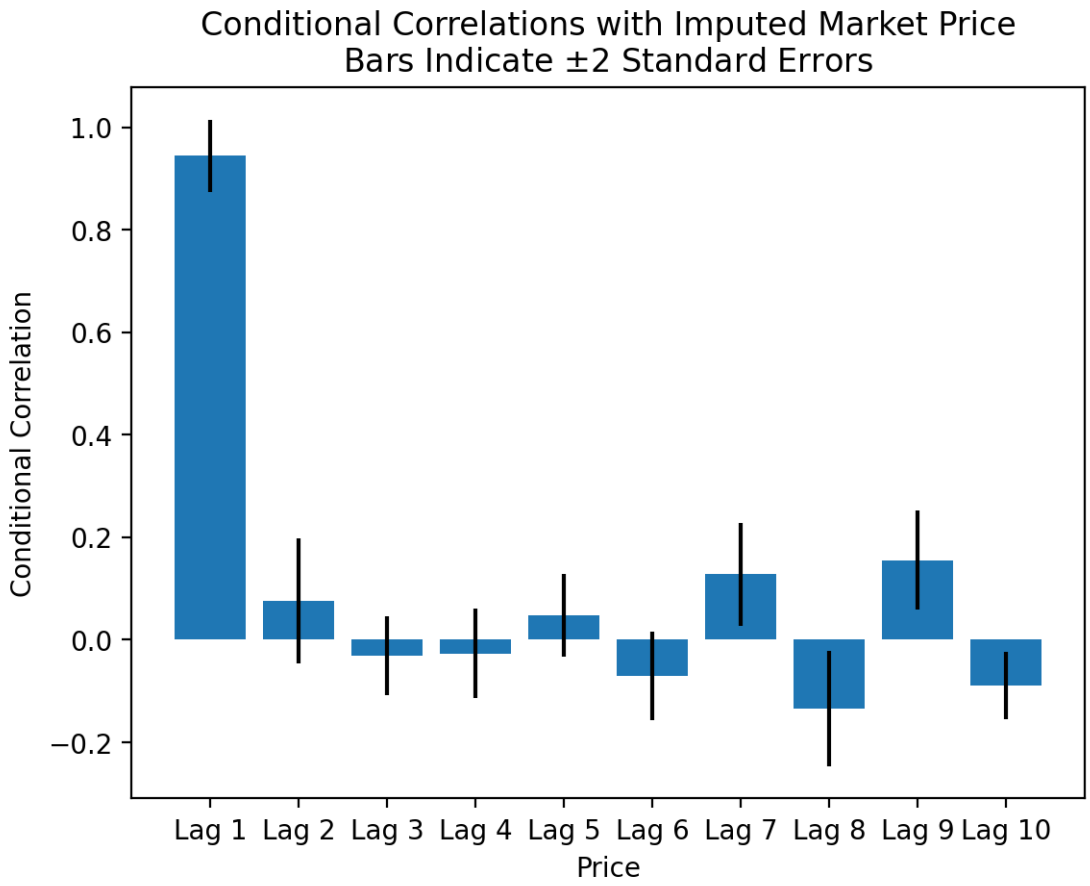
Dep. Variable:	Lag 0	**R-squared:**	1.000
Model:	OLS	**Adj. R-squared:**	1.000
Method:	Least Squares	**F-statistic:**	1.768e+06
Date:	Fri, 22 Dec 2023	**Prob (F-statistic):**	0.00
Time:	10:00:20	**Log-Likelihood:**	-1.2187e+05
No. Observations:	25598	**AIC:**	2.438e+05
Df Residuals:	25587	**BIC:**	2.438e+05
Df Model:	10		
Covariance Type:	HAC		

	coef	**std err**	**z**	**P> |z|**	**[0.025**	**0.975]**
const	0.1283	0.145	0.888	0.375	-0.155	0.412
Lag 1	0.9447	0.035	26.932	0.000	0.876	1.013
Lag 2	0.0764	0.061	1.256	0.209	-0.043	0.196
Lag 3	-0.0311	0.039	-0.805	0.421	-0.107	0.045
Lag 4	-0.0268	0.043	-0.618	0.536	-0.112	0.058
Lag 5	0.0477	0.040	1.187	0.235	-0.031	0.127
Lag 6	-0.0706	0.043	-1.651	0.099	-0.154	0.013
Lag 7	0.1277	0.050	2.548	0.011	0.029	0.226
Lag 8	-0.1339	0.056	-2.373	0.018	-0.244	-0.023
Lag 9	0.1552	0.048	3.215	0.001	0.061	0.250
Lag 10	-0.0890	0.033	-2.729	0.006	-0.153	-0.025

Omnibus:	16175.844	**Durbin-Watson:**	1.995
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	5235480.028
Skew:	-1.860	**Prob(JB):**	0.00
Kurtosis:	72.963	**Cond. No.**	8.70e+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, 8.7e+03. This might indicate that there are strong multicollinearity or other numerical problems.

```
plt.bar(
    x=price_lags.columns[1:],
    height=fit.params[1:],
    yerr=2*fit.bse[1:]
)
plt.title('Conditional Correlations with Imputed Market Price\nBars Indicate $\pm 2$ Stand
plt.ylabel('Conditional Correlation')
plt.xlabel('Price')
plt.show()
```



19.2.2.2 Signal-to-Noise Ratio

Recall, we can express a random walk as $P_t = \rho P_{t-1} + mP_{t-1} + \varepsilon_t$. 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 $E[e_t] = 0$ and $SD[e_t] = s$, so $E[r_{t-1,t}] = m$. 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 4 percent!

```
m/s
```

```
0.0388
```

Means grow linearly with time, while standard deviations growth with the square-root of time. Therefore, even with a 1 basis point signal, we need a lot of data to make sure this 1 basis point signal is real! For example, if we want $\sqrt{t} \times \frac{1}{s} \geq 2$, we need $t \geq \left(2 \times \frac{s}{1}\right)^2$ days! Even with market noise, which is diversified and low, we need more than 100 years of data!

```
(2 * s / 0.0001)**2 / 365
```

```
128.0929
```

19.3 Implement a simple moving average (SMA) trading strategy

One 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. Market prices below their SMA might be buying opportunities, and market prices below their SMA might be selling opportunities. Here, we will implement a long-only 20-day SMA (SMA(20)) 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

Because we will not sell short, we can simplify this strategy to "long if above SMA(20), otherwise neutral". First, we will need Bitcoin returns data.

```
import yfinance as yf
```

```
btc = (
    yf.download(tickers='BTC-USD')
    .assign(Return = lambda x: x['Adj Close'].pct_change())
    .rename_axis(columns='Variable')
)

btc.head()
```

```
[**********100%%**********]  1 of 1 completed
```

Variable Date	Open	High	Low	Close	Adj Close	Volume	Return
2014-09-17	465.8640	468.1740	452.4220	457.3340	457.3340	21056800	NaN
2014-09-18	456.8600	456.8600	413.1040	424.4400	424.4400	34483200	-0.0719
2014-09-19	424.1030	427.8350	384.5320	394.7960	394.7960	37919700	-0.0698
2014-09-20	394.6730	423.2960	389.8830	408.9040	408.9040	36863600	0.0357
2014-09-21	408.0850	412.4260	393.1810	398.8210	398.8210	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 could use `np.where()` instead of `np.select()`, but using `np.select()` provides a more flexible framework for more complex examples
    4. **We use `.shift()` to compare yesterday's closing prices, avoiding a look-ahead bias**
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

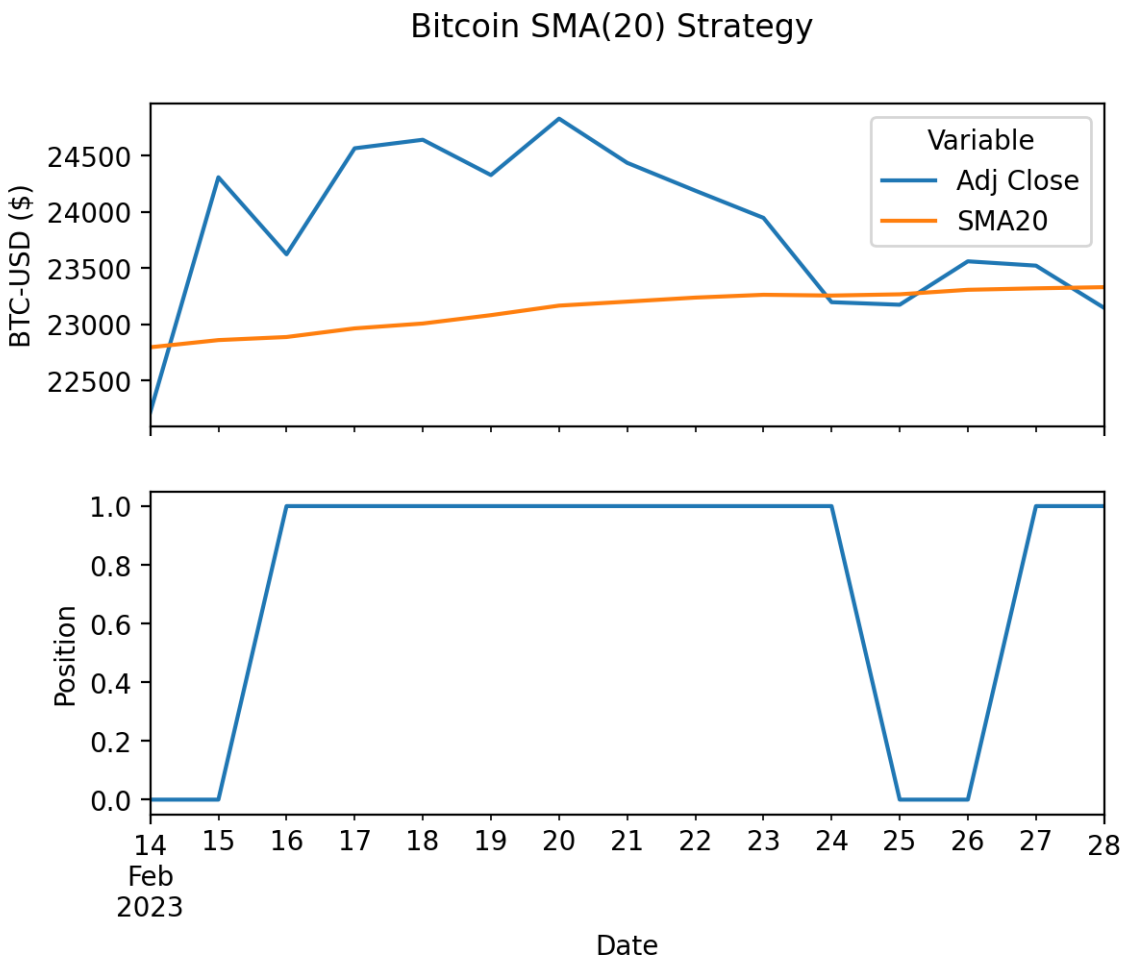
```
btc = (
    btc
    .assign(
```

```
        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()
            choicelist=[1, 0],
            default=np.nan
        ),
        Strategy = lambda x: x['Position'] * x['Return']
    )
)

btc.tail()
```

Variable Date	Open	High	Low	Close	Adj Close	Volume	Return	SMA2
2023-12-18	41348.2031	42720.2969	40530.2578	42623.5391	42623.5391	25224642008	0.0304	41762
2023-12-19	42641.5117	43354.2969	41826.3359	42270.5273	42270.5273	23171001281	-0.0083	41983
2023-12-20	42261.3008	44275.5859	42223.8164	43652.2500	43652.2500	27868908174	0.0327	42280
2023-12-21	43648.1250	44240.6680	43330.0508	43869.1523	43869.1523	22452766169	0.0050	42539
2023-12-22	43868.9883	44367.9570	43441.9688	43665.7266	43665.7266	23145105408	-0.0046	42748

I find it helpful to plot `Adj Close`, `SMA20`, and `Position` for a sort window with one or more crossings.

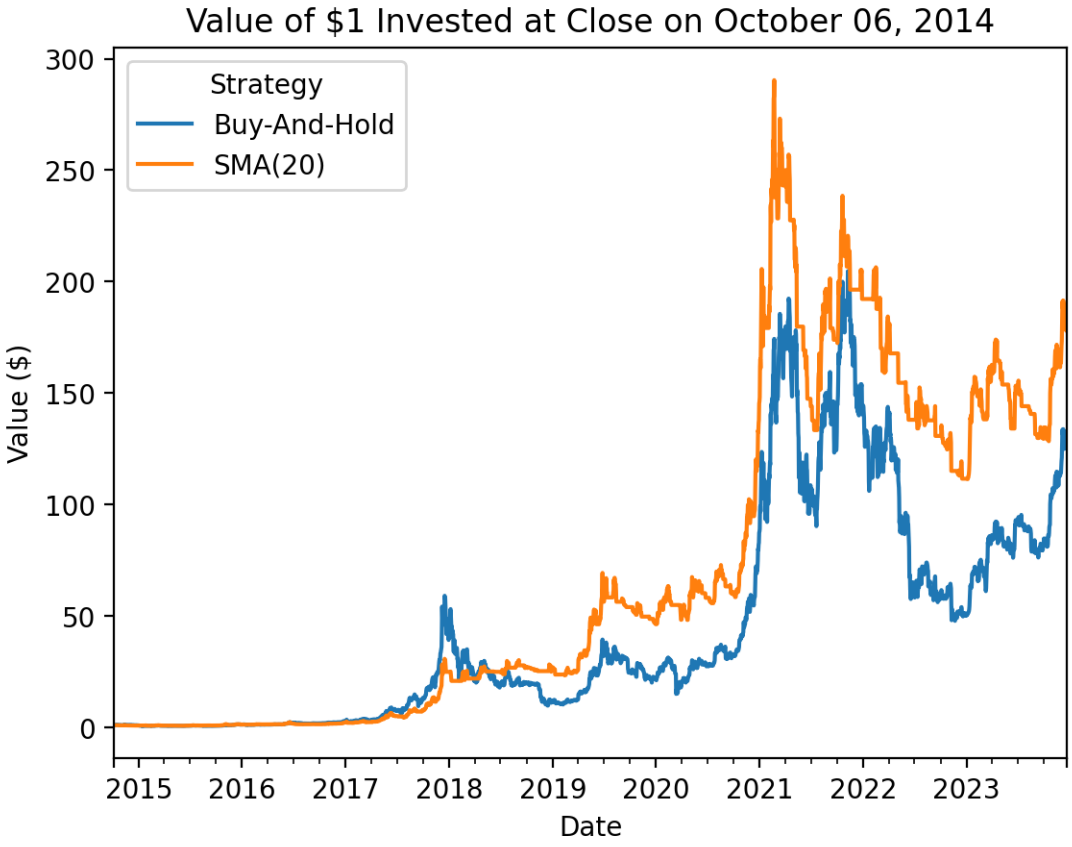
```
fig, ax = plt.subplots(2, 1, sharex=True)
_ = btc.loc['2023-02'].iloc[-15:]
_[['Adj Close', 'SMA20']].plot(ax=ax[0], ylabel='BTC-USD ($)')
_[['Position']].plot(ax=ax[1], ylabel='Position', legend=False)
plt.suptitle('Bitcoin SMA(20) Strategy')
plt.show()
```



We can compare the long-run performance of buy-and-hold and SMA(20).

```
_ = btc[['Return', 'Strategy']].dropna()

(
    _
    .add(1)
    .cumprod()
    .rename_axis(columns='Strategy')
    .rename(columns={'Return': 'Buy-And-Hold', 'Strategy': 'SMA(20)'})
    .plot()
)
plt.ylabel('Value ($)')
```

```
plt.title(f'Value of $1 Invested at Close on {_.index[0] - pd.offsets.Day(1):%B %d, %Y}')
plt.show()
```



In the practice notebook, we will dig deeper on this strategy and others.

20 Herron Topic 2 - Practice for Section 04

20.1 Announcements

20.2 Practice

20.2.1 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 `=`).

20.2.2 How does SMA(20) outperform buy-and-hold with this sample?

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.

20.2.3 Implement the SMA(20) strategy with the market factor from French

We need to impute a market price before we calculate SMA(20).

20.2.4 How often does SMA(20) outperform buy-and-hold with 10-year rolling windows?

20.2.5 Implement a long-only BB(20, 2) strategy with Bitcoin

More on Bollinger Bands here and here. In short, Bollinger Bands are bands around a trend, typically defined in terms of simple moving averages and volatilities. Here, long-only BB(20, 2) implies we have upper and lower bands at 2 standard deviations above and below SMA(20):

1. Buy when the closing price crosses LB(20) from below, where LB(20) is SMA(20) minus 2 sigma
2. Sell when the closing price crosses UB(20) from above, where UB(20) is SMA(20) plus 2 sigma
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.

20.2.6 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.

Here is the RSI formula: $RSI(n) = 100 - \frac{100}{1+RS(n)}$, where $RS(n) = \frac{SMA(U,n)}{SMA(D,n)}$. For "up days", $U = \Delta Adj\ Close$ and $D = 0$, and, for "down days", $U = 0$ and $D = -\Delta Adj\ Close$. Therefore, U and D are always non-negative. We can learn more about RSI here.

We will implement a long-short RSI(14) as follows:

1. Enter a long position when the RSI crosses 30 from below, and exit the position when the RSI crosses 50 from below
2. Enter a short position when the RSI crosses 70 from above, and exit the position when the RSI crosses 50 from above

Part XI

Week 11

21 Project 2

FINA 6333 – Spring 2023

To be determined

Part XII

Week 12

22 Herron Topic 3 - Multifactor Models

This notebook covers multifactor models, emphasizing the capital asset pricing model (CAPM) and the Fama-French three-factor model (FF3). Ivo Welch provides a high-level review of the CAPM and multifactor models in Chapter 10 of his free *Corporate Finance* textbook. The Wikipedia page for the CAPM is surprisingly good and includes its assumptions and shortcomings.

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
```

```
%precision 4
pd.options.display.float_format = '{:.4f}'.format
%config InlineBackend.figure_format = 'retina'
```

22.1 The Capital Asset Pricing Model (CAPM)

The CAPM explains the relation between non-diversifiable risk and expected return, and it has applications throughout finance. We use the CAPM to estimate costs of capital in corporate finance, assemble portfolios with a given risk-return tradeoff in portfolio management, and estimate expected rates of return in investment analysis. The formula for the CAPM is $E(R_i) = R_F + \beta_i[E(R_M) - R_F]$, where:

1. R_F is the risk-free rate of return,
2. β_i is the measure of non-diversifiable risk for asset i, and
3. $E(R_M)$ is the expected rate of return on the market.

Here, β_i measures asset i's risk exposure or sensitivity to market returns. The value-weighted mean of β_i's is 1 by construction, but a range of values is possible:

1. $\beta_i < -1$: Asset i moves in the opposite direction as the market, in larger magnitudes
2. $-1 \leq \beta_i < 0$: Asset i moves in the opposite direction as the market
3. $\beta_i = 0$: Asset i has no correlation between with the market
4. $0 < \beta_i \leq 1$: Asset i moves in the same direction as the market, in smaller magnitudes

5. $\beta_i = 1$: Asset i moves in the same direction with the same magnitude as the market
6. $\beta_i > 1$: Asset i moves in the same direction as the market, in larger magnitudes

22.1.1 β Estimation

We can use three (equivalent) approaches to estimate β_i:

1. From covariance and variance as $\beta_i = \frac{Cov(R_i - R_F, R_M - R_F)}{Var(R_M - R_F)}$
2. From correlation and standard deviations as $\beta_i = \rho_{i,M} \cdot \frac{\sigma_i}{\sigma_M}$, where all statistics use *excess* returns (i.e., $R_i - R_F$ and $R_M - R_F$)
3. From a linear regression of $R_i - R_F$ on $R_M - R_F$

The first two approaches are computationally simpler. However, the third approach also estimates the intercept α and goodness-of-fit statistics. We can use Apple (AAPL) to convince ourselves these three approaches are equivalent.

```
import yfinance as yf
import pandas_datareader as pdr
```

Note, we will leave returns in percent to make it easier to interpret our regression output! Leaving returns in percent does not affect the βs (slopes) but makes the α (intercept) easier to interpret by removing two leading zeros.

```
aapl = (
    yf.download(tickers='AAPL')
    .assign(Ri=lambda x: x['Adj Close'].pct_change().mul(100))
    .rename_axis(columns='Variable')
)

aapl.tail()
```

```
[**********************100%%**********************]  1 of 1 completed
```

Variable Date	Open	High	Low	Close	Adj Close	Volume	Ri
2023-12-18	196.0900	196.6300	194.3900	195.8900	195.8900	55751900	-0.8503
2023-12-19	196.1600	196.9500	195.8900	196.9400	196.9400	40714100	0.5360
2023-12-20	196.9000	197.6800	194.8300	194.8300	194.8300	52242800	-1.0714
2023-12-21	196.1000	197.0800	193.5000	194.6800	194.6800	46432100	-0.0770
2023-12-22	195.1800	195.4100	194.4250	195.2200	195.2200	5005484	0.2774

```
ff = (
    pdr.DataReader(
        name='F-F_Research_Data_Factors_daily',
        data_source='famafrench',
        start='1900',
    )
)
```

```
C:\Users\r.herron\AppData\Local\Temp\ipykernel_9472\3206174192.py:2: FutureWarning: The argu
  pdr.DataReader(
```

```
aapl = (
    aapl
    .join(ff[0])
    .assign(RiRF = lambda x: x['Ri'] - x['RF'])
    .rename(columns={'Mkt-RF': 'MktRF'})
)
```

```
aapl.head()
```

Date	Open	High	Low	Close	Adj Close	Volume	Ri	MktRF	SMB	HML
1980-12-12	0.1283	0.1289	0.1283	0.1283	0.0993	469033600	NaN	1.3800	-0.0100	-1.0500
1980-12-15	0.1222	0.1222	0.1217	0.1217	0.0941	175884800	-5.2171	0.1100	0.2500	-0.4600
1980-12-16	0.1133	0.1133	0.1127	0.1127	0.0872	105728000	-7.3398	0.7100	-0.7500	-0.4700
1980-12-17	0.1155	0.1161	0.1155	0.1155	0.0894	86441600	2.4750	1.5200	-0.8600	-0.3400
1980-12-18	0.1189	0.1194	0.1189	0.1189	0.0920	73449600	2.8993	0.4100	0.2200	1.2600

22.1.1.1 Covariance and Variance

```
vcv = aapl[['MktRF', 'RiRF']].dropna().cov()
vcv
```

	MktRF	RiRF
MktRF	1.2350	1.5564
RiRF	1.5564	7.8792

```
print(f"Apple beta from cov/var: {vcv.loc['MktRF', 'RiRF'] / vcv.loc['MktRF', 'MktRF']:0.4
```

```
Apple beta from cov/var: 1.2602
```

22.1.1.2 Correlation and Standard Deviations

```
_ = aapl[['MktRF', 'RiRF']].dropna()
rho = _.corr()
sigma = _.std()
print(f'rho:\n{rho}\n\nsigma:\n{sigma}')
```

```
rho:
       MktRF   RiRF
MktRF 1.0000 0.4989
RiRF  0.4989 1.0000

sigma:
MktRF   1.1113
RiRF    2.8070
dtype: float64
```

```
print(f"Apple beta from rho and sigmas: {rho.loc['MktRF', 'RiRF'] * sigma.loc['RiRF'] / si
```

```
Apple beta from rho and sigmas: 1.2602
```

22.1.1.3 Linear Regression

We will use the statsmodels package to estimate linear. I typically use the formula application programming interface (API).

```
import statsmodels.formula.api as smf
```

With statsmodels (and most Python model estimation packages), we have three steps:

1. Specify the model
2. Fit the model
3. Summarize the model

```
model = smf.ols('RiRF ~ MktRF', aapl)
fit = model.fit()
summary = fit.summary()
summary
```

Dep. Variable:	RiRF	**R-squared:**	0.249
Model:	OLS	**Adj. R-squared:**	0.249
Method:	Least Squares	**F-statistic:**	3583.
Date:	Fri, 22 Dec 2023	**Prob (F-statistic):**	0.00
Time:	09:59:32	**Log-Likelihood:**	-24950.
No. Observations:	10811	**AIC:**	4.990e+04
Df Residuals:	10809	**BIC:**	4.992e+04
Df Model:	1		
Covariance Type:	nonrobust		

	coef	**std err**	**t**	**P> |t|**	**[0.025**	**0.975]**
Intercept	0.0527	0.023	2.249	0.025	0.007	0.099
MktRF	1.2602	0.021	59.856	0.000	1.219	1.302

Omnibus:	3208.502	**Durbin-Watson:**	1.916
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	329571.227
Skew:	-0.380	**Prob(JB):**	0.00
Kurtosis:	30.038	**Cond. No.**	1.12

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

```
fit.params
```

```
Intercept   0.0527
MktRF       1.2602
dtype: float64
```

```
print(f"Apple beta from linear regression: {fit.params['MktRF']:0.4f}")
```

```
Apple beta from linear regression: 1.2602
```

We can chain these operations, but it often makes sense to save the intermediate results (i.e., `model` and `fit`).

```
smf.ols('RiRF ~ MktRF', aapl).fit().summary()
```

Dep. Variable:	RiRF	**R-squared:**	0.249
Model:	OLS	**Adj. R-squared:**	0.249
Method:	Least Squares	**F-statistic:**	3583.
Date:	Fri, 22 Dec 2023	**Prob (F-statistic):**	0.00
Time:	09:59:32	**Log-Likelihood:**	-24950.
No. Observations:	10811	**AIC:**	4.990e+04
Df Residuals:	10809	**BIC:**	4.992e+04
Df Model:	1		
Covariance Type:	nonrobust		

	coef	**std err**	**t**	**P> |t|**	**[0.025**	**0.975]**
Intercept	0.0527	0.023	2.249	0.025	0.007	0.099
MktRF	1.2602	0.021	59.856	0.000	1.219	1.302

Omnibus:	3208.502	**Durbin-Watson:**	1.916
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	329571.227
Skew:	-0.380	**Prob(JB):**	0.00
Kurtosis:	30.038	**Cond. No.**	1.12

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

22.1.2 β Visualization

We can visualize Apple's β, using seaborn's `regplot()` to add a best-fit line.

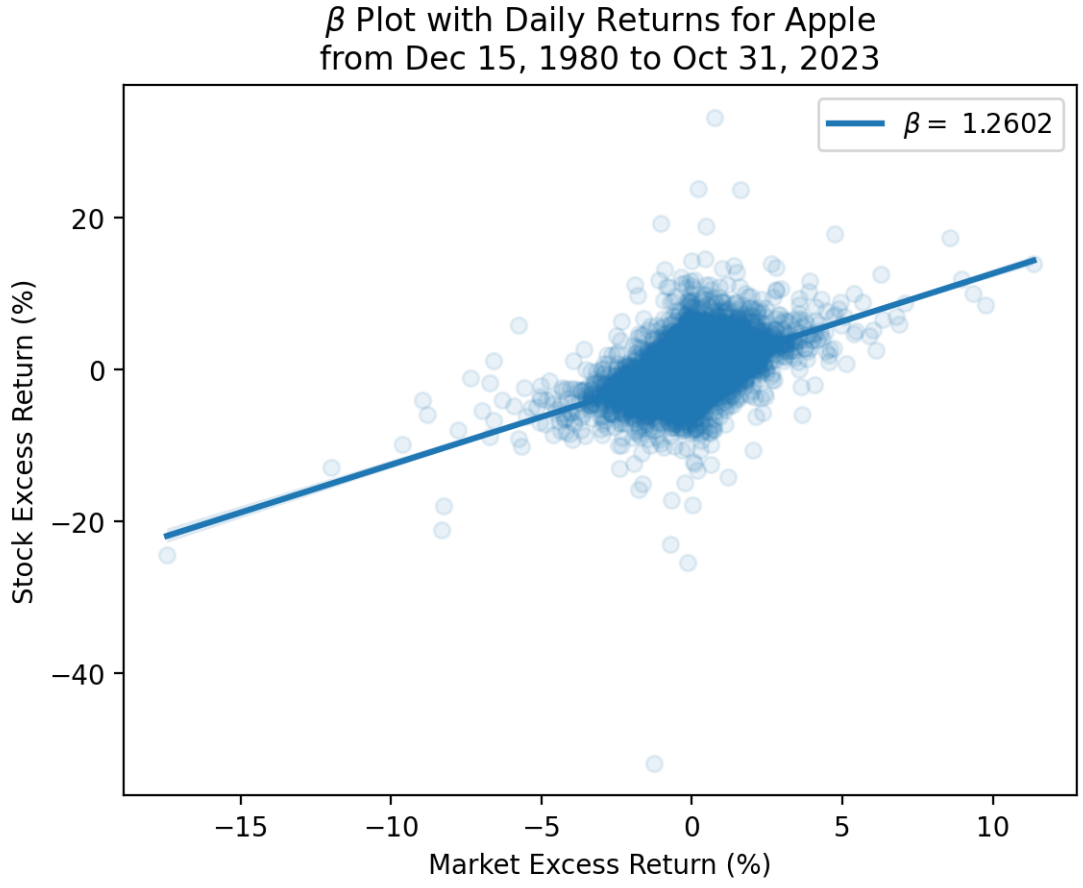
```
import seaborn as sns
```

We can write a couple of function to more easily make prettier plots.

```
def label_beta(x):
    vcv = x.dropna().cov()
    beta = vcv.loc['RiRF', 'MktRF'] / vcv.loc['MktRF', 'MktRF']
    return r'$\beta=$' + f'{beta: 0.4f}'
```

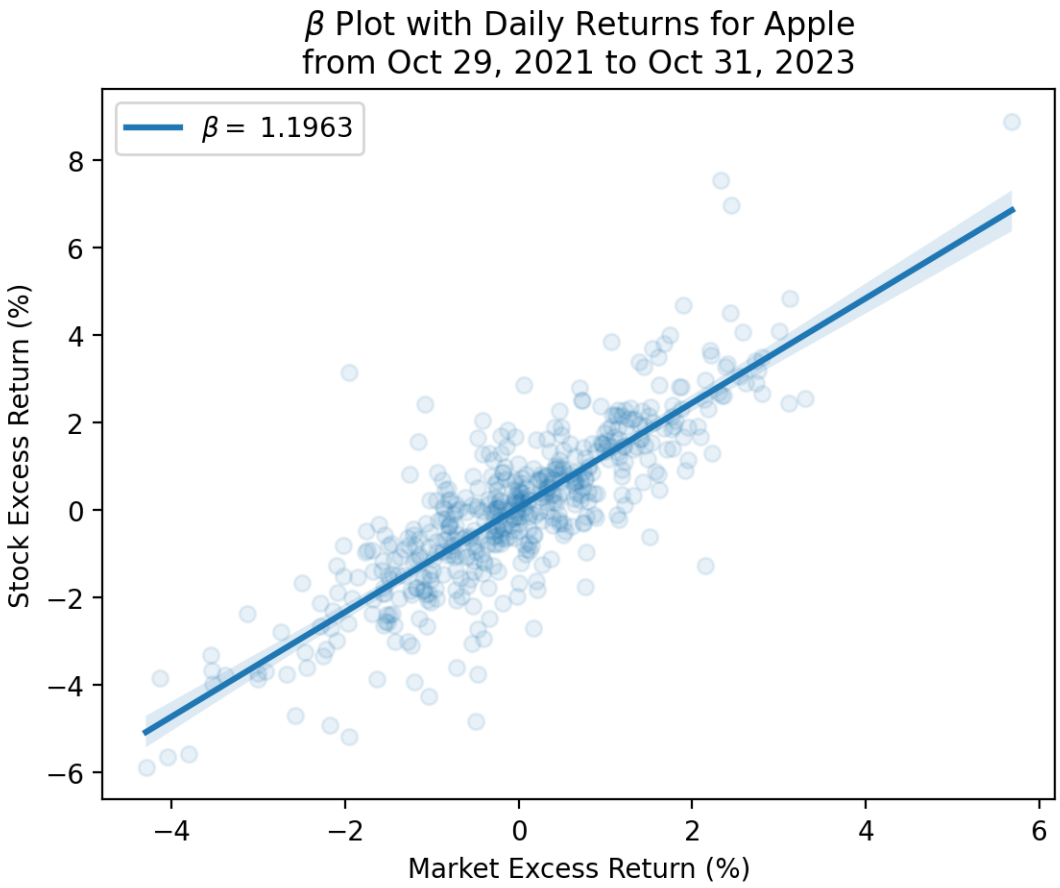
```
def label_dates(x):
    y = x.dropna()
    return f'from {y.index[0]:%b %d, %Y} to {y.index[-1]:%b %d, %Y}'
```

```
_ = aapl[['MktRF', 'RiRF']].dropna()

sns.regplot(
    x='MktRF',
    y='RiRF',
    data=_,
    scatter_kws={'alpha': 0.1},
    line_kws={'label': _.pipe(label_beta)}
)
plt.legend()
plt.xlabel('Market Excess Return (%)')
plt.ylabel('Stock Excess Return (%)')
plt.title(r'$\beta$ Plot with Daily Returns for Apple' + '\n' + _.pipe(label_dates))
plt.show()
```



We see a strong relation between Apple and market (excess) returns, but there is a lot of unexplained variation in Apple (excess) returns. The best practice is to estimate β with one to three years of daily data.

```
_ = aapl[['MktRF', 'RiRF']].dropna().iloc[-504:]

sns.regplot(
    x='MktRF',
    y='RiRF',
    data=_,
    scatter_kws={'alpha': 0.1},
    line_kws={'label': _.pipe(label_beta)}
)
plt.legend()
plt.xlabel('Market Excess Return (%)')
plt.ylabel('Stock Excess Return (%)')
plt.title(r'$\beta$ Plot with Daily Returns for Apple' + '\n' + _.pipe(label_dates))
plt.show()
```



22.1.3 The Security Market Line (SML)

The SML is a visualization of the CAPM. We can think of $E(R_i) = R_F + \beta_i[E(R_M) - R_F]$ as $y = b + mx$, where:

1. The equity premium $E(R_M) - R_F$ is the slope m of the SML, and
2. The risk-free rate of return R_F is its intercept b.

We will explore the SML more in the practice notebook.

22.1.4 How well does the CAPM work?

The CAPM *appears* to work well as a single-period model. We can see this with portfolios formed on β from Ken French.

```python
ff_beta = pdr.DataReader(
    name='Portfolios_Formed_on_BETA',
    data_source='famafrench',
    start='1900'
)
```

```
C:\Users\r.herron\AppData\Local\Temp\ipykernel_9472\1792222927.py:1: FutureWarning: The argu
  ff_beta = pdr.DataReader(
C:\Users\r.herron\AppData\Local\Temp\ipykernel_9472\1792222927.py:1: FutureWarning: The argu
  ff_beta = pdr.DataReader(
C:\Users\r.herron\AppData\Local\Temp\ipykernel_9472\1792222927.py:1: FutureWarning: The argu
  ff_beta = pdr.DataReader(
C:\Users\r.herron\AppData\Local\Temp\ipykernel_9472\1792222927.py:1: FutureWarning: The argu
  ff_beta = pdr.DataReader(
C:\Users\r.herron\AppData\Local\Temp\ipykernel_9472\1792222927.py:1: FutureWarning: The argu
  ff_beta = pdr.DataReader(
C:\Users\r.herron\AppData\Local\Temp\ipykernel_9472\1792222927.py:1: FutureWarning: The argu
  ff_beta = pdr.DataReader(
C:\Users\r.herron\AppData\Local\Temp\ipykernel_9472\1792222927.py:1: FutureWarning: The argu
  ff_beta = pdr.DataReader(
```

```python
print(ff_beta['DESCR'])
```

```
Portfolios Formed on BETA
-------------------------

This file was created by CMPT_BETA_RETS using the 202310 CRSP database. It contains value- a

  0 : Value Weighted Returns -- Monthly (724 rows x 15 cols)
  1 : Equal Weighted Returns -- Monthly (724 rows x 15 cols)
  2 : Value Weighted Returns -- Annual from January to December (59 rows x 15 cols)
  3 : Equal Weighted Returns -- Annual from January to December (59 rows x 15 cols)
  4 : Number of Firms in Portfolios (724 rows x 15 cols)
  5 : Average Firm Size (724 rows x 15 cols)
  6 : Value-Weighted Average of Prior Beta (61 rows x 15 cols)
```

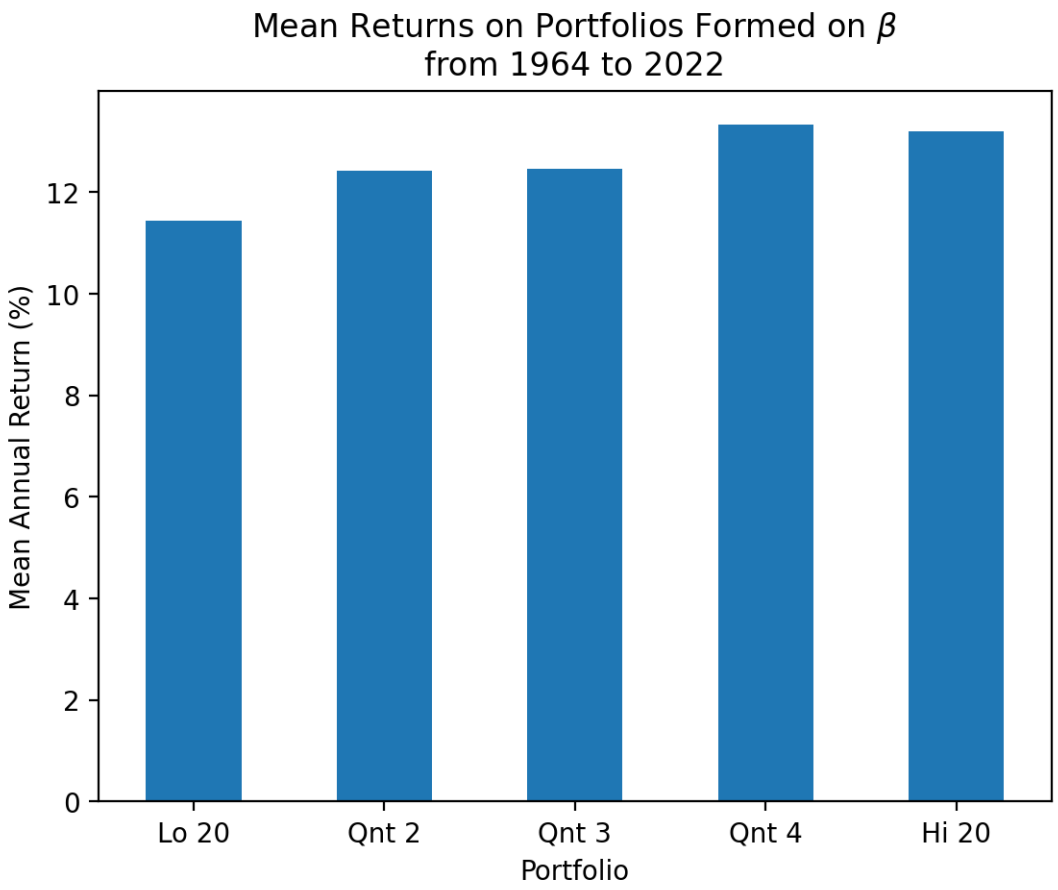
This file contains seven data frames. We want the data frame at [2], which contains the annual returns on two sets of portfolios formed on the previous year's βs.

```
ff_beta[2].head()
```

Date	Lo 20	Qnt 2	Qnt 3	Qnt 4	Hi 20	Lo 10	Dec 2	Dec 3	Dec 4	Dec 5	D
1964	16.8700	19.7500	17.6600	8.7300	12.5500	24.7000	13.5900	20.2700	19.0100	19.7900	1
1965	8.7200	6.9800	15.4000	24.8800	49.6700	12.4400	6.7600	10.0200	5.2500	13.7100	1
1966	-9.2500	-12.1700	-9.1000	-2.7300	-2.2000	-9.4300	-9.4400	-12.5700	-12.1600	-7.5000	-1
1967	13.4300	22.0600	31.9500	42.7900	51.1500	8.9200	18.6400	22.7900	21.6500	31.1100	3
1968	15.8100	9.1200	13.6400	14.4300	24.2400	18.4800	12.8400	14.5100	5.8400	12.6600	1

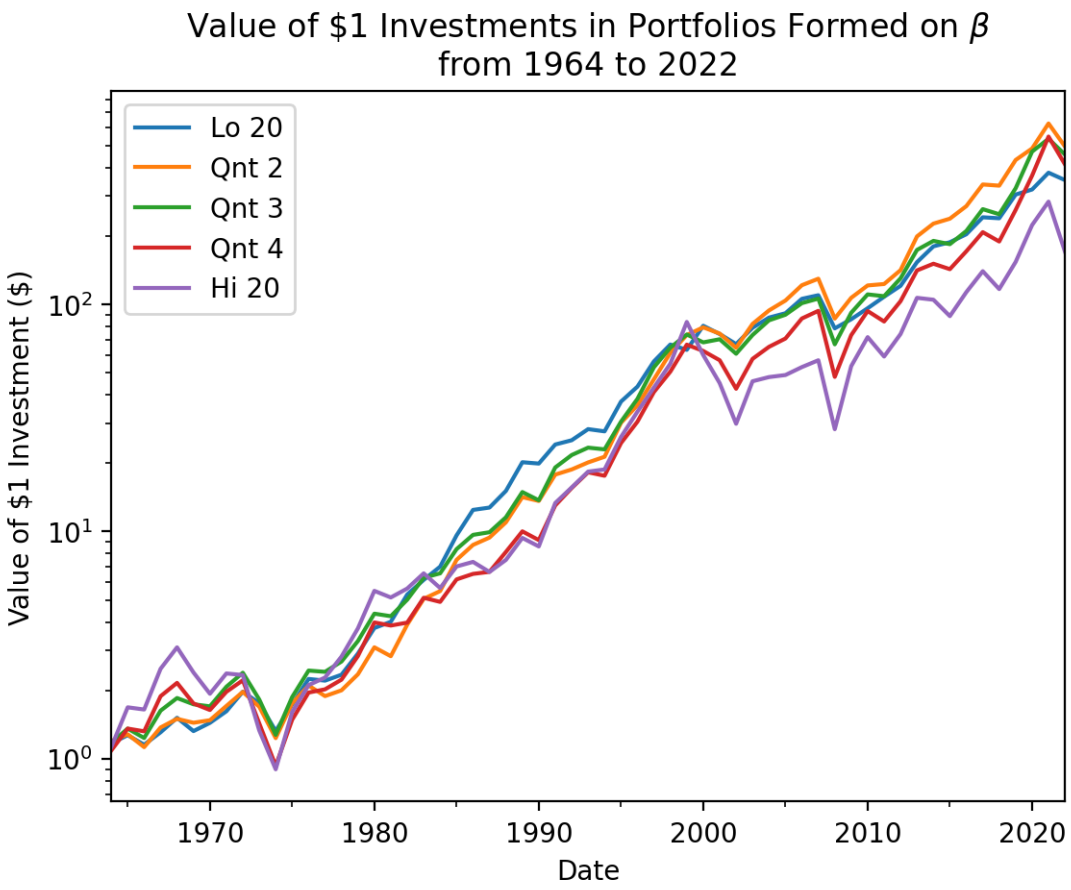
We can plot the mean annual return on each of the five portfolios in the first set of portflios.
We do not need to annualize these numbers because they are the means of annual returns.

```
_ = ff_beta[2].iloc[:, :5]
_.mean().plot(kind='bar')
plt.ylabel('Mean Annual Return (%)')
plt.xlabel('Portfolio')
plt.xticks(rotation=0)
plt.title(r'Mean Returns on Portfolios Formed on $\beta$' + '\n' + f'from {_.index[0]} to
plt.show()
```



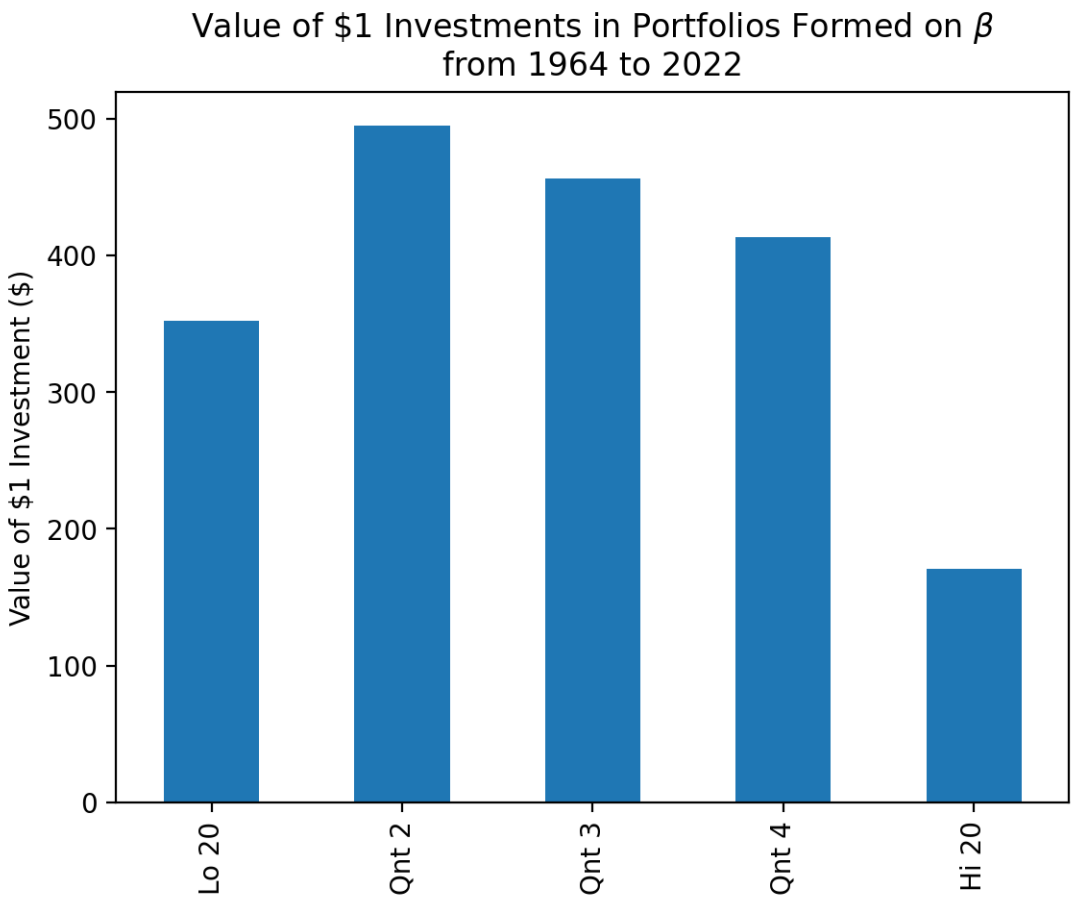
We can think of the plot above as a binned plot of the SML. The x axis above is an ordinal measure of β, and the y axis above is the mean return. Recall the slope of the SML is the market risk premium. If the market risk premium is too low, then high β stocks do not have high enough returns. We can see this failure of the CAPM by plotting long-term or cumulative returns on these five portfolios.

```
_ = ff_beta[2].iloc[:, :5]
_.div(100).add(1).cumprod().plot()
plt.semilogy()
plt.ylabel('Value of \$1 Investment (\$)')
plt.title(r'Value of \$1 Investments in Portfolios Formed on $\beta$' + '\n' + f'from {_.i
plt.show()
```



In the plot above, the highest-β portfolio has the lowest cumulative returns! The log scale masks a lot of variation, too!

```
_ = ff_beta[2].iloc[:, :5]
_.div(100).add(1).prod().plot(kind='bar')
plt.ylabel('Value of \$1 Investment (\$)')
plt.title(r'Value of \$1 Investments in Portfolios Formed on $\beta$' + '\n' + f'from {_.i
plt.show()
```



If the CAPM does not work well, especially over the horizons we use it for (e.g., capital budgeting), why do we continue to learn it?

1. The CAPM works well *across* asset classes. We will explore this more in the practice notebook.
2. The CAPM intuition that diversification matters is correct and important
3. The CAPM assigns high costs of capital to high-β projects (i.e., high-risk projects), which is a hidden benefit
4. In practice, everyone uses the CAPM

Ivo Welch provides a more complete discussion in section 10.5 of chapter 10 of this his free *Corporate Finance* textbook.

22.2 Multifactor Models

Another shortcoming of the CAPM is that it fails to explain the returns on portfolios formed on size (market capitalization) and value (book-to-market equity ratio), which we will explore in the practice notebook. These shortcomings have led to an explosion in multifactor models, which we will explore here.

22.2.1 The Fama-French Three-Factor Model

Fama and French (1993) expand the CAPM by adding two additional factors to explain the excess returns on size and value. The size factor, SMB or small-minus-big, is a diversified portfolio that measures the excess returns of small market cap. stocks over large market cap. stocks. The value factor, HML of high-minus-low, is a diversified portfolio that measures the excess returns of high book-to-market stocks over low high book-to-market stocks. We typically call this model the "Fama-French three-factor model" and express it as: $E(R_i) = R_F + \alpha + \beta_M[E(R_M) - R_M)] + \beta_{SMB}SMB + \beta_{HML}HML$. There are two common uses for the three-factor model:

1. Use the coefficient estimate on the intercept (i.e., α, often called "Jensen's α") as a risk-adjusted performance measure. If α is positive and statistically significant, we may attribute fund performance to manager skill.
2. Use the remaining coefficient estimates to evaluate how the fund manager generates returns. If the regression R^2 is high, we may replace the fund manager with the factor itself.

We can use the Fama-French three-factor model to evaluate Warren Buffett at Berkshire Hathaway (BRK-A). We will focus on the first three-years of easily available returns because Buffett had a larger edge when BRK was much smaller.

```
brk = (
    yf.download(tickers='BRK-A')
    .assign(Ri=lambda x: x['Adj Close'].pct_change().mul(100))
    .join(ff[0])
    .assign(RiRF = lambda x: x['Ri'] - x['RF'])
    .rename(columns={'Mkt-RF': 'MktRF'})
    .rename_axis(columns='Variable')
)
```

```
[**********************100%%**********************]  1 of 1 completed
```

```
model = smf.ols(formula='RiRF ~ MktRF + SMB + HML', data=brk.iloc[:756])
fit = model.fit()
summary = fit.summary()
summary
```

Dep. Variable:	RiRF	**R-squared:**	0.053
Model:	OLS	**Adj. R-squared:**	0.049
Method:	Least Squares	**F-statistic:**	13.91
Date:	Fri, 22 Dec 2023	**Prob (F-statistic):**	7.81e-09
Time:	09:59:35	**Log-Likelihood:**	-1208.9
No. Observations:	755	**AIC:**	2426.
Df Residuals:	751	**BIC:**	2444.
Df Model:	3		
Covariance Type:	nonrobust		

	coef	**std err**	**t**	**P> |t|**	**[0.025**	**0.975]**
Intercept	0.0801	0.044	1.809	0.071	-0.007	0.167
MktRF	0.3484	0.075	4.643	0.000	0.201	0.496
SMB	0.4021	0.093	4.302	0.000	0.219	0.586
HML	0.0907	0.125	0.724	0.469	-0.155	0.336

Omnibus:	118.864	**Durbin-Watson:**	1.797
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	1308.034
Skew:	0.284	**Prob(JB):**	9.20e-285
Kurtosis:	9.423	**Cond. No.**	3.51

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

The α above seems small, but this is a *daily* value. We can multiple α by 252 to annualize it.

```
print(f"Buffet's annualized alpha in the early 1980s: {fit.params['Intercept'] * 252:0.4f}
```

```
Buffet's annualized alpha in the early 1980s: 20.1836
```

We will explore rolling αs and βs in the practice notebook using `RollingOLS()` from `statsmodels.regression.rolling`.

22.2.2 The Four-Factor and Five-Factor Models

There are literally hundreds of published factors! However, many of them have little explanatory power, in or out of sample. Two more factor models that have explanatory power, economic intuition, and widespread adoption are the four-factor model and five-factor model.

The four-factor model adds a momentum factor to the Fama-French three-factor model. The momentum factor is a diversified portfolio that measures the excess returns of winner stocks over the loser stocks over the past 12 months. The momentum factor is often called UMD for up-minus-down or WML for winners-minus-losers. French stores the momentum factor in a different file because Fama and French are skeptical of momentum as a foundational risk factor.

The five-factor model adds profitability and investment policy factors. The profitability factor, RMW or robust-minus-weak, measures the excess returns of stocks with high profits over those with low profits. The investment policy factor, CMA or conservative-minus-aggressive, measures the excess returns of stocks with low corporate investment (conservative) over those with high corporate investment (aggressive).

We will explore the four-factor and five-factor models in the practice notebook.

23 Herron Topic 3 - Practice for Section 04

23.1 Announcements

23.2 Practice

23.2.1 Plot the security market line (SML) for a variety of asset classes

Use the past three years of daily data for the following exhange traded funds (ETFs):

1. SPY (SPDR—Standard and Poor's Depository Receipts—ETF for the S&P 500 index)
2. BIL (SPDR ETF for 1-3 month Treasury bills)
3. GLD (SPDR ETF for gold)
4. JNK (SPDR ETF for high-yield debt)
5. MDY (SPDR ETF for S&P 400 mid-cap index)
6. SLY (SPDR ETF for S&P 600 small-cap index)
7. SPBO (SPDR ETF for corporate bonds)
8. SPMB (SPDR ETF for mortgage-backed securities)
9. SPTL (SPDR ETF for long-term Treasury bonds)

23.2.2 Plot the SML for the Dow Jones Industrial Average (DJIA) stocks

Use the past three years of daily returns data for the stocks listed on the DJIA Wikipedia page. Compare the DJIA SML to the asset class SML above.

23.2.3 Plot the SML for the five portfolios formed on beta

Download data for portfolios formed on β (`Portfolios_Formed_on_BETA`) from Ken French. For the value-weighted portfolios, plot realized returns versus β. These data should elements `[2]` and `[6]`, respectively.

23.2.4 Estimate the CAPM βs on several levered and inverse exchange traded funds (ETFs)

Try the following ETFs:

1. SPY
2. UPRO
3. SPXU

Can you determine what these products do from the data alone? Estimate βs and plot cumulative returns. You may want to pick short periods of time with large market swings.

23.2.5 Explore the size factor

23.2.5.1 Estimate αs for the ten portfolios formed on size

Academics started researching size-based portfolios in the early 1980s, so you may want to focus on the pre-1980 sample.

23.2.5.2 Are the returns on these ten portfolios formed on size concentrated in a specific month?

23.2.5.3 Compare the size factor to the market factor

You may want to consider mean excess returns by decade.

23.2.6 Repeat the exercises above with the value factor

23.2.7 Repeat the exercises above with the momentum factor

You may find it helpful to consider the worst months and years for the momentum factor.

23.2.8 Plot the coefficient estimates from a rolling Fama-French three-factor model for Berkshire Hathaway

Use a three-year window with daily returns. How has Buffett's α and βs changed over the past four decades?

23.2.9 Use the three-, four-, and five-factor models to determine how the ARKK Innovation ETF generates returns

Part XIII

Week 13

24 Herron Topic 4 - Portfolio Optimization

This notebook covers portfolio optimization. I have not found a perfect reference that combines portfolio optimization and Python, but here are two references that I find useful:

1. Ivo Welch discusses the mathematics and finance of portfolio optimization in Chapter 12 of his draft textbook on investments.
2. Eryk Lewinson provides Python code for portfolio optimization in chapter 7 of his *Python for Finance Cookbook*, but he uses several packages that are either non-free or abandoned.

In this notebook, we will:

1. Review the $\frac{1}{n}$ portfolio (or equal-weighted portfolio) from Herron Topic 1
2. Use SciPy's `minimize()` function to:
    1. Find the minimum variance portfolio
    2. Find the (mean-variance) efficient frontier

In the practice notebook, we will use SciPy's `minimize()` function to achieve any objective.

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
```

```
%precision 4
pd.options.display.float_format = '{:.4f}'.format
%config InlineBackend.figure_format = 'retina'
```

```
import yfinance as yf
import pandas_datareader as pdr
```

24.1 The $\frac{1}{n}$ Portfolio

We first saw the $\frac{1}{n}$ portfolio (or equal-weighted portfolio) in Herron Topic 1. In the $\frac{1}{n}$ portfolio, each of n assets receives an equal portfolio weight $w_i = \frac{1}{n}$. While the $\frac{1}{n}$ strategy seems too

simple to be useful, DeMiguel, Garlappi, and Uppal (2007) show that it is difficult to beat $\frac{1}{n}$ strategy, even with more advanced strategies.

```
tickers = 'MSFT AAPL TSLA AMZN NVDA GOOG'

matana = (
    yf.download(tickers=tickers)
    .rename_axis(columns=['Variable', 'Ticker'])
)

matana.tail()
```

```
[**********************100%%**********************]  6 of 6 completed
```

Variable	Adj Close						Close		
Ticker	AAPL	AMZN	GOOG	MSFT	NVDA	TSLA	AAPL	AMZN	GOOG
Date									
2023-12-18	195.8900	154.0700	137.1900	372.6500	500.7700	252.0800	195.8900	154.0700	137.1900
2023-12-19	196.9400	153.7900	138.1000	373.2600	496.0400	257.2200	196.9400	153.7900	138.1000
2023-12-20	194.8300	152.1200	139.6600	370.6200	481.1100	247.1400	194.8300	152.1200	139.6600
2023-12-21	194.6800	153.8400	141.8000	373.5400	489.9000	254.5000	194.6800	153.8400	141.8000
2023-12-22	195.1250	153.8300	142.6201	373.8800	NaN	256.5600	195.1250	153.8300	142.6201

```
returns = matana['Adj Close'].pct_change().iloc[(-3 * 252):]

returns.describe()
```

```
C:\Users\r.herron\AppData\Local\Temp\ipykernel_5180\2146458459.py:1: FutureWarning: The defau
  returns = matana['Adj Close'].pct_change().iloc[(-3 * 252):]
```

Ticker	AAPL	AMZN	GOOG	MSFT	NVDA	TSLA
count	756.0000	756.0000	756.0000	756.0000	756.0000	756.0000
mean	0.0007	0.0002	0.0009	0.0009	0.0023	0.0009
std	0.0176	0.0236	0.0199	0.0175	0.0333	0.0370
min	-0.0587	-0.1405	-0.0963	-0.0772	-0.0947	-0.1224
25%	-0.0089	-0.0128	-0.0098	-0.0089	-0.0171	-0.0194
50%	0.0009	0.0003	0.0009	0.0007	0.0024	0.0018

Ticker	AAPL	AMZN	GOOG	MSFT	NVDA	TSLA
75%	0.0115	0.0127	0.0110	0.0112	0.0208	0.0205
max	0.0890	0.1354	0.0775	0.0823	0.2437	0.1964

Before we revisit the advanced techniques from Herron Topic 1, we can calculate $\frac{1}{n}$ portfolio returns manually, where $R_P = \frac{\sum_i^n R_i}{n}$ Since our weights are constant (i.e., do not change over time), we rebalance our portfolio every return period. If we have daily data, rebalance daily. If we have monthly data, we rebalance monthly, and so on.

```
n = returns.shape[1]
p1 = returns.sum(axis=1).div(n)

p1.describe()
```

```
count   756.0000
mean      0.0010
std       0.0200
min      -0.0632
25%      -0.0113
50%       0.0016
75%       0.0124
max       0.0980
dtype: float64
```

Recall from Herron Topic 1 we have two better options:

1. The `.mean(axis=1)` method for the $\frac{1}{n}$ portfolio
2. The `.dot(weights)` method where `weights` is a pandas series or NumPy array of portfolio weights, allowing different weights for each asset

```
p2 = returns.mean(axis=1)

p2.describe()
```

```
count   756.0000
mean      0.0010
std       0.0200
min      -0.0632
25%      -0.0113
```

```
50%       0.0016
75%       0.0124
max       0.0980
dtype: float64
```

```
weights = np.ones(n) / n
p3 = returns.dot(weights)

p3.describe()
```

```
count   756.0000
mean      0.0010
std       0.0200
min      -0.0632
25%      -0.0113
50%       0.0016
75%       0.0124
max       0.0980
dtype: float64
```

The `.describe()` method provides summary statistics for data, letting us make quick comparisons. However, we should use `np.allclose()` if we want to be sure that `p1`, `p2`, and `p3` are similar.

```
np.allclose(p1, p2)
```

```
True
```

```
np.allclose(p1, p3)
```

```
True
```

Here is a simple example to help understand the `.dot()` method.

```
silly_n = 3
silly_R = pd.DataFrame(np.arange(2*silly_n).reshape(2, silly_n))
silly_w = np.ones(3) / 3
```

```
print(
    f'silly_n:\n{silly_n}',
    f'silly_R:\n{silly_R}',
    f'silly_w:\n{silly_w}',
    sep='\n\n'
)
```

```
silly_n:
3

silly_R:
   0  1  2
0  0  1  2
1  3  4  5

silly_w:
[0.3333 0.3333 0.3333]
```

```
silly_R.dot(silly_w)
```

```
0   1.0000
1   4.0000
dtype: float64
```

Under the hood, Python and the `.dot()` method (effectively) do the following calculation:

```
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
```

24.2 SciPy's `minimize()` Function

24.2.1 A Crash Course in SciPy's `minimize()` Function

The `minimize()` function from SciPy's `optimize` module finds the input array `x` that minimizes the output of the function `fun`. The `minimize()` function uses optimization techniques that are outside this course, but you can consider these optimization techniques to be sophisticated trial and error.

Here are the most common arguments we will pass to the `minimize()` function:

1. We pass our first guess for input array `x` to argument `x0=`.
2. We pass additional arguments for function `fun` as a tuple to argument `args=`.
3. We pass lower and upper bounds on `x` as a tuple of tuples to argument `bounds=`.
4. We constrain our results with a tuple of dictionaries of functions to argument `contraints=`.

Here is a simple example that minimizes the function `quadratic()` that accepts arguments `x` and `a` and returns $y = (x - a)^2$.

```
import scipy.optimize as sco
```

```
def quadratic(x, a=5):
    return (x - a) ** 2
```

```
quadratic(x=5, a=5)
```

```
0
```

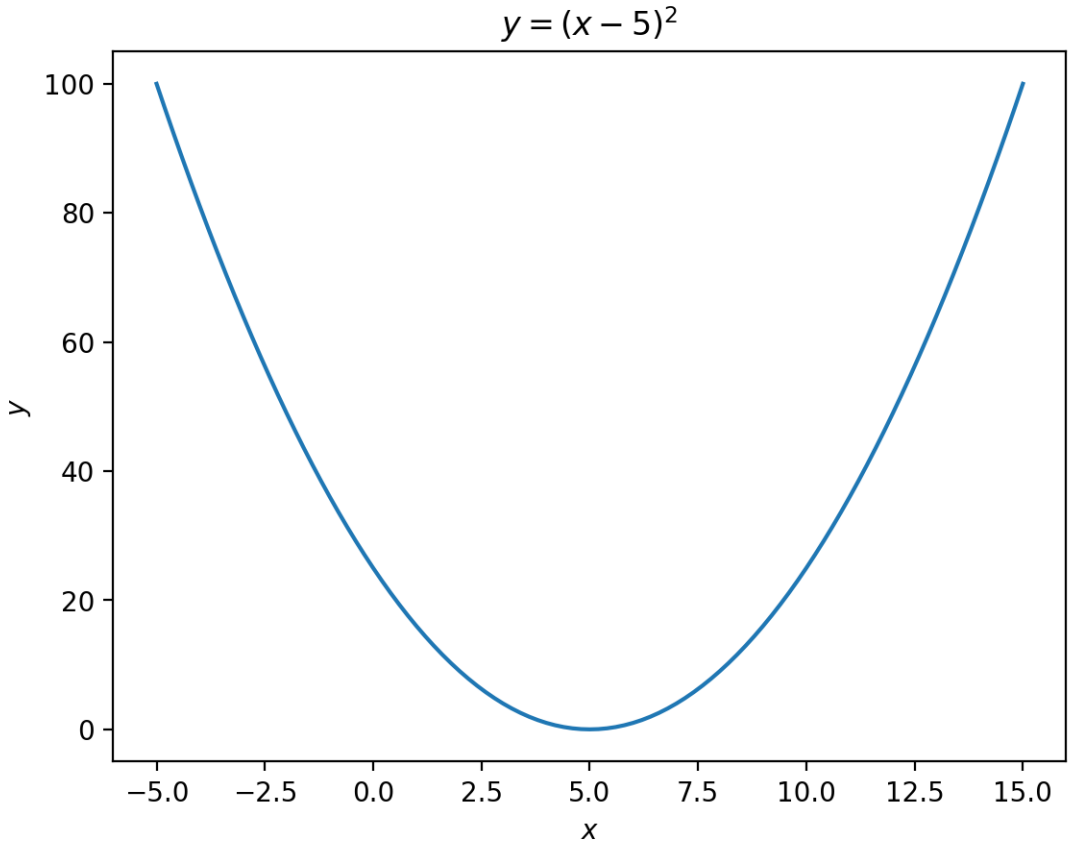
```
quadratic(x=10, a=5)
```

```
25
```

It is helpful to plot $y = (x - a)$ first.

```
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'$y = (x - 5)^2$')
plt.show()
```



The minimum output of `quadratic()` occurs at $x = 5$ if we do not use bounds or constraints, even if we 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 `x` between 6 and 10 (i.e., $6 \leq x \leq 10$).

```
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()` occurs at $x = 6$, again, if we constrain `x - 6` to be non-negative. We use bounds to limit the search space directly, and we use constraints to limit the search space indirectly based on a formula.

```
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 can use the `args=` argument to pass additional arguments to `fun`. For example, we change the `a=` argument in `quadratic()` from the default of `a=5` to `a=20` with `args=(20,)`. Note that `args=` expects a tuple, so we need a trailing comma `,` if we have 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
```

24.2.2 The Minimum Variance Portfolio

We can find the minimum variance portfolio with `minimize()` function from SciPy's `optimize` module. The `minimize()` function with vary an input array `x` (starting from argument `x0=`) to minimize the objective function `fun=` subject to the bounds and constraints in `bounds=` and `constraints=`. We will define a function `port_vol()` to calculate portfolio volatility. The first argument to `port_vol()` must be the input array `x` that the `minimize()` function searches over. For clarity, we will call this first argument `x`, but the argument's name is not important.

```
def port_vol(x, r, ppy):
    return np.sqrt(ppy) * r.dot(x).std()
```

We will eventually need a mean portfolio return function, too.

```
def port_mean(x, r, ppy):
    return ppy * r.dot(x).mean()
```

```
res_mv = sco.minimize(
    fun=port_vol, # objective function that we minimize
    x0=np.ones(returns.shape[1]) / returns.shape[1], # initial portfolio weights
    args=(returns, 252), # additional arguments to our objective function
    bounds=[(0,1) for _ in returns], # bounds limit the search space for each portfolio we
    constraints=(
        {'type': 'eq', 'fun': lambda x: x.sum() - 1} # minimize drives "eq" constraints to
    )
)

print(res_mv)
```

```
message: Optimization terminated successfully
success: True
 status: 0
    fun: 0.2571111710752664
      x: [ 4.478e-01  5.399e-17  1.281e-01  4.241e-01  2.060e-17
           1.323e-17]
    nit: 10
    jac: [ 2.571e-01  2.645e-01  2.571e-01  2.571e-01  3.738e-01
           3.130e-01]
   nfev: 70
   njev: 10
```

What are the attributes of this minimum variance portfolio?

```
def print_port_res(w, r, title, ppy=252, tgt=None):
    width = len(title)
    rp = r.dot(w)
    mu = ppy * rp.mean()
    sigma = np.sqrt(ppy) * rp.std()
    if tgt is not None:
```

```
        er = rp.sub(tgt)
        sr = np.sqrt(ppy) * er.mean() / er.std()
    else:
        sr = None

    return print(
        title,
        '=' * width,
        '',
        'Performance',
        '-' * width,
        'Return:'.ljust(width - 6) + f'{mu:0.4f}',
        'Volatility:'.ljust(width - 6) + f'{sigma:0.4f}',
        'Sharpe Ratio:'.ljust(width - 6) + f'{sr:0.4f}\n' if sr is not None else '',
        'Weights',
        '-' * width,
        '\n'.join([f'{_r}:'.ljust(width - 6) + f'{_w:0.4f}' for _r, _w in zip(r.columns, w
        sep='\n',
    )
```

```
print_port_res(w=res_mv['x'], r=returns, title='Minimum Variance Portfolio')
```

```
Minimum Variance Portfolio
==========================

Performance
--------------------------
Return:             0.2037
Volatility:         0.2571

Weights
--------------------------
AAPL:               0.4478
AMZN:               0.0000
GOOG:               0.1281
MSFT:               0.4241
NVDA:               0.0000
TSLA:               0.0000
```

24.2.3 The Mean-Variance Efficient Frontier

We will use the `minimize()` function to map the efficient frontier. Here is a basic outline:

1. Create a NumPy array `tret` of target returns
2. Create an empty list `res_ef` of `minimize()` results
3. Loop over `tret`, passing each as a constraint to the `minimize()` function
4. Append each `minimize()` result to `res_ef`

```
tret = 252 * np.linspace(returns.mean().min(), returns.mean().max(), 25)

tret
```

```
array([0.0561, 0.0776, 0.0991, 0.1206, 0.1421, 0.1637, 0.1852, 0.2067,
       0.2282, 0.2497, 0.2713, 0.2928, 0.3143, 0.3358, 0.3574, 0.3789,
       0.4004, 0.4219, 0.4434, 0.465 , 0.4865, 0.508 , 0.5295, 0.551 ,
       0.5726])
```

We will loop over these target returns, finding the minimum variance portfolio for each target return.

```
res_ef = []

for t in tret:
    _ = sco.minimize(
        fun=port_vol, # minimize portfolio volatility
        x0=np.ones(returns.shape[1]) / returns.shape[1], # initial portfolio weights
        args=(returns, 252), # additional arguments to fun, in order
        bounds=[(0, 1) for c in returns.columns], # bounds limit the search space for each
        constraints=(
            {'type': 'eq', 'fun': lambda x: x.sum() - 1}, # constrain sum of weights to on
            {'type': 'eq', 'fun': lambda x: port_mean(x=x, r=returns, ppy=252) - t} # cons

        )
    )
    res_ef.append(_)
```

List `res_ef` contains the results of all 25 minimum-variance portfolios. For example, `res_ef[0]` is the minimum variance portfolio for the lowest target return.

```
res_ef[0]
```

```
 message: Optimization terminated successfully
 success: True
  status: 0
     fun: 0.37384794293656093
       x: [ 1.031e-09  1.000e+00  6.384e-16  0.000e+00  0.000e+00
            8.327e-17]
     nit: 2
     jac: [ 1.697e-01  3.738e-01  2.114e-01  1.859e-01  3.090e-01
            2.698e-01]
    nfev: 14
    njev: 2
```

I typically check that all portfolio volatility minimization succeeds. If a portfolio volatility minimization fails, we should check our function, bounds, and constraints.

```
for r in res_ef:
    assert r['success']
```

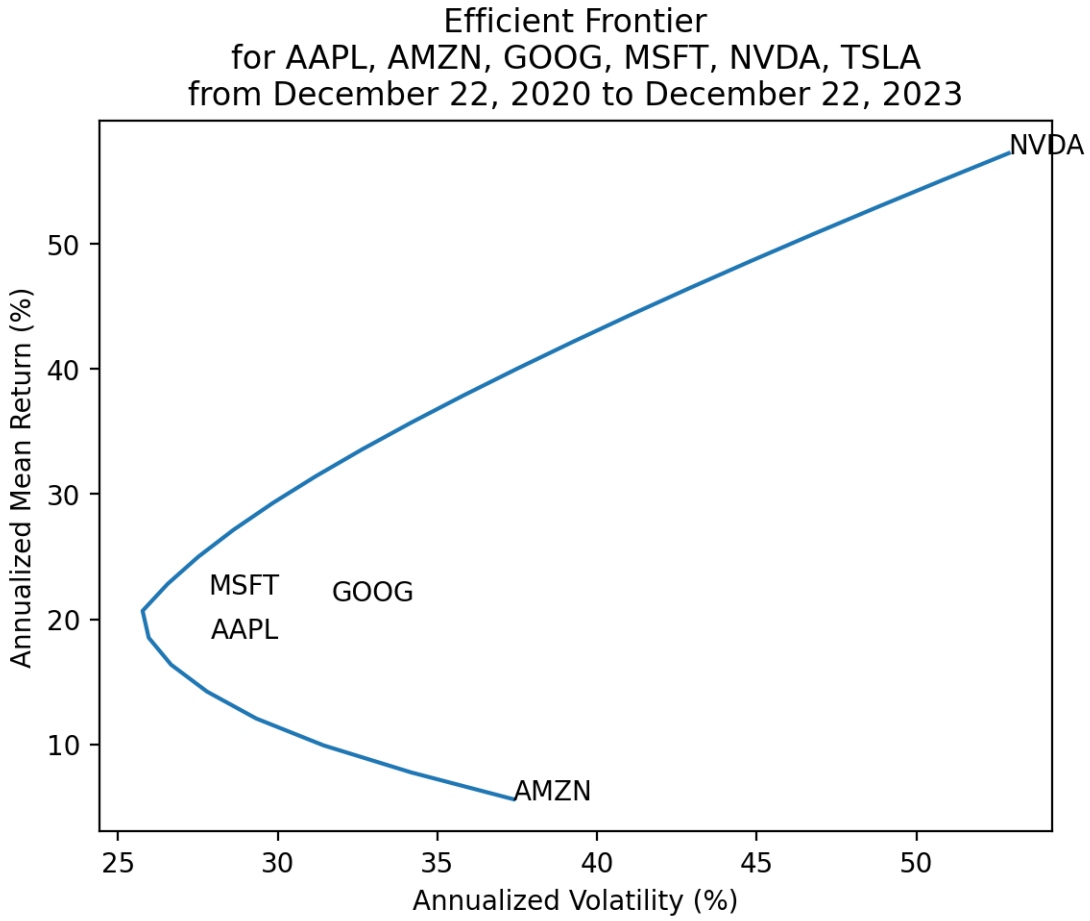
We can combine the target returns and volatilities into a data frame `ef`.

```
ef = pd.DataFrame(
    {
        'tret': tret,
        'tvol': np.array([r['fun'] if r['success'] else np.nan for r in res_ef])
    }
)

ef.head()
```

	tret	tvol
0	0.0561	0.3738
1	0.0776	0.3417
2	0.0991	0.3144
3	0.1206	0.2933
4	0.1421	0.2778

```
ef.mul(100).plot(x='tvol', y='tret', legend=False)
plt.ylabel('Annualized Mean Return (%)')
plt.xlabel('Annualized Volatility (%)')
plt.title(
    f'Efficient Frontier' +
```

```
    f'\nfor {", ".join(returns.columns)}' +
    f'\nfrom {returns.index[0]:%B %d, %Y} to {returns.index[-1]:%B %d, %Y}'
)

for t, x, y in zip(
    returns.columns,
    returns.std().mul(100*np.sqrt(252)),
    returns.mean().mul(100*252)
):
    plt.annotate(text=t, xy=(x, y))

plt.show()
```



25 Herron Topic 4 - Practice for Section 04

25.1 Announcements

25.2 Practice

25.2.1 Find the maximum Sharpe Ratio portfolio of MATANA stocks over the last three years

Note that `sco.minimize()` finds *minimums*, so you need to minimize the *negative* Sharpe Ratio.

25.2.2 Find the maximum Sharpe Ratio portfolio of MATANA stocks over the last three years, but allow short weights up to 10% on each stock

25.2.3 Find the maximum Sharpe Ratio portfolio of MATANA stocks over the last three years, but allow total short weights of up to 30%

25.2.4 Find the maximum Sharpe Ratio portfolio of MATANA stocks over the last three years, but do not allow any weight to exceed 30% in magnitude

25.2.5 Find the minimum 95% Value at Risk (Var) portfolio of MATANA stocks over the last three years

More on VaR here.

25.2.6 Find the minimum maximum draw down portfolio of MATANA stocks over the last three years

25.2.7 Find the minimum maximum draw down portfolio with all available data for the current Dow-Jones Industrial Average (DJIA) stocks

You can find the DJIA tickers on Wikipedia.

25.2.8 Plot the (mean-variance) efficient frontier with all available data for the current the DJIA stocks

25.2.9 Find the maximum Sharpe Ratio portfolio with all available data for the current the DJIA stocks

25.2.10 Compare the $\frac{1}{n}$ and maximum Sharpe Ratio portfolios with all available data for the current DJIA stocks

Use all but the last 252 trading days to estimate the maximum Sharpe Ratio portfolio weights. Then use the last 252 trading days of data to compare the $\frac{1}{n}$ maximum Sharpe Ratio portfolios.

Part XIV

Week 14

26 Herron Topic 5 - Simulations

In finance, we typically perform simulations with Monte Carlo methods. Monte Carlo methods are used throughout science, engineering, and mathematics, and they are especially useful in finance due to the randomness of asset prices. This lecture notebook provides an introduction to Monte Carlo methods in finance.

The basic idea behind Monte Carlo methods in finance is to create many possible paths of financial variables (e.g., stock prices, interest rates, exchange rates) based on their historical behavior, statistical properties, and any other relevant factors. We use these paths to simulate the performance of financial instruments or portfolios, and then we aggregate these results to estimate metrics of interest. This approach helps us model and analyze complex financial problems that may be difficult or impossible to solve analytically.

We could spend a semester (or more) on simulations and Monte Carlo methods. In this lecture notebook, we will limit our focus to:

1. Option pricing: Monte Carlo methods can be used to estimate the fair value of financial derivatives, such as options and warrants. By simulating the potential future paths of the underlying asset and calculating the payoffs of the derivative at each path, the expected payoff can be computed and discounted to present value to determine the option's price.
2. Value at Risk (VaR): Monte Carlo simulations can be used to compute VaR, a widely used risk management metric that estimates the potential loss in the value of a portfolio over a specific time horizon, given a certain level of confidence (e.g., 95% or 99%). By simulating numerous scenarios and observing the distribution of potential losses, the VaR can be calculated as the threshold below which a certain percentage of losses fall.

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
```

```
%precision 2
pd.options.display.float_format = '{:.2f}'.format
%config InlineBackend.figure_format = 'retina'
```

```
import yfinance as yf
import pandas_datareader as pdr
```

26.1 Option Pricing

We can use Monte Carlo methods to value stock options. First, we simulate several hundred or thousand possible (but random) price paths for the underlying stock. Then, we calculate the payoff for each path. On some paths, the option will expire "in the money" with $S_T > K$ and pay $S_T - K$. On 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 these payoffs is the option price. This is an illustrative example, and there is a lot of depth to Monte Carlo methods for option pricing.

26.1.1 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-dependent volatility. So GBM is often used for stocks and not for bond prices, which tend to display long-term reversion to the face value. In the GBM SDE:

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 apply this closed form solution recursively: $S(t_{i+1}) = S(t_i) \exp\left(\left(\mu - \frac{1}{2}\sigma^2\right)(t_{i+1} - t_i) + \sigma\sqrt{t_{i+1} - t_i} Z_{i+1}\right)$. Here, Z_i is a Standard Normal random variable (dW_t are independent and normally distributed) and $i = 0, \ldots, T-1$ is the time index.

We can use this closed form solution to simulate stock prices for AAPL.

```
aapl = (
    yf.download(tickers='AAPL')
    .assign(Return=lambda x: x['Adj Close'].pct_change())
    .rename_axis(columns=['Variable'])
)
```

```
[**********************100%%**********************]  1 of 1 completed
```

```
aapl.describe()
```

Variable	Open	High	Low	Close	Adj Close	Volume	Return
count	10849.00	10849.00	10849.00	10849.00	10849.00	10849.00	10848.00
mean	19.99	20.21	19.78	20.01	19.26	321584833.93	0.00
std	41.82	42.28	41.38	41.85	41.46	336387347.28	0.03
min	0.05	0.05	0.05	0.05	0.04	0.00	-0.52
25%	0.29	0.30	0.29	0.29	0.24	116124600.00	-0.01
50%	0.51	0.52	0.50	0.51	0.42	209059200.00	0.00
75%	18.89	19.02	18.68	18.89	16.44	401464000.00	0.01
max	198.02	199.62	197.00	198.11	198.11	7421640800.00	0.33

We will use returns from 2021 to predict prices in 2022.

```
train = aapl.loc['2021']
test = aapl.loc['2022']
```

We will use the following function to simulate price paths. 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
```

Now we will simulate price paths. Here is one simulated price path:

```
simulate_gbm(
    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]
)
```

```
array([175.56, 177.13, 176.93, 178.94, 183.5 , 183.01, 182.53, 187.34,
       189.83, 188.62, 190.45, 189.26, 188.07, 188.99, 183.55, 178.8 ,
       177.41, 174.78, 175.83, 173.51, 169.86, 174.02, 173.59, 173.95,
       170.26, 168.98, 169.46, 166.58, 167.75, 166.34, 165.75, 164.35,
       169.41, 169.55, 166.92, 169.28, 166.22, 166.95, 162.03, 158.83,
       159.49, 161.53, 162.14, 162.02, 161.42, 157.86, 156.24, 155.27,
       158.05, 159.08, 154.87, 155.83, 155.05, 153.56, 155.22, 157.93,
       160.44, 158.5 , 157.89, 158.89, 161.53, 160.48, 160.18, 157.57,
       154.78, 156.94, 160.51, 160.5 , 163.24, 164.35, 162.85, 163.96,
       168.17, 168.25, 172.65, 165.82, 168.17, 168.57, 167.96, 168.38,
       163.34, 162.95, 164.04, 168.1 , 166.9 , 164.96, 163.83, 166.39,
       167.44, 166.21, 167.75, 168.18, 170.96, 169.25, 168.55, 167.69,
       164.02, 164.97, 165.83, 166.01, 165.57, 162.08, 161.18, 160.48,
       158.62, 158.39, 159.57, 164.57, 165.2 , 166.05, 166.03, 161.24,
       161.34, 161.67, 168.26, 167.93, 168.91, 168.99, 166.08, 169.28,
       171.49, 173.83, 171.53, 175.56, 171.89, 173.68, 179.99, 177.38,
       175.98, 176.45, 175.23, 171.17, 171.54, 168.86, 170.31, 168.03,
       172.38, 170.44, 169.75, 172.13, 168.99, 169.78, 173.51, 169.33,
       170.01, 170.89, 173.19, 170.02, 166.68, 168.24, 169.21, 170.06,
       171.18, 169.53, 170.33, 171.3 , 169.56, 174.82, 176.32, 173.21,
```

```
       175.2 , 172.7 , 175.05, 178.48, 176.36, 179.26, 180.62, 183.18,
       188.95, 188.42, 186.38, 183.97, 181.81, 181.78, 182.96, 183.95,
       186.57, 186.8 , 191.35, 190.75, 199.34, 201.54, 199.03, 195.9 ,
       197.61, 197.12, 199.57, 201.28, 201.26, 198.79, 194.29, 193.13,
       195.97, 196.84, 193.21, 193.94, 195.33, 192.83, 193.5 , 193.88,
       190.61, 191.9 , 193.81, 197.36, 200.89, 196.77, 194.08, 195.87,
       197.68, 199.5 , 212.26, 214.41, 218.52, 222.07, 224.61, 223.73,
       226.67, 224.15, 223.55, 222.08, 222.6 , 231.14, 224.65, 227.34,
       221.85, 220.44, 224.5 , 224.97, 221.4 , 219.14, 221.74, 219.43,
       220.42, 220.81, 218.78, 226.56, 229.08, 222.1 , 222.99, 220.9 ,
       224.13, 221.58, 221.41, 223.42, 226.74, 222.71, 221.77, 220.34,
       218.31, 224.72, 226.41, 222.17])
```

We will combine `simulate_gbm()` with a list comprehension and `pd.concat()` to simulate many price paths. To simplify this combination, we will write a helper function `simulate_gbm_series()` that:

1. Returns a series
2. Helps us vary the `seed` argument

```
def simulate_gbm_series(seed, train=train, test=test):
    S_t = simulate_gbm(
        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],
        seed=seed
    )
    return pd.Series(data=S_t, index=test.index.insert(0, train.index[-1]))
```

```
n = 100

S_t = pd.concat(
    objs=[simulate_gbm_series(seed=seed) for seed in range(n)],
    axis=1,
    keys=range(n),
    names=['Simulation']
)
```

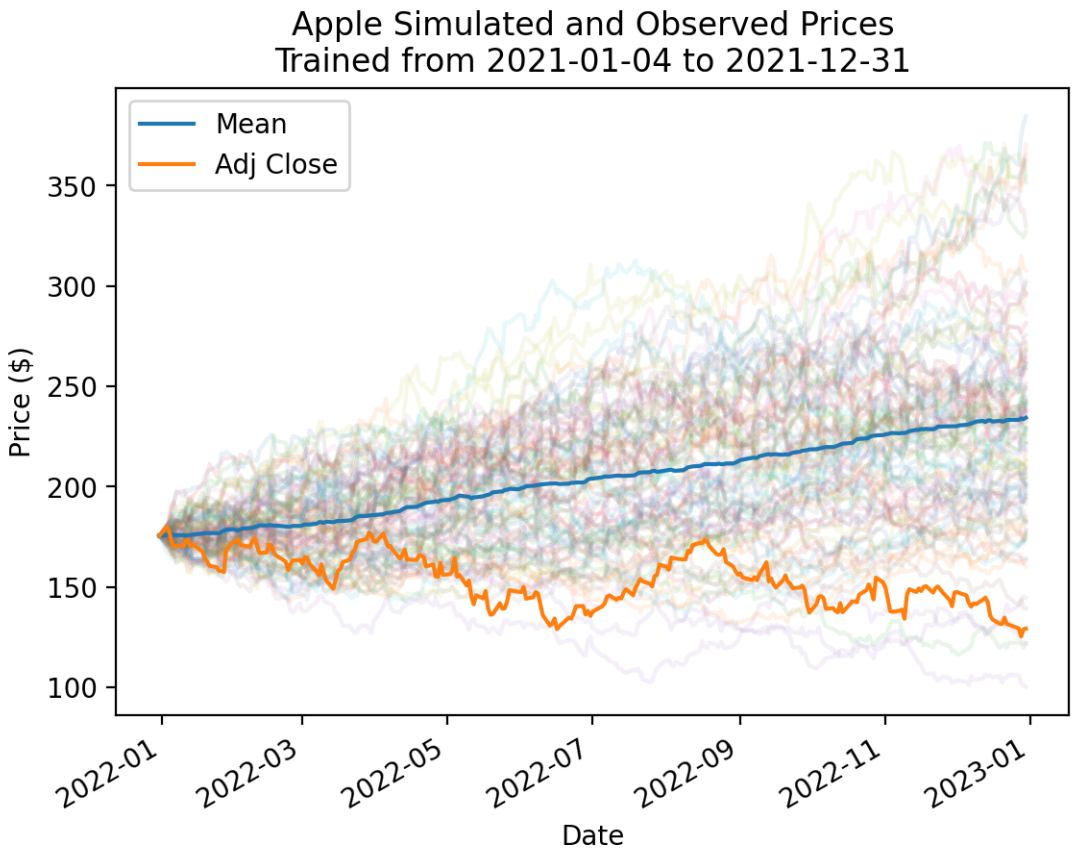
```
S_t
```

Simulation Date	0	1	2	3	4	5	6	7	8	9	...	90
2021-12-31	175.56	175.56	175.56	175.56	175.56	175.56	175.56	175.56	175.56	175.56	...	175.5
2022-01-03	180.72	180.32	174.60	180.79	175.89	176.98	174.89	180.51	176.00	175.75	...	175.1
2022-01-04	182.06	178.77	174.62	182.23	177.47	176.24	177.10	179.37	179.25	175.13	...	174.6
2022-01-05	185.09	177.47	169.00	182.70	174.89	183.34	177.90	179.66	174.01	172.25	...	173.2
2022-01-06	191.97	174.67	173.63	177.59	177.00	182.80	175.57	181.01	170.41	172.40	...	171.2
...	...	...	...	...	...	...	...	...	...	...	...	...
2022-12-23	263.30	307.78	208.45	242.89	279.85	227.13	237.51	207.87	216.25	252.80	...	238.9
2022-12-27	268.12	305.95	209.92	251.23	278.82	226.46	234.31	206.16	212.62	259.22	...	232.6
2022-12-28	264.97	315.16	211.85	255.79	275.85	224.22	238.65	206.03	211.34	255.58	...	237.0
2022-12-29	259.17	309.05	210.96	256.73	274.65	225.95	238.08	201.73	211.18	258.81	...	241.6
2022-12-30	261.59	307.70	210.24	264.08	275.53	225.67	242.50	201.41	213.21	261.36	...	236.7

Below, we prefix the simulated price path column names with _ to hide them from the legend. However, this feature triggers a warning *100 times*! We will suppress these 100 warnings with the warnings package. Generally, we should avoid suppressing warnings, however it is the easiest option here.

```
import warnings
```

```
fig, ax = plt.subplots(1,1)
with warnings.catch_warnings():
    warnings.simplefilter("ignore")
    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(
    'Apple Simulated and Observed Prices' +
    f'\nTrained from {train.index[0]:%Y-%m-%d} to {train.index[-1]:%Y-%m-%d}'
)
plt.show()
```



26.1.2 Pricing Stock Options

We can use simulated price paths to price options! We will use the Black and Scholes (1973) formula as a benchmark. Black and Scholes (1973) provide a closed form (analytic) solution to price European options.

```
from scipy.stats import norm
```

```
def price_bs(S_0, K, T, r, sigma, type='call'):
    '''
    Function used for calculating the price of European options using the analytical form

    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('Wrong input for type!')

    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 expiration at the end of 2022, assuming 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()
```

```
S_0
```

```
175.56
```

```
_ = price_bs(
    S_0=S_0,
    K=K,
    T=T,
    r=r,
    sigma=sigma
)

print(f'Black and Scholes (1973) option price: {_:0.2f}')
```

```
Black and Scholes (1973) option price: 80.50
```

To simulate the Black and Scholes (1973) option price, we must simulate AAPL prices with the same 5% risk-free rate as the drift. We will write a new helper function to use the same inputs as above.

```
def simulate_gbm_series(seed, S_0=S_0, T=T, r=r, sigma=sigma, train=train, test=test):
    S_t = simulate_gbm(
        S_0=S_0,
        mu=r,
        sigma=sigma,
        n_steps=T,
        seed=seed
    )
    return pd.Series(data=S_t, index=test.index.insert(0, train.index[-1]))
```

We will simulate more price paths to increase the precision of out option price.

```
n = 10_000

S_t = pd.concat(
    objs=[simulate_gbm_series(seed=seed) for seed in range(n)],
    axis=1,
    keys=range(n),
    names=['Simulation']
)
```

```
S_t
```

Simulation Date	0	1	2	3	4	5	6	7	8	9	...	9990
2021-12-31	175.56	175.56	175.56	175.56	175.56	175.56	175.56	175.56	175.56	175.56	...	175.5
2022-01-03	180.54	180.14	174.42	180.61	175.72	176.81	174.71	180.33	175.83	175.58	...	177.4
2022-01-04	181.70	178.42	174.28	181.88	177.12	175.90	176.75	179.02	178.90	174.79	...	176.7
2022-01-05	184.55	176.95	168.51	182.17	174.37	182.80	177.37	179.13	173.49	171.75	...	179.4
2022-01-06	191.21	173.99	172.95	176.89	176.31	182.09	174.88	180.30	169.75	171.72	...	182.3
...	...	...	...	...	...	...	...	...	...	...	...	...
2022-12-23	206.58	241.48	163.55	190.57	219.57	178.21	186.35	163.09	169.67	198.34	...	155.5
2022-12-27	210.16	239.81	164.54	196.92	218.54	177.50	183.66	161.60	166.65	203.18	...	159.6
2022-12-28	207.49	246.79	165.89	200.30	216.00	175.58	186.88	161.33	165.49	200.14	...	157.4
2022-12-29	202.75	241.77	165.03	200.84	214.86	176.76	186.25	157.81	165.20	202.46	...	155.9
2022-12-30	204.44	240.47	164.31	206.39	215.33	176.37	189.52	157.40	166.63	204.26	...	155.8

We can compare this price to a simulated price. The payoff on the call option is $S_T - K$ or 0, whichever is higher. The price of the option is the present value of the mean payoff, discounted at the risk-free rate.

```
payoff = np.maximum(S_t.iloc[-1] - K, 0)
_ = payoff.mean() * np.exp(-r * T)

print(f'Simulated option price: {_:0.2f}')
```

```
Simulated option price: 81.21
```

The option prices do not match exactly. However, we can simulate more price paths to bring our simulated option price closer to the analytic solution.

26.2 Estimating Value-at-Risk using Monte Carlo

Value-at-Risk (VaR) measures the risk associated with a portfolio VaR reports the worst expected loss, at a given level of confidence, over a certain horizon under normal market conditions. For example, say the 1-day 95% VaR of our portfolio is \$100. This implies that that 95% of the time (under normal market conditions), we should not lose more than \$100 over one day. We typically present VaR as a positive value, so a VaR of \$100 implies a loss of less than \$100.

We can calculate VaR several ways, including:

- Parametric Approach (Variance-Covariance)

- Historical Simulation Approach
- Monte Carlo simulations

We only consider the last method to calculate the 1-day VaR of an portfolio of 20 shares each of META and GOOG.

```
tickers = ['GOOG', 'META']
shares = np.array([20, 20])
T = 1
n_sims = 10_000
```

However, we will download all data from Yahoo! Finance and subset our data later.

```
df = (
    yf.download(tickers=tickers)
    .rename_axis(columns=['Variable', 'Ticker'])
)
```

```
[**********100%%**********]  2 of 2 completed
```

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']
```

```
returns
```

Ticker Date	GOOG	META
2022-01-03	0.00	0.01
2022-01-04	-0.00	-0.01
2022-01-05	-0.05	-0.04
2022-01-06	-0.00	0.03
2022-01-07	-0.00	-0.00
...	...	...
2022-12-23	0.02	0.01
2022-12-27	-0.02	-0.01
2022-12-28	-0.02	-0.01
2022-12-29	0.03	0.04
2022-12-30	-0.00	0.00

We will need the variance-covariance matrix.

```
cov_mat = returns.cov()

cov_mat * 1_000_000
```

Ticker Ticker	GOOG	META
GOOG	596.48	674.26
META	674.26	1638.50

We will use the variance-covariance matrix to calculate the Cholesky decomposition.

```
chol_mat = np.linalg.cholesky(cov_mat)

chol_mat
```

```
array([[0.02, 0.  ],
       [0.03, 0.03]])
```

The Cholesky decomposition helps us generate random variables with the same variance and covariance as the observed data.

```
rv = np.random.normal(size=(n_sims, 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.  ]])
```

These random variables have a variance-covariance matrix similar to the real data.

```
  np.cov(correlated_rv.T)  * 1_000_000
```

```
array([[ 597.18,  685.37],
       [ 685.37, 1647.31]])
```

```
  np.allclose(cov_mat, np.cov(correlated_rv.T), rtol=0.05)
```

```
True
```

```
  np.mean(correlated_rv, axis=0) * 100
```

```
array([0.02, 0.02])
```

```
  returns.mean().values * 100
```

```
array([-0.16, -0.32])
```

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([[144.79, 336.82],
       [144.74, 336.38],
       [144.61, 335.63],
       ...,
       [144.53, 336.3 ],
       [144.65, 334.95],
       [144.6 , 336.2 ]])
```

Calculate terminal portfolio values and returns. Note that these are dollar values, since VaR is typically expressed in dollar values.

```
P_T = S_T.dot(shares)

P_T
```

```
array([9632.23, 9622.51, 9604.65, ..., 9616.48, 9592.05, 9615.98])
```

```
P_diff = P_T - P_0

P_diff
```

```
array([ 11.64,   1.92, -15.94, ...,  -4.11, -28.54,  -4.61])
```

```
P_diff.mean()
```

```
-4.38
```

Next, we calculate VaR.

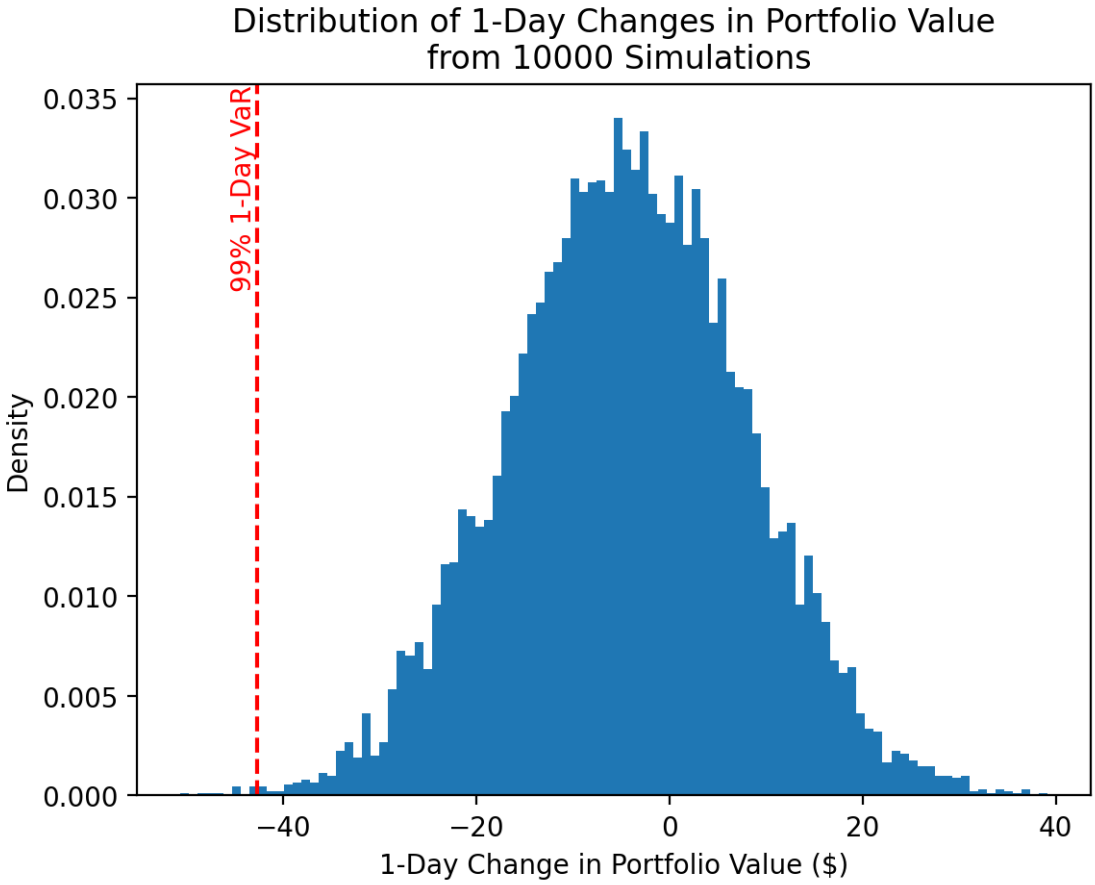
```
percentiles = [0.01, 0.05, 0.1]
var = np.percentile(P_diff, percentiles)

for x, y in zip(percentiles, var):
    print(f'1-day VaR with {100-x}% confidence: ${-y:.2f}')
```

```
1-day VaR with 99.99% confidence: $48.02
1-day VaR with 99.95% confidence: $44.65
1-day VaR with 99.9% confidence: $42.63
```

Finally, we will plot VaR:

```
fig, ax = plt.subplots()
ax.hist(P_diff, bins=100, density=True)
ax.set_title(f'Distribution of 1-Day Changes in Portfolio Value\n from {n_sims} Simulation
ax.axvline(x=var[2], color='red', ls='--')
```

```
ax.text(x=var[2], y=1, s='99% 1-Day VaR', color='red', ha='right', va='top', rotation=90,
ax.set_ylabel('Density')
ax.set_xlabel('1-Day Change in Portfolio Value ($)')
plt.show()
```



27 Herron Topic 5 - Practice for Section 04

27.1 Announcements

27.2 Practice

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
```

```
%precision 2
pd.options.display.float_format = '{:.2f}'.format
%config InlineBackend.figure_format = 'retina'
```

```
import yfinance as yf
import pandas_datareader as pdr
```

27.2.1 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 $f = \frac{\pi r^2}{4r^2}$.

27.2.2 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.

27.2.3 Repeat the exercise above but add end-of-month investments C_t

Part XV

Week 15

28 Project 3

FINA 6333 – Spring 2023

To be determined