

Data Analytics in Finance

FINA 6333 for 2024 Spring

Richard Herron

Table of contents

Welcome!	10
I Week 1	11
1 McKinney Chapter 2 - Python Language Basics, IPython, and Jupyter Notebooks	12
1.1 Introduction	12
1.2 Language Semantics	12
1.3 Scalar Types	23
1.4 Control Flow	27
2 McKinney Chapter 2 - Practice for Section 02	32
2.1 Announcements	32
2.2 5-minute Recap	32
2.3 Practice	34
3 McKinney Chapter 2 - Practice for Section 03	43
3.1 Announcements	43
3.2 5-minute Recap	43
3.3 Practice	45
4 McKinney Chapter 2 - Practice for Section 04	55
4.1 Announcements	55
4.2 5-minute Recap	55
4.3 Practice	57
5 McKinney Chapter 2 - Practice for Section 05	68
5.1 Announcements	68
5.2 5-Minutes Recap	68
5.3 Practice	70
II Week 2	79
6 McKinney Chapter 3 - Built-In Data Structures, Functions, and Files	80
6.1 Introduction	80

6.2	Data Structures and Sequences	80
6.3	List, Set, and Dict Comprehensions	91
6.4	Functions	93
7	McKinney Chapter 3 - Practice for Section 02	96
7.1	Announcements	96
7.2	10-minute Recap	96
7.3	Practice	99
8	McKinney Chapter 3 - Practice for Section 03	108
8.1	Announcements	108
8.2	10-minute Recap	108
8.3	Practice	111
9	McKinney Chapter 3 - Practice for Section 04	121
9.1	Announcements	121
9.2	10-minute Recap	121
9.3	Practice	125
10	McKinney Chapter 3 - Practice for Section 05	134
10.1	Announcements	134
10.2	10-minute Recap	134
10.3	Practice	137
III	Week 3	147
11	McKinney Chapter 4 - NumPy Basics: Arrays and Vectorized Computation	148
11.1	Introduction	148
11.2	The NumPy ndarray: A Multidimensional Array Object	150
11.3	Universal Functions: Fast Element-Wise Array Functions	162
11.4	Array-Oriented Programming with Arrays	165
12	McKinney Chapter 4 - Practice for Section 02	170
12.1	Announcements	170
12.2	10-minute Recap	170
12.3	Practice	174
13	McKinney Chapter 4 - Practice for Section 03	186
13.1	Announcements	186
13.2	10-minute Recap	186
13.3	Practice	190

14 McKinney Chapter 4 - Practice for Section 04	202
14.1 Announcements	202
14.2 10-minute Recap	202
14.3 Practice	206
15 McKinney Chapter 4 - Practice for Section 05	217
15.1 Announcements	217
15.2 10-minute Recap	217
15.3 Practice	221
IV Week 4	232
16 McKinney Chapter 5 - Getting Started with pandas	233
16.1 Introduction	233
16.2 Introduction to pandas Data Structures	234
16.3 Essential Functionality	243
16.4 Summarizing and Computing Descriptive Statistics	258
17 McKinney Chapter 5 - Practice for Section 02	263
17.1 Announcements	263
17.2 10-Minute Recap	263
17.3 Practice	266
18 McKinney Chapter 5 - Practice for Section 03	280
18.1 Announcements	280
18.2 10-Minute Recap	280
18.3 Practice	282
19 McKinney Chapter 5 - Practice for Section 04	296
19.1 Announcements	296
19.2 10-Minute Recap	296
19.3 Practice	298
20 McKinney Chapter 5 - Practice for Section 05	312
20.1 Announcements	312
20.2 10-Minute Recap	312
20.3 Practice	314
V Week 5	329
21 Herron Topic 1 - Web Data, Log and Simple Returns, and Portfolio Math	330
21.1 Web Data	330

21.2 Log and Simple Returns	335
21.3 Portfolio Math	341
22 Herron Topic 1 - Practice for Section 02	344
22.1 Announcements	344
22.2 10-Minute Recap	344
22.3 Practice	345
23 Herron Topic 1 - Practice for Section 03	365
23.1 Announcements	365
23.2 10-Minute Recap	365
23.3 Practice	366
24 Herron Topic 1 - Practice for Section 04	383
24.1 Announcements	383
24.2 10-Minute Recap	383
24.3 Practice	384
25 Herron Topic 1 - Practice for Section 05	401
25.1 Announcements	401
25.2 10-Minute Recap	401
25.3 Practice	402
VI Week 6	420
26 McKinney Chapter 8 - Data Wrangling: Join, Combine, and Reshape	421
26.1 Introduction	421
26.2 Hierarchical Indexing	421
26.3 Combining and Merging Datasets	430
26.4 Reshaping and Pivoting	444
27 McKinney Chapter 8 - Practice for Section 02	448
27.1 Announcements	448
27.2 10-Minute Recap	448
27.3 Practice	449
28 McKinney Chapter 8 - Practice for Section 03	457
28.1 Announcements	457
28.2 10-Minute Recap	457
28.3 Practice	458
29 McKinney Chapter 8 - Practice for Section 04	461
29.1 Announcements	461

29.2 10-Minute Recap	461
29.3 Practice	462
30 McKinney Chapter 8 - Practice for Section 05	465
30.1 Announcements	465
30.2 10-Minute Recap	465
30.3 Practice	466
VII Week 7	469
31 Project 1	470
VIII Week 8	471
32 McKinney Chapter 10 - Data Aggregation and Group Operations	472
32.1 Introduction	472
32.2 GroupBy Mechanics	473
32.3 Data Aggregation	480
32.4 Apply: General split-apply-combine	483
32.5 Pivot Tables and Cross-Tabulation	483
33 McKinney Chapter 10 - Practice for Section 02	486
33.1 Announcements	486
33.2 Practice	486
34 McKinney Chapter 10 - Practice for Section 03	489
34.1 Announcements	489
34.2 Practice	489
35 McKinney Chapter 10 - Practice for Section 04	492
35.1 Announcements	492
35.2 Practice	492
36 McKinney Chapter 10 - Practice for Section 05	495
36.1 Announcements	495
36.2 Practice	495
IX Week 9	498
37 McKinney Chapter 11 - Time Series	499
37.1 Introduction	499
37.2 Time Series Basics	500

37.3 Date Ranges, Frequencies, and Shifting	508
37.4 Resampling and Frequency Conversion	513
37.5 Moving Window Functions	517
38 McKinney Chapter 11 - Practice for Section 02	523
38.1 Announcements	523
38.2 Practice	523
39 McKinney Chapter 11 - Practice for Section 03	525
39.1 Announcements	525
39.2 Practice	525
40 McKinney Chapter 11 - Practice for Section 04	527
40.1 Announcements	527
40.2 Practice	527
41 McKinney Chapter 11 - Practice for Section 05	529
41.1 Announcements	529
41.2 Practice	529
X Week 10	531
42 Herron Topic 2 - Trading Strategies	532
42.1 What is technical analysis?	532
42.2 Why might trading strategies based on technical analysis work or not?	533
42.3 Implement a simple moving average (SMA) trading strategy	540
43 Herron Topic 2 - Practice for Section 02	545
43.1 Announcements	545
43.2 Practice	545
44 Herron Topic 2 - Practice for Section 03	547
44.1 Announcements	547
44.2 Practice	547
45 Herron Topic 2 - Practice for Section 04	549
45.1 Announcements	549
45.2 Practice	549
46 Herron Topic 2 - Practice for Section 05	551
46.1 Announcements	551
46.2 Practice	551

XI Week 11	553
47 Project 2	554
XII Week 12	555
48 Herron Topic 3 - Multifactor Models	556
48.1 The Capital Asset Pricing Model (CAPM)	556
48.2 Multifactor Models	570
49 Herron Topic 3 - Practice for Section 02	573
49.1 Announcements	573
49.2 Practice	573
50 Herron Topic 3 - Practice for Section 03	576
50.1 Announcements	576
50.2 Practice	576
51 Herron Topic 3 - Practice for Section 04	579
51.1 Announcements	579
51.2 Practice	579
52 Herron Topic 3 - Practice for Section 05	582
52.1 Announcements	582
52.2 Practice	582
XIIIWeek 13	585
53 Herron Topic 4 - Portfolio Optimization	586
53.1 The $\frac{1}{n}$ Portfolio	586
53.2 SciPy's <code>minimize()</code> Function	591
54 Herron Topic 4 - Practice for Section 02	600
54.1 Announcements	600
54.2 Practice	600
55 Herron Topic 4 - Practice for Section 03	602
55.1 Announcements	602
55.2 Practice	602
56 Herron Topic 4 - Practice for Section 04	604
56.1 Announcements	604
56.2 Practice	604

57 Herron Topic 4 - Practice for Section 05	606
57.1 Announcements	606
57.2 Practice	606
XIV Week 14	608
58 Herron Topic 5 - Simulations	609
58.1 Option Pricing	610
58.2 Estimating Value-at-Risk using Monte Carlo	618
59 Herron Topic 5 - Practice for Section 02	624
59.1 Announcements	624
59.2 Practice	624
60 Herron Topic 5 - Practice for Section 03	625
60.1 Announcements	625
60.2 Practice	625
61 Herron Topic 5 - Practice for Section 04	626
61.1 Announcements	626
61.2 Practice	626
62 Herron Topic 5 - Practice for Section 05	627
62.1 Announcements	627
62.2 Practice	627
XV Week 15	628
63 Project 3	629

Welcome!

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

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

1.1 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.

1.2 Language Semantics

1.2.1 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.

1.2.2 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.

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

1.2.3 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
```

```
%who
```

```
add_numbers  array  greater      less      pivot      x
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
```

```
%who
```

```
add_numbers  add_strings      array  greater      less      pivot      x
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_strings_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)'
```

1.2.4 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]
```

```
a is b
```

```
True
```

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]
```

```
a is b
```

```
True
```

The behavior is useful but a double-edged sword! [Here](#) is a deeper discussion of this behavior.

1.2.5 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
```

1.2.6 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'  
  
a.upper().lower()  
  
'foo'
```

1.2.7 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
```

1.2.8 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 (**!=** in Python is like **<>** in Excel).

```
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

1.2.9 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)  
  
a_list  
  
['foo', 2, (3, 4)]
```

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

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

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

```
# a_tuple[1] = 'four'
```

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.

1.3 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

1.3.1 Numeric types

In Python, integers are unbounded, and `**` raises numbers to a power. So, `ival ** 6` is 17239781⁶.

```
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
```

1.3.2 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

1.3.3 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'
```

1.3.4 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
```

1.4 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`.

1.4.1 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

1.4.2 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

1.4.3 `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 iterator 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
```



```
val
```

4

1.4.4 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'
```

2 McKinney Chapter 2 - Practice for Section 02

2.1 Announcements

There are no due dates this week, but please take note of a few due dates next week:

1. Before class on Tuesday, complete the week 2 pre-class quiz
2. By Friday 1/19 at 11:59 PM:
 1. Acknowledge the academic integrity section of our syllabus
 2. Complete the first DataCamp course and upload your certificate

2.2 5-minute Recap

2.2.1 First, Python variables are references to objects.

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

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

```
b = a
b
```

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

```
a is b
```

```
True
```

Python is zero-indexed!

```
a[0]
```

```
1
```

```
a[0] = 2_001  
a
```

```
[2001, 2, 3]
```

So both variables reflect changes to the referenced object.

```
b
```

```
[2001, 2, 3]
```

We can also reassign a new object, which does not change the original object.

```
a = '0ch'  
a
```

```
'0ch'
```

```
b
```

```
[2001, 2, 3]
```

```
b = a  
b
```

```
'0ch'
```

2.2.2 Second, we can define functions to re-use code

```
def square_root(x):  
    return x ** 0.5
```

```
square_root(4)
```

2.0

```
square_root(9)
```

3.0

Note that white space is required and not just cosmetic! Our `square_root()` function does not work without the white space!

```
# def square_root(x):
#     return x ** 0.5
#     Cell In[26], line 2
#         return x ** 0.5
#             ^
# IndentationError: expected an indented block after function definition on line 1
```

2.2.3 Third, if-elif-else blocks let us conditionally run code blocks

Again, note the white space is required and not just cosmetic.

```
x = 5
if x < 0:
    print(f'{x} is negative')
elif x == 0:
    print(f'{x} is zero')
else:
    print(f'{x} is positive')
```

5 is positive

2.3 Practice

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

Try 20080915.

```
lb_collapse = 20080915
```

```
lb_collapse / 10_000
```

2008.0915

Integer division // returns the integer portion.

```
lb_collapse // 10_000
```

2008

Modulo division % returns the remainder.

```
lb_collapse % 10_000 // 100
```

9

```
lb_collapse % 100
```

15

2.3.2 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)).

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

```
date(lb_collapse)
```

(2008, 9, 15)

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

```
def date_2(ymd):
    if type(ymd) is str:
        ymd = int(ymd)
    year = ymd // 10_000
    month = ymd % 10_000 // 100
    day = ymd % 100
    return (year, month, day)
```

```
date_2('20080915')
```

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

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

```
dates = [20080915, '20080916']
```

```
def date_3(ymd):
    ymds = [] # empty list to store our tuples of (year, month, day)
    for ymd in dates:
        if type(ymd) is str:
            ymd = int(ymd)
        year = ymd // 10_000
        month = ymd % 10_000 // 100
        day = ymd % 100
        ymds.append((year, month, day)) # add new (year, month, day) to end of our list

    return ymds
```

```
date_3(dates)
```

```
[(2008, 9, 15), (2008, 9, 16)]
```

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

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

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

1 4 9 16 25 36 49 64 81 100

Above, I change the `end` argument to a space to shorten the output.

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

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

4 16 36 64 100

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

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

total
```

385

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

```
total = 0
for i in range(1, 11):
    if (total + i**2) > 50:
        break
    total += i**2
```

```
total
```

30

2.3.9 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](#).

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

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

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

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

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

The solution above is shorter and uses some neat tricks, but I consider the previous solution easier to read and troubleshoot. The solution above uses the trick that we can multiply a string by `True` or `False` to return the string itself or an empty string.

```
'Hey!' * True
```

```
'Hey! '
```

```
'Hey!' *False
```

```
'''
```

2.3.11 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
```

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

triangle(6)
```

```
1
22
333
4444
55555
```

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

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

```
'TestTestTest'
```

```
'Test' * 3
```

```
'TestTestTest'
```

2.3.12 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](#).

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

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

```
[0, 1]
```

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

```
[1, 2]
```

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

```
[0, 1]
```

We can write more efficient code once we learn other data structures in chapter 3 of McKinney!

2.3.13 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](#).

```
def max_profit(prices):
    min_price = prices[0]
    max_profit = 0
    for price in prices:
        min_price = price if price < min_price else min_price
        profit = price - min_price
```

```
        max_profit = profit if profit > max_profit else max_profit
    return max_profit
```

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

5

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

0

3 McKinney Chapter 2 - Practice for Section 03

3.1 Announcements

There are no due dates this week, but please take note of a few due dates next week:

1. Before class on Tuesday, complete the week 2 pre-class quiz
2. By Friday 1/19 at 11:59 PM:
 1. Acknowledge the academic integrity section of our syllabus
 2. Complete the first DataCamp course and upload your certificate

3.2 5-minute Recap

3.2.1 First, Python variables are references to objects.

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

```
a
```

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

```
b = a
```

```
b
```

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

Python is zero-indexed!

```
b[0]
```

```
1
```

```
b[0] = 2_001
```

```
b
```

```
[2001, 2, 3]
```

So both variables reflect changes to the referenced object.

```
a
```

```
[2001, 2, 3]
```

Variables `a` and `b` are two names for the same list in memory!

```
b is a
```

```
True
```

We can also reassign a new object, which does not change the original object.

```
a = '0ch'
```

```
a
```

```
'0ch'
```

```
b
```

```
[2001, 2, 3]
```

3.2.2 Second, we can define functions to re-use code

```
def square_root(x):
    return x ** 0.5
```

```
square_root(4)
```

```
2.0
```

```
square_root(9)
```

```
3.0
```

Note that white space is required and not just cosmetic! Our `square_root()` function does not work without the white space!

```
# def square_root(x):
#     return x ** 0.5
#     Cell In[26], line 2
#         return x ** 0.5
#             ^
# IndentationError: expected an indented block after function definition on line 1
```

3.2.3 Third, if-elif-else blocks let us conditionally run code blocks

Again, note the white space is required and not just cosmetic.

```
x = 5
if x < 0:
    print(f'{x} is negative')
elif x == 0:
    print(f'{x} is zero')
else:
    print(f'{x} is positive')
```

```
5 is positive
```

3.3 Practice

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

Try 20080915.

```
1234 / 100
```

12.34

```
1234 // 100
```

12

```
1234 % 100
```

34

```
12 / 5
```

2.4

```
12 // 5
```

2

```
12 % 5
```

2

```
lb_collapse = 20080915
```

```
lb_collapse // 10_000
```

2008

```
lb_collapse % 10_000 // 100
```

9

```
lb_collapse % 100
```

15

3.3.2 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)).

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

```
date(lb_collapse)
```

(2008, 9, 15)

```
date(20240112)
```

(2024, 1, 12)

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

```
def date_2(ymd):
    if type(ymd) is str:
        ymd = int(ymd)
    year = ymd // 10_000
    month = ymd % 10_000 // 100
    day = ymd % 100
    return (year, month, day)
```

```
date_2('20080915')
```

(2008, 9, 15)

3.3.4 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.

```
ymds = [20080915, '20080916']

def date_3(ymds):
    dates = []
    for ymd in ymds:
        if type(ymd) is str:
            ymd = int(ymd)
        year = ymd // 10_000
        month = ymd % 10_000 // 100
        day = ymd % 100
        dates.append((year, month, day))
    return dates

date_3(ymds)
```

```
[(2008, 9, 15), (2008, 9, 16)]
```

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

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

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

Above, I change the `end` argument to a space to shorten the output.

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

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

```
4 16 36 64 100
```

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

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

total
```

385

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

```
total = 0
for i in range(1, 11):
    if (total + i**2) > 50:
        break
    total += i**2

total
```

30

3.3.9 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](#).

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

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

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

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

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

The solution above is shorter and uses from neat tricks, but I consider the previous solution easier to read and troubleshoot. The solution above uses the trick that we can multiply a string by `True` or `False` to return the string itself or an empty string.

```
'Hey! '*True
```

```
'Hey! '
```

```
'Hey! '*False
```

3.3.11 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
```

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

```
triangle(6)
```

```
1
22
333
4444
55555
```

The solution above works because multiplying a string by *i* concatenates *i* copies of the string.

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

```
'TestTestTest'
```

```
'Test' * 3
```

```
'TestTestTest'
```

3.3.12 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](#).

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

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

```
[0, 1]
```

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

```
[1, 2]
```

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

```
[0, 1]
```

We can write more efficient code once we learn other data structures in chapter 3 of McKinney!

3.3.13 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](#).

```
def max_profit(prices):
    min_price = prices[0]
    max_profit = 0
    for price in prices:
        min_price = price if price < min_price else min_price
        profit = price - min_price
        max_profit = profit if profit > max_profit else max_profit
    return max_profit
```

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

5

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

0

4 McKinney Chapter 2 - Practice for Section 04

4.1 Announcements

There are no due dates this week, but please take note of a few due dates next week:

1. Before class on Tuesday, complete the week 2 pre-class quiz
2. By Friday 1/19 at 11:59 PM:
 1. Acknowledge the academic integrity section of our syllabus
 2. Complete the first DataCamp course and upload your certificate

4.2 5-minute Recap

4.2.1 First, Python variables are references to objects.

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

```
a
```

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

```
b = a
```

```
b
```

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

Python is zero-indexed!

```
b[0]
```

```
1
```

```
b[0] = 2_001
```

```
b
```

```
[2001, 2, 3]
```

So both variables reflect changes to the referenced object.

```
a
```

```
[2001, 2, 3]
```

Variables `a` and `b` are two names for the same list in memory!

```
b is a
```

```
True
```

We can also reassign a new object, which does not change the original object.

```
a = '0ch'
```

```
a
```

```
'0ch'
```

```
b
```

```
[2001, 2, 3]
```

4.2.2 Second, we can define functions to re-use code

```
def square_root(x):
    return x ** 0.5
```

```
square_root(4)
```

```
2.0
```

```
square_root(9)
```

```
3.0
```

Note that white space is required and not just cosmetic! Our `square_root()` function does not work without the white space!

```
# def square_root(x):
#     return x ** 0.5
#     Cell In[26], line 2
#         return x ** 0.5
#             ^
# IndentationError: expected an indented block after function definition on line 1
```

4.2.3 Third, if-elif-else blocks let us conditionally run code blocks

Again, note the white space is required and not just cosmetic.

```
x = 5
if x < 0:
    print(f'{x} is negative')
elif x == 0:
    print(f'{x} is zero')
else:
    print(f'{x} is positive')
```

```
5 is positive
```

4.3 Practice

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

Try 20080915.

```
1234 / 100
```

12.34

```
1234 // 100
```

12

```
1234 % 100
```

34

```
12 / 5
```

2.4

```
12 // 5
```

2

```
12 % 5
```

2

```
lb_collapse = 20080915
```

```
lb_collapse // 10_000
```

2008

```
lb_collapse % 10_000 // 100
```

9

```
lb_collapse % 100
```

15

4.3.2 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, day)).

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

```
date(lb_collapse)
```

(2008, 9, 15)

```
date(20240112)
```

(2024, 1, 12)

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

```
def date_2(ymd):
    if type(ymd) is str:
        ymd = int(ymd)
    year = ymd // 10_000
    month = ymd % 10_000 // 100
    day = ymd % 100
    return (year, month, day)
```

```
date_2('20080915')
```

(2008, 9, 15)

4.3.4 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.

```
ymds = [20080915, '20080916']

def date_3(ymds):
    dates = []
    for ymd in ymds:
        if type(ymd) is str:
            ymd = int(ymd)
        year = ymd // 10_000
        month = ymd % 10_000 // 100
        day = ymd % 100
        dates.append((year, month, day))
    return dates

date_3(ymds)
```

```
[(2008, 9, 15), (2008, 9, 16)]
```

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

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

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

Above, I change the `end` argument to a space to shorten the output.

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

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

```
4 16 36 64 100
```

We can also change the arguments to `range()` to start at 2 and take steps of 2!

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

4 16 36 64 100

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

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

sum
```

385

We can replace the `sum = sum + i**2` with `sum += i**2`.

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

sum
```

385

There are also `-=`, `*=`, and `/=` operators.

```
sum = 1
for i in range(1, 11):
    sum *= i**2

sum
```

13168189440000

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

```
total = 0
for i in range(1, 11):
    if (total + i**2) > 50:
        break
    total += i**2

total
```

30

4.3.9 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](#).

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

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

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

```
'Fizz' + 'Buzz'
```

```
'FizzBuzz'
```

```
'Fizz' + 'Fizz'
```

```

'FizzFizz'

    'Fizz' * 2

'FizzFizz'

    'Fizz' * 0

    ...

    'Fizz'*True + 'Buzz'*True

'FizzBuzz'

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

```

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

The solution above is shorter and uses some neat tricks, but I consider the previous solution easier to read and troubleshoot. The solution above uses the trick that we can multiply a string by `True` or `False` to return the string itself or an empty string.

```

'Hey!' *True

'Hey!'

'Hey!' *False

    ...

```

Someone pointed out an even shorter solution with `or!`

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

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

4.3.11 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
```

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

triangle(6)
```

```
1
22
333
4444
55555
```

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

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

'TestTestTest'

'Test' * 3

'TestTestTest'
```

4.3.12 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](#).

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

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

```
[0, 1]
```

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

```
[1, 2]
```

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

```
[0, 1]
```

We can write more efficient code once we learn other data structures in chapter 3 of McKinney!

4.3.13 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](#).

```
def max_profit(prices):
    min_price = prices[0]
    max_profit = 0
    for price in prices:
        min_price = price if price < min_price else min_price
        profit = price - min_price
```

```
        max_profit = profit if profit > max_profit else max_profit
    return max_profit
```

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

5

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

0

5 McKinney Chapter 2 - Practice for Section 05

5.1 Announcements

There are no due dates this week, but please take note of a few due dates next week:

1. Before class on Tuesday, complete the week 2 pre-class quiz
2. By Friday 1/19 at 11:59 PM:
 1. Acknowledge the academic integrity section of our syllabus
 2. Complete the first DataCamp course and upload your certificate

5.2 5-Minutes Recap

5.2.1 First, Python variables are references to objects.

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

[1, 2, 3]

Two (or more) variables can refer to the same object.

```
b = a
b
```

[1, 2, 3]

Remember, Python is zero-indexed!

```
b[0] = 2001  
b
```

```
[2001, 2, 3]
```

So both variables reflect changes to the referenced object.

```
a
```

```
[2001, 2, 3]
```

We can also reassign a new object, which does not change the original object.

```
b = 'W.T. Door'  
b
```

```
'W.T. Door'
```

```
a
```

```
[2001, 2, 3]
```

5.2.2 Second, we can define functions to re-use code

```
def square_root(x):  
    return x ** 0.5  
  
square_root(4)
```

```
2.0
```

```
square_root(9)
```

```
3.0
```

Note that white space is required and not just cosmetic! Our `square_root()` function does not work without the white space!

```
# def square_root(x):
#     return x ** 0.5
#     Cell In[26], line 2
#         return x ** 0.5
#         ^
# IndentationError: expected an indented block after function definition on line 1
```

5.2.3 Third, if-elif-else blocks let us conditionally run code blocks

Again, note the white space is required and not just cosmetic.

```
x = 5
if x < 0:
    print(f'{x} is negative')
elif x == 0:
    print(f'{x} is zero')
else:
    print(f'{x} is positive')
```

5 is positive

5.3 Practice

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

Try 20080915.

```
1234 / 100
```

12.34

```
1234 // 100
```

12

```
1234 % 100
```

34

```
lb_collapse = 20080915
```

```
lb_collapse // 10_000
```

2008

```
lb_collapse % 10_000 // 100
```

9

```
lb_collapse % 100
```

15

5.3.2 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)).

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

```
date(lb_collapse)
```

(2008, 9, 15)

```
date(20240112)
```

(2024, 1, 12)

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

```
def date_2(ymd):
    if type(ymd) is str:
        ymd = int(ymd)
    year = ymd // 10_000
    month = ymd % 10_000 // 100
    day = ymd % 100
    return (year, month, day)
```

```
date_2('20090915')
```

```
(2009, 9, 15)
```

5.3.4 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.

```
ymds = [20080915, '20080916']

def date_3(ymds):
    dates = []
    for ymd in ymds:
        if type(ymd) is str:
            ymd = int(ymd)
        year = ymd // 10_000
        month = ymd % 10_000 // 100
        day = ymd % 100
        dates.append((year, month, day))
    return dates
```

```
date_3(ymds)
```

```
[(2008, 9, 15), (2008, 9, 16)]
```

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

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

1 4 9 16 25 36 49 64 81 100

Above, I change the `end` argument to a space to shorten the output.

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

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

4 16 36 64 100

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

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

total
```

385

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

```
total = 0
for i in range(1, 11):
    if (total + i**2) > 50:
        break
    total += i**2
```

```
total
```

30

5.3.9 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](#).

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

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

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

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

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

The solution above is shorter and uses some neat tricks, but I consider the previous solution easier to read and troubleshoot. The solution above uses the trick that we can multiply a string by `True` or `False` to return the string itself or an empty string.

```
'Hey!' * True
```

```
'Hey! '
```

```
'Hey!' *False
```

```
'''
```

5.3.11 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
```

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

```
triangle(6)
```

```
1
22
333
4444
55555
```

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

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

```
'TestTestTest'
```

```
'Test' * 3
```

```
'TestTestTest'
```

5.3.12 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](#).

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

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

```
[0, 1]
```

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

```
[1, 2]
```

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

```
[0, 1]
```

We can write more efficient code once we learn other data structures in chapter 3 of McKinney!

5.3.13 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](#).

```
def max_profit(prices):
    min_price = prices[0]
    max_profit = 0
    for price in prices:
        min_price = price if price < min_price else min_price
        profit = price - min_price
```

```
        max_profit = profit if profit > max_profit else max_profit
    return max_profit
```

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

5

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

0

Part II

Week 2

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

6.1 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.

6.2 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.

6.2.1 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')
```

6.2.1.1 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
```

6.2.1.2 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

6.2.2 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)
```

Python is zero-indexed!

```
a_list[0]
```

2

6.2.2.1 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
```

6.2.2.2 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!

6.2.2.3 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)
```

6.2.2.4 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.

6.2.3 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}
```

6.3 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]
```

6.4 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
```

6.4.1 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
```

6.4.2 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']
```

7 McKinney Chapter 3 - Practice for Section 02

7.1 Announcements

1. Due Friday, 1/19, at 11:59 PM:
 1. Complete *Introduction to Python* course on DataCamp (and upload certificate to Canvas)
 2. Acknowledge academic integrity statement on Canvas
2. I will record and post the next lecture video on Thursday, 1/18, and the associate pre-class quiz is due before class on Tuesday, 1/23
3. Start joining groups on Canvas (Canvas > People > Team Projects); I removed joining groups as a scored assignment, but please prioritize joining groups

7.2 10-minute Recap

7.2.1 List

A list is a collection of items that are ordered and changeable. You can add, remove, or modify elements in a list. In Python, you can create an empty list using either [] or `list()`.

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

```
my_list
```

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

Python is zero-indexed!

```
my_list[0]
```

```
1
```

```
my_list[-1][-1][-1]
```

```
3
```

```
my_list[-1][-1][-1] = 'Matilda'
```

```
my_list
```

```
[1, 2, 3, [1, 2, 3, [1, 2, 'Matilda']]])
```

```
my_list[3] is my_list[-1]
```

```
True
```

7.2.2 Tuple

A tuple is similar to a list, but it is immutable, meaning that you cannot change its contents once it has been created. You can create a tuple using parentheses () or the `tuple()` function.

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

```
my_tuple
```

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

Tuples are immutable!

```
# my_tuple[-1][-1][-1] = 'Matilda'  
# -----  
# TypeError Traceback (most recent call last)  
# Cell In[14], line 1  
# ----> 1 my_tuple[-1][-1][-1] = 'Matilda'
```

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

In the following example, we can replace the innermost 3 with 'Matilda' because it is an object in a list.

```
my_tuple_2 = (1, 2, 3, (1, 2, 3, [1, 2, 3]))
```

```
my_tuple_2
```

```
(1, 2, 3, (1, 2, 3, [1, 2, 3]))
```

```
my_tuple_2[-1][-1][-1] = 'Matilda'
```

7.2.3 Dictionary

A dictionary is a collection of key-value pairs that are unordered and changeable. You can add, remove, or modify elements in a dictionary. In Python, you can create an empty dictionary using either {} or the `dict()` function.

```
prices_list = [10, 20, 300] # for AAPL, MSFT, TSLA
```

```
prices_dict = {'AAPL': 10, 'MSFT': 20, 'TSLA': 300}
```

```
prices_dict['AAPL']
```

```
10
```

```
my_dict = {'A': 1, 'B': [1, 2, 3], 'C': {'A': 1, 2: 2}}
```

```
my_dict
```

```
{'A': 1, 'B': [1, 2, 3], 'C': {'A': 1, 2: 2}}
```

7.2.4 List Comprehension

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

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

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

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

```
even_numbers = []
for i in range(9):
    if i%2 == 0:
        even_numbers.append(i)

even_numbers
```

```
[0, 2, 4, 6, 8]
```

Here is the cooler, more Pythonic list comprehension version:

```
[i for i in range(9) if i%2 == 0]
```

```
[0, 2, 4, 6, 8]
```

7.3 Practice

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

```
a = 1
```

```
b = 2
```

```
c = a  
a = b  
b = c  
del c  
print(f'a: {a}, b: {b}')
```

a: 2, b: 1

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

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

a: 2, b: 1

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

The parentheses () are optional for tuples, but they are often useful!

```
_ = 1, 1, 1  
type(_)  
  
tuple  
  
1, 1, 1 == (1, 1, 1)  
  
(1, 1, False)
```

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

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

Here we can wrap both sides with parentheses to remove ambiguity!

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

```
True
```

7.3.4 Create a list 11 of integers from 1 to 100.

```
l1 = list(range(1, 101))
# the last entry in a code cell prints automatically...
# here I wrap l1 with print for fewer lines of printout
print(l1)
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100]
```

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

Name this list 12.

```
# with zero-indexing:
# left edge included, right excluded
# so length of list is right - left
l2 = l1[49:60]
l2.reverse()
l2 # most Python methods modify objects "in place"
```

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

```
l2_alt = l1[49:60][::-1]
```

```
l2_alt
```

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

```
11[-41:-52:-1]
```

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

7.3.6 Create a list 13 of odd integers from 1 to 21.

```
13 = []
for i in range(1, 22):
    if i%2 == 1:
        13.append(i)
```

```
13
```

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

```
13_alt = [i for i in range(1, 22) if i%2 == 1]
13_alt
```

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

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

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

7.3.7 Create a list 14 of the squares of integers from 1 to 100.

```
14 = [i**2 for i in range(1, 101)]
print(14)
```

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, 256, 289, 324, 361, 400, 441]
```

7.3.8 Create a list 15 that contains the squares of *odd* integers from 1 to 100.

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

```
print(15)
```

```
[1, 9, 25, 49, 81, 121, 169, 225, 289, 361, 441, 529, 625, 729, 841, 961, 1089, 1225, 1369, ...]
```

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

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

```
def get_last_letter(x):  
    return x[-1]
```

```
get_last_letter(strings[0])
```

```
'd'
```

I would usually use the `.sort()` method, which would modify the `strings` list *in place*. However, for this exercise I want to leave the `strings` list as is, so I can repeat this exercise a few times.

```
sorted(strings)
```

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

```
sorted(strings, key=get_last_letter)
```

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

Here is the anonymous function version.

```
sorted(strings, key=lambda x: x[-1])
```

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

Anonymous functions are very Pythonic and the ideal way to implement a calculation that you only plan to use once, thereby avoiding the overhead of writing a function.

7.3.10 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](#).

```
def get_largest(x, k):
    return sorted(x)[-k]

get_largest(x=[3,2,1,5,6,4], k=2)
```

5

```
get_largest(x=[3,2,3,1,2,4,5,5,6], k=4)
```

4

7.3.11 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](#).

```

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

get_frequent(nums=[1,1,1,2,2,3], k=2)

```

[1, 2]

7.3.12 Test whether the given strings are palindromes.

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

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

```

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

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

```

[True, False]

7.3.13 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]

```

This type problem is the rare case where I use a loop counter in Python.

```

def returns(p, d):
    rs = []
    for i in range(1, len(p)):
        r = (p[i] + d[i] - p[i-1]) / p[i-1]
        rs.append(r)
    return rs

```

```

        rs.append(r)

    return rs

%precision 4

'%.4f'

returns(p=prices, d=dividends)

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

```

Here is a solution that avoids a loop counter and combines a list comprehension `zip()` to loop over prices, dividends, and lagged prices. This solution may be a little more Pythonic, but I find it harder to follow than our loop-counter solution above.

```

[(p + d - p_lag) / p_lag for p, d, p_lag in zip(prices[1:], dividends[1:], prices[:-1])]

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

```

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

```

def returns_2(p, d):
    rs, d_ps, cg_ps = [], [], []
    for i in range(1, len(p)):
        r = (p[i] + d[i] - p[i-1]) / p[i-1]
        d_p = d[i] / p[i-1]
        cg_p = r - d_p
        rs.append(r)
        d_ps.append(d_p)
        cg_ps.append(cg_p)

    return {'r': rs, 'd_p': d_ps, 'cg': cg_ps}

returns_2(p=prices, d=dividends)

```

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

7.3.15 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]

def rescale(x):
    x_min = min(x)
    x_max = max(x)

    # Here is the for-loop equivalent of the list comprehension below
    # rescaled = []
    # for y in x:
    #     rescaled.append((y - x_min) / (x_max - x_min))

    return [(y - x_min) / (x_max - x_min) for y in x]

rescale(numbers)
```

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

8 McKinney Chapter 3 - Practice for Section 03

8.1 Announcements

1. Due Friday, 1/19, at 11:59 PM:
 1. Complete *Introduction to Python* course on DataCamp (and upload certificate to Canvas)
 2. Acknowledge academic integrity statement on Canvas
2. I will record and post the next lecture video on Thursday, 1/18, and the associate pre-class quiz is due before class on Tuesday, 1/23
3. Start joining groups on Canvas (Canvas > People > Team Projects); I removed joining groups as a scored assignment, but please prioritize joining groups

8.2 10-minute Recap

8.2.1 List

A list is a collection of items that are ordered and changeable. You can add, remove, or modify elements in a list. In Python, you can create an empty list using either [] or `list()`.

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

Python is zero-indexed!

```
my_list[1]
```

2

```
my_list[:1]
```

```
[1]
```

```
my_list[-1] [-1] [-1]
```

```
3
```

```
my_list[-1] [-1] [-1] = 'McKinney'
```

```
my_list
```

```
[1, 2, 3, [1, 2, 3, [1, 2, 'McKinney']]])
```

8.2.2 Tuple

A tuple is similar to a list, but it is immutable, meaning that you cannot change its contents once it has been created. You can create a tuple using parentheses () or the `tuple()` function.

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

```
my_tuple
```

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

Unlike lists, tuples are immutable and cannot be changed!

```
# my_tuple[-1] [-1] [-1] = 'McKinney'  
# -----  
# TypeError Traceback (most recent call last)  
# Cell In[15], line 1  
# ----> 1 my_tuple[-1] [-1] [-1] = 'McKinney'  
# TypeError: 'tuple' object does not support item assignment
```

The following result is surprising! Tuples are immutable, but a list nested inside a tuple is mutable!

```
my_tuple_2 = (1, 2, 3, (1, 2, 3, [1, 2, 3]))  
  
my_tuple_2[-1][-1][-1] = 'McKinney'  
  
my_tuple_2  
  
(1, 2, 3, (1, 2, 3, [1, 2, 'McKinney']))
```

8.2.3 Dictionary

A dictionary is a collection of key-value pairs that are unordered and changeable. You can add, remove, or modify elements in a dictionary. In Python, you can create an empty dictionary using either {} or the `dict()` function.

```
stock_prices = [100, 200, 300] # AAPL, MSFT, TSLA
```

Dictionaries let us index a value by its key!

```
stock_prices = {'AAPL': 100, 'MSFT': 200, 'TSLA': 300}  
  
stock_prices['AAPL']
```

100

8.2.4 List Comprehension

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

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

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

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

```
even_numbers = []
for i in range(9):
    if i%2 == 0:
        even_numbers.append(i)

even_numbers
```

```
[0, 2, 4, 6, 8]
```

The list comprehension version of the code above is:

```
[i for i in range(9) if i%2 == 0]
```

```
[0, 2, 4, 6, 8]
```

List comprehensions are “very Pythonic” (the code style that other Python programmers expect).

8.3 Practice

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

```
a = 1

b = 2

c = a

a = b

b = c

del c
```

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

```
a is 2, b is 1
```

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

```
a = 1
```

```
b = 2
```

```
b, a = a, b
```

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

```
a is 2, b is 1
```

The code above uses “f strings”! Here is the non-f-string equivalent:

```
print('a is ' + str(a) + ', b is ' + str(b))
```

```
a is 2, b is 1
```

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

The parentheses () around tuples are optional to create tuples, but avoid ambiguity!

```
_ = 1, 1, 1  
type(_)
```

```
tuple
```

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

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

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

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

If we want to remove ambiguity, we should add parentheses () to both sides.

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

```
True
```

8.3.4 Create a list 11 of integers from 1 to 100.

Jupyter prints the last line of code in a code cell automatically. In the next cell, I wrap 11 with `print()` so the list contents wrap around.

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

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50]
```

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

Name this list 12.

```
l1.index(50)
```

```
49
```

With zero-indexing, left edges are included and right edges are excluded. So, right - left is the number of elements in a sequence.

In the code below, the first [] slices 50 to 60, and the second [::-1] reverses.

```
l2 = l1[49:60][::-1]
```

```
l2
```

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

```
11[59:48:-1]
```

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

8.3.6 Create a list 13 of odd integers from 1 to 21.

```
13 = 11[0:21][::2]
```

```
13
```

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

```
11[0:21:2]
```

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

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

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

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

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

What is the advantage of the list comprehension? There is no advantage in this example, but the list comprehension lets us specify several conditions and variable step sizes.

8.3.7 Create a list 14 of the squares of integers from 1 to 100.

```
14 = [i**2 for i in range(1, 101)]
```

```
print(14)
```

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, 256, 289, 324, 361, 400, 441]
```

8.3.8 Create a list 15 that contains the squares of *odd* integers from 1 to 100.

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

print(15)
```

```
[1, 9, 25, 49, 81, 121, 169, 225, 289, 361, 441, 529, 625, 729, 841, 961, 1089, 1225, 1369, 1]
```

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

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

Almost all core Python methods modify the object in place. Here we use the `sorted()` function so we can see the output/results more easily.

```
sorted(strings, key=len)
```

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

How do we get the last letter in a string?

```
'Friday' [-1]
```

```
'y'
```

```
def get_last_letter(x):
    return x[-1]
```

```
get_last_letter('Friday')
```

```
'y'
```

```
sorted(strings, key=get_last_letter)
```

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

We can get the same result without the hassle of writing the `get_last_letter()` function! Enter, anonymous (or lambda) functions!

```
sorted(strings, key=lambda x: x[-1])
```

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

We can translate `lambda x: x[-1]` into the following:

```
def function_that_I_do_not_want_to_use_again(x):  
    return x[-1]
```

8.3.10 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](#).

```
def get_largest(x, k):  
    return sorted(x)[-k]
```



```
get_largest(x=[3,2,1,5,6,4], k=2)
```

5

```
get_largest(x=[3,2,3,1,2,4,5,5,6], k=4)
```

4

8.3.11 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](#).

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



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

[1, 2]

8.3.12 Test whether the given strings are palindromes.

Input: ["aba", "no"]

Output: [True, False]

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



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

[True, False]

8.3.13 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]
```

This example is one the rare where I use a loop counter.

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

    return rs
```

We can use a magic called `%precision` to print fewer decimal places!

```
%precision 4
```

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

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

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

Here is a solution that avoids a loop counter and combines a list comprehension `zip()` to loop over prices, dividends, and lagged prices. This solution may be a little more Pythonic, but I find it harder to follow than our loop-counter solution above.

```
[(p + d - p_lag) / p_lag for p, d, p_lag in zip(prices[1:], dividends[1:], prices[:-1])]
```

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

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

```
def returns_2(p, d):
    rs, dps, cgs = [], [], []
    for i in range(1, len(p)):
        r = (p[i] + d[i] - p[i-1]) / p[i-1]
        dp = d[i] / p[i-1]
        cg = r - dp
        rs.append(r)
        dps.append(dp)
        cgs.append(cg)

    return {'r':rs, 'dp':dps, 'cg':cgs}

returns_2(p=prices, d=dividends)
```

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

```
returns_2(p=prices, d=dividends) ['r']
```

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

8.3.15 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]

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

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

rescale(x)
```

[0.7500, 0.0000, 0.5000, 1.0000, 0.5000]

Now we can rescale any list!

9 McKinney Chapter 3 - Practice for Section 04

9.1 Announcements

1. Due Friday, 1/19, at 11:59 PM:
 1. Complete *Introduction to Python* course on DataCamp (and upload certificate to Canvas)
 2. Acknowledge academic integrity statement on Canvas
2. I will record and post the next lecture video on Thursday, 1/18, and the associate pre-class quiz is due before class on Tuesday, 1/23
3. Start joining groups on Canvas (Canvas > People > Team Projects); I removed joining groups as a scored assignment, but please prioritize joining groups

9.2 10-minute Recap

9.2.1 List

A list is a collection of items that are ordered and changeable. You can add, remove, or modify elements in a list. In Python, you can create an empty list using either [] or `list()`.

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

Python is zero-indexed!

```
my_list[1]
```

2

We can chain index operations!

```
my_list[-1][-1][-1]
```

3

```
my_list[-1][-1][-1] = 'Foobar'  
my_list  
  
[1, 2, 3, [1, 2, 3, [1, 2, 'Foobar']]])
```

9.2.2 Tuple

A tuple is similar to a list, but it is immutable, meaning that you cannot change its contents once it has been created. You can create a tuple using parentheses () or the `tuple()` function.

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

Tuples are immutable and cannot be changed!

```
# my_tuple[-1][-1][-1] = 'Foobar'  
# -----  
# TypeError Traceback (most recent call last)  
# Cell In[10], line 1  
# ----> 1 my_tuple[-1][-1][-1] = 'Foobar'  
  
# TypeError: 'tuple' object does not support item assignment  
  
my_tuple_2 = (1, 2, 3, (1, 2, 3, [1, 2, 3]))
```

We cannot change the tuple, but we can change the list *inside* the tuple.

```
my_tuple_2[-1][-1][-1] = 'Foobar'
```

```
my_tuple_2
```

```
(1, 2, 3, (1, 2, 3, [1, 2, 'Foobar']))
```

Here is a crazy-making example of when we might want to use tuples instead of lists to prevent modifying `my_list`.

```
my_list

[1, 2, 3, [1, 2, 3, [1, 2, 'Foobar']]]

def foobar():
    my_list[0] += 7

foobar()

my_list

[8, 2, 3, [1, 2, 3, [1, 2, 'Foobar']]]
```

9.2.3 Dictionary

A dictionary is a collection of key-value pairs that are unordered and changeable. You can add, remove, or modify elements in a dictionary. In Python, you can create an empty dictionary using either {} or the `dict()` function.

```
stock_prices = [1, 2, 3] # AAPL, MSFT, and TSLA

stock_prices = {'AAPL': 1, 'MSFT': 2, 'TSLA': 3}

stock_prices['AAPL']
```

1

9.2.4 List Comprehension

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

1. `expression` is the operation to be performed on each element of the iterable
2. `item` is the current element being processed
3. `iterable` is the list or other iterable object being iterated over

4. condition is an optional filter that only accepts items that evaluate to True.

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

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

```
even_numbers = []
for i in range(9):
    if i%2 == 0:
        even_numbers.append(i)

even_numbers
```

```
[0, 2, 4, 6, 8]
```

```
%%timeit
[i for i in range(9) if i%2 == 0]
```

```
405 ns ± 23.3 ns per loop (mean ± std. dev. of 7 runs, 1,000,000 loops each)
```

Here is another way to get even up to 8:

```
%%timeit
list(range(0, 9, 2))
```

```
152 ns ± 7.38 ns per loop (mean ± std. dev. of 7 runs, 10,000,000 loops each)
```

The first is more flexible and allows testing multiple conditions. The second is more streamlined. Above, we use the `%%timeit` magic to time our code, but, remember, premature optimization is the root of all evil!

9.3 Practice

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

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

a is 2, and b is 1

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

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

a is 2, and b is 1

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

```
_ = 1, 1, 1
```

```
type(_)
```

```
tuple
```

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

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

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

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

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

```
True
```

Parentheses around tuples are optional, but eliminate ambiguity! Also, always check your output!

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

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

```
print(l1) # we can use print to print a list with less white space
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100]
```

9.3.5 Slice 11 to create a list of integers from 60 to 50 (inclusive).

Name this list 12.

Below, there are two steps:

1. The [49:60] slices from 50 to 60
2. The [::-1] reverse the list

```
12 = 11[49:60] [::-1]
```

```
12
```

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

Most methods on core Python data structures modify the data structure “in place”.

```
12_alt = 11[49:60]  
12_alt.reverse()  
  
12_alt
```

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

9.3.6 Create a list 13 of odd integers from 1 to 21.

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

```
13
```

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

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

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

9.3.7 Create a list 14 of the squares of integers from 1 to 100.

```
14 = [i**2 for i in range(1, 101)]  
print(14)
```

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, 256, 289, 324, 361, 400, 441]
```

9.3.8 Create a list 15 that contains the squares of *odd* integers from 1 to 100.

```
15 = [i**2 for i in range(1, 101) if i%2 != 0]  
print(15)
```

```
[1, 9, 25, 49, 81, 121, 169, 225, 289, 361, 441, 529, 625, 729, 841, 961, 1089, 1225, 1369, 1521]
```

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

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

Most core Python methods modify their object *in place*. Here we will use `sorted()`, which is the function equivalent of the `.sort()` method. Using the function helps us focus on the output.

```
sorted(strings, key=len)
```

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

What if, we want sort by length, then by alphabetical order within length?

1. Sort by alphabetical order
2. Then, sort by length

```
sorted(sorted(strings), key=len)
```

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

Just like we get the last element in a list or tuple with `[-1]`, we get the last character in a string with `[-1]`!

```
'Friday!' [-1]
```

```
'''
```

How can I reverse a string?

```
'Friday!' [::-1]
```

```
'!yadirF'
```

```
def get_last_letter(x):  
    return x[-1]
```

```
get_last_letter('Friday!')
```

```
'''
```

```
sorted(strings, key=get_last_letter)
```

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

For cases when you would only use a function once, we can skip all the overhead of writing a function and use an anonymous function (or a lambda function).

```
sorted(strings, key=lambda x: x[-1])
```

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

```
sorted(strings, key=lambda x: x[1])
```

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

Here `lambda` is a keyword that indicates what follows in a nameless function that only exists in this specific context.

9.3.10 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](#).

```
def get_largest(x, k):
    return sorted(x)[-k]

get_largest(x=[3,2,1,5,6,4], k=2)
```

5

```
get_largest(x=[3,2,3,1,2,4,5,5,6], k=4)
```

4

9.3.11 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](#).

```

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

get_frequent(nums=[1,1,1,2,2,3], k=2)

```

[1, 2]

9.3.12 Test whether the given strings are palindromes.

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

```

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

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

```

[True, False]

9.3.13 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]

def returns(p, d):
    rs = []
    for i in range(1, len(p)):
        r = (p[i] - p[i-1] + d[i]) / p[i-1]
        rs.append(r)
    return rs

```

We can use the magic `%precision 4` to display floats to only 4 decimal places.

```
%precision 4
```

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

We have 2+ arguments, it is a good practice to specify argument names (or keywords).

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

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

Here is a solution that avoids a loop counter and combines a list comprehension `zip()` to loop over prices, dividends, and lagged prices. This solution may be a little more Pythonic, but I find it harder to follow than our loop-counter solution above.

```
[(p + d - p_lag) / p_lag for p, d, p_lag in zip(prices[1:], dividends[1:], prices[:-1])]
```

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

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

```
def returns_2(p, d):
    rs, dps, cgs = [], [], []
    for i in range(1, len(p)):
        r = (p[i] - p[i-1] + d[i]) / p[i-1]
        dp = d[i] / p[i-1]
        cg = r - dp
        rs.append(r)
        dps.append(dp)
        cgs.append(cg)
    return {'r':rs, 'dp':dps, 'cg':cgs}
```

```
returns_2(p=prices, d=dividends)
```

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

```
returns_2(p=prices, d=dividends) ['r']
```

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

9.3.15 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]

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

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

```
x_rescaled = rescale(x)
```

```
x_rescaled
```

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

10 McKinney Chapter 3 - Practice for Section 05

10.1 Announcements

1. Due Friday, 1/19, at 11:59 PM:
 1. Complete *Introduction to Python* course on DataCamp (and upload certificate to Canvas)
 2. Acknowledge academic integrity statement on Canvas
2. I will record and post the next lecture video on Thursday, 1/18, and the associate pre-class quiz is due before class on Tuesday, 1/23
3. Start joining groups on Canvas (Canvas > People > Team Projects); I removed joining groups as a scored assignment, but please prioritize joining groups

10.2 10-minute Recap

10.2.1 List

A list is a collection of items that are ordered and changeable. You can add, remove, or modify elements in a list. In Python, you can create an empty list using either [] or `list()`.

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

```
my_list
```

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

Python is zero-indexed!

```
my_list[1]
```

2

We can chain indexes!

```
my_list[-1][-1][-1]
```

3

```
my_list[-1][-1][-1] = 2_001
```

```
my_list
```

```
[1, 2, 3, [1, 2, 3, [1, 2, 2001]]]
```

10.2.2 Tuple

A tuple is similar to a list, but it is immutable, meaning that you cannot change its contents once it has been created. You can create a tuple using parentheses () or the `tuple()` function.

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

```
my_tuple
```

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

```
my_tuple[-1][-1][-1]
```

3

Tuples are immutable, so they cannot be changed!

```
# my_tuple[-1][-1][-1] = 2_001
# -----
# TypeError                                     Traceback (most recent call last)
# Cell In[12], line 1
```

```
# ----> 1 my_tuple[-1] [-1] [-1] = 2_001  
# TypeError: 'tuple' object does not support item assignment
```

10.2.3 Dictionary

A dictionary is a collection of key-value pairs that are unordered and changeable. You can add, remove, or modify elements in a dictionary. In Python, you can create an empty dictionary using either {} or the dict() function.

```
my_dict = {'A': 1, 'B': 2, 3: 2_001, 'D': [1, 2, 3]}  
  
my_dict  
  
{'A': 1, 'B': 2, 3: 2001, 'D': [1, 2, 3]}
```

```
my_dict['A']
```

```
1
```

```
my_dict['E'] = [4, 5, 6]  
  
my_dict  
  
{'A': 1, 'B': 2, 3: 2001, 'D': [1, 2, 3], 'E': [4, 5, 6]}
```

10.2.4 List Comprehension

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

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

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

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

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

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

```
[3, 6, 9, 12, 15, 18, 21, 24]
```

The list comprehension above is a simplification of:

```
empty_list = []
for i in range(1, 26):
    if i%3 == 0:
        empty_list.append(i)

empty_list
```

```
[3, 6, 9, 12, 15, 18, 21, 24]
```

10.3 Practice

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

```
a = 1

b = 2

c = a

a = b

b = c

del c
```

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

```
Variable a is 2, and variable b is 1
```

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

```
a = 1
```

```
b = 2
```

```
b, a = a, b
```

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

```
Variable a is 2, and variable b is 1
```

The parentheses () around tuples are optional, but often very helpful! The commas , make the tuple, not the parentheses.

```
a = 1
```

```
b = 2
```

```
(b, a) = (a, b)
```

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

```
Variable a is 2, and variable b is 1
```

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

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

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

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

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

True

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

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

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

10.3.4 Create a list 11 of integers from 1 to 100.

```
11 = list(range(1, 101))
# the last entry in a code cell automatically prints
# here we use print() to wrap the list to take up less space
print(11)
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60]
```

10.3.5 Slice 11 to create a list of integers from 60 to 50 (inclusive).

Name this list 12.

```
11.index(50)
```

49

```
11[49:60]
```

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

In the solution above, we have to “brute force” picking the left edge at 49. However these are 11 values in 50 to 60 inclusive (i.e., $60 - 50 + 1$), so we can add 11 to 49 to find the right edge. This trick is one of the benefits of Python’s zero-indexing.

How do we reverse? I prefer `[::-1]`.

```
12 = 11[49:60][::-1] # the first [] slices, and the second [] reverses
12
[60, 59, 58, 57, 56, 55, 54, 53, 52, 51, 50]

12_alt = 11[49:60]
12_alt.reverse() # note, most core python operators modify the object "in place"
12_alt
[60, 59, 58, 57, 56, 55, 54, 53, 52, 51, 50]
```

10.3.6 Create a list 13 of odd integers from 1 to 21.

```
13 = 11[0:21][::2]
13
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21]

11[0:21:2]
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21]

list(range(1, 22, 2))
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21]

[i for i in range(22) if i%2 != 0]
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21]
```

What is the advantage of the list comprehension? There is no advantage in this example, but the list comprehension lets us specify several conditions and variable step sizes.

10.3.7 Create a list 14 of the squares of integers from 1 to 100.

```
14 = [i**2 for i in range(1, 101)]  
print(14)
```

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, 256, 289, 324, 361, 400, 441]
```

10.3.8 Create a list 15 that contains the squares of *odd* integers from 1 to 100.

```
15 = [i**2 for i in range(1, 101) if i%2 != 0]  
print(15)
```

```
[1, 9, 25, 49, 81, 121, 169, 225, 289, 361, 441, 529, 625, 729, 841, 961, 1089, 1225, 1369, ...]
```

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

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

Most core Python methods modify their objects *in place*. For clarity, I will use the `sorted()` function, which *does not modify its argument in place*.

```
sorted(strings)  
  
['aaaa', 'abab', 'bar', 'card', 'foo']
```

We can pass a function name to the `key=` argument. Python applies this function to every element in the list, then uses the function output to sort the list.

```
sorted(strings, key=len)  
  
['foo', 'bar', 'card', 'aaaa', 'abab']  
  
sorted(strings, key=len, reverse=True)  
  
['card', 'aaaa', 'abab', 'foo', 'bar']
```

We can index and slice strings just like we do lists!

```

'string'[-1]

'g'

'string'[-2:]

'ng'

def get_last_letter(x):
    return x[-1]

get_last_letter('string')

'g'

sorted(strings, key=get_last_letter)

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

```

Anonymous functions (also called lambda functions) let us skip the function writing and naming steps!

```

sorted(strings, key=lambda x: x[-1])

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

```

10.3.10 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](#).

```
def get_largest(x, k):
    return sorted(x)[-k]

get_largest(x=[3,2,1,5,6,4], k=2)

5

get_largest(x=[3,2,3,1,2,4,5,5,6], k=4)

4
```

10.3.11 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](#).

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

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

```
[1, 2]
```

10.3.12 Test whether the given strings are palindromes.

Input: ["aba", "no"]

Output: [True, False]

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

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

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

```
[True, False]
```

10.3.13 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]
```

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

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

    return rs

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

```
[0.51, -0.3266666666666666, -0.49, 1.04, 0.52, -0.32, 0.52]
```

Here is a solution that avoids a loop counter and combines a list comprehension `zip()` to loop over prices, dividends, and lagged prices. This solution may be a little more Pythonic, but I find it harder to follow than our loop-counter solution above.

```
[(p + d - p_lag) / p_lag for p, d, p_lag in zip(prices[1:], dividends[1:], prices[:-1])]
```

```
[0.51, -0.3266666666666666, -0.49, 1.04, 0.52, -0.32, 0.52]
```

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

```
def returns_2(p, d):
    rs, d_ps, cgs = [], [], []
    for i in range(1, len(p)):
        r = (p[i] + d[i] - p[i-1]) / p[i-1]
        d_p = d[i] / p[i-1]
        cg = r - d_p
        rs.append(r)
        d_ps.append(d_p)
        cgs.append(cg)

    return {'r':rs, 'd_p':d_ps, 'cg':cgs}
```

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

```
%precision 4
```

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

```
returns_2(p=prices, d=dividends)
```

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

By returning a dictionary instead of tuple, we can index returns, dividend yields, and capital gains yields by name instead of position.

```
returns_2(p=prices, d=dividends) ['r']
```

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

10.3.15 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]

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

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

```
rescale(x)
```

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

Part III

Week 3

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

11.1 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 discusses 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]
```

```
61.2 ms ± 2.09 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

```
%timeit my_arr * 2
```

```
1.18 ms ± 47.9 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!

11.2 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 lists 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
```

11.2.1 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 because 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)

11.2.2 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.]])
```

11.2.3 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[:] = 2_001  
arr_slice_2
```

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

```
arr
```

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

11.2.4 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!*

11.2.5 Boolean Indexing

We can use Booleans (Trues and Falses) to slice arrays, too. Boolean indexing in Python is like combining `index()` and `match()` in Excel.

```
names = np.array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'])
np.random.seed(42)
data = np.random.randn(7, 4)

names

array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'], dtype='<U4')

data

array([[ 0.4967, -0.1383,  0.6477,  1.523 ],
       [-0.2342, -0.2341,  1.5792,  0.7674],
       [-0.4695,  0.5426, -0.4634, -0.4657],
       [ 0.242 , -1.9133, -1.7249, -0.5623],
       [-1.0128,  0.3142, -0.908 , -1.4123],
       [ 1.4656, -0.2258,  0.0675, -1.4247],
       [-0.5444,  0.1109, -1.151 ,  0.3757]])
```

Here `names` provides seven names for the seven rows in `data`.

```
names == 'Bob'

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

data[names == 'Bob']

array([[ 0.4967, -0.1383,  0.6477,  1.523 ],
       [ 0.242 , -1.9133, -1.7249, -0.5623]])
```

We can combine Boolean slicing with : slicing.

```
data[names == 'Bob', 2:]

array([[ 0.6477,  1.523 ],
       [-1.7249, -0.5623]])
```

We can use `~` to invert a Boolean.

```
cond = names == 'Bob'  
data[~cond]  
  
array([[-0.2342, -0.2341,  1.5792,  0.7674],  
       [-0.4695,  0.5426, -0.4634, -0.4657],  
       [-1.0128,  0.3142, -0.908 , -1.4123],  
       [ 1.4656, -0.2258,  0.0675, -1.4247],  
       [-0.5444,  0.1109, -1.151 ,  0.3757]])
```

For NumPy arrays, we must use `&` and `|` instead of `and` and `or`.

```
cond = (names == 'Bob') | (names == 'Will')  
data[cond]  
  
array([[ 0.4967, -0.1383,  0.6477,  1.523 ],  
       [-0.4695,  0.5426, -0.4634, -0.4657],  
       [ 0.242 , -1.9133, -1.7249, -0.5623],  
       [-1.0128,  0.3142, -0.908 , -1.4123]])
```

We can also create a Boolean for each element.

```
data  
  
array([[ 0.4967, -0.1383,  0.6477,  1.523 ],  
       [-0.2342, -0.2341,  1.5792,  0.7674],  
       [-0.4695,  0.5426, -0.4634, -0.4657],  
       [ 0.242 , -1.9133, -1.7249, -0.5623],  
       [-1.0128,  0.3142, -0.908 , -1.4123],  
       [ 1.4656, -0.2258,  0.0675, -1.4247],  
       [-0.5444,  0.1109, -1.151 ,  0.3757]])  
  
data < 0  
  
array([[False,  True, False, False],  
       [ True,  True, False, False],  
       [ True, False,  True,  True],  
       [False,  True,  True,  True],  
       [ True, False,  True,  True],  
       [False,  True, False,  True],  
       [ True, False,  True, False]])
```

```
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]])
```

11.3 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(4)

2.0000

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)
```

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

- **abs, fabs:** Compute the absolute value element-wise for integer, floating-point, or complex values
- **sqrt:** Compute the square root of each element (equivalent to `arr ** 0.5`)
- **square:** Compute the square of each element (equivalent to `arr ** 2`)
- **exp:** Compute the exponent e^x of each element
- **log, log10, log2, log1p:** Natural logarithm (base e), log base 10, log base 2, and $\log(1 + x)$, respectively
- **sign:** Compute the sign of each element: 1 (positive), 0 (zero), or -1 (negative)
- **ceil:** Compute the ceiling of each element (i.e., the smallest integer greater than or equal to `thatnumber`)
- **floor:** Compute the floor of each element (i.e., the largest integer less than or equal to each element)
- **rint:** Round elements to the nearest integer, preserving the `dtype`
- **modf:** Return fractional and integral parts of array as a separate array
- **isnan:** Return boolean array indicating whether each value is NaN (Not a Number)
- **isfinite, isinf:** Return boolean array indicating whether each element is finite (non-inf, non-NaN) or infinite, respectively
- **cos, cosh, sin, sinh, tan, tanh:** Regular and hyperbolic trigonometric functions
- **arccos, arccosh, arcsin, arcsinh, arctan, arctanh:** Inverse trigonometric functions
- **logical_not:** Compute truth value of not `x` element-wise (equivalent to `~arr`).

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 `&`, `|`, `^`)

11.4 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.

11.4.1 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 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])
```

11.4.2 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]) # same output as np.arange(8)
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

11.4.3 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,  True,  True, False,
       True, False, False,  True, False, False, False,  True,
      False, False,  True, False,  True, False, False,  True, False,
      True, False, False,  True, False,  True, False, False,  True,
      True, False, False,  True,  True,  True, False, False, False,
      False, False,  True,  True, False,  True, False, False,  True,
      True,  True, False,  True,  True,  True, False, False, False,
      False,  True,  True, False,  True, False,  True,  True,  True,
      False,  True, False,  True,  True, False,  True, False, False,
      True,  True, False, False,  True,  True, False,  True,  True,
      True,  True, False, False, False,  True,  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

12 McKinney Chapter 4 - Practice for Section 02

12.1 Announcements

1. Our second DataCamp course, *Intermediate Python*, is due Friday, 1/26, at 11:59 PM
2. I will record our week 4 lecture video on McKinney chapter 5 this Thursday evening, and the week 4 pre-class quiz is due before class next Tuesday, 1/30
3. Team projects
 1. Continue to join teams on Canvas > People > Team Projects
 2. I removed the join-a-team assignment, but I will give the first project assignment in early February, so join a team by then

12.2 10-minute Recap

12.2.1 NumPy Arrays

NumPy arrays are multidimensional data structures that can store numerical data efficiently and perform fast mathematical operations on them. The `%precision` magic displays floats (including in arrays) to 4 digits.

NumPy arrays are multidimensional data structures that can store numerical data efficiently and perform fast mathematical operations on them.

```
import this
```

The Zen of Python, by Tim Peters

```
Beautiful is better than ugly.  
Explicit is better than implicit.  
Simple is better than complex.  
Complex is better than complicated.  
Flat is better than nested.
```

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

```
import numpy as np
%precision 4

'%.4f'

np.arange(10)

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

np.ones(10)

array([1., 1., 1., 1., 1., 1., 1., 1., 1., 1.])

np.ones((2, 2)) # 2 rows and 2 columns

array([[1., 1.],
       [1., 1.]]) 

np.ones((2, 2, 2, 2)) # 2 rows, 2 columns, 2 stacks, 2 times
```

```
array([[[[1., 1.],  
       [1., 1.]],  
  
      [[[1., 1.],  
       [1., 1.]],  
  
      [[1., 1.],  
       [1., 1.]]])
```

The `np.random.rand()` function creates standard normal random variables (i.e., mean of 0 and standard deviation of 1). We generally use the `np.random.seed()` function to make our random numbers repeatable.

```
np.random.seed(42) # from the *Hitch-Hiker's Guide to the Galaxy*  
np.random.randn(2, 2)  
  
array([[ 0.4967, -0.1383],  
       [ 0.6477,  1.523 ]])
```

12.2.2 Vectorized Functions

Vectorized computation is the process of applying an operation to an entire array or a subset of an array without using explicit loops. NumPy supports vectorized computation using universal functions (ufuncs), which are functions that operate on arrays element-wise.

```
4**0.5  
  
2.0000
```

```
[i**0.5 for i in range(10)]  
  
[0.0000,  
 1.0000,  
 1.4142,  
 1.7321,
```

```
2.0000,
2.2361,
2.4495,
2.6458,
2.8284,
3.0000]

np.sqrt(np.arange(10))

array([0.      , 1.      , 1.4142, 1.7321, 2.      , 2.2361, 2.4495, 2.6458,
       2.8284, 3.      ])
```

12.2.3 Indexing and Slicing

Indexing and slicing are techniques to access or modify specific elements or subsets of an array. NumPy also supports advanced indexing methods, such as fancy indexing and boolean indexing, which allow more flexible and complex selection of array elements.

```
np.random.seed(42)
my_array = np.random.randn(3, 3)

my_array

array([[ 0.4967, -0.1383,  0.6477],
       [ 1.523 , -0.2342, -0.2341],
       [ 1.5792,  0.7674, -0.4695]])
```

How can we get the first row in `my_array`?

```
my_array[0]

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

How can we get the 0.4967? Chain the [0] indexes.

```
my_array[0][0]

0.4967
```

This indexing is common enough, the we can replace [0] [0] with [0, 0], which is i,j notation.

```
my_array[0, 0]
```

```
0.4967
```

What if we want the first two columns of the first two rows?

```
my_array[:, :2]
```

```
array([[ 0.4967, -0.1383],
       [ 1.523 , -0.2342]])
```

12.3 Practice

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

```
a1 = np.arange(25)
```

```
a1
```

```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
       17, 18, 19, 20, 21, 22, 23, 24])
```

Here, `np.arange()` replaces `np.array()`, `list()`, and `range()`!

```
np.array(list(range(25)))
```

```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
       17, 18, 19, 20, 21, 22, 23, 24])
```

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

```
a2 = np.arange(0, 25, 3) # start, stop, step size  
a2  
  
array([ 0,  3,  6,  9, 12, 15, 18, 21, 24])
```

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

```
a3 = np.array([i for i in range(101) if (i%3==0) or (i%5==0)]) # here we could replace "or"  
a3  
  
array([ 0,  3,  5,  6,  9, 10, 12, 15, 18, 20, 21, 24, 25,  
       27, 30, 33, 35, 36, 39, 40, 42, 45, 48, 50, 51, 54,  
       55, 57, 60, 63, 65, 66, 69, 70, 72, 75, 78, 80, 81,  
       84, 85, 87, 90, 93, 95, 96, 99, 100])
```

We can also use Booleans to slice the output of `np.arange(101)`.

```
a3_alt = np.arange(101)  
a3_alt = a3_alt[ (a3_alt%3==0) | (a3_alt%5==0) ]  
  
a3_alt  
  
array([ 0,  3,  5,  6,  9, 10, 12, 15, 18, 20, 21, 24, 25,  
       27, 30, 33, 35, 36, 39, 40, 42, 45, 48, 50, 51, 54,  
       55, 57, 60, 63, 65, 66, 69, 70, 72, 75, 78, 80, 81,  
       84, 85, 87, 90, 93, 95, 96, 99, 100])  
  
(a3 == a3_alt).all()
```

True

The `np.allclose()` functions helps us test equality with some non-zero tolerance. More later in the class!

```
np.allclose(a3, a3_alt)

True

(a3 == 1.000001*a3_alt).all()

False

np.allclose(a3, 1.000001*a3_alt)
```

True

12.3.4 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?

```
np.arange(0, 100_001, 2)**2

array([ 0,  4, 16, ..., 1409265424, 1409665412,
       1410065408])
```

On some computers, the output above is wrong because NumPy defaults to 32-bit integers, depending on the computer! *Always check your output!* To avoid this problem, we can force `np.arange()` to use 64-bit integers with the `dtype=` argument.

```
np.arange(0, 100_001, 2, dtype=np.int64)**2

array([ 0,  4, 16, ..., 9999200016,
       9999600004, 10000000000], dtype=int64)
```

We can use the `%timeit` magic to time which code is faster! The `%timeit` magic runs the code on the same line many times and reports the mean computation time. The `%%timet` magic with two percent signs runs the code in the same cell many times and reports the mean computation time.

```
%timeit np.arange(0, 100_001, 2, dtype=np.int64)**2
```

32.5 μ s \pm 3.41 μ s per loop (mean \pm std. dev. of 7 runs, 10,000 loops each)

```
%timeit np.array([i**2 for i in range(0, 100_001)])
```

13.1 ms \pm 700 μ s per loop (mean \pm std. dev. of 7 runs, 100 loops each)

The NumPy version is about 1,000 times faster!

12.3.5 Write a function that mimic Excel's pv function.

Here is how we call Excel's pv function: =PV(rate, nper, pmt, [fv], [type]) We can use the annuity and lump sum present value formulas.

Present value of an annuity payment pmt: $PV_{pmt} = \frac{pmt}{rate} \times \left(1 - \frac{1}{(1+rate)^{nper}}\right)$

Present value of a lump sum fv: $PV_{fv} = \frac{fv}{(1+rate)^{nper}}$

```
def calc_pv(rate, nper, pmt=None, fv=None, type=None):
    if pmt is None:
        pmt = 0
    if fv is None:
        fv = 0
    if type is None:
        type = 'END'

    pv_pmt = (pmt / rate) * (1 - 1 / (1 + rate)**nper)
    pv_fv = fv / (1 + rate)**nper
    pv = pv_pmt + pv_fv

    if type == 'BGN':
        pv *= (1 + rate) # same as pv = pv*(1 + rate)

    return -1 * pv

calc_pv(rate=0.05, nper=14, pmt=50, fv=1_000, type='END')
```

-1000.0000

```
calc_pv(rate=0.05, nper=14, fv=1_000, type='END')
```

-505.0680

```
a = 5
a*= 5
a*= 5
a
```

125

12.3.6 Write a function that mimic Excel's fv function.

```
def calc_fv(rate, nper, pmt=None, pv=None, type=None):
    if pmt is None:
        pmt = 0
    if pv is None:
        pv = 0
    if type is None:
        type = 'END'

    fv_pmt = (pmt / rate) * ((1 + rate)**nper - 1)
    fv_pv = pv * (1 + rate)**nper
    fv = fv_pmt + fv_pv

    if type == 'BGN':
        fv *= (1 + rate) # same as pv = pv*(1 + rate)

    return -1 * fv
```

```
calc_fv(rate=0.072, nper=10, pv=-1000)
```

2004.2314

```
calc_fv(rate=0.072, nper=10, pmt=72, pv=-1000)
```

1000.0000

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

```
np.random.seed(42)
data = np.random.randn(3, 3)
data

array([[ 0.4967, -0.1383,  0.6477],
       [ 1.523 , -0.2342, -0.2341],
       [ 1.5792,  0.7674, -0.4695]])

data[data < 0] = -1
data[data > 0] = +1
data

array([[ 1., -1.,  1.],
       [ 1., -1., -1.],
       [ 1.,  1., -1.]])
```

Here is a second option! NumPy's `np.where()` function has the same logic as Excel's `if()` function! I will recreate `data` so we start from the same point.

```
np.random.seed(42)
data = np.random.randn(3, 3)
data_where = np.where(data < 0, -1, np.where(data > 0, +1, 0))
data_where

array([[ 1, -1,  1],
       [ 1, -1, -1],
       [ 1,  1, -1]])
```

Here is a third option! NumPy's `np.select()` function lets us test *many* conditions! I will recreate `data` so we start from the same point.

```
np.random.seed(42)
data = np.random.randn(3, 3)
data_select = np.select(
    condlist=[data<0, data>0],
    choicelist=[-1, +1],
    default=0
```

```

)
data_select

array([[ 1, -1,  1],
       [ 1, -1, -1],
       [ 1,  1, -1]])

(data_where == data_select).all()

```

True

Here is a more complex application of `np.select()`. Say, we want to truncate values to be between -0.5 and $+0.5$.

```

np.random.seed(42)
data = np.random.randn(3, 3)
data

array([[ 0.4967, -0.1383,  0.6477],
       [ 1.523 , -0.2342, -0.2341],
       [ 1.5792,  0.7674, -0.4695]])

np.select(
    condlist=[data<-0.5, data>+0.5],
    choicelist=[-0.5, +0.5],
    default=data
)

array([[ 0.4967, -0.1383,  0.5    ],
       [ 0.5    , -0.2342, -0.2341],
       [ 0.5    ,  0.5    , -0.4695]])

```

12.3.8 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 c_1 , r , and g that represent C_1 , r , and g . The present value of a growing perpetuity is $PV = \frac{C_1}{r-g}$, and the present value of a growing annuity is $PV = \frac{C_1}{r-g} \left[1 - \left(\frac{1+g}{1+r} \right)^t \right]$.

We can use the growing annuity and perpetuity formulas to show: $x = \left[1 - \left(\frac{1+g}{1+r}\right)^t\right]$.

Then: $1 - x = \left(\frac{1+g}{1+r}\right)^t$.

Finally: $t = \frac{\log(1-x)}{\log\left(\frac{1+g}{1+r}\right)}$

We do not need to accept an argument c_1 because C_1 cancels out!

```
def npmts(x, r, g):
    return np.log(1-x) / np.log((1 + g) / (1 + r))
```

```
npmts(0.5, 0.1, 0.05)
```

14.9000

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

Here are some data where the *IRR* is obvious!

```
c = np.array([-100, +110])
r = 0.1
```

First, write a function that calculates net present value (NPV) given cash flows in a NumPy array c and a discount rate in a scalar r . The `npv()` function below uses NumPy arrays to calculate NPV as:

$$NPV = \sum_{t=0}^T \frac{c_t}{(1+r)^t}$$

```
def calc_npv(r, c):
    t = np.arange(len(c))
    return (c / (1 + r)**t).sum()

calc_npv(r=r, c=c)
```

-0.0000

We can use a `while` loop to guess IRR values until we find an NPV close to zero. We can use the [Newton-Rapshon method](#) to make smarter guesses. If we have function $f(x)$ and guess x_t , our next guess should be $x_{t+1} = x_t - \frac{f(x_t)}{f'(x_t)}$. Here our $f(x)$ is $NPV(r)$, and we can approximate $f'(x_t)$ as $\frac{NPV(r+0.000001)-NPV(r)}{0.000001}$. We will make guess until $|NPV| < 0.000001$.

```
def calc_irr(c, guess=0, tol=1e-6, step=1e-6):
    irr = guess
    npv = calc_npv(r=irr, c=c) # I made this change after class to possibly save us an ite
    while np.abs(npv) > tol:
        npv = calc_npv(r=irr, c=c)
        deriv = (calc_npv(r=irr+step, c=c) - npv) / step
        irr = irr - npv / deriv
        # print(f'IRR is {irr}, and NPV is {npv}')
    return irr

calc_irr(c)
```

0.1000

12.3.10 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])
```

We want to slice our arrays to “lag” or “shift” them! For example, we slice the `prices` array to calculate capital gains as follows.

```
prices[1:] - prices[:-1]

array([ 50, -50, -50,  50,  50, -50,  50])

def returns(p, d):
    return (p[1:] - p[:-1] + d[1:]) / p[:-1]

returns(p=prices, d=dividends)

array([ 0.51 , -0.3267, -0.49 ,  1.04 ,  0.52 , -0.32 ,  0.52 ])
```

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

```
def returns_2(p, d):
    r = (p[1:] - p[:-1] + d[1:]) / p[:-1]
    dp = d[1:] / p[:-1]
    cgp = r - dp
    return {'r':r, 'dp':dp, 'cgp':cgp}

returns_2(p=prices, d=dividends)

{'r': array([ 0.51 , -0.3267, -0.49 ,  1.04 ,  0.52 , -0.32 ,  0.52 ]),
 'dp': array([0.01 ,  0.0067,  0.01 ,  0.04 ,  0.02 ,  0.0133,  0.02 ]),
 'cgp': array([ 0.5   , -0.3333, -0.5   ,  1.     ,  0.5   , -0.3333,  0.5   ])}
```

12.3.12 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])

(numbers - numbers.min()) / (numbers.max() - numbers.min())

array([0.75, 0.  , 0.5 , 1.  , 0.5 ])
```

12.3.13 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 `numbers` to compare your functions with NumPy's `.var()` and `.std()` methods.

```
((numbers - numbers.mean())**2).mean()
```

```
0.4400
```

```
numbers.var()
```

```
0.4400
```

```
def var(x, sample=True):
    mu_x = x.mean()
    return ((x - mu_x)**2).sum() / (len(x) - sample)
```

```
var(numbers)
```

```
0.5500
```

```
var(numbers) == numbers.var(ddof=1)
```

```
True
```

```
var(numbers, sample=False)
```

```
0.4400
```

```
var(numbers, sample=False) == numbers.var()
```

```
True
```

```
def std(x, sample=True):
    return np.sqrt(var(x=x, sample=sample))
```

```
std(numbers, sample=False)
```

```
0.6633
```

```
    std(numbers, sample=False) == numbers.std()
```

```
True
```

```
    std(numbers)
```

```
0.7416
```

```
    std(numbers) == numbers.std(ddof=1)
```

```
True
```

13 McKinney Chapter 4 - Practice for Section 03

13.1 Announcements

1. Our second DataCamp course, *Intermediate Python*, is due Friday, 1/26, at 11:59 PM
2. I will record our week 4 lecture video on McKinney chapter 5 this Thursday evening, and the week 4 pre-class quiz is due before class next Tuesday, 1/30
3. Team projects
 1. Continue to join teams on Canvas > People > Team Projects
 2. I removed the join-a-team assignment, but I will give the first project assignment in early February, so join a team by then

13.2 10-minute Recap

13.2.1 NumPy Arrays

NumPy arrays are multidimensional data structures that can store numerical data efficiently and perform fast mathematical operations on them.

```
import this
```

The Zen of Python, by Tim Peters

```
Beautiful is better than ugly.  
Explicit is better than implicit.  
Simple is better than complex.  
Complex is better than complicated.  
Flat is better than nested.  
Sparse is better than dense.  
Readability counts.  
Special cases aren't special enough to break the rules.  
Although practicality beats purity.
```

Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!

```
import numpy as np
%precision 4

'%.4f'

np.arange(10)

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

np.ones((2, 2))

array([[1., 1.],
       [1., 1.]]) 

np.ones((2, 2, 2, 2))

array([[[[1., 1.],
          [1., 1.]],

         [[1., 1.],
          [1., 1.]]],

        [[[1., 1.],
          [1., 1.]],

         [[1., 1.]]]])
```

We can use `np.random.rand()` to create n-dimensional arrays of standard normal random variables (i.e., $\mu = 0, \sigma = 1$). We can use `np.random.seed()` to set the random number generator seed to make our analysis repeatable.

```
np.random.seed(42)
np.random.randn(2, 2)
```

```
array([[ 0.4967, -0.1383],
       [ 0.6477,  1.523 ]])
```

13.2.2 Vectorized Functions

Vectorized computation is the process of applying an operation to an entire array or a subset of an array without using explicit loops. NumPy supports vectorized computation using universal functions (ufuncs), which are functions that operate on arrays element-wise.

Say we want to calculate square roots.

```
4**0.5
```

```
2.0000
```

```
[i**0.5 for i in range(10)]
```

```
[0.0000,
 1.0000,
 1.4142,
 1.7321,
 2.0000,
 2.2361,
 2.4495,
 2.6458,
 2.8284,
 3.0000]
```

```
np.sqrt(4)
```

```
2.0000
```

```
np.sqrt(np.arange(10))

array([0.      , 1.      , 1.4142, 1.7321, 2.      , 2.2361, 2.4495, 2.6458,
       2.8284, 3.      ])
```

13.2.3 Indexing and Slicing

Indexing and slicing are techniques to access or modify specific elements or subsets of an array. NumPy also supports advanced indexing methods, such as fancy indexing and boolean indexing, which allow more flexible and complex selection of array elements.

```
np.random.seed(42)
my_array = np.random.randn(3, 3)

my_array

array([[ 0.4967, -0.1383,  0.6477],
       [ 1.523 , -0.2342, -0.2341],
       [ 1.5792,  0.7674, -0.4695]])

my_array[0] # indexes first array in array of arrays

array([ 0.4967, -0.1383,  0.6477])

my_array[0][0] # indexes first element in first array in array of arrays
```

```
0.4967
```

In NumPy, we typically replace the chained `[i][j]` notation with the `[i, j]` from linear algebra class.

```
my_array[0, 0]
```

```
0.4967
```

```
my_array[:, :2]  
  
array([[ 0.4967, -0.1383],  
       [ 1.523 , -0.2342]])
```

13.3 Practice

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

```
a1 = np.arange(25)  
  
a1  
  
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,  
       17, 18, 19, 20, 21, 22, 23, 24])  
  
a1.ndim  
  
1  
  
a1.dtype  
  
dtype('int32')
```

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

```
a2 = np.arange(0, 25, 3)  
  
a2  
  
array([ 0,  3,  6,  9, 12, 15, 18, 21, 24])
```

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

```
a3 = np.array([i for i in range(101) if (i%3==0) or (i%5==0)]) # core Python allows "or" or
```

```
a3
```

```
array([ 0,  3,  5,  6,  9, 10, 12, 15, 18, 20, 21, 24, 25,
       27, 30, 33, 35, 36, 39, 40, 42, 45, 48, 50, 51, 54,
       55, 57, 60, 63, 65, 66, 69, 70, 72, 75, 78, 80, 81,
       84, 85, 87, 90, 93, 95, 96, 99, 100])
```

```
a3.ndim
```

```
1
```

```
a3_alt = np.arange(101)
a3_alt = a3_alt[(a3_alt%3==0) | (a3_alt%5==0)] # NumPy does not allow "or", only "|"
```

```
a3_alt
```

```
array([ 0,  3,  5,  6,  9, 10, 12, 15, 18, 20, 21, 24, 25,
       27, 30, 33, 35, 36, 39, 40, 42, 45, 48, 50, 51, 54,
       55, 57, 60, 63, 65, 66, 69, 70, 72, 75, 78, 80, 81,
       84, 85, 87, 90, 93, 95, 96, 99, 100])
```

How can we test if `a3` and `a3_alt` have the same values?

The `==` operators tests if NumPy arrays are the same, element by element. We can append the `.all()` to make sure all element-by-element comparisons are `True`!

```
(a3 == a3_alt).all()
```

```
True
```

if we want to compare within some tolerance, we can use `np.allclose()`.

```
np.allclose(a3, a3_alt)
```

```
True
```

Minor detour for another example! How do I slice values greater than 5 in an array from 0 to 9?

```
ten = np.arange(10)
gt_five = ten[ten > 5]

gt_five
```

```
array([6, 7, 8, 9])
```

13.3.4 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?

```
np.arange(0, 100_001, 2)**2

array([ 0, 4, 16, ..., 1409265424, 1409665412,
       1410065408])
```

On some computers, the output above is wrong because NumPy defaults to 32-bit integers, depending on the computer! *Always check your output!* To avoid this problem, we can force `np.arange()` to use 64-bit integers with the `dtype=` argument.

```
np.arange(0, 100_001, 2, dtype=np.int64)**2

array([ 0, 4, 16, ..., 9999200016,
       9999600004, 10000000000], dtype=int64)
```

We can use the `%timeit` magic to time which code is faster! The `%timeit` magic runs the code on the same line many times and reports the mean computation time. The `%%timet` magic with two percent signs runs the code in the same cell many times and reports the mean computation time.

```
%timeit np.arange(0, 100_001, 2, dtype=np.int64)**2
```

43.8 μ s \pm 5.15 μ s per loop (mean \pm std. dev. of 7 runs, 10,000 loops each)

```
%timeit np.array([i**2 for i in range(0, 100_001)])
```

13.4 ms \pm 918 μ s per loop (mean \pm std. dev. of 7 runs, 100 loops each)

The NumPy version is about 1,000 times faster!

13.3.5 Write a function that mimic Excel's pv function.

Here is how we call Excel's pv function: =PV(rate, nper, pmt, [fv], [type]) We can use the annuity and lump sum present value formulas.

Present value of an annuity payment pmt: $PV_{pmt} = \frac{pmt}{rate} \times \left(1 - \frac{1}{(1+rate)^{nper}}\right)$

Present value of a lump sum fv: $PV_{fv} = \frac{fv}{(1+rate)^{nper}}$

```
def pv(rate, nper, pmt=None, fv=None, type=None):
    if pmt is None:
        pmt = 0
    if fv is None:
        fv = 0
    if type is None:
        type = 'END'

    pv_pmt = (pmt / rate) * (1 - 1 / (1 + rate)**nper)
    pv_fv = fv / (1 + rate)**nper
    pv = pv_pmt + pv_fv

    if type == 'BGN':
        pv *= (1 + rate) # same as pv = pv*(1 + rate)

    return -1 * pv

pv(rate = 0.05, nper = 14, fv = 1_000, type = 'END')
```

-505.0680

```
a = 5
a*= 5
a*= 5
a
```

125

13.3.6 Write a function that mimic Excel's fv function.

```
def calc_fv(rate, nper, pmt=None, pv=None, type=None):
    if pmt is None:
        pmt = 0
    if pv is None:
        pv = 0
    if type is None:
        type = 'END'

    fv_pmt = (pmt / rate) * ((1 + rate)**nper - 1)
    fv_pv = pv * (1 + rate)**nper
    fv = fv_pmt + fv_pv

    if type == 'BGN':
        fv *= (1 + rate) # same as pv = pv*(1 + rate)

    return -1 * fv

calc_fv(rate=0.072, nper=10, pv=-1000)
```

2004.2314

```
calc_fv(rate=0.072, nper=10, pmt=72, pv=-1000)
```

1000.0000

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

```
np.random.seed(42)
data = np.random.randn(4, 4)
data

array([[ 0.4967, -0.1383,  0.6477,  1.523 ],
       [-0.2342, -0.2341,  1.5792,  0.7674],
       [-0.4695,  0.5426, -0.4634, -0.4657],
       [ 0.242 , -1.9133, -1.7249, -0.5623]])

data[data < 0] = -1
data[data > 0] = +1

data

array([[ 1., -1.,  1.,  1.],
       [-1., -1.,  1.,  1.],
       [-1.,  1., -1., -1.],
       [ 1., -1., -1., -1.]])
```

We have a second option in NumPy! NumPy's `np.where()` function works the same as Excel's `if()` function.

```
np.random.seed(42)
data = np.random.randn(4, 4)

np.where( # we can add whitespace inside parentheses ()!
         data < 0, # condition to test
         -1, # result if true
         np.where(data > 0, +1, data) # result if false
     )

array([[ 1., -1.,  1.,  1.],
       [-1., -1.,  1.,  1.],
       [-1.,  1., -1., -1.],
       [ 1., -1., -1., -1.]])
```

We also have a third option in NumPy! NumPy's `np.select()` function lets test multiple conditions!

```

np.random.seed(42)
data = np.random.randn(4, 4)

np.select( # we can add whitespace inside parentheses ()!
    condlist=[data<0, data>0],
    choicelist=[-1, +1],
    default=data
)

array([[ 1., -1.,  1.,  1.],
       [-1., -1.,  1.,  1.],
       [-1.,  1., -1., -1.],
       [ 1., -1., -1., -1.]])

```

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

We can use the growing annuity and perpetuity formulas to show: $x = \left[1 - \left(\frac{1+g}{1+r} \right)^t \right]$.

Then: $1 - x = \left(\frac{1+g}{1+r} \right)^t$.

Finally: $t = \frac{\log(1-x)}{\log(\frac{1+g}{1+r})}$

We do not need to accept an argument `c1` because C_1 cancels out!

```

def npmts(x, r, g):
    return np.log(1-x) / np.log((1 + g) / (1 + r))

npmts(0.5, 0.1, 0.05)

```

14.9000

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

Here are some data where the *IRR* is obvious!

```
c = np.array([-100, 110])
r = 0.10
```

First, write a function that calculates net present value (NPV) given cash flows in a NumPy array *c* and a discount rate in a scalar *r*. The *npv()* function below uses NumPy arrays to calculate NPV as:

$$NPV = \sum_{t=0}^T \frac{c_t}{(1+r)^t}$$

```
def calc_npv(r, c):
    t = np.arange(len(c))
    return (c / (1 + r)**t).sum()

calc_npv(r=r, c=c)
```

-0.0000

We can use a `while` loop to guess IRR values until we find an NPV close to zero. We can use the [Newton-Rapshon method](#) to make smarter guesses. If we have function $f(x)$ and guess x_t , our next guess should be $x_{t+1} = x_t - \frac{f(x_t)}{f'(x_t)}$. Here our $f(x)$ is $NPV(r)$, and we can approximate $f'(x_t)$ as $\frac{NPV(r+0.000001)-NPV(r)}{0.000001}$. We will make guess until $|NPV| < 0.000001$.

```
def calc_irr(c, guess=0):
    irr = guess
    npv = calc_npv(r=irr, c=c)
    while np.abs(npv) > 1e-6:
        slope = (calc_npv(r=irr+1e-6, c=c) - npv) / 1e-6 # Delta NPV / Delta r
        irr = irr - npv / slope
        npv = calc_npv(r=irr, c=c)
        # print(f'NPV is {npv}, IRR is {irr}')

    return irr

calc_irr(c=c) # c is the first positional argument, so we can omit "c="
```

0.1000

13.3.10 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])
```

Here is the return for the first period.

```
(prices[1] - prices[0] + dividends[1]) / prices[0]
```

0.5100

Here are the capital gains for every period.

```
prices[1:] - prices[:-1]
```

```
array([ 50, -50, -50,  50,  50, -50,  50])
```

```
(prices[1:] - prices[:-1] + dividends[1:]) / prices[:-1]
```

```
array([ 0.51 , -0.3267, -0.49 ,  1.04 ,  0.52 , -0.32 ,  0.52 ])
```

```
def returns(p, d):
    return (p[1:] - p[:-1] + d[1:]) / p[:-1]
```

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

```
array([ 0.51 , -0.3267, -0.49 ,  1.04 ,  0.52 , -0.32 ,  0.52 ])
```

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

```

def returns_2(p, d):
    r = (p[1:] - p[:-1] + d[1:]) / p[:-1]
    dp = d[1:] / p[:-1]
    cgp = r - dp
    return {'r':r, 'dp':dp, 'cgp':cgp}

returns_2(p=prices, d=dividends)

{'r': array([ 0.51 , -0.3267, -0.49 ,  1.04 ,  0.52 , -0.32 ,  0.52 ]),
 'dp': array([0.01 ,  0.0067,  0.01 ,  0.04 ,  0.02 ,  0.0133,  0.02 ]),
 'cgp': array([ 0.5   , -0.3333, -0.5   ,  1.     ,  0.5   , -0.3333,  0.5   ])}

```

13.3.12 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])

(numbers - numbers.min()) / (numbers.max() - numbers.min())

array([0.75, 0.  , 0.5 , 1.  , 0.5 ])

```

13.3.13 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 `numbers` to compare your functions with NumPy's `.var()` and `.std()` methods.

The population variance is “the average squared distance from the average.”

```
((numbers - numbers.mean())**2).mean()
```

```
0.4400
```

```
numbers.var()
```

```
0.4400
```

```
def var(x, sample=True):
    mu_x = x.mean()
    return ((x - mu_x)**2).sum() / (len(x) - sample)
```

```
var(numbers)
```

```
0.5500
```

```
var(numbers) == numbers.var(ddof=1)
```

```
True
```

```
var(numbers, sample=False)
```

```
0.4400
```

```
var(numbers, sample=False) == numbers.var()
```

```
True
```

```
def std(x, sample=True):
    return np.sqrt(var(x=x, sample=sample))
```

```
std(numbers, sample=False)
```

```
0.6633
```

```
    std(numbers, sample=False) == numbers.std()
```

```
True
```

```
    std(numbers)
```

```
0.7416
```

```
    std(numbers) == numbers.std(ddof=1)
```

```
True
```

14 McKinney Chapter 4 - Practice for Section 04

14.1 Announcements

1. Our second DataCamp course, *Intermediate Python*, is due Friday, 1/26, at 11:59 PM
2. I will record our week 4 lecture video on McKinney chapter 5 this Thursday evening, and the week 4 pre-class quiz is due before class next Tuesday, 1/30
3. Team projects
 1. Continue to join teams on Canvas > People > Team Projects
 2. I removed the join-a-team assignment, but I will give the first project assignment in early February, so join a team by then

14.2 10-minute Recap

14.2.1 NumPy Arrays

NumPy arrays are multidimensional data structures that can store numerical data efficiently and perform fast mathematical operations on them.

We use the abbreviation `np` for NumPy, and the magic `%precision` to display floats to fewer decimal places (here 4).

```
import this
```

The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.

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

```
import numpy as np
%precision 4

'%.4f'

np.arange(10)

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

np.zeros((2, 2))

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

  
np.zeros((2, 2, 2)) # x, y, z, and time?

array([[[[0., 0.],
          [0., 0.]],  

         [[0., 0.],
          [0., 0.]]],
```

```
[[[0., 0.],  
 [0., 0.]],  
  
 [[0., 0.],  
 [0., 0.]]])  
  
np.random.seed(42)  
np.random.randn(2, 2)  
  
array([[ 0.4967, -0.1383],  
 [ 0.6477,  1.523 ]])
```

14.2.2 Vectorized Functions

Vectorized computation is the process of applying an operation to an entire array or a subset of an array without using explicit loops. NumPy supports vectorized computation using universal functions (ufuncs), which are functions that operate on arrays element-wise.

```
4**0.5
```

```
2.0000
```

```
[i**0.5 for i in range(10)]
```

```
[0.0000,  
 1.0000,  
 1.4142,  
 1.7321,  
 2.0000,  
 2.2361,  
 2.4495,  
 2.6458,  
 2.8284,  
 3.0000]
```

```
np.sqrt(4)
```

```
2.0000
```

```
np.sqrt(np.arange(10))

array([0.      , 1.      , 1.4142, 1.7321, 2.      , 2.2361, 2.4495, 2.6458,
       2.8284, 3.      ])
```

14.2.3 Indexing and Slicing

Indexing and slicing are techniques to access or modify specific elements or subsets of an array. NumPy also supports advanced indexing methods, such as fancy indexing and boolean indexing, which allow more flexible and complex selection of array elements.

```
np.random.seed(42)
my_array = np.random.randn(3, 3)

my_array

array([[ 0.4967, -0.1383,  0.6477],
       [ 1.523 , -0.2342, -0.2341],
       [ 1.5792,  0.7674, -0.4695]])
```

How do we get the first “row”?

```
my_array[0]
```

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

How do we get the first element in the first row?

```
my_array[0][0]
```

```
0.4967
```

But, NumPy is designed for matrix algebra and lets us use the i, j notation from linear algebra class (and MATLAB).

```
my_array[0, 0]
```

```
0.4967
```

How do we get the first 2 rows? Slice it!

```
my_array[:2]
```

```
array([[ 0.4967, -0.1383,  0.6477],  
       [ 1.523 , -0.2342, -0.2341]])
```

How do we get the first 2 columns in the first 2 rows? Slice it using i, j notation!

```
my_array[:2, :2]
```

```
array([[ 0.4967, -0.1383],  
       [ 1.523 , -0.2342]])
```

14.3 Practice

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

```
a1 = np.arange(25)
```

```
a1
```

```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,  
       17, 18, 19, 20, 21, 22, 23, 24])
```

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

```
a2 = np.arange(0, 25, 3)
```

```
a2
```

```
array([ 0,  3,  6,  9, 12, 15, 18, 21, 24])
```

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

```
5 / 3
```

```
1.6667
```

```
5 // 3
```

```
1
```

```
5 % 3
```

```
2
```

```
a3 = np.array([i for i in range(101) if (i%3==0) or (i%5==0)])
```

```
a3
```

```
array([ 0,  3,  5,  6,  9, 10, 12, 15, 18, 20, 21, 24, 25,
       27, 30, 33, 35, 36, 39, 40, 42, 45, 48, 50, 51, 54,
       55, 57, 60, 63, 65, 66, 69, 70, 72, 75, 78, 80, 81,
       84, 85, 87, 90, 93, 95, 96, 99, 100])
```

```
a3_alt = np.arange(101)
a3_alt = a3_alt[(a3_alt%3==0) | (a3_alt%5==0)] # must use " | " instead of "or" in NumPy
```

```
a3_alt
```

```
array([ 0,  3,  5,  6,  9, 10, 12, 15, 18, 20, 21, 24, 25,
       27, 30, 33, 35, 36, 39, 40, 42, 45, 48, 50, 51, 54,
       55, 57, 60, 63, 65, 66, 69, 70, 72, 75, 78, 80, 81,
       84, 85, 87, 90, 93, 95, 96, 99, 100])
```

```
(a3 == a3_alt).all()
```

```
True
```

```
np.allclose(a3, a3_alt)
```

True

14.3.4 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?

```
np.arange(0, 100_001, 2)**2
```

```
array([0, 4, 16, ..., 1409265424, 1409665412,  
1410065408])
```

On some computers, the output above is wrong because NumPy defaults to 32-bit integers, depending on the computer! *Always check your output!* To avoid this problem, we can force `np.arange()` to use 64-bit integers with the `dtype=` argument.

```
np.arange(0, 100_001, 2, dtype=np.int64)**2
```

```
array([0, 4, 16, ..., 9999200016,  
9999600004, 10000000000], dtype=int64)
```

We can use the `%timeit` magic to time which code is faster! The `%timeit` magic runs the code on the same line many times and reports the mean computation time. The `%%timeit` magic with two percent signs runs the code in the same cell many times and reports the mean computation time.

```
%timeit np.arange(0, 100_001, 2, dtype=np.int64)**2
```

41.6 μ s \pm 3.42 μ s per loop (mean \pm std. dev. of 7 runs, 10,000 loops each)

```
%%timeit np.array([i**2 for i in range(0, 100_001)])
```

12 ms \pm 2.06 ms per loop (mean \pm std. dev. of 7 runs, 100 loops each)

The NumPy version is about 1,000 times faster!

14.3.5 Write a function that mimic Excel's pv function.

Here is how we call Excel's pv function: =PV(rate, nper, pmt, [fv], [type]) We can use the annuity and lump sum present value formulas.

Present value of an annuity payment pmt: $PV_{pmt} = \frac{pmt}{rate} \times \left(1 - \frac{1}{(1+rate)^{nper}}\right)$

Present value of a lump sum fv: $PV_{fv} = \frac{fv}{(1+rate)^{nper}}$

```
def pv(rate, nper, pmt=None, fv=None, type=None):
    if pmt is None:
        pmt = 0
    if fv is None:
        fv = 0
    if type is None:
        type = 'END'

    pv_pmt = (pmt / rate) * (1 - 1 / (1 + rate)**nper)
    pv_fv = fv / (1 + rate)**nper
    pv = pv_pmt + pv_fv

    if type == 'BGN':
        pv *= (1 + rate) # same as pv = pv*(1 + rate)

    return -1 * pv

pv(rate = 0.05, nper = 14, fv = 1_000, type = 'END')
```

-505.0680

```
a = 5
a*= 5
a*= 5
a
```

125

14.3.6 Write a function that mimic Excel's fv function.

```
def calc_fv(rate, nper, pmt=None, pv=None, type=None):
    if pmt is None:
        pmt = 0
    if pv is None:
        pv = 0
    if type is None:
        type = 'END'

    fv_pmt = (pmt / rate) * ((1 + rate)**nper - 1)
    fv_pv = pv * (1 + rate)**nper
    fv = fv_pmt + fv_pv

    if type == 'BGN':
        fv *= (1 + rate) # same as pv = pv*(1 + rate)

    return -1 * fv

calc_fv(rate=0.072, nper=10, pv=-1000)
```

2004.2314

```
calc_fv(rate=0.072, nper=10, pmt=72, pv=-1000)
```

1000.0000

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

```
np.random.seed(42)
data = np.random.randn(4, 4)
data

array([[ 0.4967, -0.1383,  0.6477,  1.523 ],
       [-0.2342, -0.2341,  1.5792,  0.7674],
       [-0.4695,  0.5426, -0.4634, -0.4657],
       [ 0.242 , -1.9133, -1.7249, -0.5623]])
```

One approach, is to slice `data` with a Boolean array, just like we did above for the multiples of 3 and 5 exercise.

```
data[data < 0] = -1
data[data > 0] = +1

data

array([[ 1., -1.,  1.,  1.],
       [-1., -1.,  1.,  1.],
       [-1.,  1., -1., -1.],
       [ 1., -1., -1., -1.]])
```

A second approach, is to use NumPy's `np.where()` function. `np.where()` is similar to Excel's `if()`, but vectorized!

```
np.random.seed(42)
data = np.random.randn(4, 4)
np.where( # Python ignores whitespace inside parentheses ()!
         data < 0, # condition to test
         -1, # result if true
         np.where(data > 0, +1, data) # result if false
     )

array([[ 1., -1.,  1.,  1.],
       [-1., -1.,  1.,  1.],
       [-1.,  1., -1., -1.],
       [ 1., -1., -1., -1.]])
```

A THIRD approach, is to use NumPy's `np.select()` function. `np.select()` is similar to Excel's `if()`, but vectorized and allows many conditions!

```
np.random.seed(42)
data = np.random.randn(4, 4)
np.select( # Python ignores whitespace inside parentheses ()!
          condlist=[data<0, data>0],
          choicelist=[-1, +1],
          default=data
      )
```

```
array([[ 1., -1.,  1.,  1.],
       [-1., -1.,  1.,  1.],
       [-1.,  1., -1., -1.],
       [ 1., -1., -1., -1.]])
```

14.3.8 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 c_1 , r , and g that represent C_1 , r , and g . The present value of a growing perpetuity is $PV = \frac{C_1}{r-g}$, and the present value of a growing annuity is $PV = \frac{C_1}{r-g} \left[1 - \left(\frac{1+g}{1+r} \right)^t \right]$.

We can use the growing annuity and perpetuity formulas to show: $x = \left[1 - \left(\frac{1+g}{1+r} \right)^t \right]$.

Then: $1 - x = \left(\frac{1+g}{1+r} \right)^t$.

Finally: $t = \frac{\log(1-x)}{\log\left(\frac{1+g}{1+r}\right)}$

We do not need to accept an argument c_1 because C_1 cancels out!

```
def npmts(x, r, g):
    return np.log(1-x) / np.log((1 + g) / (1 + r))

npmts(0.5, 0.1, 0.05)
```

14.9000

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

Here are some data where the IRR is obvious!

```
c = np.array([-100, +110])
r = 0.1
```

First, write a function that calculates net present value (NPV) given cash flows in a NumPy array c and a discount rate in a scalar r . The `npv()` function below uses NumPy arrays to calculate NPV as:

$$NPV = \sum_{t=0}^T \frac{c_t}{(1+r)^t}$$

```
def calc_npv(r, c):
    t = np.arange(len(c))
    return (c / (1 + r)**t).sum()
```

```
calc_npv(r=r, c=c)
```

```
-0.0000
```

We can use a `while` loop to guess IRR values until we find an NPV close to zero. We can use the [Newton-Rapshon method](#) to make smarter guesses. If we have function $f(x)$ and guess x_t , our next guess should be $x_{t+1} = x_t - \frac{f(x_t)}{f'(x_t)}$. Here our $f(x)$ is $NPV(r)$, and we can approximate $f'(x_t)$ as $\frac{NPV(r+0.000001)-NPV(r)}{0.000001}$. We will make guess until $|NPV| < 0.000001$.

```
def calc_irr(c, guess=0):
    irr = guess
    npv = calc_npv(r=irr, c=c)
    while np.abs(npv) > 1e-6:
        slope = (calc_npv(r=irr+1e-6, c=c) - npv) / 1e-6
        irr = irr - npv / slope # N
        npv = calc_npv(r=irr, c=c)
        # print(f'NPV is {npv}, and IRR is {irr}')
```



```
return irr
```



```
calc_irr(c=c)
```

```
0.1000
```

14.3.10 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])
```

We can slice `prices` to calculate capital gains without a for loop!

```
prices[1:] - prices[:-1]
```

```

array([ 50, -50, -50,  50,  50, -50,  50])

(prices[1:] - prices[:-1] + dividends[1:]) / prices[:-1]

array([ 0.51 , -0.3267, -0.49 ,  1.04 ,  0.52 , -0.32 ,  0.52 ])

def returns(p, d):
    return (p[1:] - p[:-1] + d[1:]) / p[:-1]

returns(p=prices, d=dividends)

array([ 0.51 , -0.3267, -0.49 ,  1.04 ,  0.52 , -0.32 ,  0.52 ])

```

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

14.3.12 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])

(numbers - numbers.min()) / (numbers.max() - numbers.min())

array([0.75, 0. , 0.5 , 1. , 0.5 ])

```

14.3.13 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 `numbers` to compare your functions with NumPy's `.var()` and `.std()` methods.

```
((numbers - numbers.mean())**2).mean()

0.4400

numbers.var()

0.4400

def var(x, sample=True):
    mu_x = x.mean()
    return ((x - mu_x)**2).sum() / (len(x) - sample)

var(numbers)

0.5500

var(numbers) == numbers.var(ddof=1)

True

var(numbers, sample=False)

0.4400

var(numbers, sample=False) == numbers.var()

True

def std(x, sample=True):
    return np.sqrt(var(x=x, sample=sample))

std(numbers, sample=False)

0.6633
```

```
    std(numbers, sample=False) == numbers.std()
```

```
True
```

```
    std(numbers)
```

```
0.7416
```

```
    std(numbers) == numbers.std(ddof=1)
```

```
True
```

15 McKinney Chapter 4 - Practice for Section 05

15.1 Announcements

1. Our second DataCamp course, *Intermediate Python*, is due Friday, 1/26, at 11:59 PM
2. I will record our week 4 lecture video on McKinney chapter 5 this Thursday evening, and the week 4 pre-class quiz is due before class next Tuesday, 1/30
3. Team projects
 1. Continue to join teams on Canvas > People > Team Projects
 2. I removed the join-a-team assignment, but I will give the first project assignment in early February, so join a team by then

15.2 10-minute Recap

15.2.1 NumPy Arrays

NumPy arrays are multidimensional data structures that can store numerical data efficiently and perform fast mathematical operations on them.

```
import this
```

The Zen of Python, by Tim Peters

```
Beautiful is better than ugly.  
Explicit is better than implicit.  
Simple is better than complex.  
Complex is better than complicated.  
Flat is better than nested.  
Sparse is better than dense.  
Readability counts.  
Special cases aren't special enough to break the rules.  
Although practicality beats purity.
```

Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!

```
import numpy as np
%precision 4

'%.4f'

np.arange(10)

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

np.ones(10)

array([1., 1., 1., 1., 1., 1., 1., 1., 1., 1.])

np.ones((2, 2))

array([[1., 1.],
       [1., 1.]]) 

np.ones((2, 2, 2, 2))

array([[[[1., 1.],
          [1., 1.]],

         [[1., 1.],
```

```
[1., 1.]],
```

```
[[[1., 1.],  
 [1., 1.]],
```

```
[[1., 1.],  
 [1., 1.]])
```

We will use standard normal random variables from `np.random.randn()` to tinker with ideas. If we want to get the same random numbers (every time and across computers), we should use `np.random.seed()` to set the seed of the random number generator.

```
np.random.seed(42)  
np.random.randn(2, 2)
```

```
array([[ 0.4967, -0.1383],  
       [ 0.6477,  1.523 ]])
```

15.2.2 Vectorized Functions

Vectorized computation is the process of applying an operation to an entire array or a subset of an array without using explicit loops. NumPy supports vectorized computation using universal functions (ufuncs), which are functions that operate on arrays element-wise.

```
4**(1/2)
```

```
2.0000
```

```
np.sqrt(4)
```

```
2.0000
```

```
np.arange(5)
```

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

```
np.sqrt(np.arange(5))

array([0.      , 1.      , 1.4142, 1.7321, 2.      ])
```

15.2.3 Indexing and Slicing

Indexing and slicing are techniques to access or modify specific elements or subsets of an array. NumPy also supports advanced indexing methods, such as fancy indexing and boolean indexing, which allow more flexible and complex selection of array elements.

```
np.random.seed(42) # 42 is the answer to everything from *The Hitchhikers Guide to the Gal
my_array = np.random.randn(3, 3)

my_array

array([[ 0.4967, -0.1383,  0.6477],
       [ 1.523 , -0.2342, -0.2341],
       [ 1.5792,  0.7674, -0.4695]])
```

We can index the “first row” with the [0] index.

```
my_array[0]

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

We can index the first element in the first row by chaining a [0] index.

```
my_array[0][0]

0.4967
```

A simpler syntax for [0][0] is [0, 0], which has a i, j interpretation!

```
my_array[0, 0]

0.4967
```

we can combine this notation with slices, like for lists! What if we want the first two columns in the first two rows?

```
my_array[:2] # first two rows

array([[ 0.4967, -0.1383,  0.6477],
       [ 1.523 , -0.2342, -0.2341]])

my_array[:2][:2] # first two rows and first two columns

array([[ 0.4967, -0.1383,  0.6477],
       [ 1.523 , -0.2342, -0.2341]])

my_array[:2, :2] # first two rows and first two columns

array([[ 0.4967, -0.1383],
       [ 1.523 , -0.2342]])
```

15.3 Practice

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

```
a1 = np.arange(25)

a1

array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
       17, 18, 19, 20, 21, 22, 23, 24])
```

The above is much easier than a step-by-step solution!

```
np.array(list(range(25)))

array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
       17, 18, 19, 20, 21, 22, 23, 24])
```

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

```
a2 = np.arange(0, 25, 3) # start, stop, size (if we want to give size, we must give start)
```

```
a2
```

```
array([ 0,  3,  6,  9, 12, 15, 18, 21, 24])
```

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

```
a3 = np.array([i for i in range(101) if (i%3==0) | (i%5==0)])
```

```
a3
```

```
array([ 0,  3,  5,  6,  9, 10, 12, 15, 18, 20, 21, 24, 25,
       27, 30, 33, 35, 36, 39, 40, 42, 45, 48, 50, 51, 54,
       55, 57, 60, 63, 65, 66, 69, 70, 72, 75, 78, 80, 81,
       84, 85, 87, 90, 93, 95, 96, 99, 100])
```

```
a3_alt = np.arange(101)
```

```
a3_alt = a3_alt[ (a3_alt%3==0) | (a3_alt%5==0) ]
```

```
a3_alt
```

```
array([ 0,  3,  5,  6,  9, 10, 12, 15, 18, 20, 21, 24, 25,
       27, 30, 33, 35, 36, 39, 40, 42, 45, 48, 50, 51, 54,
       55, 57, 60, 63, 65, 66, 69, 70, 72, 75, 78, 80, 81,
       84, 85, 87, 90, 93, 95, 96, 99, 100])
```

How can we make sure these two answers are identical?

```
a3 == a3_alt
```

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

```
(a3 == a3_alt).all()
```

```
True
```

```
np.allclose(a3, a3_alt)
```

```
True
```

15.3.4 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?

```
np.arange(0, 100_001, 2)**2
```

```
array([ 0, 4, 16, ..., 1409265424, 1409665412,
       1410065408])
```

On some computers, the output above is wrong because NumPy defaults to 32-bit integers, depending on the computer! *Always check your output!* To avoid this problem, we can force `np.arange()` to use 64-bit integers with the `dtype=` argument.

```
np.arange(0, 100_001, 2, dtype=np.int64)**2
```

```
array([ 0, 4, 16, ..., 9999200016,
       9999600004, 10000000000], dtype=int64)
```

We can use the `%timeit` magic to time which code is faster! The `%timeit` magic runs the code on the same line many times and reports the mean computation time. The `%%timeit` magic with two percent signs runs the code in the same cell many times and reports the mean computation time.

```
%timeit np.arange(0, 100_001, 2, dtype=np.int64)**2
```

```
47.8 µs ± 4.61 µs per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

```
%timeit np.array([i**2 for i in range(0, 100_001)])
```

```
13.6 ms ± 761 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

The NumPy version is about 1,000 times faster!

15.3.5 Write a function that mimic Excel's pv function.

Here is how we call Excel's pv function: =PV(rate, nper, pmt, [fv], [type]) We can use the annuity and lump sum present value formulas.

Present value of an annuity payment pmt: $PV_{pmt} = \frac{pmt}{rate} \times \left(1 - \frac{1}{(1+rate)^{nper}}\right)$

Present value of a lump sum fv: $PV_{fv} = \frac{fv}{(1+rate)^{nper}}$

```
def pv(rate, nper, pmt=None, fv=None, type=None):
    if pmt is None:
        pmt = 0
    if fv is None:
        fv = 0
    if type is None:
        type = 'END'

    pv_pmt = (pmt / rate) * (1 - 1 / (1 + rate)**nper)
    pv_fv = fv / (1 + rate)**nper
    pv = -1 * (pv_pmt + pv_fv)

    if type == 'BGN':
        pv *= (1 + rate) # same as pv = pv*(1 + rate)

    return -1 * pv

pv(rate = 0.05, nper = 1_000, pmt = 5, type = 'BGN')
```

```
105.0000
```

```
a = 5
a*= 5
a*= 5
```

a

125

15.3.6 Write a function that mimic Excel's fv function.

```
def calc_fv(rate, nper, pmt=None, pv=None, type=None):
    if pmt is None:
        pmt = 0
    if pv is None:
        pv = 0
    if type is None:
        type = 'END'

    fv_pmt = (pmt / rate) * ((1 + rate)**nper - 1)
    fv_pv = pv * (1 + rate)**nper
    fv = fv_pmt + fv_pv

    if type == 'BGN':
        fv *= (1 + rate) # same as pv = pv*(1 + rate)

    return -1 * fv

calc_fv(rate=0.072, nper=10, pv=-1000)
```

2004.2314

```
calc_fv(rate=0.072, nper=10, pmt=72, pv=-1000)
```

1000.0000

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

```
np.random.seed(42)
data = np.random.randn(4, 4)
data
```

```

array([[ 0.4967, -0.1383,  0.6477,  1.523 ],
       [-0.2342, -0.2341,  1.5792,  0.7674],
       [-0.4695,  0.5426, -0.4634, -0.4657],
       [ 0.242 , -1.9133, -1.7249, -0.5623]])
```

```

data[data < 0] = -1
data[data > 0] = +1

data # here "1." and "-1." indicate that these values are floats
```

```

array([[ 1., -1.,  1.,  1.],
       [-1., -1.,  1.,  1.],
       [-1.,  1., -1., -1.],
       [ 1., -1., -1., -1.]])
```

Here is a second option! NumPy's `np.where()` function has the same logic as Excel's `if()` function! I will recreate `data` so we start from the same point.

```

np.random.seed(42)
data = np.random.randn(4, 4)

np.where(data < 0, -1, np.where(data > 0, +1, data))

array([[ 1., -1.,  1.,  1.],
       [-1., -1.,  1.,  1.],
       [-1.,  1., -1., -1.],
       [ 1., -1., -1., -1.]])
```

The nested `np.where()` function calls can get confusing! An option is to insert white space!

```

np.random.seed(42)
data = np.random.randn(4, 4)

np.where(
    data < 0, # condition
    -1, # result if True
    np.where(data > 0, +1, data) # result if False
)
```

```
array([[ 1., -1.,  1.,  1.],
       [-1., -1.,  1.,  1.],
       [-1.,  1., -1., -1.],
       [ 1., -1., -1., -1.]])
```

Here is a third option! NumPy's `np.select()` function lets us test *many* conditions! I will recreate `data` so we start from the same point.

```
np.random.seed(42)
data = np.random.randn(4, 4)

np.select(
    condlist=[data<0, data>0],
    choicelist=[-1, +1],
    default=data
)

array([[ 1., -1.,  1.,  1.],
       [-1., -1.,  1.,  1.],
       [-1.,  1., -1., -1.],
       [ 1., -1., -1., -1.]])
```

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

We can use the growing annuity and perpetuity formulas to show: $x = \left[1 - \left(\frac{1+g}{1+r} \right)^t \right]$.

Then: $1 - x = \left(\frac{1+g}{1+r} \right)^t$.

Finally: $t = \frac{\log(1-x)}{\log(\frac{1+g}{1+r})}$

We do not need to accept an argument `c1` because C_1 cancels out!

```
def npmts(x, r, g):
    return np.log(1-x) / np.log((1 + g) / (1 + r))
```

```
npmts(0.5, 0.1, 0.05)
```

14.9000

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

Let us pick a set of cash flows c where we know the internal rate of return! For the following c , the IRR is 10%.

```
c = np.array([-100, +110])
r = 0.1
```

First we need a function to calculate net present value (NPV) from c and r ! The `npv()` function below uses NumPy arrays to calculate NPV as:

$$NPV = \sum_{t=0}^T \frac{c_t}{(1+r)^t}$$

```
def calc_npv(r, c):
    t = np.arange(len(c))
    return (c / (1 + r)**t).sum()

calc_npv(r=r, c=c)
```

-0.0000

We can use a `while` loop to guess IRR values until we find an NPV close to zero. We can use the [Newton-Rapshon method](#) to make smarter guesses. If we have function $f(x)$ and guess x_t , our next guess should be $x_{t+1} = x_t - \frac{f(x_t)}{f'(x_t)}$. Here our $f(x)$ is $NPV(r)$, and we can approximate $f'(x_t)$ as $\frac{NPV(r+0.000001)-NPV(r)}{0.000001}$. We will make guess until $|NPV| < 0.000001$.

```
def calc_irr(c, guess=0, tol=1e-6, step=1e-6):
    irr = guess
    npv = calc_npv(r=irr, c=c)
    while np.abs(npv) > tol:
        deriv = (calc_npv(r=irr+step, c=c) - npv) / step # Delta NPV / Delta r
        irr = irr - npv / deriv
```

```
    npv = calc_npv(r=IRR, c=c)
    # print(f'IRR is {IRR}, and NPV is {npv}')
    return IRR

calc_irr(c=c)
```

0.1000

15.3.10 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])
```

We want to slice our arrays to “lag” or “shift” them! For example, we slice the `prices` array to calculate capital gains as follows.

```
prices[1:] - prices[:-1]
```

```
array([-50, -50, -50,  50,  50, -50,  50])
```

```
def returns(p, d):
    return (p[1:] - p[:-1] + d[1:]) / p[:-1]

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

```
array([ 0.51, -0.3267, -0.49,  1.04,  0.52, -0.32,  0.52])
```

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

15.3.12 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])
```

15.3.13 Write functions var() and std() that calculate variance and standard deviation.

```
((numbers - numbers.mean())**2).mean()
```

0.4400

```
numbers.var()
```

0.4400

```
def var(x, sample=True):
    mu_x = x.mean()
    return ((x - mu_x)**2).sum() / (len(x) - sample)
```

```
var(numbers)
```

0.5500

```
var(numbers) == numbers.var(ddof=1)
```

True

```
var(numbers, sample=False)
```

0.4400

```
var(numbers, sample=False) == numbers.var()
```

True

```
def std(x, sample=True):
    return np.sqrt(var(x=x, sample=sample))
```

```
std(numbers, sample=False)
```

0.6633

```
std(numbers, sample=False) == numbers.std()
```

True

```
std(numbers)
```

0.7416

```
std(numbers) == numbers.std(ddof=1)
```

True

Part IV

Week 4

16 McKinney Chapter 5 - Getting Started with pandas

16.1 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 abbreviation for *panel data*, 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.

16.2 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.

16.2.1 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
```

16.2.2 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
```

	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

```
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]` syntax 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

16.2.3 Index Objects

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

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

Index objects are immutable!

```
# 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'])
dup_labels
```

```
Index(['foo', 'foo', 'bar', 'bar'], dtype='object')
```

16.3 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.

16.3.1 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

	one	two	three	four
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

16.3.2 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
```

	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[data < 5] = 0  
data
```

	one	two	three	four
Ohio	0	0	0	0
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.

16.3.3 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', 'Texas', 'Colorado'])
df2 = pd.DataFrame(np.arange(12.).reshape((4, 3)), columns=list('bde'), index=['Utah', 'Ohio', 'Texas', 'Oregon'])
```

```
df1
```

	b	c	d
Ohio	0.0000	1.0000	2.0000
Texas	3.0000	4.0000	5.0000
Colorado	6.0000	7.0000	8.0000

```
df2
```

	b	d	e
Utah	0.0000	1.0000	2.0000
Ohio	3.0000	4.0000	5.0000
Texas	6.0000	7.0000	8.0000
Oregon	9.0000	10.0000	11.0000

```
df1 + df2
```

	b	c	d	e
Colorado	NaN	NaN	NaN	NaN
Ohio	3.0000	NaN	6.0000	NaN
Oregon	NaN	NaN	NaN	NaN
Texas	9.0000	NaN	12.0000	NaN
Utah	NaN	NaN	NaN	NaN

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

```
df1
```

	A
0	1
1	2

```
df2
```

	B
0	3
1	4

```
df1 - df2
```

	A	B
0	NaN	NaN
1	NaN	NaN

Always check your output!

16.3.3.1 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
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

16.3.3.2 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
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

16.3.4 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”:

```
1.5792 - 0.4967
```

```
1.0825
```

```
frame.apply(lambda x: x.max() - x.min()) # implied axis=0
```

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

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

```
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 slower than optimized, built-in methods. Here is an example of the speed costs of `.apply()`:

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

```
15.2 µs ± 4.5 µs per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

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

32.6 μ s \pm 9.67 μ s per loop (mean \pm std. dev. of 7 runs, 10,000 loops each)

16.4 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() # implied axis=0
```

```
one    9.2500  
two   -5.8000  
dtype: float64
```

```
df.sum(axis=1)
```

```
a    1.4000  
b    2.6000  
c    0.0000  
d   -0.5500  
dtype: float64
```

```
df.mean(axis=1, skipna=False)
```

```
a      NaN
b    1.3000
c      NaN
d   -0.2750
dtype: float64
```

The `.idxmax()` method returns the label for the maximum observation.

```
df.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
```

16.4.1 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.tail()
```

Date	Adj Close				Close				Dividends	
	AAPL	GOOG	IBM	MSFT	AAPL	GOOG	IBM	MSFT	AAPL	
2024-01-19	191.5600	147.9700	171.4800	398.6700	191.5600	147.9700	171.4800	398.6700	0.0000	
2024-01-22	193.8900	147.7100	172.8300	396.5100	193.8900	147.7100	172.8300	396.5100	0.0000	
2024-01-23	195.1800	148.6800	173.9400	398.9000	195.1800	148.6800	173.9400	398.9000	0.0000	
2024-01-24	194.5000	150.3500	173.9300	402.5600	194.5000	150.3500	173.9300	402.5600	0.0000	
2024-01-25	194.1700	153.6400	190.4300	404.8700	194.1700	153.6400	190.4300	404.8700	0.0000	

```
prices['Adj Close'].tail()
```

Date	AAPL	GOOG	IBM	MSFT
2024-01-19	191.5600	147.9700	171.4800	398.6700
2024-01-22	193.8900	147.7100	172.8300	396.5100
2024-01-23	195.1800	148.6800	173.9400	398.9000
2024-01-24	194.5000	150.3500	173.9300	402.5600
2024-01-25	194.1700	153.6400	190.4300	404.8700

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
...
2024-01-19	0.0155	0.0206	0.0278	0.0122
2024-01-22	0.0122	-0.0018	0.0079	-0.0054
2024-01-23	0.0067	0.0066	0.0064	0.0060
2024-01-24	-0.0035	0.0112	-0.0001	0.0092
2024-01-25	-0.0017	0.0219	0.0949	0.0057

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    0.3648
GOOG    0.2591
IBM     0.0990
MSFT    0.2005
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    0.3277
GOOG    0.3071
```

```
IBM    0.2272
MSFT   0.2722
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'])
```

```
0.4926
```

We can also calculate correlation matrices.

```
returns.corr()
```

	AAPL	GOOG	IBM	MSFT
AAPL	1.0000	0.5189	0.4275	0.5235
GOOG	0.5189	1.0000	0.3953	0.5626
IBM	0.4275	0.3953	1.0000	0.4926
MSFT	0.5235	0.5626	0.4926	1.0000

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

```
0.4926
```

17 McKinney Chapter 5 - Practice for Section 02

17.1 Announcements

1. No DataCamp this week, but I suggest you keep working on it
2. Keep forming groups, and I will post our first project early next week

17.2 10-Minute Recap

```
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'
```

There are two pandas data structures:

1. Data frames are like worksheets in an Excel workbook (2-D, mixed data type)
2. Series are like a column in a worksheet (1-D, only one data type)

```
np.random.seed(42)
df = pd.DataFrame(
    data=np.random.randn(3, 5),
    index=list('ABC'),
    columns=list('abcde')
)
df
```

	a	b	c	d	e
A	0.4967	-0.1383	0.6477	1.5230	-0.2342
B	-0.2341	1.5792	0.7674	-0.4695	0.5426
C	-0.4634	-0.4657	0.2420	-1.9133	-1.7249

We can slice data frames two ways!

1. By integer locations with the `.iloc[]` method
2. By row and column names with the `.loc[]` method

How can we grab the first 2 rows and 3 columns?

```
df.iloc[:2, :3] # pandas .iloc[] uses j,k notation, like NumPy
```

	a	b	c
A	0.4967	-0.1383	0.6477
B	-0.2341	1.5792	0.7674

When we slice by names or string, pandas includes both left and right edges!

```
df.loc['A':'B', 'a':'c']
```

	a	b	c
A	0.4967	-0.1383	0.6477
B	-0.2341	1.5792	0.7674

How can we add a column?

```
df['f'] = 2_001
```

```
df
```

	a	b	c	d	e	f
A	0.4967	-0.1383	0.6477	1.5230	-0.2342	2001
B	-0.2341	1.5792	0.7674	-0.4695	0.5426	2001
C	-0.4634	-0.4657	0.2420	-1.9133	-1.7249	2001

What if we want to insert a column between “b” and “c”? We can use the `.insert()` method!
Note. I was surprised the `.insert()` operates “in place” without an option to override!

```
df.insert(  
    loc=2,  
    column='C',  
    value=5  
)  
  
df.loc[:, 'a':'c']
```

	a	b	C	c
A	0.4967	-0.1383	5	0.6477
B	-0.2341	1.5792	5	0.7674
C	-0.4634	-0.4657	5	0.2420

```
df[['a', 'b', 'C', 'c']]
```

	a	b	C	c
A	0.4967	-0.1383	5	0.6477
B	-0.2341	1.5792	5	0.7674
C	-0.4634	-0.4657	5	0.2420

A series is the other data structure in pandas!

```
ser = pd.Series(data=np.arange(2.), index=['B', 'C'])
```

```
ser
```

```
B    0.0000  
C    1.0000  
dtype: float64
```

```
df['g'] = ser
```

```
df
```

	a	b	C	c	d	e	f	g
A	0.4967	-0.1383	5	0.6477	1.5230	-0.2342	2001	NaN
B	-0.2341	1.5792	5	0.7674	-0.4695	0.5426	2001	0.0000
C	-0.4634	-0.4657	5	0.2420	-1.9133	-1.7249	2001	1.0000

17.3 Practice

```

tickers = 'AAPL IBM MSFT GOOG'
prices = yf.download(tickers=tickers)

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

returns = (
    prices['Adj Close'] # slice adj close column
    .iloc[:-1] # drop last row with intra day prices, which are sometimes missing
    .pct_change() # calculate returns
    .dropna() # drop leading rows with at least one missing value
)

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
...
2024-01-26	-0.0090	0.0010	-0.0158	-0.0023
2024-01-29	-0.0036	0.0068	-0.0015	0.0143
2024-01-30	-0.0192	-0.0116	0.0039	-0.0028
2024-01-31	-0.0194	-0.0735	-0.0224	-0.0269
2024-02-01	0.0133	0.0064	0.0176	0.0156

```

returns = (
    prices['Adj Close']
    .iloc[:-1]
    .pct_change()
    .dropna()
)
returns

```

	AAPL	GOOG	IBM	MSFT
Date				
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
...
2024-01-26	-0.0090	0.0010	-0.0158	-0.0023
2024-01-29	-0.0036	0.0068	-0.0015	0.0143
2024-01-30	-0.0192	-0.0116	0.0039	-0.0028
2024-01-31	-0.0194	-0.0735	-0.0224	-0.0269
2024-02-01	0.0133	0.0064	0.0176	0.0156

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

```
returns.mean()
```

```

AAPL    0.0014
GOOG    0.0010
IBM     0.0004
MSFT    0.0008
dtype: float64

```

We if want an equally-weighted portfolio return? We could take the mean of each *row* with `.mean(axis=1)`. The mean is the same as the sum of 0.25 times each of the 4 columns.

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

```
Date
```

```
2004-08-20    0.0224
2004-08-23    0.0041
2004-08-24   -0.0032
2004-08-25    0.0152
2004-08-26    0.0146
...
2024-01-26   -0.0065
2024-01-29    0.0040
2024-01-30   -0.0074
2024-01-31   -0.0356
2024-02-01    0.0132
Length: 4896, dtype: float64
```

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

```
returns.std()
```

```
AAPL    0.0206
GOOG    0.0194
IBM     0.0143
MSFT    0.0171
dtype: float64
```

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

We multiply by T to annualize means, where T is the number of observations per year.

```
returns.mean().mul(252)
```

```
AAPL    0.3625
GOOG    0.2552
IBM     0.0980
MSFT    0.2002
dtype: float64
```

We multiply by \sqrt{T} to annualize volatilities, where T is the number of observations per year.

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

```
AAPL    0.3276
GOOG    0.3074
IBM     0.2272
MSFT    0.2722
dtype: float64
```

17.3.4 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).

```
vols = returns.std().mul(np.sqrt(252) * 100)
means = returns.mean().mul(252 * 100)

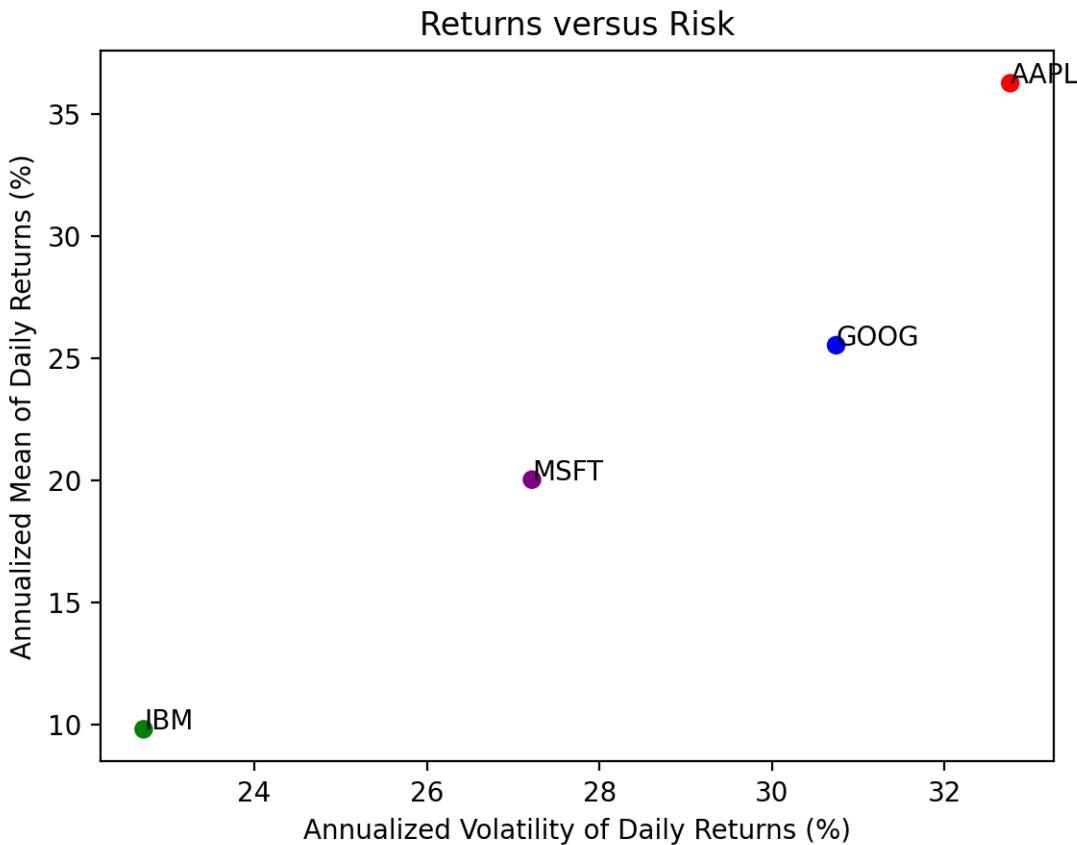
plt.scatter(
    x=vols,
    y=means,
    c=['red', 'blue', 'green', 'purple']
)

plt.xlabel('Annualized Volatility of Daily Returns (%)')
plt.ylabel('Annualized Mean of Daily Returns (%)')

# plt.xlim((0, vols.max() + 5))
# plt.ylim((0, means.max() + 5))

# add tickers to each point
for i in means.index: # loop over ticker index
    plt.text( # plots string s at coordinates x and y
        x=vols[i], # indexes volatility
        y=means[i], # indexes mean return
        s=i # ticker index
    )

plt.title('Returns versus Risk')
plt.show() # suppresses output of last function call
```



17.3.5 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](https://en.wikipedia.org/wiki/Dow_Jones_Industrial_Average). We will need to download new data, into `tickers2`, `prices2`, and `returns2`.

```
url2 = 'https://en.wikipedia.org/wiki/Dow_Jones_Industrial_Average'
```

```
wiki2 = pd.read_html(io=url2)
```

```
tickers2 = wiki2[1]['Symbol'].to_list()
```

```
tickers2[:5]
```

```
['MMM', 'AXP', 'AMGN', 'AAPL', 'BA']
```

```
prices2 = yf.download(tickers=tickers2)
```

```
[*****100%*****] 30 of 30 completed
```

```
returns2 = (
    prices2['Adj Close'] # slice the adj close columns for all 30 tickers
    .iloc[:-1] # drop last row with incomplete prices because we are before the close
    .pct_change() # calculate returns
    .dropna() # drops any row with incomplete data
)

returns2
```

Date	AAPL	AMGN	AXP	BA	CAT	CRM	CSCO	CVX	DIS	DOW	...
2019-03-21	0.0368	0.0040	0.0095	-0.0092	0.0079	0.0210	0.0128	0.0094	-0.0121	-0.0165	...
2019-03-22	-0.0207	-0.0270	-0.0211	-0.0283	-0.0320	-0.0326	-0.0222	-0.0220	-0.0040	-0.0078	...
2019-03-25	-0.0121	-0.0006	-0.0038	0.0229	0.0124	-0.0038	-0.0002	-0.0016	-0.0041	0.0113	...
2019-03-26	-0.0103	0.0090	0.0042	-0.0002	0.0035	-0.0092	0.0095	0.0101	0.0218	-0.0061	...
2019-03-27	0.0090	-0.0104	-0.0047	0.0103	-0.0049	-0.0269	-0.0017	-0.0108	0.0013	0.0256	...
...
2024-01-26	-0.0090	0.0049	0.0710	0.0178	-0.0045	0.0033	-0.0036	0.0038	0.0053	-0.0160	...
2024-01-29	-0.0036	0.0054	-0.0028	-0.0014	0.0128	0.0283	0.0029	-0.0004	0.0223	0.0002	...
2024-01-30	-0.0192	0.0037	0.0164	-0.0231	0.0050	-0.0005	-0.0010	0.0070	-0.0056	0.0074	...
2024-01-31	-0.0194	-0.0011	-0.0167	0.0529	-0.0146	-0.0231	-0.0394	-0.0179	-0.0092	-0.0160	...
2024-02-01	0.0133	0.0328	0.0124	-0.0058	0.0246	0.0096	0.0000	0.0031	0.0105	-0.0011	...

```
vols2 = returns2.std().mul(np.sqrt(252) * 100)
means2 = returns2.mean().mul(252 * 100)

plt.scatter(
    x=vols2,
    y=means2
)

plt.xlabel('Annualized Volatility of Daily Returns (%)')
plt.ylabel('Annualized Mean of Daily Returns (%)')

# plt.xlim((0, vols2.max() + 5))
# plt.ylim((0, means2.max() + 5))

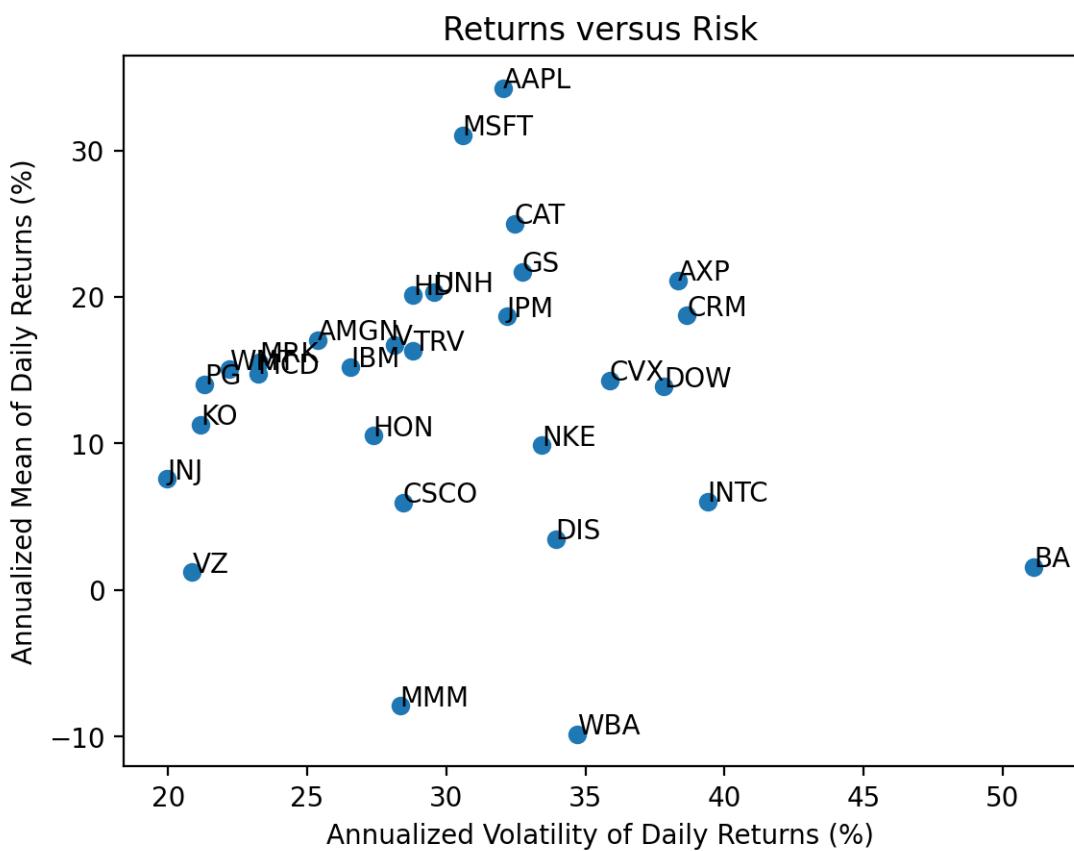
# add tickers to each point
```

```

for i in means2.index: # loop over ticker index
    plt.text( # plots string s at coordinates x and y
        x=vols2[i], # indexes volatility
        y=means2[i], # indexes mean return
        s=i # ticker index
    )

plt.title('Returns versus Risk')
plt.show() # suppresses output of last function call

```



With 30 stocks we see there is no relation between returns and volatility because most volatility is *diversifiable* and uncompensated.

17.3.6 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.

I prefer to chain these operations, with `.add(1)`, then `.prod()`, then `.sub(1)`.

```
total_returns2 = returns2.add(1).prod().sub(1)

total_returns2.iloc[:5]
```

```
AAPL    3.1210
AMGN    0.9635
AXP     0.9687
BA      -0.4289
CAT     1.6083
dtype: float64
```

17.3.7 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.

A histogram is a great way to visualize data!

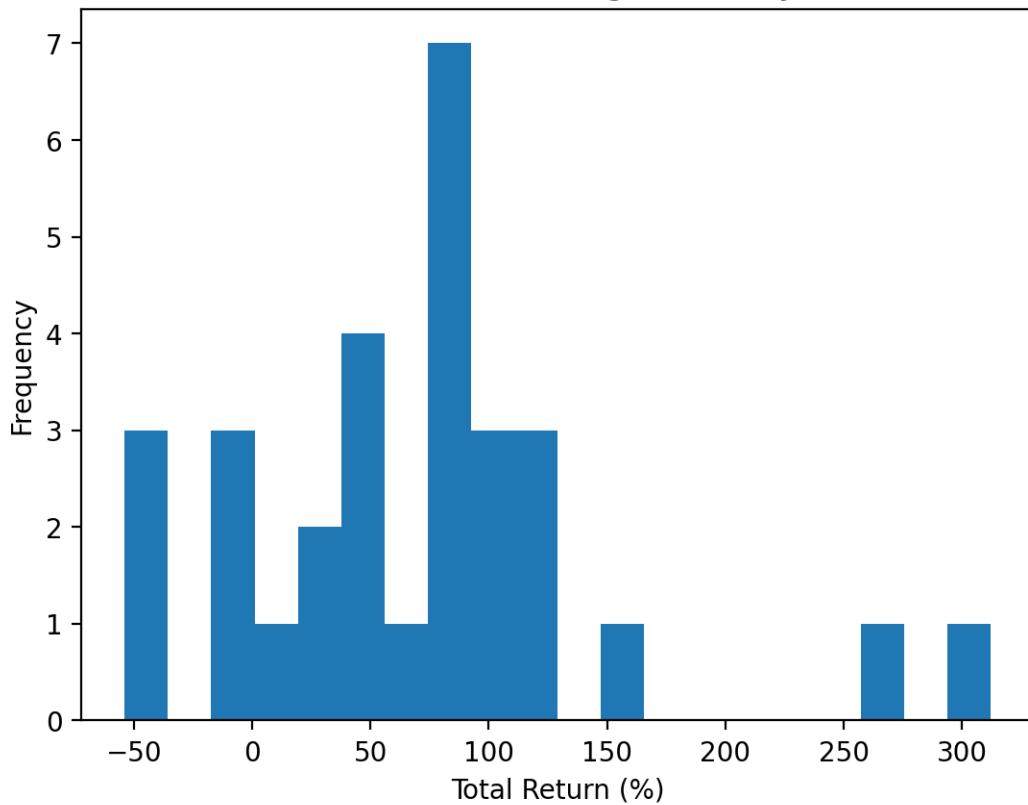
```
(

    returns2
        .add(1)
        .prod()
        .sub(1)
        .mul(100)
        .plot(kind='hist', bins=20)
)

start_date = returns2.index.min()
stop_date = returns2.index.max()

plt.xlabel('Total Return (%)')
plt.title(f'Distribution of Total Returns for DJIA Stocks\n from {start_date:%B %Y} through {stop_date:%B %Y}')
plt.show()
```

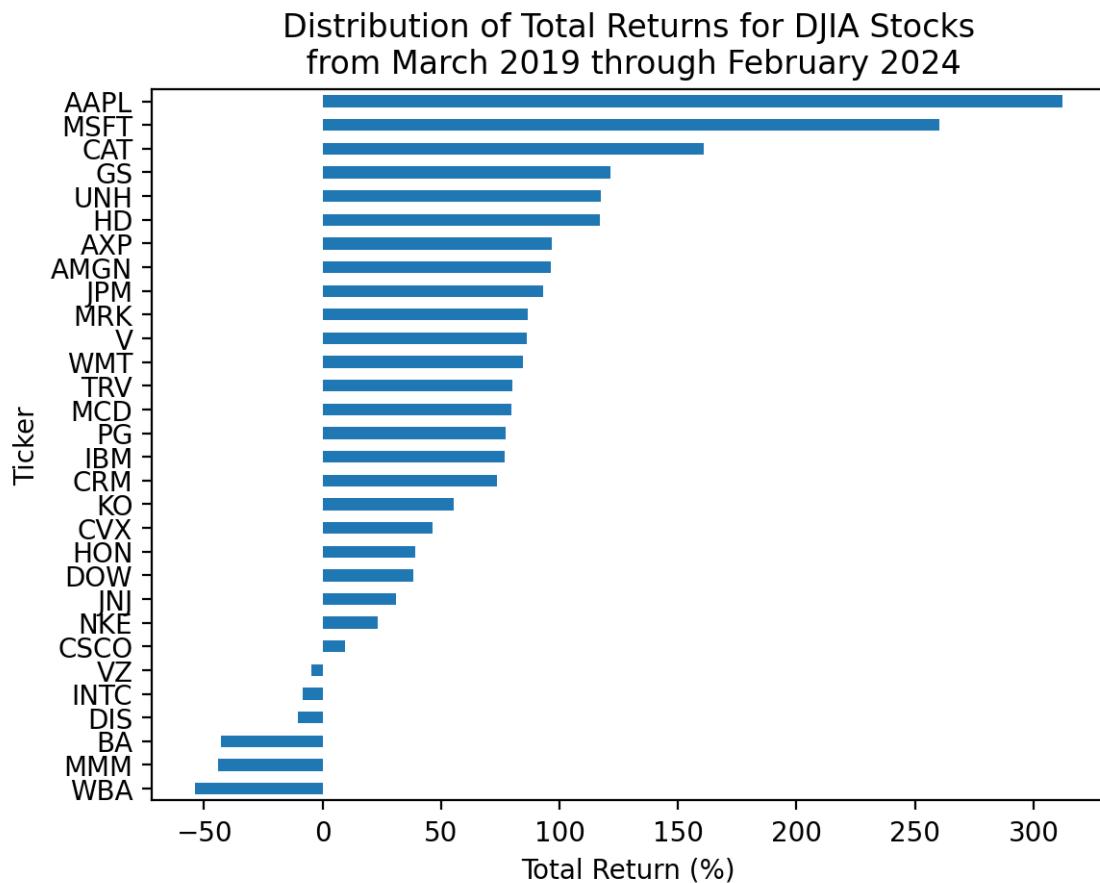
Distribution of Total Returns for DJIA Stocks
from March 2019 through February 2024



With only 30 stocks, we can actually visualize each total return!

```
(  
    returns2  
    .add(1)  
    .prod()  
    .sub(1)  
    .sort_values() # sort by total returns  
    .mul(100)  
    .plot(kind='barh') # horizontal bar chart  
)  
  
start_date = returns2.index.min()  
stop_date = returns2.index.max()
```

```
plt.xlabel('Total Return (%)')
plt.ylabel('Ticker')
plt.title(f'Distribution of Total Returns for DJIA Stocks\n from {start_date:%B %Y} through {end_date:%B %Y}')
plt.show()
```



17.3.8 Which stocks have the minimum and maximum total returns?

If we want the *values*, the `.min()` and `.max()` methods are the way to go!

```
total_returns2.min()
```

-0.5384

```
total_returns2.max()
```

```
3.1210
```

If we want the *ticker*, the `.idxmin()` and `.idxmax()` methods are the way to go!

```
total_returns2.idxmin()
```

```
'WBA'
```

```
total_returns2.idxmax()
```

```
'AAPL'
```

If we want the smallest and the largest together, we can chain a few methods!

```
total_returns2.sort_values().iloc[[0, -1]]
```

```
WBA    -0.5384
AAPL    3.1210
dtype: float64
```

Not the exactly right tool here, but the `.nsmallest()` and `.nlargest()` methods are really useful!

```
total_returns2.nsmallest(3)
```

```
WBA    -0.5384
MMM    -0.4408
BA     -0.4289
dtype: float64
```

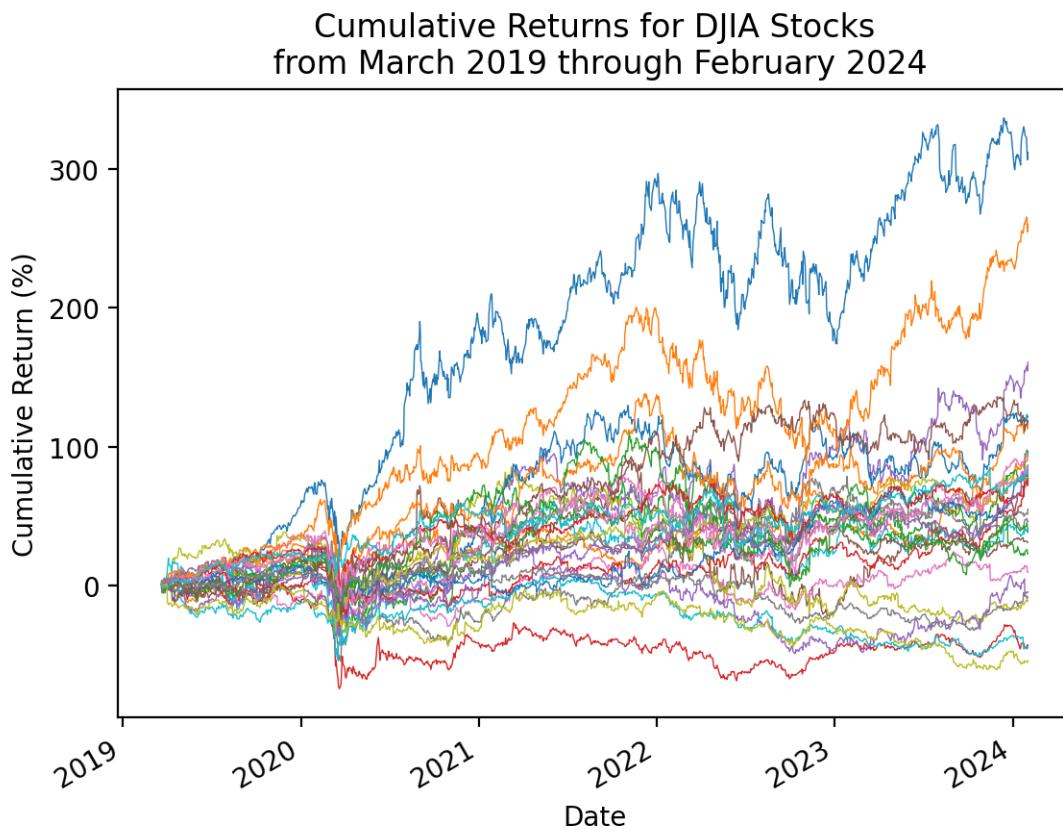
```
total_returns2.nlargest(3)
```

```
AAPL    3.1210
MSFT    2.6028
CAT     1.6083
dtype: float64
```

17.3.9 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.

```
(  
    returns2  
    .add(1)  
    .cumprod()  
    .sub(1)  
    .mul(100)  
    .plot(legend=False, linewidth=0.5) # with 30 stocks, this legend is too big to be useful  
)  
start_date = returns2.index.min()  
stop_date = returns2.index.max()  
  
plt.ylabel('Cumulative Return (%)')  
plt.title(f'Cumulative Returns for DJIA Stocks\nfrom {start_date:%B %Y} through {stop_date:  
plt.show()
```



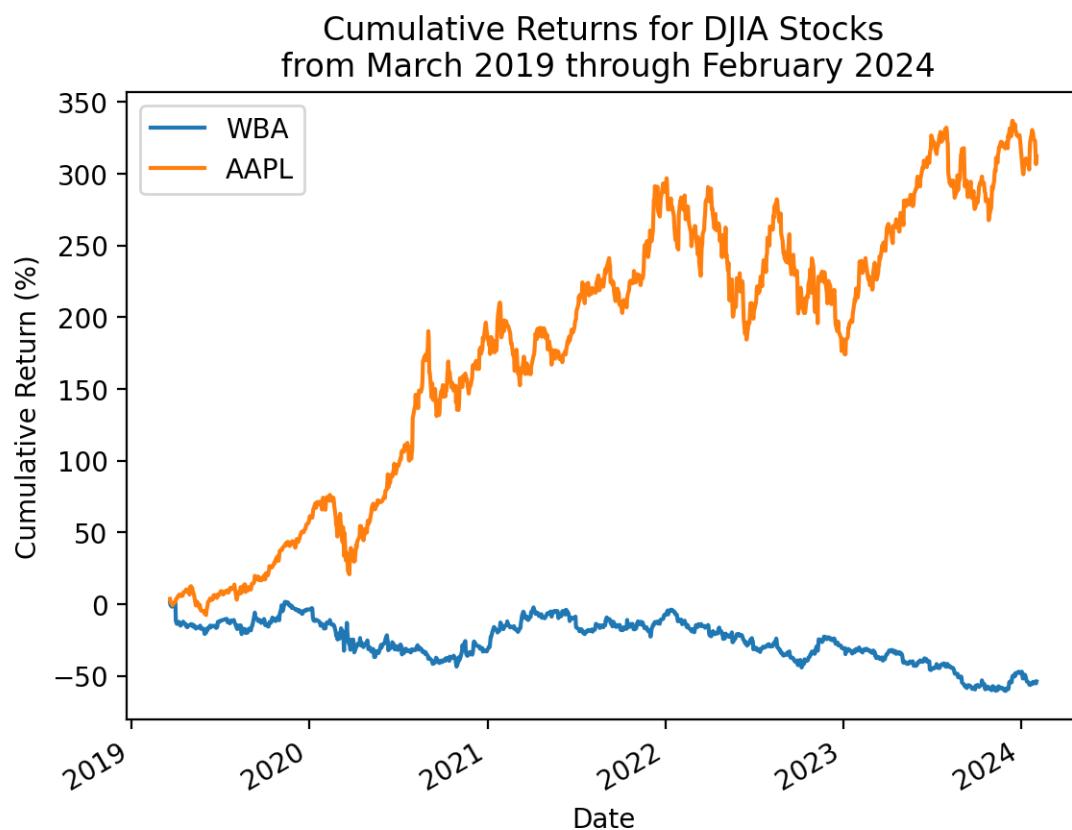
17.3.10 Repeat the plot above with only the minimum and maximum total returns

```
total_returns2.sort_values().iloc[[0, -1]].index

Index(['WBA', 'AAPL'], dtype='object')

(
    returns2 # all returns for all stocks
    [total_returns2.sort_values().iloc[[0, -1]].index] # slice min and max total return st
    .add(1)
    .cumprod()
    .sub(1)
    .mul(100)
    .plot()
```

```
)  
start_date = returns2.index.min()  
stop_date = returns2.index.max()  
  
plt.ylabel('Cumulative Return (%)')  
plt.title(f'Cumulative Returns for DJIA Stocks\nfrom {start_date} through {stop_date}')  
plt.show()
```



18 McKinney Chapter 5 - Practice for Section 03

18.1 Announcements

1. No DataCamp this week, but I suggest you keep working on it
2. Keep forming groups, and I will post our first project early next week

18.2 10-Minute Recap

```
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'
```

pandas provides two *very useful* data structures:

1. Data frames are like a worksheet in an Excel workbook (2-D, mixed data type)
2. Series are like a column in a worksheet in an Excel workbook (1-D, one data type)

```
np.random.seed(42)
df = pd.DataFrame(
    data=np.random.randn(4, 4),
    index=list('ABCD'),
    columns=list('abcd')
)
df
```

	a	b	c	d
A	0.4967	-0.1383	0.6477	1.5230
B	-0.2342	-0.2341	1.5792	0.7674
C	-0.4695	0.5426	-0.4634	-0.4657
D	0.2420	-1.9133	-1.7249	-0.5623

How can we slice the first two rows and three columns? We can slice data frames two ways:

1. Using integer locations and the `.iloc[]` method
2. Using row and column names with the `.loc[]` method

```
df.iloc[:2, :3] # slice with j,k notation, like NumPy
```

	a	b	c
A	0.4967	-0.1383	0.6477
B	-0.2342	-0.2341	1.5792

When we slice by names or labels, we get both left and right edges included!

```
df.loc['A':'B', 'a':'c']
```

	a	b	c
A	0.4967	-0.1383	0.6477
B	-0.2342	-0.2341	1.5792

We can easily add columns!

```
df['e'] = 5 # broadcasts to every row in df
```

```
df
```

	a	b	c	d	e
A	0.4967	-0.1383	0.6477	1.5230	5
B	-0.2342	-0.2341	1.5792	0.7674	5
C	-0.4695	0.5426	-0.4634	-0.4657	5
D	0.2420	-1.9133	-1.7249	-0.5623	5

```
df['e'] = np.random.randn(4)
```

```
df
```

	a	b	c	d	e
A	0.4967	-0.1383	0.6477	1.5230	-1.0128
B	-0.2342	-0.2341	1.5792	0.7674	0.3142
C	-0.4695	0.5426	-0.4634	-0.4657	-0.9080
D	0.2420	-1.9133	-1.7249	-0.5623	-1.4123

A series is the other, 1-D data structure in pandas!

```
ser = pd.Series(data=np.arange(2.), index=['C', 'D']) # the . in np.arange() makes the arr
```

```
ser
```

```
C    0.0000
D    1.0000
dtype: float64
```

```
df['f'] = ser
```

```
df
```

	a	b	c	d	e	f
A	0.4967	-0.1383	0.6477	1.5230	-1.0128	NaN
B	-0.2342	-0.2341	1.5792	0.7674	0.3142	NaN
C	-0.4695	0.5426	-0.4634	-0.4657	-0.9080	0.0000
D	0.2420	-1.9133	-1.7249	-0.5623	-1.4123	1.0000

18.3 Practice

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

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

```

returns = (
    prices['Adj Close'] # slice adj close column
    .iloc[:-1] # drop last row with intra day prices, which are sometimes missing
    .pct_change() # calculate returns
    .dropna() # drop leading rows with at least one missing value
)

returns

```

Date	AAPL	GOOG	IBM	MSFT
2004-08-20	0.0029	0.0794	0.0042	0.0030
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
...
2024-01-26	-0.0090	0.0010	-0.0158	-0.0023
2024-01-29	-0.0036	0.0068	-0.0015	0.0143
2024-01-30	-0.0192	-0.0116	0.0039	-0.0028
2024-01-31	-0.0194	-0.0735	-0.0224	-0.0269
2024-02-01	0.0133	0.0064	0.0176	0.0156

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

```
returns.mean() # default axis=0 takes the mean of each column
```

```

AAPL    0.0014
GOOG    0.0010
IBM     0.0004
MSFT    0.0008
dtype: float64

```

If we pass `axis=1`, we get the mean return on each day. We can this an “equally-weighted portfolio return.”

$$r_{p,t} = \sum_{i=0}^4 \frac{1}{4} r_{i,t}$$

```
returns.mean(axis=1)

Date
2004-08-20    0.0224
2004-08-23    0.0041
2004-08-24   -0.0032
2004-08-25    0.0152
2004-08-26    0.0146
...
2024-01-26   -0.0065
2024-01-29    0.0040
2024-01-30   -0.0074
2024-01-31   -0.0356
2024-02-01    0.0132
Length: 4896, dtype: float64
```

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

pandas calculates sample statistics by default.

```
returns.std()

AAPL    0.0206
GOOG    0.0194
IBM     0.0143
MSFT    0.0171
dtype: float64
```

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

We annualize mean returns by multiplying by T ($T = 252$ for daily returns, $T = 12$ for month returns, and so on). We annualize standard deviations by multiplying by \sqrt{T} .

```
returns.mean().mul(252)

AAPL    0.3625
GOOG    0.2552
IBM     0.0980
```

```
MSFT    0.2002
dtype: float64
```

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

```
AAPL    0.3276
GOOG    0.3074
IBM     0.2272
MSFT    0.2722
dtype: float64
```

18.3.4 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).

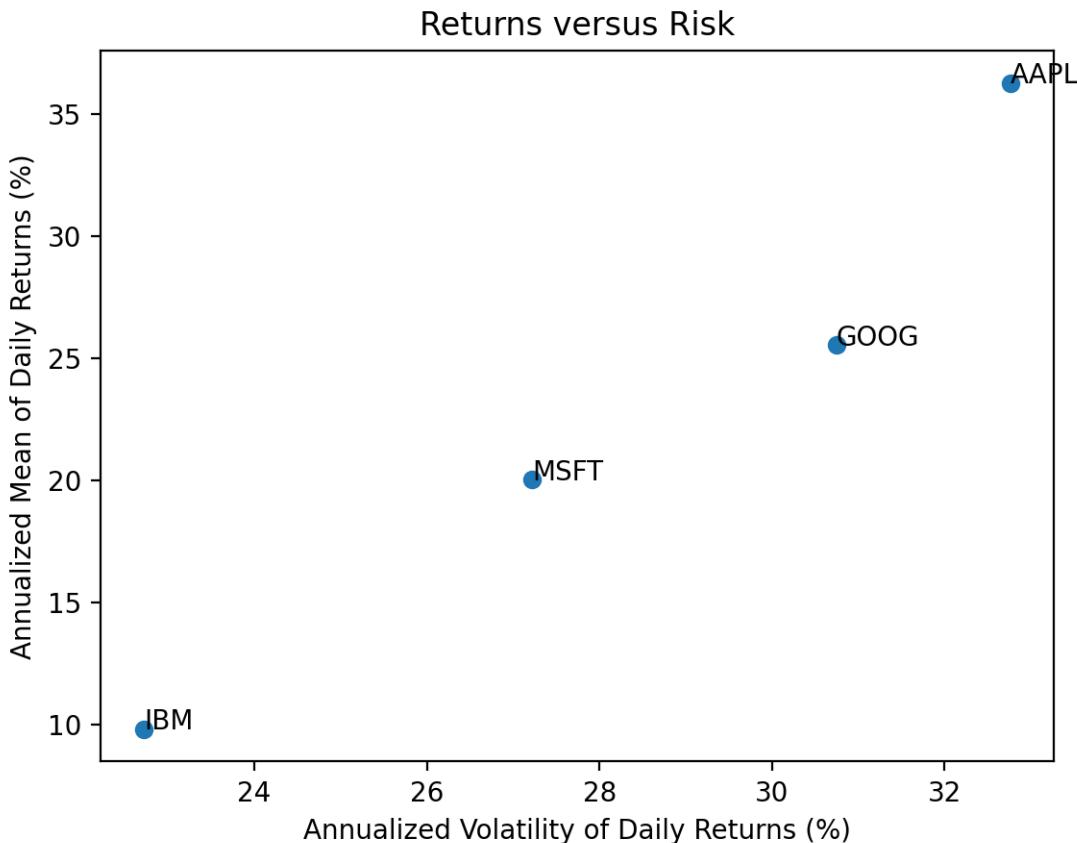
```
vols = returns.std().mul(np.sqrt(252) * 100)
means = returns.mean().mul(252 * 100)

plt.scatter(
    x=vols,
    y=means
)

# add tickers to each point
for i in means.index: # loop over ticker index
    plt.text( # plots string s at coordinates x and y
        x=vols[i], # indexes volatility
        y=means[i], # indexes mean return
        s=i # ticker index
    )

plt.xlabel('Annualized Volatility of Daily Returns (%)')
plt.ylabel('Annualized Mean of Daily Returns (%)')

plt.title('Returns versus Risk')
plt.show()
```



18.3.5 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](https://en.wikipedia.org/wiki/Dow_Jones_Industrial_Average). We will need to download new data, into `tickers2`, `prices2`, and `returns2`.

```
url2 = 'https://en.wikipedia.org/wiki/Dow_Jones_Industrial_Average'
wiki2 = pd.read_html(io=url2)
```

```
tickers2 = wiki2[1]['Symbol'].to_list()
```

```
tickers2[:5]
```

```
['MMM', 'AXP', 'AMGN', 'AAPL', 'BA']
```

```
prices2 = yf.download(tickers=tickers2)
```

```
[*****100%*****] 30 of 30 completed
```

```
returns2 = (
    prices2['Adj Close'] # slide adj close for all stocks
    .iloc[:-1] # drop last day of adj close, which are intraday values before 4 PM
    .pct_change() # calculate returns
    .dropna() # drop any dates with incomplete data
)

returns2
```

Date	AAPL	AMGN	AXP	BA	CAT	CRM	CSCO	CVX	DIS	DOW	...
2019-03-21	0.0368	0.0040	0.0095	-0.0092	0.0079	0.0210	0.0128	0.0094	-0.0121	-0.0165	...
2019-03-22	-0.0207	-0.0270	-0.0211	-0.0283	-0.0320	-0.0326	-0.0222	-0.0220	-0.0040	-0.0078	...
2019-03-25	-0.0121	-0.0006	-0.0038	0.0229	0.0124	-0.0038	-0.0002	-0.0016	-0.0041	0.0113	...
2019-03-26	-0.0103	0.0090	0.0042	-0.0002	0.0035	-0.0092	0.0095	0.0101	0.0218	-0.0061	...
2019-03-27	0.0090	-0.0104	-0.0047	0.0103	-0.0049	-0.0269	-0.0017	-0.0108	0.0013	0.0256	...
...
2024-01-26	-0.0090	0.0049	0.0710	0.0178	-0.0045	0.0033	-0.0036	0.0038	0.0053	-0.0160	...
2024-01-29	-0.0036	0.0054	-0.0028	-0.0014	0.0128	0.0283	0.0029	-0.0004	0.0223	0.0002	...
2024-01-30	-0.0192	0.0037	0.0164	-0.0231	0.0050	-0.0005	-0.0010	0.0070	-0.0056	0.0074	...
2024-01-31	-0.0194	-0.0011	-0.0167	0.0529	-0.0146	-0.0231	-0.0394	-0.0179	-0.0092	-0.0160	...
2024-02-01	0.0133	0.0328	0.0124	-0.0058	0.0246	0.0096	0.0000	0.0031	0.0105	-0.0011	...

```
vols = returns2.std().mul(np.sqrt(252) * 100)
means = returns2.mean().mul(252 * 100)

plt.scatter(
    x=vols,
    y=means
)

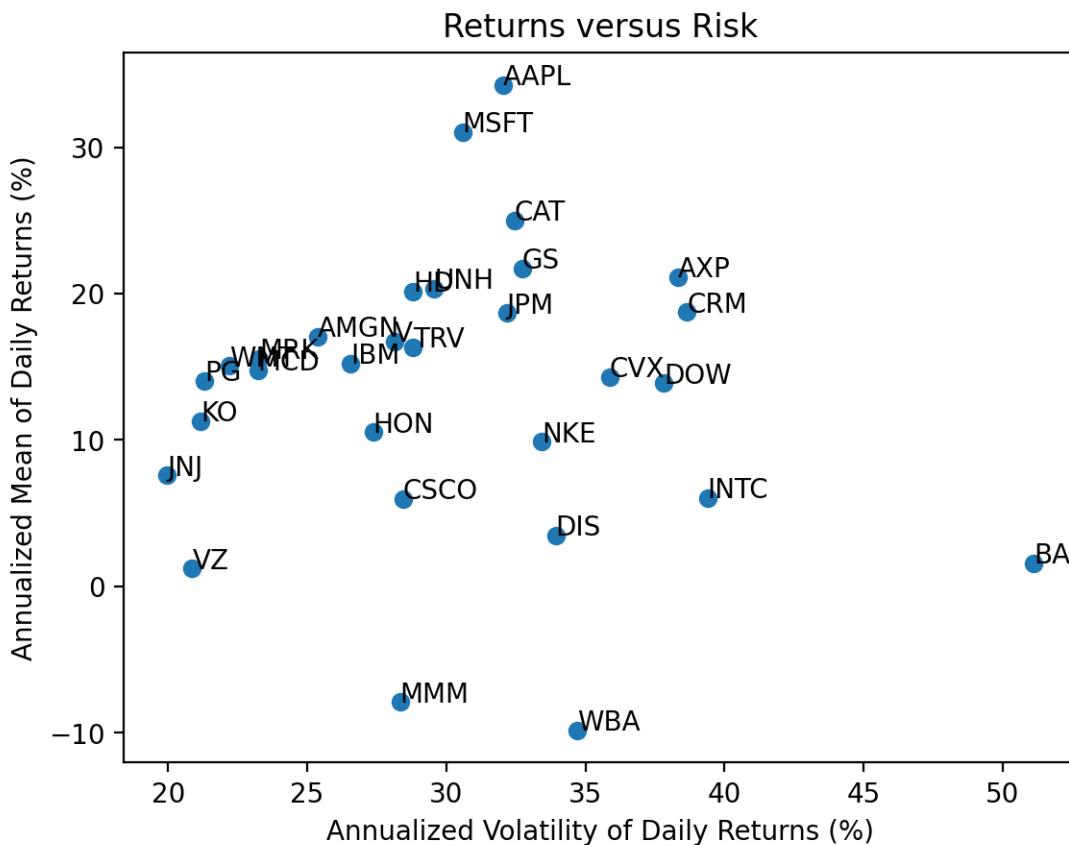
# add tickers to each point
for i in means.index: # loop over ticker index
    plt.text( # plots string s at coordinates x and y
        x=vols[i], # indexes volatility
        y=means[i], # indexes mean return
        s=i # ticker index
    )
```

```

plt.xlabel('Annualized Volatility of Daily Returns (%)')
plt.ylabel('Annualized Mean of Daily Returns (%)')

plt.title('Returns versus Risk')
plt.show()

```



18.3.6 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.

```

total_returns2 = returns2.add(1).prod().sub(1)

total_returns2.iloc[:5]

```

```
AAPL    3.1210
AMGN    0.9635
AXP     0.9687
BA     -0.4289
CAT     1.6083
dtype: float64
```

```
np.allclose(((returns2 + 1).prod() - 1), total_returns2)
```

```
True
```

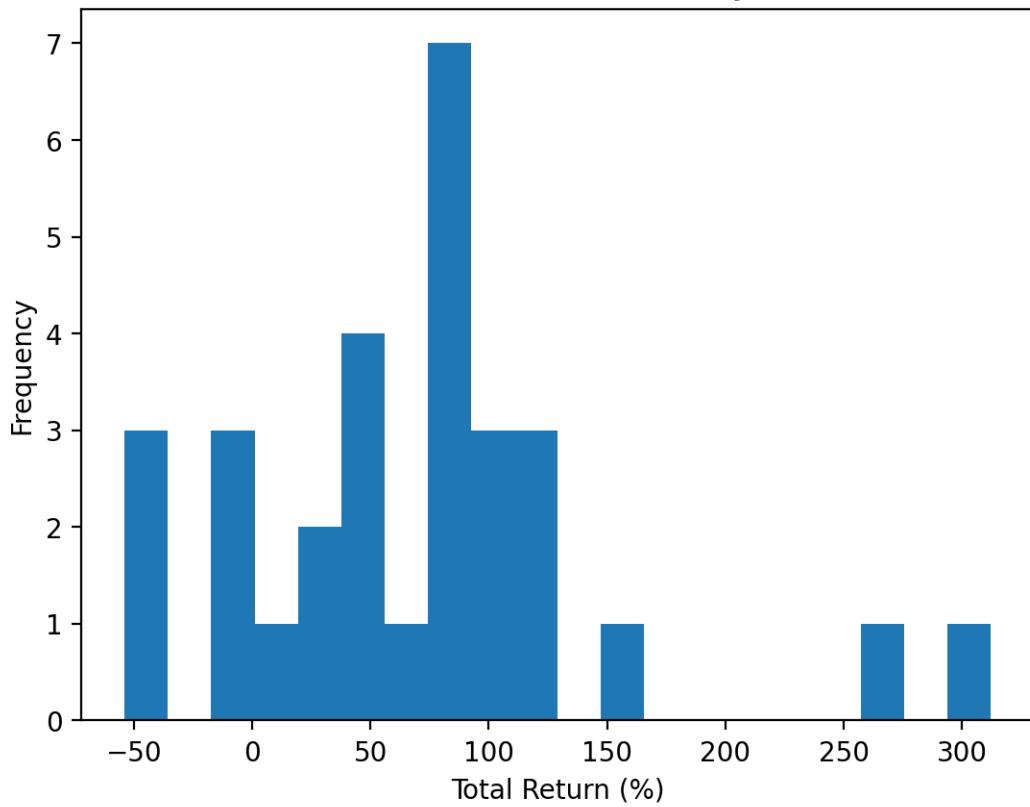
18.3.7 Plot the distribution of total returns for the stocks in the DJIA

We can plot a histogram, using either the `plt.hist()` function or the `.plot(kind='hist')` method.

```
start_date = returns2.index.min()
stop_date = returns2.index.max()

(
    returns2
    .add(1)
    .prod()
    .sub(1)
    .mul(100)
    .plot(kind='hist', bins=20)
)
plt.xlabel('Total Return (%)')
plt.title(f'Distribution of Total Returns for DJIA Stocks\n from {start_date:%B %Y} to {stop_date:%B %Y}')
plt.show()
```

Distribution of Total Returns for DJIA Stocks
from March 2019 to February 2024

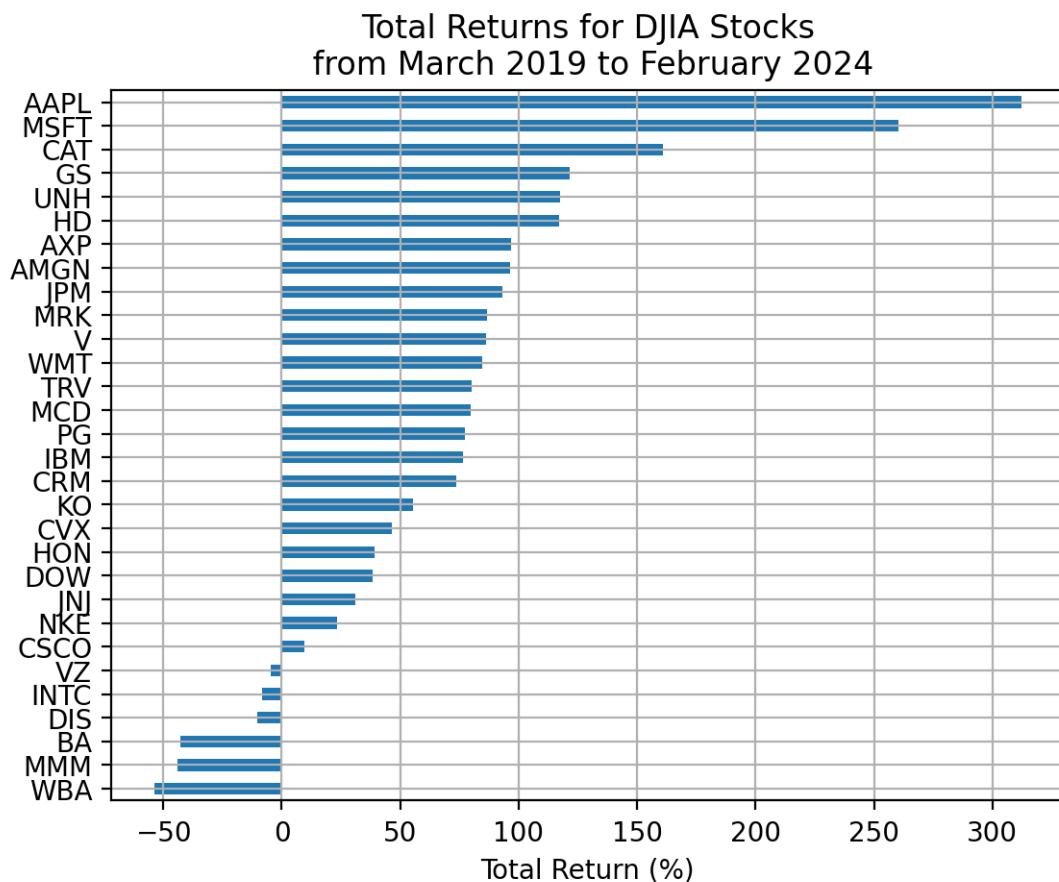


With only 30 stocks, we can visualize and interpret each stock separately!

```
start_date = returns2.index.min()
stop_date = returns2.index.max()

(
    returns2
    .add(1)
    .prod()
    .sub(1)
    .mul(100)
    .sort_values()
    .plot(kind='barh', grid=True)
)
plt.xlabel('Total Return (%)')
```

```
plt.title(f'Total Returns for DJIA Stocks\n from {start_date:%B %Y} to {stop_date:%B %Y}')
plt.show()
```



18.3.8 Which stocks have the minimum and maximum total returns?

If we want the *values*, the `.min()` and `.max()` methods are the way to go!

```
total_returns2.min()
```

-0.5384

```
total_returns2.max()
```

```
3.1210
```

The `.min()` and `.max()` methods give the values but not the tickers (or index). We use the `.idxmin()` and `.idxmax()` to get the tickers (or index).

```
total_returns2.idxmin()
```

```
'WBA'
```

```
total_returns2.idxmax()
```

```
'AAPL'
```

Here is what I would use!

```
total_returns2.sort_values().iloc[[0, -1]]
```

```
WBA    -0.5384
AAPL    3.1210
dtype: float64
```

Not the exactly right tool here, but the `.nsmallest()` and `.nlargest()` methods are really useful!

```
total_returns2.nsmallest(3)
```

```
WBA    -0.5384
MMM    -0.4408
BA     -0.4289
dtype: float64
```

```
total_returns2.nlargest(3)
```

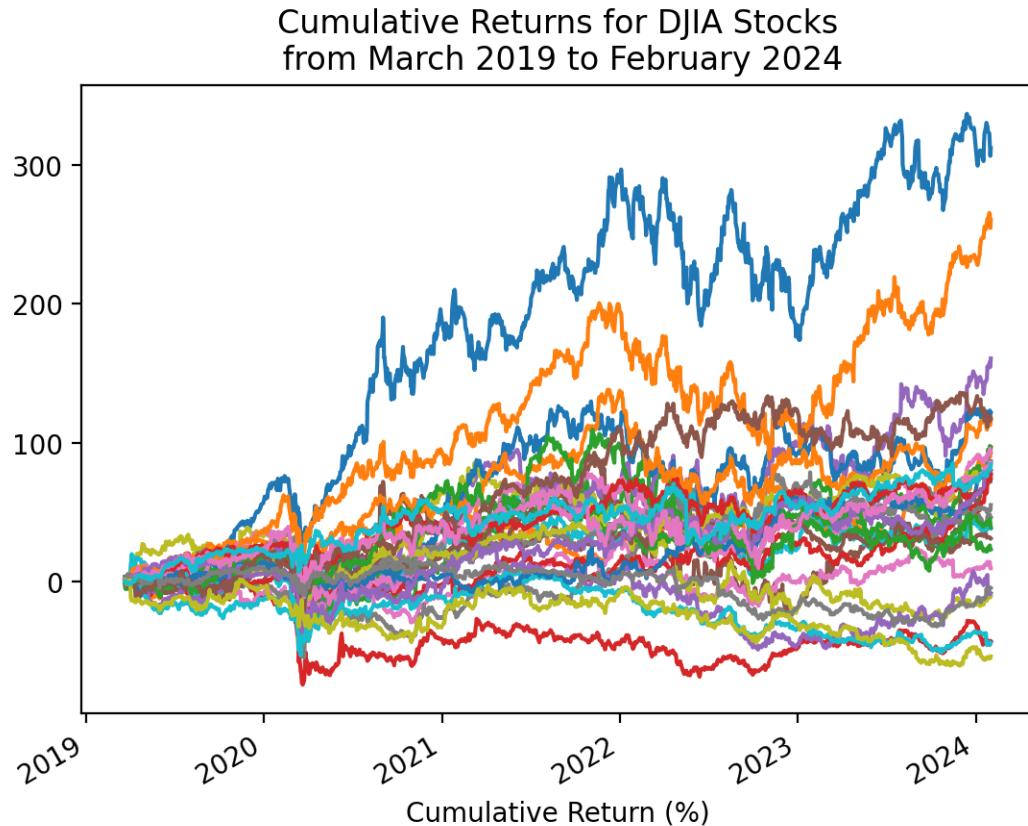
```
AAPL    3.1210
MSFT    2.6028
CAT     1.6083
dtype: float64
```

18.3.9 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.

```
start_date = returns2.index.min()
stop_date = returns2.index.max()

(
    returns2
    .add(1)
    .cumprod()
    .sub(1)
    .mul(100)
    .plot(legend=False)
)
plt.xlabel('Cumulative Return (%)')
plt.title(f'Cumulative Returns for DJIA Stocks\n from {start_date:%B %Y} to {stop_date:%B %Y}')
plt.show()
```



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

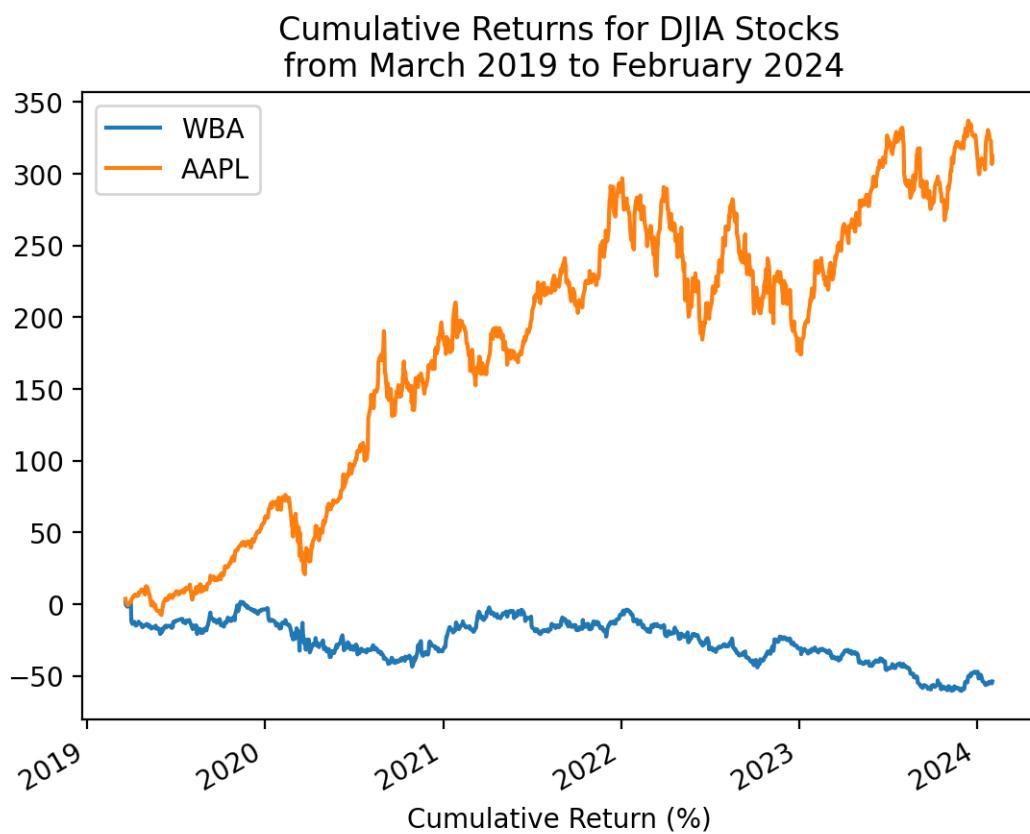
```
total_returns2.sort_values().iloc[[0, -1]].index

Index(['WBA', 'AAPL'], dtype='object')

start_date = returns2.index.min()
stop_date = returns2.index.max()

(
    returns2[total_returns2.sort_values().iloc[[0, -1]].index]
    .add(1)
    .cumprod()
    .sub(1)
```

```
.mul(100)
.plot()
)
plt.xlabel('Cumulative Return (%)')
plt.title(f'Cumulative Returns for DJIA Stocks\n from {start_date:%B %Y} to {stop_date:%B %Y}')
plt.show()
```



19 McKinney Chapter 5 - Practice for Section 04

19.1 Announcements

1. No DataCamp this week, but I suggest you keep working on it
2. Keep forming groups, and I will post our first project early next week

19.2 10-Minute Recap

```
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'
```

pandas provides two data structures:

1. Data frame is a 2-D, mixed data type structure, like a worksheet in an Excel workbook
2. Series is a 1-D, one data type structure, like a column in a worksheet (or data frame)

```
np.random.seed(42)
df = pd.DataFrame(
    data=np.random.randn(3, 4), # 12 random numbers
    index=list('ABC'), # labels the rows for easy indexing and slicing
    columns=list('abcd') # labels the columns for easy indexing and slicing
)
df
```

	a	b	c	d
A	0.4967	-0.1383	0.6477	1.5230
B	-0.2342	-0.2341	1.5792	0.7674
C	-0.4695	0.5426	-0.4634	-0.4657

How do we index or slice data frames?

1. With integer locations and the `.iloc[]` method
2. With row and column names and the `.loc[]` method

Say we want the first two rows and first three columns.

```
df.iloc[:2, :3] # j,k slicing, as in NumPy
```

	a	b	c
A	0.4967	-0.1383	0.6477
B	-0.2342	-0.2341	1.5792

```
df.loc[['A', 'B'], ['a', 'b', 'c']]
```

	a	b	c
A	0.4967	-0.1383	0.6477
B	-0.2342	-0.2341	1.5792

Both left and right edges of named slices are included in pandas!

```
df.loc['A':'B', 'a':'c']
```

	a	b	c
A	0.4967	-0.1383	0.6477
B	-0.2342	-0.2341	1.5792

How do I add a column?

```
df['e'] = 5 # pandas broadcasts this 5 to all rows
```

```
df
```

	a	b	c	d	e
A	0.4967	-0.1383	0.6477	1.5230	5
B	-0.2342	-0.2341	1.5792	0.7674	5
C	-0.4695	0.5426	-0.4634	-0.4657	5

A series is the other data structure.

```
ser = pd.Series(data=np.arange(2.), index=list('BC'))
```

```
ser
```

```
B    0.0000
C    1.0000
dtype: float64
```

```
df['f'] = ser
```

```
df
```

	a	b	c	d	e	f
A	0.4967	-0.1383	0.6477	1.5230	5	NaN
B	-0.2342	-0.2341	1.5792	0.7674	5	0.0000
C	-0.4695	0.5426	-0.4634	-0.4657	5	1.0000

19.3 Practice

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

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

```

returns = (
    prices['Adj Close'] # slices the adj close columns
    .iloc[:-1] # drop last date with intraday price
    .pct_change() # calculate returns
    .dropna() # drop dates with incomplete returns data
)

```

`returns`

Date	AAPL	GOOG	IBM	MSFT
2004-08-20	0.0029	0.0794	0.0042	0.0030
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
...
2024-01-26	-0.0090	0.0010	-0.0158	-0.0023
2024-01-29	-0.0036	0.0068	-0.0015	0.0143
2024-01-30	-0.0192	-0.0116	0.0039	-0.0028
2024-01-31	-0.0194	-0.0735	-0.0224	-0.0269
2024-02-01	0.0133	0.0064	0.0176	0.0156

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

```

returns.mean() # default is axis=0

AAPL    0.0014
GOOG    0.0010
IBM     0.0004
MSFT    0.0008
dtype: float64

```

If we use `.mean(axis=1)` on stock returns, we get the equally-weighted portfolio returns on each day.

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

```
Date
2004-08-20    0.0224
2004-08-23    0.0041
2004-08-24   -0.0032
2004-08-25    0.0152
2004-08-26    0.0146
...
2024-01-26   -0.0065
2024-01-29    0.0040
2024-01-30   -0.0074
2024-01-31   -0.0356
2024-02-01    0.0132
Length: 4896, dtype: float64
```

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

pandas methods give us *sample* statistics, instead of population statistics in NumPy.

```
returns.std()
```

```
AAPL    0.0206
GOOG    0.0194
IBM     0.0143
MSFT    0.0171
dtype: float64
```

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

We annualize mean returns by multiplying by T ($T = 252$ for daily returns, $T = 12$ for month returns, and so on). We annualize standard deviations by multiplying by \sqrt{T} .

```
returns.mean().mul(252)
```

```
AAPL    0.3625
GOOG    0.2552
IBM     0.0980
MSFT    0.2002
dtype: float64
```

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

```
AAPL    0.3276
GOOG    0.3074
IBM     0.2272
MSFT    0.2722
dtype: float64
```

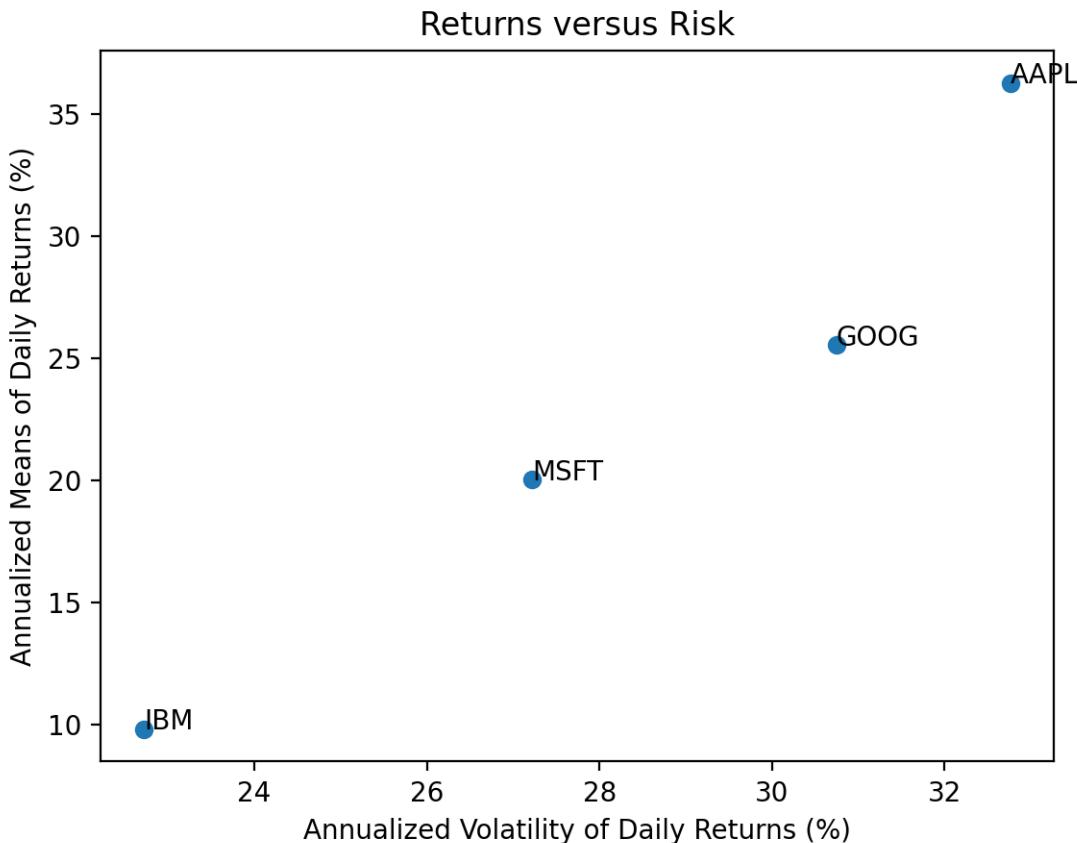
19.3.4 Plot *annualized* means versus standard deviations of daily returns for these four stocks

```
means = returns.mean().mul(252 * 100)
vols = returns.std().mul(np.sqrt(252) * 100)

plt.scatter(
    x=vols,
    y=means
)

# add tickers to each point
for i in means.index: # loop over ticker index
    plt.text( # plots string s at coordinates x and y
        x=vols[i], # indexes volatility
        y=means[i], # indexes mean return
        s=i # ticker index
    )

plt.xlabel('Annualized Volatility of Daily Returns (%)')
plt.ylabel('Annualized Means of Daily Returns (%)')
plt.title('Returns versus Risk')
plt.show()
```



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

19.3.5 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](https://en.wikipedia.org/wiki/Dow_Jones_Industrial_Average). We will need to download new data, into `tickers2`, `prices2`, and `returns2`.

```
url2 = 'https://en.wikipedia.org/wiki/Dow_Jones_Industrial_Average'
wiki2 = pd.read_html(url2)
tickers2 = wiki2[1]['Symbol'].to_list()
tickers2[:5]
```

```
['MMM', 'AXP', 'AMGN', 'AAPL', 'BA']
```

```
prices2 = yf.download(tickers=tickers2)
```

```
[*****100%*****] 30 of 30 completed
```

```
returns2 = (
    prices2['Adj Close']
    .iloc[:-1]
    .pct_change()
    .dropna()
)

returns2
```

Date	AAPL	AMGN	AXP	BA	CAT	CRM	CSCO	CVX	DIS	DOW	...
2019-03-21	0.0368	0.0040	0.0095	-0.0092	0.0079	0.0210	0.0128	0.0094	-0.0121	-0.0165	...
2019-03-22	-0.0207	-0.0270	-0.0211	-0.0283	-0.0320	-0.0326	-0.0222	-0.0220	-0.0040	-0.0078	...
2019-03-25	-0.0121	-0.0006	-0.0038	0.0229	0.0124	-0.0038	-0.0002	-0.0016	-0.0041	0.0113	...
2019-03-26	-0.0103	0.0090	0.0042	-0.0002	0.0035	-0.0092	0.0095	0.0101	0.0218	-0.0061	...
2019-03-27	0.0090	-0.0104	-0.0047	0.0103	-0.0049	-0.0269	-0.0017	-0.0108	0.0013	0.0256	...
...
2024-01-26	-0.0090	0.0049	0.0710	0.0178	-0.0045	0.0033	-0.0036	0.0038	0.0053	-0.0160	...
2024-01-29	-0.0036	0.0054	-0.0028	-0.0014	0.0128	0.0283	0.0029	-0.0004	0.0223	0.0002	...
2024-01-30	-0.0192	0.0037	0.0164	-0.0231	0.0050	-0.0005	-0.0010	0.0070	-0.0056	0.0074	...
2024-01-31	-0.0194	-0.0011	-0.0167	0.0529	-0.0146	-0.0231	-0.0394	-0.0179	-0.0092	-0.0160	...
2024-02-01	0.0133	0.0328	0.0124	-0.0058	0.0246	0.0096	0.0000	0.0031	0.0105	-0.0011	...

```
means = returns2.mean().mul(252 * 100)
vols = returns2.std().mul(np.sqrt(252) * 100)

plt.scatter(
    x=vols,
    y=means
)

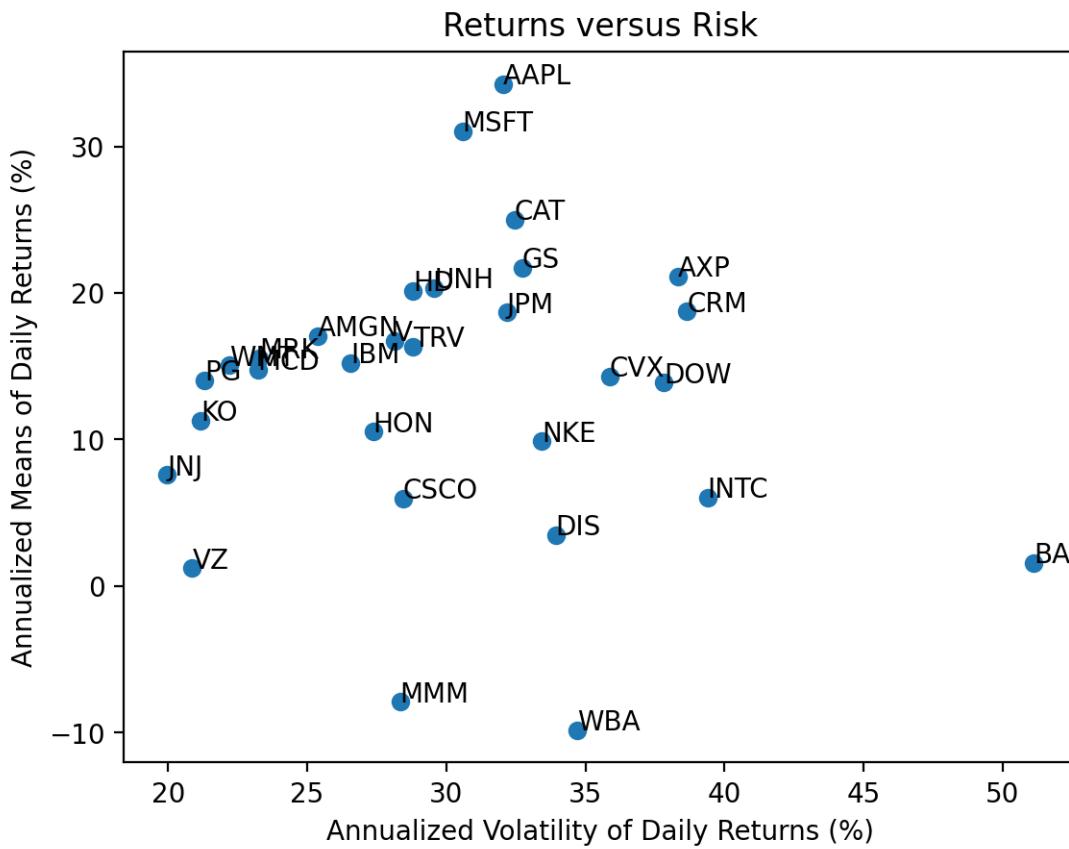
# add tickers to each point
for i in means.index: # loop over ticker index
    plt.text( # plots string s at coordinates x and y
        x=vols[i], # indexes volatility
```

```

y=means[i], # indexes mean return
s=i # ticker index
)

plt.xlabel('Annualized Volatility of Daily Returns (%)')
plt.ylabel('Annualized Means of Daily Returns (%)')
plt.title('Returns versus Risk')
plt.show()

```



19.3.6 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.

```
total_returns2 = returns2.add(1).prod().sub(1)
total_returns2.iloc[:5]
```

```
AAPL    3.1210
AMGN    0.9635
AXP     0.9687
BA     -0.4289
CAT     1.6083
dtype: float64
```

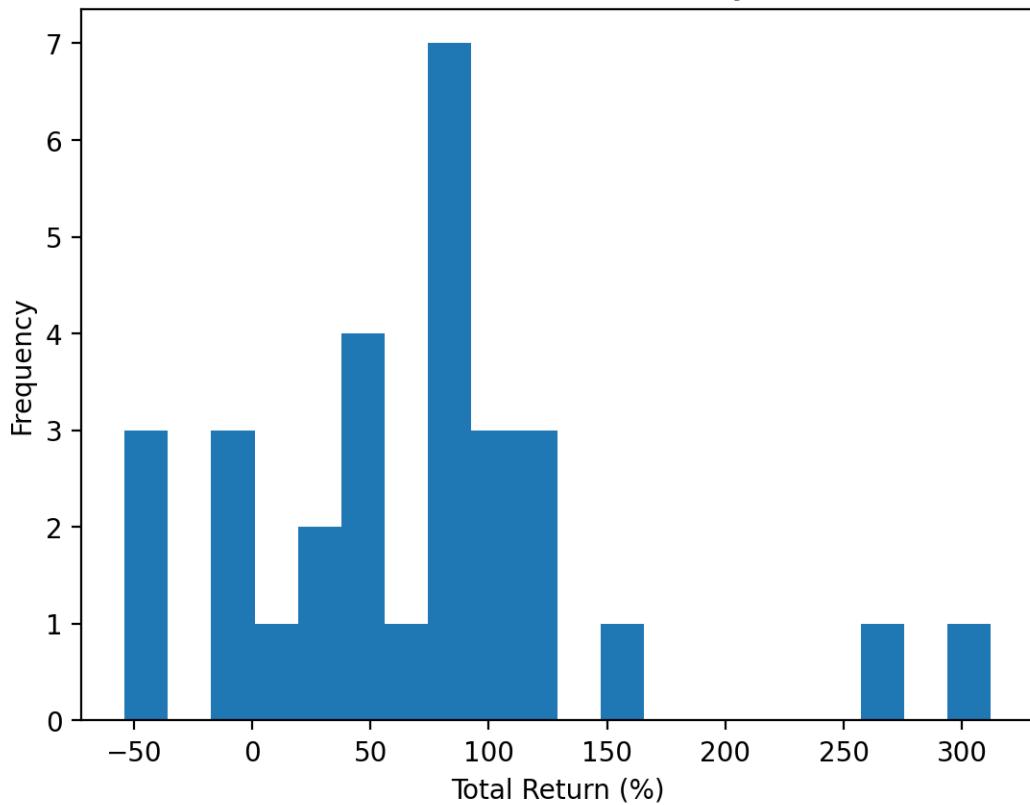
19.3.7 Plot the distribution of total returns for the stocks in the DJIA

We can plot a histogram, using either the `plt.hist()` function or the `.plot(kind='hist')` method.

```
start_date = returns2.index.min()
stop_date = returns2.index.max()

(
    returns2
    .add(1)
    .prod()
    .sub(1)
    .mul(100)
    .plot(kind='hist', bins=20)
)
plt.xlabel('Total Return (%)')
plt.title(f'Distribution of Total Returns for DJIA Stocks\n from {start_date:%B %Y} to {stop_date:%B %Y}')
plt.show()
```

Distribution of Total Returns for DJIA Stocks
from March 2019 to February 2024

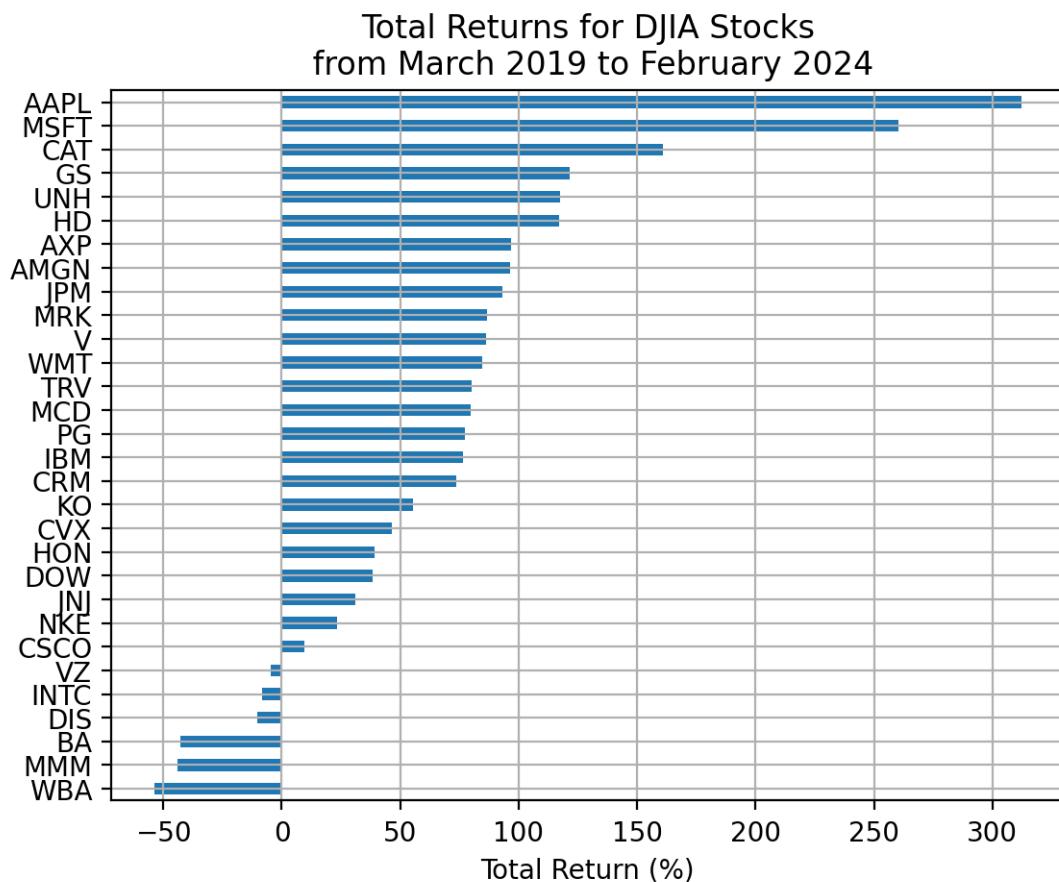


With only 30 stocks, we can visualize and interpret each stock separately!

```
start_date = returns2.index.min()
stop_date = returns2.index.max()

(
    returns2
    .add(1)
    .prod()
    .sub(1)
    .mul(100)
    .sort_values()
    .plot(kind='barh', grid=True)
)
plt.xlabel('Total Return (%)')
```

```
plt.title(f'Total Returns for DJIA Stocks\n from {start_date:%B %Y} to {stop_date:%B %Y}')
plt.show()
```



19.3.8 Which stocks have the minimum and maximum total returns?

If we want the *values*, the `.min()` and `.max()` methods are the way to go!

```
total_returns2.min()
```

-0.5384

```
total_returns2.max()
```

```
3.1210
```

The `.min()` and `.max()` methods give the values but not the tickers (or index). We use the `.idxmin()` and `.idxmax()` to get the tickers (or index).

```
total_returns2.idxmin()
```

```
'WBA'
```

```
total_returns2.idxmax()
```

```
'AAPL'
```

Here is what I would use!

```
total_returns2.sort_values().iloc[[0, -1]]
```

```
WBA    -0.5384
AAPL    3.1210
dtype: float64
```

Not the exactly right tool here, but the `.nsmallest()` and `.nlargest()` methods are really useful!

```
total_returns2.nsmallest(3)
```

```
WBA    -0.5384
MMM    -0.4408
BA     -0.4289
dtype: float64
```

```
total_returns2.nlargest(3)
```

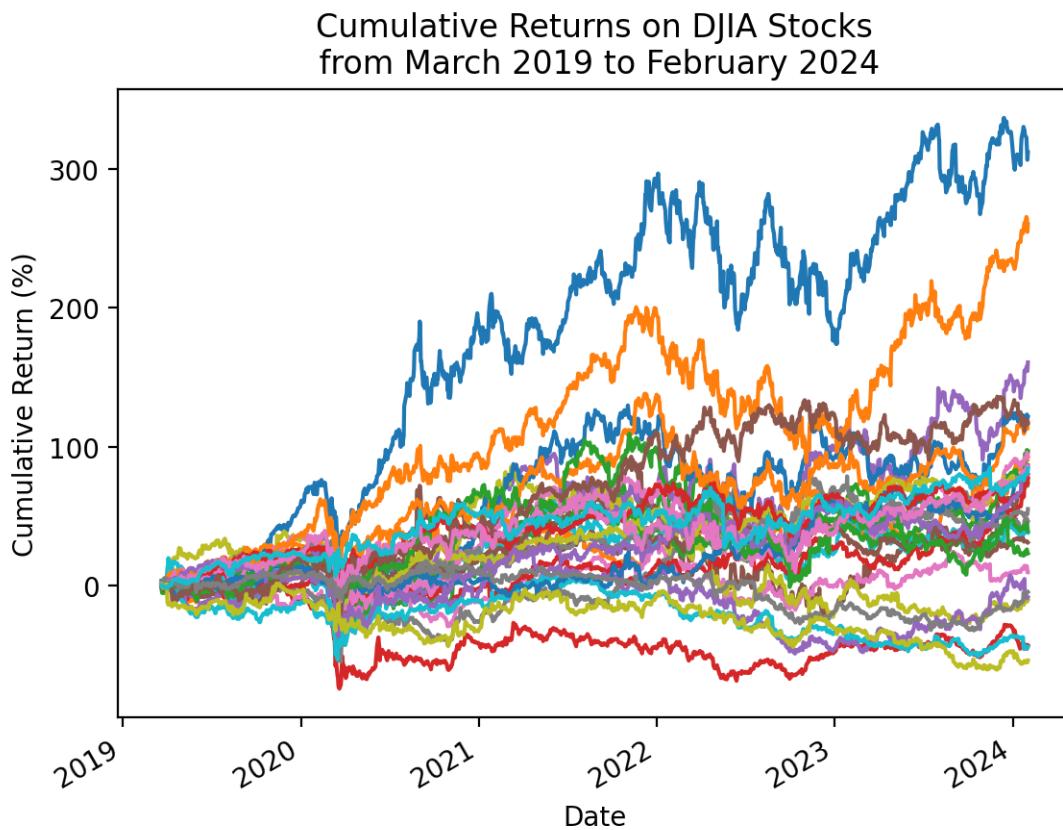
```
AAPL    3.1210
MSFT    2.6028
CAT     1.6083
dtype: float64
```

19.3.9 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.

```
start_date = returns2.index.min()
stop_date = returns2.index.max()

(
    returns2
    .add(1)
    .cumprod()
    .sub(1)
    .mul(100)
    .plot(legend=False)
)
plt.ylabel('Cumulative Return (%)')
plt.title(f'Cumulative Returns on DJIA Stocks\n from {start_date:%B %Y} to {stop_date:%B %Y}')
plt.show()
```



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

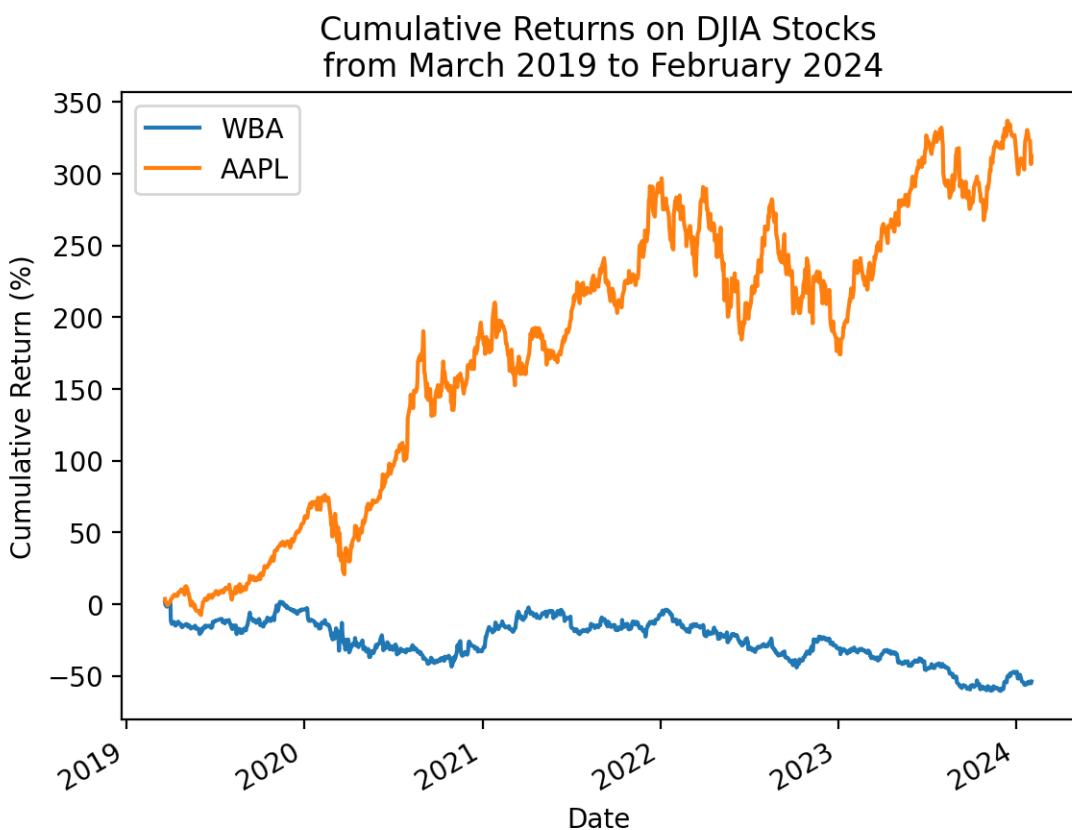
```
total_returns2.sort_values().iloc[[0, -1]].index

Index(['WBA', 'AAPL'], dtype='object')

start_date = returns2.index.min()
stop_date = returns2.index.max()

(
    returns2[total_returns2.sort_values().iloc[[0, -1]].index]
    .add(1)
    .cumprod()
    .sub(1)
```

```
.mul(100)
.plot()
)
plt.ylabel('Cumulative Return (%)')
plt.title(f'Cumulative Returns on DJIA Stocks\n from {start_date:%B %Y} to {stop_date:%B %Y}')
plt.show()
```



20 McKinney Chapter 5 - Practice for Section 05

20.1 Announcements

1. No DataCamp this week, but I suggest you keep working on it
2. Keep forming groups, and I will post our first project early next week

20.2 10-Minute Recap

```
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'
```

pandas gives us two data structures:

1. Data Frames
2. Series

A data frame is like a worksheet in an Excel workbook.

```
np.random.seed(42)
df = pd.DataFrame(
    data=np.random.randn(3, 5), # 15 random numbers
    index=list('ABC'), # labels the rows for easy indexing and slicing
    columns=list('abcde') # labels the columns for easy indexing and slicing
)
```

```
df
```

	a	b	c	d	e
A	0.4967	-0.1383	0.6477	1.5230	-0.2342
B	-0.2341	1.5792	0.7674	-0.4695	0.5426
C	-0.4634	-0.4657	0.2420	-1.9133	-1.7249

How can we get the first two rows and first three columns?

1. We can slice by integer location with the `.iloc[]` method
2. We can slice by names with the `.loc[]` method

```
df.iloc[:2, :3] # j,k slicing, as in NumPy
```

	a	b	c
A	0.4967	-0.1383	0.6477
B	-0.2341	1.5792	0.7674

When we slice by names, both left and right edges are included!

```
df.loc['A':'B', 'a':'c']
```

	a	b	c
A	0.4967	-0.1383	0.6477
B	-0.2341	1.5792	0.7674

We can use the `.head()` method to show the first n rows in df.

```
df.head(2)
```

	a	b	c	d	e
A	0.4967	-0.1383	0.6477	1.5230	-0.2342
B	-0.2341	1.5792	0.7674	-0.4695	0.5426

How do I add a column?

```
df['f'] = 5 # as in NumPy, this 5 will broadcast to all the rows
```

```
df
```

	a	b	c	d	e	f
A	0.4967	-0.1383	0.6477	1.5230	-0.2342	5
B	-0.2341	1.5792	0.7674	-0.4695	0.5426	5
C	-0.4634	-0.4657	0.2420	-1.9133	-1.7249	5

A series is like a single column in an Excel worksheet (or a pandas data frame).

```
ser = pd.Series(data=np.arange(2.), index=list('BC'))
```

```
ser
```

```
B    0.0000
C    1.0000
dtype: float64
```

pandas aligns operations on row and column names

```
df['g'] = ser
```

```
df
```

	a	b	c	d	e	f	g
A	0.4967	-0.1383	0.6477	1.5230	-0.2342	5	NaN
B	-0.2341	1.5792	0.7674	-0.4695	0.5426	5	0.0000
C	-0.4634	-0.4657	0.2420	-1.9133	-1.7249	5	1.0000

20.3 Practice

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

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

```

returns = (
    prices['Adj Close'] # slice adj close column
    .iloc[:-1] # drop last row with intra day prices, which are sometimes missing
    .pct_change() # calculate returns
    .dropna() # drop leading rows with at least one missing value
)

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
...
2024-01-26	-0.0090	0.0010	-0.0158	-0.0023
2024-01-29	-0.0036	0.0068	-0.0015	0.0143
2024-01-30	-0.0192	-0.0116	0.0039	-0.0028
2024-01-31	-0.0194	-0.0735	-0.0224	-0.0269
2024-02-01	0.0133	0.0064	0.0176	0.0156

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

```
returns.mean() # default is axis=0, so we get the mean of each column
```

```

AAPL    0.0014
GOOG    0.0010
IBM     0.0004
MSFT    0.0008
dtype: float64

```

We if want an equally-weighted portfolio return? We could take the mean of each *row* with `.mean(axis=1)`. The mean is the same as the sum of 0.25 times each of the 4 columns.

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

```
Date
2004-08-20    0.0224
2004-08-23    0.0041
2004-08-24   -0.0032
2004-08-25    0.0152
2004-08-26    0.0146
...
2024-01-26   -0.0065
2024-01-29    0.0040
2024-01-30   -0.0074
2024-01-31   -0.0356
2024-02-01    0.0132
Length: 4896, dtype: float64
```

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

We can use the `.std()` method to find the *sample* standard deviation of each column.

```
returns.std()
```

```
AAPL    0.0206
GOOG    0.0194
IBM     0.0143
MSFT    0.0171
dtype: float64
```

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

We annualize mean returns by multiplying by T ($T = 252$ for daily returns, $T = 12$ for month returns, and so on). We annualize standard deviations by multiplying by \sqrt{T} .

```
returns.mean().mul(252) # whenever I can, I use the .mul() method, so I can keep chaining!
```

```
AAPL    0.3625
GOOG    0.2552
IBM     0.0980
MSFT    0.2002
dtype: float64
```

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

```
AAPL    0.3276
GOOG    0.3074
IBM     0.2272
MSFT    0.2722
dtype: float64
```

20.3.4 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).

```
vols = returns.std().mul(np.sqrt(252) * 100)
means = returns.mean().mul(252 * 100)

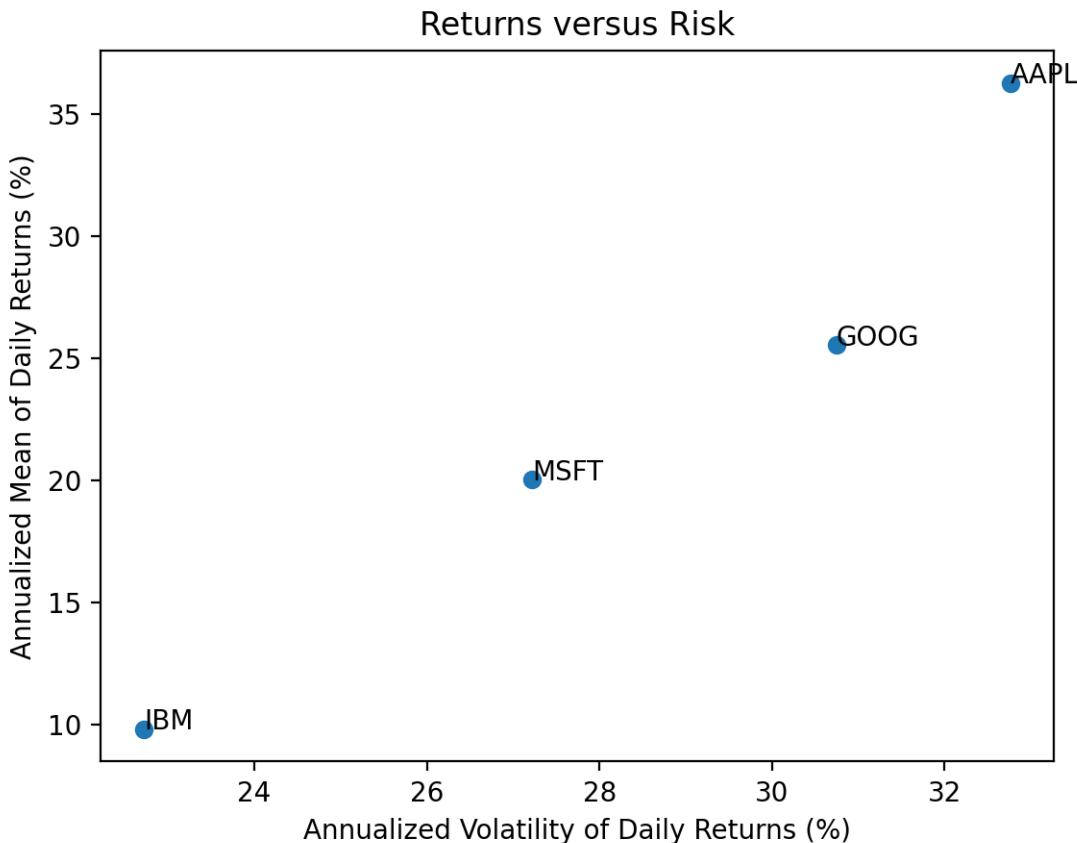
plt.scatter(
    x=vols,
    y=means
)

plt.xlabel('Annualized Volatility of Daily Returns (%)')
plt.ylabel('Annualized Mean of Daily Returns (%)')

# plt.xlim((0, vols.max() + 5))
# plt.ylim((0, means.max() + 5))

# add tickers to each point
for i in means.index: # loop over ticker index
    plt.text( # plots string s at coordinates x and y
        x=vols[i], # indexes volatility
        y=means[i], # indexes mean return
        s=i # ticker index
    )

plt.title('Returns versus Risk')
plt.show() # suppresses output of last function call
```



20.3.5 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](https://en.wikipedia.org/wiki/Dow_Jones_Industrial_Average). We will need to download new data, into `tickers2`, `prices2`, and `returns2`.

```
url2 = 'https://en.wikipedia.org/wiki/Dow_Jones_Industrial_Average'
wiki2 = pd.read_html(io=url2)
```

```
tickers2 = wiki2[1]['Symbol'].to_list()
```

```
tickers2[:5]
```

```
['MMM', 'AXP', 'AMGN', 'AAPL', 'BA']
```

```
prices2 = yf.download(tickers=tickers2)
```

```
[*****100%*****] 30 of 30 completed
```

```
returns2 = (
    prices2['Adj Close'] # slices the adj close group of columns from prices
    .iloc[:-1] # drop last row (which is intraday during class) to avoid fill_method warning
    .pct_change() # calculate percent in adj closes (row[n] - row[n-1]) / row[n-1]
    .dropna() # drops rows with any missing values
)

returns2
```

Date	AAPL	AMGN	AXP	BA	CAT	CRM	CSCO	CVX	DIS	DOW	...
2019-03-21	0.0368	0.0040	0.0095	-0.0092	0.0079	0.0210	0.0128	0.0094	-0.0121	-0.0165	...
2019-03-22	-0.0207	-0.0270	-0.0211	-0.0283	-0.0320	-0.0326	-0.0222	-0.0220	-0.0040	-0.0078	...
2019-03-25	-0.0121	-0.0006	-0.0038	0.0229	0.0124	-0.0038	-0.0002	-0.0016	-0.0041	0.0113	...
2019-03-26	-0.0103	0.0090	0.0042	-0.0002	0.0035	-0.0092	0.0095	0.0101	0.0218	-0.0061	...
2019-03-27	0.0090	-0.0104	-0.0047	0.0103	-0.0049	-0.0269	-0.0017	-0.0108	0.0013	0.0256	...
...
2024-01-26	-0.0090	0.0049	0.0710	0.0178	-0.0045	0.0033	-0.0036	0.0038	0.0053	-0.0160	...
2024-01-29	-0.0036	0.0054	-0.0028	-0.0014	0.0128	0.0283	0.0029	-0.0004	0.0223	0.0002	...
2024-01-30	-0.0192	0.0037	0.0164	-0.0231	0.0050	-0.0005	-0.0010	0.0070	-0.0056	0.0074	...
2024-01-31	-0.0194	-0.0011	-0.0167	0.0529	-0.0146	-0.0231	-0.0394	-0.0179	-0.0092	-0.0160	...
2024-02-01	0.0133	0.0328	0.0124	-0.0058	0.0246	0.0096	0.0000	0.0031	0.0105	-0.0011	...

```
vols = returns2.std().mul(np.sqrt(252) * 100)
means = returns2.mean().mul(252 * 100)

plt.scatter(
    x=vols,
    y=means
)

plt.xlabel('Annualized Volatility of Daily Returns (%)')
plt.ylabel('Annualized Mean of Daily Returns (%)')

# plt.xlim((0, vols.max() + 5))
# plt.ylim((0, means.max() + 5))

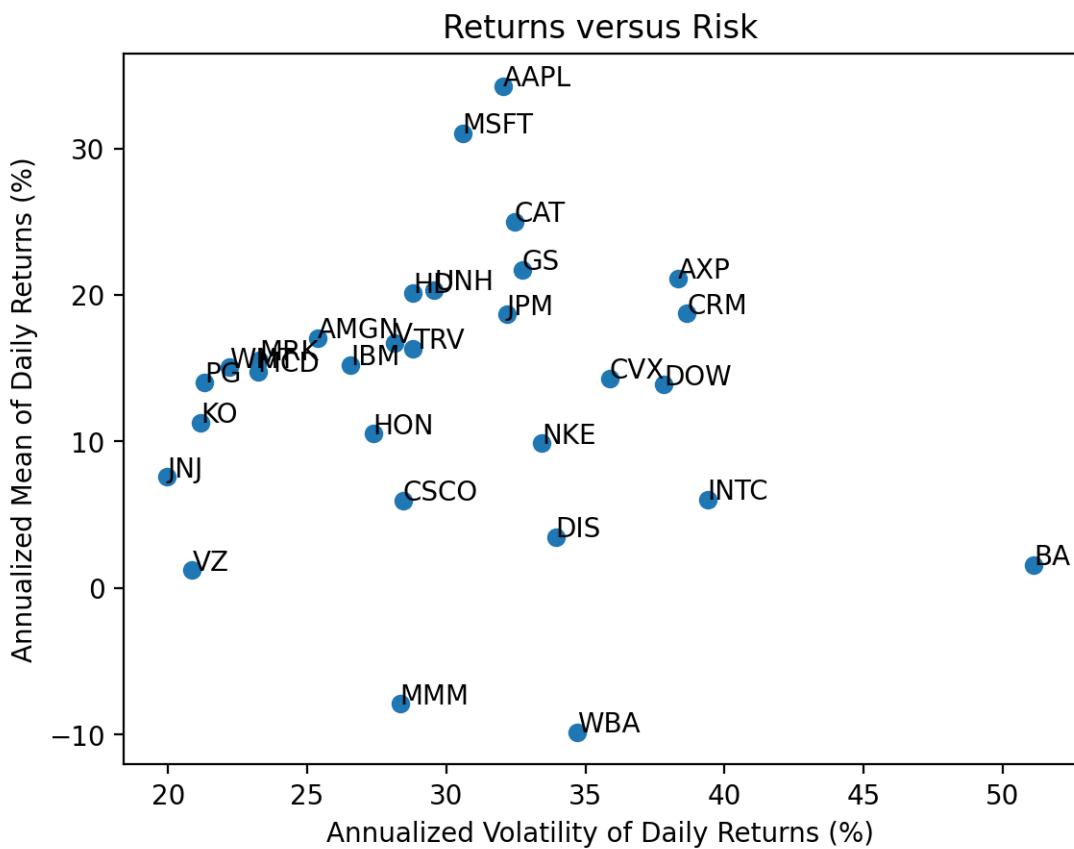
# add tickers to each point
```

```

for i in means.index: # loop over ticker index
    plt.text( # plots string s at coordinates x and y
        x=vols[i], # indexes volatility
        y=means[i], # indexes mean return
        s=i # ticker index
    )

plt.title('Returns versus Risk')
plt.show() # suppresses output of last function call

```



With 30 stocks we see there is no relation between returns and volatility because most volatility is *diversifiable* and uncompensated.

20.3.6 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.

In general, I prefer to do simple math on pandas objects (data frames and series) with methods instead of operators:

For example:

1. `.add(1)` instead of `+ 1`
2. `.sub(1)` instead of `- 1`
3. `.div(1)` instead of `/ 1`
4. `.mul(1)` instead of `* 1`

The advantage of methods over operators, is that we can easily chain methods without lots of parentheses.

```
total_returns2 = returns2.add(1).prod().sub(1)
```

```
total_returns2.iloc[:5]
```

```
AAPL    3.1210
AMGN    0.9635
AXP     0.9687
BA      -0.4289
CAT     1.6083
dtype: float64
```

20.3.7 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.

A histogram is a great way to visualize data!

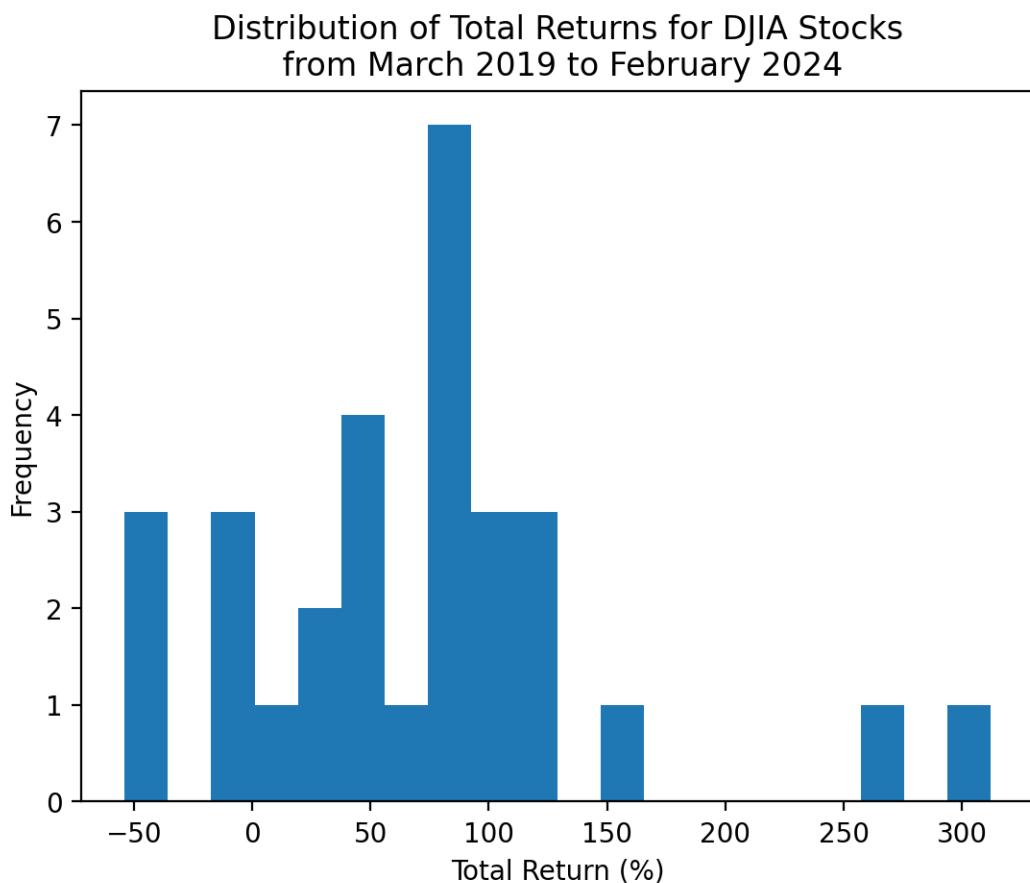
```
start_date = returns2.index.min()
stop_date = returns2.index.max()

(
    returns2
    .add(1)
    .prod()
```

```

    .sub(1)
    .mul(100)
    .plot(kind='hist', bins=20) # grids=True to add grid lines
)
plt.xlabel('Total Return (%)')
plt.title(f'Distribution of Total Returns for DJIA Stocks\n from {start_date:%B %Y} to {st
plt.show()

```



With only 30 stocks, we can actually connect a stock to its return in a plot!

```

start_date = returns2.index.min()
stop_date = returns2.index.max()

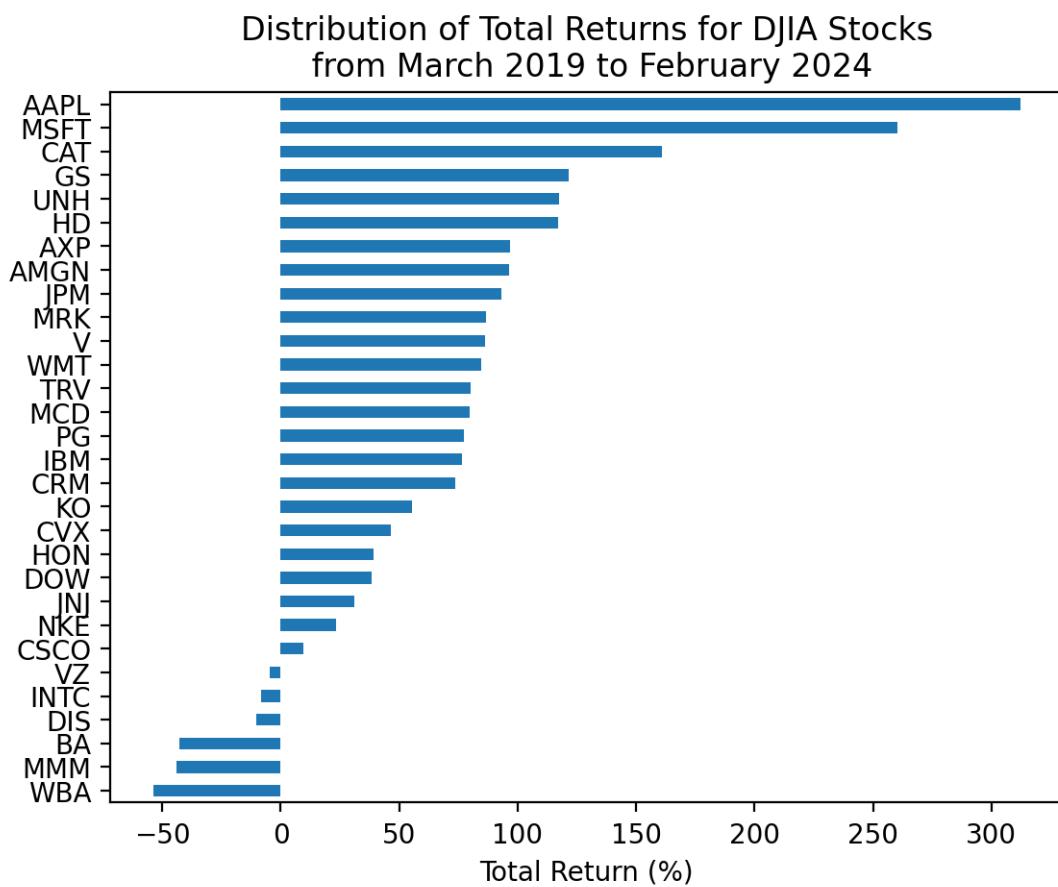
(
    returns2

```

```

    .add(1)
    .prod()
    .sub(1)
    .sort_values() # sort by total returns
    .mul(100)
    .plot(kind='barh') # horizontal bar chart
)
plt.xlabel('Total Return (%)')
plt.title(f'Distribution of Total Returns for DJIA Stocks\n from {start_date:%B %Y} to {st
plt.show()

```



20.3.8 Which stocks have the minimum and maximum total returns?

If we want the *values*, the `.min()` and `.max()` methods are the way to go!

```
total_returns2.min()
```

```
-0.5384
```

```
total_returns2.max()
```

```
3.1210
```

The `.min()` and `.max()` methods give the values but not the tickers (or index). We use the `.idxmin()` and `.idxmax()` to get the tickers (or index).

```
total_returns2.idxmin()
```

```
'WBA'
```

```
total_returns2.idxmax()
```

```
'AAPL'
```

Here is what I would use!

```
total_returns2.sort_values().iloc[[0, -1]]
```

```
WBA    -0.5384
AAPL    3.1210
dtype: float64
```

Not the exactly right tool here, but the `.nsmallest()` and `.nlargest()` methods are really useful!

```
total_returns2.nsmallest(3)
```

```
WBA    -0.5384
MMM    -0.4408
BA     -0.4289
dtype: float64
```

```
total_returns2.nlargest(3)
```

```
AAPL    3.1210
MSFT    2.6028
CAT     1.6083
dtype: float64
```

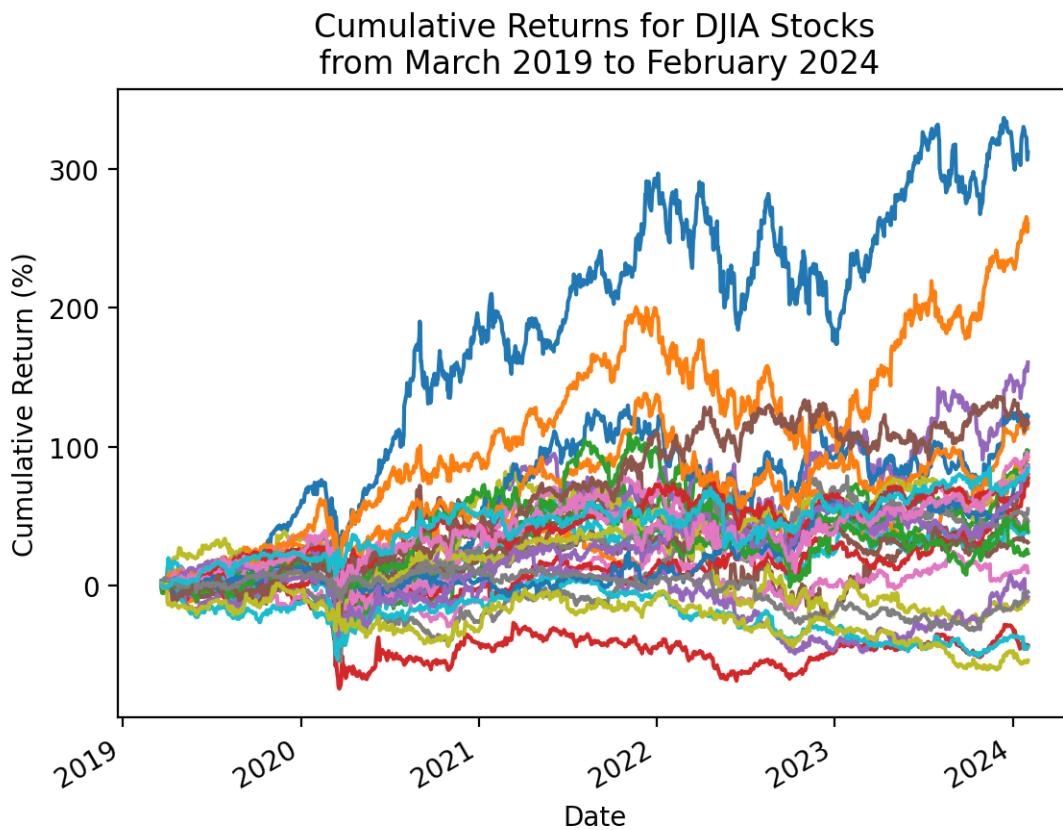
20.3.9 Plot the cumulative returns for the stocks in the DJIA

We can use the cumulative product method `.cumprod()` to calculate the right hand side of the formula above.

```
start_date = returns2.index.min()
stop_date = returns2.index.max()

(
    returns2
    .add(1)
    .cumprod()
    .sub(1)
    .mul(100)
    .plot(legend=False) # with 30 stocks, this legend is too big to be useful
)

plt.ylabel('Cumulative Return (%)')
plt.title(f'Cumulative Returns for DJIA Stocks\n from {start_date:%B %Y} to {stop_date:%B %Y}')
plt.show()
```



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

```
total_returns2.sort_values().iloc[[0, -1]].index
```

```
Index(['WBA', 'AAPL'], dtype='object')
```

```
returns2[total_returns2.sort_values().iloc[[0, -1]].index]
```

	WBA	AAPL
Date		
2019-03-21	0.0129	0.0368
2019-03-22	-0.0187	-0.0207
2019-03-25	-0.0115	-0.0121

	WBA	AAPL
Date		
2019-03-26	0.0037	-0.0103
2019-03-27	0.0050	0.0090
...
2024-01-26	-0.0113	-0.0090
2024-01-29	-0.0057	-0.0036
2024-01-30	0.0018	-0.0192
2024-01-31	-0.0083	-0.0194
2024-02-01	0.0301	0.0133

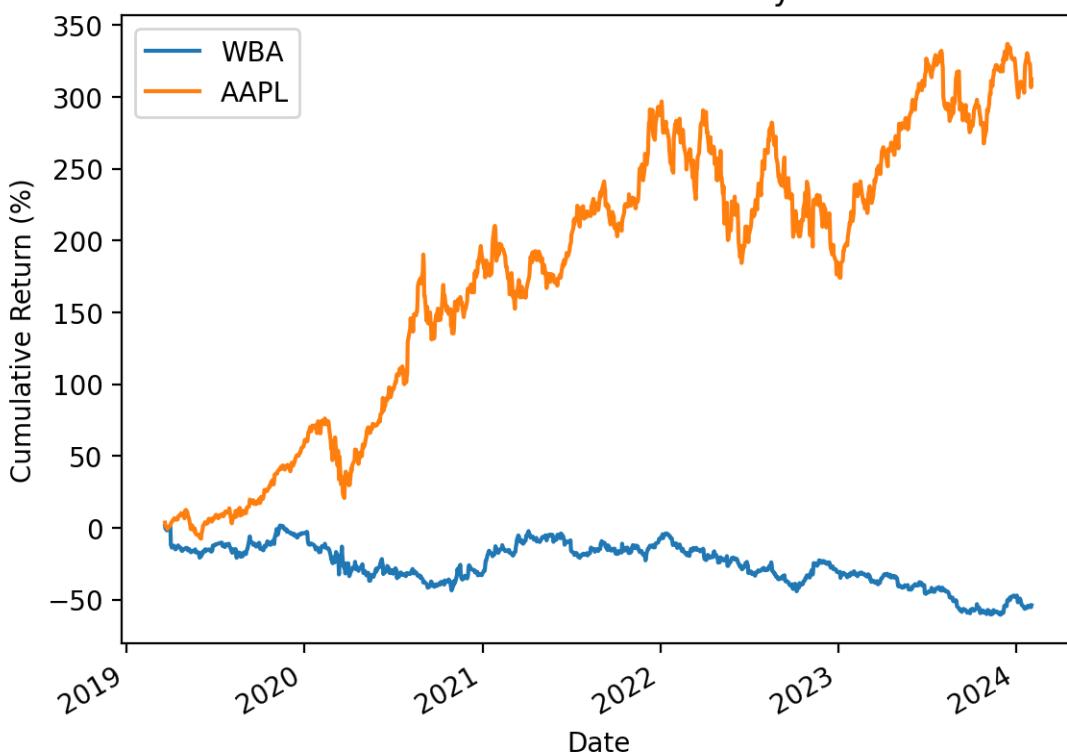
```

start_date = returns2.index.min()
stop_date = returns2.index.max()

(
    returns2 # all returns for all stocks
    [total_returns2.sort_values().iloc[[0, -1]].index] # slice min and max total return st
    .add(1)
    .cumprod()
    .sub(1)
    .mul(100)
    .plot()
)
plt.ylabel('Cumulative Return (%)')
plt.title(f'Cumulative Returns for DJIA Stocks\n from {start_date:%B %Y} to {stop_date:%B
plt.show()

```

Cumulative Returns for DJIA Stocks
from March 2019 to February 2024



Part V

Week 5

21 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'
```

21.1 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.

21.1.1 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.

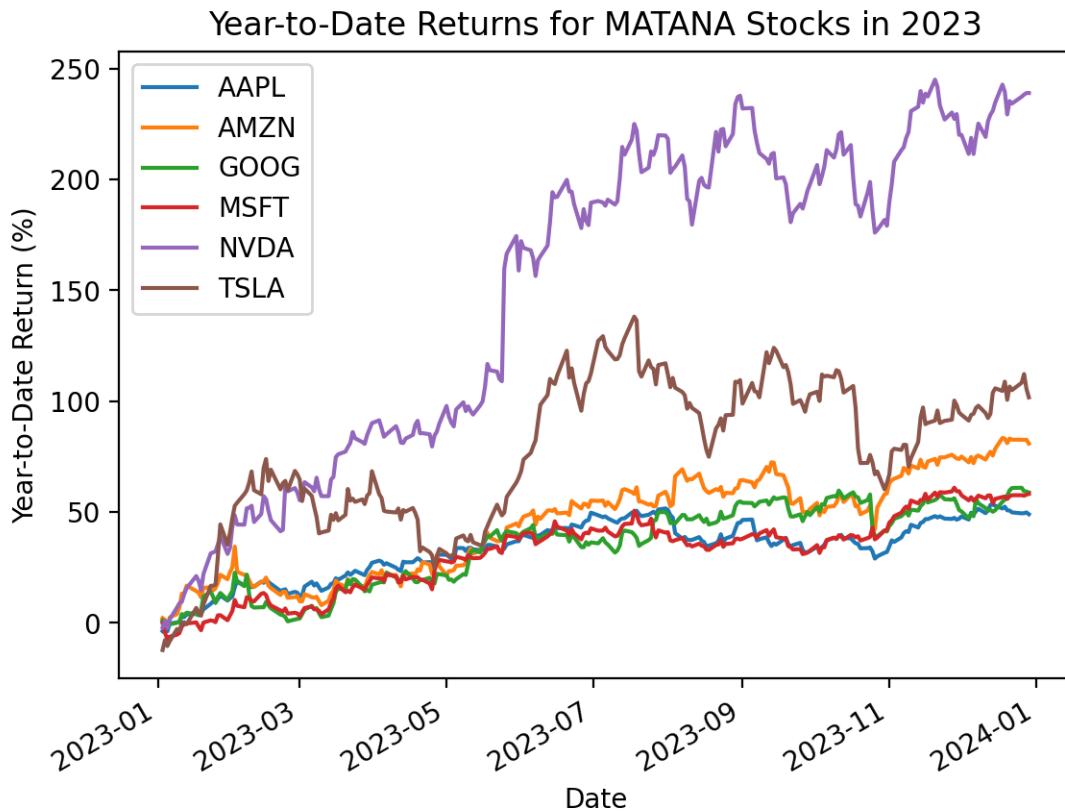
```
df = yf.download(tickers='MSFT GOOG TSLA AMZN NVDA AAPL')
df
```

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

Date	Adj Close						Close AAPL	AMZN	GOOG
	AAPL	AMZN	GOOG	MSFT	NVDA	TSLA			
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
...
2024-01-26	192.4200	159.1200	153.7900	403.9300	610.3100	183.2500	192.4200	159.1200	153.7900
2024-01-29	191.7300	161.2600	154.8400	409.7200	624.6500	190.9300	191.7300	161.2600	154.8400
2024-01-30	188.0400	159.0000	153.0500	408.5900	627.7400	191.5900	188.0400	159.0000	153.0500
2024-01-31	184.4000	155.2000	141.8000	397.5800	615.2700	187.2900	184.4000	155.2000	141.8000
2024-02-01	186.8600	159.2800	142.7100	403.7800	630.2700	188.8600	186.8600	159.2800	142.7100

```

(
    df
    ['Adj Close']
    .pct_change()
    .loc['2023']
    .add(1)
    .cumprod()
    .sub(1)
    .mul(100)
    .plot()
)
plt.ylabel('Year-to-Date Return (%)')
plt.title('Year-to-Date Returns for MATANA Stocks in 2023')
plt.show()
```



21.1.2 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_26796\2526882917.py:1: FutureWarning: The arg
```

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

```
type(ff_all)
```

```
dict
```

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

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

```
-----  
This file was created by CMPT_ME_BEME_RET_DAILY using the 202312 CRSP database. The Tbill re
```

```
0 : (25649 rows x 4 cols)
```

```
ff_all[0]
```

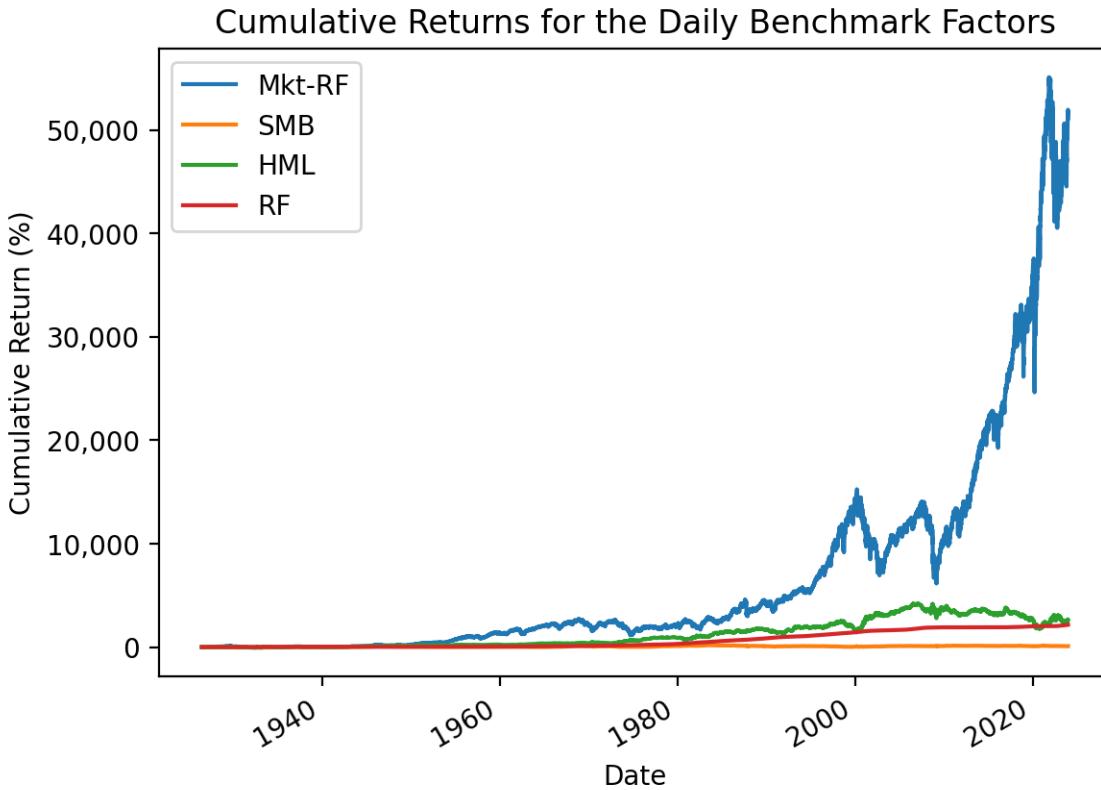
Date	Mkt-RF	SMB	HML	RF
1926-07-01	0.1000	-0.2500	-0.2700	0.0090
1926-07-02	0.4500	-0.3300	-0.0600	0.0090
1926-07-06	0.1700	0.3000	-0.3900	0.0090
1926-07-07	0.0900	-0.5800	0.0200	0.0090

Date	Mkt-RF	SMB	HML	RF
1926-07-08	0.2100	-0.3800	0.1900	0.0090
...
2023-12-22	0.2100	0.6400	0.0900	0.0210
2023-12-26	0.4800	0.6900	0.4600	0.0210
2023-12-27	0.1600	0.1400	0.1200	0.0210
2023-12-28	-0.0100	-0.3600	0.0300	0.0210
2023-12-29	-0.4300	-1.1200	-0.3700	0.0210

```

(
    ff_all[0] # slice factors
    .div(100) # convert to decimal
    .add(1) # calculate cumulative returns
    .cumprod()
    .sub(1)
    .mul(100) # convert to percent
    .plot() # 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.yscale('log') # log scale impractical here with negative returns
plt.show()

```



21.2 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.

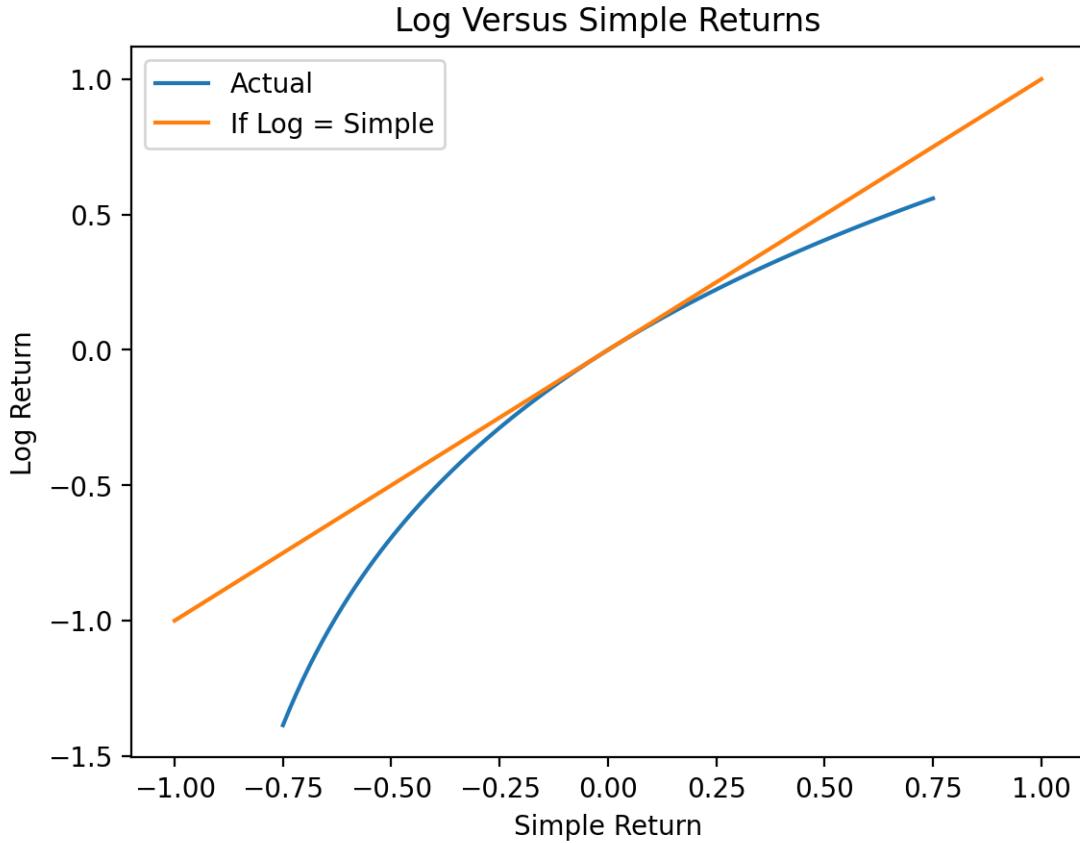
We will typically use *simple* returns instead of *log* returns. However, this section explains the differences between simple and log returns and where each is appropriate.

21.2.1 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 $R_{\text{simple},t} \approx R_{\text{log},t}$ for small R_s .

```
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()
```



21.2.2 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.

21.2.3 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
```

<hr/>		R
0		0.6433
1		-0.1291
2		0.9111
3		3.5861
4		-0.2088
...		...
9995		2.6733

R	
9996	-0.8644
9997	-0.5060
9998	0.6418
9999	0.9048

```
df2.describe()
```

R	
count	10000.0000
mean	0.6529
std	2.1918
min	-0.9802
25%	-0.4896
50%	-0.0026
75%	0.9564
max	49.7158

We can time the calculation of 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)
)
```

284 ms ± 80.7 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)
)
```

```
557 µs ± 60.6 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

```
df2.head(15)
```

	R	R12_via_simple	R12_via_log
0	0.6433	NaN	NaN
1	-0.1291	NaN	NaN
2	0.9111	NaN	NaN
3	3.5861	NaN	NaN
4	-0.2088	NaN	NaN
5	-0.2087	NaN	NaN
6	3.8511	NaN	NaN
7	1.1542	NaN	NaN
8	-0.3747	NaN	NaN
9	0.7204	NaN	NaN
10	-0.3709	NaN	NaN
11	-0.3723	33.8643	33.8643
12	0.2737	26.0236	26.0236
13	-0.8524	3.5800	3.5800
14	-0.8218	-0.5730	-0.5730

```
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!

21.3 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 .

21.3.1 The 1/N Portfolio

The $\frac{1}{N}$ portfolio equally weights portfolio assets, so $w_1 = w_2 = \dots = w_N = \frac{1}{N}$. 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['2023']
```

```
returns
```

Date	AAPL	AMZN	GOOG	MSFT	NVDA	TSLA
2023-01-03	-0.0374	0.0217	0.0109	-0.0010	-0.0205	-0.1224
2023-01-04	0.0103	-0.0079	-0.0110	-0.0437	0.0303	0.0512
2023-01-05	-0.0106	-0.0237	-0.0219	-0.0296	-0.0328	-0.0290
2023-01-06	0.0368	0.0356	0.0160	0.0118	0.0416	0.0247
2023-01-09	0.0041	0.0149	0.0073	0.0097	0.0518	0.0593
...
2023-12-22	-0.0055	-0.0027	0.0065	0.0028	-0.0033	-0.0077
2023-12-26	-0.0028	-0.0001	0.0007	0.0002	0.0092	0.0161
2023-12-27	0.0005	-0.0005	-0.0097	-0.0016	0.0028	0.0188
2023-12-28	0.0022	0.0003	-0.0011	0.0032	0.0021	-0.0316
2023-12-29	-0.0054	-0.0094	-0.0025	0.0020	0.0000	-0.0186

```
returns.mean()
```

```
AAPL    0.0017
AMZN    0.0026
GOOG    0.0020
MSFT    0.0020
NVDA    0.0053
TSLA    0.0034
dtype: float64
```

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

```
Date
2023-01-03    -0.0248
2023-01-04     0.0049
2023-01-05    -0.0246
2023-01-06     0.0278
2023-01-09     0.0245
...
2023-12-22    -0.0017
2023-12-26     0.0039
2023-12-27     0.0017
2023-12-28    -0.0041
2023-12-29    -0.0056
Length: 250, 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.

21.3.2 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
2023-01-03    -0.0248
```

```
2023-01-04    0.0049
2023-01-05   -0.0246
2023-01-06    0.0278
2023-01-09    0.0245
...
2023-12-22   -0.0017
2023-12-26    0.0039
2023-12-27    0.0017
2023-12-28   -0.0041
2023-12-29   -0.0056
Length: 250, dtype: float64
```

Both approaches give the same answer!

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

```
True
```

22 Herron Topic 1 - Practice for Section 02

22.1 Announcements

1. DataCamp
 1. *Data Manipulation with pandas* due by Friday, 2/9, at 11:59 PM
 2. *Joining Data with pandas* due by Friday, 2/16, at 11:59 PM
 3. *Earn 10,000 XP* due by Friday, 3/15, at 11:59 PM
2. I posted Project 1 to Canvas
 1. Slides and notebook due by Friday, 2/23, at 11:59 PM
 2. Keep joining teams and let me know if you need help

22.2 10-Minute Recap

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import pandas_datareader as pdr
import yfinance as yf

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

First, we will use two packages to download data from the web:

1. `yfinance` for Yahoo! Finance data
2. `pandas-datareader` for Ken French data (and FRED data)

Second, there are “simple returns” and “log returns”

1. Simple returns are the returns that investors receive that we learned in FINA 6331 and FINA 6333: $r_t = \frac{p_t + d_t - p_{t-1}}{p_{t-1}}$

2. Log returns are the log of one plus simple returns. Why do we use them?

1. **Log returns are additive**, while simple returns are multiplicative. This additive property makes math really easy with log returns: $\log(\prod_{t=0}^T (1+r_t)) = \sum_{t=0}^T \log(1+r_t)$, so $r_{0,T} = \prod_{t=0}^T (1+r_t) - 1 = e^{\sum_{t=0}^T \log(1+r_t)} - 1$
2. **Log returns are almost normally distributed**

We will almost always use simple returns. The exception is time-consuming calculations, which we will often do in log returns to save us time.

Third, we can calculate portfolio returns a few ways

1. We can calculate equally-weighted returns with `returns.mean(axis=1)`, **rebalanced at the same frequency as returns**
2. We can calculate any-weighted with `returns.dot(weights)`, where `weights` is an array of portfolio weights that are not necessarily equally-weighted, still **rebalanced at the same frequency as returns**

22.3 Practice

22.3.1 Download all available daily price data for tickers TSLA, F, AAPL, AMZN, and META to data frame prices

```
tickers = 'TSLA F AAPL AMZN META'  
# tickers = ['TSLA', 'F', 'AAPL', 'AMZN', 'META']  
prices = yf.download(tickers=tickers)  
  
prices
```

[*****100%*****] 5 of 5 completed

Date	Adj Close					Close			
	AAPL	AMZN	F	META	TSLA	AAPL	AMZN	F	META
1972-06-01	NaN	NaN	0.2419	NaN	NaN	NaN	NaN	2.1532	NaN
1972-06-02	NaN	NaN	0.2414	NaN	NaN	NaN	NaN	2.1492	NaN
1972-06-05	NaN	NaN	0.2414	NaN	NaN	NaN	NaN	2.1492	NaN
1972-06-06	NaN	NaN	0.2387	NaN	NaN	NaN	NaN	2.1248	NaN
1972-06-07	NaN	NaN	0.2373	NaN	NaN	NaN	NaN	2.1127	NaN
...

Date	Adj Close					Close			
	AAPL	AMZN	F	META	TSLA	AAPL	AMZN	F	META
2024-02-05	187.6800	170.3100	11.5900	459.4100	181.0600	187.6800	170.3100	11.5900	459.4100
2024-02-06	189.3000	169.1500	12.0700	454.7200	185.1000	189.3000	169.1500	12.0700	454.7200
2024-02-07	189.4100	170.5300	12.8000	469.5900	187.5800	189.4100	170.5300	12.8000	469.5900
2024-02-08	188.3200	169.8400	12.8300	470.0000	189.5600	188.3200	169.8400	12.8300	470.0000
2024-02-09	188.8500	174.4500	12.6800	468.1100	193.5700	188.8500	174.4500	12.6800	468.1100

22.3.2 Calculate all available daily returns and save to data frame returns

```

returns = (
    prices['Adj Close'] # slice adj close
    .iloc[:-1] # drop the last price because it might be intraday (i.e., not a close)
    .pct_change() # calculate simple returns
)

returns

```

Date	AAPL	AMZN	F	META	TSLA
1972-06-01	NaN	NaN	NaN	NaN	NaN
1972-06-02	NaN	NaN	-0.0019	NaN	NaN
1972-06-05	NaN	NaN	0.0000	NaN	NaN
1972-06-06	NaN	NaN	-0.0113	NaN	NaN
1972-06-07	NaN	NaN	-0.0057	NaN	NaN
...
2024-02-02	-0.0054	0.0787	0.0033	0.2032	-0.0050
2024-02-05	0.0098	-0.0087	-0.0453	-0.0328	-0.0365
2024-02-06	0.0086	-0.0068	0.0414	-0.0102	0.0223
2024-02-07	0.0006	0.0082	0.0605	0.0327	0.0134
2024-02-08	-0.0058	-0.0040	0.0023	0.0009	0.0106

22.3.3 Slices returns for the 2020s and assign to returns_2020s

```
returns_2020s = returns.loc['2020':] # always use an unambiguous date format, like YYYY-MM-  
returns_2020s
```

Date	AAPL	AMZN	F	META	TSLA
2020-01-02	0.0228	0.0272	0.0129	0.0221	0.0285
2020-01-03	-0.0097	-0.0121	-0.0223	-0.0053	0.0296
2020-01-06	0.0080	0.0149	-0.0054	0.0188	0.0193
2020-01-07	-0.0047	0.0021	0.0098	0.0022	0.0388
2020-01-08	0.0161	-0.0078	0.0000	0.0101	0.0492
...
2024-02-02	-0.0054	0.0787	0.0033	0.2032	-0.0050
2024-02-05	0.0098	-0.0087	-0.0453	-0.0328	-0.0365
2024-02-06	0.0086	-0.0068	0.0414	-0.0102	0.0223
2024-02-07	0.0006	0.0082	0.0605	0.0327	0.0134
2024-02-08	-0.0058	-0.0040	0.0023	0.0009	0.0106

What if we want a more complicated slice? We can combine .loc[] and Boolean conditions!

```
returns.loc[  
    (returns.index.year == 2020) &  
    ((returns.index.month == 1) | (returns.index.month == 12))  
]
```

Date	AAPL	AMZN	F	META	TSLA
2020-01-02	0.0228	0.0272	0.0129	0.0221	0.0285
2020-01-03	-0.0097	-0.0121	-0.0223	-0.0053	0.0296
2020-01-06	0.0080	0.0149	-0.0054	0.0188	0.0193
2020-01-07	-0.0047	0.0021	0.0098	0.0022	0.0388
2020-01-08	0.0161	-0.0078	0.0000	0.0101	0.0492
2020-01-09	0.0212	0.0048	0.0011	0.0143	-0.0219
2020-01-10	0.0023	-0.0094	-0.0011	-0.0011	-0.0066
2020-01-13	0.0214	0.0043	-0.0011	0.0177	0.0977

	AAPL	AMZN	F	META	TSLA
Date					
2020-01-14	-0.0135	-0.0116	0.0054	-0.0128	0.0249
2020-01-15	-0.0043	-0.0040	-0.0108	0.0095	-0.0361
2020-01-16	0.0125	0.0085	-0.0022	0.0028	-0.0097
2020-01-17	0.0111	-0.0070	-0.0011	0.0017	-0.0058
2020-01-21	-0.0068	0.0146	0.0055	-0.0032	0.0719
2020-01-22	0.0036	-0.0024	-0.0054	-0.0005	0.0409
2020-01-23	0.0048	-0.0015	-0.0022	-0.0070	0.0046
2020-01-24	-0.0029	-0.0122	-0.0153	-0.0083	-0.0129
2020-01-27	-0.0294	-0.0179	-0.0122	-0.0141	-0.0120
2020-01-28	0.0283	0.0136	0.0090	0.0136	0.0159
2020-01-29	0.0209	0.0026	0.0045	0.0250	0.0249
2020-01-30	-0.0014	0.0068	-0.0023	-0.0614	0.1030
2020-01-31	-0.0443	0.0738	-0.0023	-0.0364	0.0152
2020-12-01	0.0308	0.0164	0.0176	0.0346	0.0302
2020-12-02	0.0029	-0.0051	-0.0043	0.0034	-0.0273
2020-12-03	-0.0011	-0.0052	0.0011	-0.0197	0.0432
2020-12-04	-0.0056	-0.0076	0.0141	-0.0076	0.0095
2020-12-07	0.0123	-0.0014	-0.0128	0.0210	0.0713
2020-12-08	0.0051	0.0061	0.0033	-0.0076	0.0127
2020-12-09	-0.0209	-0.0230	0.0216	-0.0193	-0.0699
2020-12-10	0.0120	-0.0009	-0.0349	-0.0029	0.0374
2020-12-11	-0.0067	0.0048	-0.0110	-0.0129	-0.0272
2020-12-14	-0.0051	0.0130	-0.0122	0.0023	0.0489
2020-12-15	0.0501	0.0026	0.0269	0.0050	-0.0103
2020-12-16	-0.0005	0.0240	-0.0120	0.0004	-0.0165
2020-12-17	0.0070	-0.0015	0.0044	-0.0043	0.0532
2020-12-18	-0.0159	-0.0106	-0.0143	0.0070	0.0596
2020-12-21	0.0124	0.0014	-0.0022	-0.0131	-0.0649
2020-12-22	0.0285	0.0001	-0.0157	-0.0209	-0.0146
2020-12-23	-0.0070	-0.0066	0.0228	0.0038	0.0088
2020-12-24	0.0077	-0.0039	-0.0145	-0.0026	0.0244
2020-12-28	0.0358	0.0351	0.0034	0.0359	0.0029
2020-12-29	-0.0133	0.0116	-0.0079	-0.0008	0.0035
2020-12-30	-0.0085	-0.0109	0.0045	-0.0177	0.0432
2020-12-31	-0.0077	-0.0088	-0.0079	0.0047	0.0157

22.3.4 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']

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

```
C:\Users\r.herron\AppData\Local\Temp\ipykernel_28696\2526882917.py:1: FutureWarning: The arg
```

```
ff_all = pdr.DataReader(
```

The DESCR key in the dictionary tells us about the data frames that pandas-datareader returns.

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

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

```
-----
```

```
This file was created by CMPT_ME_BEME_RET_DAILY using the 202312 CRSP database. The Tbill r
```

```
0 : (25649 rows x 4 cols)
```

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

```
ff = ff_all[0].div(100)
```

```
ff
```

Date	Mkt-RF	SMB	HML	RF
1926-07-01	0.0010	-0.0025	-0.0027	0.0001
1926-07-02	0.0045	-0.0033	-0.0006	0.0001
1926-07-06	0.0017	0.0030	-0.0039	0.0001
1926-07-07	0.0009	-0.0058	0.0002	0.0001
1926-07-08	0.0021	-0.0038	0.0019	0.0001
...
2023-12-22	0.0021	0.0064	0.0009	0.0002
2023-12-26	0.0048	0.0069	0.0046	0.0002
2023-12-27	0.0016	0.0014	0.0012	0.0002
2023-12-28	-0.0001	-0.0036	0.0003	0.0002
2023-12-29	-0.0043	-0.0112	-0.0037	0.0002

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

We use the `.prod()` method to calculate *total* returns, because $r_{total} = r_{0,T} = \left[\prod_{t=0}^T (1 + r_t) \right] - 1$.

```
totret = (
    returns_2020s # returns during the 2020s
    .add(1) # add 1 before we compound
    .prod() # compound all returns
    .sub(1) # subtract 1 to recover total returns
)

totret
```

```
AAPL    1.6331
AMZN    0.8383
F        0.6090
```

```
META    1.2899  
TSLA    5.7970  
dtype: float64
```

```
cumret_cumprod = returns_2020s.add(1).cumprod().sub(1)
```

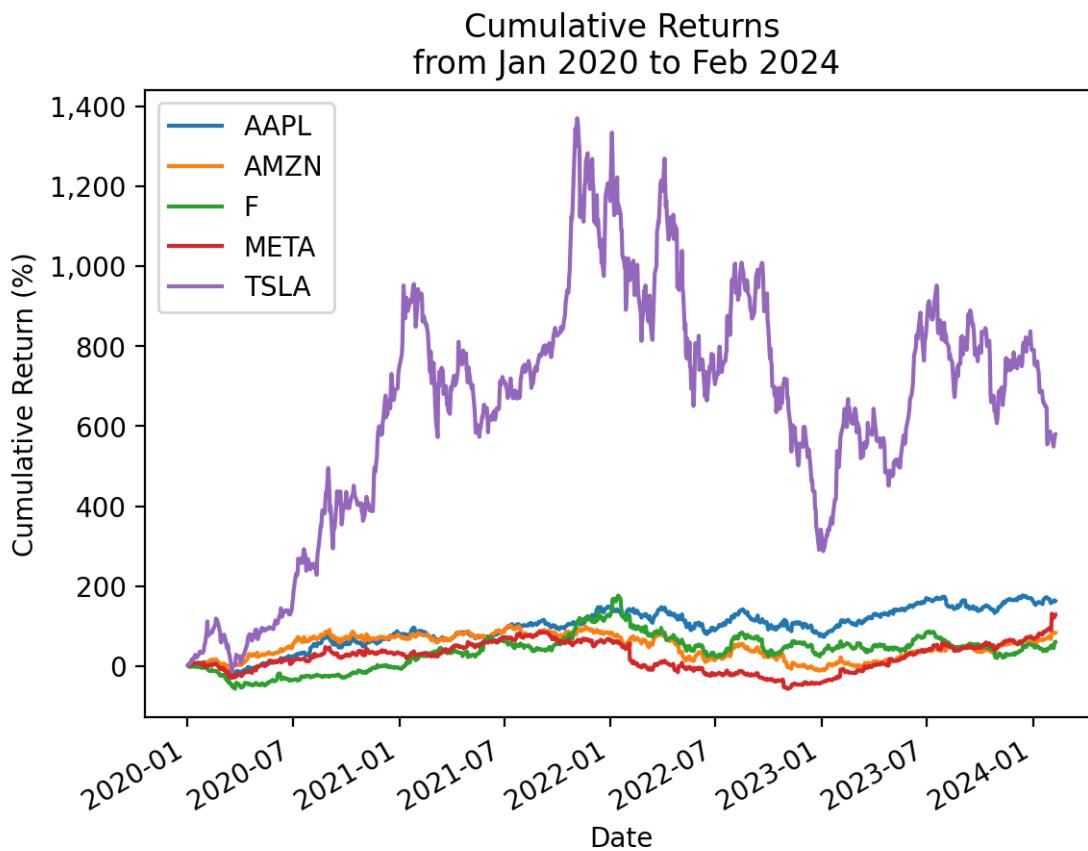
The last row in the cumulative returns data frame `cumret_cumprod` is the same as the total returns series `totret`!

```
np.allclose(totret, cumret_cumprod.iloc[-1])
```

```
True
```

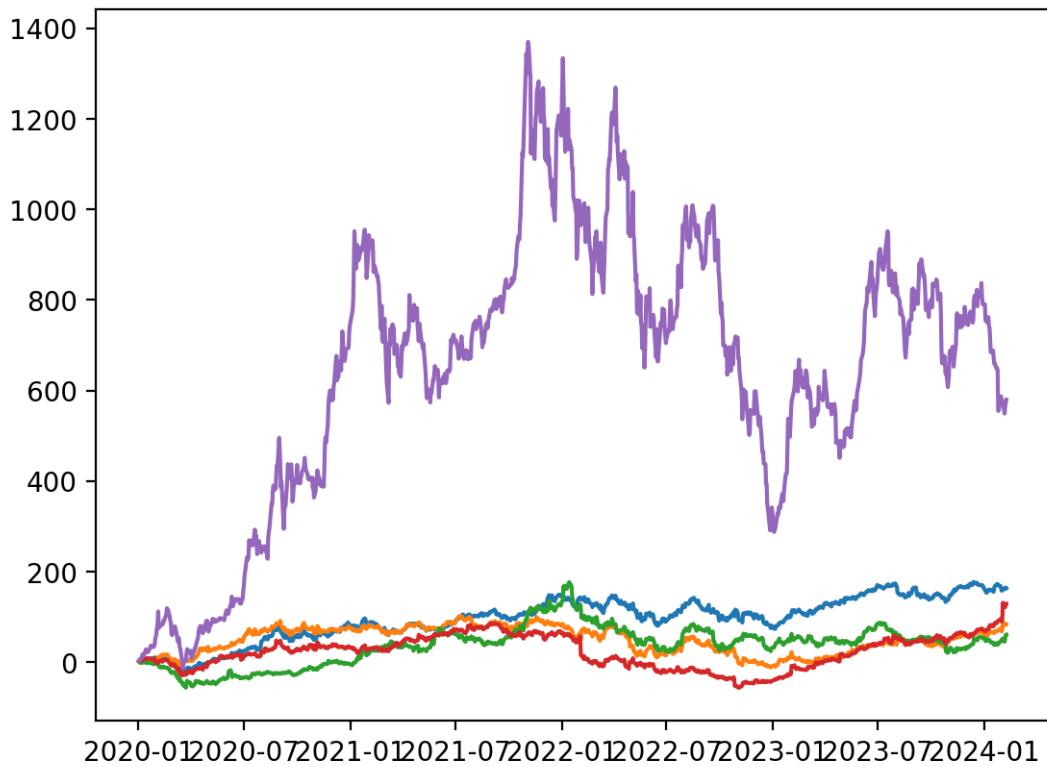
We can use the `.plot()` method to plot these cumulative returns!

```
cumret_cumprod.mul(100).plot()  
  
# https://stackoverflow.com/questions/25973581/how-to-format-axis-number-format-to-thousan  
from matplotlib import ticker  
plt.gca().get_yaxis().set_major_formatter(ticker.FuncFormatter(lambda x, p: format(int(x),  
  
plt.ylabel('Cumulative Return (%)')  
plt.title(f'Cumulative Returns\n from {cumret_cumprod.index.min():%b %Y} to {cumret_cumpro  
plt.show()
```



We can also `plt.plot()`, but lose a few of the formatting defaults built into the pandas method `.plot()`.

```
plt.plot(cumret_cumprod.mul(100))
```



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

```
 cumret_cumsum = (
    returns_2020s
    .pipe(np.log1p) # converts simple returns to log returns
    .cumsum() # log returns are additive
    .pipe(np.expm1) # converts log returns to simple returns
)
# cumret_cumsum = returns_2020s.add(1).pipe(np.log).cumsum().pipe(np.exp).sub(1)

np.allclose(totret, cumret_cumsum.iloc[-1])
```

True

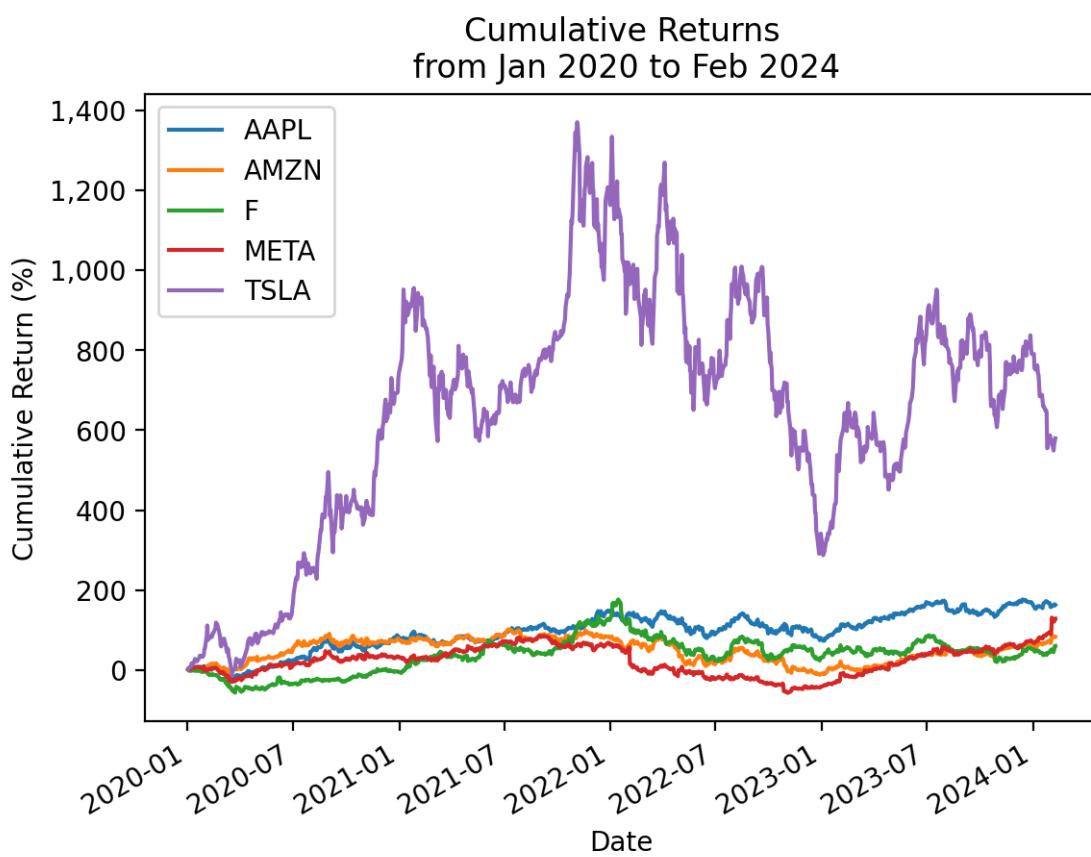
```

cumret_cumsum.mul(100).plot()

# https://stackoverflow.com/questions/25973581/how-to-format-axis-number-format-to-thousan
from matplotlib import ticker
plt.gca().get_yaxis().set_major_formatter(ticker.FuncFormatter(lambda x, p: format(int(x),

plt.ylabel('Cumulative Return (%)')
plt.title(f'Cumulative Returns\n from {cumret_cumsum.index.min():%b %Y} to {cumret_cumsum.
plt.show()

```



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

We can also calculate cumulative returns as the ratio of adjusted closes. That is $R_{0,T} = \frac{AC_T}{AC_0} - 1$. The trick here is that $FV_t = PV(1 + r)^t$, so $(1 + r)^t = \frac{FV_t}{PV}$.

```
returns_2020s.iloc[0]
```

```
AAPL    0.0228  
AMZN    0.0272  
F        0.0129  
META    0.0221  
TSLA    0.0285  
Name: 2020-01-02 00:00:00, dtype: float64
```

Note: We drop the last row in `prices['Adj Close']` with `.iloc[:-1]`. We drop this last row here here because we did the same above when we calculated `returns` to exclude possible intraday returns.

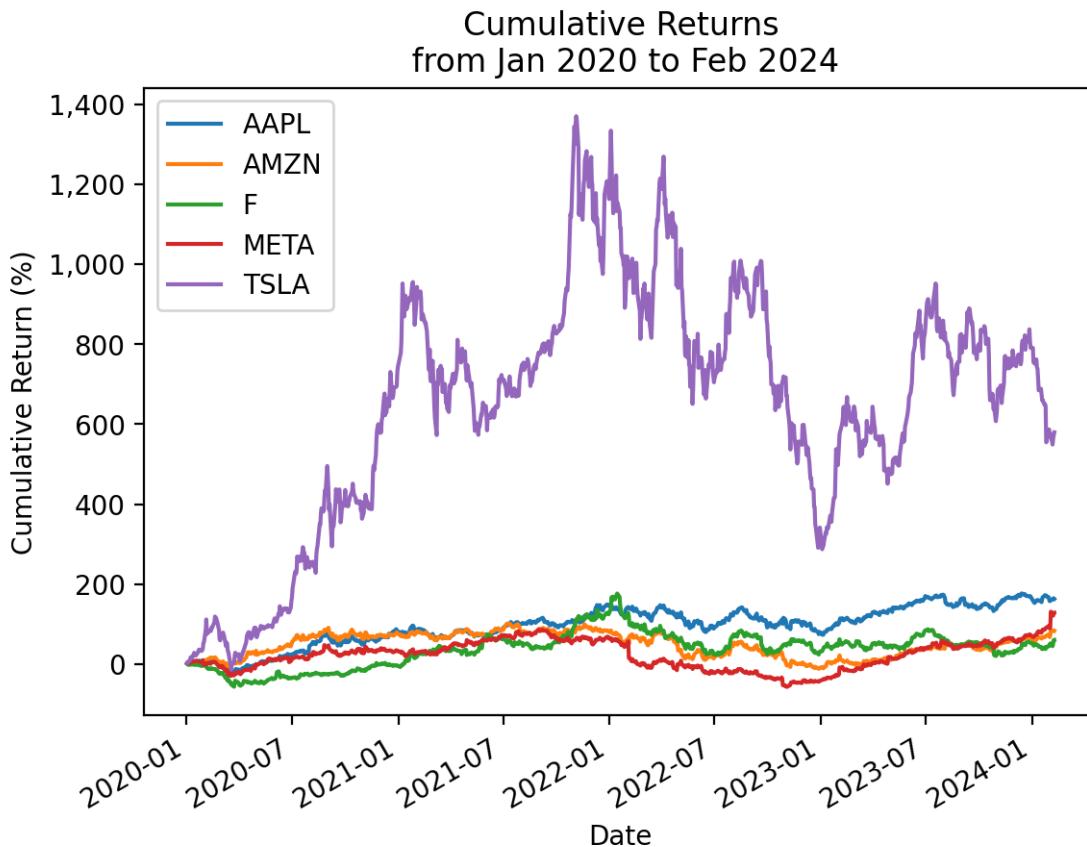
```
cumret_prices = prices['Adj Close'].loc['2020':].iloc[:-1] / prices['Adj Close'].loc['2019':]  
cumret_prices.iloc[0]
```

```
AAPL    0.0228  
AMZN    0.0272  
F        0.0129  
META    0.0221  
TSLA    0.0285  
Name: 2020-01-02 00:00:00, dtype: float64
```

```
np.allclose(cumret_cumprod, cumret_prices)
```

```
True
```

```
cumret_prices.mul(100).plot()  
  
# https://stackoverflow.com/questions/25973581/how-to-format-axis-number-format-to-thousan  
from matplotlib import ticker  
plt.gca().get_yaxis().set_major_formatter(ticker.FuncFormatter(lambda x, p: format(int(x),  
  
plt.ylabel('Cumulative Return (%)')  
plt.title(f'Cumulative Returns\n from {cumret_prices.index.min():%b %Y} to {cumret_prices.  
plt.show()
```



22.3.9 Calculate the Sharpe Ratio for TSLA

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

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

```
def calc_sharpe(ri, rf=ff['RF'], ppy=252):
    ri_rf = ri.sub(rf).dropna()
    return np.sqrt(ppy) * ri_rf.mean() / ri_rf.std()

calc_sharpe(ri=returns['TSLA'])
```

0.9261

```
calc_sharpe(ri=returns_2020s['TSLA'])
```

1.1212

We can use the `.pipe()` method here, too, since `ri` is the first argument to `calc_sharpe()`!

```
returns['TSLA'].pipe(calc_sharpe)
```

0.9261

```
returns_2020s['TSLA'].pipe(calc_sharpe)
```

1.1212

22.3.10 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 (i.e., $\beta_i = \frac{Cov(R_i - R_f, R_m - R_f)}{Var(R_m - R_f)}$) above for a univariate regression if we do not need goodness of fit statistics.

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

```
def calc_beta(ri, rf=ff['RF'], rm_rf=ff['Mkt-RF']):
    ri_rf = ri.sub(rf).dropna()
    rm_rf = rm_rf.loc[ri_rf.index]
    return ri_rf.cov(rm_rf) / rm_rf.var()
```

```
calc_beta(ri=returns['TSLA'])
```

1.4417

```
calc_beta(ri=returns_2020s['TSLA'])
```

1.5780

We can use the `.pipe()` method here, too, since `ri` is the first argument to `calc_beta()`!

```
    returns['TSLA'].pipe(calc_beta)

1.4417

    returns_2020s['TSLA'].pipe(calc_beta)

1.5780
```

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

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

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

How good were your guesses?

```
for i in returns_2020s:
    sharpe_i = returns_2020s[i].pipe(calc_sharpe)
    print(f'Sharpe Ratio for {i}:\t {sharpe_i:.2f}')

Sharpe Ratio for AAPL: 0.86
Sharpe Ratio for AMZN: 0.48
Sharpe Ratio for F: 0.42
Sharpe Ratio for META: 0.49
Sharpe Ratio for TSLA: 1.12
```

We can also use pandas notation to vectorize this calculation. First calculate *excess* returns as $r_i - r_f$.

```
returns_2020s_excess = returns_2020s.sub(FF['RF'], axis=0).dropna()
```

Then use pandas notation to calculate means, standard deviations, and annualize.

```
(

    returns_2020s_excess
    .mean()
    .div(returns_2020s_excess.std())
    .mul(np.sqrt(252))
```

```
)
```

```
AAPL    0.8576
AMZN    0.4750
F       0.4240
META    0.4949
TSLA    1.1212
dtype: float64
```

Note: In a few weeks we will learn the `.apply()` method, which avoids the loop syntax.

```
returns_2020s.apply(calc_sharpe)
```

```
AAPL    0.8576
AMZN    0.4750
F       0.4240
META    0.4949
TSLA    1.1212
dtype: float64
```

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

How good were your guesses?

```
for i in returns_2020s:
    beta_i = returns_2020s[i].pipe(calc_beta)
    print(f'Beta for {i}:\t {beta_i:.2f}')
```

```
Beta for AAPL:  1.15
Beta for AMZN:  1.04
Beta for F:    1.22
Beta for META:  1.27
Beta for TSLA:  1.58
```

Or we can follow out approach above to vectorize this calculation. First, we need to add a market excess return column to `returns_2020s_excess`.

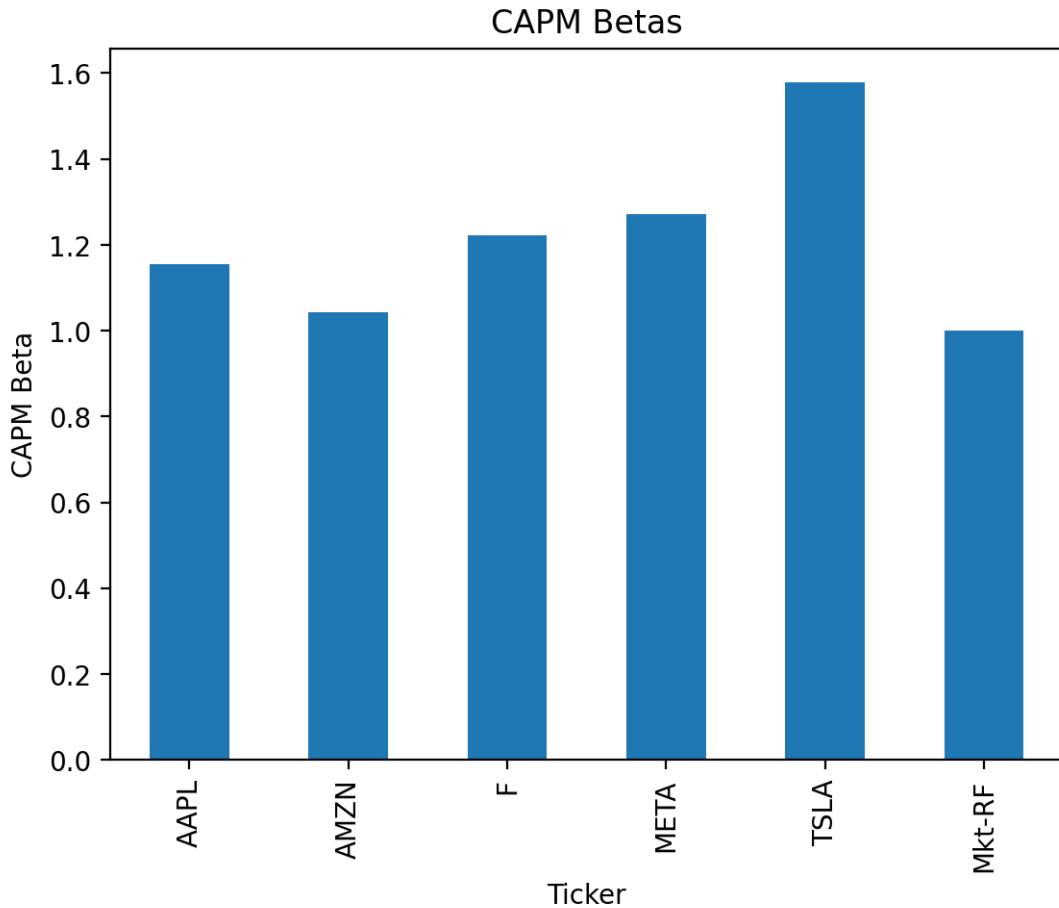
```
returns_2020s_excess['Mkt-RF'] = ff['Mkt-RF']
returns_2020s_excess.head()
```

Date	AAPL	AMZN	F	META	TSLA	Mkt-RF
2020-01-02	0.0228	0.0271	0.0128	0.0220	0.0285	0.0086
2020-01-03	-0.0098	-0.0122	-0.0224	-0.0054	0.0296	-0.0067
2020-01-06	0.0079	0.0148	-0.0055	0.0188	0.0192	0.0036
2020-01-07	-0.0048	0.0020	0.0098	0.0021	0.0387	-0.0019
2020-01-08	0.0160	-0.0079	-0.0001	0.0101	0.0491	0.0047

```
vcv = returns_2020s_excess.cov()
vcv
```

	AAPL	AMZN	F	META	TSLA	Mkt-RF
AAPL	0.0004	0.0003	0.0002	0.0004	0.0005	0.0003
AMZN	0.0003	0.0006	0.0002	0.0004	0.0005	0.0002
F	0.0002	0.0002	0.0009	0.0003	0.0005	0.0003
META	0.0004	0.0004	0.0003	0.0009	0.0005	0.0003
TSLA	0.0005	0.0005	0.0005	0.0005	0.0018	0.0003
Mkt-RF	0.0003	0.0002	0.0003	0.0003	0.0003	0.0002

```
vcv['Mkt-RF'].div(vcv.loc['Mkt-RF', 'Mkt-RF']).plot(kind='bar')
plt.xlabel('Ticker')
plt.ylabel('CAPM Beta')
plt.title('CAPM Betas')
plt.show()
```



Note: In a few weeks we will learn the `.apply()` method, which avoids the loop syntax.

```
returns_2020s.apply(calc_beta)
```

```
AAPL    1.1541
AMZN    1.0429
F        1.2231
META    1.2710
TSLA    1.5780
dtype: float64
```

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

What do you notice?

```
returns_2020s.mean(axis=1).pipe(calc_sharpe)
```

0.9573

Because diversification reduces portfolio standard deviation less than the sum of its parts, the Sharpe Ratio of the equally weighted portfolio is less than the equally weighted mean of the single-stock Sharpe Ratios.

```
returns_2020s.apply(calc_sharpe).mean()
```

0.6745

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

What do you notice?

Beta measures *nondiversifiable* risk, so $\beta_P = \sum w_i \beta_i$!

```
returns_2020s.mean(axis=1).pipe(calc_beta)
```

1.2538

```
returns_2020s.apply(calc_beta).mean()
```

1.2538

22.3.17 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!

```

betas = pd.DataFrame(
    index=range(1972, 2024),
    columns=returns.columns
)

betas.columns.name = 'Ticker'
betas.index.name = 'Year'

betas.tail()

```

Ticker	AAPL	AMZN	F	META	TSLA
Year					
2019	NaN	NaN	NaN	NaN	NaN
2020	NaN	NaN	NaN	NaN	NaN
2021	NaN	NaN	NaN	NaN	NaN
2022	NaN	NaN	NaN	NaN	NaN
2023	NaN	NaN	NaN	NaN	NaN

```

for i in betas.index:
    for c in betas.columns:
        betas.at[i, c] = returns.loc[str(i), c].pipe(calc_beta)

betas.tail()

```

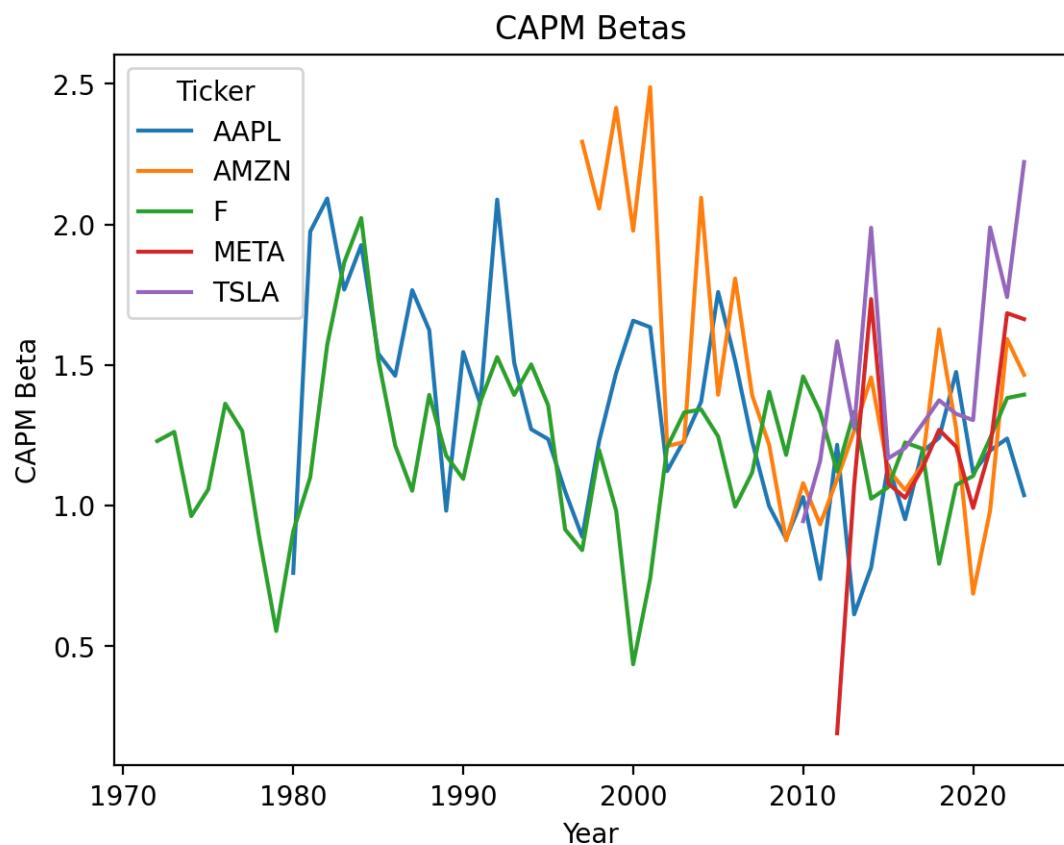
Ticker	AAPL	AMZN	F	META	TSLA
Year					
2019	1.4751	1.2752	1.0733	1.2094	1.3262
2020	1.1174	0.6866	1.1052	0.9913	1.3041
2021	1.1957	0.9822	1.2396	1.2014	1.9891
2022	1.2386	1.5922	1.3824	1.6843	1.7414
2023	1.0369	1.4649	1.3947	1.6630	2.2218

22.3.18 Plot the time series of market betas

```

betas.plot()
plt.ylabel('CAPM Beta')
plt.title('CAPM Betas')
plt.show()

```



23 Herron Topic 1 - Practice for Section 03

23.1 Announcements

1. DataCamp
 1. *Data Manipulation with pandas* due by Friday, 2/9, at 11:59 PM
 2. *Joining Data with pandas* due by Friday, 2/16, at 11:59 PM
 3. *Earn 10,000 XP* due by Friday, 3/15, at 11:59 PM
2. I posted Project 1 to Canvas
 1. Slides and notebook due by Friday, 2/23, at 11:59 PM
 2. Keep joining teams and let me know if you need help

23.2 10-Minute Recap

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import pandas_datareader as pdr
import yfinance as yf

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

First, we can easily download data in Python with two packages:

1. `yfinance` to download data from Yahoo! Finance
2. `pandas-datareader` to download data from Ken French (and FRED and others)

Second, there are “simple returns” and “log returns”

1. Simple returns are the returns that investors receive that we learned in FINA 6331 and FINA 6333: $r_t = \frac{p_t + d_t - p_{t-1}}{p_{t-1}}$

2. Log returns are the log of one plus simple returns. Why do we use them?

1. ***Log returns are additive***, while simple returns are multiplicative. This additive property makes math really easy with log returns: $\log(\prod_{t=0}^T (1+r_t)) = \sum_{t=0}^T \log(1+r_t)$, so $r_{0,T} = \prod_{t=0}^T (1+r_t) - 1 = e^{\sum_{t=0}^T \log(1+r_t)} - 1$
2. ***Log returns are almost normally distributed***

We will almost always use simple returns. The exception is time-consuming calculations, which we will often do in log returns to save us time.

Third, we can easily calculate portfolio returns in pandas!

1. `returns.mean(axis=1)` is the equally-weighted portfolio return for the stocks in `returns`, ***rebalanced at the same frequency as the returns in returns!***
2. `returns.dot(weights)` lets us use weights in array `weights` instead of equally-weighted, but is still ***rebalanced at the same frequency as the returns in returns!***

23.3 Practice

23.3.1 Download all available daily price data for tickers TSLA, F, AAPL, AMZN, and META to data frame prices

```
tickers = 'TSLA F AAPL AMZN META'
prices = yf.download(tickers=tickers)

prices
```

[*****100%*****] 5 of 5 completed

Date	Adj Close					Close			
	AAPL	AMZN	F	META	TSLA	AAPL	AMZN	F	META
1972-06-01	NaN	NaN	0.2419	NaN	NaN	NaN	NaN	2.1532	NaN
1972-06-02	NaN	NaN	0.2414	NaN	NaN	NaN	NaN	2.1492	NaN
1972-06-05	NaN	NaN	0.2414	NaN	NaN	NaN	NaN	2.1492	NaN
1972-06-06	NaN	NaN	0.2387	NaN	NaN	NaN	NaN	2.1248	NaN
1972-06-07	NaN	NaN	0.2373	NaN	NaN	NaN	NaN	2.1127	NaN
...
2024-02-05	187.6800	170.3100	11.5900	459.4100	181.0600	187.6800	170.3100	11.5900	459.4100
2024-02-06	189.3000	169.1500	12.0700	454.7200	185.1000	189.3000	169.1500	12.0700	454.7200

Date	Adj Close					Close			
	AAPL	AMZN	F	META	TSLA	AAPL	AMZN	F	META
2024-02-07	189.4100	170.5300	12.8000	469.5900	187.5800	189.4100	170.5300	12.8000	469.5900
2024-02-08	188.3200	169.8400	12.8300	470.0000	189.5600	188.3200	169.8400	12.8300	470.0000
2024-02-09	188.8500	174.4500	12.6800	468.1100	193.5700	188.8500	174.4500	12.6800	468.1100

23.3.2 Calculate all available daily returns and save to data frame returns

```

returns = (
    prices['Adj Close'] # slice adj close
    .iloc[:-1] # drop the last price because it might be intraday (i.e., not a close)
    .pct_change() # calculate simple returns
)

returns

```

Date	AAPL	AMZN	F	META	TSLA
1972-06-01	NaN	NaN	NaN	NaN	NaN
1972-06-02	NaN	NaN	-0.0019	NaN	NaN
1972-06-05	NaN	NaN	0.0000	NaN	NaN
1972-06-06	NaN	NaN	-0.0113	NaN	NaN
1972-06-07	NaN	NaN	-0.0057	NaN	NaN
...
2024-02-02	-0.0054	0.0787	0.0033	0.2032	-0.0050
2024-02-05	0.0098	-0.0087	-0.0453	-0.0328	-0.0365
2024-02-06	0.0086	-0.0068	0.0414	-0.0102	0.0223
2024-02-07	0.0006	0.0082	0.0605	0.0327	0.0134
2024-02-08	-0.0058	-0.0040	0.0023	0.0009	0.0106

23.3.3 Slices returns for the 2020s and assign to returns_2020s

```

returns_2020s = returns.loc['2020':] # always use an unambiguous date format, like YYYY-MM-
returns_2020s

```

Date	AAPL	AMZN	F	META	TSLA
2020-01-02	0.0228	0.0272	0.0129	0.0221	0.0285
2020-01-03	-0.0097	-0.0121	-0.0223	-0.0053	0.0296
2020-01-06	0.0080	0.0149	-0.0054	0.0188	0.0193
2020-01-07	-0.0047	0.0021	0.0098	0.0022	0.0388
2020-01-08	0.0161	-0.0078	0.0000	0.0101	0.0492
...
2024-02-02	-0.0054	0.0787	0.0033	0.2032	-0.0050
2024-02-05	0.0098	-0.0087	-0.0453	-0.0328	-0.0365
2024-02-06	0.0086	-0.0068	0.0414	-0.0102	0.0223
2024-02-07	0.0006	0.0082	0.0605	0.0327	0.0134
2024-02-08	-0.0058	-0.0040	0.0023	0.0009	0.0106

23.3.4 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']

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

```
C:\Users\r.herron\AppData\Local\Temp\ipykernel_27592\2526882917.py:1: FutureWarning: The arg
ff_all = pdr.DataReader(
```

The DESCR key in the dictionary tells us about the data frames that pandas-datareader returns.

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

F-F Research Data Factors daily

This file was created by CMPT_ME_BEME_RETURNS_DAILY using the 202312 CRSP database. The Tbill r

0 : (25649 rows x 4 cols)

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

```
ff = ff_all[0].div(100)
```

```
ff
```

Date	Mkt-RF	SMB	HML	RF
1926-07-01	0.0010	-0.0025	-0.0027	0.0001
1926-07-02	0.0045	-0.0033	-0.0006	0.0001
1926-07-06	0.0017	0.0030	-0.0039	0.0001
1926-07-07	0.0009	-0.0058	0.0002	0.0001
1926-07-08	0.0021	-0.0038	0.0019	0.0001
...
2023-12-22	0.0021	0.0064	0.0009	0.0002
2023-12-26	0.0048	0.0069	0.0046	0.0002
2023-12-27	0.0016	0.0014	0.0012	0.0002
2023-12-28	-0.0001	-0.0036	0.0003	0.0002
2023-12-29	-0.0043	-0.0112	-0.0037	0.0002

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

We use the .prod() method to calculate *total* returns, because $r_{total} = r_{0,T} = [\prod_{t=0}^T (1 + r_t)] - 1$.

```
(  
    returns_2020s # returns during the 2020s
```

```
.add(1) # add 1 before we compound
.prod() # compound all returns
.sub(1) # subtract 1 to recover total returns
)

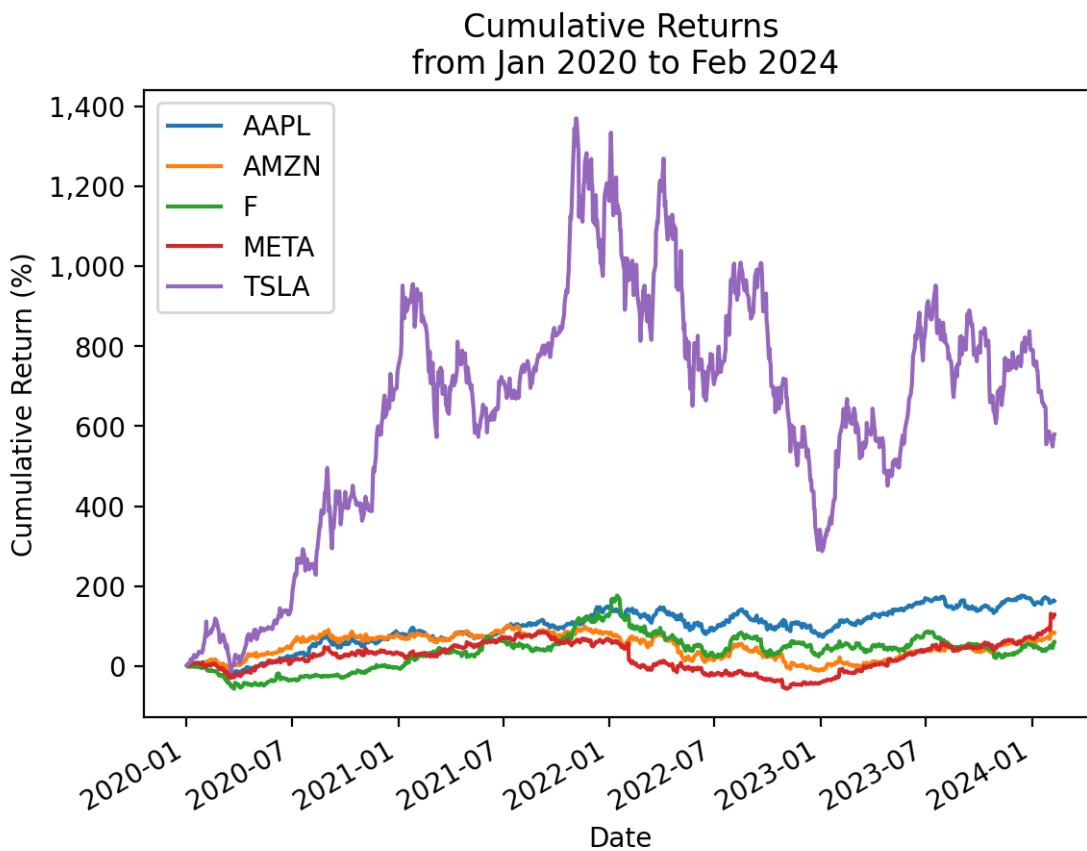
AAPL    1.6331
AMZN    0.8383
F       0.6090
META    1.2899
TSLA    5.7970
dtype: float64
```

```
cumret_cumprod = (
    returns_2020s
    .add(1)
    .cumprod()
    .sub(1)
)

cumret_cumprod.mul(100).plot()

# https://stackoverflow.com/questions/25973581/how-to-format-axis-number-format-to-thousan
from matplotlib import ticker
plt.gca().get_yaxis().set_major_formatter(ticker.FuncFormatter(lambda x, p: format(int(x),

plt.ylabel('Cumulative Return (%)')
plt.title(f'Cumulative Returns\n from {returns_2020s.index.min():%b %Y} to {returns_2020s.
plt.show()
```



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

```

cumret_cumsum = (
    returns_2020s # start with simple returns
    .pipe(np.log1p) # convert to log returns
    .cumsum() # log returns are additive
    .pipe(np.expm1) # convert back to simple returns
)
np.allclose(cumret_cumsum, cumret_cumprod)

```

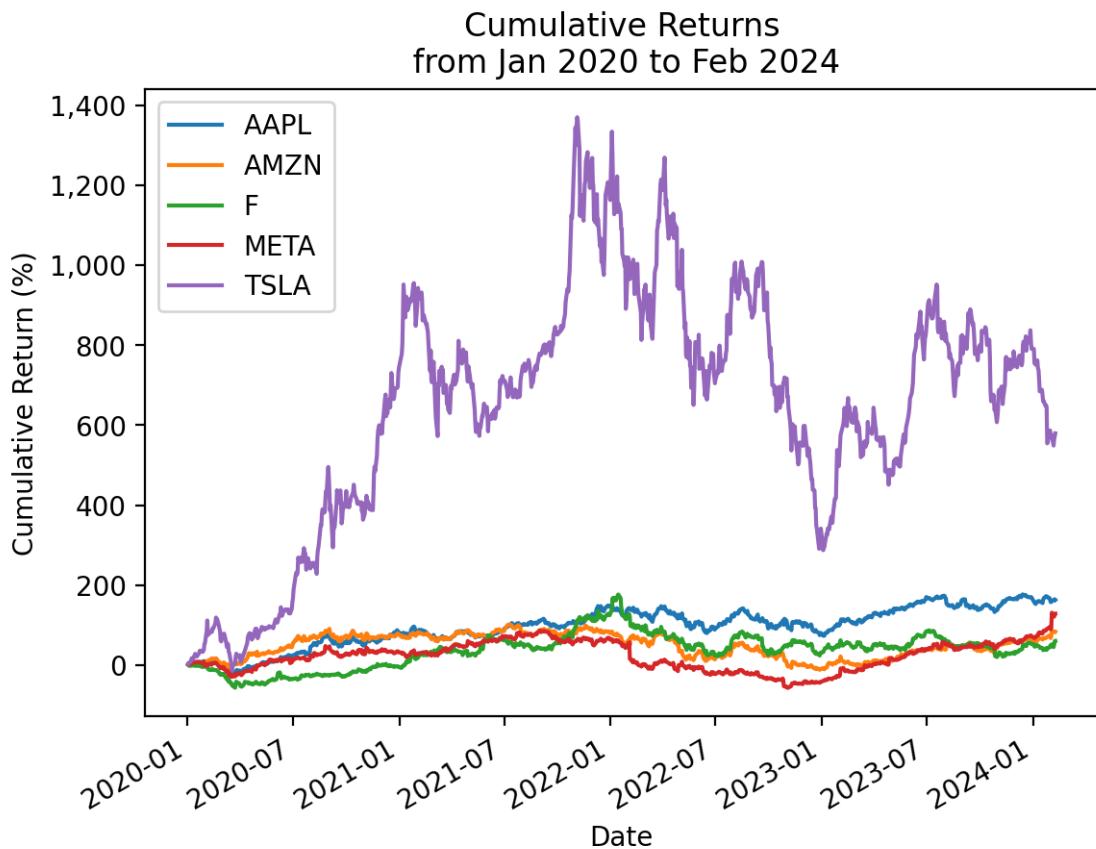
True

```
np.allclose(1*10***-9, 1.1*10***-9)
```

```
True
```

```
cumret_cumsum.mul(100).plot()
```

```
# https://stackoverflow.com/questions/25973581/how-to-format-axis-number-format-to-thousan
from matplotlib import ticker
plt.gca().get_yaxis().set_major_formatter(ticker.FuncFormatter(lambda x, p: format(int(x),
plt.ylabel('Cumulative Return (%)')
plt.title(f'Cumulative Returns\n from {returns_2020s.index.min():%b %Y} to {returns_2020s.i
plt.show()
```



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

We can also calculate cumulative returns as the ratio of adjusted closes. That is $R_{0,T} = \frac{AC_T}{AC_0} - 1$. The trick here is that $FV_t = PV(1 + r)^t$, so $(1 + r)^t = \frac{FV_t}{PV}$.

```
returns_2020s.iloc[0]
```

```
AAPL    0.0228
AMZN    0.0272
F        0.0129
META    0.0221
TSLA    0.0285
Name: 2020-01-02 00:00:00, dtype: float64
```

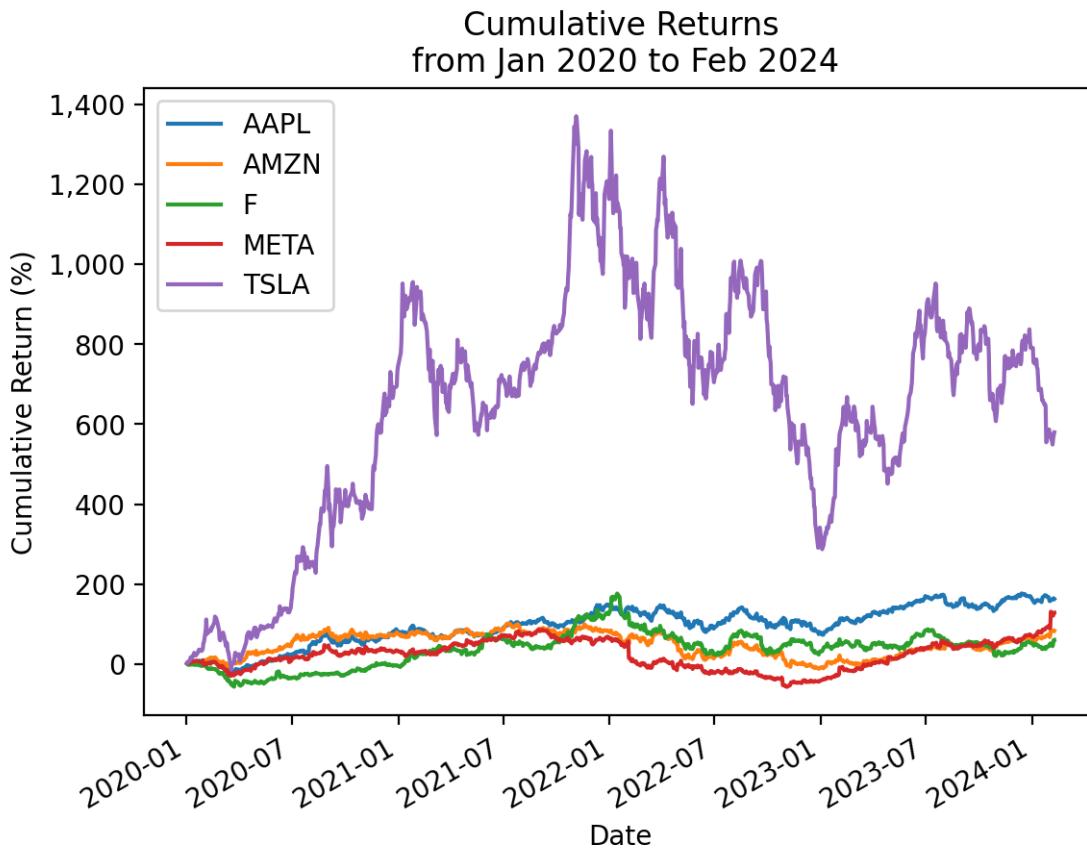
```
cumret_prices = prices['Adj Close'].loc['2020':].iloc[:-1] / prices['Adj Close'].loc['2019':0]
np.allclose(cumret_prices, cumret_cumprod)
```

```
True
```

```
cumret_prices.mul(100).plot()

# https://stackoverflow.com/questions/25973581/how-to-format-axis-number-format-to-thousand
from matplotlib import ticker
plt.gca().get_yaxis().set_major_formatter(ticker.FuncFormatter(lambda x, p: format(int(x), ',')))

plt.ylabel('Cumulative Return (%)')
plt.title(f'Cumulative Returns\n from {returns_2020s.index.min():%b %Y} to {returns_2020s.index.max():%b %Y}')
plt.show()
```



23.3.9 Calculate the Sharpe Ratio for TSLA

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

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

```
def calc_sharpe(ri, rf=ff['RF'], ppy=252):
    ri_rf = ri.sub(rf).dropna()
    return np.sqrt(ppy) * ri_rf.mean() / ri_rf.std()

calc_sharpe(ri=returns['TSLA'])
```

0.9261

```
calc_sharpe(ri=returns_2020s['TSLA'])
```

1.1212

Or, we can use the .pipe() method because ri is the first argument to calc_sharpe()!

```
returns['TSLA'].pipe(calc_sharpe)
```

0.9261

```
returns_2020s['TSLA'].pipe(calc_sharpe)
```

1.1212

23.3.10 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 (i.e., $\beta_i = \frac{Cov(R_i - R_f, R_m - R_f)}{Var(R_m - R_f)}$) above for a univariate regression if we do not need goodness of fit statistics.

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

```
def calc_beta(ri, rf=ff['RF'], rm_rf=ff['Mkt-RF']):
    ri_rf = ri.sub(rf).dropna()
    return ri_rf.cov(rm_rf) / rm_rf.loc[ri_rf.index].var()
```

```
calc_beta(ri=returns['TSLA'])
```

1.4417

```
calc_beta(ri=returns_2020s['TSLA'])
```

1.5780

Or, we can use the `.pipe()` method because `ri` is the first argument to `calc_sharpe()`!

```
returns['TSLA'].pipe(calc_beta)
```

1.4417

```
returns_2020s['TSLA'].pipe(calc_beta)
```

1.5780

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

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

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

How good were your guesses?

```
for i in returns_2020s:  
    sharpe_i = returns_2020s[i].pipe(calc_sharpe)  
    print(f'Sharpe Ratio for {i}:\t{sharpe_i:.2f}')
```

```
Sharpe Ratio for AAPL: 0.86  
Sharpe Ratio for AMZN: 0.48  
Sharpe Ratio for F: 0.42  
Sharpe Ratio for META: 0.49  
Sharpe Ratio for TSLA: 1.12
```

We can also use pandas notation to vectorize this calculation. First calculate *excess* returns as $r_i - r_f$.

```
returns_2020s_excess = returns_2020s.sub(FF['RF'], axis=0).dropna()
```

Then use pandas notation to calculate means, standard deviations, and annualize.

```
(  
    returns_2020s_excess
```

```
.mean()  
.div(returns_2020s_excess.std())  
.mul(np.sqrt(252))  
)
```

```
AAPL    0.8576  
AMZN    0.4750  
F        0.4240  
META    0.4949  
TSLA    1.1212  

```

Note: In a few weeks we will learn the `.apply()` method, which avoids the loop syntax.

```
returns_2020s.apply(calc_sharpe)
```

```
AAPL    0.8576  
AMZN    0.4750  
F        0.4240  
META    0.4949  
TSLA    1.1212  
dtype: float64
```

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

How good were your guesses?

```
for i in returns_2020s:  
    beta_i = returns_2020s[i].pipe(calc_beta)  
    print(f'Beta for {i}:\t{beta_i:.2f}')
```

```
Beta for AAPL:    1.15  
Beta for AMZN:   1.04  
Beta for F:      1.22  
Beta for META:   1.27  
Beta for TSLA:   1.58
```

Or we can follow out approach above to vectorize this calculation. First, we need to add a market excess return column to `returns_2020s_excess`.

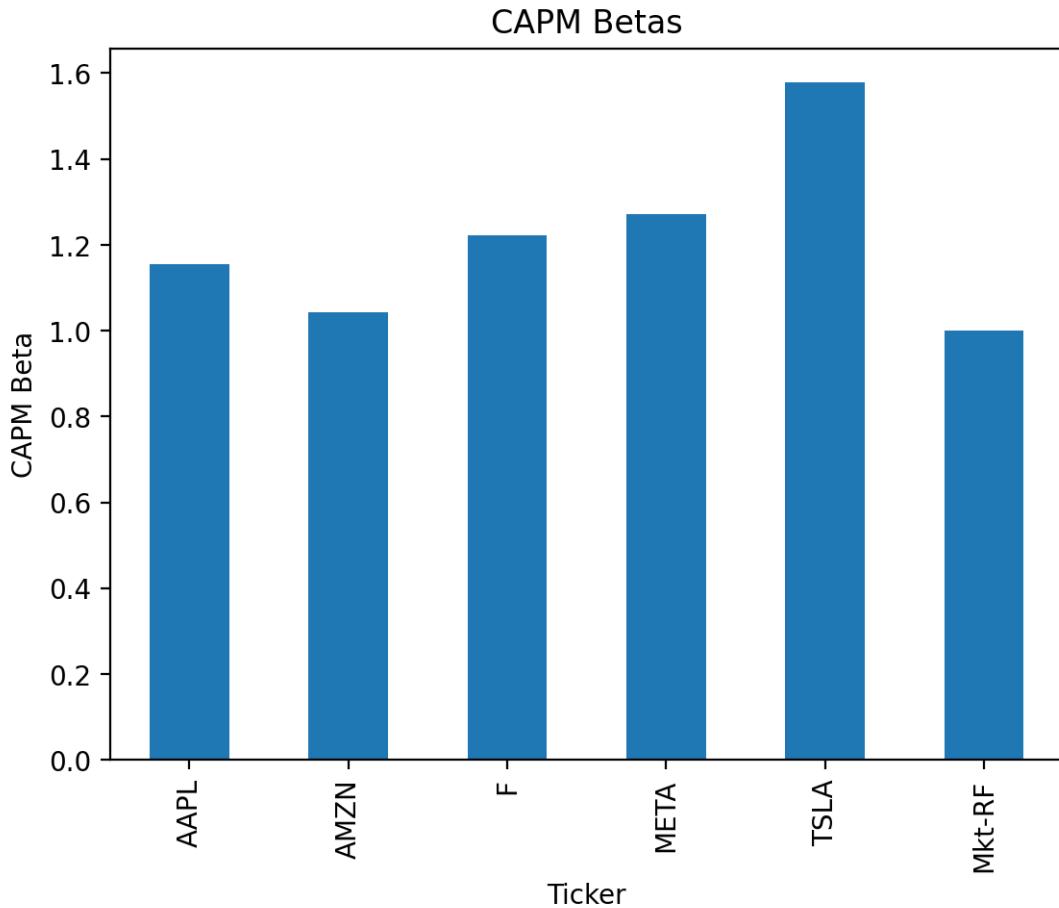
```
returns_2020s_excess['Mkt-RF'] = ff['Mkt-RF']
returns_2020s_excess.head()
```

Date	AAPL	AMZN	F	META	TSLA	Mkt-RF
2020-01-02	0.0228	0.0271	0.0128	0.0220	0.0285	0.0086
2020-01-03	-0.0098	-0.0122	-0.0224	-0.0054	0.0296	-0.0067
2020-01-06	0.0079	0.0148	-0.0055	0.0188	0.0192	0.0036
2020-01-07	-0.0048	0.0020	0.0098	0.0021	0.0387	-0.0019
2020-01-08	0.0160	-0.0079	-0.0001	0.0101	0.0491	0.0047

```
vcv = returns_2020s_excess.cov()
vcv
```

	AAPL	AMZN	F	META	TSLA	Mkt-RF
AAPL	0.0004	0.0003	0.0002	0.0004	0.0005	0.0003
AMZN	0.0003	0.0006	0.0002	0.0004	0.0005	0.0002
F	0.0002	0.0002	0.0009	0.0003	0.0005	0.0003
META	0.0004	0.0004	0.0003	0.0009	0.0005	0.0003
TSLA	0.0005	0.0005	0.0005	0.0005	0.0018	0.0003
Mkt-RF	0.0003	0.0002	0.0003	0.0003	0.0003	0.0002

```
vcv['Mkt-RF'].div(vcv.loc['Mkt-RF', 'Mkt-RF']).plot(kind='bar')
plt.xlabel('Ticker')
plt.ylabel('CAPM Beta')
plt.title('CAPM Betas')
plt.show()
```



Note: In a few weeks we will learn the `.apply()` method, which avoids the loop syntax.

```
returns_2020s.apply(calc_beta)
```

```
AAPL    1.1541
AMZN    1.0429
F        1.2231
META    1.2710
TSLA    1.5780
dtype: float64
```

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

What do you notice?

```
returns_2020s.mean(axis=1).pipe(calc_sharpe)
```

0.9573

Because diversification reduces portfolio standard deviation less than the sum of its parts, the Sharpe Ratio of the equally weighted portfolio is less than the equally weighted mean of the single-stock Sharpe Ratios.

```
returns_2020s.apply(calc_sharpe).mean()
```

0.6745

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

What do you notice?

Beta measures *nondiversifiable* risk, so $\beta_P = \sum w_i \beta_i$!

```
returns_2020s.mean(axis=1).pipe(calc_beta)
```

1.2538

```
returns_2020s.apply(calc_beta).mean()
```

1.2538

23.3.17 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!

```

betas = pd.DataFrame(
    index=range(1972, 2024),
    columns=returns.columns
)

betas.columns.name = 'Ticker'
betas.index.name = 'Year'

betas.tail()

```

Ticker	AAPL	AMZN	F	META	TSLA
Year					
2019	NaN	NaN	NaN	NaN	NaN
2020	NaN	NaN	NaN	NaN	NaN
2021	NaN	NaN	NaN	NaN	NaN
2022	NaN	NaN	NaN	NaN	NaN
2023	NaN	NaN	NaN	NaN	NaN

```

for i in betas.index:
    for c in betas.columns:
        betas.at[i, c] = returns.loc[str(i), c].pipe(calc_beta)

betas.tail()

```

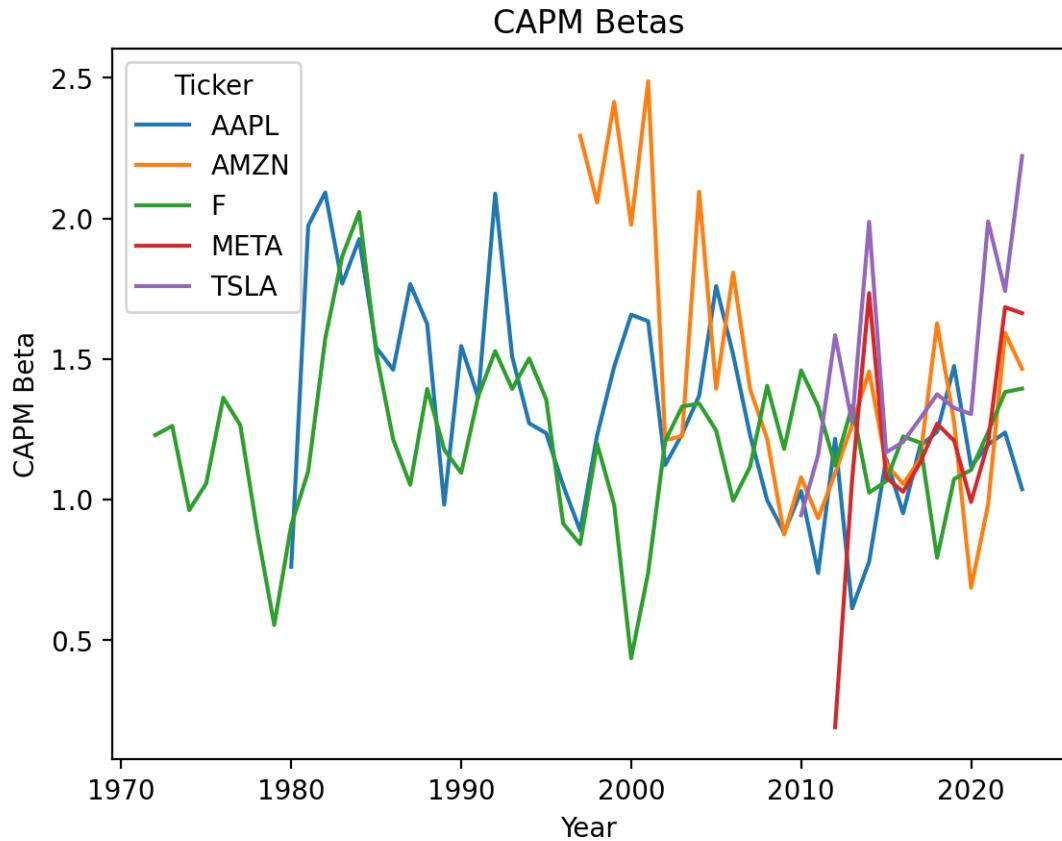
Ticker	AAPL	AMZN	F	META	TSLA
Year					
2019	1.4751	1.2752	1.0733	1.2094	1.3262
2020	1.1174	0.6866	1.1052	0.9913	1.3041
2021	1.1957	0.9822	1.2396	1.2014	1.9891
2022	1.2386	1.5922	1.3824	1.6843	1.7414
2023	1.0369	1.4649	1.3947	1.6630	2.2218

23.3.18 Plot the time series of market betas

```

betas.plot()
plt.ylabel('CAPM Beta')
plt.title('CAPM Betas')
plt.show()

```



24 Herron Topic 1 - Practice for Section 04

24.1 Announcements

1. DataCamp
 1. *Data Manipulation with pandas* due by Friday, 2/9, at 11:59 PM
 2. *Joining Data with pandas* due by Friday, 2/16, at 11:59 PM
 3. *Earn 10,000 XP* due by Friday, 3/15, at 11:59 PM
2. I posted Project 1 to Canvas
 1. Slides and notebook due by Friday, 2/23, at 11:59 PM
 2. Keep joining teams and let me know if you need help

24.2 10-Minute Recap

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import pandas_datareader as pdr
import yfinance as yf

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

np.expm1(np.log1p(0.1))
```

0.1000

First, we will use two packages to download data from the web:

1. `yfinance` for Yahoo! Finance data

2. `pandas-datareader` for Ken French data (or FRED data to others)

Second, there are “simple returns” and “log returns”

1. Simple returns are the returns that investors receive that we learned in FINA 6331 and FINA 6333: $r_t = \frac{p_t + d_t - p_{t-1}}{p_{t-1}}$
2. Log returns are the log of one plus simple returns. Why do we use them?
 1. ***Log returns are additive***, while simple returns are multiplicative. This additive property makes math really easy with log returns: $\log(\prod_{t=0}^T (1+r_t)) = \sum_{t=0}^T \log(1+r_t)$, so $r_{0,T} = \prod_{t=0}^T (1+r_t) - 1 = e^{\sum_{t=0}^T \log(1+r_t)} - 1$
 2. ***Log returns are almost normally distributed***

We will almost always use simple returns. The exception is time-consuming calculations, which we will often do in log returns to save us time.

Third, we can calculate portfolio returns easily in pandas!

1. `returns.mean(axis=1)` is the return on an equally-weighted portfolio, ***rebalanced at the same frequency as the returns in returns!***
2. `returns.dot(weights)` is the return on a portfolio weighted according to the weights in the `weights` array, ***still rebalanced at the same frequency as the returns in returns!***

24.3 Practice

24.3.1 Download all available daily price data for tickers TSLA, F, AAPL, AMZN, and META to data frame prices

```
tickers = 'TSLA F AAPL AMZN META'
prices = yf.download(tickers=tickers)

prices
```

[*****100%*****] 5 of 5 completed

Date	Adj Close					Close			
	AAPL	AMZN	F	META	TSLA	AAPL	AMZN	F	META
1972-06-01	NaN	NaN	0.2419	NaN	NaN	NaN	NaN	2.1532	NaN

Date	Adj Close					Close			
	AAPL	AMZN	F	META	TSLA	AAPL	AMZN	F	META
1972-06-02	NaN	NaN	0.2414	NaN	NaN	NaN	NaN	2.1492	NaN
1972-06-05	NaN	NaN	0.2414	NaN	NaN	NaN	NaN	2.1492	NaN
1972-06-06	NaN	NaN	0.2387	NaN	NaN	NaN	NaN	2.1248	NaN
1972-06-07	NaN	NaN	0.2373	NaN	NaN	NaN	NaN	2.1127	NaN
...
2024-02-05	187.6800	170.3100	11.5900	459.4100	181.0600	187.6800	170.3100	11.5900	459.4100
2024-02-06	189.3000	169.1500	12.0700	454.7200	185.1000	189.3000	169.1500	12.0700	454.7200
2024-02-07	189.4100	170.5300	12.8000	469.5900	187.5800	189.4100	170.5300	12.8000	469.5900
2024-02-08	188.3200	169.8400	12.8300	470.0000	189.5600	188.3200	169.8400	12.8300	470.0000
2024-02-09	188.8500	174.4500	12.6800	468.1100	193.5700	188.8500	174.4500	12.6800	468.1100

24.3.2 Calculate all available daily returns and save to data frame returns

24.3.3 Calculate all available daily returns and save to data frame returns

```

returns = (
    prices['Adj Close'] # slice adj close
    .iloc[:-1] # drop the last price because it might be intraday (i.e., not a close)
    .pct_change() # calculate simple returns
)

returns

```

Date	AAPL AMZN F META TSLA				
	AAPL	AMZN	F	META	TSLA
1972-06-01	NaN	NaN	NaN	NaN	NaN
1972-06-02	NaN	NaN	-0.0019	NaN	NaN
1972-06-05	NaN	NaN	0.0000	NaN	NaN
1972-06-06	NaN	NaN	-0.0113	NaN	NaN
1972-06-07	NaN	NaN	-0.0057	NaN	NaN
...
2024-02-02	-0.0054	0.0787	0.0033	0.2032	-0.0050
2024-02-05	0.0098	-0.0087	-0.0453	-0.0328	-0.0365
2024-02-06	0.0086	-0.0068	0.0414	-0.0102	0.0223
2024-02-07	0.0006	0.0082	0.0605	0.0327	0.0134
2024-02-08	-0.0058	-0.0040	0.0023	0.0009	0.0106

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

```
returns_2020s = returns.loc['2020':] # always use an unambiguous date format, like YYYY-MM-  
returns_2020s
```

Date	AAPL	AMZN	F	META	TSLA
2020-01-02	0.0228	0.0272	0.0129	0.0221	0.0285
2020-01-03	-0.0097	-0.0121	-0.0223	-0.0053	0.0296
2020-01-06	0.0080	0.0149	-0.0054	0.0188	0.0193
2020-01-07	-0.0047	0.0021	0.0098	0.0022	0.0388
2020-01-08	0.0161	-0.0078	0.0000	0.0101	0.0492
...
2024-02-02	-0.0054	0.0787	0.0033	0.2032	-0.0050
2024-02-05	0.0098	-0.0087	-0.0453	-0.0328	-0.0365
2024-02-06	0.0086	-0.0068	0.0414	-0.0102	0.0223
2024-02-07	0.0006	0.0082	0.0605	0.0327	0.0134
2024-02-08	-0.0058	-0.0040	0.0023	0.0009	0.0106

24.3.5 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']
```

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

```
)
```

```
C:\Users\r.herron\AppData\Local\Temp\ipykernel_22508\2526882917.py:1: FutureWarning: The arg  
ff_all = pdr.DataReader(  
  
    type(ff_all)  
  
dict
```

The DESCR key in the dictionary tells us about the data frames that pandas-datareader returns.

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

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

```
This file was created by CMPT_ME_BEME_RET_DAILY using the 202312 CRSP database. The Tbill re  
0 : (25649 rows x 4 cols)
```

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

```
ff = ff_all[0].div(100)
```

```
ff
```

Date	Mkt-RF	SMB	HML	RF
1926-07-01	0.0010	-0.0025	-0.0027	0.0001
1926-07-02	0.0045	-0.0033	-0.0006	0.0001
1926-07-06	0.0017	0.0030	-0.0039	0.0001
1926-07-07	0.0009	-0.0058	0.0002	0.0001
1926-07-08	0.0021	-0.0038	0.0019	0.0001
...

Date	Mkt-RF	SMB	HML	RF
2023-12-22	0.0021	0.0064	0.0009	0.0002
2023-12-26	0.0048	0.0069	0.0046	0.0002
2023-12-27	0.0016	0.0014	0.0012	0.0002
2023-12-28	-0.0001	-0.0036	0.0003	0.0002
2023-12-29	-0.0043	-0.0112	-0.0037	0.0002

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

We use the `.prod()` method to calculate *total* returns, because $r_{total} = r_{0,T} = [\prod_{t=0}^T (1 + r_t)] - 1$.

```
totret = (
    returns_2020s # returns during the 2020s
    .add(1) # add 1 before we compound
    .prod() # compound all returns
    .sub(1) # subtract 1 to recover total returns
)

totret
```

```
AAPL    1.6331
AMZN    0.8383
F       0.6090
META    1.2899
TSLA    5.7970
dtype: float64
```

```
cumret_cumprod = (
    returns_2020s # returns during the 2020s
    .add(1) # add 1 before we compound
    .cumprod() # compound all returns
    .sub(1) # subtract 1 to recover total returns
)

cumret_cumprod.tail()
```

Date	AAPL	AMZN	F	META	TSLA
2024-02-02	1.5985	0.8596	0.5224	1.3142	5.7379
2024-02-05	1.6241	0.8433	0.4535	1.2383	5.4922
2024-02-06	1.6468	0.8308	0.5136	1.2154	5.6371
2024-02-07	1.6483	0.8457	0.6052	1.2879	5.7260
2024-02-08	1.6331	0.8383	0.6090	1.2899	5.7970

```

np.allclose(
    totret, # total returns from 2020 through 2023
    cumret_cumprod.iloc[-1] # cumulative returns on last day of 2023
)

```

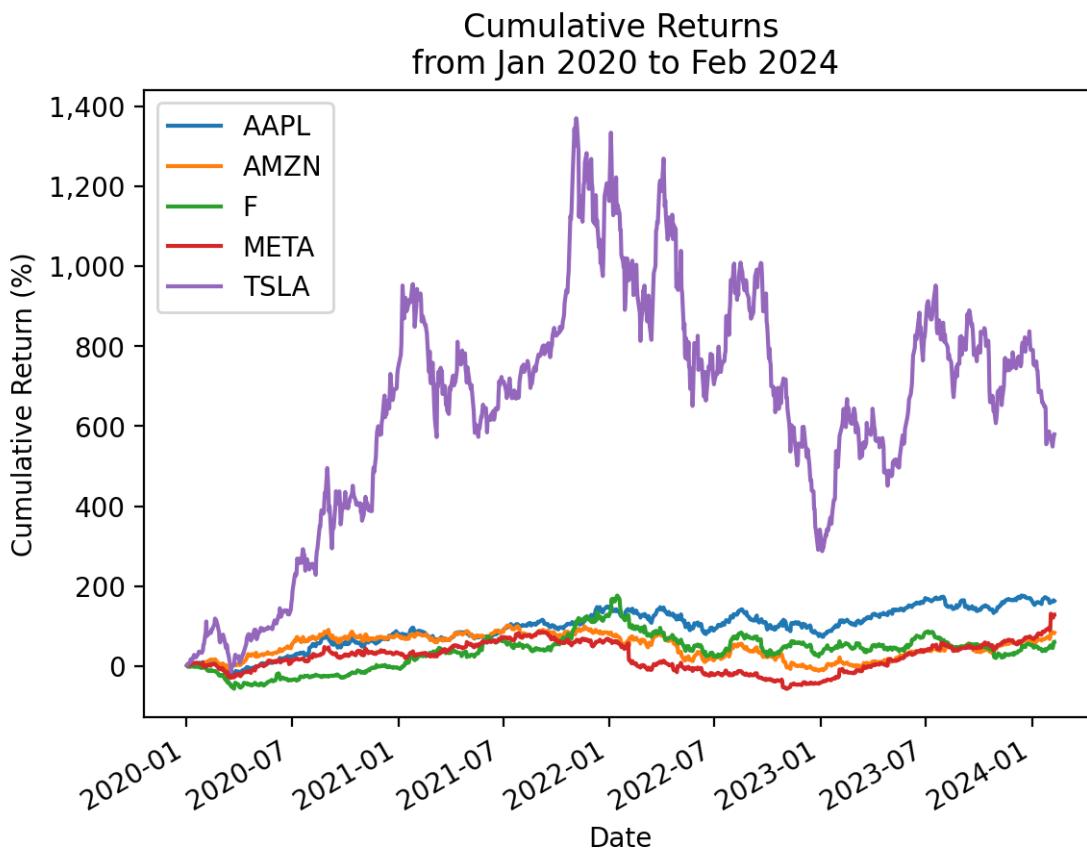
True

```

cumret_cumprod.mul(100).plot()

# https://stackoverflow.com/questions/25973581/how-to-format-axis-number-format-to-thousan
from matplotlib import ticker
plt.gca().get_yaxis().set_major_formatter(ticker.FuncFormatter(lambda x, p: format(int(x),
plt.ylabel('Cumulative Return (%)')
plt.title(f'Cumulative Returns\n from {returns_2020s.index.min():%b %Y} to {returns_2020s.
plt.show()

```



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

The log of products is the sum of logs!

```

cumret_cumsum = (
    returns_2020s # returns during the 2020s
    .pipe(np.log1p) # convert simple returns to log returns
    .cumsum() # compound all returns
    .pipe(np.expm1) # convert log returns to simple returns
)

cumret_cumsum.tail()

```

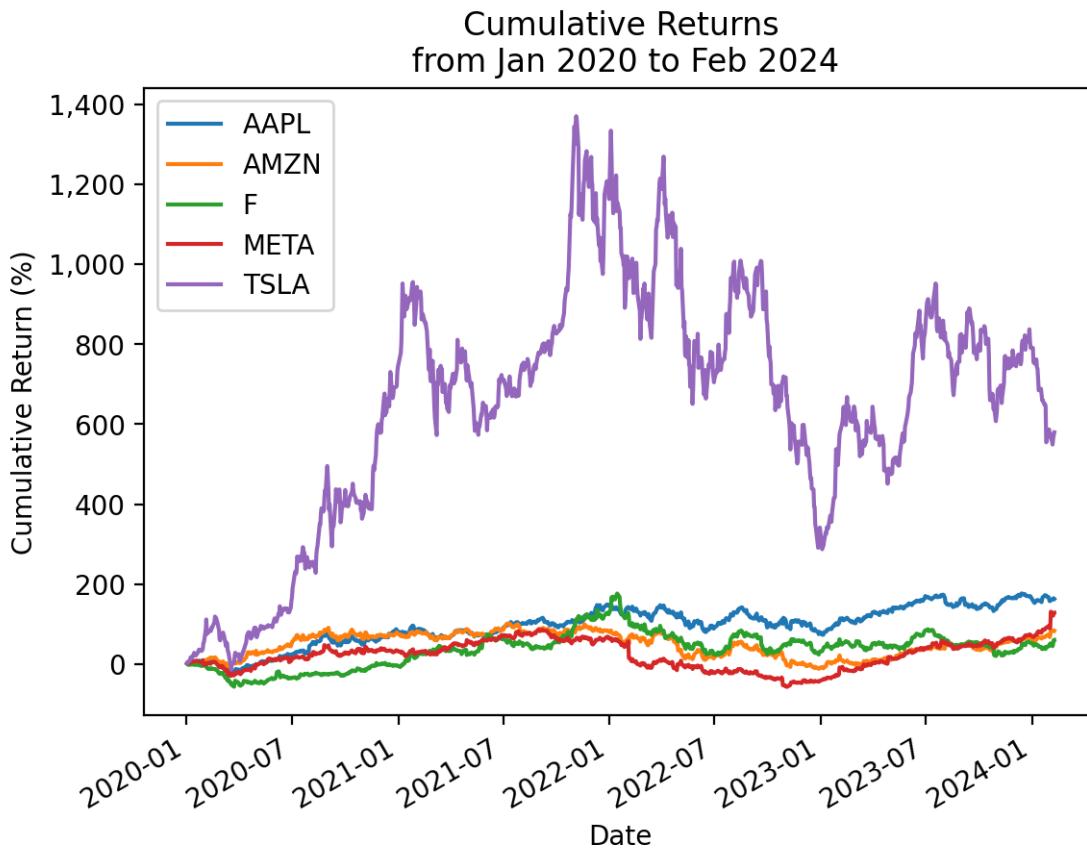
Date	AAPL	AMZN	F	META	TSLA
2024-02-02	1.5985	0.8596	0.5224	1.3142	5.7379
2024-02-05	1.6241	0.8433	0.4535	1.2383	5.4922
2024-02-06	1.6468	0.8308	0.5136	1.2154	5.6371
2024-02-07	1.6483	0.8457	0.6052	1.2879	5.7260
2024-02-08	1.6331	0.8383	0.6090	1.2899	5.7970

```
np.allclose(cumret_cumprod, cumret_cumsum)
```

True

```
cumret_cumsum.mul(100).plot()
```

```
# https://stackoverflow.com/questions/25973581/how-to-format-axis-number-format-to-thousan
from matplotlib import ticker
plt.gca().get_yaxis().set_major_formatter(ticker.FuncFormatter(lambda x, p: format(int(x),
plt.ylabel('Cumulative Return (%)')
plt.title(f'Cumulative Returns\n from {returns_2020s.index.min():%b %Y} to {returns_2020s.
plt.show()
```



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

We can also calculate cumulative returns as the ratio of adjusted closes. That is $R_{0,T} = \frac{AC_T}{AC_0} - 1$. The trick here is that $FV_t = PV(1+r)^t$, so $(1+r)^t = \frac{FV_t}{PV}$.

```
returns_2020s.iloc[0]
```

```

AAPL    0.0228
AMZN    0.0272
F        0.0129
META    0.0221
TSLA    0.0285
Name: 2020-01-02 00:00:00, dtype: float64
```

```
cumret_prices = prices['Adj Close'].loc['2020':].iloc[:-1] / prices['Adj Close'].loc['2019':]

np.allclose(cumret_prices, cumret_cumprod)
```

```
True
```

24.3.10 Calculate the Sharpe Ratio for TSLA

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

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

```
def calc_sharpe(ri, rf=ff['RF'], ppy=252):
    ri_rf = ri.sub(rf).dropna()
    return np.sqrt(ppy) * ri_rf.mean() / ri_rf.std()

calc_sharpe(ri=returns['TSLA'])
```

```
0.9261
```

```
calc_sharpe(ri=returns_2020s['TSLA'])
```

```
1.1212
```

Now we know about the .pipe() method, which lets us “convert” functions into methods! We can use .pipe() because ri is the first argument in calc_sharpe().

```
returns['TSLA'].pipe(calc_sharpe)
```

```
0.9261
```

```
returns_2020s['TSLA'].pipe(calc_sharpe)
```

```
1.1212
```

24.3.11 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 (i.e., $\beta_i = \frac{Cov(R_i - R_f, R_m - R_f)}{Var(R_m - R_f)}$) above for a univariate regression if we do not need goodness of fit statistics.

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

```
def calc_beta(ri, rf=ff['RF'], rm_rf=ff['Mkt-RF']):
    ri_rf = ri.sub(rf).dropna()
    return ri_rf.cov(rm_rf) / rm_rf.loc[ri_rf.index].var()

calc_beta(ri=returns['TSLA'])
```

1.4417

```
calc_beta(ri=returns_2020s['TSLA'])
```

1.5780

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

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

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

How good were your guesses?

```
for i in returns_2020s:
    sharpe_i = returns_2020s[i].pipe(calc_sharpe)
    print(f'Sharpe Ratio for {i}: {sharpe_i:.2f}')
```

```
Sharpe Ratio for AAPL: 0.86
Sharpe Ratio for AMZN: 0.48
Sharpe Ratio for F: 0.42
Sharpe Ratio for META: 0.49
Sharpe Ratio for TSLA: 1.12
```

We can also use pandas notation to vectorize this calculation. First calculate *excess* returns as $r_i - r_f$.

```
returns_2020s_excess = returns_2020s.sub(FF['RF'], axis=0).dropna()
```

Then use pandas notation to calculate means, standard deviations, and annualize.

```
(  
    returns_2020s_excess  
    .mean()  
    .div(returns_2020s_excess.std())  
    .mul(np.sqrt(252))  
)
```

```
AAPL    0.8576  
AMZN    0.4750  
F       0.4240  
META    0.4949  
TSLA    1.1212  
dtype: float64
```

Note: In a few weeks we will learn the `.apply()` method, which avoids the loop syntax.

```
returns_2020s.apply(calc_sharpe)
```

```
AAPL    0.8576  
AMZN    0.4750  
F       0.4240  
META    0.4949  
TSLA    1.1212  
dtype: float64
```

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

How good were your guesses?

```
for i in returns_2020s:  
    beta_i = returns_2020s[i].pipe(calc_beta)  
    print(f'Beta for {i}:\t{beta_i:.2f}')
```

```

Beta for AAPL: 1.15
Beta for AMZN: 1.04
Beta for F: 1.22
Beta for META: 1.27
Beta for TSLA: 1.58

```

Or we can follow out approach above to vectorize this calculation. First, we need to add a market excess return column to `returns_2020s_excess`.

```

returns_2020s_excess['Mkt-RF'] = ff['Mkt-RF']
returns_2020s_excess.head()

```

Date	AAPL	AMZN	F	META	TSLA	Mkt-RF
2020-01-02	0.0228	0.0271	0.0128	0.0220	0.0285	0.0086
2020-01-03	-0.0098	-0.0122	-0.0224	-0.0054	0.0296	-0.0067
2020-01-06	0.0079	0.0148	-0.0055	0.0188	0.0192	0.0036
2020-01-07	-0.0048	0.0020	0.0098	0.0021	0.0387	-0.0019
2020-01-08	0.0160	-0.0079	-0.0001	0.0101	0.0491	0.0047

```

vcv = returns_2020s_excess.cov()
vcv

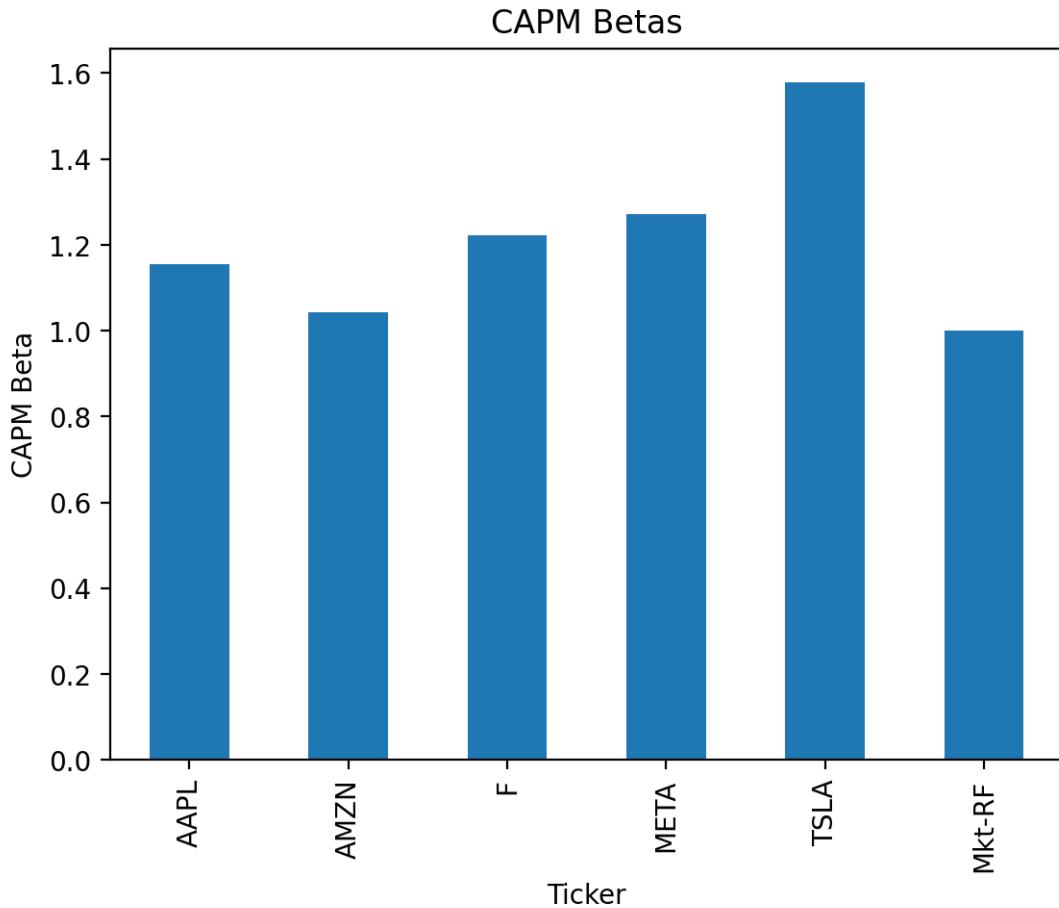
```

	AAPL	AMZN	F	META	TSLA	Mkt-RF
AAPL	0.0004	0.0003	0.0002	0.0004	0.0005	0.0003
AMZN	0.0003	0.0006	0.0002	0.0004	0.0005	0.0002
F	0.0002	0.0002	0.0009	0.0003	0.0005	0.0003
META	0.0004	0.0004	0.0003	0.0009	0.0005	0.0003
TSLA	0.0005	0.0005	0.0005	0.0005	0.0018	0.0003
Mkt-RF	0.0003	0.0002	0.0003	0.0003	0.0003	0.0002

```

vcv['Mkt-RF'].div(vcv.loc['Mkt-RF', 'Mkt-RF']).plot(kind='bar')
plt.xlabel('Ticker')
plt.ylabel('CAPM Beta')
plt.title('CAPM Betas')
plt.show()

```



Note: In a few weeks we will learn the `.apply()` method, which avoids the loop syntax.

```
returns_2020s.apply(calc_beta)
```

```
AAPL    1.1541
AMZN    1.0429
F        1.2231
META    1.2710
TSLA    1.5780
dtype: float64
```

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

What do you notice?

```
returns_2020s.mean(axis=1).pipe(calc_sharpe)
```

0.9573

Because diversification reduces portfolio standard deviation less than the sum of its parts, the Sharpe Ratio of the equally weighted portfolio is less than the equally weighted mean of the single-stock Sharpe Ratios.

```
returns_2020s.apply(calc_sharpe).mean()
```

0.6745

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

What do you notice?

Beta measures *nondiversifiable* risk, so $\beta_P = \sum w_i \beta_i$!

```
returns_2020s.mean(axis=1).pipe(calc_beta)
```

1.2538

```
returns_2020s.apply(calc_beta).mean()
```

1.2538

24.3.18 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!

```

betas = pd.DataFrame(
    index=range(1972, 2024),
    columns=returns.columns
)

betas.columns.name = 'Ticker'
betas.index.name = 'Year'

betas.tail()

```

Ticker	AAPL	AMZN	F	META	TSLA
Year					
2019	NaN	NaN	NaN	NaN	NaN
2020	NaN	NaN	NaN	NaN	NaN
2021	NaN	NaN	NaN	NaN	NaN
2022	NaN	NaN	NaN	NaN	NaN
2023	NaN	NaN	NaN	NaN	NaN

```

for i in betas.index:
    for c in betas.columns:
        betas.at[i, c] = returns.loc[str(i), c].pipe(calc_beta)

betas.tail()

```

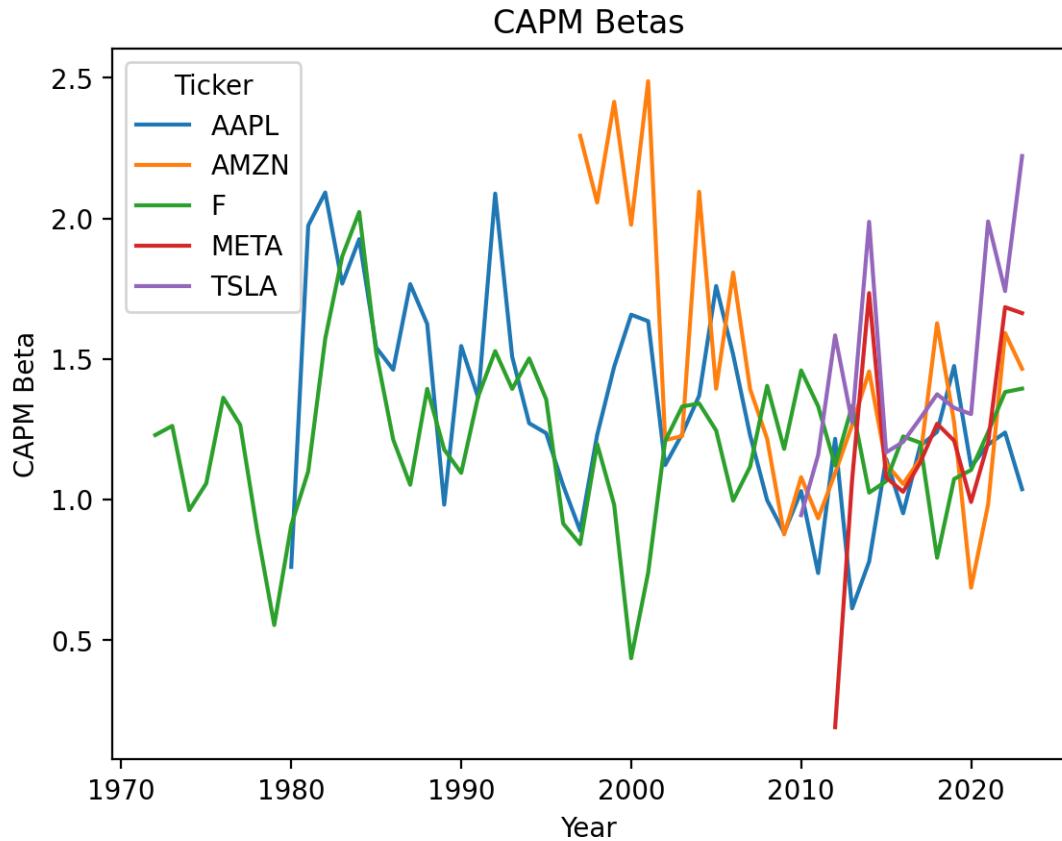
Ticker	AAPL	AMZN	F	META	TSLA
Year					
2019	1.4751	1.2752	1.0733	1.2094	1.3262
2020	1.1174	0.6866	1.1052	0.9913	1.3041
2021	1.1957	0.9822	1.2396	1.2014	1.9891
2022	1.2386	1.5922	1.3824	1.6843	1.7414
2023	1.0369	1.4649	1.3947	1.6630	2.2218

24.3.19 Plot the time series of market betas

```

betas.plot()
plt.ylabel('CAPM Beta')
plt.title('CAPM Betas')
plt.show()

```



25 Herron Topic 1 - Practice for Section 05

25.1 Announcements

1. DataCamp
 1. *Data Manipulation with pandas* due by Friday, 2/9, at 11:59 PM
 2. *Joining Data with pandas* due by Friday, 2/16, at 11:59 PM
 3. *Earn 10,000 XP* due by Friday, 3/15, at 11:59 PM
2. I posted Project 1 to Canvas
 1. Slides and notebook due by Friday, 2/23, at 11:59 PM
 2. Keep joining teams and let me know if you need help

25.2 10-Minute Recap

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import pandas_datareader as pdr
import yfinance as yf

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

First, we will use two packages to download data from the Web:

1. `yfinance` for Yahoo! Finance
2. `pandas-datareader` for Ken French (and FRED and many others)

Second, there are “simple returns” and “log returns”

1. Simple returns are the returns that investors receive that we learned in FINA 6331 and FINA 6333: $r_t = \frac{p_t + d_t - p_{t-1}}{p_{t-1}}$

2. Log returns are the log of one plus simple returns. Why do we use them?

1. **Log returns are additive**, while simple returns are multiplicative. This additive property makes math really easy with log returns: $\log(\prod_{t=0}^T (1+r_t)) = \sum_{t=0}^T \log(1+r_t)$, so $r_{0,T} = \prod_{t=0}^T (1+r_t) - 1 = e^{\sum_{t=0}^T \log(1+r_t)} - 1$
2. **Log returns are almost normally distributed**

We will almost always use simple returns. The exception is time-consuming calculations, which we will often do in log returns to save us time.

Third, we can calculate portfolio returns a few ways!

1. `returns.mean(axis=1)` is *equally-weighted* portfolio returns, rebalanced at the same frequency as the returns (i.e., rebalanced every return period)
2. `returns.dot(weights)` lets us use any weights in the `weights` array, rebalanced at the same frequency as the returns

25.3 Practice

25.3.1 Download all available daily price data for tickers TSLA, F, AAPL, AMZN, and META to data frame prices

```
tickers = 'TSLA F AAPL AMZN META'  
prices = yf.download(tickers=tickers)
```

```
[*****100%*****] 5 of 5 completed
```

```
prices.head()
```

Date	Adj Close				Close				...			
	AAPL	AMZN	F	META	TSLA	AAPL	AMZN	F	META	TSLA	...	AA
1972-06-01	NaN	NaN	0.2419	NaN	NaN	NaN	NaN	2.1532	NaN	NaN	...	Na
1972-06-02	NaN	NaN	0.2414	NaN	NaN	NaN	NaN	2.1492	NaN	NaN	...	Na
1972-06-05	NaN	NaN	0.2414	NaN	NaN	NaN	NaN	2.1492	NaN	NaN	...	Na
1972-06-06	NaN	NaN	0.2387	NaN	NaN	NaN	NaN	2.1248	NaN	NaN	...	Na
1972-06-07	NaN	NaN	0.2373	NaN	NaN	NaN	NaN	2.1127	NaN	NaN	...	Na

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

```
returns = (
    prices['Adj Close'] # slice adj close
    .iloc[:-1] # drop the last price because it might be intraday (i.e., not a close)
    .pct_change() # calculate simple returns
)

returns
```

Date	AAPL	AMZN	F	META	TSLA
1972-06-01	NaN	NaN	NaN	NaN	NaN
1972-06-02	NaN	NaN	-0.0019	NaN	NaN
1972-06-05	NaN	NaN	0.0000	NaN	NaN
1972-06-06	NaN	NaN	-0.0113	NaN	NaN
1972-06-07	NaN	NaN	-0.0057	NaN	NaN
...
2024-02-02	-0.0054	0.0787	0.0033	0.2032	-0.0050
2024-02-05	0.0098	-0.0087	-0.0453	-0.0328	-0.0365
2024-02-06	0.0086	-0.0068	0.0414	-0.0102	0.0223
2024-02-07	0.0006	0.0082	0.0605	0.0327	0.0134
2024-02-08	-0.0058	-0.0040	0.0023	0.0009	0.0106

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

```
returns_2020s = returns.loc['2020':] # always use an unambiguous date format, like YYYY-MM-
returns_2020s
```

Date	AAPL	AMZN	F	META	TSLA
2020-01-02	0.0228	0.0272	0.0129	0.0221	0.0285
2020-01-03	-0.0097	-0.0121	-0.0223	-0.0053	0.0296
2020-01-06	0.0080	0.0149	-0.0054	0.0188	0.0193
2020-01-07	-0.0047	0.0021	0.0098	0.0022	0.0388
2020-01-08	0.0161	-0.0078	0.0000	0.0101	0.0492
...

	AAPL	AMZN	F	META	TSLA
Date					
2024-02-02	-0.0054	0.0787	0.0033	0.2032	-0.0050
2024-02-05	0.0098	-0.0087	-0.0453	-0.0328	-0.0365
2024-02-06	0.0086	-0.0068	0.0414	-0.0102	0.0223
2024-02-07	0.0006	0.0082	0.0605	0.0327	0.0134
2024-02-08	-0.0058	-0.0040	0.0023	0.0009	0.0106

25.3.4 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']
```

```
ff_all = pdr.DataReader(
    name='F-F_Research_Data_Factors_daily',
    data_source='famafrench',
    start='1900' # most data start in 1926-07-01, but 1900 is easier to remember and type
)
```

```
C:\Users\r.herron\AppData\Local\Temp\ipykernel_27720\4231759426.py:1: FutureWarning: The argument
```

```
ff_all = pdr.DataReader(
```

The DESCRIPTOR key in the dictionary tells us about the data frames that pandas-datareader returns.

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

F-F Research Data Factors daily

This file was created by CMPT_ME_BEME_RETURNS_DAILY using the 202312 CRSP database. The Tbill rate is included.

```
0 : (25649 rows x 4 cols)
```

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

```
ff = ff_all[0].div(100)
```

```
ff
```

Date	Mkt-RF	SMB	HML	RF
1926-07-01	0.0010	-0.0025	-0.0027	0.0001
1926-07-02	0.0045	-0.0033	-0.0006	0.0001
1926-07-06	0.0017	0.0030	-0.0039	0.0001
1926-07-07	0.0009	-0.0058	0.0002	0.0001
1926-07-08	0.0021	-0.0038	0.0019	0.0001
...
2023-12-22	0.0021	0.0064	0.0009	0.0002
2023-12-26	0.0048	0.0069	0.0046	0.0002
2023-12-27	0.0016	0.0014	0.0012	0.0002
2023-12-28	-0.0001	-0.0036	0.0003	0.0002
2023-12-29	-0.0043	-0.0112	-0.0037	0.0002

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

We use the .prod() method to calculate *total* returns, because $r_{total} = r_{0,T} = \left[\prod_{t=0}^T (1 + r_t) \right] - 1$.

```
(  
    returns_2020s # returns during the 2020s  
    .add(1) # add 1 before we compound  
    .prod() # compound all returns  
    .sub(1) # subtract 1 to recover total returns
```

```
)
```

```
AAPL    1.6331
AMZN    0.8383
F       0.6090
META    1.2899
TSLA    5.7970
dtype: float64
```

We use the `.cumprod()` to calculate *cumulative* returns, which are the total returns for every date between 0 and T (i.e., $r_{0,t} \forall t \in 0, 1, \dots T$)

```
cumret_cumprod = (
    returns_2020s # returns during the 2020s
    .add(1) # add 1 before we compound
    .cumprod() # compound returns up to time t
    .sub(1) # subtract 1 to recover cumulative return at time t
)
```

```
cumret_cumprod
```

Date	AAPL	AMZN	F	META	TSLA
2020-01-02	0.0228	0.0272	0.0129	0.0221	0.0285
2020-01-03	0.0129	0.0147	-0.0097	0.0167	0.0590
2020-01-06	0.0209	0.0298	-0.0151	0.0358	0.0794
2020-01-07	0.0161	0.0319	-0.0054	0.0381	0.1213
2020-01-08	0.0325	0.0239	-0.0054	0.0486	0.1764
...
2024-02-02	1.5985	0.8596	0.5224	1.3142	5.7379
2024-02-05	1.6241	0.8433	0.4535	1.2383	5.4922
2024-02-06	1.6468	0.8308	0.5136	1.2154	5.6371
2024-02-07	1.6483	0.8457	0.6052	1.2879	5.7260
2024-02-08	1.6331	0.8383	0.6090	1.2899	5.7970

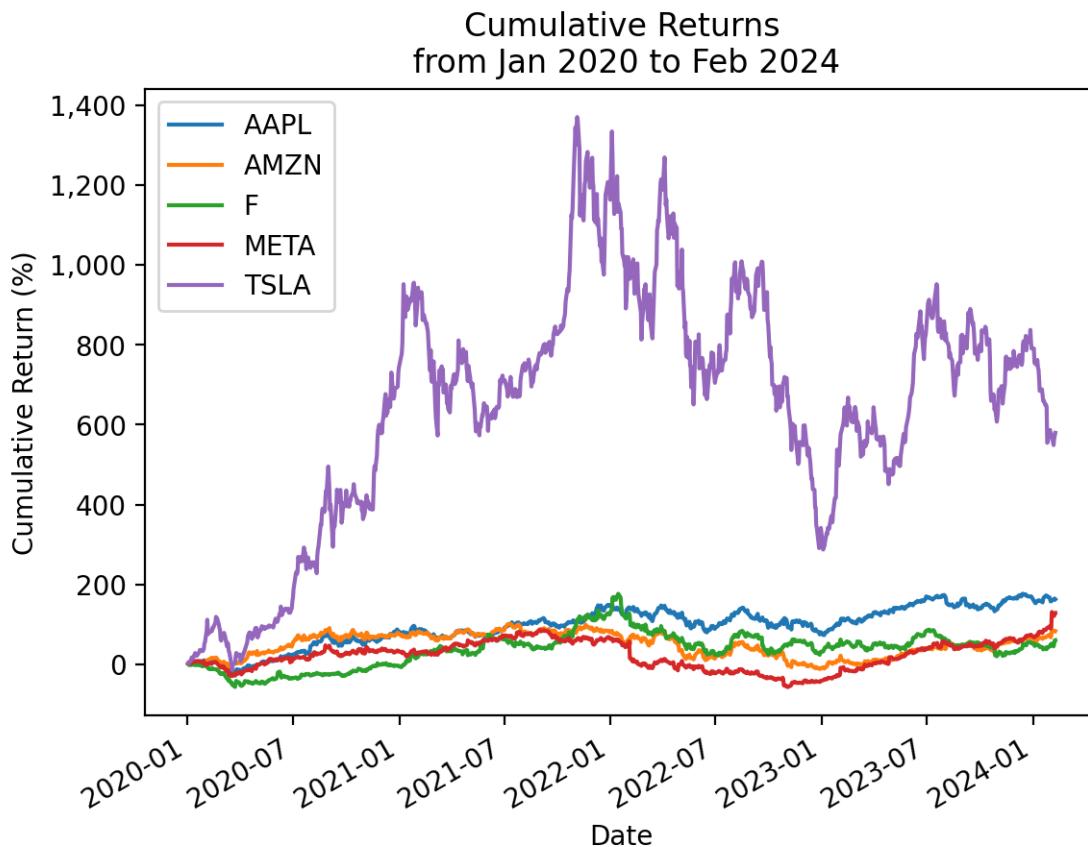
```
cumret_cumprod.mul(100).plot()
```

```
# https://stackoverflow.com/questions/25973581/how-to-format-axis-number-format-to-thousan
from matplotlib import ticker
```

```

plt.gca().get_yaxis().set_major_formatter(ticker.FuncFormatter(lambda x, p: format(int(x),
plt.ylabel('Cumulative Return (%)')
plt.title(f'Cumulative Returns\n from {returns_2020s.index.min():%b %Y} to {returns_2020s.index.max():%b %Y}')
plt.show()

```



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

```

cumret_cumsum = (
    returns_2020s # returns during the 2020s
    .add(1) # add 1 before we compound
    .pipe(np.log)
    .cumsum() # log returns are additive!

```

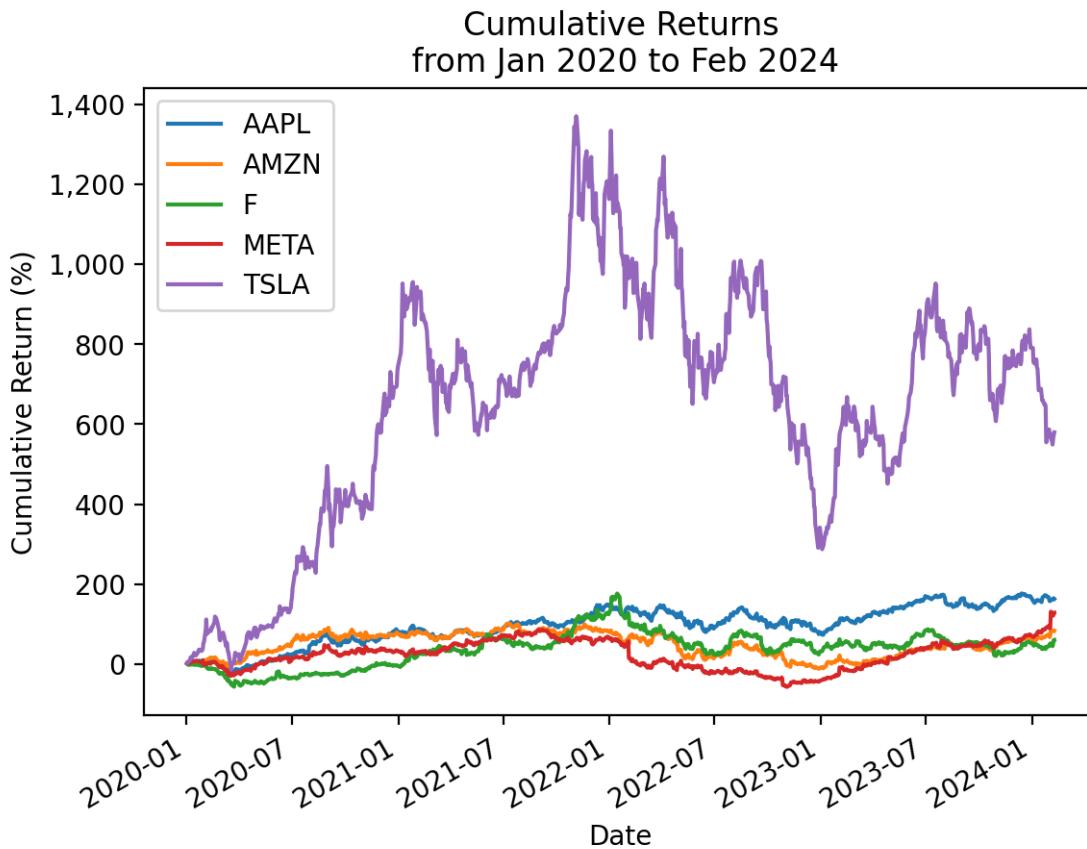
```
.pipe(np.exp)
    .sub(1) # subtract 1 to recover cumulative return at time t
)

np.allclose(cumret_cumprod, cumret_cumsum)

True

cumret_cumsum.mul(100).plot()

# https://stackoverflow.com/questions/25973581/how-to-format-axis-number-format-to-thousan
from matplotlib import ticker
plt.gca().get_yaxis().set_major_formatter(ticker.FuncFormatter(lambda x, p: format(int(x),
plt.ylabel('Cumulative Return (%)')
plt.title(f'Cumulative Returns\n from {returns_2020s.index.min():%b %Y} to {returns_2020s.
plt.show()
```



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

We can also calculate cumulative returns as the ratio of adjusted closes. That is $R_{0,T} = \frac{AC_T}{AC_0} - 1$. The trick here is that $FV_t = PV(1+r)^t$, so $(1+r)^t = \frac{FV_t}{PV}$.

```
returns_2020s.iloc[0]
```

```

AAPL    0.0228
AMZN    0.0272
F        0.0129
META    0.0221
TSLA    0.0285
Name: 2020-01-02 00:00:00, dtype: float64

```

```
prices['Adj Close'].loc['2020'].iloc[0]
```

```
AAPL    73.1526
AMZN    94.9005
F        8.0770
META    209.7800
TSLA    28.6840
Name: 2020-01-02 00:00:00, dtype: float64
```

```
prices['Adj Close'].loc['2019'].iloc[-1]
```

```
AAPL    71.5208
AMZN    92.3920
F        7.9741
META    205.2500
TSLA    27.8887
Name: 2019-12-31 00:00:00, dtype: float64
```

```
prices['Adj Close'].loc['2020'].iloc[0] / prices['Adj Close'].loc['2019'].iloc[-1] - 1
```

```
AAPL    0.0228
AMZN    0.0272
F        0.0129
META    0.0221
TSLA    0.0285
dtype: float64
```

Note: We drop the last row in `prices['Adj Close']` with `.iloc[:-1]`. We drop this last row here because we did the same above when we calculated `returns` to exclude possible intraday returns.

```
cumret_prices = prices['Adj Close'].loc['2020':].iloc[:-1] / prices['Adj Close'].loc['2019']
```

```
np.allclose(cumret_prices, cumret_cumprod)
```

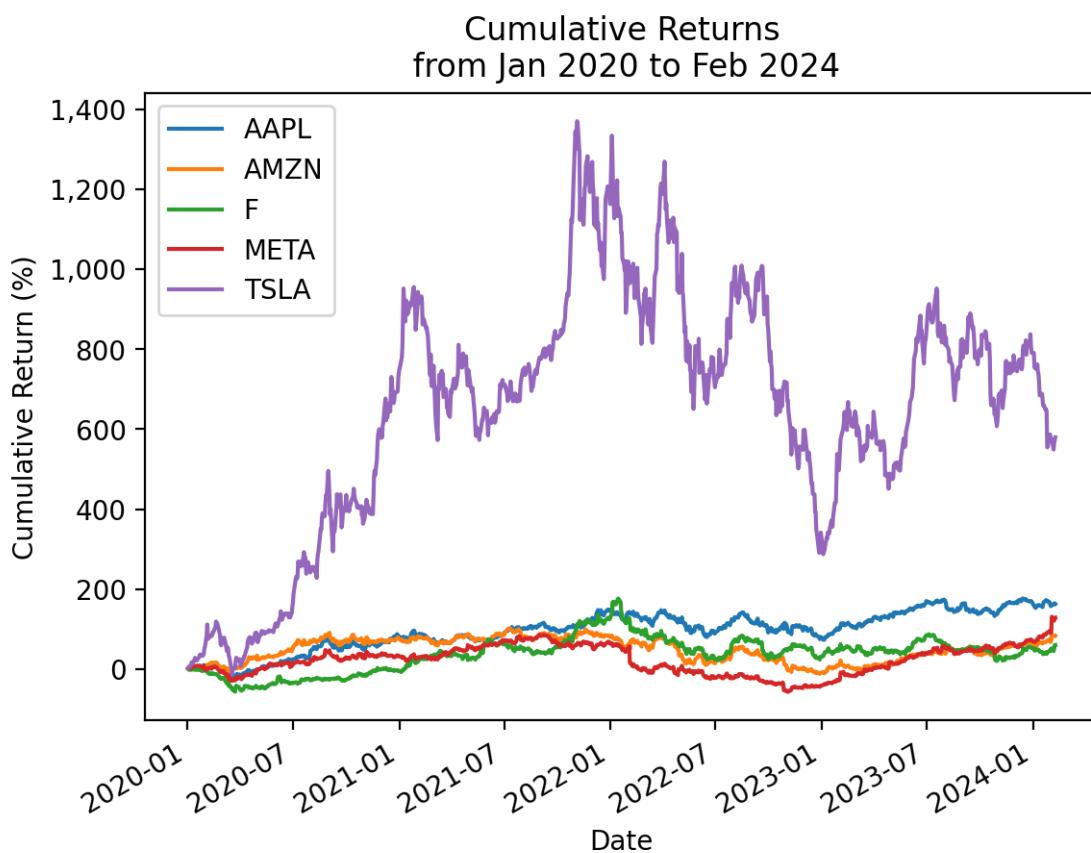
```
True
```

```

cumret_prices.mul(100).plot()

# https://stackoverflow.com/questions/25973581/how-to-format-axis-number-format-to-thousan
from matplotlib import ticker
plt.gca().get_yaxis().set_major_formatter(ticker.FuncFormatter(lambda x, p: format(int(x),
plt.ylabel('Cumulative Return (%)')
plt.title(f'Cumulative Returns\n from {returns_2020s.index.min():%b %Y} to {returns_2020s.i
plt.show()

```



25.3.9 Calculate the Sharpe Ratio for TSLA

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

I suggest you write a function named calc_sharpe() to use for the rest of this

notebook.

```
def calc_sharpe(ri, rf=ff['RF'], ppy=252):
    ri_rf = ri.sub(rf).dropna()
    return np.sqrt(ppy) * ri_rf.mean() / ri_rf.std()
```

```
calc_sharpe(ri=returns['TSLA'])
```

0.9261

```
calc_sharpe(ri=returns_2020s['TSLA'])
```

1.1212

We can use the `.pipe()` method here, too, since `ri` is the first argument to `calc_sharpe()`!

```
returns['TSLA'].pipe(calc_sharpe)
```

0.9261

```
returns_2020s['TSLA'].pipe(calc_sharpe)
```

1.1212

25.3.10 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 (i.e., $\beta_i = \frac{Cov(R_i - R_f, R_m - R_f)}{Var(R_m - R_f)}$) above for a univariate regression if we do not need goodness of fit statistics.

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

```
def calc_beta(ri, rf=ff['RF'], rm_rf=ff['Mkt-RF']):
    ri_rf = ri.sub(rf).dropna()
    return ri_rf.cov(rm_rf) / rm_rf.loc[ri_rf.index].var()
```

```
calc_beta(ri=returns['TSLA'])

1.4417

calc_beta(ri=returns_2020s['TSLA'])

1.5780
```

We can use the `.pipe()` method here, too, since `ri` is the first argument to `calc_beta()`!

```
returns['TSLA'].pipe(calc_beta)

1.4417

returns_2020s['TSLA'].pipe(calc_beta)

1.5780
```

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

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

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

How good were your guesses?

```
for i in returns_2020s:
    sharpe_i = returns_2020s[i].pipe(calc_sharpe)
    print(f'Sharpe Ratio for {i}:\t{sharpe_i:.2f}')

Sharpe Ratio for AAPL: 0.86
Sharpe Ratio for AMZN: 0.48
Sharpe Ratio for F: 0.42
Sharpe Ratio for META: 0.49
Sharpe Ratio for TSLA: 1.12
```

We can also use pandas notation to vectorize this calculation. First calculate *excess* returns as $r_i - r_f$.

```
returns_2020s_excess = returns_2020s.sub(ff['RF'], axis=0).dropna()
```

Then use pandas notation to calculate means, standard deviations, and annualize.

```
(  
    returns_2020s_excess  
    .mean()  
    .div(returns_2020s_excess.std())  
    .mul(np.sqrt(252))  
)
```

```
AAPL    0.8576  
AMZN    0.4750  
F       0.4240  
META    0.4949  
TSLA    1.1212  
dtype: float64
```

Note: In a few weeks we will learn the `.apply()` method, which avoids the loop syntax.

```
returns_2020s.apply(calc_sharpe)
```

```
AAPL    0.8576  
AMZN    0.4750  
F       0.4240  
META    0.4949  
TSLA    1.1212  
dtype: float64
```

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

How good were your guesses?

```
for i in returns_2020s:  
    beta_i = returns_2020s[i].pipe(calc_beta)  
    print(f'Beta for {i}:\t {beta_i:.2f}')
```

```
Beta for AAPL:   1.15  
Beta for AMZN:   1.04
```

```
Beta for F: 1.22
Beta for META: 1.27
Beta for TSLA: 1.58
```

Or we can follow out approach above to vectorize this calculation. First, we need to add a market excess return column to `returns_2020s_excess`.

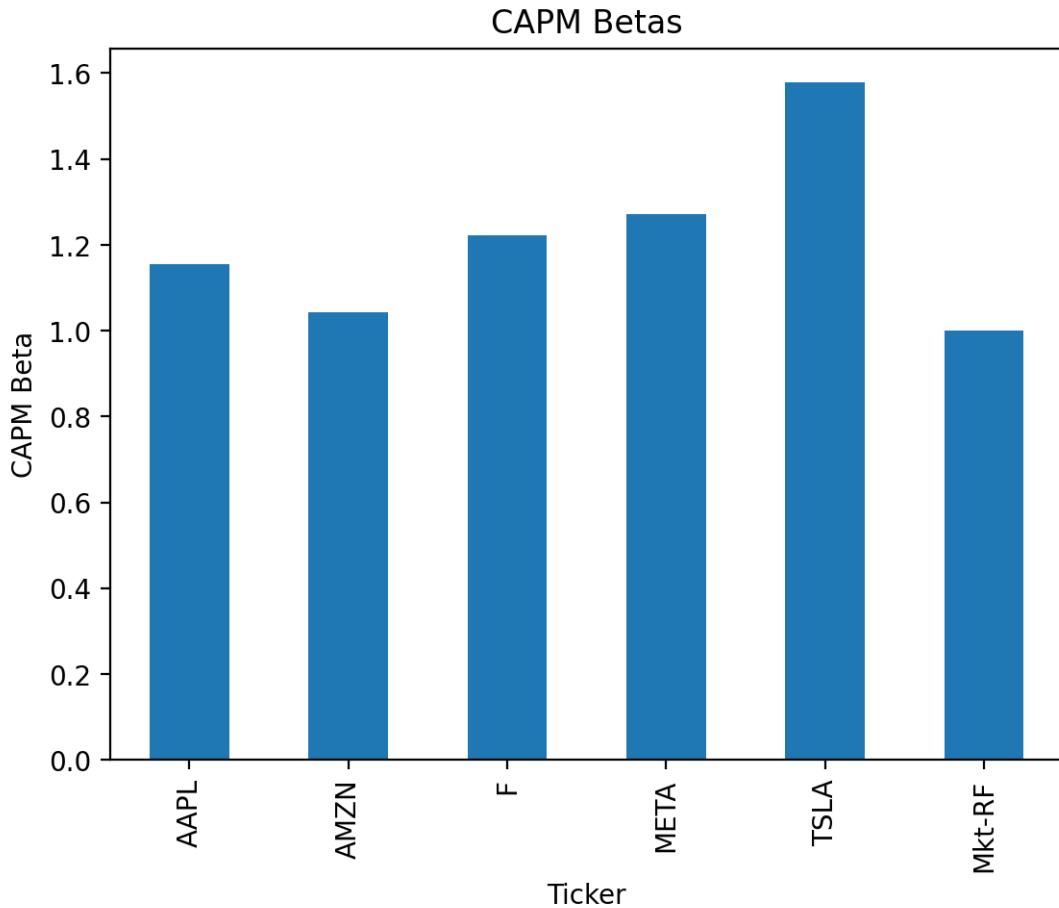
```
returns_2020s_excess['Mkt-RF'] = ff['Mkt-RF']
returns_2020s_excess.head()
```

Date	AAPL	AMZN	F	META	TSLA	Mkt-RF
2020-01-02	0.0228	0.0271	0.0128	0.0220	0.0285	0.0086
2020-01-03	-0.0098	-0.0122	-0.0224	-0.0054	0.0296	-0.0067
2020-01-06	0.0079	0.0148	-0.0055	0.0188	0.0192	0.0036
2020-01-07	-0.0048	0.0020	0.0098	0.0021	0.0387	-0.0019
2020-01-08	0.0160	-0.0079	-0.0001	0.0101	0.0491	0.0047

```
vcv = returns_2020s_excess.cov()
vcv
```

	AAPL	AMZN	F	META	TSLA	Mkt-RF
AAPL	0.0004	0.0003	0.0002	0.0004	0.0005	0.0003
AMZN	0.0003	0.0006	0.0002	0.0004	0.0005	0.0002
F	0.0002	0.0002	0.0009	0.0003	0.0005	0.0003
META	0.0004	0.0004	0.0003	0.0009	0.0005	0.0003
TSLA	0.0005	0.0005	0.0005	0.0005	0.0018	0.0003
Mkt-RF	0.0003	0.0002	0.0003	0.0003	0.0003	0.0002

```
vcv['Mkt-RF'].div(vcv.loc['Mkt-RF', 'Mkt-RF']).plot(kind='bar')
plt.xlabel('Ticker')
plt.ylabel('CAPM Beta')
plt.title('CAPM Betas')
plt.show()
```



Note: In a few weeks we will learn the `.apply()` method, which avoids the loop syntax.

```
returns_2020s.apply(calc_beta)
```

```
AAPL    1.1541
AMZN    1.0429
F        1.2231
META    1.2710
TSLA    1.5780
dtype: float64
```

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

What do you notice?

```
returns_2020s.mean(axis=1).pipe(calc_sharpe)
```

0.9573

Because diversification reduces portfolio standard deviation less than the sum of its parts, the Sharpe Ratio of the equally weighted portfolio is less than the equally weighted mean of the single-stock Sharpe Ratios.

```
returns_2020s.apply(calc_sharpe).mean()
```

0.6745

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

What do you notice?

Beta measures *nondiversifiable* risk, so $\beta_P = \sum w_i \beta_i$!

```
returns_2020s.mean(axis=1).pipe(calc_beta)
```

1.2538

```
returns_2020s.apply(calc_beta).mean()
```

1.2538

25.3.17 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!

```

betas = pd.DataFrame(
    index=range(1972, 2024),
    columns=returns.columns
)

betas.columns.name = 'Ticker'
betas.index.name = 'Year'

betas.tail()

```

Ticker	AAPL	AMZN	F	META	TSLA
Year					
2019	NaN	NaN	NaN	NaN	NaN
2020	NaN	NaN	NaN	NaN	NaN
2021	NaN	NaN	NaN	NaN	NaN
2022	NaN	NaN	NaN	NaN	NaN
2023	NaN	NaN	NaN	NaN	NaN

```

for i in betas.index:
    for c in betas.columns:
        betas.at[i, c] = returns.loc[str(i), c].pipe(calc_beta)

betas.tail()

```

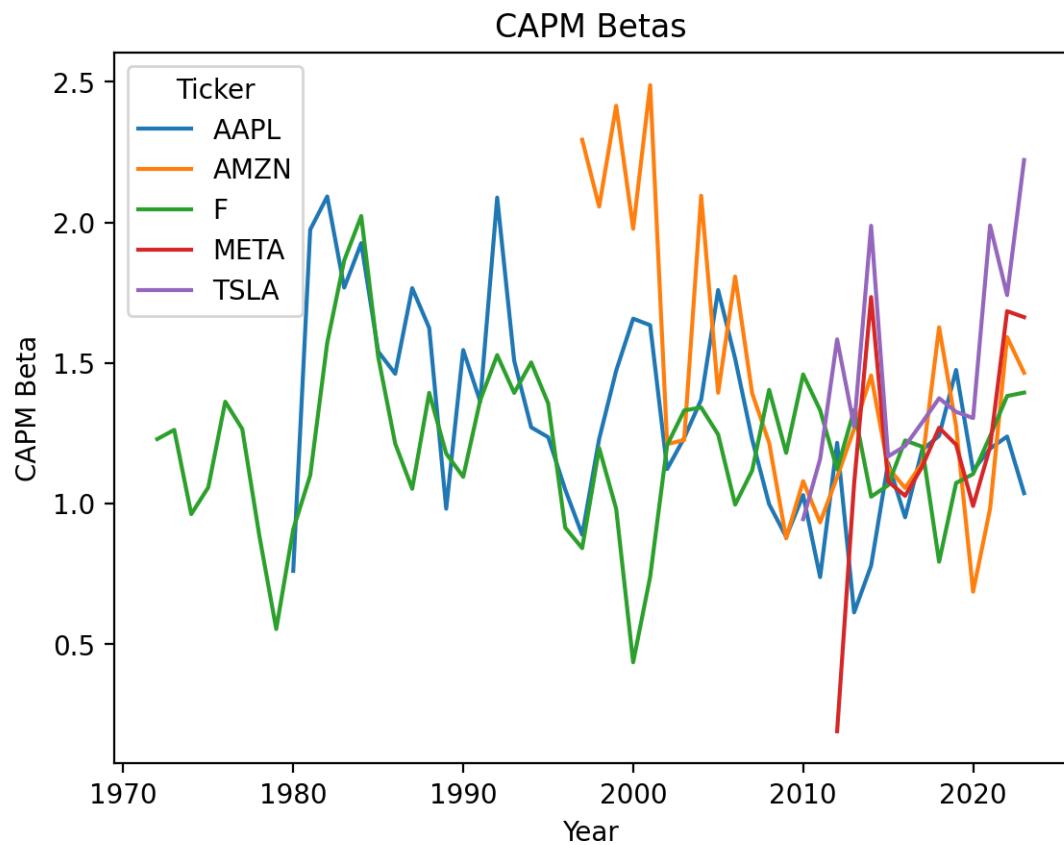
Ticker	AAPL	AMZN	F	META	TSLA
Year					
2019	1.4751	1.2752	1.0733	1.2094	1.3262
2020	1.1174	0.6866	1.1052	0.9913	1.3041
2021	1.1957	0.9822	1.2396	1.2014	1.9891
2022	1.2386	1.5922	1.3824	1.6843	1.7414
2023	1.0369	1.4649	1.3947	1.6630	2.2218

25.3.18 Plot the time series of market betas

```

betas.plot()
plt.ylabel('CAPM Beta')
plt.title('CAPM Betas')
plt.show()

```



Part VI

Week 6

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

26.1 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
import pandas_datareader as pdr
import yfinance as yf

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

26.2 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	b	c	d
1	0.4967	1.5230	-0.2341	0.7674
2	-0.1383	-0.2342	1.5792	-0.4695
3	0.6477			

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 hierarchical 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']
```

		color	Green	Red
key1	key2			
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']]
```

		state	Ohio
key1	key2	color	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')]
```

		state	Ohio	Colorado
key1	key2	color	Green	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	

26.2.1 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
```

		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.swaplevel()
```

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

We can chain these methods, too.

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

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

26.2.2 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
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	
0	0	7
one	1	6
	2	5
	0	4
two	1	3
	2	2
	3	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

26.3 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.

26.3.1 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

	key	data1
4	a	4
5	a	5
6	b	6

df2

	key	data2
0	a	0
1	b	1
2	d	2

`pd.merge(df1, df2)`

	key	data1	data2
0	b	0	1
1	b	1	1
2	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

	key	data1	data2
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
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  
# MergeError: No common columns to perform merge on. Merge options: left_on=None, right_on
```

```
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 by 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

	key	data1
2	a	2
3	c	3
4	a	4
5	b	5

df2

	key	data2
0	a	0
1	b	1
2	a	2
3	b	3
4	d	4

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

	key	data1	data2
0	b	0	1
1	b	0	3
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.

26.3.2 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

	key	value	group_val
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
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

26.3.3 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
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', 'four'])
```

```
pd.concat([df1, df2], axis=1, keys=['level1', 'level2'])
```

	level1		level2	
	one	two	three	four
a	0	1	5.0000	6.0000
b	2	3	NaN	NaN
c	4	5	7.0000	8.0000

```
pd.concat([df1, df2], axis=1, keys=['level1', 'level2'], names=['upper', 'lower'])
```

	upper	level1	level2	
	lower	one	two	three
a	0	1	5.0000	6.0000
b	2	3	NaN	NaN
c	4	5	7.0000	8.0000

26.4 Reshaping and Pivoting

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

26.4.1 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	one	two	three
state				
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	one	two	three
state				
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
number			Colorado
one	0	3	5
two	1	4	6
three	2	5	7
			10

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

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

number	state	Ohio	Colorado
	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.

27 McKinney Chapter 8 - Practice for Section 02

27.1 Announcements

1. *Joining Data with pandas*, our fourth and final DataCamp course, is due by Friday, 2/16, at 11:59 PM
2. We will use class next week, from 2/19 through 2/23, for group work on Project 1, so there is no lecture video or pre-class quiz
3. Project 1 is due by Tuesday, 2/27, at 8:25 AM
4. Please complete the [anonymous ungraded survey on Canvas](#) to help me help you learn better:

27.2 10-Minute Recap

Chapter 8 of McKinney covers 3 topics.

1. *Hierarchical Indexing*: Hierarchical indexing helps us organize data at multiple levels, rather than just a flat, two-dimensional structure. It helps us work with high-dimensional data in a low-dimensional form. For example, we can index rows by multiple levels like “ticker” and “date”, or columns by “variable” and “ticker”.
2. *Combining Data*: We can combine datasets on one or more keys.
 1. We will use the `pd.merge()` function for database-style joins, which can be `inner`, `outer`, `left`, or `right` joins.
 2. We will use the `.join()` method to combine data frames with similar indexes.
 3. We will use the `pd.concat()` to combine similarly-shaped series and data frames.
3. *Reshaping Data*: We can reshape data to change its structure, such as pivoting from wide to long format or vice versa. We will most often use the `.stack()` and `.unstack()` methods, which pivot columns to rows and rows to columns, respectively. Later in the course we will learn about the `.pivot()` method for aggregating data and the `.melt()` method for more advanced reshaping.

27.3 Practice

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import pandas_datareader as pdr
import yfinance as yf

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

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

Use `.rename_axis(columns=stocks.columns.names = ['Variable', 'Ticker'])` to assign the names Variable and Ticker to the column multi index. We could instead use `stocks.columns.names = ['Variable', 'Ticker']`, but we can chain the `.rename_axis()` method.

```
stocks = (
    yf.download(tickers='BAC C GS JPM MS PNC')
    .rename_axis(columns=['Variable', 'Ticker'])
)
```

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

```
stocks.tail()
```

Variable	Adj Close					Close				
Ticker	BAC	C	GS	JPM	MS	PNC	BAC	C	GS	JPM
Date										
2024-02-09	33.0700	53.9900	384.2600	175.0100	85.8900	147.7700	33.0700	53.9900	384.2600	175.0100
2024-02-12	33.6200	53.9200	392.6400	175.7900	86.8700	149.1400	33.6200	53.9200	392.6400	175.7900
2024-02-13	32.7500	52.7600	378.7500	174.2600	83.9700	145.2600	32.7500	52.7600	378.7500	174.2600
2024-02-14	33.1300	53.9800	378.0400	176.0300	84.0000	147.8700	33.1300	53.9800	378.0400	176.0300
2024-02-15	34.0700	55.2100	385.4200	179.8700	85.6700	149.6300	34.0700	55.2100	385.4200	179.8700

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

```
stocks_long = stocks.stack()
```

```
stocks_long.tail()
```

Date	Variable Ticker	Adj Close	Close	High	Low	Open	Volume
2024-02-15	C	55.2100	55.2100	55.4800	54.1350	54.2200	16288325.0000
	GS	385.4200	385.4200	387.2100	379.3500	379.4200	1780459.0000
	JPM	179.8700	179.8700	180.2100	176.1500	176.1500	8718729.0000
	MS	85.6700	85.6700	86.2300	84.4100	84.4500	7981854.0000
	PNC	149.6300	149.6300	150.2575	147.3700	148.7500	1991304.0000

27.3.3 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`.

```
_ = pd.MultiIndex.from_product([['Returns'], stocks['Adj Close'].columns]) # I use _ as a
stocks[_] = stocks['Adj Close'].iloc[:-1].pct_change()
```

```
stocks.tail()
```

Variable Ticker Date	Adj Close						Close					
	BAC	C	GS	JPM	MS	PNC	BAC	C	GS	JPM		
2024-02-09	33.0700	53.9900	384.2600	175.0100	85.8900	147.7700	33.0700	53.9900	384.2600	175.0100		
2024-02-12	33.6200	53.9200	392.6400	175.7900	86.8700	149.1400	33.6200	53.9200	392.6400	175.7900		
2024-02-13	32.7500	52.7600	378.7500	174.2600	83.9700	145.2600	32.7500	52.7600	378.7500	174.2600		
2024-02-14	33.1300	53.9800	378.0400	176.0300	84.0000	147.8700	33.1300	53.9800	378.0400	176.0300		
2024-02-15	34.0700	55.2100	385.4200	179.8700	85.6700	149.6300	34.0700	55.2100	385.4200	179.8700		

The easiest way to add returns to long data frame `stocks_long` is to `.stack()` the wide data frame `stocks`! We could sort `stocks_long` by ticker and date (to sort chronologically within each ticker), then use `.pct_change()`. However, this approach miscalculates the first return

for every ticker except for the first ticker. The easiest and safest solution is to `.stack()` the wide data frame `stocks`!

```
stocks_long = stocks.stack()
```

```
stocks_long.tail()
```

Date	Variable	Adj Close	Close	High	Low	Open	Volume	Returns
	Ticker							
2024-02-15	C	55.2100	55.2100	55.4800	54.1350	54.2200	16288325.0000	NaN
	GS	385.4200	385.4200	387.2100	379.3500	379.4200	1780459.0000	NaN
	JPM	179.8700	179.8700	180.2100	176.1500	176.1500	8718729.0000	NaN
	MS	85.6700	85.6700	86.2300	84.4100	84.4500	7981854.0000	NaN
	PNC	149.6300	149.6300	150.2575	147.3700	148.7500	1991304.0000	NaN

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

I rarely remember the exact combination of uppercase and lowercase letters, dashes, and underscores, so I use `pdr.famafrench.get_available_datasets()` to display the list of available names for data from Ken French's website.

```
pdr.famafrench.get_available_datasets()[:5]
```

```
['F-F_Research_Data_Factors',
 'F-F_Research_Data_Factors_weekly',
 'F-F_Research_Data_Factors_daily',
 'F-F_Research_Data_5_Factors_2x3',
 'F-F_Research_Data_5_Factors_2x3_daily']

ff = (
    pdr.DataReader(
        name='F-F_Research_Data_Factors_daily',
        data_source='famafrench',
        start='1900' # otherwise, pdr.DataReader defaults to the last five years of data
    )
[0] # pdr.DataReader returns a dictionary of data frames
.div(100) # French stores returns as percents, but our stock returns are decimals
```

```
C:\Users\r.herron\AppData\Local\Temp\ipykernel_13496\1909726130.py:2: FutureWarning: The argument  
pdr.DataReader(
```

```
ff.tail()
```

Date	Mkt-RF	SMB	HML	RF
2023-12-22	0.0021	0.0064	0.0009	0.0002
2023-12-26	0.0048	0.0069	0.0046	0.0002
2023-12-27	0.0016	0.0014	0.0012	0.0002
2023-12-28	-0.0001	-0.0036	0.0003	0.0002
2023-12-29	-0.0043	-0.0112	-0.0037	0.0002

27.3.5 Add the daily benchmark return factors to stocks and stocks_long.

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

For the wide data frame `stocks`, the simplest approach is to create a multi index with four columns, one for each of the four factors.

```
_ = pd.MultiIndex.from_product([['Factors'], ff.columns]) # I use _ as a temporary variable  
stocks[_] = ff
```

```
stocks.tail()
```

Variable	Adj Close						Close					
Ticker	BAC	C	GS	JPM	MS	PNC	BAC	C	GS	JPM		
Date												
2024-02-09	33.0700	53.9900	384.2600	175.0100	85.8900	147.7700	33.0700	53.9900	384.2600	175.0100	85.8900	147.7700
2024-02-12	33.6200	53.9200	392.6400	175.7900	86.8700	149.1400	33.6200	53.9200	392.6400	175.7900	86.8700	149.1400
2024-02-13	32.7500	52.7600	378.7500	174.2600	83.9700	145.2600	32.7500	52.7600	378.7500	174.2600	83.9700	145.2600
2024-02-14	33.1300	53.9800	378.0400	176.0300	84.0000	147.8700	33.1300	53.9800	378.0400	176.0300	84.0000	147.8700
2024-02-15	34.0700	55.2100	385.4200	179.8700	85.6700	149.6300	34.0700	55.2100	385.4200	179.8700	85.6700	149.6300

```
stocks_long = stocks_long.join(ff)
```

```
stocks_long.head()
```

Date	Ticker	Adj Close	Close	High	Low	Open	Volume	Returns	Mkt-RF	SMB
1973-02-21	BAC	1.5818	4.6250	4.6250	4.6250	4.6250	99200.0000	NaN	-0.0074	-0.003
1973-02-22	BAC	1.5872	4.6406	4.6406	4.6406	4.6406	47200.0000	0.0034	-0.0030	-0.003
1973-02-23	BAC	1.5818	4.6250	4.6250	4.6250	4.6250	133600.0000	-0.0034	-0.0108	-0.001
1973-02-26	BAC	1.5818	4.6250	4.6250	4.6250	4.6250	24000.0000	0.0000	-0.0088	-0.005
1973-02-27	BAC	1.5818	4.6250	4.6250	4.6250	4.6250	41600.0000	0.0000	-0.0115	-0.001

```
stocks_long.loc['2023-12'].iloc[-12:]
```

Date	Ticker	Adj Close	Close	High	Low	Open	Volume	Returns	Mkt-
2023-12-28	BAC	33.8800	33.8800	33.9700	33.7700	33.8200	21799600.0000	0.0012	-0.00
	C	51.0329	51.5200	51.8000	51.4000	51.4000	10218500.0000	0.0012	-0.00
	GS	386.4100	386.4100	387.7600	383.6300	384.5200	1024700.0000	0.0050	-0.00
	JPM	169.2563	170.3000	170.6600	169.0000	169.3500	6320100.0000	0.0053	-0.00
	MS	92.7316	93.6400	93.9500	93.2400	93.3100	4089500.0000	-0.0002	-0.00
	PNC	154.0486	155.6300	156.2100	154.9500	155.4400	1153300.0000	0.0041	-0.00
2023-12-29	BAC	33.6700	33.6700	33.9900	33.5500	33.9400	28037800.0000	-0.0062	-0.00
	C	50.9537	51.4400	51.6100	51.2200	51.5600	13147900.0000	-0.0016	-0.00
	GS	385.7700	385.7700	386.6400	383.5700	385.5700	881300.0000	-0.0017	-0.00
	JPM	169.0575	170.1000	170.6900	169.6300	170.0000	6431800.0000	-0.0012	-0.00
	MS	92.3454	93.2500	93.7700	93.0600	93.4900	4772100.0000	-0.0042	-0.00
	PNC	153.2765	154.8500	156.1100	154.5200	155.2700	1572600.0000	-0.0050	-0.00

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

We can even add a `shape` argument to return a wide or long data frame!

```
def download(tickers, shape='wide'):
    if shape.lower() not in ['wide', 'long']:
        raise ValueError('Invalid shape: must be "wide" or "long".')
    stocks = yf.download(tickers)
```

```

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

factors = factors_all[0].div(100)

# if stocks has a multi index on the columns because we asked for more than one ticker
# then we have more work to do
if type(stocks.columns) is pd.MultiIndex:

    # whether we want a wide or long data frame, it is easier to add returns to a wide
    _ = pd.MultiIndex.from_product([['Returns'], stocks['Adj Close'].columns])
    stocks[_] = stocks['Adj Close'].pct_change()

    # if we want a wide data frame, then we need a factors multi index
    if shape.lower() == 'wide':
        _ = pd.MultiIndex.from_product([['Factors'], factors.columns])
        stocks[_] = factors
        return stocks.rename_axis(columns=['Variable', 'Ticker'])

    # if we want a long data frame, then we need to stack stocks and join factors
    # because we tested for wide and long above, we know that here shape must be long
    else:
        return stocks.stack().join(factors).rename_axis(columns=['Variable'], index=[

# if stocks does not have a multi index on the columns
# then we only have join factors
else:
    return stocks.join(ff).rename_axis(columns=['Variable'])

download(tickers=['AAPL', 'TSLA'], shape='long')

[*****100%*****] 2 of 2 completed
C:\Users\r.herron\AppData\Local\Temp\ipykernel_13496\3667605341.py:8: FutureWarning: The arg
pdr.DataReader(

```

Date	Variable	Adj Close	Close	High	Low	Open	Volume	Returns	M
	Ticker								
1980-12-12	AAPL	0.0992	0.1283	0.1289	0.1283	0.1283	469033600.0000	NaN	0.
1980-12-15	AAPL	0.0940	0.1217	0.1222	0.1217	0.1222	175884800.0000	-0.0522	0.
1980-12-16	AAPL	0.0871	0.1127	0.1133	0.1127	0.1133	105728000.0000	-0.0734	0.
1980-12-17	AAPL	0.0893	0.1155	0.1161	0.1155	0.1155	86441600.0000	0.0248	0.
1980-12-18	AAPL	0.0919	0.1189	0.1194	0.1189	0.1189	73449600.0000	0.0290	0.
...
2024-02-13	TSLA	184.0200	184.0200	187.2600	182.1100	183.9900	86759500.0000	-0.0218	N
2024-02-14	AAPL	184.1500	184.1500	185.5300	182.4400	185.3200	54630500.0000	-0.0048	N
	TSLA	188.7100	188.7100	188.8900	183.3500	185.3000	81203000.0000	0.0255	N
2024-02-15	AAPL	183.8600	183.8600	184.4900	181.4000	183.5500	63936069.0000	-0.0016	N
	TSLA	200.4500	200.4500	200.8800	188.8595	189.1600	119932134.0000	0.0622	N

27.3.7 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`.

27.3.8 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.

27.3.9 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

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

27.3.11 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.

28 McKinney Chapter 8 - Practice for Section 03

28.1 Announcements

1. *Joining Data with pandas*, our fourth and final DataCamp course, is due by Friday, 2/16, at 11:59 PM
2. We will use class next week, from 2/19 through 2/23, for group work on Project 1, so there is no lecture video or pre-class quiz
3. Project 1 is due by Tuesday, 2/27, at 8:25 AM
4. Please complete the [anonymous ungraded survey on Canvas](#) to help me help you learn better:

28.2 10-Minute Recap

Chapter 8 of McKinney covers 3 topics.

1. *Hierarchical Indexing*: Hierarchical indexing helps us organize data at multiple levels, rather than just a flat, two-dimensional structure. It helps us work with high-dimensional data in a low-dimensional form. For example, we can index rows by multiple levels like “ticker” and “date”, or columns by “variable” and “ticker”.
2. *Combining Data*: We can combine datasets on one or more keys.
 1. We will use the `pd.merge()` function for database-style joins, which can be `inner`, `outer`, `left`, or `right` joins.
 2. We will use the `.join()` method to combine data frames with similar indexes.
 3. We will use the `pd.concat()` to combine similarly-shaped series and data frames.
3. *Reshaping Data*: We can reshape data to change its structure, such as pivoting from wide to long format or vice versa. We will most often use the `.stack()` and `.unstack()` methods, which pivot columns to rows and rows to columns, respectively. Later in the course we will learn about the `.pivot()` method for aggregating data and the `.melt()` method for more advanced reshaping.

28.3 Practice

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import pandas_datareader as pdr
import yfinance as yf

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

28.3.1 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.

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

28.3.3 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`.

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

28.3.5 Add the daily benchmark return factors to stocks and stocks_long.

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

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

28.3.7 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`.

28.3.8 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.

28.3.9 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

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

28.3.11 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.

28.3.12 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.

29 McKinney Chapter 8 - Practice for Section 04

29.1 Announcements

1. *Joining Data with pandas*, our fourth and final DataCamp course, is due by Friday, 2/16, at 11:59 PM
2. We will use class next week, from 2/19 through 2/23, for group work on Project 1, so there is no lecture video or pre-class quiz
3. Project 1 is due by Tuesday, 2/27, at 8:25 AM
4. Please complete the [anonymous ungraded survey on Canvas](#) to help me help you learn better:

29.2 10-Minute Recap

Chapter 8 of McKinney covers 3 topics.

1. *Hierarchical Indexing*: Hierarchical indexing helps us organize data at multiple levels, rather than just a flat, two-dimensional structure. It helps us work with high-dimensional data in a low-dimensional form. For example, we can index rows by multiple levels like “ticker” and “date”, or columns by “variable” and “ticker”.
2. *Combining Data*: We can combine datasets on one or more keys.
 1. We will use the `pd.merge()` function for database-style joins, which can be `inner`, `outer`, `left`, or `right` joins.
 2. We will use the `.join()` method to combine data frames with similar indexes.
 3. We will use the `pd.concat()` to combine similarly-shaped series and data frames.
3. *Reshaping Data*: We can reshape data to change its structure, such as pivoting from wide to long format or vice versa. We will most often use the `.stack()` and `.unstack()` methods, which pivot columns to rows and rows to columns, respectively. Later in the course we will learn about the `.pivot()` method for aggregating data and the `.melt()` method for more advanced reshaping.

29.3 Practice

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import pandas_datareader as pdr
import yfinance as yf

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

29.3.1 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.

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

29.3.3 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`.

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

29.3.5 Add the daily benchmark return factors to stocks and stocks_long.

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

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

29.3.7 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`.

29.3.8 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.

29.3.9 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

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

29.3.11 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.

29.3.12 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.

30 McKinney Chapter 8 - Practice for Section 05

30.1 Announcements

1. *Joining Data with pandas*, our fourth and final DataCamp course, is due by Friday, 2/16, at 11:59 PM
2. We will use class next week, from 2/19 through 2/23, for group work on Project 1, so there is no lecture video or pre-class quiz
3. Project 1 is due by Tuesday, 2/27, at 8:25 AM
4. Please complete the [anonymous ungraded survey on Canvas](#) to help me help you learn better:

30.2 10-Minute Recap

Chapter 8 of McKinney covers 3 topics.

1. *Hierarchical Indexing*: Hierarchical indexing helps us organize data at multiple levels, rather than just a flat, two-dimensional structure. It helps us work with high-dimensional data in a low-dimensional form. For example, we can index rows by multiple levels like “ticker” and “date”, or columns by “variable” and “ticker”.
2. *Combining Data*: We can combine datasets on one or more keys.
 1. We will use the `pd.merge()` function for database-style joins, which can be `inner`, `outer`, `left`, or `right` joins.
 2. We will use the `.join()` method to combine data frames with similar indexes.
 3. We will use the `pd.concat()` to combine similarly-shaped series and data frames.
3. *Reshaping Data*: We can reshape data to change its structure, such as pivoting from wide to long format or vice versa. We will most often use the `.stack()` and `.unstack()` methods, which pivot columns to rows and rows to columns, respectively. Later in the course we will learn about the `.pivot()` method for aggregating data and the `.melt()` method for more advanced reshaping.

30.3 Practice

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import pandas_datareader as pdr
import yfinance as yf

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

30.3.1 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.

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

30.3.3 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`.

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

30.3.5 Add the daily benchmark return factors to stocks and stocks_long.

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

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

30.3.7 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`.

30.3.8 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.

30.3.9 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

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

30.3.11 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.

30.3.12 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

31 Project 1

FINA 6333 – Spring 2023

To be determined

Part VIII

Week 8

32 McKinney Chapter 10 - Data Aggregation and Group Operations

32.1 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
```

32.2 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 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()
```

data1	
key1	
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	one	two
key1		
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()
```

		data1	data2
	key1		
	a	0.0414	0.6292
	b	1.0854	0.1490

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

		data1	data2
key1	key2		
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.

32.2.1 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

```

32.2.2 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
	two	-0.5623	-1.0128	0.3142	-0.9080
5	one	-0.2341	1.5792	0.7674	-0.4695
	two	1.4656	-0.2258	0.0675	-1.4247
					-1.7249
					-1.4123

```
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
	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
	Nerd	-0.5623	-1.0128	0.3142	-0.9080
6	Cool	1.4656	-0.2258	0.0675	-1.4247

32.2.3 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

32.3 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

32.3.1 Column-Wise and Multiple Function Application

The `.agg()` method 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'])
```

	mean	median	min	max
key1				
a	0.0414	-0.1383	-0.2342	0.4967
b	1.0854	1.0854	0.6477	1.5230

```
df.groupby('key1')[['data1', 'data2']].agg(['mean', 'median', 'min', 'max'])
```

key1	data1				data2			
	mean	median	min	max	mean	median	min	max
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
	mean	median	median
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'})
```

key1	data1		data2
	mean	std	median
a	0.0414	0.3972	0.5426
b	1.0854	0.6190	0.1490

32.4 Apply: General split-apply-combine

The `.agg()` method aggregates 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	key2	data1	data2
key1				
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	key2	data1	data2
key1				
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

32.5 Pivot Tables and Cross-Tabulation

Above we manually made pivot tables with the `groupby()`, `.agg()`, `.apply()` and `.unstack()` methods. pandas provides a literal interpretation 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	Adj Close	Close	High	Low	Open	Volume
Index						
^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']
    )
)
)

```

Index Date	min						max		
	^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	

33 McKinney Chapter 10 - Practice for Section 02

33.1 Announcements

33.2 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
```

33.2.1 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']
)
)

```

Index Date	min ^DJI	^FTSE	^GSPC	^HSI	^IXIC	^N225	max ^DJI	^FTS
2015-12-31	15666.4404	5874.1001	1867.6100	20556.5996	4506.4902	16795.9609	18312.3906	7104.0
2016-12-31	15660.1797	5537.0000	1829.0800	18319.5801	4266.8398	14952.0195	19974.6191	7142.7
2017-12-31	19732.4004	7099.2002	2257.8301	22134.4707	5429.0801	18335.6309	24837.5098	7687.7
2018-12-31	21792.1992	6584.7002	2351.1001	24585.5293	6192.9199	19155.7402	26828.3906	7877.5
2019-12-31	22686.2207	6692.7002	2447.8899	25064.3594	6463.5000	19561.9609	28645.2598	7686.0
2020-12-31	18591.9297	4993.8999	2237.3999	21696.1309	6860.6699	16552.8301	30606.4805	7674.0
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

33.2.2 Calculate 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.

33.2.3 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.

33.2.4 Calculate monthly means and volatilities for SPY and GOOG returns

33.2.5 Plot the monthly means and volatilities from the previous exercise

33.2.6 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.

33.2.7 Plot the time-series volatilities of these five portfolios

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

33.2.8 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.

33.2.9 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

34 McKinney Chapter 10 - Practice for Section 03

34.1 Announcements

34.2 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
```

34.2.1 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']
)
)

```

Index Date	min ^DJI	^FTSE	^GSPC	^HSI	^IXIC	^N225	max ^DJI	^FTS
2015-12-31	15666.4404	5874.1001	1867.6100	20556.5996	4506.4902	16795.9609	18312.3906	7104.0
2016-12-31	15660.1797	5537.0000	1829.0800	18319.5801	4266.8398	14952.0195	19974.6191	7142.7
2017-12-31	19732.4004	7099.2002	2257.8301	22134.4707	5429.0801	18335.6309	24837.5098	7687.7
2018-12-31	21792.1992	6584.7002	2351.1001	24585.5293	6192.9199	19155.7402	26828.3906	7877.5
2019-12-31	22686.2207	6692.7002	2447.8899	25064.3594	6463.5000	19561.9609	28645.2598	7686.0
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

34.2.2 Calculate 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.

34.2.3 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.

34.2.4 Calculate monthly means and volatilities for SPY and GOOG returns

34.2.5 Plot the monthly means and volatilities from the previous exercise

34.2.6 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.

34.2.7 Plot the time-series volatilities of these five portfolios

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

34.2.8 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.

34.2.9 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

35 McKinney Chapter 10 - Practice for Section 04

35.1 Announcements

35.2 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
```

35.2.1 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']
)
)

```

Index Date	min ^DJI	^FTSE	^GSPC	^HSI	^IXIC	^N225	max ^DJI	^FTS
2015-12-31	15666.4404	5874.1001	1867.6100	20556.5996	4506.4902	16795.9609	18312.3906	7104.0
2016-12-31	15660.1797	5537.0000	1829.0800	18319.5801	4266.8398	14952.0195	19974.6191	7142.7
2017-12-31	19732.4004	7099.2002	2257.8301	22134.4707	5429.0801	18335.6309	24837.5098	7687.7
2018-12-31	21792.1992	6584.7002	2351.1001	24585.5293	6192.9199	19155.7402	26828.3906	7877.5
2019-12-31	22686.2207	6692.7002	2447.8899	25064.3594	6463.5000	19561.9609	28645.2598	7686.0
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

35.2.2 Calculate 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.

35.2.3 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.

35.2.4 Calculate monthly means and volatilities for SPY and GOOG returns

35.2.5 Plot the monthly means and volatilities from the previous exercise

35.2.6 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.

35.2.7 Plot the time-series volatilities of these five portfolios

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

35.2.8 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.

35.2.9 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

36 McKinney Chapter 10 - Practice for Section 05

36.1 Announcements

36.2 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
```

36.2.1 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']
)
)

```

Index Date	min ^DJI	^FTSE	^GSPC	^HSI	^IXIC	^N225	max ^DJI	^FTS
2015-12-31	15666.4404	5874.1001	1867.6100	20556.5996	4506.4902	16795.9609	18312.3906	7104.0
2016-12-31	15660.1797	5537.0000	1829.0800	18319.5801	4266.8398	14952.0195	19974.6191	7142.7
2017-12-31	19732.4004	7099.2002	2257.8301	22134.4707	5429.0801	18335.6309	24837.5098	7687.7
2018-12-31	21792.1992	6584.7002	2351.1001	24585.5293	6192.9199	19155.7402	26828.3906	7877.5
2019-12-31	22686.2207	6692.7002	2447.8899	25064.3594	6463.5000	19561.9609	28645.2598	7686.0
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

36.2.2 Calculate 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.

36.2.3 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.

36.2.4 Calculate monthly means and volatilities for SPY and GOOG returns

36.2.5 Plot the monthly means and volatilities from the previous exercise

36.2.6 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.

36.2.7 Plot the time-series volatilities of these five portfolios

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

36.2.8 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.

36.2.9 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

37 McKinney Chapter 11 - Time Series

37.1 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 efficiently 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.

37.2 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
```

37.2.1 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 DataFrames'

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.

37.2.2 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
```

```
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
```

37.3 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.

37.3.1 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 positive 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
```

37.3.2 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')
```

37.4 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()`.

37.4.1 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.

37.4.2 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

37.5 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()
```

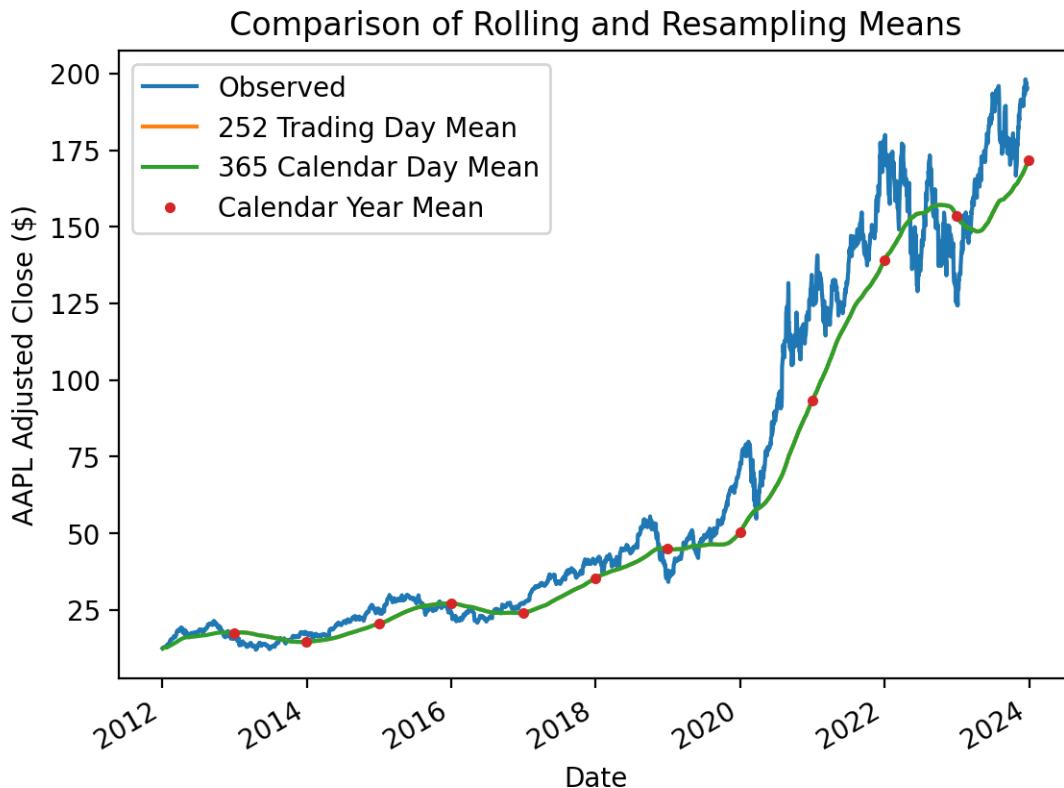
```
[*****100%*****] 3 of 3 completed
```

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.

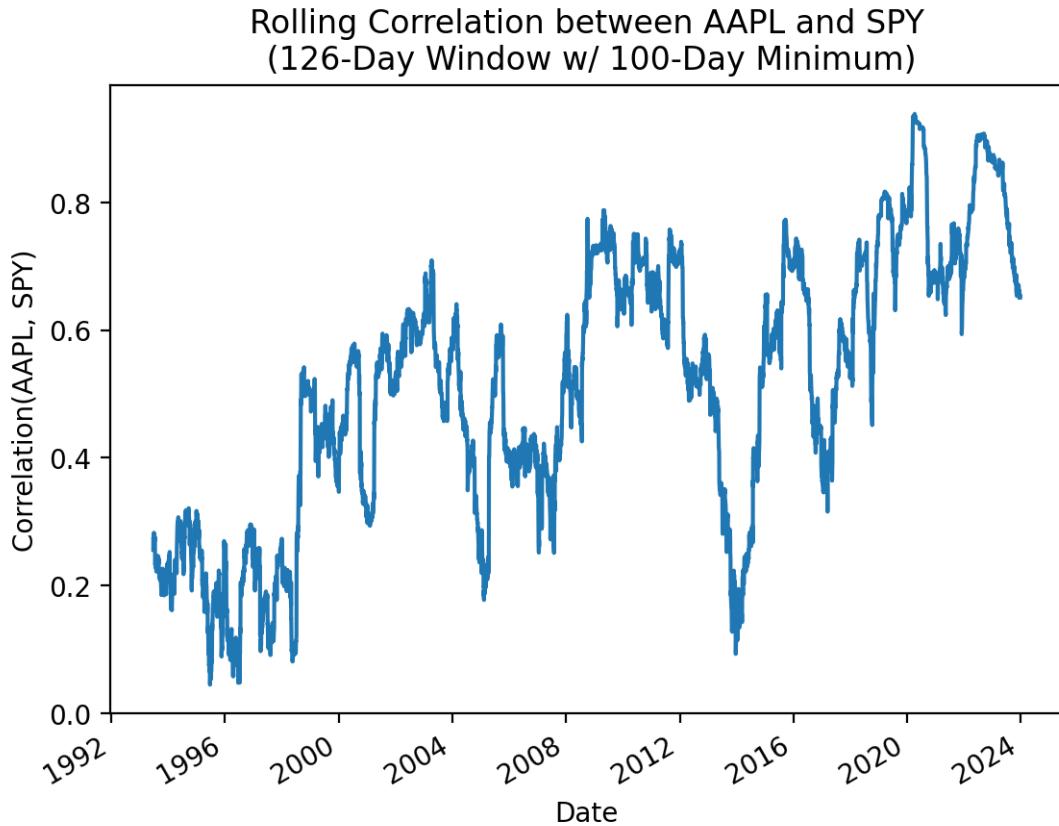
37.5.1 Binary Moving Window Functions

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

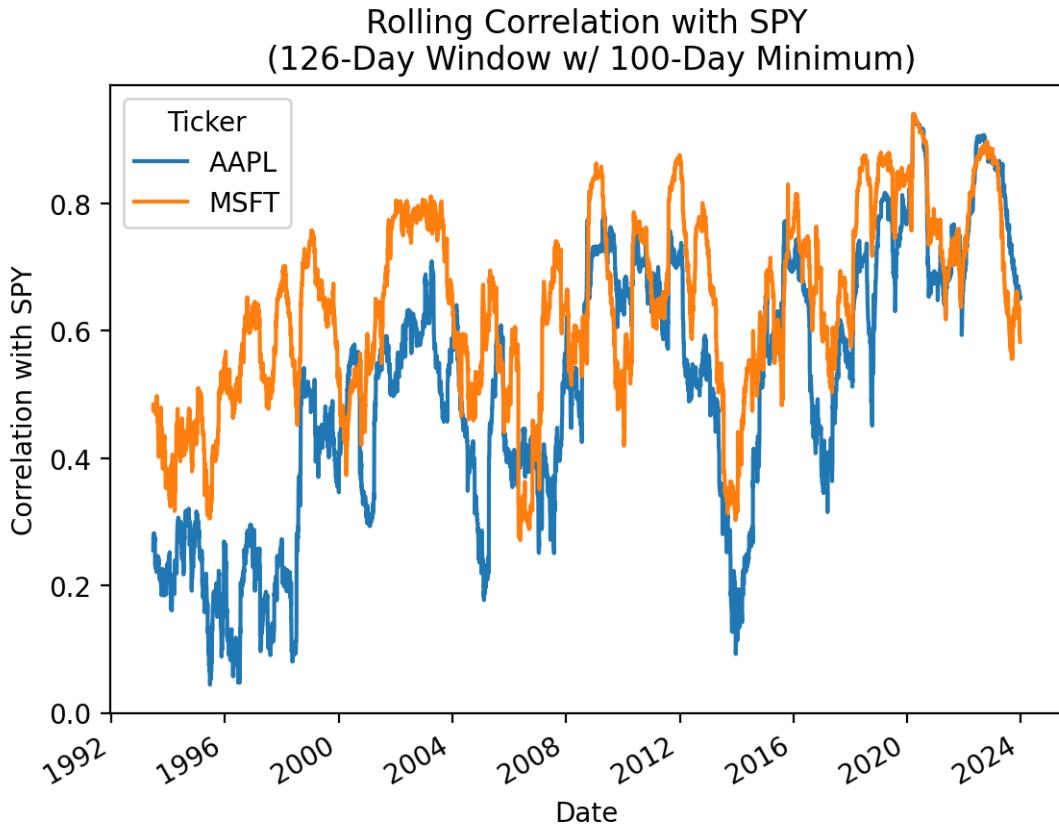
```
returns = df['Adj Close'].pct_change()  
returns.head()
```

Ticker	AAPL	MSFT	SPY
Date			
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()
```

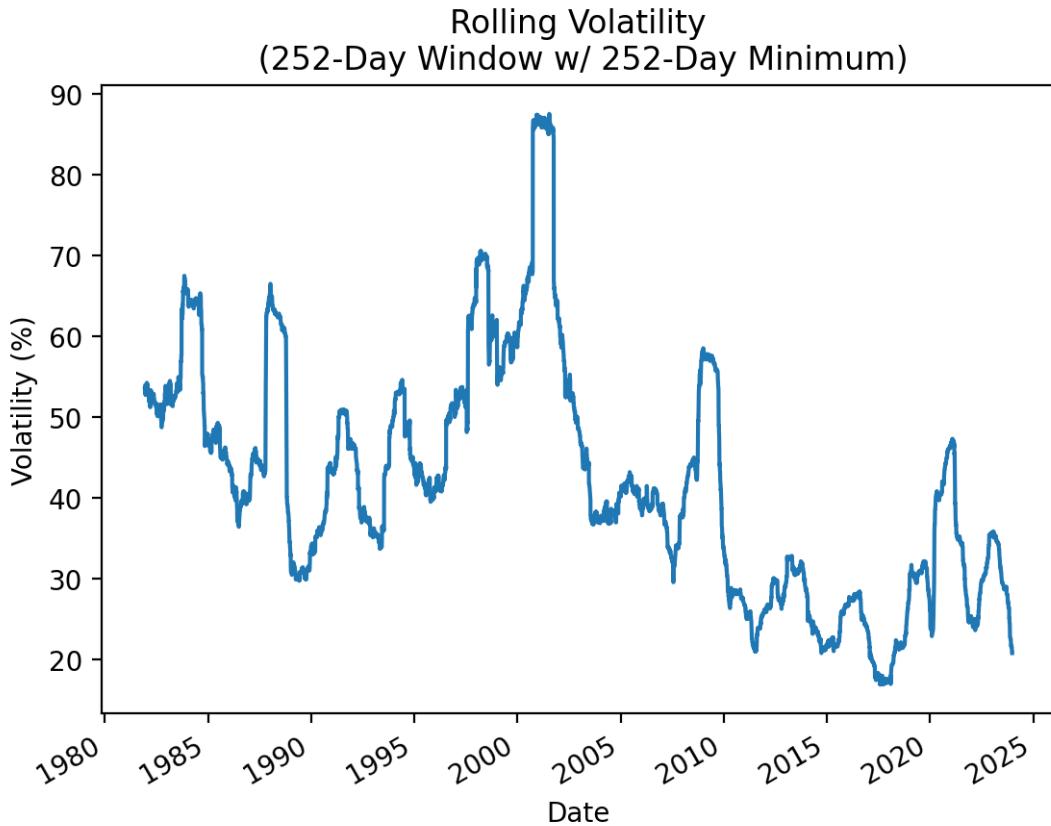


37.5.2 User-Defined Moving Window Functions

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 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)

38 McKinney Chapter 11 - Practice for Section 02

38.1 Announcements

38.2 Practice

38.2.1 Cumulative returns with all available data

38.2.2 Total returns for each calendar year

38.2.3 Total returns over rolling 252-trading-day windows

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

38.2.5 Sharpe Ratios for each calendar year

38.2.6 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: $\beta_{stock} = \frac{Cov(R_{stock} - R_f, R_{market} - R_f)}{Var(R_{market} - R_f)}$.

38.2.7 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.

39 McKinney Chapter 11 - Practice for Section 03

39.1 Announcements

39.2 Practice

39.2.1 Cumulative returns with all available data

39.2.2 Total returns for each calendar year

39.2.3 Total returns over rolling 252-trading-day windows

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

39.2.5 Sharpe Ratios for each calendar year

39.2.6 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: $\beta_{stock} = \frac{Cov(R_{stock} - R_f, R_{market} - R_f)}{Var(R_{market} - R_f)}$.

39.2.7 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.

40 McKinney Chapter 11 - Practice for Section 04

40.1 Announcements

40.2 Practice

40.2.1 Cumulative returns with all available data

40.2.2 Total returns for each calendar year

40.2.3 Total returns over rolling 252-trading-day windows

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

40.2.5 Sharpe Ratios for each calendar year

40.2.6 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: $\beta_{stock} = \frac{Cov(R_{stock}-R_f, R_{market}-R_f)}{Var(R_{market}-R_f)}$.

40.2.7 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.

41 McKinney Chapter 11 - Practice for Section 05

41.1 Announcements

41.2 Practice

41.2.1 Cumulative returns with all available data

41.2.2 Total returns for each calendar year

41.2.3 Total returns over rolling 252-trading-day windows

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

41.2.5 Sharpe Ratios for each calendar year

41.2.6 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: $\beta_{stock} = \frac{Cov(R_{stock} - R_f, R_{market} - R_f)}{Var(R_{market} - R_f)}$.

41.2.7 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

42 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'
```

42.1 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.

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

42.2.1 ...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.

42.2.2 ...Or Not?

The logic above is reasonable. However, if past market prices reflect all relevant information, they should also reflect any price 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.

42.2.2.1 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 arg
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()
```

```

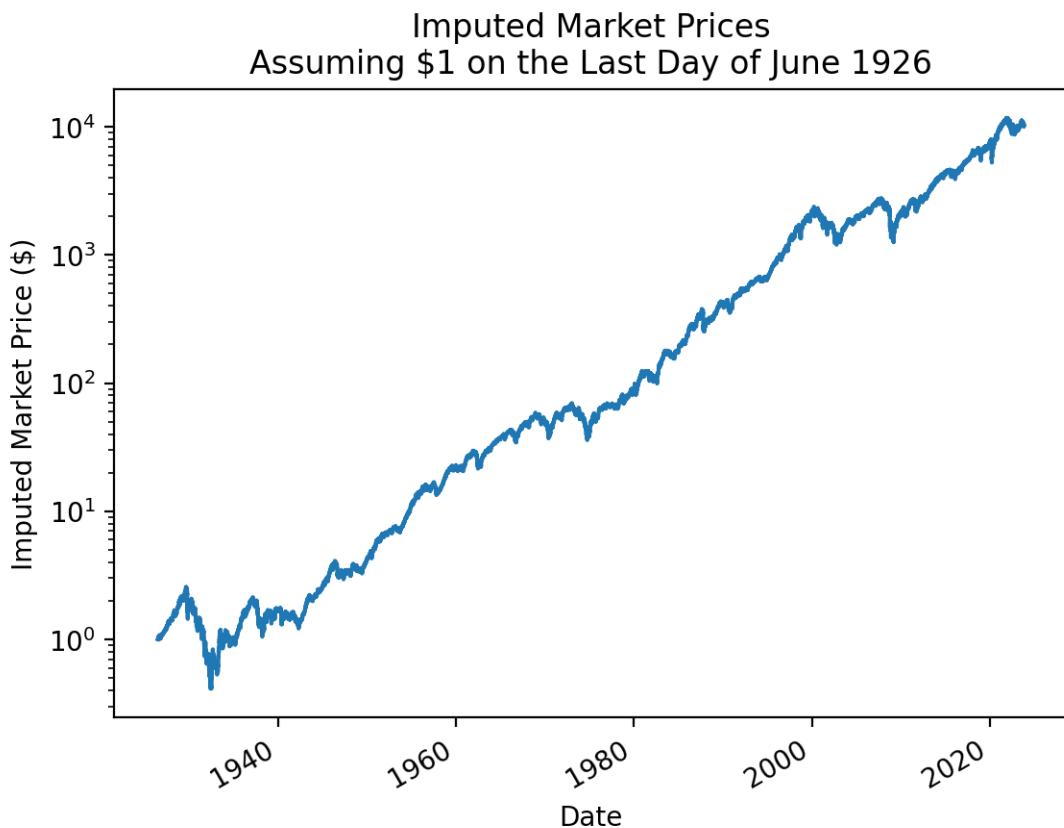
Date
2023-10-25    10168.8352
2023-10-26    10054.0290
2023-10-27    10002.8540
2023-10-30    10119.9874
2023-10-31    10185.8686
Name: Mkt, dtype: float64

```

```

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()

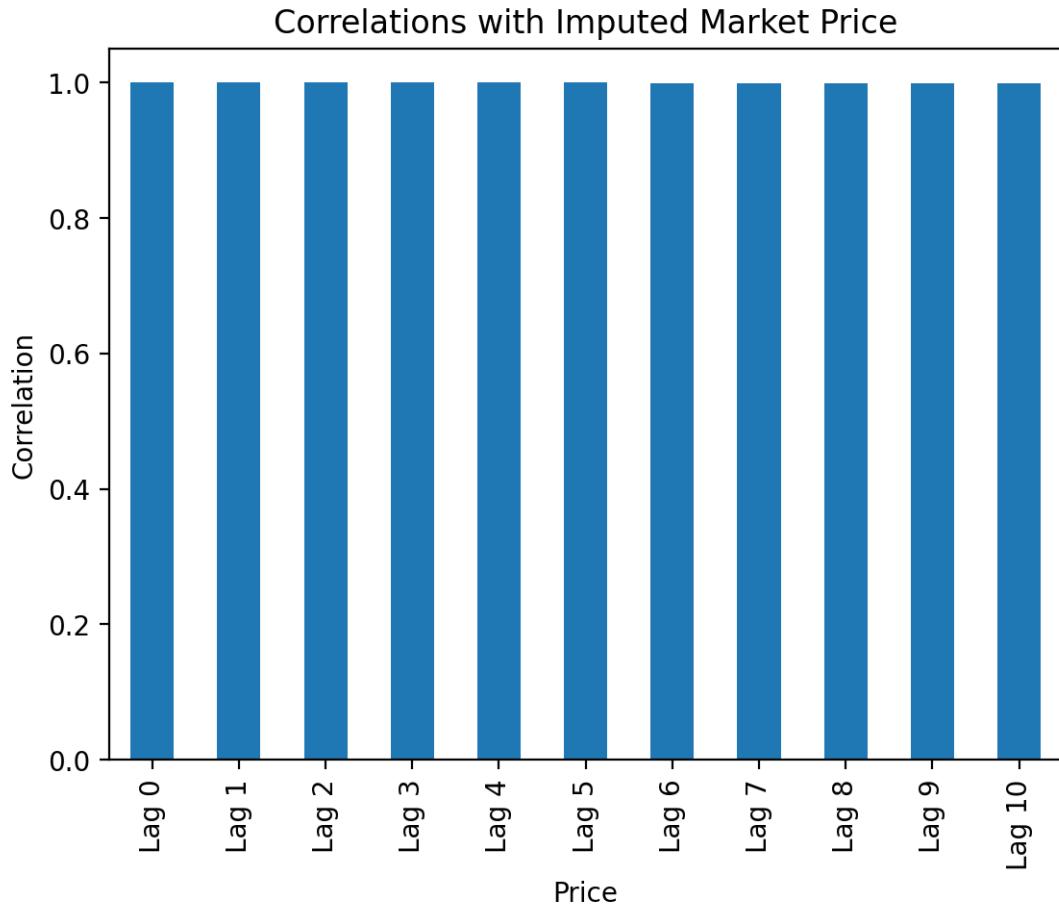
```

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			

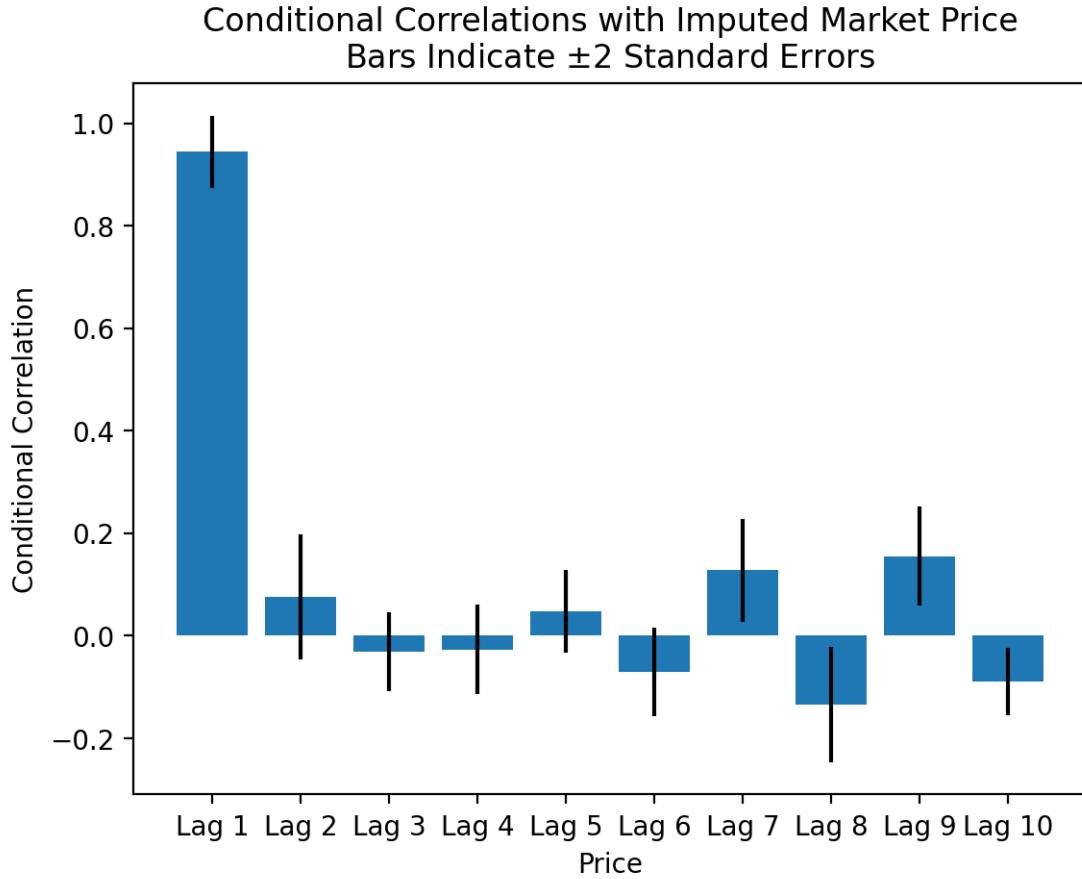
Notes:

- [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 $\pm 2$ Standard Deviations')
plt.ylabel('Conditional Correlation')
plt.xlabel('Price')
plt.show()

```



42.2.2.2 Signal-to-Noise Ratio

Recall, we can express a random walk as $P_t = \rho P_{t-1} + m P_{t-1} + \varepsilon_t$. Since $\rho = 1$, we can subtract P_{t-1} from both sides, then divide by P_{t-1} on both sides. This transformation expresses a random walk in terms of returns: $r_{t-1,t} = m + e_t$, where $E[e_t] = 0$ 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 $t \geq (2 \times \frac{s}{1})^2$ 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`

42.3 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 $\text{SMA}(20)$ to our `btc` data frame
2. Use `np.select()` to add a `Position` column containing:
 - 1 (long) when the adjusted close is greater than $\text{SMA}(20)$
 - 0 (neutral) when the adjusted close is less than (or equal to) $\text{SMA}(20)$
 - We could use `np.where()` instead of `np.select()`, but using `np.select()` provides a more flexible framework for more complex examples
 - 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`
 - 0 if `Position == 0`
 - We could earn the risk-free rate instead of 0 percent, but earning 0 percent simplifies this example

```

btc = (
    btc
    .assign(

```

```

SMA20 = lambda x: x['Adj Close'].rolling(20).mean(),
Position = lambda x: np.select(
    condlist=[x['Adj Close'].shift() > x['SMA20'].shift(), x['Adj Close'].shift() < x['SMA20'].shift()],
    choicelist=[1, 0],
    default=np.nan
),
Strategy = lambda x: x['Position'] * x['Return']
)
)

btc.tail()

```

Variable	Open	High	Low	Close	Adj Close	Volume	Return	SMA20
Date								
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

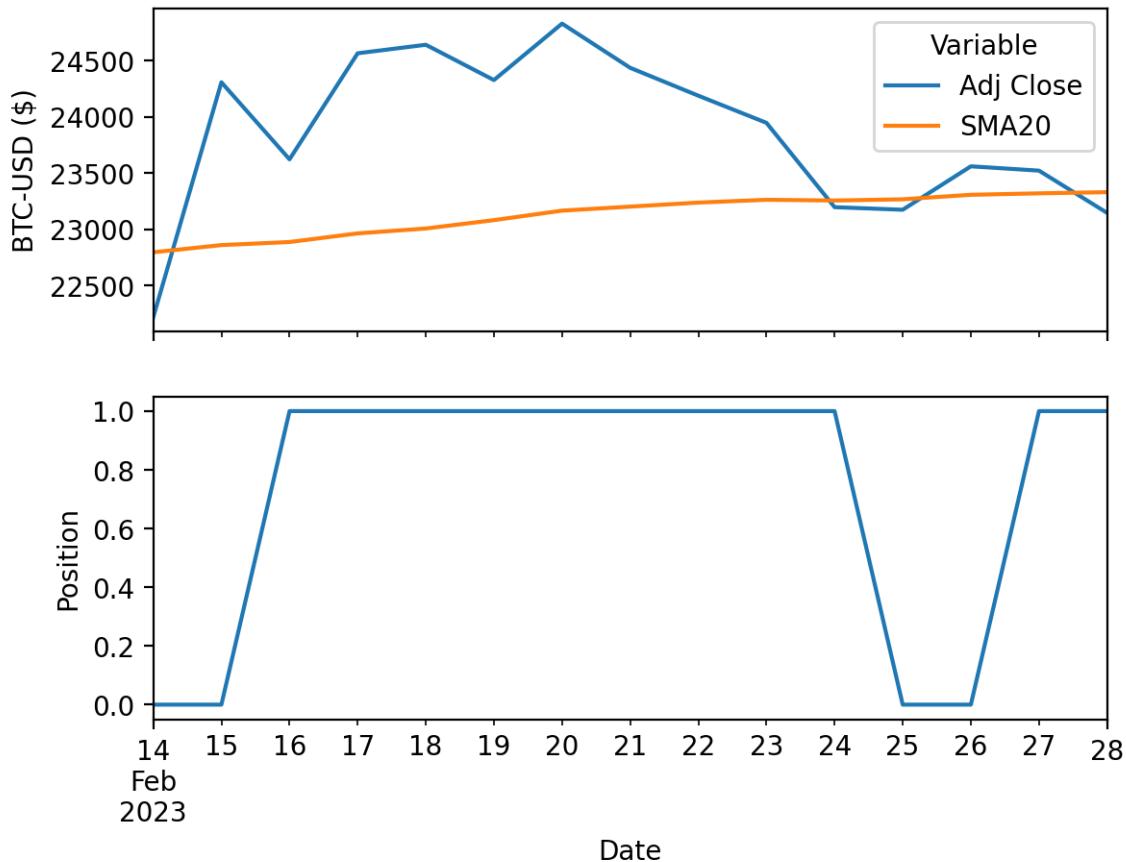
I find it helpful to plot Adj Close, SMA20, and Position for a sort window with one or more crossings.

```

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()

```

Bitcoin SMA(20) Strategy

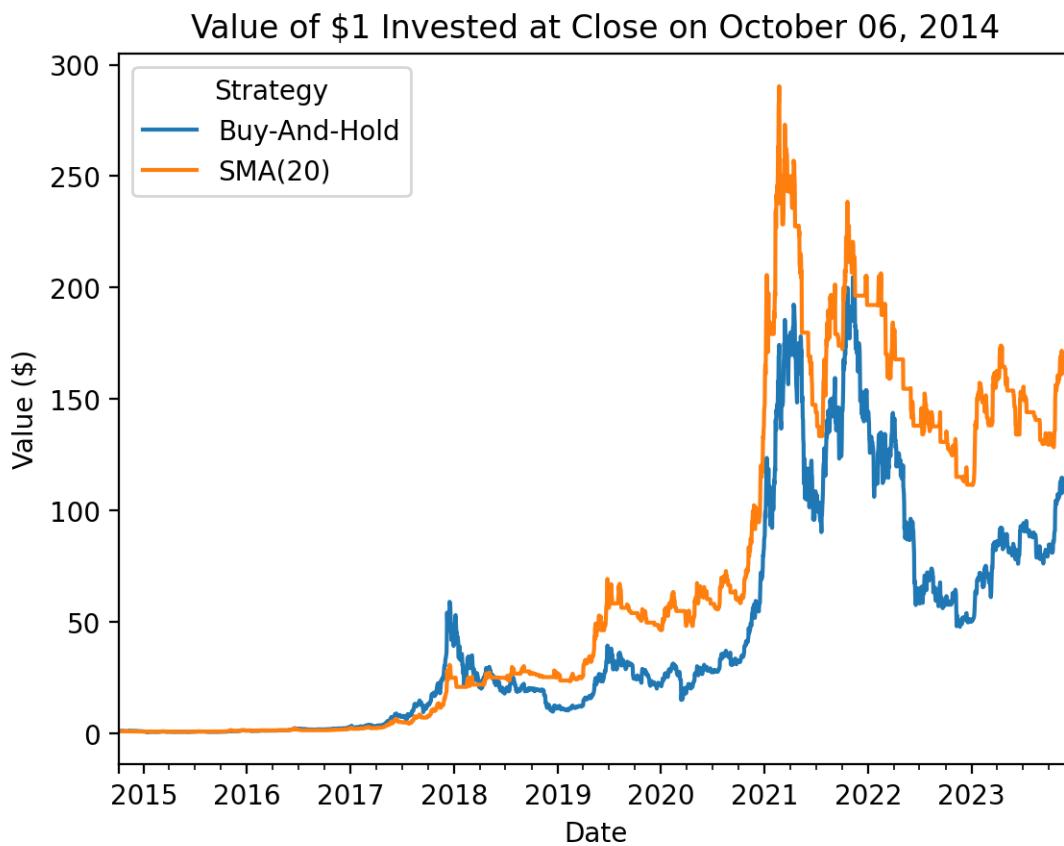


We can compare the long-run performance of buy-and-hold and SMA(20).

```
_ = 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)}:{_.index[0].year}, {_.index[0].month}, {_.index[0].day}')
```

```
plt.show()
```



In the practice notebook, we will dig deeper on this strategy and others.

43 Herron Topic 2 - Practice for Section 02

43.1 Announcements

43.2 Practice

43.2.1 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 `=`).

43.2.2 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.

43.2.3 Implement the SMA(20) strategy with the market factor from French

We need to impute a market price before we calculate SMA(20).

43.2.4 How often does SMA(20) outperform buy-and-hold with 10-year rolling windows?

43.2.5 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.

43.2.6 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

44 Herron Topic 2 - Practice for Section 03

44.1 Announcements

44.2 Practice

44.2.1 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 `=`).

44.2.2 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.

44.2.3 Implement the SMA(20) strategy with the market factor from French

We need to impute a market price before we calculate SMA(20).

44.2.4 How often does SMA(20) outperform buy-and-hold with 10-year rolling windows?

44.2.5 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.

44.2.6 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

45 Herron Topic 2 - Practice for Section 04

45.1 Announcements

45.2 Practice

45.2.1 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 `=`).

45.2.2 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.

45.2.3 Implement the SMA(20) strategy with the market factor from French

We need to impute a market price before we calculate SMA(20).

45.2.4 How often does SMA(20) outperform buy-and-hold with 10-year rolling windows?

45.2.5 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.

45.2.6 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

46 Herron Topic 2 - Practice for Section 05

46.1 Announcements

46.2 Practice

46.2.1 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 `=`).

46.2.2 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.

46.2.3 Implement the SMA(20) strategy with the market factor from French

We need to impute a market price before we calculate SMA(20).

46.2.4 How often does SMA(20) outperform buy-and-hold with 10-year rolling windows?

46.2.5 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.

46.2.6 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

47 Project 2

FINA 6333 – Spring 2023

To be determined

Part XII

Week 12

48 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'
```

48.1 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 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

48.1.1 β Estimation

We can use three (equivalent) approaches to estimate β_i :

1. From covariance and variance as $\beta_i = \frac{\text{Cov}(R_i - R_F, R_M - R_F)}{\text{Var}(R_M - R_F)}$
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 β s (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='famafrrench',
        start='1900',
    )
)

```

C:\Users\r.herron\AppData\Local\Temp\ipykernel_9472\3206174192.py:2: FutureWarning: The argument

```

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

48.1.1.1 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
```

48.1.1.2 Correlation and Standard Deviations

```
_ = aapl[['MktRF', 'RiRF']].dropna()
rho = _.corr()
sigma = _.std()
print(f'rho:\n{rho}\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'] / sigma['MktRF']}")
```

```
Apple beta from rho and sigmas: 1.2602
```

48.1.1.3 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']:.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.

48.1.2 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}''

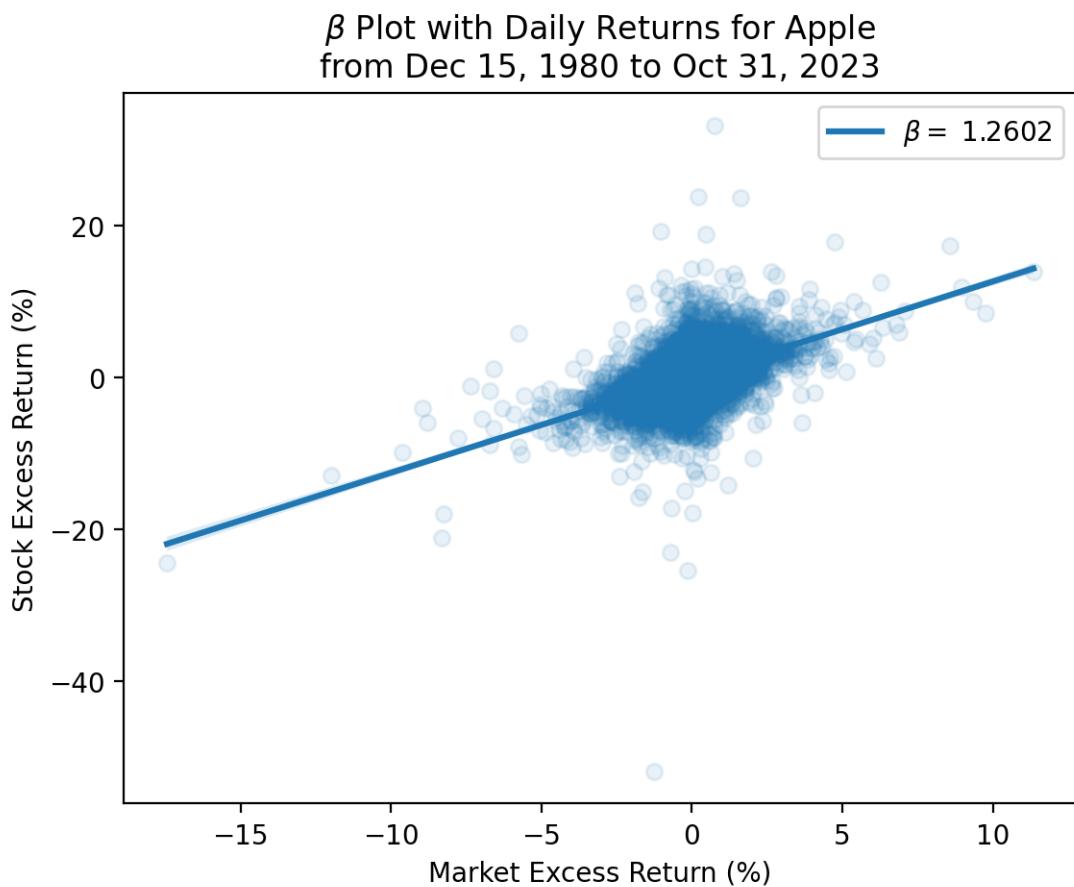
def label_dates(x):
    y = x.dropna()
    return f'from {y.index[0]:%b %d, %Y} to {y.index[-1]:%b %d, %Y}'
```

```

_ = 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()

```

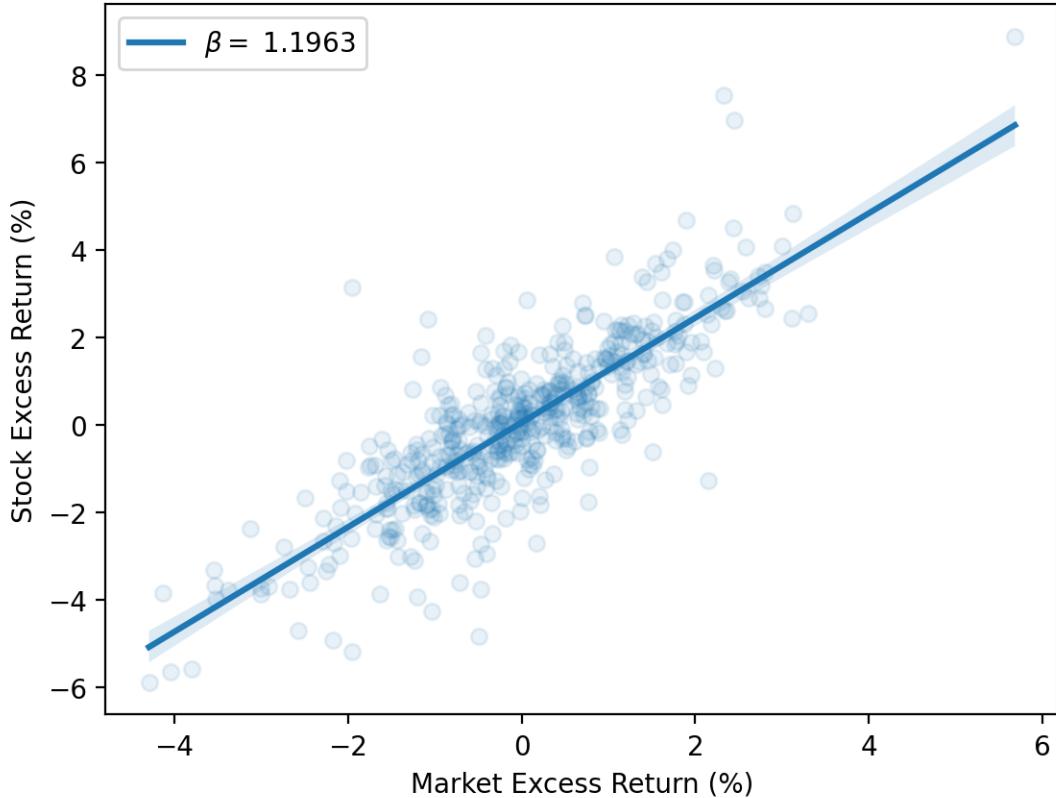


We see a strong relation between Apple and market (excess) returns, but there is a lot of unexplained variation in Apple (excess) returns. The best practice is to estimate β with one to three years of daily data.

```
_ = 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()
```

β Plot with Daily Returns for Apple
from Oct 29, 2021 to Oct 31, 2023



48.1.3 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.

48.1.4 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.

```
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 argument  
    ff_beta = pdr.DataReader()  
  
print(ff_beta['DESCR'])  
  
Portfolios Formed on BETA  
-----  
  
This file was created by CMPT_BETA_RETURNS using the 202310 CRSP database. It contains value-annual returns.  
  
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 β s.

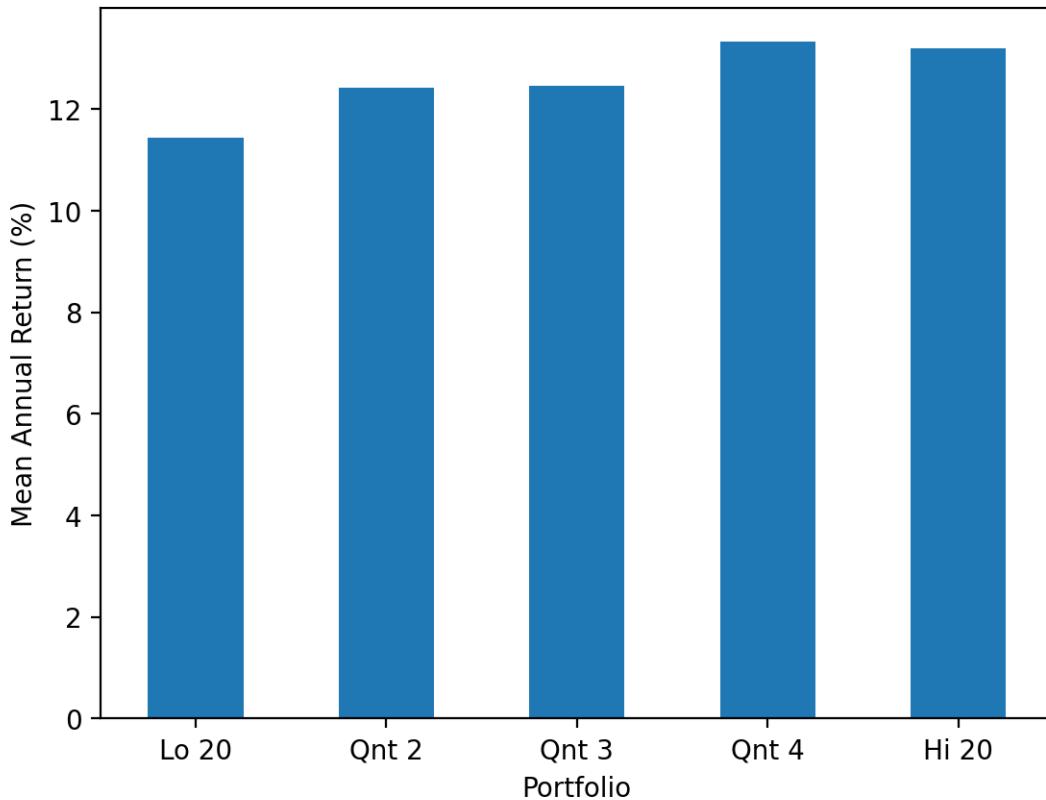
```
ff_beta[2].head()
```

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 portfolios. 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 {_ .index[-1]}')
plt.show()
```

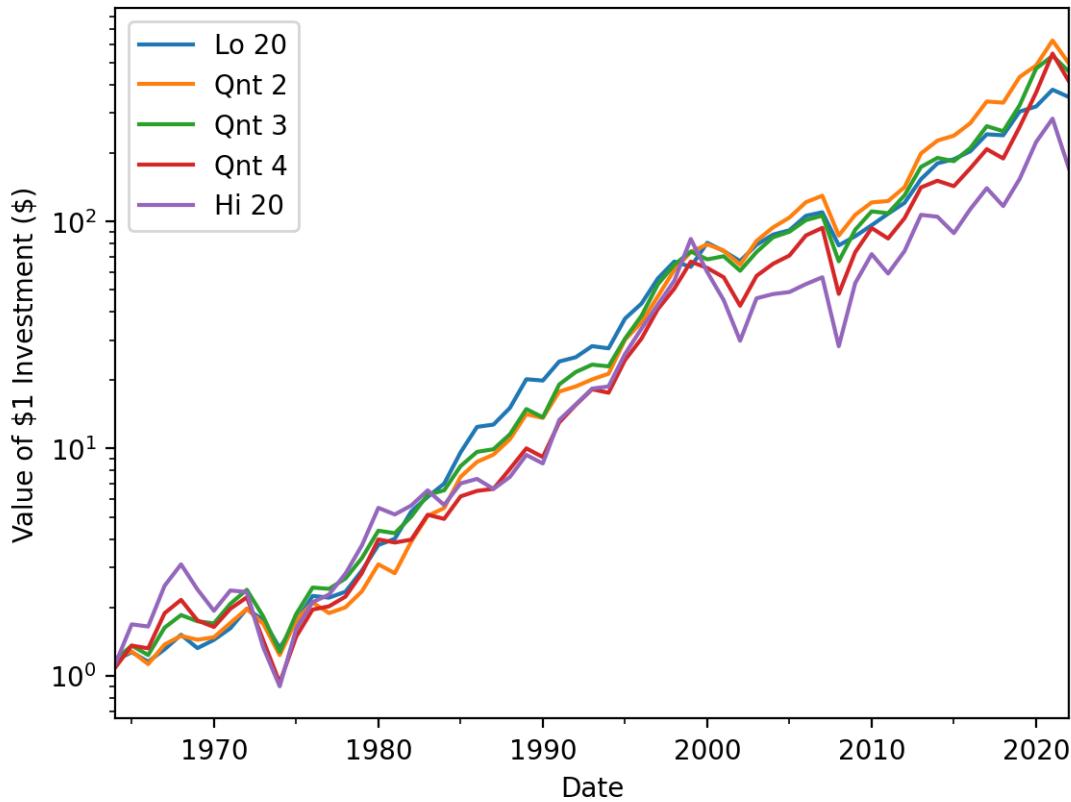
Mean Returns on Portfolios Formed on β
from 1964 to 2022



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 {_')
plt.show()
```

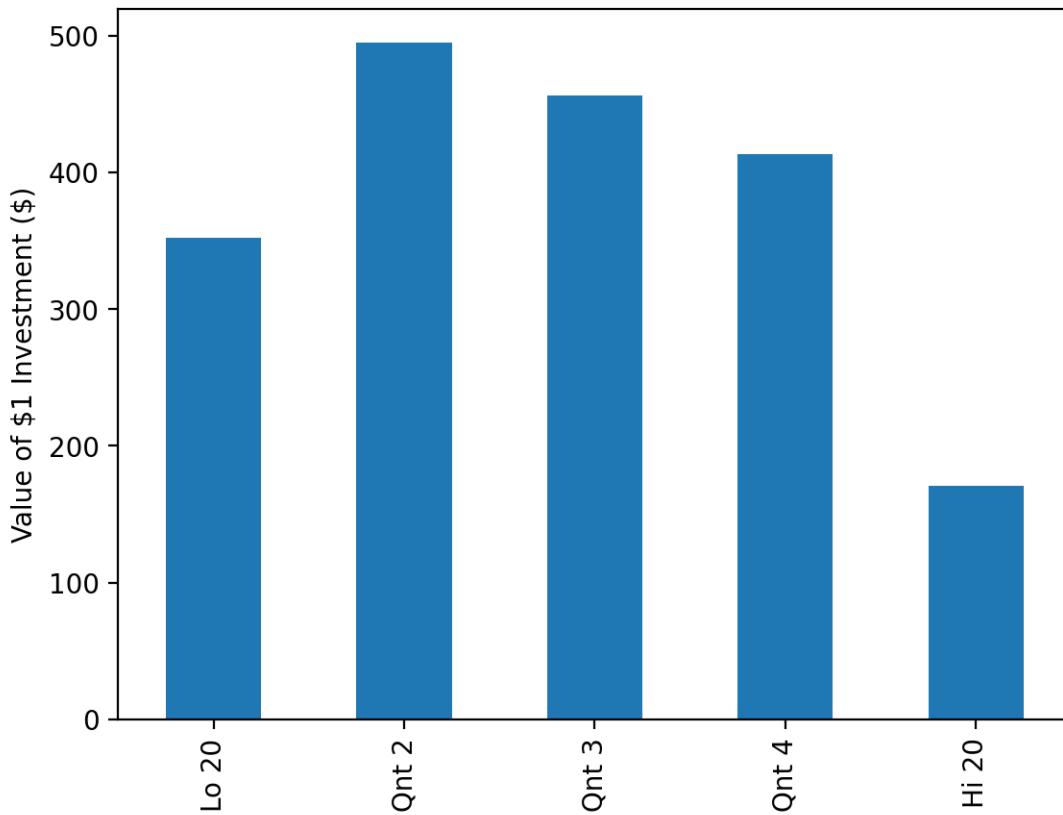
Value of \$1 Investments in Portfolios Formed on β from 1964 to 2022



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 {_.
plt.show()
```

Value of \$1 Investments in Portfolios Formed on β
from 1964 to 2022



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 his free *Corporate Finance* textbook](#).

48.2 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.

48.2.1 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 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 α s and β s in the practice notebook using `RollingOLS()` from `statsmodels.regression.rolling`.

48.2.2 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.

49 Herron Topic 3 - Practice for Section 02

49.1 Announcements

49.2 Practice

49.2.1 Plot the security market line (SML) for a variety of asset classes

Use the past three years of daily data for the following exchange 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)

49.2.2 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.

49.2.3 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.

49.2.4 Estimate the CAPM β s 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 β s and plot cumulative returns. You may want to pick short periods of time with large market swings.

49.2.5 Explore the size factor

49.2.5.1 Estimate α s 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.

49.2.5.2 Are the returns on these ten portfolios formed on size concentrated in a specific month?

49.2.5.3 Compare the size factor to the market factor

You may want to consider mean excess returns by decade.

49.2.6 Repeat the exercises above with the value factor

49.2.7 Repeat the exercises above with the momentum factor

You may find it helpful to consider the worst months and years for the momentum factor.

49.2.8 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 β s changed over the past four decades?

49.2.9 Use the three-, four-, and five-factor models to determine how the ARKK Innovation ETF generates returns

50 Herron Topic 3 - Practice for Section 03

50.1 Announcements

50.2 Practice

50.2.1 Plot the security market line (SML) for a variety of asset classes

Use the past three years of daily data for the following exchange 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)

50.2.2 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.

50.2.3 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.

50.2.4 Estimate the CAPM β s 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 β s and plot cumulative returns. You may want to pick short periods of time with large market swings.

50.2.5 Explore the size factor

50.2.5.1 Estimate α s 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.

50.2.5.2 Are the returns on these ten portfolios formed on size concentrated in a specific month?

50.2.5.3 Compare the size factor to the market factor

You may want to consider mean excess returns by decade.

50.2.6 Repeat the exercises above with the value factor

50.2.7 Repeat the exercises above with the momentum factor

You may find it helpful to consider the worst months and years for the momentum factor.

50.2.8 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 β s changed over the past four decades?

50.2.9 Use the three-, four-, and five-factor models to determine how the ARKK Innovation ETF generates returns

51 Herron Topic 3 - Practice for Section 04

51.1 Announcements

51.2 Practice

51.2.1 Plot the security market line (SML) for a variety of asset classes

Use the past three years of daily data for the following exchange 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)

51.2.2 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.

51.2.3 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.

51.2.4 Estimate the CAPM β s 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 β s and plot cumulative returns. You may want to pick short periods of time with large market swings.

51.2.5 Explore the size factor

51.2.5.1 Estimate α s 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.

51.2.5.2 Are the returns on these ten portfolios formed on size concentrated in a specific month?

51.2.5.3 Compare the size factor to the market factor

You may want to consider mean excess returns by decade.

51.2.6 Repeat the exercises above with the value factor

51.2.7 Repeat the exercises above with the momentum factor

You may find it helpful to consider the worst months and years for the momentum factor.

51.2.8 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 β s changed over the past four decades?

51.2.9 Use the three-, four-, and five-factor models to determine how the ARKK Innovation ETF generates returns

52 Herron Topic 3 - Practice for Section 05

52.1 Announcements

52.2 Practice

52.2.1 Plot the security market line (SML) for a variety of asset classes

Use the past three years of daily data for the following exchange 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)

52.2.2 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.

52.2.3 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.

52.2.4 Estimate the CAPM β s 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 β s and plot cumulative returns. You may want to pick short periods of time with large market swings.

52.2.5 Explore the size factor

52.2.5.1 Estimate α s 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.

52.2.5.2 Are the returns on these ten portfolios formed on size concentrated in a specific month?

52.2.5.3 Compare the size factor to the market factor

You may want to consider mean excess returns by decade.

52.2.6 Repeat the exercises above with the value factor

52.2.7 Repeat the exercises above with the momentum factor

You may find it helpful to consider the worst months and years for the momentum factor.

52.2.8 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 β s changed over the past four decades?

52.2.9 Use the three-, four-, and five-factor models to determine how the ARKK Innovation ETF generates returns

Part XIII

Week 13

53 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
```

53.1 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 defa
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

```

53.2 SciPy's `minimize()` Function

53.2.1 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 `constraints=`.

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

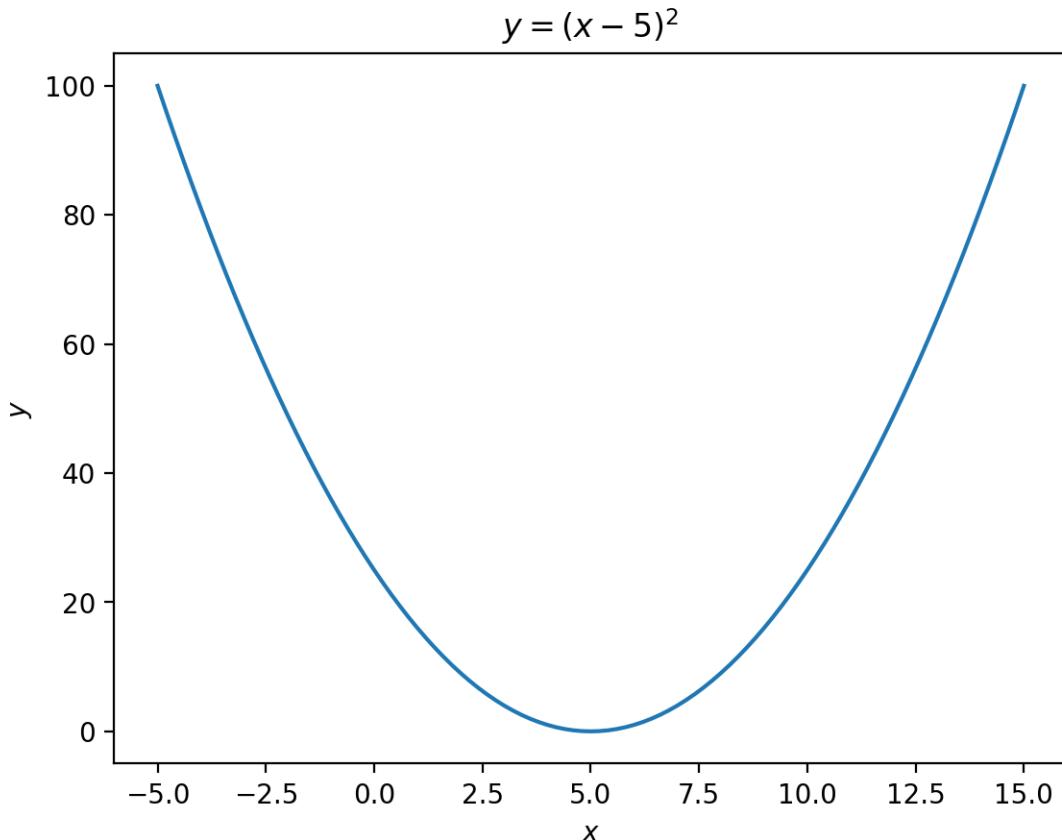
It is helpful to plot $y = (x - a)$ first.

```
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

```

53.2.2 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 weight
    constraints=(
        {'type': 'eq', 'fun': lambda x: x.sum() - 1} # minimize drives "eq" constraints to zero
    )
)

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(pp) * 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

53.2.3 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 one
            {'type': 'eq', 'fun': lambda x: port_mean(x=x, r=returns, ppy=252) - t} # constrain mean to target

    )
    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' +

```

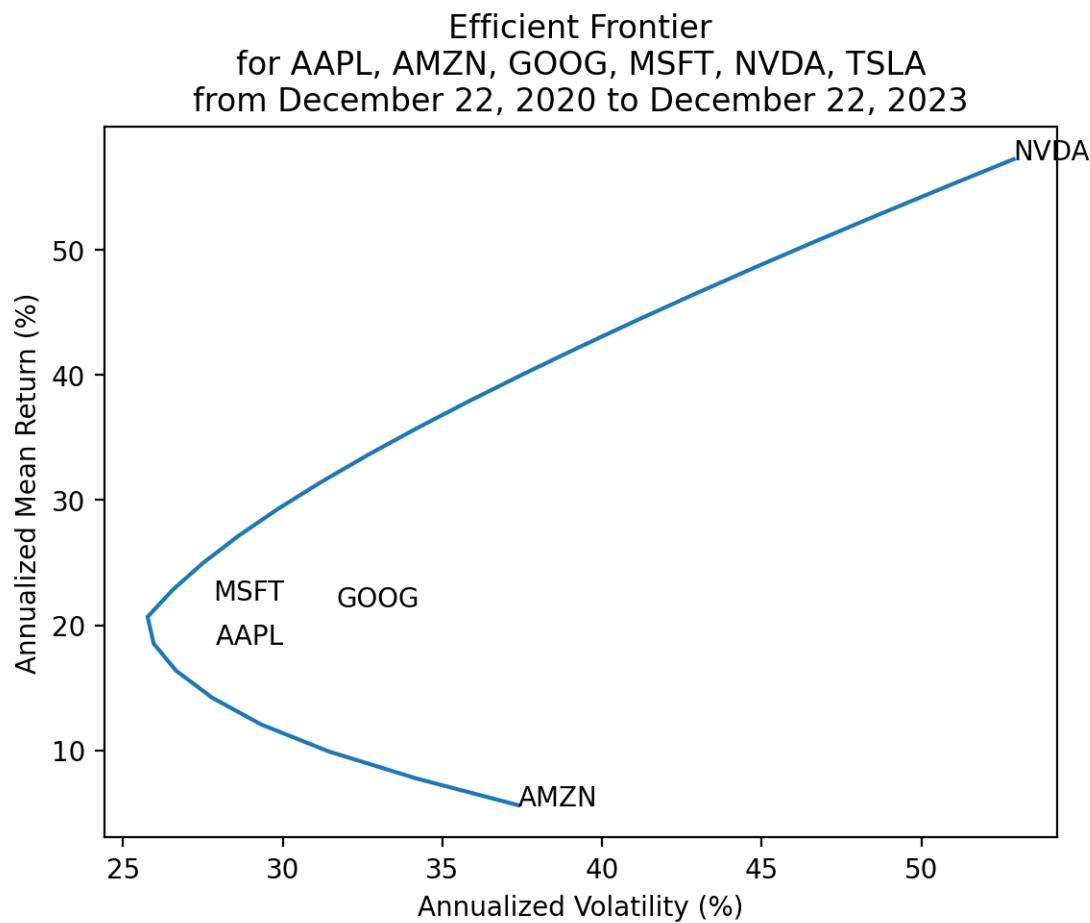
```

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()

```



54 Herron Topic 4 - Practice for Section 02

54.1 Announcements

54.2 Practice

54.2.1 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.

54.2.2 Find the maximum Sharpe Ratio portfolio of MATANA stocks over the last three years, but allow short weights up to 10% on each stock

54.2.3 Find the maximum Sharpe Ratio portfolio of MATANA stocks over the last three years, but allow total short weights of up to 30%

54.2.4 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

54.2.5 Find the minimum 95% Value at Risk (Var) portfolio of MATANA stocks over the last three years

More on VaR [here](#).

54.2.6 Find the minimum maximum draw down portfolio of MATANA stocks over the last three years

54.2.7 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](#).

54.2.8 Plot the (mean-variance) efficient frontier with all available data for the current the DJIA stocks

54.2.9 Find the maximum Sharpe Ratio portfolio with all available data for the current the DJIA stocks

54.2.10 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.

55 Herron Topic 4 - Practice for Section 03

55.1 Announcements

55.2 Practice

55.2.1 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.

55.2.2 Find the maximum Sharpe Ratio portfolio of MATANA stocks over the last three years, but allow short weights up to 10% on each stock

55.2.3 Find the maximum Sharpe Ratio portfolio of MATANA stocks over the last three years, but allow total short weights of up to 30%

55.2.4 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

55.2.5 Find the minimum 95% Value at Risk (Var) portfolio of MATANA stocks over the last three years

More on VaR [here](#).

55.2.6 Find the minimum maximum draw down portfolio of MATANA stocks over the last three years

55.2.7 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](#).

55.2.8 Plot the (mean-variance) efficient frontier with all available data for the current the DJIA stocks

55.2.9 Find the maximum Sharpe Ratio portfolio with all available data for the current the DJIA stocks

55.2.10 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.

56 Herron Topic 4 - Practice for Section 04

56.1 Announcements

56.2 Practice

56.2.1 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.

56.2.2 Find the maximum Sharpe Ratio portfolio of MATANA stocks over the last three years, but allow short weights up to 10% on each stock

56.2.3 Find the maximum Sharpe Ratio portfolio of MATANA stocks over the last three years, but allow total short weights of up to 30%

56.2.4 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

56.2.5 Find the minimum 95% Value at Risk (Var) portfolio of MATANA stocks over the last three years

More on VaR [here](#).

56.2.6 Find the minimum maximum draw down portfolio of MATANA stocks over the last three years

56.2.7 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](#).

56.2.8 Plot the (mean-variance) efficient frontier with all available data for the current the DJIA stocks

56.2.9 Find the maximum Sharpe Ratio portfolio with all available data for the current the DJIA stocks

56.2.10 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.

57 Herron Topic 4 - Practice for Section 05

57.1 Announcements

57.2 Practice

57.2.1 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.

57.2.2 Find the maximum Sharpe Ratio portfolio of MATANA stocks over the last three years, but allow short weights up to 10% on each stock

57.2.3 Find the maximum Sharpe Ratio portfolio of MATANA stocks over the last three years, but allow total short weights of up to 30%

57.2.4 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

57.2.5 Find the minimum 95% Value at Risk (Var) portfolio of MATANA stocks over the last three years

More on VaR [here](#).

57.2.6 Find the minimum maximum draw down portfolio of MATANA stocks over the last three years

57.2.7 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](#).

57.2.8 Plot the (mean-variance) efficient frontier with all available data for the current the DJIA stocks

57.2.9 Find the maximum Sharpe Ratio portfolio with all available data for the current the DJIA stocks

57.2.10 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

58 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
```

58.1 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](#).

58.1.1 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: $S(t) = S_0 \exp((\mu - \frac{1}{2}\sigma^2)t + \sigma W_t)$. We can apply this closed form solution recursively: $S(t_{i+1}) = S(t_i) \exp((\mu - \frac{1}{2}\sigma^2)(t_{i+1} - t_i) + \sigma \sqrt{t_{i+1} - t_i} Z_{i+1})$. Here, Z_i is a Standard Normal random variable (dW_t are independent and normally distributed) and $i = 0, \dots, T - 1$ is the time index.

We can use this closed form solution to simulate stock prices for AAPL.

```
aapl = (
    yf.download(tickers='AAPL')
    .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

```

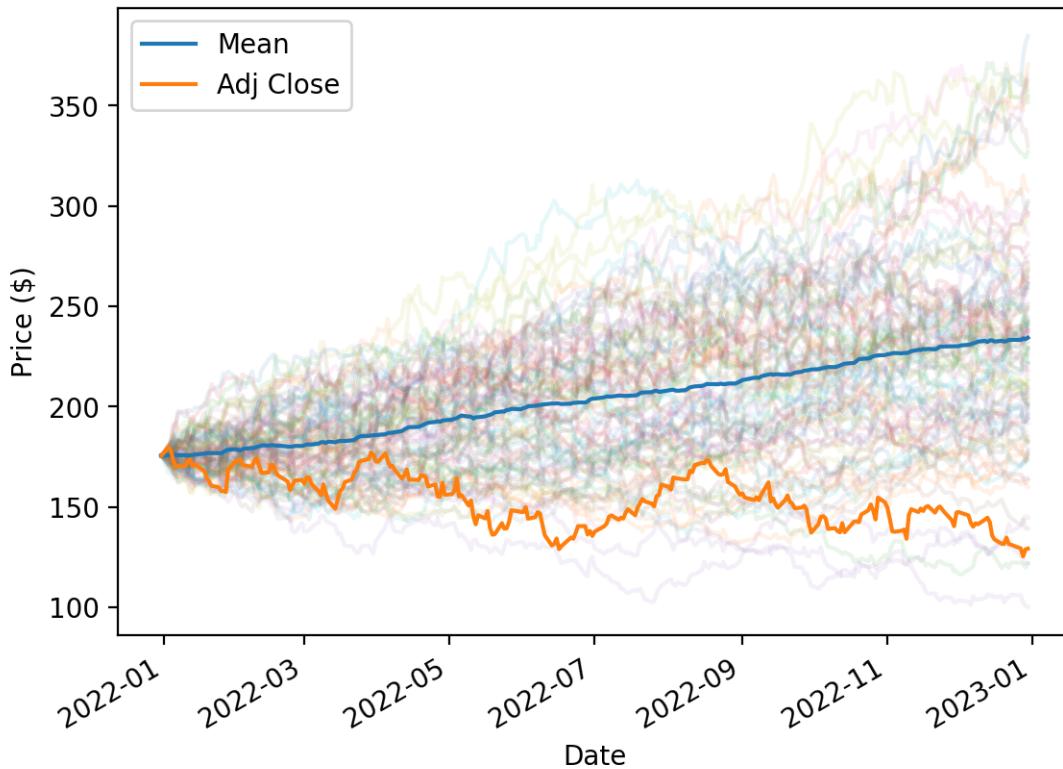
Simulation Date	0	1	2	3	4	5	6	7	8	9	...	90
2021-12-31	175.56	175.56	175.56	175.56	175.56	175.56	175.56	175.56	175.56	175.56	...	175.56
2022-01-03	180.72	180.32	174.60	180.79	175.89	176.98	174.89	180.51	176.00	175.75	...	175.75
2022-01-04	182.06	178.77	174.62	182.23	177.47	176.24	177.10	179.37	179.25	175.13	...	174.00
2022-01-05	185.09	177.47	169.00	182.70	174.89	183.34	177.90	179.66	174.01	172.25	...	173.25
2022-01-06	191.97	174.67	173.63	177.59	177.00	182.80	175.57	181.01	170.41	172.40	...	171.25
...
2022-12-23	263.30	307.78	208.45	242.89	279.85	227.13	237.51	207.87	216.25	252.80	...	238.90
2022-12-27	268.12	305.95	209.92	251.23	278.82	226.46	234.31	206.16	212.62	259.22	...	232.00
2022-12-28	264.97	315.16	211.85	255.79	275.85	224.22	238.65	206.03	211.34	255.58	...	237.00
2022-12-29	259.17	309.05	210.96	256.73	274.65	225.95	238.08	201.73	211.18	258.81	...	241.00
2022-12-30	261.59	307.70	210.24	264.08	275.53	225.67	242.50	201.41	213.21	261.36	...	236.75

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()
```

Apple Simulated and Observed Prices
Trained from 2021-01-04 to 2021-12-31



58.1.2 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 maturity (in years)
    r : float
        Risk-free interest rate
    sigma : float
        Volatility
    type : str
        Option type ('call' or 'put')
    Returns
    -----
    option_price : float
        Option price
    """
    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':
        option_price = S_0 * norm.cdf(d1) - K * np.exp(-r * T) * norm.cdf(d2)
    else:
        option_price = K * np.exp(-r * T) * norm.cdf(-d2) - S_0 * norm.cdf(-d1)
    return option_price
```

```

    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: {_:.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 our 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	...	175.56
2022-01-03	180.54	180.14	174.42	180.61	175.72	176.81	174.71	180.33	175.83	175.58	...	177.42
2022-01-04	181.70	178.42	174.28	181.88	177.12	175.90	176.75	179.02	178.90	174.79	...	176.70
2022-01-05	184.55	176.95	168.51	182.17	174.37	182.80	177.37	179.13	173.49	171.75	...	179.45
2022-01-06	191.21	173.99	172.95	176.89	176.31	182.09	174.88	180.30	169.75	171.72	...	182.31
...
2022-12-23	206.58	241.48	163.55	190.57	219.57	178.21	186.35	163.09	169.67	198.34	...	155.58
2022-12-27	210.16	239.81	164.54	196.92	218.54	177.50	183.66	161.60	166.65	203.18	...	159.06
2022-12-28	207.49	246.79	165.89	200.30	216.00	175.58	186.88	161.33	165.49	200.14	...	157.49
2022-12-29	202.75	241.77	165.03	200.84	214.86	176.76	186.25	157.81	165.20	202.46	...	155.95
2022-12-30	204.44	240.47	164.31	206.39	215.33	176.37	189.52	157.40	166.63	204.26	...	155.84

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: {_:.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.

58.2 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 beyond the scope of this lecture notebook.

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

returns
```

Ticker	GOOG	META
Date		
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	GOOG	META
Ticker		
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)

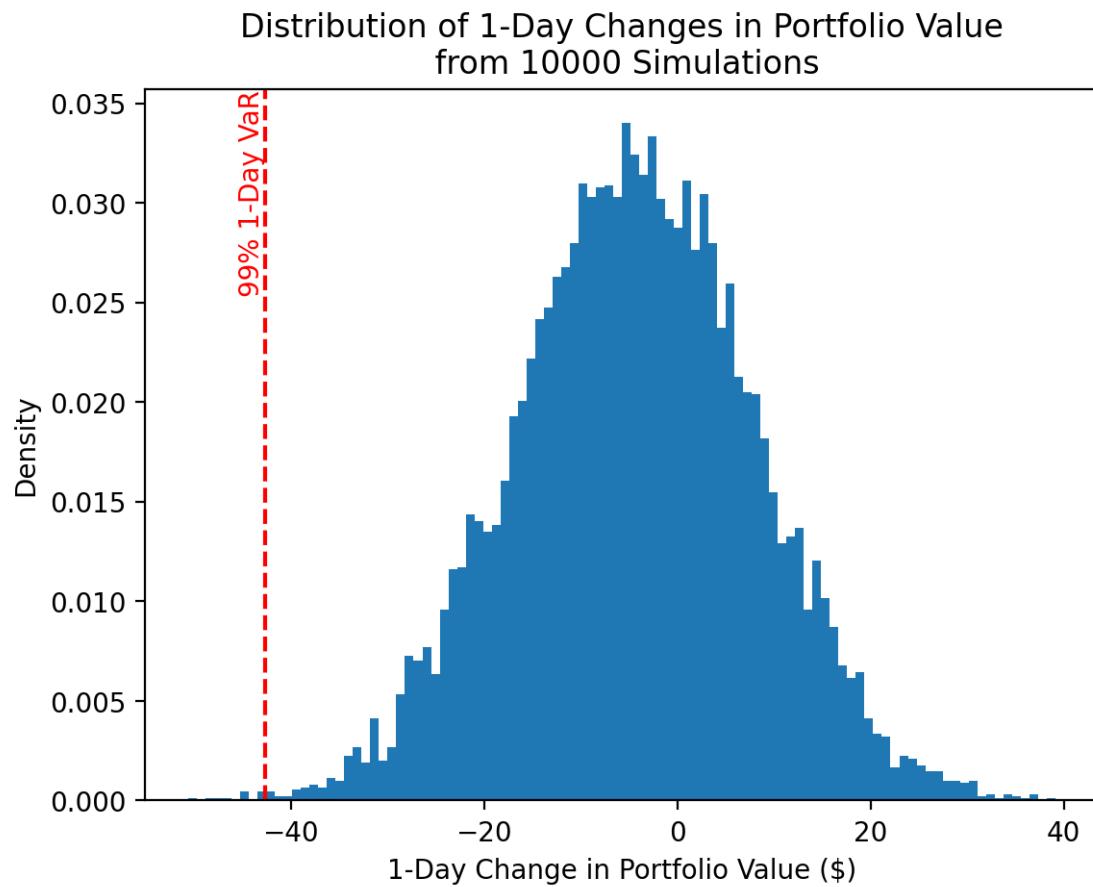
for x, y in zip(percentiles, var):
    print(f'1-day VaR with {100-x}% confidence: ${-y:.2f}')

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()
```



59 Herron Topic 5 - Practice for Section 02

59.1 Announcements

59.2 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
```

59.2.1 Estimate π by simulating darts thrown at a dart board

Hints: Select random xs and ys such that $-r \leq x \leq +r$ and $-r \leq y \leq +r$. Darts are on the board if $x^2 + y^2 \leq r^2$. The area of the circular 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 $f = \frac{\pi r^2}{4r^2}$.

59.2.2 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 .

59.2.3 Repeat the exercise above but add end-of-month investments C_t

60 Herron Topic 5 - Practice for Section 03

60.1 Announcements

60.2 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
```

60.2.1 Estimate π by simulating darts thrown at a dart board

Hints: Select random xs and ys such that $-r \leq x \leq +r$ and $-r \leq y \leq +r$. Darts are on the board if $x^2 + y^2 \leq r^2$. The area of the circular 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 $f = \frac{\pi r^2}{4r^2}$.

60.2.2 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 .

60.2.3 Repeat the exercise above but add end-of-month investments C_t

61 Herron Topic 5 - Practice for Section 04

61.1 Announcements

61.2 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
```

61.2.1 Estimate π by simulating darts thrown at a dart board

Hints: Select random xs and ys such that $-r \leq x \leq +r$ and $-r \leq y \leq +r$. Darts are on the board if $x^2 + y^2 \leq r^2$. The area of the circular 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 $f = \frac{\pi r^2}{4r^2}$.

61.2.2 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 .

61.2.3 Repeat the exercise above but add end-of-month investments C_t

62 Herron Topic 5 - Practice for Section 05

62.1 Announcements

62.2 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
```

62.2.1 Estimate π by simulating darts thrown at a dart board

Hints: Select random xs and ys such that $-r \leq x \leq +r$ and $-r \leq y \leq +r$. Darts are on the board if $x^2 + y^2 \leq r^2$. The area of the circular 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 $f = \frac{\pi r^2}{4r^2}$.

62.2.2 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 .

62.2.3 Repeat the exercise above but add end-of-month investments C_t

Part XV

Week 15

63 Project 3

FINA 6333 – Spring 2023

To be determined