Data Analytics in Finance

FINA 6333 for Spring 2025

Richard Herron

Table of contents

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

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

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

Week 1

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

Introduction

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

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

Language Semantics

Indentation, not braces

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

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

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

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

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

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

Comments

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

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

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

Function and object method calls

> You call functions using parentheses and passing zero or more arguments, optionally assigning the returned value to a variable:
>
> ```
> result = f(x, y, z)
> g()
> ```
>
> Almost every object in Python has attached functions, known as methods, that have access to the object's internal contents. You can call them using the following syntax:
>
> ```
> obj.some_method(x, y, z)
> ```
>
> Functions can take both positional and keyword arguments:
>
> ```
> result = f(a, b, c, d=5, e='foo')
> ```
>
> More on this later.

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

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

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

```
10
```

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

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

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

```
'5 5'
```

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

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

Please see the following example.

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

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

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

```
returned
```

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

Variables and argument passing

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

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

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

```
a is b
```

```
True
```

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

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

```
a
```

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

```
b
```

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

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

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

```
b
```

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

```
a
```

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

Dynamic references, strong types

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

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

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

```
a
```

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

```
type(a)
```

```
list
```

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

```
int
```

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

```
str
```

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

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

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

However, Python will implicitly convert integers to floats.

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

```
2.25
```

Attributes and methods

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

```
a = 'foo'
```

```
a.capitalize()
```

```
'Foo'
```

Binary operators and comparisons

Binary operators operate on two arguments.

```
5 - 7
```

```
-2
```

```
12 + 21.5
```

```
33.5
```

```
5 <= 2
```

```
False
```

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

- `a + b` : Add a and b
- `a - b` : Subtract b from a

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

Mutable and immutable objects

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

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

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

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

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

```
a_list[0]
```

```
'foo'
```

```
a_list[2]
```

```
[4, 5]
```

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

```
4
```

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

```
a_list
```

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

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

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

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

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

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

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

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

```
tuple
```

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

Scalar Types

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

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

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

Numeric types

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

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

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

```
3 // 2
```

```
1
```

Booleans

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

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

```
True and True
```

```
True
```

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

```
True
```

```
False and True
```

```
False
```

```
False or True
```

```
True
```

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

```
True
```

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

```
True & True
```

```
True
```

```
False & True
```

```
False
```

```
False | True
```

```
True
```

Type casting

We can "recast" variables to change their types.

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

```
str
```

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

```
4.14159
```

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

```
float
```

```
int(fval)
```

```
3
```

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

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

```
10
```

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

```
'55'
```

None

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

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

```
True
```

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

```
True
```

```
type(None)
```

```
NoneType
```

Control Flow

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

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

if, elif, and else

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

```
int
```

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

```
It's negative
```

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

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

```
x = 10
if x < 0:
    print("It's negative")
elif x == 0:
    print('Equal to zero')
elif 0 < x < 5:
    print('Positive but smaller than 5')
else:
    print('Positive and larger than or equal to 5')
```

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

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

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

```
Made it
```

for loops

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

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

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

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

```
total
```

```
12
```

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

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

```
total_until_5
```

```
13
```

range

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

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

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

```
range(10)
```

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

We can cast a range to a list.

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

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

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

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

```
[]
```

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

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

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

Ternary expressions

We can complete simple comparisons on one line in Python.

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

```
'Non-negative'
```

McKinney Chapter 2 - Practice - Blank

Announcements

Five-Minute Review

Practice

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

```
lb = 20080915
```

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

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

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

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

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

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

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

FizzBuzz

Solve FizzBuzz.

Use ternary expressions to make your FizzBuzz solution more compact.

Triangle

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

```
1
22
333
4444
55555
```

Two Sum

Write a function `two_sum` that does the following.

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

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

You can return the answer in any order.

Here are some examples:

Example 1:

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

Example 2:

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

Example 3:

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

I saw this question on LeetCode.

Best Time

Write a function `best_time` that solves the following.

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

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

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

Here are some examples:

Example 1:

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

Example 2:

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

I saw this question on LeetCode.

Week 2

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

Introduction

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

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

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

Data Structures and Sequences

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

Tuple

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

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

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

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

```
tup[0]
```

```
4
```

```
tup[1]
```

```
5
```

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

```
nested_tup[0]
```

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

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

```
4
```

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

```
tup
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Unpacking tuples

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

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

```
(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
```

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
```

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
```

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!

Slicing

Slicing is very important!

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

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

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

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

```
seq[5]
```

```
6
```

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

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

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

```
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[::2]
```

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

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

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

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

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

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

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

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

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

dict

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

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

```
empty_dict = {}
empty_dict
```

```
{}
```

A dictionary is a set of key-value pairs.

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

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

```
'some value'
```

```
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'])
```

List, Set, and Dict Comprehensions

We will focus on list comprehensions, which are Pythonic.

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

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

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

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

caps
```

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

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

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

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

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

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

squares
```

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

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

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

Functions

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

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

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

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

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

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

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

Returning Multiple Values

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

```
def f():
    a = 5
    b = 6
```

```
    c = 7
    return (a, b, c)
```

```
f()
```

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

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

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

```
f()
```

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

Anonymous (Lambda) Functions

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

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

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

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

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

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

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

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

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

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

McKinney Chapter 3 - Practice - Blank

Announcements

Five-Minute Review

Practice

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

```
a = 1
```

```
b = 2
```

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

```
a = 1
```

```
b = 2
```

What is the output of the following code and why?

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

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

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

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

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

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

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

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

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

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

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

Example 1:

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

Example 2:

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

I saw this question on LeetCode.

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

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

You may return the answer in any order.

Example 1:

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

Example 2:

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

I saw this question on LeetCode.

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

Test whether the given strings are palindromes.

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

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

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

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

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

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

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

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

Week 3

McKinney Chapter 4 - NumPy Basics: Arrays and Vectorized Computation

```
import numpy as np
```

```
%precision 4
```

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

Introduction

Chapter 4 of McKinney (2022) discusses the NumPy package (an abbreviation of numerical Python), which is the foundation for numerical computing in Python, including pandas.

We will focus on:

1. Creating arrays
2. Slicing arrays
3. Applying functions and methods to arrays
4. Using conditional logic with arrays (i.e., `np.where()` and `np.select()`)

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

Here is a simple example of NumPy's speed and syntax advantages relative to Python's built-in data structures. First, we create a list and a NumPy array with values from 0 to 999,999.

```
my_list = list(range(1_000_000))
my_arr = np.arange(1_000_000)
```

We must use a for loop or a list comprehension to double each value in `my_list`. We will comment this code because it prints a list with one million elements!

```
# [2 * x for x in my_list] # list comprehension to double each value
```

However, we can multiply `my_arr` by two because math "just works" with NumPy. Jupyter will pretty print the NumPy array, showing only the first and last few elements.

```
my_arr * 2
```

```
array([      0,       2,       4, ..., 1999994, 1999996, 1999998],
      shape=(1000000,))
```

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

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

```
57.3 ms ± 6.12 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

```
%timeit my_arr * 2
```

```
3.11 ms ± 702  s per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

The NumPy version is much faster than the list version! The NumPy version is also faster to type, read, and troubleshoot, and our time is more valuable than computer time!

The NumPy ndarray: A Multidimensional Array Object

> One of the key features of NumPy is its N-dimensional array object, or ndarray, which is a fast, flexible container for large datasets in Python. Arrays enable you to perform mathematical operations on whole blocks of data using similar syntax to the equivalent operations between scalar elements.

```
np.random.seed(42) # makes random numbers repeatable
data = np.random.randn(2, 3)
data
```

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

Multiplying `data` by 10 multiplies each element in `data` by 10, and adding `data` to itself adds each element to itself (i.e., element-wise addition). NumPy arrays must contain homogeneous data types (e.g., all floats or integers) to achieve this common-sense behavior.

```
data * 10
```

```
array([[ 4.9671, -1.3826,  6.4769],
       [15.2303, -2.3415, -2.3414]])
```

```
data_2 = data + data
data_2
```

```
array([[ 0.9934, -0.2765,  1.2954],
       [ 3.0461, -0.4683, -0.4683]])
```

NumPy arrays have attributes. Recall that IPython and Jupyter provide tab completion.

```
data.ndim
```

```
2
```

```
data.shape
```

```
(2, 3)
```

```
data.dtype
```

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

We slice NumPy arrays using `[]`, the same as we slice lists and tuples.

```
data[0]
```

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

We chain `[]`s with arrays, the same as we chain `[]`s with lists and tuples.

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

```
0.4967
```

However, with NumPy arrays, we can replace n chained `[]`s with one pair of `[]`s containing n indexes or slices, separated by commas. For example, `[i][j]` becomes `[i, j]`, and `[i][j][k]` becomes `[i, j, k]`.

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

```
0.4967
```

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

```
np.True_
```

Creating ndarrays

> The easiest way to create an array is to use the array function. This accepts any sequence-like object (including other arrays) and produces a new NumPy array containing the passed data

```
data1 = [6, 7.5, 8, 0, 1]
arr1 = np.array(data1)
arr1
```

```
array([6. , 7.5, 8. , 0. , 1. ])
```

```
arr1.dtype
```

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

Here, `np.array()` implicitly casts the integers in `data1` to floats because NumPy arrays must have homogenous data types. We could explicitly cast all values to integers but would lose information.

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

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

We can cast a list of lists to a two-dimensional NumPy array.

```
data2 = [[1, 2, 3, 4], [5, 6, 7, 8]]
arr2 = np.array(data2)
arr2
```

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

```
arr2.shape
```

```
(2, 4)
```

```
arr2.dtype
```

```
dtype('int64')
```

There are several other ways to create NumPy arrays.

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

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

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

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

```
np.ones_like(arr2)
```

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

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

```
np.array(range(15))
```

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

```
np.arange(15)
```

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

Table 4-1 from McKinney (2022) summarizes NumPy array creation functions.

- `array`: Convert input data (list, tuple, array, or other sequence type) to an ndarray either by inferring a dtype or explicitly specifying a dtype; copies the input data by default
- `asarray`: Convert input to ndarray, but do not copy if the input is already an ndarray
- `arange`: Like the built-in range but returns an ndarray instead of a list
- `ones`, `ones_like`: Produce an array of all 1s with the given shape and dtype; `ones_like` takes another array and produces a `ones` array of the - same shape and dtype
- `zeros`, `zeros_like`: Like `ones` and `ones_like` but producing arrays of 0s instead
- `empty`, `empty_like`: Create new arrays by allocating new memory, but do not populate with any values like ones and zeros
- `full`, `full_like`: Produce an array of the given shape and dtype with all values set to the indicated "fill value"
- `eye`, `identity`: Create a square N-by-N identity matrix (1s on the diagonal and 0s elsewhere)

Arithmetic with NumPy Arrays

> Arrays are important because they enable you to express batch operations on data without writing any for loops. NumPy users call this vectorization. Any arithmetic operations between equal-size arrays applies the operation element-wise

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

```
array([[1., 2., 3.],
       [4., 5., 6.]])
```

NumPy array addition is elementwise.

```
arr + arr
```

```
array([[ 2.,  4.,  6.],
       [ 8., 10., 12.]])
```

NumPy array multiplication is elementwise.

```
arr * arr
```

```
array([[ 1.,  4.,  9.],
       [16., 25., 36.]])
```

NumPy array division is elementwise.

```
1 / arr
```

```
array([[1.    , 0.5   , 0.3333],
       [0.25  , 0.2   , 0.1667]])
```

NumPy powers are elementwise, too.

```
arr ** 2
```

```
array([[ 1.,  4.,  9.],
       [16., 25., 36.]])
```

We can also raise a single value to the elements in an array!

```
2 ** arr
```

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

Basic Indexing and Slicing

We index and slice one-dimensional arrays in the same way as lists and tuples.

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

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

```
arr[5]
```

```
np.int64(5)
```

```
arr[5:8]
```

```
array([5, 6, 7])
```

```
equiv_list = list(range(10))
equiv_list
```

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

```
equiv_list[5:8]
```

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

We must jump through some hoops to replace elements 5, 6, and 7 with the value 12 in the list `equiv_list`.

```
# TypeError: can only assign an iterable
# equiv_list[5:8] = 12
```

```
equiv_list[5:8] = [12] * 3
equiv_list
```

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

However, this operation is easy with NumPy array `arr`!

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

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

We call this behavior "broadcasting".

> As you can see, if you assign a scalar value to a slice, as in `arr[5:8] = 12`, the value is propagated (or broadcasted henceforth) to the entire selection. An important first distinction from Python's built-in lists is that array slices are views on the original array. This means that the data is not copied, and any modifications to the view will be reflected in the source array.

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

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

```
arr_slice[1] = 12345
arr_slice
```

```
array([   12, 12345,    12])
```

```
arr
```

```
array([    0,     1,     2,     3,     4,    12, 12345,    12,     8,
           9])
```

The `:` slices every element in `arr_slice`.

```
arr_slice[:] = 64
arr_slice
```

```
array([64, 64, 64])
```

```
arr
```

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

> If you want a copy of a slice of an ndarray instead of a view, you will need to explicitly copy the array-for example, `arr[5:8].copy()`.

```
arr_slice_2 = arr[5:8].copy()
arr_slice_2
```

```
array([64, 64, 64])
```

```
arr_slice_2[:] = 2_001
arr_slice_2
```

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

```
arr
```

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

Indexing with slices

We can slice across two or more dimensions and use the `[i, j]` notation.

```
arr2d = np.array([[1,2,3], [4,5,6], [7,8,9]])
arr2d
```

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

```
arr2d[:2]
```

```
array([[1, 2, 3],
       [4, 5, 6]])
```

```
arr2d[:2, 1:]
```

```
array([[2, 3],
       [5, 6]])
```

A colon (`:`) by itself selects the entire dimension and is necessary to slice higher dimensions.

```
arr2d[:, :1]
```

```
array([[1],
       [4],
       [7]])
```

```
arr2d[:2, 1:] = 0
arr2d
```

```
array([[1, 0, 0],
       [4, 0, 0],
       [7, 8, 9]])
```

Always check your output!

Boolean Indexing

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

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

```
names
```

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

```
data
```

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

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

```
names == 'Bob'
```

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

```
data[names == 'Bob']
```

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

We can combine Boolean slicing with `:` slicing.

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

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

We can use `~` to invert a Boolean.

```
cond = names == 'Bob'
data[~cond]
```

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

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

```
cond = (names == 'Bob') | (names == 'Will')
data[cond]
```

```
array([[ 0.4967, -0.1383,  0.6477,  1.523 ],
       [-0.4695,  0.5426, -0.4634, -0.4657],
       [ 0.242 , -1.9133, -1.7249, -0.5623],
       [-1.0128,  0.3142, -0.908 , -1.4123]])
```

We can also create a Boolean for each element.

```
data
```

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

```
data < 0
```

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

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

```
array([[0.4967, 0.    , 0.6477, 1.523 ],
       [0.    , 0.    , 1.5792, 0.7674],
       [0.    , 0.5426, 0.    , 0.    ],
       [0.242 , 0.    , 0.    , 0.    ],
       [0.    , 0.3142, 0.    , 0.    ],
       [1.4656, 0.    , 0.0675, 0.    ],
       [0.    , 0.1109, 0.    , 0.3757]])
```

Universal Functions: Fast Element-Wise Array Functions

> A universal function, or ufunc, is a function that performs element-wise operations on data in ndarrays. You can think of them as fast vectorized wrappers for simple functions that take one or more scalar values and produce one or more scalar results.

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

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

```
np.sqrt(arr)
```

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

Like above, we can raise a single value to a NumPy array of powers.

```
2**arr
```

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

`np.exp(x)` is e^x.

```
np.exp(arr)
```

```
array([1.0000e+00, 2.7183e+00, 7.3891e+00, 2.0086e+01, 5.4598e+01,
       1.4841e+02, 4.0343e+02, 1.0966e+03, 2.9810e+03, 8.1031e+03])
```

Table 4-4 from McKinney (2022) summarizes fast, element-wise unary functions:

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

These "unary" functions operate on one array and return a new array with the same shape. There are also "binary" functions that operate on two arrays and return one array.

```
np.random.seed(42)
x = np.random.randn(8)
y = np.random.randn(8)
```

```
x
```

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

```
y
```

```
array([-0.4695,  0.5426, -0.4634, -0.4657,  0.242 , -1.9133, -1.7249,
       -0.5623])
```

```
np.maximum(x, y)
```

```
array([ 0.4967,  0.5426,  0.6477,  1.523 ,  0.242 , -0.2341,  1.5792,
        0.7674])
```

Table 4-5 from McKinney (2022) summarizes fast, element-wise binary functions:

- `add`: Add corresponding elements in arrays
- `subtract`: Subtract elements in second array from first array
- `multiply`: Multiply array elements
- `divide`, `floor_divide`: Divide or floor divide (truncating the remainder)
- `power`: Raise elements in first array to powers indicated in second array
- `maximum`, `fmax`: Element-wise maximum; `fmax` ignores `NaN`
- `minimum`, `fmin`: Element-wise minimum; `fmin` ignores `NaN`
- `mod`: Element-wise modulus (remainder of division)
- `copysign`: Copy sign of values in second argument to values in first argument
- `greater`, `greater_equal`, `less`, `less_equal`, `equal`, `not_equal`: Perform element-wise comparison, yielding boolean array (equivalent to infix operators >, >=, <, <=, ==, !=)
- `logical_and`, `logical_or`, `logical_xor`: Compute element-wise truth value of logical operation (equivalent to infix operators & |, ^)

Array-Oriented Programming with Arrays

> Using NumPy arrays enables you to express many kinds of data processing tasks as concise array expressions that might otherwise require writing loops. This practice of replacing explicit loops with array expressions is commonly referred to as vectorization. In general, vectorized array operations will often be one or two (or more) orders of magnitude faster than their pure Python equivalents, with the biggest impact in any kind of numerical computations. Later, in Appendix A, I explain broadcasting, a powerful method for vectorizing computations.

Expressing Conditional Logic as Array Operations

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

`np.where()` is an if-else statement, like Excel's `if()`.

```
xarr = np.array([1.1, 1.2, 1.3, 1.4, 1.5])
yarr = np.array([2.1, 2.2, 2.3, 2.4, 2.5])
cond = np.array([True, False, True, True, False])
```

```
np.where(cond, xarr, yarr)
```

```
array([1.1, 2.2, 1.3, 1.4, 2.5])
```

We could use a list comprehension instead, but it takes longer to type, read, and troubleshoot.

```
np.array([(x if c else y) for x, y, c in zip(xarr, yarr, cond)])
```

```
array([1.1, 2.2, 1.3, 1.4, 2.5])
```

`np.select()` lets us test more than one condition and has a default value if no condition is met.

```
np.select(
    condlist=[cond==True, cond==False],
    choicelist=[xarr, yarr]
)
```

```
array([1.1, 2.2, 1.3, 1.4, 2.5])
```

Mathematical and Statistical Methods

> A set of mathematical functions that compute statistics about an entire array or about the data along an axis are accessible as methods of the array class. You can use aggregations (often called reductions) like sum, mean, and std (standard deviation) either by calling the array instance method or using the top-level NumPy function.

```
np.random.seed(42)
arr = np.random.randn(5, 4)
arr
```

```
array([[ 0.4967, -0.1383,  0.6477,  1.523 ],
       [-0.2342, -0.2341,  1.5792,  0.7674],
       [-0.4695,  0.5426, -0.4634, -0.4657],
       [ 0.242 , -1.9133, -1.7249, -0.5623],
       [-1.0128,  0.3142, -0.908 , -1.4123]])
```

```
arr.mean()
```

```
-0.1713
```

```
arr.sum()
```

```
-3.4260
```

The aggregation methods above aggregated the whole array. We can use the `axis` argument to aggregate columns (`axis=0`) and rows (`axis=1`).

```
arr.mean(axis=1)
```

```
array([ 0.6323,  0.4696, -0.214 , -0.9896, -0.7547])
```

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

```
0.6323
```

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

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

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

```
arr = np.array([0, 1, 2, 3, 4, 5, 6, 7]) # same output as np.arange(8)
arr.cumsum()
```

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

We can also use the `.cumsum()` method along the axis of a multi-dimensional array.

```
arr = np.array([[0, 1, 2], [3, 4, 5], [6, 7, 8]])
arr
```

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

```
arr.cumsum(axis=0)
```

```
array([[ 0,  1,  2],
       [ 3,  5,  7],
       [ 9, 12, 15]])
```

```
arr.cumprod(axis=1)
```

```
array([[  0,   0,   0],
       [  3,  12,  60],
       [  6,  42, 336]])
```

Table 4-6 from McKinney (2022) summarizes basic statistical methods:

- `sum`: Sum of all the elements in the array or along an axis; zero-length arrays have sum 0
- `mean`: Arithmetic mean; zero-length arrays have NaN mean
- `std`, `var`: Standard deviation and variance, respectively, with optional degrees of freedom adjustment (default denominator n)
- `min`, `max`: Minimum and maximum
- `argmin`, `argmax`: Indices of minimum and maximum elements, respectively
- `cumsum`: Cumulative sum of elements starting from 0
- `cumprod`: Cumulative product of elements starting from 1

McKinney Chapter 4 - Practice - Blank

```
import numpy as np
```

```
%precision 4
```

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

Announcements

Five-Minute Review

Practice

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

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

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

Create a 1-dimensional array `a4` that contains the squares of the even integers through 100,000.

Write a function `calc_pv()` that mimic Excel's `PV` function.

Excel's present value function is: `=PV(rate, nper, pmt, [fv], [type])`

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

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

Write a function `calc_fv()` that mimic Excel's `FV` function.

Excel's future value function is: `=FV(rate, nper, pmt, [pv], [type])`

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

```
np.random.seed(42)
data = np.random.randn(4, 4)
```

Write a function `calc_n()` that calculates the number of payments that generate `x`% of the present value of a perpetuity.

The present value of a growing perpetuity is $PV = \frac{C_1}{r-g}$, and the present value of a growing annuity is $PV = \frac{C_1}{r-g}\left[1 - \left(\frac{1+g}{1+r}\right)^t\right]$.

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

Write a function `calc_returns()` that accepts *NumPy arrays* of prices and dividends and returns a *NumPy array* of returns.

```
prices = np.array([100, 150, 100, 50, 100, 150, 100, 150])
dividends = np.array([1, 1, 1, 1, 2, 2, 2, 2])
```

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

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

Input: `np.array([18.5, 17.0, 18.0, 19.0, 18.0])`
Output: `np.array([0.75, 0.0, 0.5, 1.0, 0.5])`

Write functions `calc_var()` and `calc_std()` that calculate variance and standard deviation.

NumPy's `.var()` and `.std()` methods return *population* statistics (i.e., denominators of n). The pandas equivalents return *sample* statistics (denominators of $n - 1$), which are more appropriate for financial data analysis where we have a sample instead of a population.

Your `calc_var()` and `calc_std()` functions should have a `sample` argument that is `True` by default so both functions return sample statistics by default.

Write a function `calc_ret()` to convert quantitative returns to qualitative returns

Returns within one standard deviation of the mean are "Medium". Returns less than one standard deviation below the mean are "Low", and returns greater than one standard deviation above the mean are "High".

```
np.random.seed(42)
returns = np.random.randn(100)
```

Week 4

McKinney Chapter 5 - Getting Started with pandas

```
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'
```

Introduction

Chapter 5 of McKinney (2022) discusses the fundamentals of pandas, which will be our main tool for the rest of the semester. pandas is an abbreviation of *pan*el *da*ta. Panel data contain observations on multiple entities (e.g., individuals, firms, or countries) over multiple time periods. Panel data combine cross-sectional data (i.e., data across entities at a single point in time) with time-series data (i.e., data over time for a single entity). Panel data are widely used in finance and economics to analyze trends, relationships, and behaviors over time and across entities. We will use pandas every day for the rest of the course!

> pandas will be a major tool of interest throughout much of the rest of the book. It contains data structures and data manipulation tools designed to make data cleaning and analysis fast and easy in Python. pandas is often used in tandem with numerical computing tools like NumPy and SciPy, analytical libraries like statsmodels and scikit-learn, and data visualization libraries like matplotlib. pandas adopts significant parts of NumPy's idiomatic style of array-based computing, especially array-based functions and a preference for data processing without for loops.
>
> While pandas adopts many coding idioms from NumPy, the biggest difference is that pandas is designed for working with tabular or heterogeneous data. NumPy, by contrast, is best suited for working with homogeneous numerical array data.

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

Introduction to pandas Data Structures

> To get started with pandas, you will need to get comfortable with its two workhorse data structures: Series and DataFrame. While they are not a universal solution for every problem, they provide a solid, easy-to-use basis for most applications.

Series

> A Series is a one-dimensional array-like object containing a sequence of values (of similar types to NumPy types) and an associated array of data labels, called its index. The simplest Series is formed from only an array of data.

The early examples use integer and string labels, but date-time labels are most useful.

```
obj = pd.Series([4, 7, -5, 3])
obj
```

```
0    4
1    7
2   -5
3    3
dtype: int64
```

Contrast `obj` with its NumPy array equivalent:

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

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

```
obj.values
```

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

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

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

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

```
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']
```

```
np.int64(-5)
```

```
obj2.loc['a']
```

```
np.int64(-5)
```

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

```
np.int64(-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 is like a NumPy array, and we can use Boolean filters and perform vectorized mathematical operations.

```
obj2 > 0
```

```
d     True
b     True
a    False
c     True
dtype: bool
```

```
obj2[obj2 > 0]
```

```
d    6
b    7
c    3
dtype: int64
```

```
obj2.loc[obj2 > 0]
```

```
d    6
b    7
c    3
dtype: int64
```

```
obj2 * 2
```

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

We can also create a pandas series from a dictionary. The dictionary keys become the series index.

```
sdata = {'Ohio': 35000, 'Texas': 71000, 'Oregon': 16000, 'Utah': 5000}
obj3 = pd.Series(sdata)
obj3
```

```
Ohio      35000
Texas     71000
Oregon    16000
Utah       5000
dtype: int64
```

If we also specify an index with list `states`, pandas will:

1. Respect the index order
2. Keep California because it was in the index
3. Drop Utah because it was not in the index

```
states = ['California', 'Ohio', 'Oregon', 'Texas']
obj4 = pd.Series(sdata, index=states)
obj4
```

```
California          NaN
Ohio          35000.0000
Oregon        16000.0000
Texas         71000.0000
dtype: float64
```

When we perform mathematical operations, pandas aligns series by their indexes. Here `NaN` is "not a number", indicating missing values. `NaN` is a float, so the data type switches from int64 to float64.

```
obj3 + obj4
```

```
California           NaN
Ohio           70000.0000
Oregon         32000.0000
Texas         142000.0000
Utah                 NaN
dtype: float64
```

DataFrame

A pandas data frame is like a worksheet in an Excel workbook with row labels and column names that provide fast indexing.

> A DataFrame represents a rectangular table of data and contains an ordered collection of columns, each of which can be a different value type (numeric, string, boolean, etc.). The DataFrame has both a row and column index; it can be thought of as a dict of Series all sharing the same index. Under the hood, the data is stored as one or more two-dimensional blocks rather than a list, dict, or some other collection of one-dimensional arrays. The exact details of DataFrame's internals are outside the scope of this book.
>
> There are many ways to construct a DataFrame, though one of the most common is from a dict of equal-length lists or NumPy arrays:

```
data = {
    'state': ['Ohio', 'Ohio', 'Ohio', 'Nevada', 'Nevada', 'Nevada'],
    'year': [2000, 2001, 2002, 2001, 2002, 2003],
    'pop': [1.5, 1.7, 3.6, 2.4, 2.9, 3.2]
}
frame = pd.DataFrame(data)

frame
```

	state	year	pop
0	Ohio	2000	1.5000
1	Ohio	2001	1.7000
2	Ohio	2002	3.6000
3	Nevada	2001	2.4000
4	Nevada	2002	2.9000

	state	year	pop
5	Nevada	2003	3.2000

We did not specify an index, so `frame` has the default index of integers starting at 0.

```
frame2 = pd.DataFrame(
    data,
    columns=['year', 'state', 'pop', 'debt'],
    index=['one', 'two', 'three', 'four', 'five', 'six']
)

frame2
```

	year	state	pop	debt
one	2000	Ohio	1.5000	NaN
two	2001	Ohio	1.7000	NaN
three	2002	Ohio	3.6000	NaN
four	2001	Nevada	2.4000	NaN
five	2002	Nevada	2.9000	NaN
six	2003	Nevada	3.2000	NaN

If we extract one column with `df.column` or `df['column']`, we get a series. We can use the `df.colname` or `df['colname']` syntax to *extract* a column from a data frame as a series. ***However, we must use the `df['colname']` syntax to add a column to a data frame.*** Also, we must use the `df['colname']` syntax to extract or add a column whose name contains whitespace.

```
frame2['state']
```

```
one        Ohio
two        Ohio
three      Ohio
four     Nevada
five     Nevada
six      Nevada
Name: state, dtype: object
```

```
frame2.state
```

```
one        Ohio
two        Ohio
three      Ohio
four     Nevada
five     Nevada
six      Nevada
Name: state, dtype: object
```

Data frames have two dimensions, so we must slice data frames more precisely than series.

1. The `.loc[]` method slices by row labels and column names
2. The `.iloc[]` method slices by *integer* row and label indexes

```
frame2.loc['three']
```

```
year      2002
state     Ohio
pop     3.6000
debt       NaN
Name: three, dtype: object
```

```
frame2.iloc[2]
```

```
year      2002
state     Ohio
pop     3.6000
debt       NaN
Name: three, dtype: object
```

We can use NumPy's `[row, column]` syntax within `.loc[]` and `.iloc[]`.

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

```
'Ohio'
```

```
frame2.iloc[2, 1] # row, column
```

```
'Ohio'
```

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

```
state     Ohio
pop     3.6000
Name: three, dtype: object
```

```
frame2.iloc[2, [1, 2]] # row, column
```

```
state     Ohio
pop     3.6000
Name: three, dtype: object
```

We can assign either scalars or arrays to data frame columns.

1. Scalars will broadcast to every row in the data frame
2. Arrays must have the same length as the column

```
frame2['debt'] = 16.5
frame2
```

	year	state	pop	debt
one	2000	Ohio	1.5000	16.5000
two	2001	Ohio	1.7000	16.5000
three	2002	Ohio	3.6000	16.5000
four	2001	Nevada	2.4000	16.5000
five	2002	Nevada	2.9000	16.5000
six	2003	Nevada	3.2000	16.5000

```
frame2['debt'] = np.arange(6.)
frame2
```

	year	state	pop	debt
one	2000	Ohio	1.5000	0.0000
two	2001	Ohio	1.7000	1.0000
three	2002	Ohio	3.6000	2.0000
four	2001	Nevada	2.4000	3.0000
five	2002	Nevada	2.9000	4.0000
six	2003	Nevada	3.2000	5.0000

If we assign a series to a data frame column, pandas will use the index to align it with the data frame. Data frame rows not in the series will be `NaN`.

```
val = pd.Series([-1.2, -1.5, -1.7], index=['two', 'four', 'five'])
val
```

```
two    -1.2000
four   -1.5000
five   -1.7000
dtype: float64
```

```
frame2['debt'] = val
frame2
```

	year	state	pop	debt
one	2000	Ohio	1.5000	NaN
two	2001	Ohio	1.7000	-1.2000
three	2002	Ohio	3.6000	NaN
four	2001	Nevada	2.4000	-1.5000
five	2002	Nevada	2.9000	-1.7000
six	2003	Nevada	3.2000	NaN

We can add columns to our data frame, then delete them with `del`.

```
frame2['eastern'] = (frame2.state == 'Ohio')
frame2
```

	year	state	pop	debt	eastern
one	2000	Ohio	1.5000	NaN	True

	year	state	pop	debt	eastern
two	2001	Ohio	1.7000	-1.2000	True
three	2002	Ohio	3.6000	NaN	True
four	2001	Nevada	2.4000	-1.5000	False
five	2002	Nevada	2.9000	-1.7000	False
six	2003	Nevada	3.2000	NaN	False

```
del frame2['eastern']
frame2
```

	year	state	pop	debt
one	2000	Ohio	1.5000	NaN
two	2001	Ohio	1.7000	-1.2000
three	2002	Ohio	3.6000	NaN
four	2001	Nevada	2.4000	-1.5000
five	2002	Nevada	2.9000	-1.7000
six	2003	Nevada	3.2000	NaN

Index Objects

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

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

Index objects are immutable!

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

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

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

```
Index(['foo', 'foo', 'bar', 'bar'], dtype='object')
```

Essential Functionality

This section provides the most common pandas operations. It is difficult to provide an exhaustive reference, but this section introduces the most common operations.

Indexing, Selection, and Filtering

Indexing, selecting, and filtering will be among our most-used pandas features.

```
obj = pd.Series(np.arange(4.), index=['a', 'b', 'c', 'd'])
obj
```

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

```
obj['b']
```

```
1.0000
```

```
obj.loc['b']
```

```
1.0000
```

```
obj.iloc[1]
```

```
1.0000
```

```
obj.iloc[1:3]
```

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

When we slice with labels, the left and right endpoints are inclusive.

```
obj['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: int64
```

Indexing columns with a list returns a data frame.

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

	three
Ohio	2
Colorado	6
Utah	10
New York	14

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

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

Table 5-4 summarizes data frame indexing and slicing options:

- `df[val]`: Select single column or sequence of columns from the DataFrame; special case conveniences: boolean array (filter rows), slice (slice rows), or boolean DataFrame (set values based on some criterion)
- `df.loc[val]`: Selects single row or subset of rows from the DataFrame by label
- `df.loc[:, val]`: Selects single column or subset of columns by label
- `df.loc[val1, val2]`: Select both rows and columns by label
- `df.iloc[where]`: Selects single row or subset of rows from the DataFrame by integer position
- `df.iloc[:, where]`: Selects single column or subset of columns by integer position
- `df.iloc[where_i, where_j]`: Select both rows and columns by integer position
- `df.at[label_i, label_j]`: Select a single scalar value by row and column label
- `df.iat[i, j]`: Select a single scalar value by row and column position (integers) reindex method Select either rows or columns by labels
- `get_value`, `set_value` methods: Select single value by row and column label

pandas is powerful and these options can be overwhelming! We will typically use `df[val]` to select columns (here `val` is either a string or list of strings), `df.loc[val]` to select rows (here `val` is a row label), and `df.loc[val1, val2]` to select both rows and columns. The other options add flexibility, and we may occasionally use them. However, our data will be large enough that counting row and column number will be tedious, making `.iloc[]` impractical.

Arithmetic and Data Alignment

An important pandas feature for some applications is the behavior of arithmetic between objects with different indexes. When you are adding together objects, if any index pairs are not the same, the respective index in the result will be the union of the index pairs. For users with database experience, this is similar to an automatic outer join on the index labels.

```
s1 = pd.Series(
    data=[7.3, -2.5, 3.4, 1.5],
    index=['a', 'c', 'd', 'e']
)
s2 = pd.Series(
    data=[-2.1, 3.6, -1.5, 4, 3.1],
    index=['a', 'c', 'e', 'f', 'g']
)
```

```
s1
```

```
a    7.3000
c   -2.5000
d    3.4000
e    1.5000
dtype: float64
```

```
s2
```

```
a   -2.1000
c    3.6000
e   -1.5000
f    4.0000
g    3.1000
dtype: float64
```

```
s1 + s2
```

```
a   5.2000
c   1.1000
d      NaN
e   0.0000
f      NaN
```

```
g      NaN
dtype: float64
```

```
df1 = pd.DataFrame(
    data=np.arange(9.).reshape((3, 3)),
    columns=list('bcd'),
    index=['Ohio', 'Texas', 'Colorado']
)
df2 = pd.DataFrame(
    data=np.arange(12.).reshape((4, 3)),
    columns=list('bde'),
    index=['Utah', 'Ohio', 'Texas', 'Oregon']
)
```

```
df1
```

	b	c	d
Ohio	0.0000	1.0000	2.0000
Texas	3.0000	4.0000	5.0000
Colorado	6.0000	7.0000	8.0000

```
df2
```

	b	d	e
Utah	0.0000	1.0000	2.0000
Ohio	3.0000	4.0000	5.0000
Texas	6.0000	7.0000	8.0000
Oregon	9.0000	10.0000	11.0000

```
df1 + df2
```

	b	c	d	e
Colorado	NaN	NaN	NaN	NaN
Ohio	3.0000	NaN	6.0000	NaN
Oregon	NaN	NaN	NaN	NaN
Texas	9.0000	NaN	12.0000	NaN
Utah	NaN	NaN	NaN	NaN

Always check your output!

Arithmetic methods with fill values

```
df1 = pd.DataFrame(
    data=np.arange(12.).reshape((3, 4)),
    columns=list('abcd')
)
df2 = pd.DataFrame(
    data=np.arange(20.).reshape((4, 5)),
    columns=list('abcde')
)
df2.loc[1, 'b'] = np.nan
```

```
df1
```

	a	b	c	d
0	0.0000	1.0000	2.0000	3.0000
1	4.0000	5.0000	6.0000	7.0000
2	8.0000	9.0000	10.0000	11.0000

```
df2
```

	a	b	c	d	e
0	0.0000	1.0000	2.0000	3.0000	4.0000
1	5.0000	NaN	7.0000	8.0000	9.0000
2	10.0000	11.0000	12.0000	13.0000	14.0000
3	15.0000	16.0000	17.0000	18.0000	19.0000

```
df1 + df2
```

	a	b	c	d	e
0	0.0000	2.0000	4.0000	6.0000	NaN
1	9.0000	NaN	13.0000	15.0000	NaN
2	18.0000	20.0000	22.0000	24.0000	NaN
3	NaN	NaN	NaN	NaN	NaN

We can specify a fill value for `NaN` values. pandas fills would-be `NaN` values in each data frame *before* the arithmetic operation.

```
df1.add(df2, fill_value=0)
```

	a	b	c	d	e
0	0.0000	2.0000	4.0000	6.0000	4.0000
1	9.0000	5.0000	13.0000	15.0000	9.0000
2	18.0000	20.0000	22.0000	24.0000	14.0000
3	15.0000	16.0000	17.0000	18.0000	19.0000

Operations between DataFrame and Series

```
arr = np.arange(12.).reshape((3, 4))
arr
```

```
array([[ 0.,  1.,  2.,  3.],
       [ 4.,  5.,  6.,  7.],
       [ 8.,  9., 10., 11.]])
```

```
arr[0]
```

```
array([0., 1., 2., 3.])
```

```
arr - arr[0]
```

```
array([[0., 0., 0., 0.],
       [4., 4., 4., 4.],
       [8., 8., 8., 8.]])
```

Arithmetic operations between series and data frames behave the same as in the example above.

```
frame = pd.DataFrame(
    data=np.arange(12.).reshape((4, 3)),
    columns=list('bde'),
    index=['Utah', 'Ohio', 'Texas', 'Oregon']
)

series = frame.iloc[0]
```

```
frame
```

	b	d	e
Utah	0.0000	1.0000	2.0000
Ohio	3.0000	4.0000	5.0000
Texas	6.0000	7.0000	8.0000
Oregon	9.0000	10.0000	11.0000

```
series
```

```
b   0.0000
d   1.0000
e   2.0000
Name: Utah, dtype: float64
```

```
frame - series
```

	b	d	e
Utah	0.0000	0.0000	0.0000
Ohio	3.0000	3.0000	3.0000
Texas	6.0000	6.0000	6.0000
Oregon	9.0000	9.0000	9.0000

```
series2 = pd.Series(data=range(3), index=['b', 'e', 'f'])
```

```
frame
```

	b	d	e
Utah	0.0000	1.0000	2.0000
Ohio	3.0000	4.0000	5.0000
Texas	6.0000	7.0000	8.0000
Oregon	9.0000	10.0000	11.0000

```
series2
```

```
b    0
e    1
f    2
dtype: int64
```

```
frame + series2
```

	b	d	e	f
Utah	0.0000	NaN	3.0000	NaN
Ohio	3.0000	NaN	6.0000	NaN
Texas	6.0000	NaN	9.0000	NaN
Oregon	9.0000	NaN	12.0000	NaN

pandas has a `.sub()` method that lets us chain operations, but we might need the `axis` argument to get the result we want!

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

```
frame
```

	b	d	e
Utah	0.0000	1.0000	2.0000
Ohio	3.0000	4.0000	5.0000
Texas	6.0000	7.0000	8.0000
Oregon	9.0000	10.0000	11.0000

```
series3
```

```
Utah      1.0000
Ohio      4.0000
Texas     7.0000
Oregon   10.0000
Name: d, dtype: float64
```

```
frame.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

Function Application and Mapping

```
np.random.seed(42)
frame = pd.DataFrame(
    data=np.random.randn(4, 3),
    columns=list('bde'),
    index=['Utah', 'Ohio', 'Texas', 'Oregon']
)

frame
```

	b	d	e
Utah	0.4967	-0.1383	0.6477
Ohio	1.5230	-0.2342	-0.2341
Texas	1.5792	0.7674	-0.4695
Oregon	0.5426	-0.4634	-0.4657

```
frame.abs()
```

	b	d	e
Utah	0.4967	0.1383	0.6477
Ohio	1.5230	0.2342	0.2341
Texas	1.5792	0.7674	0.4695
Oregon	0.5426	0.4634	0.4657

Another frequent operation is applying a function on one-dimensional arrays to each column or row. DataFrame's apply method does exactly this:

```
frame.apply(lambda x: x.max() - x.min()) # implied axis=0
```

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

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

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

However, under the hood, the `.apply()` method is a `for` loop and slower than built-in methods.

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

```
8.3  s ± 232 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
```

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

```
15.4  s ± 231 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
```

Summarizing and Computing Descriptive Statistics

```
df = pd.DataFrame(
    [[1.4, np.nan], [7.1, -4.5], [np.nan, np.nan], [0.75, -1.3]],
    index=['a', 'b', 'c', 'd'],
    columns=['one', 'two']
)

df
```

	one	two
a	1.4000	NaN

	one	two
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
```

Correlation and Covariance

```
import yfinance as yf
```

```
data = yf.download('AAPL IBM MSFT GOOG')
```

```
[**********************100%**********************]  4 of 4 completed
[**********************50%                       ]  2 of 4 completed[**********************75
```

```
data['Adj Close'].tail()
```

Ticker Date	AAPL	GOOG	IBM	MSFT
2024-12-27	255.5900	194.0400	222.7800	430.5300
2024-12-30	252.2000	192.6900	220.2500	424.8300
2024-12-31	250.4200	190.4400	219.8300	421.5000
2025-01-02	243.8500	190.6300	219.9400	418.5800
2025-01-03	NaN	193.6100	222.1800	423.5300

Data frame `data` contains daily prices and volume for AAPL, IBM, MSFT, and GOOG. The `Adj Close` columns are reverse-engineered daily closing prices that account for dividends and stock splits (and reverse splits). As a result, the `.pct_change()` of `Adj Close` correctly considers dividends and price changes, so $r_t = \frac{(P_t + D_t) - P_{t-1}}{P_{t-1}} = \frac{\text{Adj Close}_t - \text{Adj Close}_{t-1}}{\text{Adj Close}_{t-1}}$.

```
returns = data['Adj Close'].iloc[:-1].pct_change().dropna()
returns
```

Ticker 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
...	...	...	...	...
2024-12-26	0.0032	-0.0024	0.0021	-0.0028
2024-12-27	-0.0132	-0.0155	-0.0094	-0.0173
2024-12-30	-0.0133	-0.0070	-0.0114	-0.0132
2024-12-31	-0.0071	-0.0117	-0.0019	-0.0078
2025-01-02	-0.0262	0.0010	0.0005	-0.0069

We multiply by 252 to annualize mean daily returns because means grow linearly with time and there are (about) 252 trading days per year.

```
returns.mean().mul(252)
```

```
Ticker
AAPL   0.3606
GOOG   0.2598
IBM    0.1040
```

```
MSFT   0.1941
dtype: float64
```

We multiply by $\sqrt{252}$ to annualize the volatility of daily returns because standard deviation is the square root of variance, variances grow linearly with time, and there are (about) 252 trading days per year. Ivo Welch explains this calculation at the bottom of Page 7 of Chapter 8 his free corporate finance textbook.

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

```
Ticker
AAPL   0.3237
GOOG   0.3060
IBM    0.2267
MSFT   0.2694
dtype: float64
```

We can calculate pairwise correlations.

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

```
0.4855
```

We can also calculate correlation matrices.

```
returns.corr()
```

Ticker Ticker	AAPL	GOOG	IBM	MSFT
AAPL	1.0000	0.5133	0.4173	0.5222
GOOG	0.5133	1.0000	0.3821	0.5623
IBM	0.4173	0.3821	1.0000	0.4855
MSFT	0.5222	0.5623	0.4855	1.0000

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

```
0.4855
```

```
np.allclose(
    a=returns['MSFT'].corr(returns['IBM']),
    b=returns.corr().loc['MSFT', 'IBM']
)
```

```
True
```

McKinney Chapter 5 - Practice - Blank

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

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

Announcements

Five-Minute Review

Practice

What are the mean daily returns for these four stocks?

```
tickers = 'AAPL IBM MSFT GOOG'
```

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

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

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

Repeat the previous calculations and plot for the stocks in the Dow-Jones Industrial Index (DJIA)

We can find the current DJIA stocks on Wikipedia. We must download new data, into `tickers_2`, `data_2`, and `returns_2`.

Calculate total returns for the stocks in the DJIA

Plot the distribution of total returns for the stocks in the DJIA

Which stocks have the minimum and maximum total returns?

Plot the cumulative returns for the stocks in the DJIA

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

Week 5

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

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import pandas_datareader as pdr
import yfinance as yf
```

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

Introduction

Chapter 8 of McKinney (2022) introduces a few important pandas concepts:

1. Joining or merging is combining 2+ data frames on 1+ indexes or columns into 1 data frame
2. Reshaping is rearranging a data frame so it has fewer columns and more rows (wide to long) or more columns and fewer rows (long to wide)

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

Hierarchical Indexing

We must learn about hierarchical indexing before we learn about combining and reshaping data. A hierarchical index has two or more levels. For example, we could index rows by ticker and date. Or we could index columns by variable and ticker. Hierarchical indexing helps us work with high-dimensional data in a low-dimensional form.

```
np.random.seed(42)
data = pd.Series(
    data=np.random.randn(9),
    index=[
        ['a', 'a', 'a', 'b', 'b', 'c', 'c', 'd', 'd'],
        [1, 2, 3, 1, 3, 1, 2, 2, 3]
    ]
)

data
```

```
a  1    0.4967
   2   -0.1383
   3    0.6477
b  1    1.5230
   3   -0.2342
c  1   -0.2341
   2    1.5792
d  2    0.7674
   3   -0.4695
dtype: float64
```

We can index this series to subset it.

```
data['b']
```

```
1    1.5230
3   -0.2342
dtype: float64
```

```
data.loc['b']
```

```
1    1.5230
3   -0.2342
dtype: float64
```

```
data['b':'c']
```

```
b  1    1.5230
   3   -0.2342
c  1   -0.2341
   2    1.5792
dtype: float64
```

```
data.loc[['b', '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 `:` slices all values in the outer index, and the `2` slices the three values with `2` indexes.

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

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

Here, `data` has a stacked or long format. We have multiple observations for each outer index (letters) with different inner indexes (numbers). We can un-stack `data` to convert the inner index level to columns. New, we have an unstacked or wide format.

```
data.unstack()
```

	1	2	3
a	0.4967	-0.1383	0.6477
b	1.5230	NaN	-0.2342
c	-0.2341	1.5792	NaN
d	NaN	0.7674	-0.4695

We can create a data frame with hierarchical indexes or multi-indexes on rows *and* columns.

```
frame = pd.DataFrame(
    data=np.arange(12).reshape((4, 3)),
    index=[['a', 'a', 'b', 'b'], [1, 2, 1, 2]],
    columns=[['Ohio', 'Ohio', 'Colorado'], ['Green', 'Red', 'Green']]
)

frame
```

		Ohio	
		Green	Re
a	1	0	1
	2	3	4
b	1	6	7
	2	9	10

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

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

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

Recall that `df[val]` selects the `val` column. Here, `frame` has a multi-index for the columns, so `frame['Ohio']` selects all columns with Ohio as the outer index.

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

	color	Green
key1	key2	
a	1	0

key1	color key2	Green
b	2	3
	1	6
	2	9

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

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

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

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

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

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

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

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

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

Reordering and Sorting Levels

We can swap index levels with the `.swaplevel()` method. The default arguments are `i=-2` and `j=-1`, which swap the two innermost index levels.

```
frame
```

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

```
frame.swaplevel()
```

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

We can use index *names*, too.

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

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

Indexing with a DataFrame's columns

We can convert a column into an index and an index into a column with the `.set_index()` and `.reset_index()` methods.

```
frame = pd.DataFrame({
    'a': range(7),
    'b': range(7, 0, -1),
    'c': ['one', 'one', 'one', 'two', 'two','two', 'two'],
    'd': [0, 1, 2, 0, 1, 2, 3]
})

frame
```

	a	b	c	d
0	0	7	one	0
1	1	6	one	1
2	2	5	one	2
3	3	4	two	0
4	4	3	two	1
5	5	2	two	2
6	6	1	two	3

The `.set_index()` method converts columns to indexes and drops these columns by default.

```
frame2 = frame.set_index(['c', 'd'])
frame2
```

c	d	a	b
one	0	0	7
	1	1	6
	2	2	5
two	0	3	4
	1	4	3
	2	5	2
	3	6	1

The `.reset_index()` method drops indexes, adds them as columns by default, and sets an integer index.

```
frame2.reset_index()
```

	c	d	a	b
0	one	0	0	7
1	one	1	1	6
2	one	2	2	5
3	two	0	3	4
4	two	1	4	3
5	two	2	5	2
6	two	3	6	1

Combining and Merging Datasets

pandas provides several methods and functions to combine and merge data. We can typically create the same output with several these methods or functions, but one may be more efficient.

When we want to combine data frames with similar indexes, we will tend to use the `.join()` method. The `.join()` can also combine three or more data frames.

Otherwise, we will use the `.merge()` method or `pd.merge()` function. The `pd.merge()` function is more flexible than the `.join()` method, so we will start with the `pd.merge()` function.

The pandas website provides helpful visualizations.

Database-Style DataFrame Joins

Merge or join operations combine datasets by linking rows using one or more keys. These operations are central to relational databases (e.g., SQL-based). The merge function in pandas is the main entry point for using these algorithms on your data.

```
df1 = pd.DataFrame({'key': ['b', 'b', 'a', 'c', 'a', 'a', 'b'], 'data1': range(7)})
df2 = pd.DataFrame({'key': ['a', 'b', 'd'], 'data2': range(3)})
```

```
df1
```

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

```
df2
```

	key	data2
0	a	0
1	b	1
2	d	2

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

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

The default is `how='inner'`, so `pd.merge()` inner joins left and right data frames by default, keeping only rows that appear in both. We can specify `how='outer'`, so `pd.merge()` outer joins left and right data frames, keeping all rows that appear in either.

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

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

A `how='left'` merge keeps only rows that appear in the left data frame.

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

	key	data1	data2
0	b	0	1.0000
1	b	1	1.0000
2	a	2	0.0000
3	c	3	NaN
4	a	4	0.0000
5	a	5	0.0000
6	b	6	1.0000

A `how='right'` merge keeps only rows that appear in the right data frame.

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

	key	data1	data2
0	a	2.0000	0
1	a	4.0000	0
2	a	5.0000	0

	key	data1	data2
3	b	0.0000	1
4	b	1.0000	1
5	b	6.0000	1
6	d	NaN	2

By default, `pd.merge()` merges on any columns that appear in both data frames.

> `on` : label or list Column or index level names to join on. These must be found in both DataFrames. If `on` is None and not merging on indexes then this defaults to the intersection of the columns in both DataFrames.

Here, `key` is the only common column between `df1` and `df2`. We *should* specify `on='key'` to avoid unexpected results.

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

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

We *must* specify `left_on` and `right_on` if our left and right data frames do not have a common column.

```
df3 = pd.DataFrame({'lkey': ['b', 'b', 'a', 'c', 'a', 'a', 'b'], 'data1': range(7)})
df4 = pd.DataFrame({'rkey': ['a', 'b', 'd'], 'data2': range(3)})
```

```
df3
```

	lkey	data1
0	b	0
1	b	1
2	a	2
3	c	3

	lkey	data1
4	a	4
5	a	5
6	b	6

```
df4
```

	rkey	data2
0	a	0
1	b	1
2	d	2

```
# pd.merge(df3, df4) # this code fails/errors because there are not common columns
# MergeError: No common columns to perform merge on. Merge options: left_on=None, right_on=N
```

```
pd.merge(df3, df4, left_on='lkey', right_on='rkey')
```

	lkey	data1	rkey	data2
0	b	0	b	1
1	b	1	b	1
2	a	2	a	0
3	a	4	a	0
4	a	5	a	0
5	b	6	b	1

Here, `pd.merge()` drops row `c` from `df3` and row `d` from `df4` because `pd.merge()` *inner* joins by default. An inner join keeps the intersection of the left and right data frame keys. If we want to keep rows `c` and `d`, we can *outer* join `df3` and `df4` with `how='outer'`.

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

	key	data1	data2
0	a	2.0000	0.0000
1	a	4.0000	0.0000
2	a	5.0000	0.0000

	key	data1	data2
3	b	0.0000	1.0000
4	b	1.0000	1.0000
5	b	6.0000	1.0000
6	c	3.0000	NaN
7	d	NaN	2.0000

Many-to-many merges have well-defined, though not necessarily intuitive, behavior.

```
df1 = pd.DataFrame({'key': ['b', 'b', 'a', 'c', 'a', 'b'], 'data1': range(6)})
df2 = pd.DataFrame({'key': ['a', 'b', 'a', 'b', 'd'], 'data2': range(5)})
```

```
df1
```

	key	data1
0	b	0
1	b	1
2	a	2
3	c	3
4	a	4
5	b	5

```
df2
```

	key	data2
0	a	0
1	b	1
2	a	2
3	b	3
4	d	4

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

	key	data1	data2
0	b	0	1
1	b	0	3

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

> Many-to-many joins form the Cartesian product of the rows. Since there were three `b` rows in the left DataFrame and two in the right one, there are six `b` rows in the result. The join method only affects the distinct key values appearing in the result.

Be careful with many-to-many joins! In finance, we do not expect many-to-many joins because we expect at least one of the data frames to have unique observations. *pandas will not warn us if we accidentally perform a many-to-many join instead of a one-to-one or many-to-one join.*

We can merge on more than one key. For example, we can merge two data sets on ticker-date pairs or industry-date pairs.

```
left = pd.DataFrame({'key1': ['foo', 'foo', 'bar'],
                     'key2': ['one', 'two', 'one'],
                     'lval': [1, 2, 3]})
right = pd.DataFrame({'key1': ['foo', 'foo', 'bar', 'bar'],
                      'key2': ['one', 'one', 'one', 'two'],
                      'rval': [4, 5, 6, 7]})
```

```
left
```

	key1	key2	lval
0	foo	one	1
1	foo	two	2
2	bar	one	3

```
right
```

	key1	key2	rval
0	foo	one	4
1	foo	one	5
2	bar	one	6
3	bar	two	7

```
pd.merge(left, right, on=['key1', 'key2'], how='outer')
```

	key1	key2	lval	rval
0	bar	one	3.0000	6.0000
1	bar	two	NaN	7.0000
2	foo	one	1.0000	4.0000
3	foo	one	1.0000	5.0000
4	foo	two	2.0000	NaN

When column names overlap between the left and right data frames, `pd.merge()` appends `_x` and `_y` to the left and right versions of the overlapping column names.

```
pd.merge(left, right, on='key1')
```

	key1	key2_x	lval	key2_y	rval
0	foo	one	1	one	4
1	foo	one	1	one	5
2	foo	two	2	one	4
3	foo	two	2	one	5
4	bar	one	3	one	6
5	bar	one	3	two	7

I typically specify the `suffixes` argument to avoid confusion.

```
pd.merge(left, right, on='key1', suffixes=('_left', '_right'))
```

	key1	key2_left	lval	key2_right	rval
0	foo	one	1	one	4
1	foo	one	1	one	5

	key1	key2_left	lval	key2_right	rval
2	foo	two	2	one	4
3	foo	two	2	one	5
4	bar	one	3	one	6
5	bar	one	3	two	7

I read the `pd.merge()` docstring frequently! ***Table 8-2*** summarizes the commonly used arguments for `pd.merge()`.

- `left`: DataFrame to be merged on the left side.
- `right`: DataFrame to be merged on the right side.
- `how`: One of 'inner', 'outer', 'left', or 'right'; defaults to 'inner'.
- `on`: Column names to join on. Must be found in both DataFrame objects. If not specified and no other join keys given will use the intersection of the column names in left and right as the join keys.
- `left_on`: Columns in left DataFrame to use as join keys.
- `right_on`: Analogous to left_on for left DataFrame.
- `left_index`: Use row index in left as its join key (or keys, if a MultiIndex).
- `right_index`: Analogous to left_index.
- `sort`: Sort merged data lexicographically by join keys; True by default (disable to get better performance in some cases on large datasets).
- `suffixes`: Tuple of string values to append to column names in case of overlap; defaults to ('_x', '_y') (e.g., if 'data' in both DataFrame objects, would appear as 'data_x' and 'data_y' in result).
- `copy`: If False, avoid copying data into resulting data structure in some exceptional cases; by default always copies.
- `indicator`: Adds a special column _merge that indicates the source of each row; values will be 'left_only', 'right_only', or 'both' based on the origin of the joined data in each row.

Merging on Index

If we want to use `pd.merge()` to join on row indexes, we can use the `left_index` and `right_index` arguments.

```
left1 = pd.DataFrame({'key': ['a', 'b', 'a', 'a', 'b', 'c'], 'value': range(6)})
right1 = pd.DataFrame({'group_val': [3.5, 7]}, index=['a', 'b'])
```

```
left1
```

	key	value
0	a	0
1	b	1
2	a	2
3	a	3
4	b	4
5	c	5

```
right1
```

	group_val
a	3.5000
b	7.0000

```
pd.merge(left1, right1, left_on='key', right_index=True, how='outer')
```

	key	value	group_val
0	a	0	3.5000
2	a	2	3.5000
3	a	3	3.5000
1	b	1	7.0000
4	b	4	7.0000
5	c	5	NaN

The index arguments work for hierarchical indexes (multi indexes), too.

```
lefth = pd.DataFrame({'key1': ['Ohio', 'Ohio', 'Ohio', 'Nevada', 'Nevada'],
                      'key2': [2000, 2001, 2002, 2001, 2002],
                      'data': np.arange(5.)})
righth = pd.DataFrame(np.arange(12).reshape((6, 2)),
                      index=[['Nevada', 'Nevada', 'Ohio', 'Ohio', 'Ohio', 'Ohio'],
                             [2001, 2000, 2000, 2000, 2001, 2002]],
                      columns=['event1', 'event2'])
```

```
pd.merge(lefth, righth, left_on=['key1', 'key2'], right_index=True, how='outer')
```

	key1	key2	data	event1	event2
4	Nevada	2000	NaN	2.0000	3.0000
3	Nevada	2001	3.0000	0.0000	1.0000
4	Nevada	2002	4.0000	NaN	NaN
0	Ohio	2000	0.0000	4.0000	5.0000
0	Ohio	2000	0.0000	6.0000	7.0000
1	Ohio	2001	1.0000	8.0000	9.0000
2	Ohio	2002	2.0000	10.0000	11.0000

```
left2 = pd.DataFrame([[1., 2.], [3., 4.], [5., 6.]],
                     index=['a', 'c', 'e'],
                     columns=['Ohio', 'Nevada'])
right2 = pd.DataFrame([[7., 8.], [9., 10.], [11., 12.], [13, 14]],
                      index=['b', 'c', 'd', 'e'],
                      columns=['Missouri', 'Alabama'])
```

If we use both indexes, `pd.merge()` will keep the index.

```
pd.merge(left2, right2, how='outer', left_index=True, right_index=True)
```

	Ohio	Nevada	Missouri	Alabama
a	1.0000	2.0000	NaN	NaN
b	NaN	NaN	7.0000	8.0000
c	3.0000	4.0000	9.0000	10.0000
d	NaN	NaN	11.0000	12.0000
e	5.0000	6.0000	13.0000	14.0000

> DataFrame has a convenient join instance for merging by index. It can also be used to combine together many DataFrame objects having the same or similar indexes but non-overlapping columns.

We can use the `.join()` method if both data frames have similar indexes.

```
left2
```

	Ohio	Nevada
a	1.0000	2.0000
c	3.0000	4.0000

	Ohio	Nevada
e	5.0000	6.0000

```
right2
```

	Missouri	Alabama
b	7.0000	8.0000
c	9.0000	10.0000
d	11.0000	12.0000
e	13.0000	14.0000

```
left2.join(right2, how='outer')
```

	Ohio	Nevada	Missouri	Alabama
a	1.0000	2.0000	NaN	NaN
b	NaN	NaN	7.0000	8.0000
c	3.0000	4.0000	9.0000	10.0000
d	NaN	NaN	11.0000	12.0000
e	5.0000	6.0000	13.0000	14.0000

The `.join()` method left joins by default. Because the `.join()` method uses indexes, it requires fewer arguments than `.merge()`. The `.join()` method can also accept a list of data frames.

```
another = pd.DataFrame(
    data=[[7., 8.], [9., 10.], [11., 12.], [16., 17.]],
    index=['a', 'c', 'e', 'f'],
    columns=['New York', 'Oregon']
)

another
```

	New York	Oregon
a	7.0000	8.0000
c	9.0000	10.0000
e	11.0000	12.0000

	New York	Oregon
f	16.0000	17.0000

```
left2.join([right2, another])
```

	Ohio	Nevada	Missouri	Alabama	New York	Oregon
a	1.0000	2.0000	NaN	NaN	7.0000	8.0000
c	3.0000	4.0000	9.0000	10.0000	9.0000	10.0000
e	5.0000	6.0000	13.0000	14.0000	11.0000	12.0000

```
left2.join([right2, another], how='outer')
```

	Ohio	Nevada	Missouri	Alabama	New York	Oregon
a	1.0000	2.0000	NaN	NaN	7.0000	8.0000
c	3.0000	4.0000	9.0000	10.0000	9.0000	10.0000
e	5.0000	6.0000	13.0000	14.0000	11.0000	12.0000
b	NaN	NaN	7.0000	8.0000	NaN	NaN
d	NaN	NaN	11.0000	12.0000	NaN	NaN
f	NaN	NaN	NaN	NaN	16.0000	17.0000

Concatenating Along an Axis

The `pd.concat()` function provides a flexible way to combine data frames and series along an axis. I typically use `pd.concat()` to combine:

1. A list of data frames with similar layouts
2. A list of series because series do not have `.join()` or `.merge()` methods

```
s1 = pd.Series([0, 1], index=['a', 'b'])
s2 = pd.Series([2, 3, 4], index=['c', 'd', 'e'])
s3 = pd.Series([5, 6], index=['f', 'g'])
```

```
pd.concat([s1, s2, s3]) # implicit axis=0
```

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

```
pd.concat([s1, s2, s3], axis=1) # explicit axis=1
```

	0	1	2
a	0.0000	NaN	NaN
b	1.0000	NaN	NaN
c	NaN	2.0000	NaN
d	NaN	3.0000	NaN
e	NaN	4.0000	NaN
f	NaN	NaN	5.0000
g	NaN	NaN	6.0000

```
result = pd.concat([s1, s2, s3], keys=['one', 'two', 'three']) # implicit axis=0
result
```

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

```
result.unstack(level=0)
```

	one	two	three
a	0.0000	NaN	NaN
b	1.0000	NaN	NaN
c	NaN	2.0000	NaN

	one	two	three
d	NaN	3.0000	NaN
e	NaN	4.0000	NaN
f	NaN	NaN	5.0000
g	NaN	NaN	6.0000

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

	one	two	three
a	0.0000	NaN	NaN
b	1.0000	NaN	NaN
c	NaN	2.0000	NaN
d	NaN	3.0000	NaN
e	NaN	4.0000	NaN
f	NaN	NaN	5.0000
g	NaN	NaN	6.0000

```
df1 = pd.DataFrame(
    data=np.arange(6).reshape(3, 2),
    index=['a', 'b', 'c'],
    columns=['one', 'two']
)
df2 = pd.DataFrame(
    data=5 + np.arange(4).reshape(2, 2),
    index=['a', 'c'],
    columns=['three', 'four']
)
```

```
pd.concat([df1, df2], axis=1, keys=['level1', 'level2'])
```

	level1		level2	
	one	two	three	four
a	0	1	5.0000	6.0000
b	2	3	NaN	NaN
c	4	5	7.0000	8.0000

```
pd.concat([df1, df2], axis=1, keys=['level1', 'level2'], names=['upper', 'lower'])
```

upper	level1		level2	
lower	one	two	three	four
a	0	1	5.0000	6.0000
b	2	3	NaN	NaN
c	4	5	7.0000	8.0000

Reshaping and Pivoting

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

Reshaping with Hierarchical Indexing

Hierarchical indexes (multi-indexes) help reshape data.

> There are two primary actions: - stack: This "rotates" or pivots from the columns in the data to the rows - unstack: This pivots from the rows into the columns

```
data = pd.DataFrame(np.arange(6).reshape((2, 3)),
                    index=pd.Index(['Ohio', 'Colorado'], name='state'),
                    columns=pd.Index(['one', 'two', 'three'],
                    name='number'))

data
```

number	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: int64
```

```
result.unstack()
```

number state	one	two	three
Ohio	0	1	2
Colorado	3	4	5

```
s1 = pd.Series([0, 1, 2, 3], index=['a', 'b', 'c', 'd'])
s2 = pd.Series([4, 5, 6], index=['c', 'd', 'e'])
data2 = pd.concat([s1, s2], keys=['one', 'two'])
data2
```

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

Un-stacking may introduce missing values because data frames are rectangular.

```
data2.unstack()
```

	a	b	c	d	e
one	0.0000	1.0000	2.0000	3.0000	NaN
two	NaN	NaN	4.0000	5.0000	6.0000

Stacking drops these missing values by default. However, this behavior may change soon, so check your output!

```
data2.unstack().stack()
```

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

McKinney provides two more subsections on reshaping data with the `.pivot()` and `.melt()` methods. Unlike, the stacking methods, the pivoting methods can aggregate data and do not require an index. We will skip these additional aggregation methods for now.

McKinney Chapter 8 - Practice

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import pandas_datareader as pdr
import yfinance as yf
```

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

Announcements

Five-Minute Review

Practice

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

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

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

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

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

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

We can even add a `shape` argument to return a wide or long data frame!

Download earnings per share for the stocks in `stocks` and combine to a long data frame `earnings`.

Use the `.earnings_dates` method described here. Use `pd.concat()` to combine the result of each the `.earnings_date` data frames and assign them to a new data frame `earnings`. Name the row indexes `Ticker` and `Date` and swap to match the order of the row index in `stocks_long`.

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

Plot the relation between daily returns and earnings surprises

Repeat the earnings exercise with the S&P 100 stocks

With more data, we can more clearly see the positive relation between earnings surprises and returns!

Week 6

McKinney Chapter 10 - Data Aggregation and Group Operations

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import pandas_datareader as pdr
import yfinance as yf
```

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

Introduction

Chapter 10 of McKinney (2022) discusses groupby operations, the pandas equivalent of pivot tables in Excel. Pivot tables calculate statistics (e.g., sums, means, and medians) for one set of variables by groups of other variables (e.g., weekdays and tickers). For example, we could use a pivot table to calculate mean daily stock returns by weekday.

We will focus on:

1. The `.groupby()` method to group by columns and indexes
2. The `.agg()` method to aggregate columns to single values
3. The `.pivot_table()` method as an alternative to `.groupby()`

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

GroupBy Mechanics

"Split-apply-combine" is an excellent way to describe pandas groupby operations.

> Hadley Wickham, an author of many popular packages for the R programming language, coined the term split-apply-combine for describing group operations. In the first stage of the process, data contained in a pandas object, whether a Series, DataFrame, or otherwise, is split into groups based on one or more keys that you provide. The splitting is performed on a particular axis of an object. For example, a DataFrame can be grouped on its rows (axis=0) or its columns (axis=1). Once this is done, a function is applied to each group, producing a new value. Finally, the results of all those function applications are combined into a result object. The form of the resulting object will usually depend on what's being done to the data. See Figure 10-1 for a mockup of a simple group aggregation.

Figure 10-1 visualizes a split-apply-combine operation that:

1. Splits by the `key` column (i.e., "groups by `key`")
2. Applies the sum operation to the `data` column (i.e., "and sums `data`")
3. Combines the grouped sums (i.e., "combines the output")

We could describe this operation as "sum the `data` column by groups of the `key` column then combines the output."

```
np.random.seed(42)
df = pd.DataFrame({'key1' : ['a', 'a', 'b', 'b', 'a'],
                   'key2' : ['one', 'two', 'one', 'two', 'one'],
                   'data1' : np.random.randn(5),
                   'data2' : np.random.randn(5)})

df
```

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

Here is the manual way to calculate the means of `data1` by groups of `key1`.

```
df.loc[df['key1'] == 'a', 'data1'].mean()
```

```
0.0414
```

```
df.loc[df['key1'] == 'b', 'data1'].mean()
```

```
1.0854
```

We can do this calculation more easily!

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

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

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

We can wrap `data1` with two sets of square brackets if we prefer our result as a data frame instead of a series.

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

key1	data1
a	0.0414
b	1.0854

We can group by more than one variable!

```
means = df['data1'].groupby([df['key1'], df['key2']]).mean()
means
```

```
key1  key2
a     one     0.1313
      two    -0.1383
b     one     0.6477
      two     1.5230
Name: data1, dtype: float64
```

We can use the `.unstack()` method if we want to use both rows and columns to organize data. Recall that the `.unstack()` method un-stacks the inner index level (i.e., `level = -1`) by default so that `key2` values become the columns.

```
means.unstack()
```

key2 key1	one	two
a	0.1313	-0.1383
b	0.6477	1.5230

Our grouping variables are typically columns in the data frame we want to group, so the following syntax is more compact and easier to read.

```
df.groupby(['key1', 'key2'])['data1'].mean().unstack()
```

key2 key1	one	two
a	0.1313	-0.1383
b	0.6477	1.5230

We can wrap long chains in parentheses to insert line breaks and improve readability.

```
(
    df
    .groupby(['key1', 'key2'])
    ['data1']
    .mean()
    .unstack()
)
```

key2 key1	one	two
a	0.1313	-0.1383
b	0.6477	1.5230

However, we must pass only numerical columns to numerical aggregation methods. Otherwise, pandas will give a type error. For example, in the following code, pandas unsuccessfully tries to calculate the mean of string `key2`.

```
# df.groupby('key1').mean() # TypeError: agg function failed [how->mean,dtype->object]
```

We avoid this error by slicing the numerical columns.

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

key1	data1	data2
a	0.0414	0.6292
b	1.0854	0.1490

Grouping with Functions

We can also group with functions. Below, we group with the `len` function, which calculates the lengths of the labels in the row index.

```
np.random.seed(42)
people = pd.DataFrame(
    data=np.random.randn(5, 5),
    columns=['a', 'b', 'c', 'd', 'e'],
    index=['Joe', 'Steve', 'Wes', 'Jim', 'Travis']
)

people
```

	a	b	c	d	e
Joe	0.4967	-0.1383	0.6477	1.5230	-0.2342
Steve	-0.2341	1.5792	0.7674	-0.4695	0.5426
Wes	-0.4634	-0.4657	0.2420	-1.9133	-1.7249
Jim	-0.5623	-1.0128	0.3142	-0.9080	-1.4123
Travis	1.4656	-0.2258	0.0675	-1.4247	-0.5444

```
people.groupby(len).sum()
```

	a	b	c	d	e
3	-0.5290	-1.6168	1.2039	-1.2983	-3.3714
5	-0.2341	1.5792	0.7674	-0.4695	0.5426
6	1.4656	-0.2258	0.0675	-1.4247	-0.5444

We can mix functions, lists, dictionaries, etc., as arguments to the `.groupby()` method.

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

		a
3	one	-0.4634
	two	-0.5623
5	one	-0.2341
6	two	1.4656

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

		a	b
3	a	0.4967	-0
	b	-0.5623	-1

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

		a
3	Cool	0.4967
	Nerd	-0.5623
6	Cool	1.4656

Grouping by Index Levels

We can also group by index levels.

```
columns = pd.MultiIndex.from_arrays([['US', 'US', 'US', 'JP', 'JP'],
                                     [1, 3, 5, 1, 3]],
                                     names=['cty', 'tenor'])
hier_df = pd.DataFrame(np.random.randn(4, 5), columns=columns).transpose()
hier_df
```

		0
cty	tenor	
US	1	0.1109
	3	-1.1510
	5	0.3757
JP	1	-0.6006
	3	-0.2917

```
hier_df.groupby(level='cty').count()
```

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

```
hier_df.groupby(level='tenor').count()
```

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

Data Aggregation

Table 10-1 summarizes the optimized groupby methods:

- `count`: Number of non-NA values in the group
- `sum`: Sum of non-NA values
- `mean`: Mean of non-NA values

- `median`: Arithmetic median of non-NA values
- `std`, `var`: Unbiased (n – 1 denominator) standard deviation and variance
- `min`, `max`: Minimum and maximum of non-NA values
- `prod`: Product of non-NA values
- `first`, `last`: First and last non-NA values

These optimized methods are fast and efficient. Still, pandas lets us use non-optimized methods. First, any series method is available.

```
df
```

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

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

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

Second, we can write functions and pass them to the `.agg()` method. These functions should accept an array and return a single value.

```
def max_minus_min(arr):
    return arr.max() - arr.min()
```

```
df.sort_values(by=['key1', 'data1'])
```

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

	key1	key2	data1	data2
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 do not aggregate an array to a scalar.

```
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

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

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

First, here are examples of multiple functions that operate on all columns.

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

key1	mean	median	min	max
a	0.0414	-0.1383	-0.2342	0.4967
b	1.0854	1.0854	0.6477	1.5230

```
df.groupby('key1')[['data1', 'data2']].agg(['mean', 'median', 'min', 'max'])
```

	data1				data2			
	mean	median	min	max	mean	median	min	max
key1								
a	0.0414	-0.1383	-0.2342	0.4967	0.6292	0.5426	-0.2341	1.5792
b	1.0854	1.0854	0.6477	1.5230	0.1490	0.1490	-0.4695	0.7674

Second, here are examples of specific functions that operate on specific columns.

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

	data1	data2
key1		
a	0.0414	0.5426
b	1.0854	0.1490

We must do a little more work if we want to rename `data1` and `data2` to remind ourselves which is the mean and median.

```
df.groupby('key1').agg(
    data1_mean=('data1', 'mean'),
    data2_median=('data2', 'median')
)
```

	data1_mean	data2_median
key1		
a	0.0414	0.5426
b	1.0854	0.1490

We can calculate the mean *and standard deviation* of `data1` and the median of `data2` by `key1`.

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

	data1		data2
	mean	std	median
key1			
a	0.0414	0.3972	0.5426
b	1.0854	0.6190	0.1490

Apply: General split-apply-combine

The `.agg()` method aggregates an array to a scalar. We can use the `.apply()` method for more general calculations that do not return a scalar. For example, the following `top()` function selects the top `n` rows in data frame `x` sorted by column `col`. The `.sort_values()` method sorts from low to high by default.

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

```
df
```

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

The following code returns the one row with the smallest value of `data1` within each group of `key1`.

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

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

The following code returns the *two rows* with the smallest values of `data1` within each group of `key1`.

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

key1		key2	data
a	4	one	-0.23
	1	two	-0.13
b	2	one	0.64
	3	two	1.52

We must use the `.reset_index()` method if we want to drop the index from `df`.

```
(
    df
    .groupby('key1')
    .apply(top, col='data1', n=2, include_groups=False)
    .reset_index(level=1, drop=True)
)
```

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

> **Note**
>
> The `.agg()` and `.apply()` methods both operate on groups created by the `.groupby()` method. However, they serve different purposes and have distinct use cases.
> The `.agg()` method is designed for aggregating data, meaning it applies functions that reduce a group to a single value (e.g., mean, sum, or custom functions that return a single scalar). This method is useful for summarizing data across groups.
> In contrast, the `.apply()` method is more general and flexible. The `.apply()` method returns results of varying shapes. The `.agg()` method is limited to scalar outputs for each group, but the `.apply()` method is not.

Pivot Tables and Cross-Tabulation

Above, we manually made pivot tables with the `.groupby()`, `.agg()`, `.apply()` and `.unstack()` methods. pandas provides Excel-style aggregations with the `.pivot_table()` method and the `pandas.pivot_table()` function. It is worthwhile to read the `.pivot_table()` docstring several times.

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

ind.head()
```

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

Date	Variable Index	Adj
1927-12-30	^DJI ^FTSE ^GSPC ^HSI ^IXIC	NaN NaN 17.6 NaN NaN

```
[**********************67%*******               ]  4 of 6 completed[**********************83
```

Date	Variable Index	Adj
1927-12-30	^DJI ^FTSE ^GSPC ^HSI ^IXIC	NaN NaN 17.6 NaN NaN

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	27706.3194	27706.3194	27850.4849	27546.7680	27702.7036	299024855.2101
^FTSE	7141.7548	7141.7548	7182.5771	7100.0572	7141.1015	813188499.5647
^GSPC	3358.1502	3358.1502	3375.4918	3338.4761	3357.6779	4005763661.6362
^HSI	23826.9733	23826.9733	23991.8855	23657.6522	23841.0154	2121811591.9984
^IXIC	9862.4747	9862.4747	9927.2504	9788.5201	9861.0625	3503727429.7061
^N225	24831.9359	24831.9359	24967.3558	24687.4126	24831.9677	97743271.9836

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

Variable Index	Adj Close	Close	High	Low	Open	Volume
^DJI	26692.0947	26692.0947	26816.6045	26561.1299	26699.8799	303325000.0000
^FTSE	7256.0000	7256.0000	7289.5000	7215.7998	7255.5000	765527900.0000
^GSPC	3005.5850	3005.5850	3016.7800	2991.5000	3006.3800	3819130000.0000
^HSI	24249.4844	24249.4844	24381.0254	24078.3008	24265.7500	1932884000.0000
^IXIC	8520.2598	8520.2598	8539.0698	8457.1650	8475.0903	2905920000.0000
^N225	22823.2598	22823.2598	22922.8008	22728.0605	22849.9102	87300000.0000

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

	min						max	
Index	^DJI	^FTSE	^GSPC	^HSI	^IXIC	^N225	^DJI	^FTSI
Date								
2015-12-31	15666.4404	5874.1001	1867.6100	20556.5996	4506.4902	16795.9609	18312.3906	7104.0
2016-12-31	15660.1797	5537.0000	1829.0800	18319.5801	4266.8398	14952.0195	19974.6191	7142.7
2017-12-31	19732.4004	7099.2002	2257.8301	22134.4707	5429.0801	18335.6309	24837.5098	7687.7
2018-12-31	21792.1992	6584.7002	2351.1001	24585.5293	6192.9199	19155.7402	26828.3906	7877.5
2019-12-31	22686.2207	6692.7002	2447.8899	25064.3594	6463.5000	19561.9609	28645.2598	7686.6
2020-12-31	18591.9297	4993.8999	2237.3999	21696.1309	6860.6699	16552.8301	30606.4805	7674.6
2021-12-31	29982.6191	6407.5000	3700.6499	22744.8594	12609.1602	27013.2500	36488.6289	7420.7
2022-12-31	28725.5098	6826.2002	3577.0300	14687.0195	10213.2900	24717.5293	36799.6484	7672.3
2023-12-31	31819.1406	7256.8999	3808.1001	16201.4902	10305.2402	25716.8594	37710.1016	8014.2
2024-12-31	37266.6719	7446.2998	4688.6802	14961.1797	14510.2998	31458.4199	45014.0391	8445.7
2025-12-31	42392.2695	8223.9805	5868.5498	19623.3203	19280.7891	NaN	42735.2812	8260.0

McKinney Chapter 10 - Practice - Blank

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import pandas_datareader as pdr
import yfinance as yf
```

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

Announcements

Five-Minute Review

Practice

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

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

```
[**********************100%**********************]  6 of 6 completed
[****************      33%                       ]  2 of 6 completed[**********************50
```

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

Calulate the mean and standard deviation of returns by ticker for the MATANA (MSFT, AAPL, TSLA, AMZN, NVDA, and GOOG) stocks

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

```
matana = (
    yf.download(tickers='MSFT AAPL TSLA AMZN NVDA GOOG')
    .rename_axis(columns=['Variable', 'Ticker'])
)
```

```
[**********************100%**********************]  6 of 6 completed
[****************      33%                       ]  2 of 6 completed[**********************50
```

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

Calculate monthly means and volatilities for SPY and GOOG returns

Plot the monthly means and volatilities from the previous exercise

Assign the Dow Jones stocks to five portfolios based on the *preceding* month's volatility

Plot the time-series volatilities of these five portfolios

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

Is market volatility higher during wars?

Here is some guidance:

1. Download the daily factor data from Ken French's website
2. Calculate daily market returns by summing the market risk premium and risk-free rates (`Mkt-RF` and `RF`, respectively)
3. Calculate the volatility (standard deviation) of daily returns *every month* by combining `pd.Grouper()` and `.groupby()`)
4. Multiply by $\sqrt{252}$ to annualize these volatilities of daily returns
5. Plot these annualized volatilities

Is market volatility higher during wars? Consider the following dates:

1. WWII: December 1941 to September 1945
2. Korean War: 1950 to 1953
3. Viet Nam War: 1959 to 1975
4. Gulf War: 1990 to 1991
5. War in Afghanistan: 2001 to 2021

Week 7

McKinney Chapter 11 - Time Series

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import pandas_datareader as pdr
import yfinance as yf
```

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

Introduction

Chapter 11 of McKinney (2022) discusses time series and panel data, which is where pandas *excels!* We will use these time series and panel tools every day for the rest of the course.

We will focus on:

1. Slicing a data frame or series by date or date range
2. Using `.shift()` to create leads and lags of variables
3. Using `.resample()` to change the frequency of variables
4. Using `.rolling()` to aggregate data over moving or rolling windows

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

Time Series Basics

Let us create a time series to play with.

```
from datetime import datetime
dates = [
    datetime(2011, 1, 2),
    datetime(2011, 1, 5),
    datetime(2011, 1, 7),
    datetime(2011, 1, 8),
    datetime(2011, 1, 10),
    datetime(2011, 1, 12)
]
np.random.seed(42)
ts = pd.Series(np.random.randn(6), index=dates)

ts
```

```
2011-01-02    0.4967
2011-01-05   -0.1383
2011-01-07    0.6477
2011-01-08    1.5230
2011-01-10   -0.2342
2011-01-12   -0.2341
dtype: float64
```

Note that pandas converts the `datetime` objects to a pandas `DatetimeIndex` object and a single index value is a `Timestamp` object.

```
ts.index
```

```
DatetimeIndex(['2011-01-02', '2011-01-05', '2011-01-07', '2011-01-08',
               '2011-01-10', '2011-01-12'],
              dtype='datetime64[ns]', freq=None)
```

```
ts.index[0]
```

```
Timestamp('2011-01-02 00:00:00')
```

Recall that pandas automatically aligns objects on indexes.

```
ts
```

```
2011-01-02    0.4967
2011-01-05   -0.1383
2011-01-07    0.6477
2011-01-08    1.5230
2011-01-10   -0.2342
2011-01-12   -0.2341
dtype: float64
```

```
ts.iloc[::2]
```

```
2011-01-02    0.4967
2011-01-07    0.6477
2011-01-10   -0.2342
dtype: float64
```

```
ts + ts.iloc[::2]
```

```
2011-01-02    0.9934
2011-01-05       NaN
2011-01-07    1.2954
2011-01-08       NaN
2011-01-10   -0.4683
2011-01-12       NaN
dtype: float64
```

Indexing, Selection, Subsetting

pandas uses U.S.-style date strings (e.g., "M/D/Y") or unambiguous date strings (e.g., "YYYY-MM-DD") to select data.

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

```
-0.2342
```

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

```
-0.2342
```

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

```
-0.2342
```

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

```
-0.2342
```

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

```
-0.2342
```

pandas *does not* use U.K.-style date strings.

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

Let us create a *longer* time series to play with.

```
np.random.seed(42)
longer_ts = pd.Series(
    data=np.random.randn(1000),
    index=pd.date_range('1/1/2000', periods=1000)
)

longer_ts
```

```
2000-01-01    0.4967
2000-01-02   -0.1383
2000-01-03    0.6477
2000-01-04    1.5230
2000-01-05   -0.2342
               ...
2002-09-22   -0.2811
2002-09-23    1.7977
2002-09-24    0.6408
2002-09-25   -0.5712
2002-09-26    0.5726
Freq: D, Length: 1000, dtype: float64
```

We can specify a year-month to slice all of the observations in May of 2001.

```
longer_ts['2001-05']
```

```
2001-05-01   -0.6466
2001-05-02   -1.0815
2001-05-03    1.6871
2001-05-04    0.8816
2001-05-05   -0.0080
2001-05-06    1.4799
2001-05-07    0.0774
2001-05-08   -0.8613
2001-05-09    1.5231
2001-05-10    0.5389
2001-05-11   -1.0372
2001-05-12   -0.1903
2001-05-13   -0.8756
2001-05-14   -1.3828
2001-05-15    0.9262
2001-05-16    1.9094
2001-05-17   -1.3986
2001-05-18    0.5630
2001-05-19   -0.6506
2001-05-20   -0.4871
2001-05-21   -0.5924
2001-05-22   -0.8640
2001-05-23    0.0485
2001-05-24   -0.8310
2001-05-25    0.2705
2001-05-26   -0.0502
2001-05-27   -0.2389
2001-05-28   -0.9076
2001-05-29   -0.5768
2001-05-30    0.7554
2001-05-31    0.5009
Freq: D, dtype: float64
```

We can also specify a year to slice all observations in 2001.

```
longer_ts['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
```

However, we cannot date slice if our data are not sorted chronologically.

```
ts2 = ts.sort_values()
ts2
```

```
2011-01-10   -0.2342
2011-01-12   -0.2341
2011-01-05   -0.1383
2011-01-02    0.4967
2011-01-07    0.6477
2011-01-08    1.5230
dtype: float64
```

The following date slice fails because `ts2` is not sorted chronologically.

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

We can use the `.sort_index()` method first to allow date slices.

```
ts2.sort_index()['1/6/2011':'1/11/2011']
```

```
2011-01-07    0.6477
2011-01-08    1.5230
2011-01-10   -0.2342
dtype: float64
```

As with label slices, date slices are inclusive on both ends.

```
longer_ts['1/6/2001':'1/11/2001']
```

```
2001-01-06    0.4980
2001-01-07    1.4511
2001-01-08    0.9593
2001-01-09    2.1532
2001-01-10   -0.7673
2001-01-11    0.8723
Freq: D, dtype: float64
```

Recall that if we modify a slice, we modify the original series or data frame.

> Remember that slicing in this manner produces views on the source time series like slicing NumPy arrays. This means that no data is copied and modifications on the slice will be reflected in the original data.

```
ts3 = ts.copy()
ts3
```

```
2011-01-02    0.4967
2011-01-05   -0.1383
2011-01-07    0.6477
2011-01-08    1.5230
2011-01-10   -0.2342
2011-01-12   -0.2341
dtype: float64
```

```
ts4 = ts3.iloc[:3]
ts4
```

```
2011-01-02    0.4967
2011-01-05   -0.1383
2011-01-07    0.6477
dtype: float64
```

```
ts4.iloc[:] = 2001
ts4
```

```
2011-01-02   2001.0000
2011-01-05   2001.0000
2011-01-07   2001.0000
dtype: float64
```

```
ts3
```

```
2011-01-02   2001.0000
2011-01-05   2001.0000
2011-01-07   2001.0000
2011-01-08      1.5230
2011-01-10     -0.2342
2011-01-12     -0.2341
dtype: float64
```

Series `ts` is unchanged because `ts3` is a *copy* of `ts`!

```
ts
```

```
2011-01-02    0.4967
2011-01-05   -0.1383
2011-01-07    0.6477
2011-01-08    1.5230
2011-01-10   -0.2342
2011-01-12   -0.2341
dtype: float64
```

Time Series with Duplicate Indices

Most our of data in this course will be well-formed with one observation per date-time for series or one observation per individual per date-time for data frames. However, we may later receive poorly-formed data with duplicate observations. Here, series `dup_ts` has three observations on February 2nd.

```
dates = pd.DatetimeIndex(['1/1/2000', '1/2/2000', '1/2/2000', '1/2/2000', '1/3/2000'])
dup_ts = pd.Series(data=np.arange(5), index=dates)
dup_ts
```

```
2000-01-01    0
2000-01-02    1
2000-01-02    2
2000-01-02    3
2000-01-03    4
dtype: int64
```

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

```
dup_ts.index.is_unique
```

```
False
```

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

```
np.int64(4)
```

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

```
2000-01-02    1
2000-01-02    2
2000-01-02    3
dtype: int64
```

The solution to duplicate data depends on the context. For example, we may want the mean of all observations on a given date. The `.groupby()` method can help us here.

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

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

Or keep the first value on each date.

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

```
2000-01-01    0
2000-01-02    1
2000-01-03    4
dtype: int64
```

Date Ranges, Frequencies, and Shifting

> Generic time series in pandas are assumed to be irregular; that is, they have no fixed frequency. For many applications this is sufficient. However, it's often desirable to work relative to a fixed frequency, such as daily, monthly, or every 15 minutes, even if that means introducing missing values into a time series. Fortunately pandas has a full suite of standard time series frequencies and tools for resampling, inferring frequencies, and generating fixed-frequency date ranges.

Shifting Data

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

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

ts
```

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

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

1. The date index remains the same
2. Values *shift down* N observations

The `.shift()` method defaults to $N = 1$.

```
ts.shift()
```

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

```
ts.shift(1)
```

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

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

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

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

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

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

> **Note**
>
> We almost never shift with negative values to prevent a look-ahead bias. That is, assuming chronological sorting, we almost never bring values from the future back to the present. We do not want to assume that financial market participants know the future.

The `.shift()` examples above shift by N observations without considering time stamps. As a result, the time stamps are unchanged, and values shift down for positive `periods` or up for negative `periods`. However, we can specify the `freq` argument to consider time stamps. With the `freq` argument, time stamps shift by `periods` multiples of the `freq` argument.

```
ts
```

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

```
ts.shift(periods=2, freq='ME')
```

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

```
ts.shift(periods=3, freq='D')
```

```
2000-02-03    0.4967
2000-03-03   -0.1383
2000-04-03    0.6477
2000-05-03    1.5230
dtype: float64
```

`M` is already months, so `min` is minutes.

```
ts.shift(periods=1, freq='90min')
```

```
2000-01-31 01:30:00    0.4967
2000-02-29 01:30:00   -0.1383
2000-03-31 01:30:00    0.6477
2000-04-30 01:30:00    1.5230
dtype: float64
```

Calculating returns

We can calculate returns in two ways. First, easily with the `.pct_change()` method.

```
ts.pct_change()
```

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

Second, manaully with the `.shift()` method.

```
(ts - ts.shift()) / ts.shift()
```

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

These two return calculations are the same.

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

```
True
```

Two observations on these return calculations:

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

Shifting dates with offsets

We can also shift time stamps to the beginning or end of a period.

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

```
datetime.datetime(2011, 11, 17, 0, 0)
```

`MonthEnd(0)` moves to the end of the month *but does not leave the current month.*

```
now + MonthEnd(0)
```

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

`MonthEnd(1)` moves to the end of the *current* month. If already at the end of the *current* month, it moves to the end of the *next* month.

```
now + MonthEnd(1)
```

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

```
now + MonthEnd(1) + MonthEnd(1)
```

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

*Be careful! The `MonthEnd()` default is `n=1`!***

```
datetime(2021, 10, 31) + MonthEnd(0)
```

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

```
datetime(2021, 10, 31) + MonthEnd(1)
```

```
Timestamp('2021-11-30 00:00:00')
```

```
datetime(2021, 10, 31) + MonthEnd()
```

```
Timestamp('2021-11-30 00:00:00')
```

Always check your output!

Resampling and Frequency Conversion

Resampling is an important feature!

> Resampling refers to the process of converting a time series from one frequency to another. Aggregating higher frequency data to lower frequency is called downsampling, while converting lower frequency to higher frequency is called upsampling. Not all resampling falls into either of these categories; for example, converting W-WED (weekly on Wednesday) to W-FRI is neither upsampling nor downsampling.

We can resample both series and data frames. The `.resample()` method syntax is similar to `.groupby()`.

Downsampling

> Aggregating data to a regular, lower frequency is a pretty normal time series task. The data you're aggregating doesn't need to be fixed frequently; the desired frequency defines bin edges that are used to slice the time series into pieces to aggregate. For example, to convert to monthly, 'M' or 'BM', you need to chop up the data into one-month intervals. Each interval is said to be half-open; a data point can only belong to one interval, and the union of the intervals must make up the whole time frame. There are a couple things to think about when using resample to downsample data:
>
> - Which side of each interval is closed
> - How to label each aggregated bin, either with the start of the interval or the end

```
rng = pd.date_range(start='2000-01-01', periods=12, freq='min')
ts = pd.Series(np.arange(12), index=rng)
ts
```

```
2000-01-01 00:00:00     0
2000-01-01 00:01:00     1
2000-01-01 00:02:00     2
2000-01-01 00:03:00     3
2000-01-01 00:04:00     4
2000-01-01 00:05:00     5
2000-01-01 00:06:00     6
2000-01-01 00:07:00     7
2000-01-01 00:08:00     8
2000-01-01 00:09:00     9
2000-01-01 00:10:00    10
2000-01-01 00:11:00    11
Freq: min, dtype: int64
```

We can aggregate the one-minute frequency data above to five-minute frequency data. Resampling requires an aggregation method. Here, we use the `.sum()` method.

```
ts.resample('5min').sum()
```

```
2000-01-01 00:00:00    10
2000-01-01 00:05:00    35
2000-01-01 00:10:00    21
Freq: 5min, dtype: int64
```

When we resample with a minute-frequency:

1. Left edges of the resampling interval are closed (included) and right edges are open (excluded)
2. Labels are by the left edge of the resampling interval by default

In the example above, the first value of 10 at midnight is the sum of values at midnight until 00:05, excluding the value at 00:05. That is, the sums are $10 = 0 + 1 + 2 + 3 + 4$ at 00:00, $35 = 5 + 6 + 7 + 8 + 9$ at 00:05, and so on. We can use the `closed` and `label` arguments to change this behavior.

In finance, we generally prefer `closed='right'` and `label='right'` to avoid a lookahead bias.

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

```
2000-01-01 00:00:00     0
2000-01-01 00:05:00    15
2000-01-01 00:10:00    40
2000-01-01 00:15:00    11
Freq: 5min, dtype: int64
```

These defaults for minute-frequency data may seem odd, but any choice is arbitrary. The defaults for weekly and lower frequencies are `closed='right'` and `label='right'`, which are corect for finance. Still, we should read the docstring and check our output whenever we use the `.resample()` method!

Upsampling and Interpolation

To downsample (i.e., resample from higher to lower frequency), we must aggregate (e.g., `.mean()`, `.sum()`, `.first()`, or `.last()`). To upsample (i.e., resample from lower to higher frequency), we must choose how to fill in the new, higher-frequency observations.

```
np.random.seed(42)
frame = pd.DataFrame(
    data=np.random.randn(2, 4),
    index=pd.date_range('1/1/2000', periods=2, freq='W-WED'),
    columns=['Colorado', 'Texas', 'New York', 'Ohio']
)

frame
```

	Colorado	Texas	New York	Ohio
2000-01-05	0.4967	-0.1383	0.6477	1.5230
2000-01-12	-0.2342	-0.2341	1.5792	0.7674

We use the `.asfreq()` method to convert to the new frequency and leave the new observations as missing.

```
df_daily = frame.resample('D').asfreq()
df_daily
```

	Colorado	Texas	New York	Ohio
2000-01-05	0.4967	-0.1383	0.6477	1.5230

	Colorado	Texas	New York	Ohio
2000-01-06	NaN	NaN	NaN	NaN
2000-01-07	NaN	NaN	NaN	NaN
2000-01-08	NaN	NaN	NaN	NaN
2000-01-09	NaN	NaN	NaN	NaN
2000-01-10	NaN	NaN	NaN	NaN
2000-01-11	NaN	NaN	NaN	NaN
2000-01-12	-0.2342	-0.2341	1.5792	0.7674

We use the `.ffill()` method to forward fill values to replace missing values.

```
frame.resample('D').ffill()
```

	Colorado	Texas	New York	Ohio
2000-01-05	0.4967	-0.1383	0.6477	1.5230
2000-01-06	0.4967	-0.1383	0.6477	1.5230
2000-01-07	0.4967	-0.1383	0.6477	1.5230
2000-01-08	0.4967	-0.1383	0.6477	1.5230
2000-01-09	0.4967	-0.1383	0.6477	1.5230
2000-01-10	0.4967	-0.1383	0.6477	1.5230
2000-01-11	0.4967	-0.1383	0.6477	1.5230
2000-01-12	-0.2342	-0.2341	1.5792	0.7674

```
frame.resample('D').ffill(limit=2)
```

	Colorado	Texas	New York	Ohio
2000-01-05	0.4967	-0.1383	0.6477	1.5230
2000-01-06	0.4967	-0.1383	0.6477	1.5230
2000-01-07	0.4967	-0.1383	0.6477	1.5230
2000-01-08	NaN	NaN	NaN	NaN
2000-01-09	NaN	NaN	NaN	NaN
2000-01-10	NaN	NaN	NaN	NaN
2000-01-11	NaN	NaN	NaN	NaN
2000-01-12	-0.2342	-0.2341	1.5792	0.7674

```
frame.resample('W-THU').ffill()
```

	Colorado	Texas	New York	Ohio
2000-01-06	0.4967	-0.1383	0.6477	1.5230
2000-01-13	-0.2342	-0.2341	1.5792	0.7674

Moving Window Functions

Moving or rolling window functions are one of the neatest features of pandas.!

```
df = (
    yf.download(tickers=['AAPL', 'MSFT', 'SPY'])
    .rename_axis(columns=['Variable', 'Ticker'])
)

df.head()
```

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

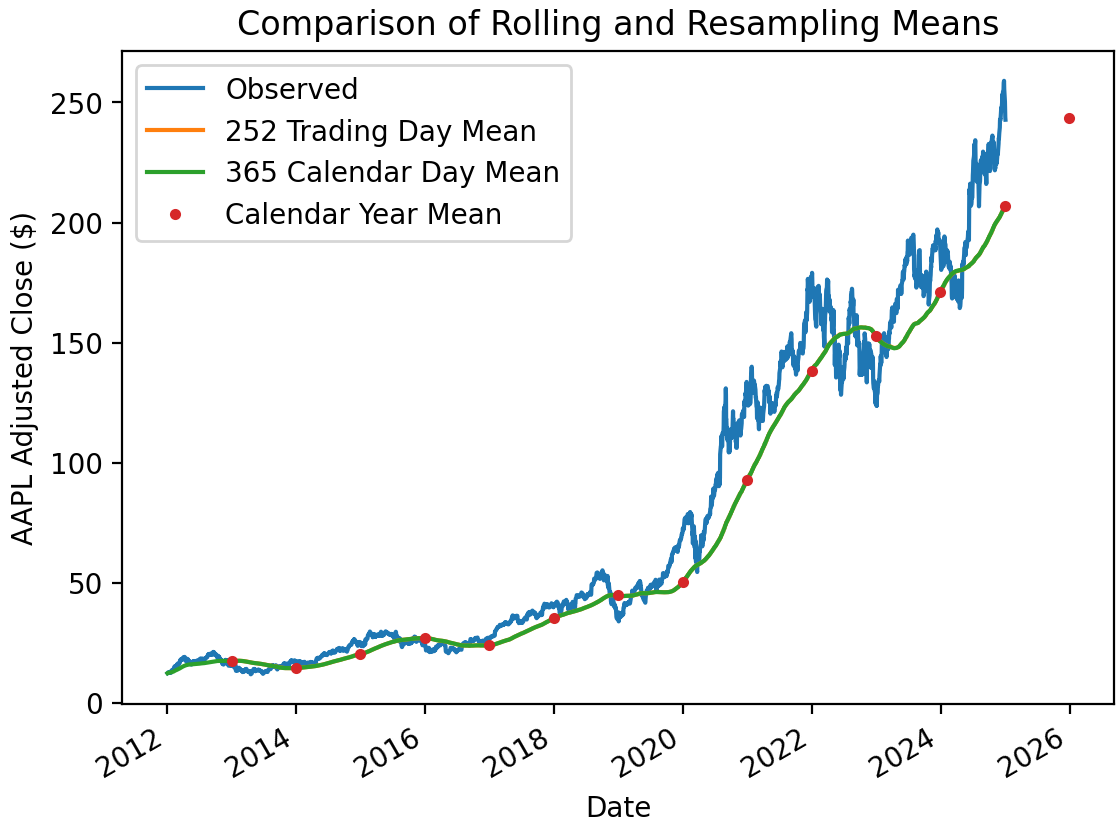
Variable	Adj Close			Close			High			Low		
Ticker	AAPL	MSFT	SPY	AAPL	MSFT	SPY	AAPL	MSFT	SPY	AAPL	MSFT	SPY
Date												
1980-12-12	0.0988	NaN	NaN	0.1283	NaN	NaN	0.1289	NaN	NaN	0.1283	NaN	NaN
1980-12-15	0.0937	NaN	NaN	0.1217	NaN	NaN	0.1222	NaN	NaN	0.1217	NaN	NaN
1980-12-16	0.0868	NaN	NaN	0.1127	NaN	NaN	0.1133	NaN	NaN	0.1127	NaN	NaN
1980-12-17	0.0890	NaN	NaN	0.1155	NaN	NaN	0.1161	NaN	NaN	0.1155	NaN	NaN
1980-12-18	0.0915	NaN	NaN	0.1189	NaN	NaN	0.1194	NaN	NaN	0.1189	NaN	NaN

```
[**********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	0.0988	NaN	NaN	0.1283	NaN	NaN	0.1289	NaN	NaN	0.1283	NaN	NaN
1980-12-15	0.0937	NaN	NaN	0.1217	NaN	NaN	0.1222	NaN	NaN	0.1217	NaN	NaN
1980-12-16	0.0868	NaN	NaN	0.1127	NaN	NaN	0.1133	NaN	NaN	0.1127	NaN	NaN
1980-12-17	0.0890	NaN	NaN	0.1155	NaN	NaN	0.1161	NaN	NaN	0.1155	NaN	NaN
1980-12-18	0.0915	NaN	NaN	0.1189	NaN	NaN	0.1194	NaN	NaN	0.1189	NaN	NaN

The `.rolling()` method accepts a window-width and requires an aggregation method. The following example plots AAPL's observed daily price alongside its 252-trading-day rolling mean, 365-calendar-day rolling mean, and calendar year mean.

```
aapl = df.loc['2012':, ('Adj Close', 'AAPL')]
aapl.plot(label='Observed')
aapl.rolling(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 1
aapl.resample('YE').mean().plot(style='.', label='Calendar Year Mean')
plt.legend()
plt.ylabel('AAPL Adjusted Close ($)')
plt.title('Comparison of Rolling and Resampling Means')
plt.show()
```



> **Note**
>
> If we specify the rolling window width as an integer:
>
> 1. Each rolling window is that many observations wide and ignores time stamps
> 2. Each rolling window must have that many non-missing observations
>
> We can specify `min_periods` to allow incomplete windows. For integer window widths, `min_periods` defaults to the given integer window width. For string date offsets, `min_periods` defaults to `1`.

Binary Moving Window Functions

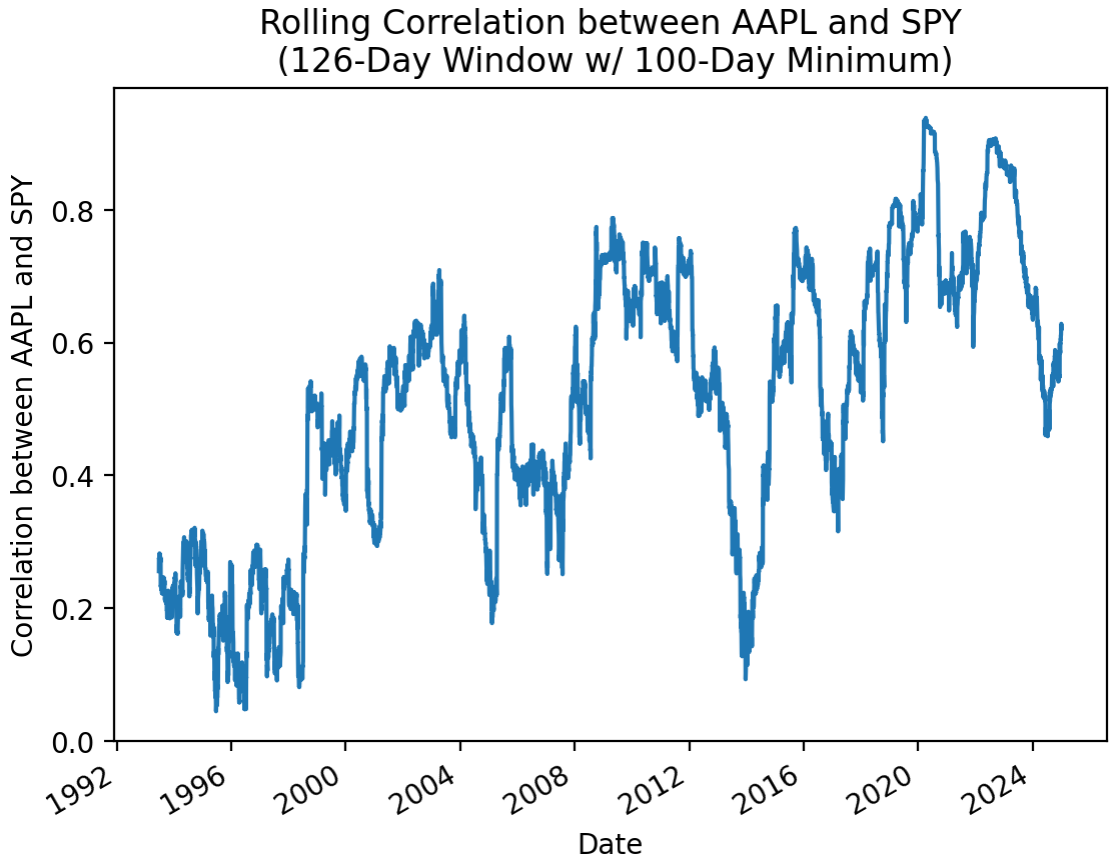
Binary moving window functions accept two inputs. The most common example is the rolling correlation between two return series.

```
returns = df['Adj Close'].iloc[:-1].pct_change()
returns
```

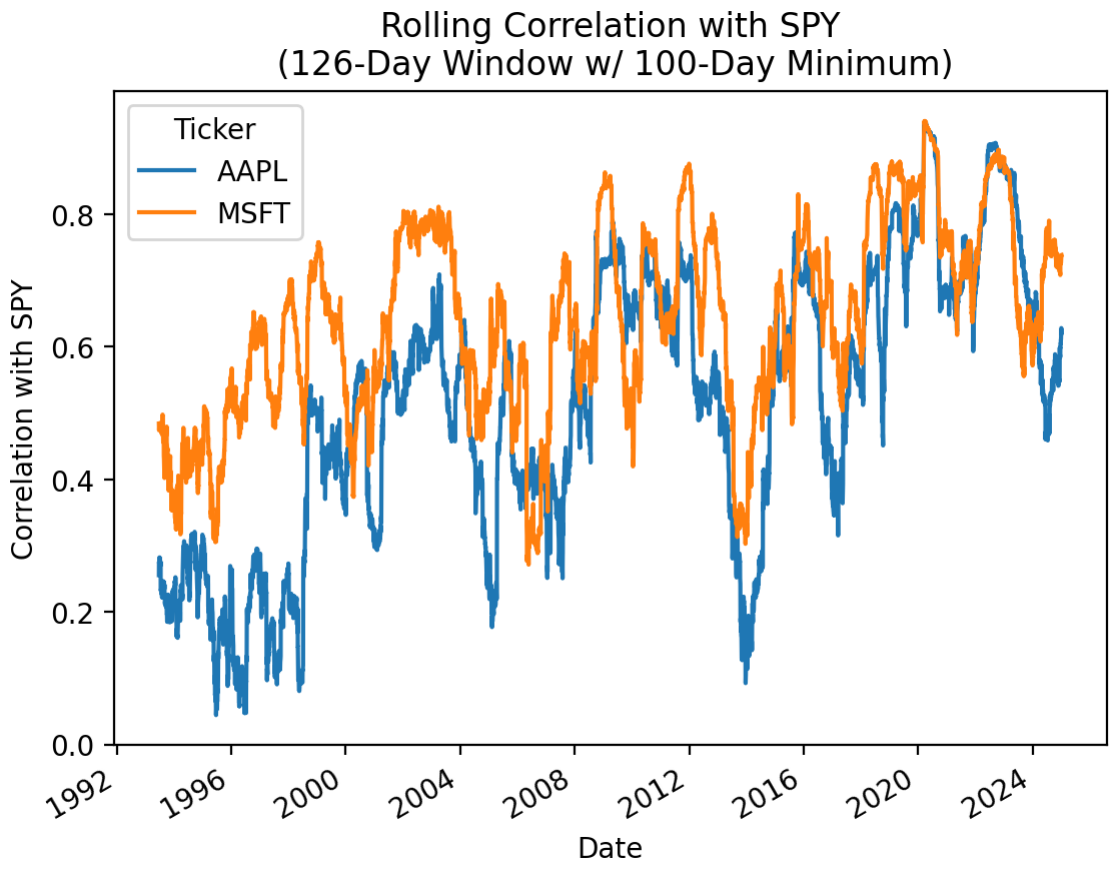
Ticker Date	AAPL	MSFT	SPY
1980-12-12	NaN	NaN	NaN
1980-12-15	-0.0522	NaN	NaN
1980-12-16	-0.0734	NaN	NaN
1980-12-17	0.0248	NaN	NaN
1980-12-18	0.0290	NaN	NaN
...	...	...	...
2024-12-26	0.0032	-0.0028	0.0001
2024-12-27	-0.0132	-0.0173	-0.0105
2024-12-30	-0.0133	-0.0132	-0.0114
2024-12-31	-0.0071	-0.0078	-0.0036
2025-01-02	-0.0262	-0.0069	-0.0025

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

```
plt.title('Rolling Correlation between AAPL and SPY\n (126-Day Window w/ 100-Day Minimum)')
plt.show()
```




```
(
    returns[['AAPL', 'MSFT']]
    .rolling(126, min_periods=100)
    .corr(returns['SPY'])
    .plot()
)
plt.ylabel('Correlation with SPY')
plt.title('Rolling Correlation with SPY\n (126-Day Window w/ 100-Day Minimum)')
plt.show()
```

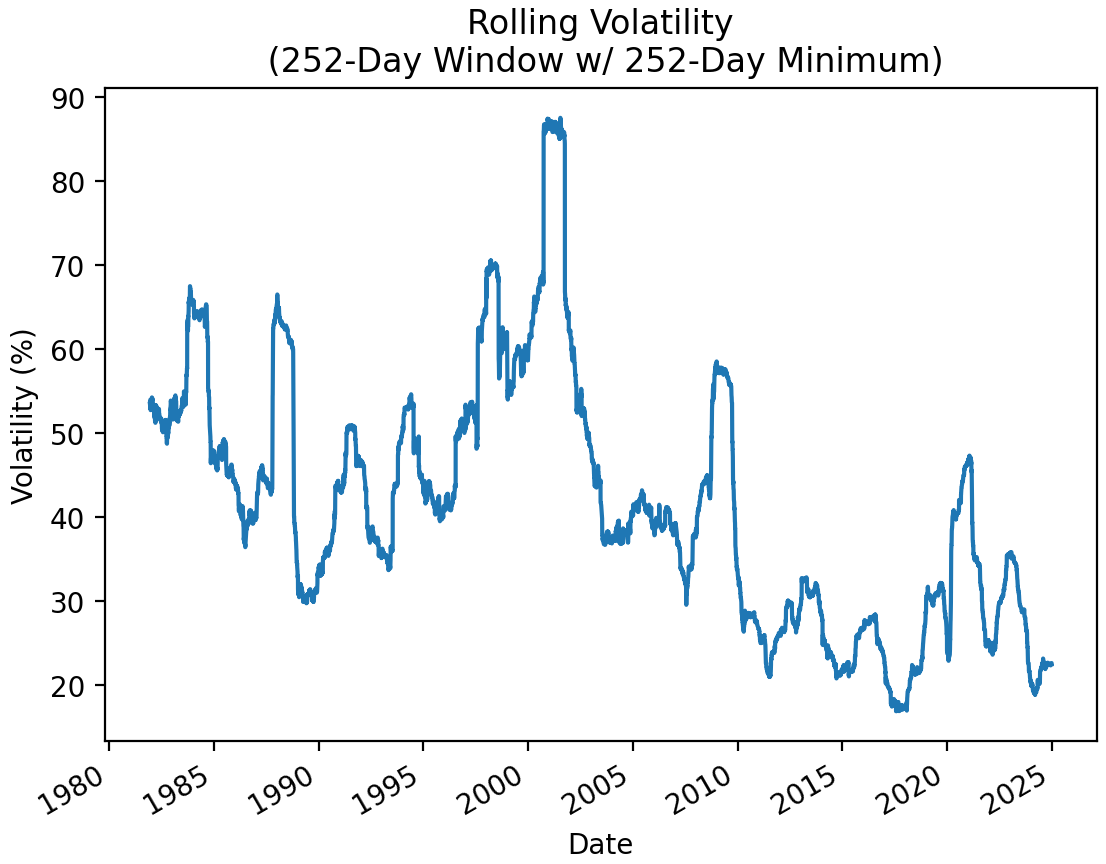


User-Defined Moving Window Functions

We can define our own moving window functions and use them with the `.apply()` method. However, note that `.apply()` method will be much slower than the optimized methods, like `.mean()` and `.std()`. Here, we will calculate rolling volatility with `.apply()` and `.std()` and compare their speeds.

```
(
    returns['AAPL']
    .rolling(252)
    .apply(np.std)
    .mul(np.sqrt(252) * 100)
    .plot()
)
plt.ylabel('Volatility (%)')
```

```
plt.title('Rolling Volatility\n (252-Day Window w/ 252-Day Minimum)')
plt.show()
```



Do not be afraid to use `.apply()`, but realize that `.apply()` is often 1000 times slower than the optimized method!

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

```
624 ms ± 121 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

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

```
296  s ± 93.6  s per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

McKinney Chapter 11 - Practice - Blank

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import pandas_datareader as pdr
import yfinance as yf
```

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

Announcements

Five-Minute Review

Practice

Download daily returns for ten portfolios formed on book-to-market ratios

Plot cumulative returns for all available data

Calculate total returns for each calendar year

Calculate total returns for all 252-trading-day windows

Calculate total returns for 12-months windows with monthly data

Calculate Sharpe Ratios for each calendar year

Calculate rolling betas

We can calculate CAPM betas as: $\beta_i = \frac{Cov(r_i - r_f, r_M - r_f)}{Var(r_M - r_f)}$

Calculate rolling Sharpe Ratios

Week 8

Project 1

I will determine assignment details at least one week before each due date.

Week 9

Student's Choice 1

We will vote to determine the "Student's Choice" topics during the first few weeks of class.

Week 10

Student's Choice 2

We will vote to determine the "Student's Choice" topics during the first few weeks of class.

Week 11

Project 2

I will determine assignment details at least one week before each due date.

Week 12

Student's Choice 3

We will vote to determine the "Student's Choice" topics during the first few weeks of class.

Week 13

Student's Choice 4

We will vote to determine the "Student's Choice" topics during the first few weeks of class.

Week 14

MSFQ Assessment Exam

We will take the assessment exam in class on Tuesday of this week.

Week 15

Project 3

I will determine assignment details at least one week before each due date.

References

McKinney, Wes (2022). *Python for Data Analysis*. 3rd ed. O'Reilly Media, Inc.