Personal Finance with Python

Using pandas, Requests, and Recurrent

Max Humber

Apress®

Personal Finance with Python

Using pandas, Requests, and Recurrent

Max Humber

Apress®

Personal Finance with Python: Using pandas, Requests, and Recurrent

Max Humber
Toronto, Ontario, Canada

ISBN-13 (pbk): 978-1-4842-3801-1 ISBN-13 (electronic): 978-1-4842-3802-8
https://doi.org/10.1007/978-1-4842-3802-8

Library of Congress Control Number: 2018951264

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Steve Anglin
Development Editor: Matthew Moodie
Coordinating Editor: Mark Powers

Cover designed by eStudioCalamar

Cover image designed by Freepik (www.freepik.com)

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail editorial@apress.com; for reprint, paperback, or audio rights, please email bookpermissions@springernature.com.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/9781484238011. For more detailed information, please visit www.apress.com/source-code.

Printed on acid-free paper

Table of Contents

About the Author

Max Humber is a Data Engineer interested in improving finance with technology. He works for Wealthsimple and previously served as the first data scientist for the online lending platform Borrowell. He has spoken at Pycon, ODSC, PyData, useR, and BigDataX in Colombia, London, Berlin, Brussels, and Toronto.

About the Technical Reviewer

Michael Thomas has worked in software development for more than 20 years as an individual contributor, team lead, program manager, and vice president of engineering. Michael has more than 10 years of experience working with mobile devices. His current focus is in the medical sector, using mobile devices to accelerate information transfer between patients and healthcare providers.

Introduction

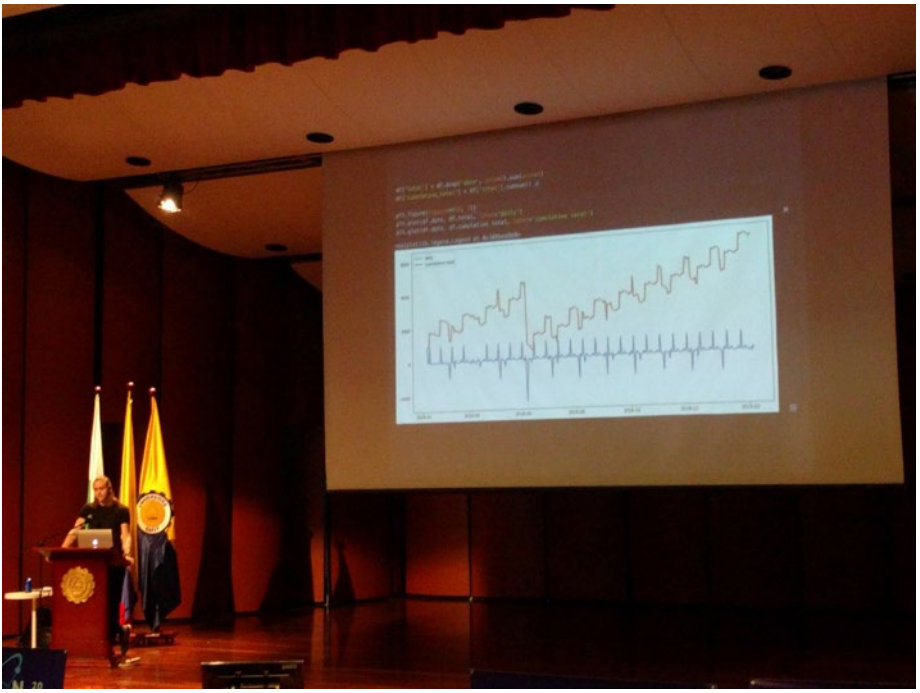
This book is about Python and personal finance and how you can effectively mix the two together. It is a crash course on how deal with data, how to build up financial formulas in code from scratch, and how to evaluate and think about money in your day-to-day life.

Although each chapter is driven by an idea in personal finance, this book is not an exhaustive compendium on the topic. I try to provide just enough theory in each chapter to get you going, but I made a conscious effort to abstract away and hide a lot of the math so that we don't get stuck in the weeds.

If I'm being completely honest with you (and myself), this book is my love letter to pandas,[1] the main data manipulation library in Python. pandas is a wonderful tool and has become the bedrock on top of which a lot (if not most) machine learning is built. If you get good at pandas (and this book should help!), you will be well positioned to dive into the world of machine learning. But if machine learning isn't your thing, don't worry, I promise that you can still get a lot of value out of this book.

The content of this book was inspired by a presentation I delivered in Medellín, Colombia, in 2018 entitled Personal Pynance. It has been adapted, refactored, stretched, extended, and polished for your enjoyment. I hope you enjoy it!

[1] https://pandas.pydata.org/



Me! Presenting Personal Pynance at Pycon, Colombia (Photo Credit: Moisés Vargas[2])

What This Book Covers

This book covers the following topics.

Profit

You'll explore the idea of spending money to make money with a timely motivating example. You will learn about DataFrames, the basics of loading data in pandas, how to get Python to play nicely with Excel, how to

[2] https://twitter.com/moisesvw/status/962414647272976384

think about and calculate net present value and internal rate return, and how to apply functions to data.

Convert

You'll learn how to convert currency with Python. You will learn how to query a third-party API, how to coerce API responses into something usable, how to manage secrets, how to handle errors, and how to create Python classes.

Amortize

You'll learn how to evaluate a buffet of loan options from different financial institutions. You will learn about how to calculate fixed-rate payments with numpy, how to build amortization schedules from scratch, how to build loops, and how to make those loops ultra-efficient and wicked quick.

Budget

You'll explore how to generate a budget that provides day-by-day cash flow resolution. You will learn how to deal with dates in Python, how to visualize data, how to use the recurrent library to parse English sentences, and how to work with the YAML file format.

Invest

You'll explore how to build a portfolio rebalancer. You will learn how to instantiate a portfolio, how to fetch stock quotes, how to update values in a DataFrame, and how to simulate order processing.

Spend

You'll explore how to forecast spending. You will learn how to use pandas and Prophet and how to use the past to generate values into the future. This chapter is a bit silly, but it provides a little window into the world of machine learning with Python and pandas.

Who This Book Is For

This book is for anyone interested in Python, personal finance, or how to combine the two! It is geared toward those who want to better understand how to manage money more effectively and toward those who just want to learn or improve their Python.

Although this book assumes some (minimal) familiarity with programming and the Python language, if you don't have any, don't worry! Everything is built up piece by piece, and the first chapters are slow enough to start. A background in finance is not required.

What You Need for This Book

To ensure that you can run all the code in the book, it is recommended that you install Python (3.6 or newer) with Anaconda. All the setup and configuration details can be found in Chapter 1.

Code Examples

To get the most out of this book, you should actually run the code examples on your own machine as you follow along. Running the code, seeing how it works, and playing with it will help you to internalize everything that is presented.

Code that you should execute will look like this:

```
import pandas as pd
```

Code that generates output (like a `print` statement, table, or chart) will look similar to this, with its output:

```
ages = pd.DataFrame(data = {
    'name': ['max', 'sunny'],
    'age': [24, 22]
})
print(ages)
```

```
   age   name
      0         24    max
      1         22  sunny
```

Here's another input-output code example:

```
print(ages['name'])
```

```
      0           max
      1         sunny
Name: name, dtype: object
```

Reader Feedback

Feedback is always welcome. Let me know what you think about this book—what you liked or may have disliked.

To provide general feedback, simply send me an e-mail and mention the book title in the subject of your message:

max.humber@gmail.com

Acknowledgments

The following reviewers provided valuable feedback on the first draft of this book: David Tingle, Radovan Kavicky, Matthew Braymer-Hayes, Daniel Schissler, Zecca Lehn, Owen Jones, Jesus Rogel-Salazar, Thomas Koller, Burhan ul haq, Eija-Leena Koponen, Moisés Vargas Martínez, Francisco Pérez Cuadrado, David Asboth, and Costin Apostol. This book is far better than it might have been because of them.

Thanks to Apress for taking a chance on me and to the following individuals for their hard work on getting this book out of the red zone and into the end zone: Steve Anglin, Matthew Moodie, Mark Powers, Amrita Stanley, Nirmal Selvaraj, Joseph Quatela, and technical reviewer Michael Thomas.

Finally, I'd like to dedicate this book to my parents, Kim and Rich. I wouldn't be where I am today without them.

CHAPTER 1

Setup

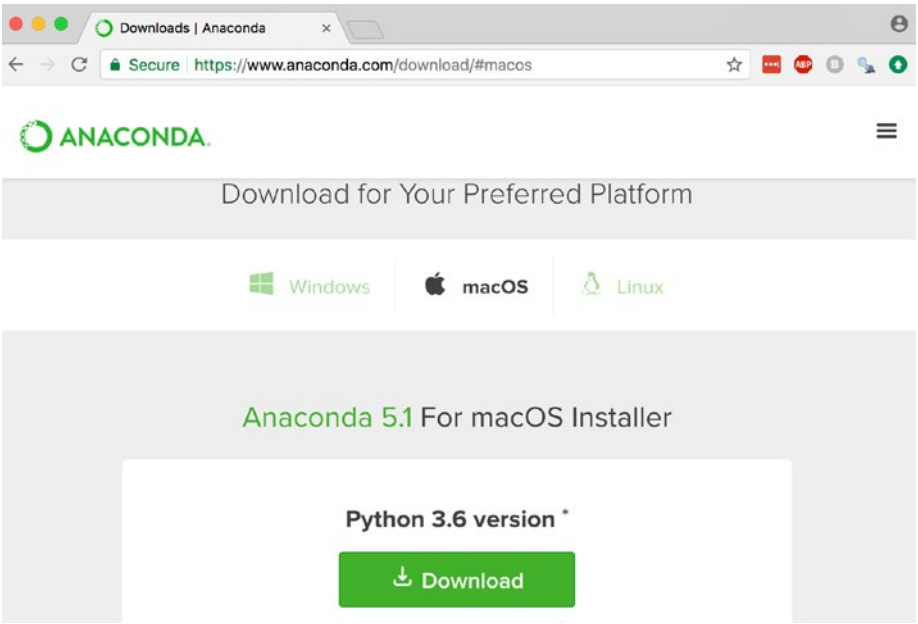
To run the code examples in this book, you will need Python 3.6 (or newer), Jupyter, and a bunch of libraries from the Python data stack.

Anaconda

The easiest way to get everything that you'll need for this book is to install Anaconda.[1] Just go to the Anaconda website and download the relevant distribution for your operating system.

[1] https://www.anaconda.com/

M. Humber, *Personal Finance with Python*, https://doi.org/10.1007/978-1-4842-3802-8_1

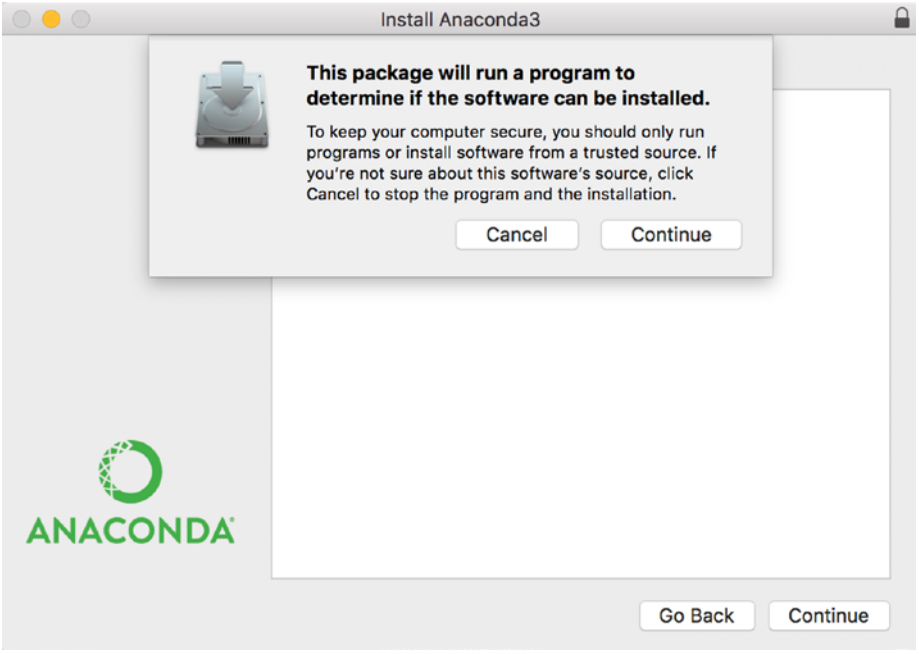


Anaconda works on Windows, macOS, and Linux. I run macOS, so I'm going to demo that install process.

For those who are not on macOS, Anaconda has some excellent install documentation available online.[2]

Once you've downloaded the distribution, open the installer and click through all the prompts.

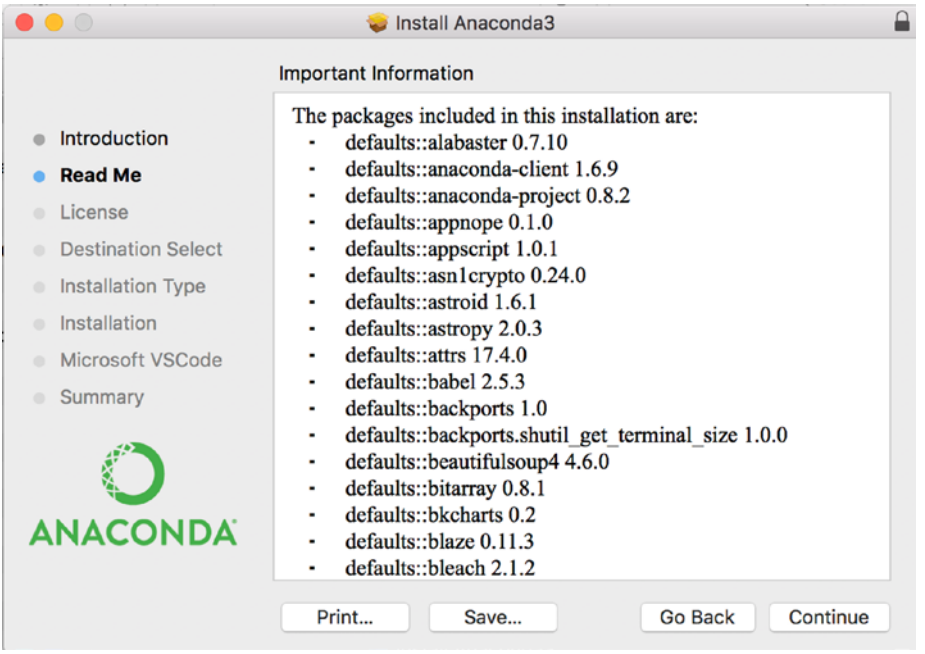
[2] https://docs.anaconda.com/anaconda/install/



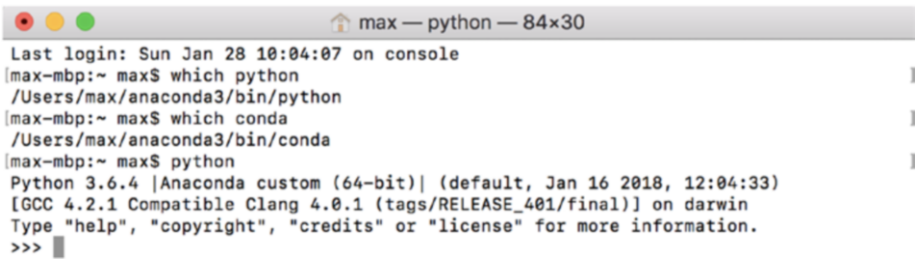
Anaconda will install an up-to-date Python 3.6 and above release, as well as a bunch of useful Python packages such as pandas, numpy,[3] and beautifulsoup.[4]

[3] www.numpy.org/

[4] https://www.crummy.com/software/BeautifulSoup/



After installation, open Terminal—or your preferred command-line interface—and run all the bits after the $ to make sure that Anaconda did its thing.

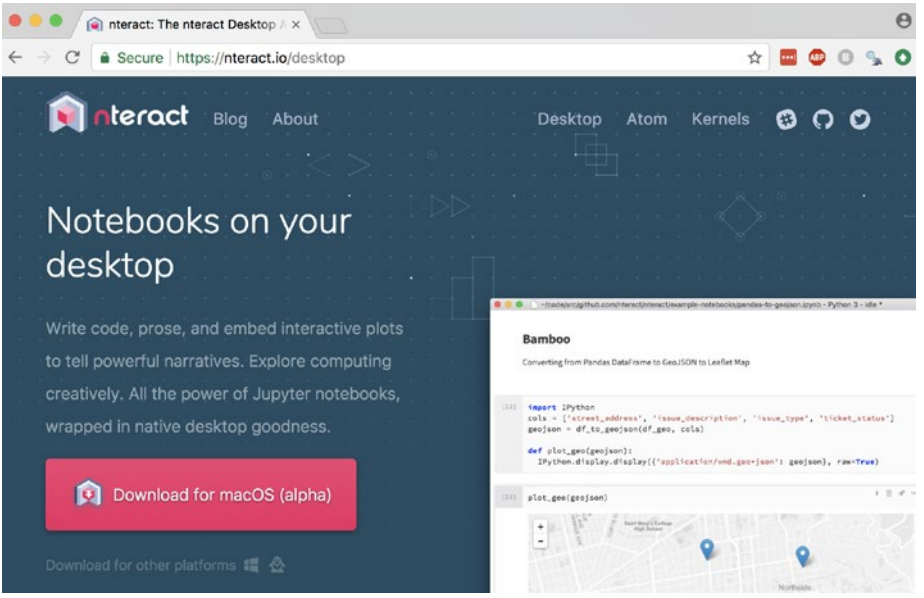


If your Python is in `Users/<your name>/anaconda3/bin/python` and typing in `python` spits out something like `Python 3.6.4 |Anaconda`, then you're almost ready to rock and roll. Just one more thing…

nteract

Note If you already know how to use Jupyter Notebooks, this step isn't strictly necessary. However, nteract is pretty darn slick; you should give it a fair shake!

To actually run Python code, you'll need a Jupyter[5] Notebook interface called nteract. While Jupyter is an open-source web application that allows you to create documents that contain live code (and was installed for you with Anaconda), nteract is a super user-friendly desktop-based skin for Jupyter. You can download nteract from the nteract.io website.[6]

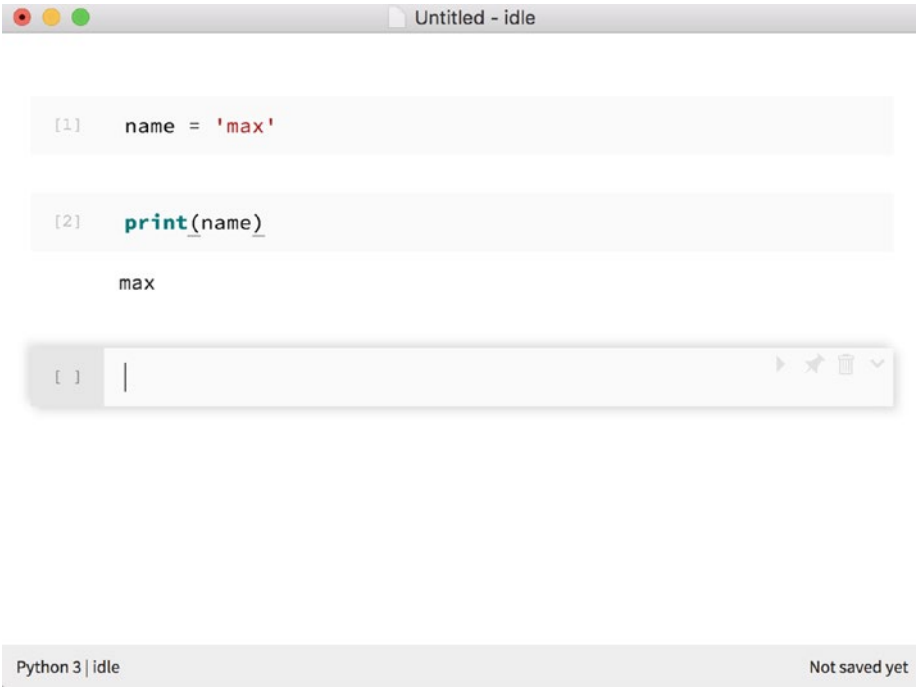


[5] https://jupyter.org/

[6] https://nteract.io/

Once it's downloaded and installed, you can load a fresh Jupyter Notebook by clicking the nteract icon. You will be greeted with a screen that looks like this:

With a blank slate, you can now run arbitrary Python code inside empty cells, the results of which will be printed just below the input cell.



To execute code inside a cell, just hit `Shift+Enter`. To insert a new cell, just select Edit ➤ New Code Cell or use the shortcut `Cmd+Shift+N`. You can find a list of all the macOS (and Windows) shortcuts in the nteract `USER_GUIDE.md`.[7]

Quickly make sure that imports are working for you by running the following:

```
import pandas as pd
```

If everything is in working order, nothing should've happened!

However, if after executing `import pandas as pd` you get something like this:

[7] https://github.com/nteract/nteract/blob/master/USER_GUIDE.md

```
---------------------------------------------------------------
ModuleNotFoundError                Traceback (most recent call last)
in ()
----> 1 import pandas as pd

ModuleNotFoundError: No module named 'pandas'
```

then reinstall Anaconda and make sure your `PATH` is properly configured.[8]

pip install

Sometimes you'll hit a legitimate `ModuleNotFoundError: No module named [PACKAGE NAME]` error. This happens when a module/library/package (I'll use these terms interchangeably) isn't yet installed on your machine. Often these errors can be remedied by running the following in a Notebook cell, or without the leading `!` inside Terminal:

```
!pip install [PACKAGE NAME]
```

I'll try my best to call out these installs when you need them. But if you hit a `ModuleNotFoundError`, I trust that you'll know what to do (Google is your friend)!

Data

Some examples in the following chapters will require data that I've curated specifically for this book. You can download these data files by clicking the **Download Source Code** button located at www.apress.com/9781484238011, or by navigating to my personal Github repository, located at https://github.com/maxhumber/pfwp.

[8] https://stackoverflow.com/questions/39438049/how-to-set-the-default-python-path-for-anaconda

CHAPTER 2

Profit

> You know, you got to spend money to make money.
>
> —Chief Keef

A couple of weeks ago my grandma asked me if she should put some money into Bitcoin. I didn't know what to tell her. But I knew that in a book about finance I would have to at least give Bitcoin and cryptocurrencies at least a little bit of lip service.

For the uninitiated, cryptocurrencies like Bitcoin (and Ethereum, Dogecoin, and Zcash) are digital assets that are designed to function as a medium of exchange and that use cryptography to secure transactions, to control the creation of new money, and to verify asset transfer.

Because I think it's hilarious, I'm going to use Dogecoin[1] as the glue for the rest of this chapter. But honestly, these ideas extend beyond Dogecoin (and crypto for that matter). They apply whenever you have to spend money to make money.

So, if you're not a fan of crypto or Dogecoin (I certainly don't blame you), bear with me. Dogecoin just seemed like a lot more fun to talk about than something else random, like a lemonade stand.

[1] https://en.wikipedia.org/wiki/Dogecoin

M. Humber, *Personal Finance with Python*, https://doi.org/10.1007/978-1-4842-3802-8_2

Mining

If you're interested in Dogecoin and want to own some, you can do one of the following:

- Buy it
- Generate it through a process called *mining*

Buying Dogecoin is straightforward. But mining is more interesting. To grossly oversimplify things, to mining Dogecoin requires the installation of a program, on your computer that uses your hardware to solve computationally expensive math problems.

Solve a problem. Get a Dogecoin. Easy-peasy.

While you can run mining applications on your laptop, those serious about mining—for Dogecoin or otherwise—opt to run these sorts of applications on specialized rigs. A decent rig can be incredibly expensive, though. A mining rig requires specialized graphical processing units (GPUs), a good motherboard, a lot of RAM, a decent CPU, a case, some fans, and a bunch of other components. Casually dropping $3,000 on parts for a mining setup is pretty standard these days.

For the purposes of this chapter, let's pretend that we front $3,000 for the necessary components.

If we mine $6,000 on top of our initial investment, some quick math will tell us that our realized profit is $3,000 and our return on investment (ROI) is as follows:

```
r = 6000
i = 3000
print(f'{(r - i) / i * 100}%')
```

```
100.0%
```

ROI

Dope, right? Nope, because ROI is a super-misleading measurement. It's misleading because it doesn't factor in how long it took to make back our money.

Perhaps we mined $6,000 in six months. Or, maybe it took us six years. Whether months or years, the calculation and the answer are the same. Both yield a 100 percent ROI.

IRR

The fact that ROI nets out to the same value for different time horizons should bug us (it bugs me). Time is money after all, right? And $3,000 today is worth more than $3,000 tomorrow, or the next year for that matter.

On top of that, investing in something like a mining rig precludes us from investing in something else, like a lemonade stand. ROI doesn't capture any of that.

The internal rate of return (IRR), however, does. IRR is the discount rate at which the net present value of a potential investment is 0.

Importantly, if your investment has an expected IRR of 5 percent but your cost of borrowing is 8 percent, you should not invest in the opportunity or venture. But when the numbers are reversed—that is to say that the IRR exceeds the cost of borrowing—you should invest!

=IRR()

At this point in our imaginary example we're in for $3,000. But instead of doubling our money, let's bring things ever so slightly back to Earth and pretend that our Dogecoin rig generates $1,000 in passive income each year over the next four years.

	date	income	expenses
0	2017-01-01	0	-3000
1	2018-01-01	1000	0
2	2019-01-01	1000	0
3	2020-01-01	1000	0
4	2021-01-01	1000	0

To calculate the IRR for this example, we can fire up Excel, total up the inflows and outflows, and use the built-in `=IRR()` formula to get what we need.

	date	income	expenses	total
0	2017-01-01	0	-3000	-3000
1	2018-01-01	1000	0	1000
2	2019-01-01	1000	0	1000
3	2020-01-01	1000	0	1000
4	2021-01-01	1000	0	1000

If `total` is in Excel column `D`, then applying the formula `=IRR(D0:D4)` will get us something close to 13 percent. However, `=IRR()` is a bit clunky and inflexible. The formula assumes that our inflows and outflows occur on a regular basis.

If our mining rig instead generates income that flows like this:

	date	income	expenses	total
0	2017-01-01	40	-3000	-2960
1	2017-01-25	40	-50	-10
2	2017-02-12	80	-50	30
3	2017-02-14	100	-30	70
4	2017-03-04	100	-20	80
5	2017-04-23	160	-30	130
6	2017-05-07	140	-20	120
7	2017-05-21	140	-40	100
8	2017-06-04	80	-40	40
9	2017-06-19	180	-30	150
10	2017-07-16	360	-40	320

(*continued*)

	date	income	expenses	total
11	2017-08-27	160	-30	130
12	2017-09-24	240	-20	220
13	2017-10-21	420	-50	370
14	2017-11-19	400	-20	380
15	2017-12-03	340	-40	300
16	2017-12-17	360	-40	320
17	2017-12-31	540	-40	500

then `=IRR()` will yield the wrong result because the formula doesn't have any concept of the dates attached. Thankfully, `IRR()` has a companion function named `=XIRR()` that allows us to deal with irregular cash flows.

The `XIRR` formula maps dollar values to date values, so running `=XIRR(D0:D17, A0:A17)` will yield the correct IRR, while `=IRR(D0:D17)` will return something super wonky.

pandas

Although Excel is perfectly suitable for calculating IRR/XIRR, it's time we move this party to Python and pandas. Because while Excel excels at this simple stuff, it's not going to cut it when we get to the fun (and more complex) parts of this book.

So, let's switch gears now, fire up nteract (or a Jupyter Notebook), and start cooking with pandas.

read_excel

One of the really nice things about pandas is that it's well equipped to deal with all of our existing `.xlsx` and `.xls` data. The pandas `read_excel` function is a great place to start as it can turn sheets and workbooks into the panda-native DataFrame format.

Note I've stored the data for this chapter in a folder called `data/` no `../data`. If the `xirr.xlsx` file you downloaded from the setup chapter file is somewhere different, like in your Downloads folder, change the path to match. You'll use either `'Downloads/xirr.xlsx'` or `'Downloads/data/xirr.xlsx'`.

```
import pandas as pd
df = pd.read_excel('data/xirr.xlsx', sheet_name="regular")
df
```

	date	income	expenses
0	2017-01-01	0	-3000
1	2018-01-01	1000	0
2	2019-01-01	1000	0
3	2020-01-01	1000	0
4	2021-01-01	1000	0

To replicate the IRR workflow we just ran in Excel, we first have to create a new `total` column in our DataFrame object by using the pandas square-bracket notation.

```
df['total'] = df.income + df.expenses
df
```

	date	income	expenses	total
0	2017-01-01	0	-3000	-3000
1	2018-01-01	1000	0	1000
2	2019-01-01	1000	0	1000
3	2020-01-01	1000	0	1000
4	2021-01-01	1000	0	1000

Once the total column is created, we can run `xirr()` on top of the `.total` and `.date` columns.

```
xirr(df.total, df.date)
```

```
0.1258660808393406
```

And boom. That's it. Chapter. Over.

SIKE![2]

Unfortunately, if you try to run `xirr()` right now, you'll hit a `NameError: name 'xirr' is not defined`. It's not defined because Python doesn't actually come with an `xirr` function.

It's cool, though. We'll roll our own![3]

xnpv

Just before we roll our own, remember when I said that the internal rate of return (IRR/XIRR) is tightly coupled with net present value (NPV/XNPV)? Well, if we want to define an `xirr` function, we first have to build an `xnpv` function.

Here it is:

```
def xnpv(rate, values, dates):
    '''Replicates the XNPV() function'''
    min_date = min(dates)
    return sum([
        value / (1 + rate)**((date - min_date).days / 365)
        for value, date
        in zip(values, dates)
    ])
```

[2] https://gph.is/1VRbuEc

[3] https://www.quora.com/Computer-Science-Where-did-the-phrase-Roll-your-own-come-from-and-why-is-it-used-in-CS?share=1

Don't worry! I know it looks kind of intimidating, but I promise it's not that bad.

Let's slow things down and break apart `xpnv` so that we can actually see what's going on. The essential part of the function is just this thing:

```
value / (1 + rate)**((date - min_date).days
```

If we grab all the values and dates from the `df` object, we can see what it does.

```
values = df.total
dates = df.date
print('Values:', list(values))
print('Dates:', list(dates))
```

```
Values: [-3000, 1000, 1000, 1000, 1000]
Dates: [Timestamp('2017-01-01 00:00:00'), Timestamp('2018-01-01
00:00:00'), Timestamp('2019-01-01 00:00:00'), Timestamp('2020-01-01
00:00:00'), Timestamp('2021-01-01 00:00:00')]
```

We also need a random discount rate (don't worry, we're going to change it later; just think of this as the cost of borrowing) and the minimum date in the `dates` list.

```
rate = 0.05
min_date = min(dates)
print(min_date)
```

```
2017-01-01 00:00:00
```

Once we have these pieces, we can run the meat of the `xpnv` function on top of the first values in each list (Python starts indexing at 0, so the first value is at `data[0]`).

```
date = dates[0]
value = values[0]
value / (1 + rate)**((date - min_date).days / 365)
```

```
-3000.0
```

All we did was bring -$3,000 to the present, but it was already in the present because it was attached to the first (minimum) date value.

If we turn to the second cash flow ($1,000) and bring it to the present, things become a little bit more instructive.

```
date = dates[1]
value = values[1]
print(value)
value / (1 + rate)**((date - min_date).days / 365)
```

```
1000

952.3809523809523
```

We could continue running this formula for each date-value pair, or we could be lazy and wrap everything in a list comprehension and use `zip` to bring all the cash flows to the present at the same time.

```
intermediate_step = [
    value / (1 + rate)**((date - min_date).days / 365)
    for value, date
    in zip(values, dates)
]
print(intermediate_step)
```

```
[-3000.0, 952.3809523809523, 907.0294784580499, 863.837598531476,
822.5925101174964]
```

> **Note** List comprehensions are a really easy way to create lists in Python. They follow this format: `[expression for item in list if conditional]`.
>
> And `zip` just acts like a zipper, aggregating values from two different lists and returning one thing.

With a list of all the present values stored in the object `intermediate_step`, the only thing left is to sum up all the values and print out the result.

```
print(sum(intermediate_step))
xnpv(0.05, df.total, df.date)
```

```
545.8405394879746
```

```
545.8405394879746
```

See, `xnpv` isn't that bad!

What does it all mean? Well, at a discount rate of 5 percent (we randomly picked this value, but think of it as your cost of capital or the return you might get from another potential investment opportunity), the net present value of all our expected cash flows is $545.84.

xirr

With `xnpv` in the bag and out of the way, building an `xirr` function is somewhat trivial. It's trivial because IRR is just NPV but set to 0. If we want, we can actually derive the internal rate of return manually through a process of trial and error.

```
xnpv(0.05, df.total, df.date)
```

```
545.8405394879746
```

```
print(xnpv(0.04, df.total, df.date))
print(xnpv(0.06, df.total, df.date))
print(xnpv(0.07, df.total, df.date))
print(xnpv(0.08, df.total, df.date))
print(xnpv(0.09, df.total, df.date))
print(xnpv(0.11, df.total, df.date))
print(xnpv(0.12, df.total, df.date))
print(xnpv(0.125, df.total, df.date))
print(xnpv(0.1255, df.total, df.date))
print(xnpv(0.1258, df.total, df.date))
print(xnpv(0.12583, df.total, df.date))
print(xnpv(0.12586, df.total, df.date))
```

```
629.8033770546891
464.97917229138625
387.0698546137953
311.9718737447598
239.55263528320688
102.25737356965487
37.15205553499129
5.437960934594116
2.2965732963834853
0.4143376303915147
0.22622105096445466
0.038123913067124704
```

Exhausting but doable!

Because the last value is pretty damn close to 0, we can conclude that the IRR for our mining rig is around 12.586 percent. Great! However, futzing around with our `xnpv` function is from ideal. It would be nice if we had a function that could find the zero for us.

This is a lot harder, so instead of rolling our own "zero-finder," let's lean on the Newton-Raphson optimization method from the SciPy library.

```
from scipy.optimize import newton

def xirr(values, dates):
    '''Replicates the XIRR() function'''
    return newton(lambda r: xnpv(r, values, dates), 0)

xirr(df.total, df.date)
```

```
0.1258660808393406
```

Breaking down the `newton()` function is a bit outside the scope of this book. If you're curious to learn more, Wikipedia[4] has a great page on how it all works.

Again!

Now that we have all the pieces built and a solid workflow for calculating IRR, let's apply our process to the irregular cash flow schedule from earlier.

```
df = pd.read_excel('data/xirr.xlsx', sheet_name="irregular")
df['total'] = df.income + df.expenses
df
```

[4] https://simple.wikipedia.org/wiki/Newton's_method

	date	income	expenses	total
0	2017-01-01	40	-3000	-2960
1	2017-01-25	40	-50	-10
2	2017-02-12	80	-50	30
3	2017-02-14	100	-30	70
4	2017-03-04	100	-20	80
5	2017-04-23	160	-30	130
6	2017-05-07	140	-20	120
7	2017-05-21	140	-40	100
8	2017-06-04	80	-40	40
9	2017-06-19	180	-30	150
10	2017-07-16	360	-40	320
11	2017-08-27	160	-30	130
12	2017-09-24	240	-20	220
13	2017-10-21	420	-50	370
14	2017-11-19	400	-20	380
15	2017-12-03	340	-40	300
16	2017-12-17	360	-40	320
17	2017-12-31	540	-40	500

```
xirr(df.total, df.date)
```

```
0.13812581670383556
```

To the moon!

Conclusion

Before we close out this chapter, I should lay down a couple of things.

Please don't invest in Dogecoin (or crypto or mining) because of me or because of this chapter. The space is crazy, and if I'm honest, you're probably going to get burned playing with the stuff. If you want to get into mining, do your homework!

It's possible that you're thinking, "That was a lot of work to calculate the IRR when Excel does it for free. Why do I need Python?" That's a super-fair question. Although Python has a steep learning curve, the payoffs are huge. My hope is that the need for Python will become self-evident in the chapters to follow.

CHAPTER 3

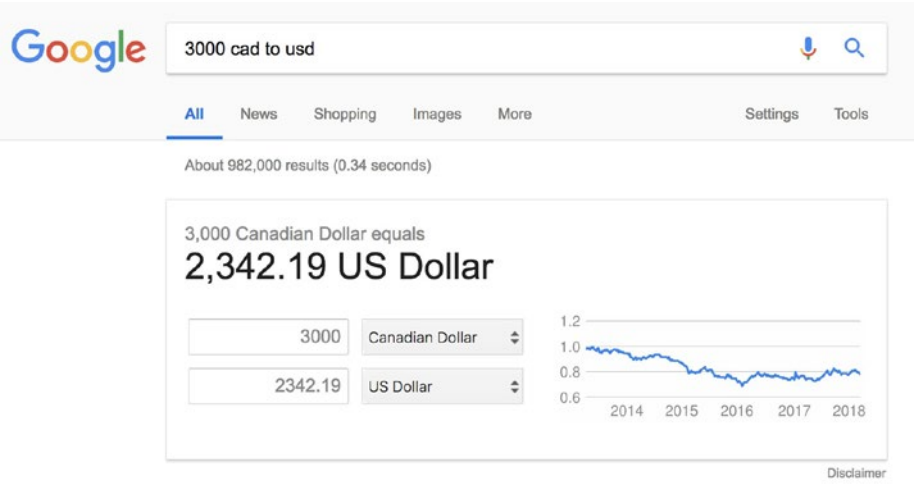
Convert

> All this foreign money I can't count what I'm making.
>
> —Yung Lean

You probably didn't notice, but all the values in the previous chapter were in Canadian dollars (CAD). Everything was in CAD because I'm from Canada.

Confusing and annoying, right? Like what even is a Canadian dollar, and what in Drake's name is it worth? Alright, I shouldn't be that patronizing; you probably have a decent idea about the value of a dollar in Canada as it's closely coupled with a United States dollar (USD).

But we should be precise. It's not enough to say that CAD is just kind of like USD! Let's actually figure out the exact exchange rate.



M. Humber, *Personal Finance with Python*, https://doi.org/10.1007/978-1-4842-3802-8_3

Although Googling for the answer is totally legit, there's no fun in that! And if we have a couple hundred values and need to convert them to handful of different currencies, Google might not be the best solution.

Faced with a repetitive problem, I'm quick to reach for Python.

Unfortunately, though, when it comes to converting in Python, the story is a lot like IRR. The language *can* do it; it just can't do it on its own.

Python needs a little help—help in the form of an exchange rate API and the requests[1] library.

You can install requests by executing the following:

```
!pip install requests
```

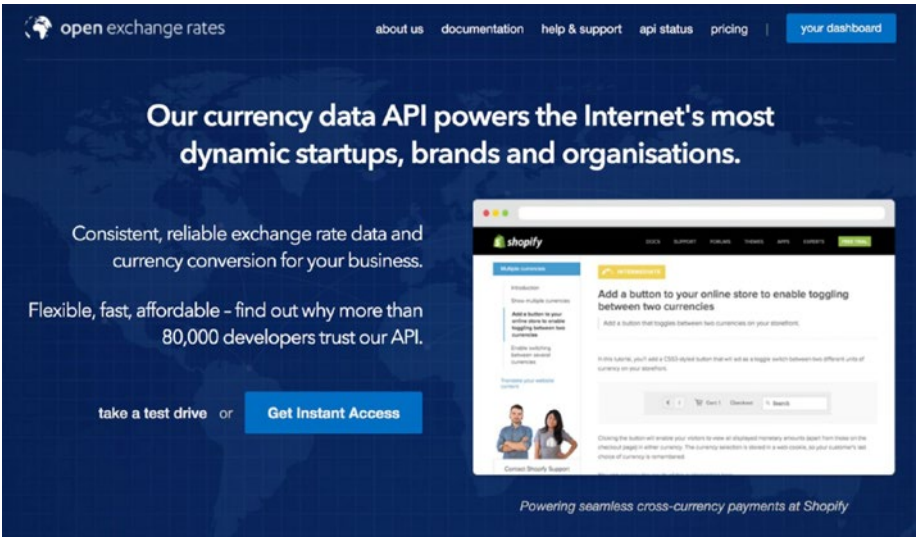
openexchangerates.org

If and when I need to convert a bunch of values, I like to use the Open Exchange Rates API.

- It's easy to use.
- It's updated hourly for more than 200 currencies.
- It's free (for up to 1,000 requests per month).

To follow along, sign up and register[2] for a free API key.

[1] http://docs.python-requests.org/en/master/
[2] https://openexchangerates.org/signup



Secrets

Once you've registered, you can find your unique API key in the Open Exchange Rates Dashboard. Your key will look something like this:

```
`9a156a49bc84f849fde848`
```

Every request we make against the Open Exchange Rates API will need this key attached; otherwise, we won't get any data back from the service.

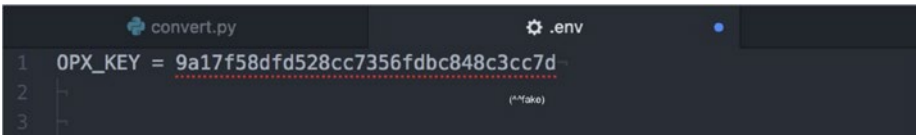
You might be inclined to shove your key into a Python script or Jupyter Notebook. Resist the urge! Keeping secrets (like a key) in plain sight is a recipe for disaster. If you decide to share your code with someone, they will have access to your key and can—if they're so inclined—abuse it and get your key blacklisted.

To hack ourselves out of this problem, we need to keep our API key a secret. There are a bunch of ways to handle secrets in Python. One way is to use environment variables and the dotenv[3] library.

```
!pip install -U python-dotenv
```

[3] https://github.com/theskumar/python-dotenv

With dotenv, you create and save an .env file (with Atom, VS Code, TextEdit or something similar) in the same director/folder that contains your script or Jupyter Notebook and then stuff your secrets inside of it.



Once that's defined, you can load up your secrets by following the pattern.

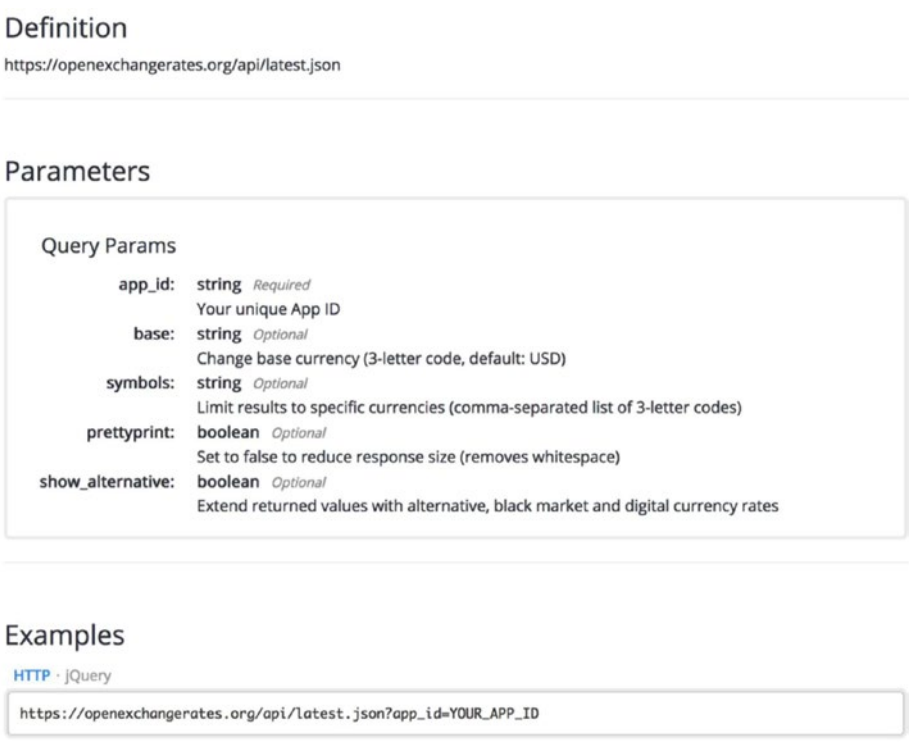
```
import os
from dotenv import load_dotenv, find_dotenv
load_dotenv(find_dotenv())

API_KEY = os.environ.get('OPX_KEY')

print(API_KEY)
>>> 9a156a49bc84f8ce156a4749bc848
```

Again, your key will be different. Mine is fake, so don't even try.

Documentation

With our keys properly hidden, we can start making requests against the Open Exchange Rates API. Reading through the API documentation,[4] we can see that our queries have to conform to the following structure:

[4] https://docs.openexchangerates.org/docs/latest-json



Definition

https://openexchangerates.org/api/latest.json

Parameters

Query Params

app_id:	**string** *Required* Your unique App ID
base:	**string** *Optional* Change base currency (3-letter code, default: USD)
symbols:	**string** *Optional* Limit results to specific currencies (comma-separated list of 3-letter codes)
prettyprint:	**boolean** *Optional* Set to false to reduce response size (removes whitespace)
show_alternative:	**boolean** *Optional* Extend returned values with alternative, black market and digital currency rates

Examples

HTTP · jQuery

```
https://openexchangerates.org/api/latest.json?app_id=YOUR_APP_ID
```

Loading up a browser and sticking a URL[5] into the search bar will yield the following:

```
{
  "disclaimer": "Usage subject to terms:
   https://openexchangerates.org/terms",
  "license": "https://openexchangerates.org/license",
  "timestamp": 1519588738,
  "base": "USD",
```

[5] https://openexchangerates.org/api/latest.json?app_id=9a156a49bc84f8ce156a4749bc848&symbols;=CAD,USD&show;_alternative=true

```
  "rates": {
    "CAD": 1.303016,
    "USD": 1
  }
}
```

Sick. We got data back!

However, manually coercing everything into a URL is not the best or preferred way of doing things.

Instead, it's a lot better (and a lot easier) to build on top of the requests library and fit all of our "query params" into a params payload, like so:

```
import requests

API_KEY = os.environ.get('OPX_KEY')

r = requests.get(
    'https://openexchangerates.org/api/latest.json',
    params = {
        'app_id': API_KEY,
        'symbols': 'CAD,USD',
        'show_alternative': 'true'
        }
    )
```

The response from Open Exchange Rates will now be stored as JSON inside the `r` object and can be accessed with the following:

```
rates_ = r.json()['rates']
rates_

{'CAD': 1.303016, 'USD': 1}
```

Given that `rates_` is just a dictionary, we can access key-value pairs with the `.get` method and run some conversions according to the formula.

```
symbol_from = 'CAD'
symbol_to = 'USD'
value = 3000
```

```
value * 1/rates_.get(symbol_from) * rates_.get(symbol_to)
```

```
2302.350853711697
```

Your values will probably be different because exchange rates move all the time.

Encapsulate

Although we did just successfully convert from CAD to USD, we have a lot of variables swirling around our environment that can quickly turn our code into spaghetti. To get out of our impending pasta doom, we should encapsulate all of our logic into a Python class.

```
class CurrencyConverter:

    def __init__(self, symbols, API_KEY):

        self.API_KEY = API_KEY
        self.symbols = symbols
        self._symbols = ','.join([str(s) for s in symbols])

        r = requests.get(
            'https://openexchangerates.org/api/latest.json',
            params = {
                'app_id': self.API_KEY,
                'symbols': self._symbols,
                'show_alternative': 'true'
                }
            )
```

```
        self.rates_ = r.json()['rates']
        self.rates_['USD'] = 1

    def convert(self, value, symbol_from, symbol_to, round_output=True):

        try:
            x = (value
                * 1/self.rates_.get(symbol_from)
                * self.rates_.get(symbol_to))
            if round_output:
                return round(x, 2)
            else:
                return x
        except TypeError:
            print('Unavailable or invalid symbol')
            return None
```

You can think of classes in Python as just a nice way to keep all our code together and a bit more legible.

Most everything in the `CurrencyConverter` class should look familiar. The only new bits are the following:

```
self._symbols = ','.join([str(s) for s in symbols])
```

This takes a list like `['CAD', 'USD']` and turns it into the comma-separated format required by the API and some error handling in the `.convert` method (the name for a function attached to a class).

With everything now inside of a class, we can instantiate a currency converter with this:

```
API_KEY = os.environ.get("OPX_KEY")
c = CurrencyConverter(['CAD', 'USD'], API_KEY)
```

Converting values now just requires us to use the `.convert` method.

```
print(c.convert(3000, 'CAD', 'USD'))
print(c.convert(5000, 'USD', 'CAD'))
```

```
2302.35
6515.08
```

show_alternative

The Open Exchange Rates API is incredibly robust, and it actually includes access points for alternative cryptocurrencies. This means that it's totally legit to instantiate a new `CurrencyConverter` with ETH (Ethereum), BTC (Bitcoin), and DOGE (Dogecoin) on top of CAD and USD.

```
c = CurrencyConverter(['CAD', 'USD', 'DOGE', 'ETH', 'BTC'], API_KEY)
```

With all the currencies stored inside of a dictionary attached to the `CurrencyConverter` object:

```
c.rates_
```

```
{'BTC': 0.00013350885,
 'CAD': 1.303016,
 'DOGE': 289.975486957,
 'ETH': 0.0017451855,
 'USD': 1}
```

we can, again, run the `.convert` method and find out that $3,000 CAD is equal to the following:

```
c.convert(3000, 'CAD', 'DOGE')
```

```
667625.31
```

.apply

The whole point of this chapter was to figure out what the values from previous chapter were in USD instead of CAD. With a working converter, let's load the mining income data and get to it.

```
import pandas as pd

df = pd.read_excel('data/xirr.xlsx', sheet_name="irregular")
df['total'] = df.income + df.expenses
df
```

	date	income	expenses	total
0	2017-01-01	40	-3000	-2960
1	2017-01-25	40	-50	-10
2	2017-02-12	80	-50	30
3	2017-02-14	100	-30	70
4	2017-03-04	100	-20	80
5	2017-04-23	160	-30	130
6	2017-05-07	140	-20	120
7	2017-05-21	140	-40	100
8	2017-06-04	80	-40	40
9	2017-06-19	180	-30	150
10	2017-07-16	360	-40	320
11	2017-08-27	160	-30	130
12	2017-09-24	240	-20	220
13	2017-10-21	420	-50	370
14	2017-11-19	400	-20	380
15	2017-12-03	340	-40	300
16	2017-12-17	360	-40	320
17	2017-12-31	540	-40	500

To convert everything at once, we just have to use an anonymous lambda function[6] and nest our converter inside of an `.apply` call.

[6] https://stackoverflow.com/questions/16501/what-is-a-lambda-function#16509

```
df['total'].apply(lambda x: c.convert(x, 'CAD', 'USD'))
0    -2271.65
1       -7.67
2       23.02
3       53.72
4       61.40
5       99.77
6       92.09
7       76.75
8       30.70
9      115.12
10     245.58
11      99.77
12     168.84
13     283.96
14     291.63
15     230.24
16     245.58
17     383.73
Name: total, dtype: float64
```

And if we want to take a page of Xzibit's book (I couldn't get the rights for the original meme, so please accept this do-it-yourself bargain image), we can convert our Dogecoin mining income back to Dogecoin from CAD.


```
df['total'].apply(lambda x: c.convert(x, 'CAD', 'DOGE'))
0    -658723.64
1      -2225.42
2       6676.25
3      15577.92
4      17803.34
5      28930.43
6      26705.01
7      22254.18
8       8901.67
```

```
9      33381.27
10     71213.37
11     28930.43
12     48959.19
13     82340.45
14     84565.87
15     66762.53
16     71213.37
17    111270.88
Name: total, dtype: float64
```

Conclusion

We kicked off this chapter by converting CAD to USD with Google. In working through the Python examples, you should be starting to see how powerful the language can be. Just imagine trying to do everything that we just did in Excel (I can't!).

CHAPTER 4

Amortize

> Yeah, I'm paid, and I don't got a debt (hah).
>
> —Migos

I have a personal aversion to debt. It seems like a lot of Millennials do. Even though I try to avoid it, I can appreciate that debt can be a fantastic tool, if (and when) it's used correctly. It's just that…most people don't use it correctly.

In my opinion, debt is used incorrectly when you buy stuff that won't help you get back out of debt. A vacation is a good example. However, if you take on debt to buy something that will propel you forward and will help you pay it back, debt can super powerful.

A couple of chapters ago we talked about Dogecoin mining. Implicit in the example was the concept of spending money to make money. The chapter assumed that a decent mining rig would set us back about $3,000 CAD (or as we learned previous chapter, around $2,341 USD). Most people don't have that kind of money sitting liquid.

If you have only a couple hundred dollars to your name, the Dogecoin game is out of the question, unless, of course, you decide to go into debt.

(If you're sick of Dogecoin at this point, pretend that you need $3,000 for a new laptop!)

Quick disclaimer: debt is a super-complex subject. To narrow the focus for this book, we'll just explore a personal loan.

M. Humber, *Personal Finance with Python*, https://doi.org/10.1007/978-1-4842-3802-8_4

Banks

When it comes to the amount of total interest that you will pay on a personal loan, banks often obfuscate the details.

Let's say we need $3,000 for our Dogecoin rig. We shop around and come back with three viable options from Pineapple Imperial Bank, Orange National Bank, and Banana Dominion Bank.

Our decision will, of course, be dependent on the monthly payment we can realistically afford; however, let's eschew that constraint for a moment and evaluate things on a total interest basis.

Here are the options: Pineapple Imperial is offering us $3,000 at an interest rate of 5.75 percent for 14 months. Orange National will do $3,000 at 3.99 percent for 20 months. And, Banana Dominion is prepared to lend us what we need at 8.99 percent for 8 months. Notice that each bank lists only the term length and the interest rate.

If the term lengths were all the same, Orange National would be a no-brainer. But given that each option has a different term length, how do we choose between them? To figure out which option is the least expensive on a total interest cost basis, we need to turn to amortization schedules.

Amortization

An *amortization schedule* is a table that details each periodic payment on an amortizing loan (like a personal loan). *Amortization* refers to the process of paying off a loan over time through regular payments. On an amortizing loan, a portion of each payment is for interest, while the remaining amount is applied against the principal balance.

Payment

Because each of the three options are fixed-payment personal loans, we can calculate the monthly payments with the following formula:

$$P = \frac{r(PV)}{1-(1+r)-n}$$

where:

```
P = Payment
PV = Present value (the loan)
r = Rate per period
n = Number of periods
```

If we use the Pineapple Imperial loan to start, we can apply the payment formula in Python with the following code block:

```
loan = 3000.00
rate = 0.0575
term = 14

payment = loan * (rate / 12) / (1 - (1 + (rate / 12))**(-term))
print(round(payment, 2))

222.07
```

There are a couple of things to note. First, the double asterisk (**) is Python notation for exponentiation. Second, we divide the (interest) rate by 12 because most banks charge interest on a monthly basis (and there are 12 months in a year).

If you can't be bothered to remember the payment formula for an amortizing loan, numpy has a handy `pmt` function that can do it for you.

```
import numpy as np
payment = np.round(-np.pmt(rate/12, term, loan), 2)
print(payment)
```

```
222.07
```

Loop A

With the payment calculated, we can build an amortization schedule on top of a pandas DataFrame. To start, let's instantiate the first row of the schedule.

```
import pandas as pd

balance = loan
df = pd.DataFrame({
    'month': [0],
    'payment': [np.NaN],
    'interest': [np.NaN],
    'principal': [np.NaN],
    'balance': [balance]
})
print(df)
```

```
   balance  interest  month  payment  principal
0   3000.0       NaN      0      NaN        NaN
```

While the first row (at index 0) has the balance set to the amount borrowed, the interest, payment, and principal (remaining) are all set to NaN (not a number) because loan payments kick in after a full month has elapsed.

With our DataFrame instantiated, we can now calculate the interest and principal portions for the first payment by running the following:

```
interest = round(rate/12 * balance, 2)
principal = payment - interest
balance = balance - principal

print(interest)
print(principal)
print(balance)
```

```
14.38
207.69
2792.31
```

To execute these calculations for each payment in sequence, we can wrap the logic in a loop and append the calculated values to the `df` object with month 0 already filled in.

```
balance = loan

for i in range(1, term + 1):
    interest = round(rate/12 * balance, 2)
    principal = payment - interest
    balance = balance - principal

    df = df.append(
        pd.DataFrame({
            'month': [i],
            'payment': [payment],
            'interest': [interest],
```

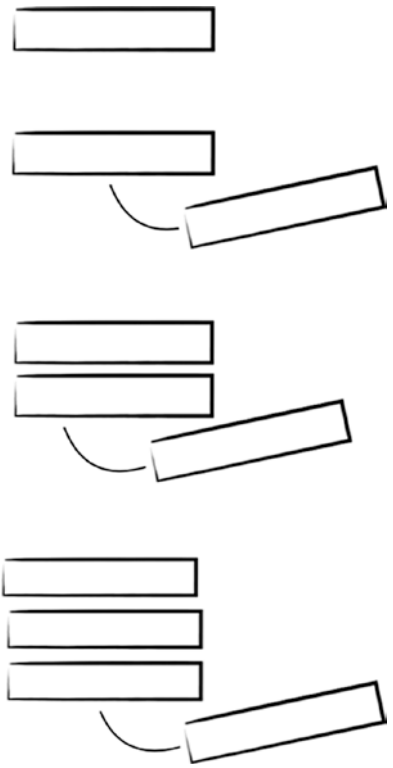
```
            'principal': [principal],
            'balance': [balance]
        })
    )

df = df.reset_index(drop=True)
df[['month', 'payment', 'interest', 'principal', 'balance']]
```

	month	payment	interest	principal	balance
0	0	NaN	NaN	NaN	3000.00
1	1	222.07	14.38	207.69	2792.31
2	2	222.07	13.38	208.69	2583.62
3	3	222.07	12.38	209.69	2373.93
4	4	222.07	11.38	210.69	2163.24
5	5	222.07	10.37	211.70	1951.54
6	6	222.07	9.35	212.72	1738.82
7	7	222.07	8.33	213.74	1525.08
8	8	222.07	7.31	214.76	1310.32
9	9	222.07	6.28	215.79	1094.53
10	10	222.07	5.24	216.83	877.70
11	11	222.07	4.21	217.86	659.84
12	12	222.07	3.16	218.91	440.93
13	13	222.07	2.11	219.96	220.97
14	14	222.07	1.06	221.01	-0.04

Yay, our amortization logic worked! (We're a few pennies off zero, but that's okay because we had to round the payment to two decimal places because there's no such thing as a fractional penny.)

Though we did just successfully build an amortization schedule in Python and pandas, the loop strategy that we used isn't exactly super efficient.

What we just did basically looks like this:

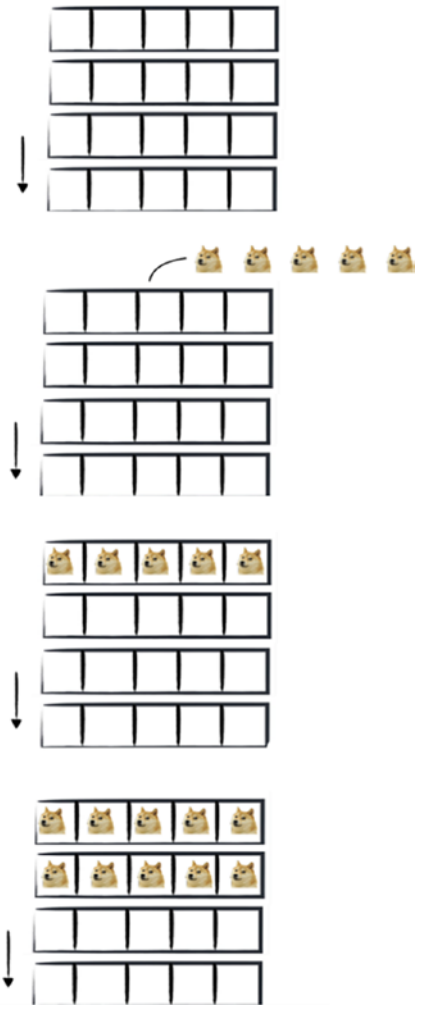


While appending rows onto rows is fine for when we have to loop through only a couple of items, if and when faced with several hundred thousand (or even millions of) rows, things will quickly slow to a crawl.

Loop B

An alternative (and more efficient) way to construct an amortization schedule with a loop involves building all the rows that we need before we iterate instead of appending them at runtime.

Building everything (or pre-allocating space) at the beginning allows us to achieve some pretty incredible speed gains and kind of looks like this:



With all of the rows prebuilt, we basically fill them up with values as we calculate them.

Adjusting our code to fit this pre-allocation pattern is pretty easy. All we have to do is build out an empty DataFrame with 5 columns and 15 rows (15 because the example requires 14 months plus a zeroth month).

```
balance = loan
index = range(0, term + 1)
columns = ['month', 'payment', 'interest', 'principal', 'balance']
df = pd.DataFrame(index=index, columns=columns)
```

Instantiating the first row can be achieved with the `.iloc` method from pandas.

```
df.iloc[0]['month'] = 0
df.iloc[0]['balance'] = balance
```

Running the actual loop is accomplished with `df.iloc[i][COLUMN]` for each column.

I've stopped the loop prematurely so that we can see what's happening at each step.

```
for i in range(1, 11):
    interest = round(rate/12 * balance, 2)
    principal = payment - interest
    balance = balance - principal

    df.iloc[i]['month'] = i
    df.iloc[i]['payment'] = payment
    df.iloc[i]['interest'] = interest
    df.iloc[i]['principal'] = principal
    df.iloc[i]['balance'] = balance

df
```

	month	payment	interest	principal	balance
0	0	NaN	NaN	NaN	3000
1	1	222.07	14.38	207.69	2792.31
2	2	222.07	13.38	208.69	2583.62
3	3	222.07	12.38	209.69	2373.93
4	4	222.07	11.38	210.69	2163.24
5	5	222.07	10.37	211.7	1951.54
6	6	222.07	9.35	212.72	1738.82
7	7	222.07	8.33	213.74	1525.08
8	8	222.07	7.31	214.76	1310.32
9	9	222.07	6.28	215.79	1094.53
10	10	222.07	5.24	216.83	877.7
11	NaN	NaN	NaN	NaN	NaN
12	NaN	NaN	NaN	NaN	NaN
13	NaN	NaN	NaN	NaN	NaN
14	NaN	NaN	NaN	NaN	NaN

At index 10 the remaining balance of the loan is $877, and we have four more payments to make. Without truncating the loop at `range(1, 11)`, the amortization schedule will continue to run through month 14 and will generate the same schedule as before.

Functionize

Now that we have the efficient amortization logic assembled, let's wrap our code into a function called `am`.

```
def am(loan, rate, term):

    payment = np.round(-np.pmt(rate/12, term, loan), 2)
    balance = loan

    index = range(0, term + 1)
    columns = ['month', 'payment', 'interest', 'principal', 'balance']
    df = pd.DataFrame(index=index, columns=columns)

    df.iloc[0]['month'] = 0
    df.iloc[0]['balance'] = balance

    for i in range(1, term + 1):
        interest = round(rate/12 * balance, 2)
        principal = payment - interest
        balance = balance - principal

        df.iloc[i]['month'] = i
        df.iloc[i]['payment'] = payment
        df.iloc[i]['interest'] = interest
        df.iloc[i]['principal'] = principal
        df.iloc[i]['balance'] = balance

    return df
```

Evaluate

With `am` defined, we can run the function on top of the Pineapple Imperial (5.75 percent, 14 months), Orange National (3.99 percent, 20 months), and Banana Dominion (8.99 percent, 8 months) loans without repeating ourselves.

```
loan = 3000
pineapple = am(loan, 0.0575, 14)
orange = am(loan, 0.0399, 20)
banana = am(loan, 0.0889, 8)
```

Peering into one of the schedules, we can see everything working as expected.

```
orange
```

	month	payment	interest	principal	balance
0	0	NaN	NaN	NaN	3000
1	1	155.29	9.97	145.32	2854.68
2	2	155.29	9.49	145.8	2708.88
3	3	155.29	9.01	146.28	2562.6
4	4	155.29	8.52	146.77	2415.83
5	5	155.29	8.03	147.26	2268.57
6	6	155.29	7.54	147.75	2120.82
7	7	155.29	7.05	148.24	1972.58
8	8	155.29	6.56	148.73	1823.85
9	9	155.29	6.06	149.23	1674.62
10	10	155.29	5.57	149.72	1524.9
11	11	155.29	5.07	150.22	1374.68
12	12	155.29	4.57	150.72	1223.96
13	13	155.29	4.07	151.22	1072.74
14	14	155.29	3.57	151.72	921.02
15	15	155.29	3.06	152.23	768.79
16	16	155.29	2.56	152.73	616.06
17	17	155.29	2.05	153.24	462.82
18	18	155.29	1.54	153.75	309.07
19	19	155.29	1.03	154.26	154.81
20	20	155.29	0.51	154.78	0.03

Because each object is an amortization schedule inside of a pandas DataFrame, we can access the interest column like this:

```
banana['interest']
```

```
0      NaN
1    22.23
2    19.52
3    16.79
4    14.04
5    11.28
6     8.49
7     5.68
8     2.85
Name: interest, dtype: object
```

We sum everything up with a call to `.sum()`.

```
banana['interest'].sum()
```

```
100.88
```

We sum up the interest for each loan.

```
print(banana['interest'].sum())
print(orange['interest'].sum())
print(pineapple['interest'].sum())
```

```
100.88
105.82999999999997
108.93999999999998
```

We can see that the Banana Dominion Bank loan is the best bang for our buck (if by bang we mean the least amount of interest paid).

However, it's important to note that the monthly payment for the Banana Dominion option is a lot more than the Orange National loan. Because of this, we might decide that an extra $5, for what would ultimately be less of a monthly burden, is worth it!

Conclusion

I casually dropped the idea that you should pre-allocate where possible and expected you to believe me on faith. Never believe anyone without evidence!

If you want to see how much quicker Loop B is against Loop A, you can run some `%%timeit` Jupyter magic against the function.

```
%%timeit

am(3000, 0.0575, 14)
```

```
6.9 ms ± 877 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

Unfortunately, we never wrapped the logic for Loop A into a function. So, consider it homework! Once you've functionized Loop A, you can run the same `%%timeit` magic on top of it and see the speed difference for yourself.

Try it with a few rows and then with a whole bunch more. The speed differences become incredible as the number of rows increases!

CHAPTER 5

Budget

> Going broke is not an option, always on that cash flow.
>
> —A$AP Rocky

I've been obsessed with budgeting for as long as I can remember, first with pen and paper, then with Excel, then with R, and now with Python.

There are some wonderful online tools that can help you build a budget; however, I've always found them to be lacking. While most budget apps are great for measuring monthly inflows and outflows, I haven't found them to be particularly great for scenario planning or for capturing the holistic cash flow picture.

When I build a budget, I want to know what my bank balance will be at each moment in time so that I might allocate more money to savings, spend less on booze, or plan when I can take a vacation.

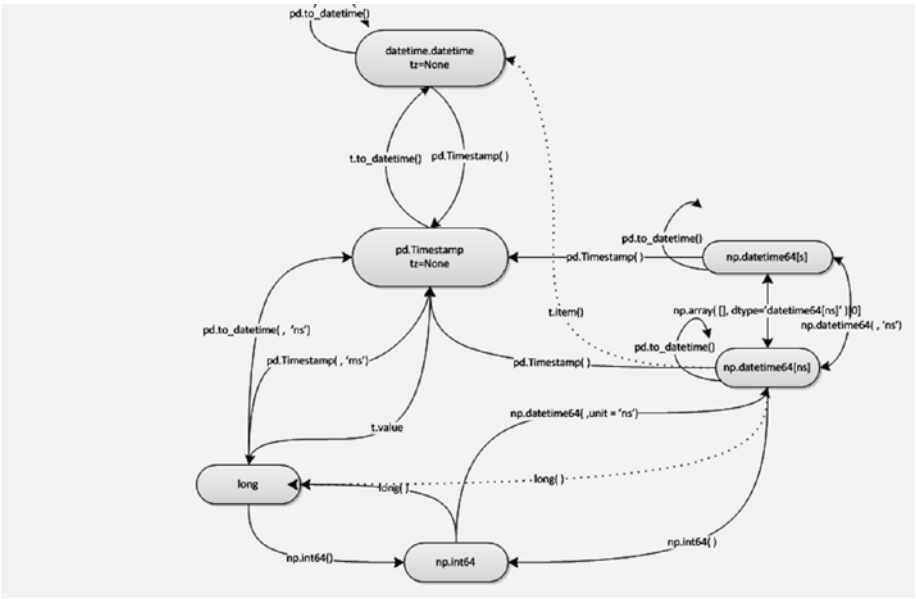
When budgeting, it's important to remember that cash flow is king. If at any moment during the month your budget has you spending more money than you actually have, it really doesn't matter if everything balances at the end of the month. Everything needs to balance every day.

Dates

To build a budget, we have to work with dates. Unfortunately, working with dates is a bit of a nightmare in Python. There are effectively six separate date/time formats in Python that are commonly used, but making each format work in concert is tricky and requires some conversion.

M. Humber, *Personal Finance with Python*, https://doi.org/10.1007/978-1-4842-3802-8_5

Here's a great "state of the union" picture posted on StackOverflow[1] in 2014 (sadly not much has changed in four years):



A full discussion of date formatting is beyond the scope of this book; however, I'm going to enforce two rules that will make our lives tremendously more easy and allow us to sidestep most of the inherent problems in working with dates in Python.

Here are my two rules:

- Coerce everything into a `Timestamp`.
- `.normalize` all the things.

[1] https://i.stack.imgur.com/uiXQd.png

datetime

Python ships with `datetime`. Consequently, we can create `datetime.datetime` objects and `datetime.date` objects out of the box like this:

```
import datetime

date_1 = datetime.datetime.now()
print(date_1)
print(date_1.__repr__())
print(type(date_1))
date_2 = datetime.date.today()
print(date_2)
print(date_2.__repr__())
print(type(date_2))
```

```
2018-05-29 21:34:10.877373
datetime.datetime(2018, 5, 29, 21, 34, 10, 877373)

2018-05-29
datetime.date(2018, 5, 29)
```

`datetime.date` and `datetime.datetime` objects can also be created manually by filling out each object according to the following pattern: `(year, month, day, hour, minute, second, millisecond)`.

```
datetime.datetime(1993, 6, 7, 15, 16, 0)

datetime.datetime(1993, 6, 7, 15, 16)
```

Timestamp

pandas ships with the `Timestamp` type that largely behaves like a `datetime.datetime` object but plays nice with DataFrames and DatetimeIndexes (both of which we'll leverage later in the chapter). Additionally, `Timestamp` comes packaged with a couple of really neat features (namely, `.normalize`).

To convert from a `datetime.datetime` or `datetime.date` object to a `Timestamp`, we can lean on the `to_datetime` function or just wrap our object in a `Timestamp`. Both ways are valid.

```
import pandas as pd

print(pd.Timestamp(date_1))
print(pd.to_datetime(date_1))
```

```
2018-05-29 21:34:10.877373
2018-05-29 21:34:10.877373
```

What's more, we can create `Timestamps` from scratch by manually filling in the function arguments much like we did for `datetime.datetime()`.

```
date_3 = pd.Timestamp(1993, 6, 7, 15, 16, 0)
date_3
```

```
Timestamp('1993-06-07 15:16:00')
```

.normalize

Normalizing in the context of datetimes means stripping all of the time information and just leaving the date bits attached to the object. If everything is in a `Timestamp`, this is trivial:

```
print(date_1)
date_1 = pd.Timestamp(date_1)
print(date_1)
print(date_1.normalize())
```

```
2018-05-29 21:34:10.877373
2018-05-29 21:34:10.877373
2018-05-29 00:00:00
```

As you can see, the `.normalize` method sets the time metadata to 00:00:00, which will help us to glue things together later.

Horizon

To build a budget, we need a time horizon. I like to constrain everything to a year because, honestly, predicting further into the future is a fool's errand.

I'll be referencing the globals defined here for the rest of the chapter, so make sure you run this code:

```
TODAY = pd.Timestamp(today').normalize()
print(TODAY)
END = TODAY + datetime.timedelta(days=365)
print(END)
```

```
2018-05-29 00:00:00
2019-05-29 00:00:00
```

Feel free to arbitrarily adjust the `TODAY` (start) and `END` variables to match your needs.[2]

Just please remember to `.normalize` everything! Otherwise, you're going to smash your head against the wall as we work through all the examples.

After we've defined our start and end dates, we can create an empty calendar object with pandas that leverages a `DatetimeIndex` created by the `date_range` function.

```
calendar = pd.DataFrame(index=pd.date_range(start=TODAY, end=END))
```

[2] https://en.wikipedia.org/wiki/ISO_8601

Peeking inside the DataFrame object with .head(), we can see that there are a bunch of dates that increment by one day.

```
print(calendar.head())
```

```
Empty DataFrame
Columns: []
Index: [2018-05-29 00:00:00, 2018-05-30 00:00:00, 2018-05-31
00:00:00, 2018-06-01 00:00:00, 2018-06-02 00:00:00]
```

The pd.date_range function is super flexible. It can accept different date increments by passing offset aliases[3] to the freq argument.

Two offset aliases that are endlessly useful in budgeting are SM and MS. The former, an alias for "semi-month-end frequency (15th and end of month)," is great for something like income, and the latter, an alias for "month start frequency" is useful for an expense like rent.

Flows

Here are some examples of how to use the offset aliases:

```
# semi-month end frequency (15th and end of month)
print('Semi-month End:')
sm = pd.date_range(start=TODAY, end=END, freq='SM')
print(sm)
print('\n')
# month start frequency
print('Month Start:')
ms = pd.date_range(start=TODAY, end=END, freq='MS')
print(ms)
```

[3] https://pandas.pydata.org/pandas-docs/stable/timeseries.html#offset-aliases

```
Semi-month End:
DatetimeIndex(['2018-05-31', '2018-06-15', '2018-06-30', '2018-07-15',
               '2018-07-31', '2018-08-15', '2018-08-31', '2018-09-15',
               '2018-09-30', '2018-10-15', '2018-10-31', '2018-11-15',
               '2018-11-30', '2018-12-15', '2018-12-31', '2019-01-15',
               '2019-01-31', '2019-02-15', '2019-02-28', '2019-03-15',
               '2019-03-31', '2019-04-15', '2019-04-30', '2019-05-15'],
              dtype='datetime64[ns]', freq='SM-15')

Month Start:
DatetimeIndex(['2018-06-01', '2018-07-01', '2018-08-01', '2018-09-01',
               '2018-10-01', '2018-11-01', '2018-12-01', '2019-01-01',
               '2019-02-01', '2019-03-01', '2019-04-01', '2019-05-01'],
              dtype='datetime64[ns]', freq='MS')
```

If we want to map earnings ($1,000 bimonthly, for example) onto our calendar object, we can create a new DataFrame with an index that spans the entire year and that has `freq='SM'`.

```
income = pd.DataFrame(
    data={'income': 1000},
    index=pd.date_range(start=TODAY, end=END, freq='SM')
)
print(income.head())
```

```
            income
2018-05-31    1000
2018-06-15    1000
2018-06-30    1000
2018-07-15    1000
2018-07-31    1000
```

Similarly, for rent ($1,500 per month, for instance), we can wrap a DataFrame around a `DatetimeIndex` (created by the convenient `date_range` function) with `freq='MS'`.

```
rent = pd.DataFrame(
    data={'rent': -1500},
    index=pd.date_range(start=TODAY, end=END, freq='MS')
)
print(rent.head())
```

```
            rent
2018-06-01 -1500
2018-07-01 -1500
2018-08-01 -1500
2018-09-01 -1500
2018-10-01 -1500
```

Now that we have an empty calendar object with each date for an entire year and two "cash flow" objects (income and rent), we can stitch these DataFrames together with `pd.concat`. If your `Timestamps` aren't normalized, this won't work.

```
calendar = pd.concat([calendar, income], axis=1).fillna(0)
calendar = pd.concat([calendar, rent], axis=1).fillna(0)
calendar.head(5)
```

	income	rent
2018-05-29	0.0	0.0
2018-05-30	0.0	0.0
2018-05-31	1000.0	0.0
2018-06-01	0.0	-1500.0
2018-06-02	0.0	0.0

Note the following:

- `axis` is set to `1` in `pd.concat` to tell pandas that we want to stitch (concatenate) along the column's axis.
- The `.fillna(0)` method is called after each concatenate operation to fill NaN values with 0. (Because we pay rent on the first of each month, the cell for rent at index 2018-XX-02 will be a NaN value.)

Peeking inside this "filled-in" `calendar` object with `.loc` (a label-location-based indexer method from pandas), we can see that concatenations worked!

```
calendar.loc[
    (calendar.index >= '2019-01-30') &
    (calendar.index <= '2019-02-02')
]
```

	income	rent
2019-01-30	0.0	0.0
2019-01-31	1000.0	0.0
2019-02-01	0.0	-1500.0
2019-02-02	0.0	0.0

Totals

Now, to get a holistic picture of our cash flows for each moment in time, we just have to total up the income and rent columns for each day and compute the running total with `cumsum()`.

```
calendar['total'] = calendar.sum(axis=1)
calendar['cum_total'] = calendar['total'].cumsum()
calendar.tail(1)
```

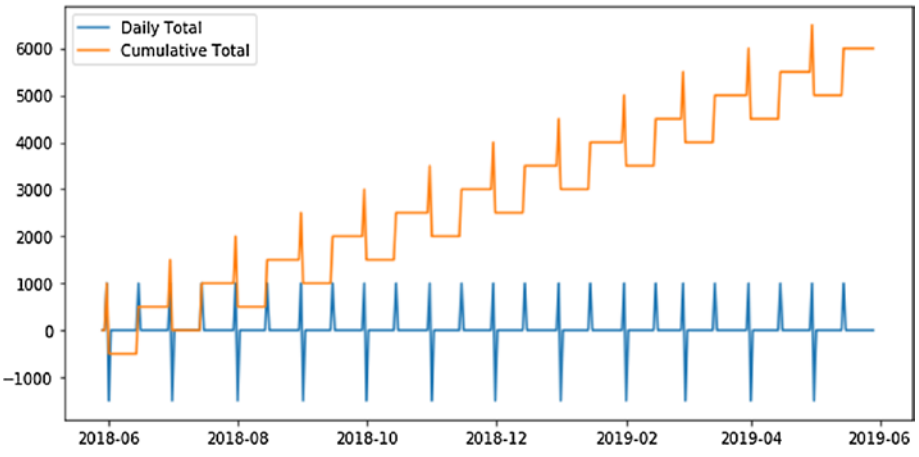
	income	rent	total	cum_total
2019-05-29	0.0	0.0	0.0	6000.0

Running the `.tail` method on the calendar object, we can see that we're meant to end the year with $6,000 in the bank. Great! However, we can't celebrate just yet because there's a problem lurking in our budget that can be unearthed with some plotting.

Visualization

Running some vanilla matplotlib on top of our total and cum_total columns yields the following:

```
from matplotlib import pyplot as plt
%matplotlib inline

plt.figure(figsize=(10, 5))
plt.plot(calendar.index, calendar.total, label='Daily Total')
plt.plot(calendar.index, calendar.cum_total, label='Cumulative
Total')
plt.legend()
```



Although our budget has us ending the year with $6,000 in the bank, there's a moment near the start of the year where we will be forced into overdraft because of how our rent outflows stack against our income inflows. Again, your graph might be different depending on your starting date.

Updating

Because we're working with a toy example, we can just wave a magic wand over the problem and add $2,000 to our starting bank account balance (if only life were that easy).

```
bank = pd.DataFrame(
    data={'bank': 2000},
    index=pd.date_range(start=TODAY, end=TODAY)
)
print(bank)
```

```
            bank
2018-05-29  2000
```

```
calendar = pd.concat([calendar, bank], axis=1).fillna(0)
```

That should fix things! Now we have to recalculate our totals.

Unfortunately, there's another problem. If we run `.sum` on the entire DataFrame, it will add the totals to the totals and our daily values will get wacky.

```
calendar.sum(axis=1).head()
```

```
2018-05-29    2000.0
2018-05-30       0.0
2018-05-31    3000.0
2018-06-01   -3500.0
2018-06-02    -500.0
Freq: D, dtype: float64
```

To prevent this from happening, we can either drop the total and cum_total columns or set them to zero.

I'll demonstrate the first option. And because we're going to be doing this a couple more times, it probably makes sense to wrap up the logic in a function.

```
def update_totals(df):
    # check to see if these columns exit in our dataframe
    if df.columns.isin(['total', 'cum_total']).any():
        # if they do exist set the them to 0
        df['total'] = 0
        df['cum_total'] = 0
    # recalculate total and cumulative_total
    df['total'] = df.sum(axis=1)
    df['cum_total'] = df['total'].cumsum()
    return df
```

Note DataFrames kind of behave like lists/sets/dictionaries, so we can check column membership with a simple `in` call.

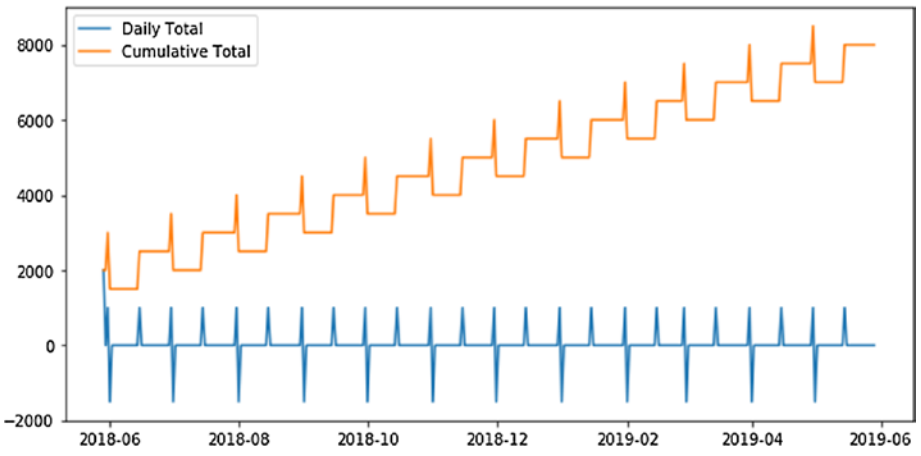
```
calendar = update_totals(calendar)

calendar.tail(1)
```

	income	rent	total	cum_total	bank
2019-05-29	0.0	0.0	0.0	8000.0	0.0

Great! Now we're meant to end the year on $8,000.

Let's plot our calendar object again to make sure that we stay positive for the entire year. And while we're at it, we might as well capture our plotting logic in a function.

```
def plot_budget(df):
    plt.figure(figsize=(10, 5))
    plt.plot(df.index, df.total, label='Daily Total')
    plt.plot(df.index, df.cum_total, label='Cumulative Total')
    plt.legend()
plot_budget(calendar)
```



Problems solved! We're cash flow positive, and our totals aren't wonky.

Vacation I

Given that we're cash flow positive and on an upward trajectory for the entire year, let's go on vacation! What do you think about spending $2,500 at the beginning of July (or 2 months from your TODAY date) for a trip to Colombia? Sounds good to me! We can add a vacation to our budget in the same fashion as rent or income.

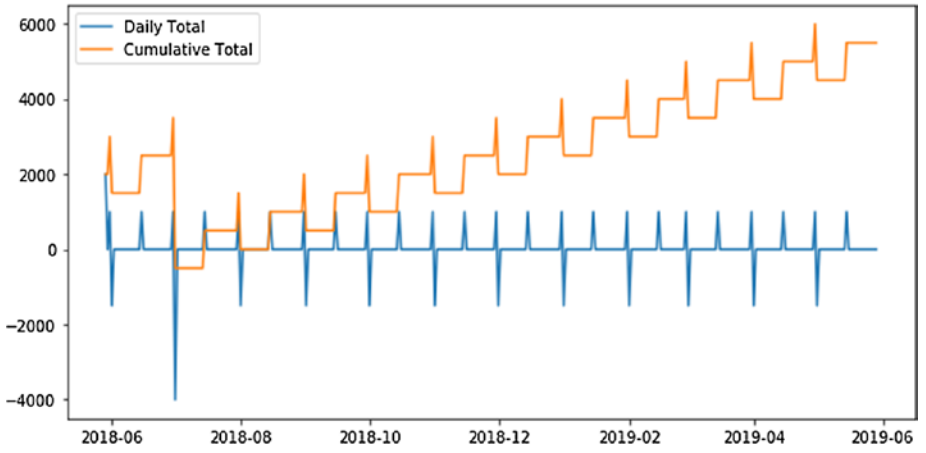
```
vacation = pd.DataFrame(
    data={'vacation': -2500},
    index=[pd.Timestamp('2018-07-01').normalize()]
)
print(vacation)
```

```
            vacation
2018-07-01     -2500
```

```
calendar = pd.concat([calendar, vacation], axis=1).fillna(0)
```

```
calendar = update_totals(calendar)
```

```
plot_budget(calendar)
```



Whoa. Hold up. If we spend $2,500 in July, we're going to get wiped out.

I think we should postpone our imaginary vacation by a couple of months or figure out how to do Colombia on a shoestring budget.

For now let's drop the vacation and consider some more "pressing" issues.

```
calendar = calendar.drop('vacation', axis=1)
```

English

Cash flows are irregular. We might have expenses that occur on specific days or get paid on increasingly weird schedules. It would be nice to define these frequencies in raw English, instead of having to rely on pandas offset aliases.

Fortunately, there's a wonderful library called Recurrent[4] that can do most of the heavy lifting parsing for us.

For instance, we can define an arbitrary frequency in English.

```
frequency = 'every week until July 10th' # try a couple of
different month-day combinations!
```

We then import the `RecurringEvent` class from `recurrent`.

```
!pip install recurrent
```

```
from recurrent import RecurringEvent
```

We then parse it with the `.parse` method.

```
r = RecurringEvent()
r.parse(frequency)
```

```
'RRULE:INTERVAL=1;FREQ=WEEKLY;UNTIL=20180710'
```

`.parse` will allow us to generate a recurrence rule (called an *rrule*) that is iCalendar RFC[5] compliant. The corresponding rrule will enable us to do a bunch of cool things such as checking for the first occurrence before or after an arbitrary date, calculating how many times an event will occur in a given timeframe, or, for our purposes, generating a schedule of dates.

To handle rrules, we'll need one more import.

```
from dateutil import rrule
```

[4] https://github.com/kvh/recurrent

[5] https://www.ietf.org/rfc/rfc2445.txt

Passing the RFC rrule string from our `RecurrentEvent` to `rrule.rrulestr` will create the rrule object that we need.

```
rr = rrule.rrulestr(r.get_RFC_rrule())
rr
```

Now that "every week until July 10th" has been transformed into an rrule, we can generate all the dates that will happen between today and a year from today.

```
rr.between(TODAY, END)
```

```
[datetime.datetime(2018, 5, 29, 21, 38, 42),
 datetime.datetime(2018, 6, 5, 21, 38, 42),
 datetime.datetime(2018, 6, 12, 21, 38, 42),
 datetime.datetime(2018, 6, 19, 21, 38, 42),
 datetime.datetime(2018, 6, 26, 21, 38, 42),
 datetime.datetime(2018, 7, 3, 21, 38, 42)]
```

I think that the output of this is hugely impressive. However, the `.between` method breaks both of the date rules we set at the beginning of this chapter.

The output isn't of type `Timestamp`, and the values aren't normalized. Let's fix both problems at the same time with a list comprehension.

```
[pd.to_datetime(date).normalize() for date in rr.between(TODAY, END)]
```

```
[Timestamp('2018-05-29 00:00:00'),
 Timestamp('2018-06-05 00:00:00'),
 Timestamp('2018-06-12 00:00:00'),
 Timestamp('2018-06-19 00:00:00'),
 Timestamp('2018-06-26 00:00:00'),
 Timestamp('2018-07-03 00:00:00')]
```

get_dates

Now, we could wrap a DatetimeIndex around these dates and put them inside of a DataFrame object. But before we do, let's create a `get_dates` function that outputs a list of normalized dates of type `Timestamp` and that can handle raw dates (like "2018-06-07") and English (like "every week until July 10th"):

```
def get_dates(frequency):
    # let pandas try and handle single dates
    try:
        return [pd.Timestamp(frequency).normalize()]
    except ValueError:
        pass
    # parse frequency with recurrent
    try:
        r = RecurringEvent()
        r.parse(frequency)
        rr = rrule.rrulestr(r.get_RFC_rrule())
        return [
            pd.to_datetime(date).normalize()
            for date in rr.between(TODAY, END)
        ]
    except ValueError as e:
        raise ValueError('Invalid frequency')
```

The logic in this function is mostly identical to the operations that we performed earlier; however, I've added some additional error handling in the pattern of "Ask forgiveness, not permission"[6] that tries to let `pd.Timestamp` do its thing with raw dates and then falls back on `.parse` from the Recurrent library.

[6] https://stackoverflow.com/questions/12265451/ask-forgiveness-not-permission-explain

Let's take our new function for a spin.

```
get_dates('2019-01-01')
```

```
[Timestamp('2019-01-01 00:00:00')]
```

```
get_dates('every week until July 10th')
```

```
[Timestamp('2018-05-29 00:00:00'),
 Timestamp('2018-06-05 00:00:00'),
 Timestamp('2018-06-12 00:00:00'),
 Timestamp('2018-06-19 00:00:00'),
 Timestamp('2018-06-26 00:00:00'),
 Timestamp('2018-07-03 00:00:00')]
```

```
get_dates('this will not work')
---------------------------------------------------------------
ValueError                    Traceback (most recent call last)
                              in get_dates(frequency)
     10         r.parse(frequency)
---> 11         rr = rrule.rrulestr(r.get_RFC_rrule())
     12         return [pd.to_datetime(date).normalize() for
                date in rr.between(TODAY, END)]

ValueError: not enough values to unpack (expected 2, got 1)

 in get_dates(frequency)
     12         return [
     13              pd.to_datetime(date).normalize()
     14              for date in rr.between(TODAY, END)
     15         ]
     16     except ValueError as e:
---> 17         raise ValueError('Invalid frequency')

ValueError: Invalid frequency
```

It looks like everything seems to be behaving as expected!

Fun

Our budget is still fairly simplistic. Honestly, living exactly according to it doesn't inspire much fun. In this toy example, we work (to generate an income) to pay off our rent, and, well, that's pretty much it. Plus, we had to scrap a vacation because we couldn't afford it.

Let's fix our bleak imaginary lives in this section by embedding some nights out ($40 each Friday and Saturday).

```
dates = get_dates('every week on Friday and Saturday')
dates[:10] # first ten instances of the recurrance rule
[Timestamp('2018-06-01 00:00:00'),
 Timestamp('2018-06-02 00:00:00'),
 Timestamp('2018-06-08 00:00:00'),
 Timestamp('2018-06-09 00:00:00'),
 Timestamp('2018-06-15 00:00:00'),
 Timestamp('2018-06-16 00:00:00'),
 Timestamp('2018-06-22 00:00:00'),
 Timestamp('2018-06-23 00:00:00'),
 Timestamp('2018-06-29 00:00:00'),
 Timestamp('2018-06-30 00:00:00')]
```

To wrap these dates into a DataFrame object, we have to turn the list into a `DatetimeIndex` object. The only problem is that `pd.DatetimeIndex` accepts only a pandas `Series`. But it's actually no problem!

```
pd.Series(dates).head()
```

```
0   2018-06-01
1   2018-06-02
2   2018-06-08
3   2018-06-09
4   2018-06-15
dtype: datetime64[ns]
```

```
pd.DatetimeIndex(pd.Series(dates))
```

```
DatetimeIndex(['2018-06-01', '2018-06-02', '2018-06-08', '2018-06-09',
               '2018-06-15', '2018-06-16', '2018-06-22', '2018-06-23',
               '2018-06-29', '2018-06-30',
               ...
               '2019-04-26', '2019-04-27', '2019-05-03', '2019-05-04',
               '2019-05-10', '2019-05-11', '2019-05-17', '2019-05-18',
               '2019-05-24', '2019-05-25'],
              dtype='datetime64[ns]', length=104, freq=None)
```

If we wrap our list of Timestamps in a Series in a DatetimeIndex, we can follow largely the same pattern that we've been using for the entire chapter.

```
dates = get_dates('every week on Friday and Saturday')
```

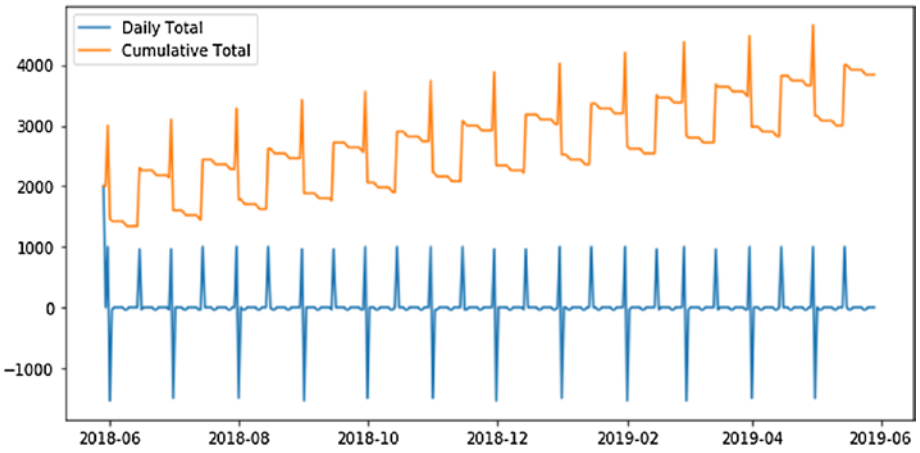
```
fun = pd.DataFrame(
    data={'fun': -40},
    index=pd.DatetimeIndex(pd.Series(dates))
)
```

```
print(fun.head())
```

```
            fun
2018-06-01  -40
2018-06-02  -40
2018-06-08  -40
2018-06-09  -40
2018-06-15  -40
```

Stitching this `fun` object onto our `calendar` object is a cinch.

```
calendar = pd.concat([calendar, fun], axis=1).fillna(0)
calendar = update_totals(calendar)
plot_budget(calendar)
```




```
calendar.tail()
```

	income	rent	total	cum_total	bank	fun
2019-05-25	0.0	0.0	-40.0	3840.0	0.0	-40.0
2019-05-26	0.0	0.0	0.0	3840.0	0.0	0.0
2019-05-27	0.0	0.0	0.0	3840.0	0.0	0.0
2019-05-28	0.0	0.0	0.0	3840.0	0.0	0.0
2019-05-29	0.0	0.0	0.0	3840.0	0.0	0.0

It looks like our budget can stomach a little weekend fun!

YAML

Although it's worked so far, adding a new outflow/inflow to our budget is a bit of an arduous process. We have to first generate the DataFrame object, stitch it onto our main `calendar` object, and then update the totals. It would be nice to define our budget items all at once and calculate the totals just once.

YAML to the rescue! YAML is a file format that's a bit like JSON but is slightly more human-readable. We can take all of our budget items and stuff it into a YAML file like this:

Note PyYAML should have been installed with Anaconda. If not, try the following:

```
!pip install pyyaml
```

```
import yaml

budget = yaml.load('''
bank:
    frequency: today
    amount: 2000
income:
    frequency: every 2 weeks on Friday
    amount: 1000
rent:
    frequency: every month
    amount: -1500
fun:
    frequency: every week on Friday and Saturday
    amount: -40
''')
```

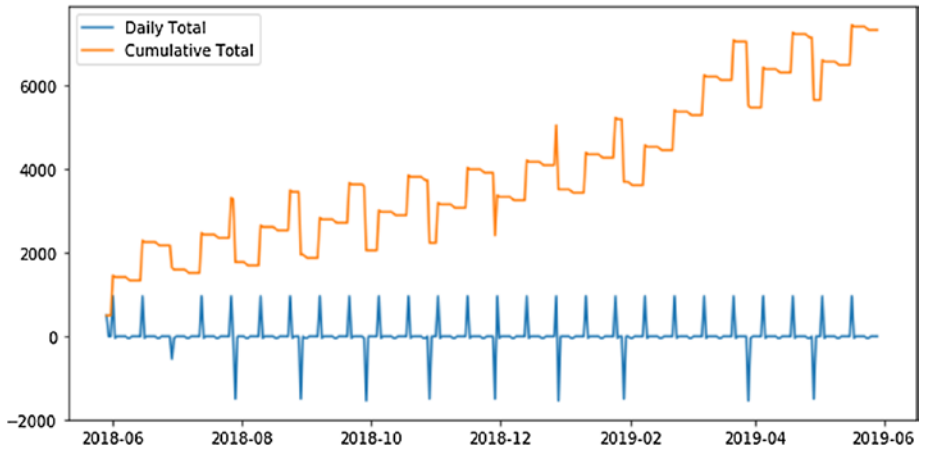
After running `yaml.load`, this just turns everything into a Python `dict`.

```
budget
```

```
{'bank': {'amount': 2000, 'frequency': 'today'},
 'fun': {'amount': -40, 'frequency': 'every week on Friday and
 Saturday'},
 'income': {'amount': 1000, 'frequency': 'every 2 weeks on Friday'},
 'rent': {'amount': -1500, 'frequency': 'every month'}}
```

With our budget in a dictionary, we can now iterate through all of the keys (`k`) and values (`v`), get the frequency and amounts attached to each `v`, create a DataFrame object, and concatenate it to the main `calendar` object. After this point we can compute the totals without having to update them after each step.

```
calendar = pd.DataFrame(index=pd.date_range(start=TODAY, end=END))

for k, v in budget.items():
    frequency = v.get('frequency')
    amount = v.get('amount')
    dates = get_dates(frequency)
    i = pd.DataFrame(
        data={k: amount},
        index=pd.DatetimeIndex(pd.Series(dates))
    )
    calendar = pd.concat([calendar, i], axis=1).fillna(0)

calendar['total'] = calendar.sum(axis=1)
calendar['cum_total'] = calendar['total'].cumsum()

plot_budget(calendar)
```



Functionize

Let's quickly functionize our budget logic so that we might test a couple more scenarios.

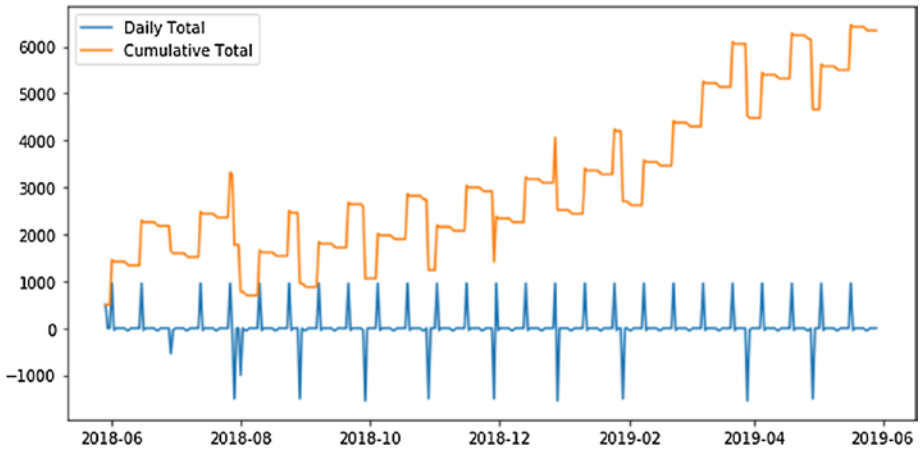
```
def build_calendar(budget):

    calendar = pd.DataFrame(index=pd.date_range(start=TODAY,
    end=END))

    for k, v in budget.items():
        frequency = v.get('frequency')
        amount = v.get('amount')
        dates = get_dates(frequency)
        i = pd.DataFrame(
            data={k: amount},
            index=pd.DatetimeIndex(pd.Series(dates))
        )
        calendar = pd.concat([calendar, i], axis=1).fillna(0)
```

```
    calendar['total'] = calendar.sum(axis=1)

    calendar['cum_total'] = calendar['total'].cumsum()

    return calendar
```

Vacation II

Although we might not be able to stomach $2,500 in Colombia in July, what if we did Halifax in August for $1,000?

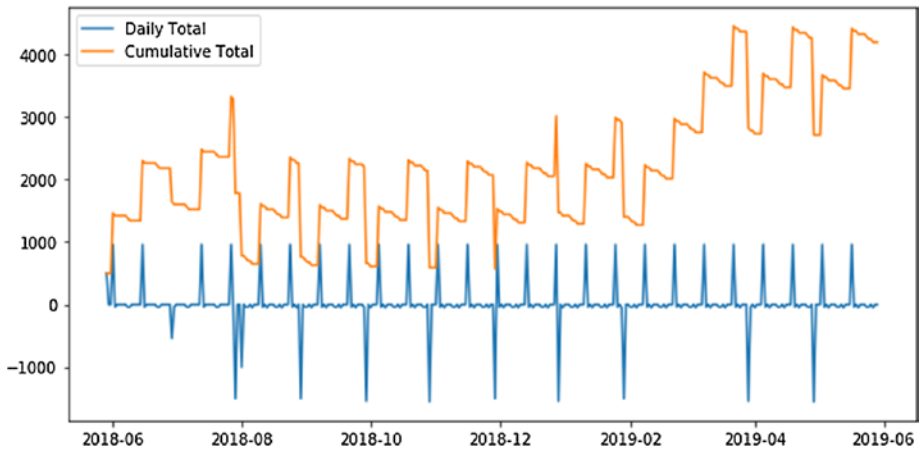
```
budget = yaml.load('''
bank:
    frequency: today
    amount: 2000
income:
    frequency: every 2 weeks on Friday
    amount: 1000
rent:
    frequency: every month
    amount: -1500
fun:
    frequency: every week on Friday and Saturday
    amount: -40
vacation:
    frequency: 2018-08-01
    amount: -1000
''')

calendar = build_calendar(budget)

plot_budget(calendar)
```



Looks good! And, it looks like we might even be able to stow away some money for a rainy day. Let's sketch out what $50 to savings on "every Monday starting in August" would look like.

```
budget = yaml.load('''
bank:
    frequency: today
    amount: 2000
income:
    frequency: every 2 weeks on Friday
    amount: 1000
rent:
    frequency: every month
    amount: -1500
fun:
    frequency: every week on Friday and Saturday
    amount: -40
vacation:
    frequency: 2018-08-01
    amount: -1000
```

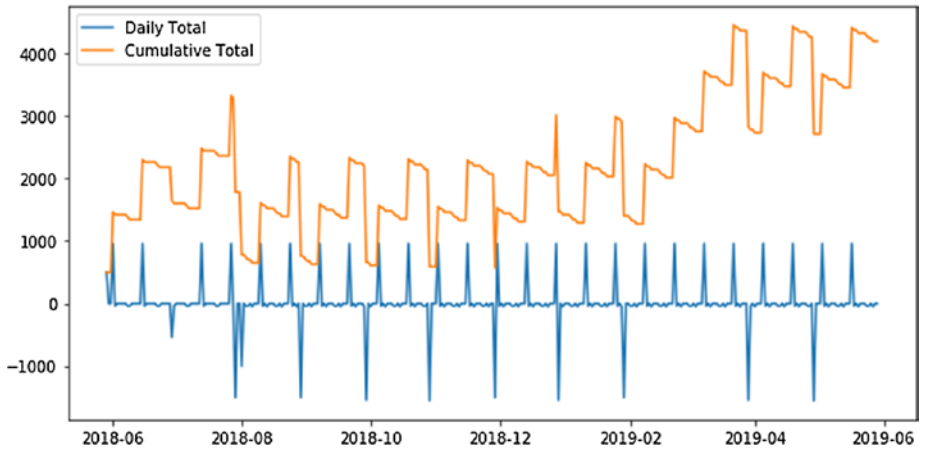
```
savings:
    frequency: every Monday starting in August
    amount: -50
''')

calendar = build_calendar(budget)
plot_budget(calendar)
```



Loading YAML

We have been defining our budget as inline YAML. If you want to keep your inputs separate from your code (which is a good idea), we can adjust our workflow ever so slightly to accommodate.

```
with open('data/budget.yaml', 'r') as f:
    inputs = yaml.load(f)

calendar = build_calendar(budget)
plot_budget(calendar)
```



Using a `with` block to open and close the `.yaml` file is really the only adjustment that we have to make!

Conclusion

I hope you had fun with this chapter. I had a bunch of fun writing it. But a word of caution: although this budget tool can be incredibly powerful, if you do decide to use it, don't be obsessive about it. Update your `.yaml` file every quarter to see where you're at and where you're headed. Otherwise, if you try to update it every day, you're going to go crazy.

CHAPTER 6

Invest

> Invest in your future, don't dilute your finances.
>
> —Kendrick Lamar

I am not an authority on investing. So, I can't (and won't) tell you which stocks you should pick or how you should structure your personal investment portfolio. I can, however, give you an awesome mechanism for setting up asset allocations and adhering to a routine of continuous rebalancing.

Rebalancing[1] your investment portfolio against target allocations is a good idea because it keeps risk in check and forces you to remove the emotion from your investment decisions.

When your portfolio is up, it's really hard to sell, and when you take a hit, it's just as hard to buy. A strategy of automated or semi-automated rebalancing will force you to buy and sell even when your emotions are trying to get the best of you.

But before we explore investing and rebalancing with Python, here are a couple of words on trade-offs.

[1] https://www.moneyunder30.com/rebalance-your-portfolio

M. Humber, *Personal Finance with Python*, https://doi.org/10.1007/978-1-4842-3802-8_6

Trade-Offs

I had a hard time writing the technical parts for this chapter because you can really go hard on all this stuff if you want to (and I encourage you to!).

Initially, I wrote everything so that it was encapsulated in a Python class, then I refactored everything to work with SQLite, and then I refactored it all again to simple functions powered by pandas.

I ultimately settled on pandas and functions because this is a pandas book and it kept with the theme. Besides, functions are just way easier to read and use. Pandas might not be the best choice for scaling things up to production.

There are real trade-offs in programming. Often we have to choose between good enough and heavily polished, with the latter often taking ten times the time and effort of the former. Polished will get you coverage for all the edge cases, but good enough will get you where you need to go fast.

I'm all in for "good enough."

Instantiate

With the preamble out of the way, let's design a portfolio and get investing! I've chosen three random stocks to get us started: Amazon (AMZN[2]), Cisco (CSCO[3]), and General Electric (GE[4]). We'll try to hold the securities in target allocations of 40 percent, 30 percent, and 30 percent, respectively.

To maintain balance against our target allocations, we will need to execute buy and sell trades after market movements, deposits, and withdrawals.

[2] https://finance.yahoo.com/quote/AMZN
[3] https://finance.yahoo.com/quote/CSCO
[4] https://finance.yahoo.com/quote/GE

Let's start by defining our target allocations as a dictionary.

```
targets = {
    'AMZN': 0.40, # Amazon
    'CSCO': 0.30, # Cisco
    'GE': 0.30 # GE
}
```

To instantiate a portfolio, we might build up a DataFrame from scratch like this:

```
import pandas as pd
import numpy as np

portfolio = pd.DataFrame(
    index=list(targets.keys()) + ['CASH'],
    data={
        'date': '2018-01-01',
        'price': [np.NaN, np.NaN, np.NaN, 1],
        'target': [0.4, 0.3, 0.3, 0],
        'allocation': [0, 0, 0, 1],
        'shares': [0, 0, 0, 10000],
        'market_value': [0, 0, 0, 10000]
    }
)
print(portfolio)
```

```
      allocation        date   market_value  price  shares  target
AMZN          0  2018-01-01              0    NaN       0     0.4
CSCO          0  2018-01-01              0    NaN       0     0.3
GE            0  2018-01-01              0    NaN       0     0.3
CASH          1  2018-01-01          10000    1.0   10000     0.0
```

Alternatively, we can build a reusable function that will work for arbitrary stocks and targets.

```
def instantiate_portfolio(targets, starting_balance):
    targets['CASH'] = 0
    tickers = list(targets.keys())

    df = pd.DataFrame(
        index=tickers,
        columns=[
            'date','price','target',
            'allocation','shares','market_value'
        ]
    )
    df.shares = 0
    df.market_value = 0
    df.allocation = 0
    df.update(
        pd.DataFrame
            .from_dict(targets, orient='index')
            .rename(columns={0:'target'})
    )

    df.at['CASH', 'shares'] = starting_balance

    return df
```

Instantiating a portfolio with this function will look like this:

```
portfolio = instantiate_portfolio(
    {'AMZN': 0.4, 'CSCO': 0.3, 'GE': 0.3},
    10000
)
print(portfolio)
```

```
      date price target allocation shares market_value
AMZN   NaN   NaN    0.4          0      0            0
CSCO   NaN   NaN    0.3          0      0            0
GE     NaN   NaN    0.3          0      0            0
CASH   NaN   NaN      0          0  10000            0
```

Given that I've packed a lot into the `instantiate_portfolio` function, let's spend a few moments unpacking it.

First, you might've noticed that we set `targets['CASH'] = 0`. This is required because cash will behave a little differently from regular securities in our model portfolio.

Though we will rebalance stocks, we don't have to, nor should we, rebalance cash. Cash in our portfolio will just act as a buffer and overflow. Adding `CASH` to our dictionary has this effect:

```
print(targets)
targets['CASH'] = 0
print(targets)
```

```
{'AMZN': 0.4, 'CSCO': 0.3, 'GE': 0.3}
{'AMZN': 0.4, 'CSCO': 0.3, 'GE': 0.3, 'CASH': 0}
```

Second, just after the line where we set the value for the `CASH` key, there's this: `list(targets.keys())`.

This code snippet converts all the dictionary keys to a list so that we can use it as an index to build a pandas DataFrame.

```
list(targets.keys())
```

```
['AMZN', 'CSCO', 'GE']
```

Further down the `instantiate_portfolio` function, we call `.from_dict`. This piece of code simply generates a pandas DataFrame from a dictionary.

```
print(pd.DataFrame.from_dict(targets, orient='index'))
```

```
        0
AMZN  0.4
CSCO  0.3
GE    0.3
```

Unfortunately, the method isn't smart enough to set the column name, so we manually rename the 0 column.

```
print(
    pd.DataFrame
    .from_dict(targets, orient='index')
    .rename(columns={0:'target'})
)
```

```
      target
AMZN     0.4
CSCO     0.3
GE       0.3
```

This DataFrame created by `.from_dict` is used in a call to update the target values at their respective index locations.

The last piece of the `instantiate_portfolio` function worth highlighting is the `.at` method. Using `.at` is a great way to set values at specific index and column locations.

Here's how `.at` works in practice:

```
df = pd.DataFrame(index=['CASH', 'GE'], data={'shares': [0, 1]})
print(df)
df.at['CASH', 'shares'] = 10000
print(df)
```

```
      shares
CASH       0
GE         1
      shares
CASH   10000
GE         1
```

Prices

At this point our portfolio is instantiated, but there are still a lot of gaps in the data structure. Gaps are caused by a lack of prices.

```
print(portfolio)
```

```
     date price target  allocation  shares  market_value
AMZN  NaN   NaN    0.4           0       0             0
CSCO  NaN   NaN    0.3           0       0             0
GE    NaN   NaN    0.3           0       0             0
CASH  NaN   NaN      0           0   10000             0
```

To fill in these gaps, let's build a function with which we can update prices. Although we could theoretically use the `.at` method from before to build this function, the `.update` method is a little more legible here.

```
def update_prices(portfolio, prices):
    prices['CASH'] = 1
    portfolio.update(pd.DataFrame({'price': prices}))
    portfolio.date = prices.name
    portfolio.market_value = portfolio.shares * portfolio.price
```

To use `update_prices`, we pass the function our portfolio and a pandas `Series` object that contains prices for a specific day.

```
# fake for right now
prices = pd.Series(
    name='2018-01-01',
    data={'AMZN': 945.21,'CSCO': 30.52, 'GE': 29.27}
)
print(prices)
update_prices(portfolio, prices)
AMZN    945.21
CSCO     30.52
GE       29.27
Name: 2018-01-01, dtype: float64
```

The neat thing about this function is that it updates the values in place. Running `print(portfolio)` in a Jupyter cell, we can see how the function changed the object.

```
print(portfolio)
```

	date	price	target	allocation	shares	market_value
AMZN	2018-01-01	945.21	0.4	0	0	0
CSCO	2018-01-01	30.52	0.3	0	0	0
GE	2018-01-01	29.27	0.3	0	0	0
CASH	2018-01-01	1	0	0	10000	10000

Right now all the `market_values` are set to 0 because we haven't executed any buy orders yet.

Orders

With our `portfolio` object complete, let's build a `get_order` function that will calculate the buy and sell orders we need to execute to achieve balance against target allocations at arbitrary moments in time.

```
def get_order(portfolio):
    total_value = portfolio.market_value.sum()

    order = (
        (total_value * portfolio.target // portfolio.price)
        - portfolio.shares
    ).drop('CASH')

    return order
```

The get_order function is fairly straightforward; the only interesting part is this thing: //.

The // is the Python floor division operator. get_order uses floor division because we can't buy fractional shares on the stock market!

If we want to buy AMZN, we have to buy whole shares.

```
total_value = 10000
target = 0.4
price = 945.21
AMZN = (total_value * target // price) - 0
print(AMZN)
```

```
4.0
```

Without floor division, the get_order function would tell us to buy fractional units.

```
(total_value * target / price) - 0
```

```
4.231863818622316
```

Using get_order is as simple as passing the function our portfolio object as the sole argument.

```
order = get_order(portfolio)
print(order)
```

```
AMZN      4
CSCO     98
GE      102
dtype: object
```

Importantly, `get_order` doesn't actually do anything to the `portfolio` object. This because we might want to maintain control over whether to place the order and make the necessary trades to rebalance, or not.

Deposit

Because we're disciplined investors (*wink wink*), let's add a `deposit` function into the mix. Most of this should be self-explanatory.

```
def deposit(portfolio, amount):
    portfolio.at['CASH', 'shares'] += amount
    portfolio.at['CASH', 'market_value'] = portfolio.at['CASH',
    'shares']

deposit(portfolio, 1000)
```

Running `get_order` on top of `portfolio` now yields the following:

```
order = get_order(portfolio)
print(order)
```

```
AMZN      4
CSCO    108
GE      112
dtype: object
```

That's pretty much it! With the `instantiate_portfolio`, `update_prices`, `get_order`, and `deposit` functions built, we have all the ingredients we need to rebalance any basket of securities.

It's a bit anticlimatic, isn't it?

Let's keep going and put all our functions into action. While we're at it, we might as well use real stock quotes, instead of faking it like when we tested `update_prices`.

Simulate

Because our `get_order` function doesn't alter the `portfolio` object, let's implement a function that simulates the execution of buy/sell orders.

```
def simulate_process_order(portfolio, order):
    starting_cash = portfolio.at['CASH', 'shares']
    cash_adjustment = np.sum(order * portfolio.price)
    portfolio.shares += order
    portfolio.market_value = portfolio.shares * portfolio.price
    portfolio.at['CASH', 'shares'] = starting_cash - cash_adjustment
    portfolio.market_value = portfolio.shares * portfolio.price
    portfolio.allocation = (
        portfolio.market_value / portfolio.market_value.sum()
    )
```

The `simulate_process_order` function takes the `portfolio` object and a pandas `series` object to buy and sell securities.

```
simulate_process_order(portfolio, order)
print(portfolio)
```

```
            date   price target allocation  shares market_value
AMZN  2018-01-01  945.21    0.4   0.343713       4      3780.84
CSCO  2018-01-01   30.52    0.3   0.299651     108      3296.16
GE    2018-01-01   29.27    0.3   0.298022     112      3278.24
CASH  2018-01-01       1      0  0.0586145  644.76       644.76
```

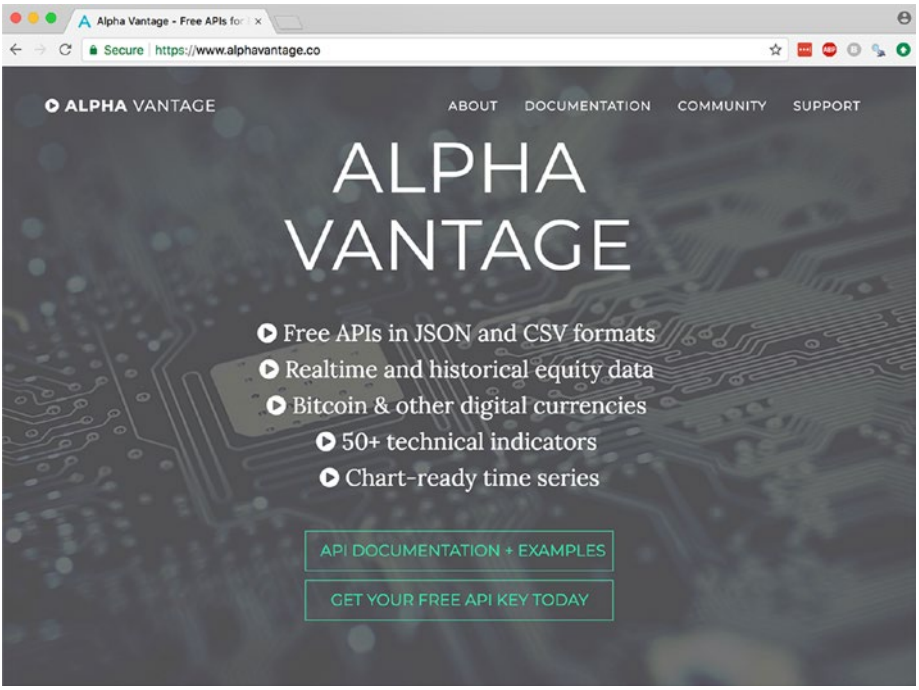
Quotes

Years ago everyone was using the Google and Yahoo Finance APIs to access financial market data. For some reason both companies shuttered access to these services, and ever since we've all been scrambling to find a replacement.

As of this writing, I'm using a free API from Alpha Vantage to access stock quotes. It's a bit clunky, but it's accurate and will be perfect for our purposes.

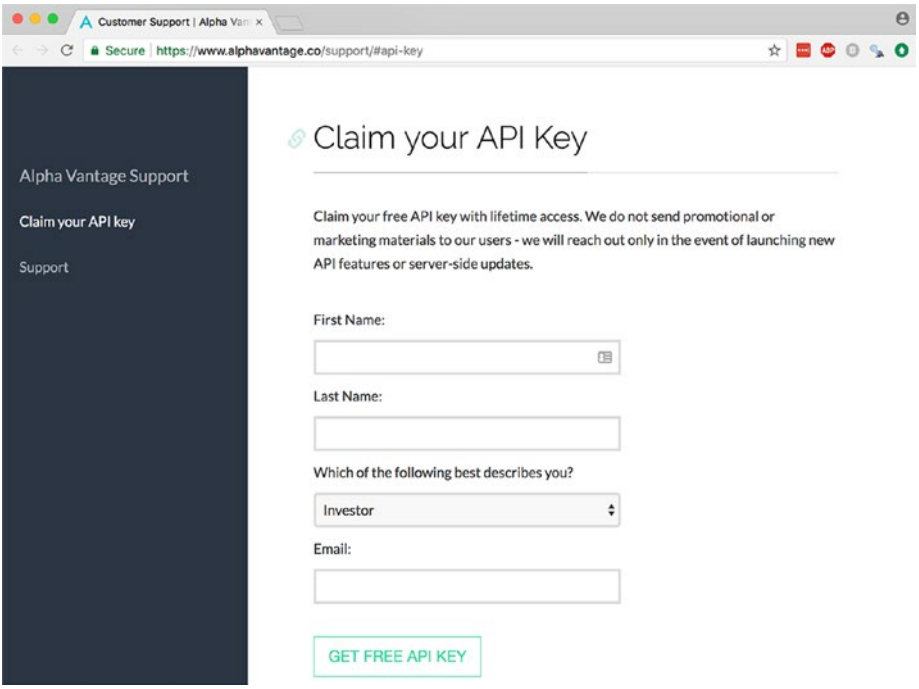
Given that the process for connecting to the Alpha Vantage API closely parallels the process for connecting to the Open Exchange Rates API, I'm going to go through this section a bit more quickly.

Register for an API key at alphavantage.co.[5]

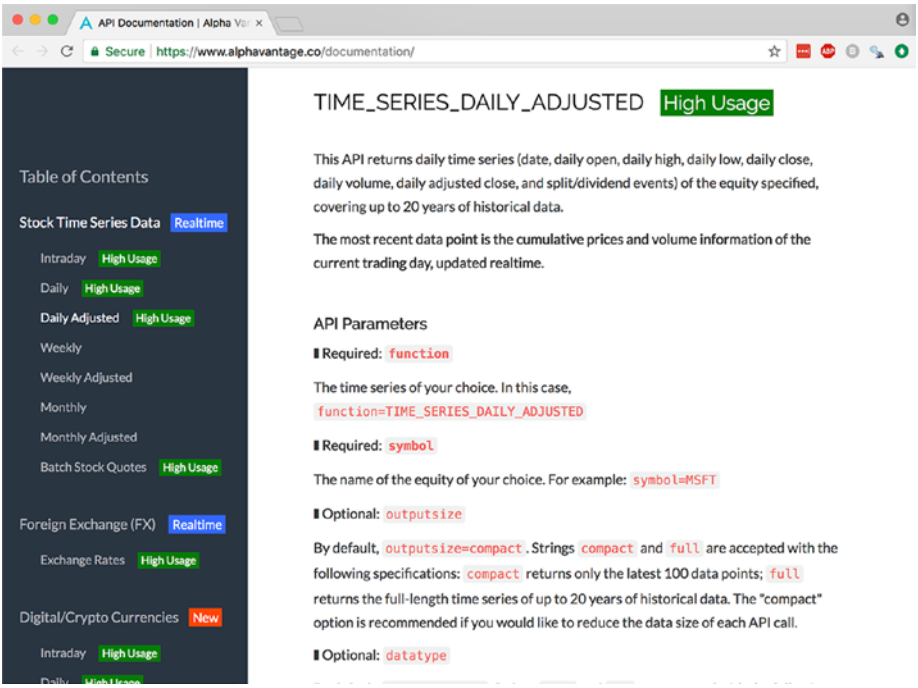


[5] https://www.alphavantage.co/

(You barely have to create an account; it's awesome.)



The API has a couple of different API endpoints. We're going to work with the `TIME_SERIES_DAILY_ADJUSTED` endpoint for this chapter.



Further down the documentation page we can see how requests against the API are supposed to be structured.

Required: `apikey`

Your API key. Claim your free API key here.

Examples (click for JSON output)

`https://www.alphavantage.co/query?function=TIME_SERIES_DAILY_ADJUSTED&symbol=MSFT&apikey=demo`

`https://www.alphavantage.co/query?function=TIME_SERIES_DAILY_ADJUSTED&symbol=MSFT&outputsize=full&apikey=demo`

Downloadable CSV file:

`https://www.alphavantage.co/query?function=TIME_SERIES_DAILY_ADJUSTED&symbol=MSFT&apikey=demo&datatype=csv`

With our key in hand, let's store it in an `.env` file and load everything (if you skipped Chapter 3, I suggest going back and reviewing how to use `.env` files).

```
import requests
from dotenv import load_dotenv, find_dotenv
load_dotenv(find_dotenv())

API_KEY = os.environ.get('AV_KEY')
TODAY = pd.Timestamp.today().normalize()
```

get_price

Now we can write a `get_price` function that hits Alpha Vantage, does a bit of data cleaning, and returns a pandas DataFrame.

```
def get_price(ticker, outputsize='compact', most_recent=False):
    URL = 'https://www.alphavantage.co/query?'
    payload = {
        'function': 'TIME_SERIES_DAILY_ADJUSTED',
        'symbol': ticker,
        'apikey': API_KEY,
        'outputsize': outputsize
    }
    r = requests.get(URL, params=payload)
    p = pd.DataFrame(r.json()['Time Series (Daily)']).T['4. close']
    df = pd.DataFrame({ticker: p.apply(float)})
    df.index = pd.to_datetime(df.index)
    if most_recent:
        return df.tail(1)
    return df

print(get_price('AMZN')[:10])
```

```
              AMZN
2018-01-04  1209.59
2018-01-05  1229.14
2018-01-08  1246.87
2018-01-09  1252.70
2018-01-10  1254.33
2018-01-11  1276.68
2018-01-12  1305.20
2018-01-16  1304.86
2018-01-17  1295.00
2018-01-18  1293.32
```

If we dissect the `requests` call inside the `get_price` function, we get this:

```
URL = 'https://www.alphavantage.co/query?'
payload = {
    'function': 'TIME_SERIES_DAILY_ADJUSTED',
    'symbol': 'AMZN',
    'apikey': API_KEY,
    'outputsize': 'compact'
}
r = requests.get(URL, params=payload)
print(r.json().keys())
```

```
dict_keys(['Meta Data', 'Time Series (Daily)'])
```

We can see that the data that we need will be returned in the Time Series (Daily) key of the JSON object.

```
p = pd.DataFrame(r.json()['Time Series (Daily)'])
print(p.head()[p.columns[:4]])
```

```
                  2018-01-04 2018-01-05 2018-01-08 2018-01-09
1. open           1205.0000  1217.5100  1236.0000  1256.9000
2. high           1215.8700  1229.1400  1253.0800  1259.3300
3. low            1204.6600  1210.0000  1232.0300  1241.7600
4. close          1209.5900  1229.1400  1246.8700  1252.7000
5. adjusted close 1209.5900  1229.1400  1246.8700  1252.7000
```

Unfortunately, the data returned is "wide" instead of "long." Let's tidy it up[6] and transpose it to long with a call to `.T`. At the same time, let's grab just the `'4. close'` column and ignore the rest.

```
p = pd.DataFrame(r.json()['Time Series (Daily)']).T['4. close']
p.head()
```

[6] http://vita.had.co.nz/papers/tidy-data.pdf

```
2018-01-04    1209.5900
2018-01-05    1229.1400
2018-01-08    1246.8700
2018-01-09    1252.7000
2018-01-10    1254.3300
Name: 4. close, dtype: object
```

Because all the values returned are strings, we have to apply a couple more transformations to the object. Let's convert the numbers to floats, stuff it back into a DataFrame, and convert the dates to dates.

```
ticker = 'AMZN'
df = pd.DataFrame({ticker: p.apply(float)})
df.index = pd.to_datetime(df.index)
print(df.head())
```

```
               AMZN
2018-01-04  1209.59
2018-01-05  1229.14
2018-01-08  1246.87
2018-01-09  1252.70
2018-01-10  1254.33
```

Given that we are constructing portfolios with multiple tickers, let's wrap our `get_prices` function in something bigger that can handle multiple tickers.

get_historical

Here's the code for `get_historical`:

```
def get_historical(tickers, start_date, end_date):
    df = pd.DataFrame(index=pd.date_range(start_date, end_date,
         freq='D'))
```

```
    for t in tickers:
        df = pd.concat([
            df,
            get_price(t, outputsize='full')],
            axis=1,
            join_axes=[df.index]
        )
    df = df.fillna(method='ffill').dropna()
    return df
```

This get_historical function will loop through each ticker and bind the cleaned-up response to a main DataFrame object. The function will enable us to fetch data and close prices for any number of listed companies across any arbitrary length of time.

```
historical_prices = get_historical(
    tickers=['AMZN', 'CSCO', 'GE'],
    start_date=pd.Timestamp(2016, 1, 1),
    end_date=TODAY
)

print(historical_prices.tail())
```

```
               AMZN   CSCO     GE
2018-05-25  1610.15  43.26  14.63
2018-05-26  1610.15  43.26  14.63
2018-05-27  1610.15  43.26  14.63
2018-05-28  1610.15  43.26  14.63
2018-05-29  1612.87  42.97  14.18
```

With all the historical prices stored inside a DataFrame, we can grab prices for a specific date by using `.loc`.

```
prices = historical_prices.loc['2016-01-04']
prices
```

```
AMZN    636.99
CSCO     26.41
GE       30.71
Name: 2016-01-04 00:00:00, dtype: float64
```

Portfolio

Because I know you just want to see the full thing in action, here it is:

```
portfolio = instantiate_portfolio(targets, 100000.00)
prices = historical_prices.loc['2017-01-01']
update_prices(portfolio, prices)
order = get_order(portfolio)
simulate_process_order(portfolio, order)
portfolio.market_value.sum()
```

```
100000.0
```

This will be our starting portfolio:

```
print(portfolio)
```

```
           date   price target allocation  shares market_value
AMZN 2017-01-01  749.87    0.4   0.397431      53      39743.1
CSCO 2017-01-01   30.22    0.3   0.299782     992      29978.2
GE   2017-01-01    31.6    0.3   0.299884     949      29988.4
CASH 2017-01-01       1      0  0.0029025  290.25       290.25
```

Rebalance

To test our rebalancing logic, we'll back-test across 2017 and execute orders on a quarterly-end frequency by using the Q offset alias from pandas.

```
dates = pd.date_range('2017-01-01', '2017-12-31', freq='Q').tolist()
for d in dates:
    prices = historical_prices.loc[d]
    update_prices(portfolio, prices)
    order = get_order(portfolio)
    print(f'{d}:\n{order}')
    simulate_process_order(portfolio, order)

portfolio.market_value.sum()
```

```
2017-03-31 00:00:00:
AMZN     -4
CSCO    -24
GE      149
dtype: object
2017-06-30 00:00:00:
AMZN    -5
CSCO    63
GE      97
dtype: object
2017-09-30 00:00:00:
AMZN      0
CSCO    -83
GE      124
dtype: object
2017-12-31 00:00:00:
AMZN     -7
CSCO    -79
```

```
GE      589
dtype: object

111030.14
```

After four rebalancing moves, we can verify that our portfolio will follow and maintain target allocations quite closely.

```
print(portfolio)
```

	date	price	target	allocation	shares	market_value
AMZN	2017-12-31	1169.47	0.4	0.389718	37	43270.4
CSCO	2017-12-31	38.3	0.3	0.299763	869	33282.7
GE	2017-12-31	17.45	0.3	0.29987	1908	33294.6
CASH	2017-12-31	1	0	0.0106498	1182.45	1182.45

Conclusion

In this chapter, you learned how to build a portfolio in pandas, update values in a DataFrame, generate buy and sell orders that aim to hold target allocations in balance, retrieve stock quotes from Alpha Vantage, and simulate back-testing.

If you want to actually put these pieces to work, you will need to set up an account with an online brokerage and manually exercise buy and sell orders on its platform.

The good news is that if you think that rebalancing is an appropriate investment strategy for you, you don't actually have to do it that often. If you adhere to monthly or quarterly rebalancing, you'll be money!

CHAPTER 7

Spend

> We ain't 'bout to go and spend money just to flex on 'em.
>
> —Lil Dickey

Think of this as a bonus chapter. I've included it in this book to give you a taste of how open-source projects and tools such as scikit-learn,[1] XGBoost,[2] and Prophet build on the work of pandas.

I promised in the introduction to this book that if you built a foundation in pandas, you would be well positioned to dive into machine learning. I want to attempt to deliver on that promise.

A quick disclaimer before we continue: machine learning is a massive and complex topic. Given that this book uses personal finance as the glue, I will defer on most of the theory. Additionally, I'll focus on only one small slice of machine learning: time-series forecasting.

Prophet

There are hundreds of open-source libraries for forecasting in Python. One that I really like is Prophet. According to the GitHub repo,[3] "Prophet is a procedure for forecasting time series data. It is based on an additive model

[1] http://scikit-learn.org/stable/index.html
[2] https://github.com/dmlc/xgboost
[3] https://github.com/facebook/prophet

M. Humber, *Personal Finance with Python*, https://doi.org/10.1007/978-1-4842-3802-8_7

where non-linear trends are fit with yearly and weekly seasonality, plus holidays. It works best with daily periodicity data with at least one year of historical data. Prophet is robust to missing data, shifts in the trend, and large outliers." That's all to say that if you give Prophet historical data, it will attempt to extrapolate into the future for you.

Installing Prophet on your machine is a bit more involved, but the README[4] is incredibly detailed. Install the library now before you continue with this chapter.

Purchases

To demonstrate how Prophet works, we need some historical data. Given that this book concerns personal finance, I thought it would be fun to go ultra-personal and curate data on me.

The `purchases.csv` file, which was included in the initial data download from Chapter 1, contains all of my Amazon purchases since 2012. It has 83 rows and 2 columns (date and amount spent).

Note If you don't care for my data (and you really shouldn't!), just coerce your own values into the same structure, and you'll be ready to rock.

Load pandas and the purchases data.

```
import pandas as pd
purchases = pd.read_csv('data/purchases.csv')
```

We can see that I spent $17.99 on January 7, 2018, and $158.19 on January 31, 2018.

[4] https://github.com/facebook/prophet#installation-in-python

```
print(purchases.tail())
```

```
          date  amount
78  2017-12-24   62.53
79  2017-12-27   43.99
80  2017-12-28   21.99
81  2018-01-07   17.99
82  2018-01-31  158.19
```

Rolling the values into a running total can be accomplished with the following:

```
purchases['cumsum'] = purchases['amount'].cumsum()
print(purchases.tail())
```

```
          date   amount   cumsum
78  2017-12-24    62.53  4906.19
79  2017-12-27    43.99  4950.18
80  2017-12-28    21.99  4972.17
81  2018-01-07    17.99  4990.16
82  2018-01-31   158.19  5148.35
```

We can now see that I've spent a grand total of $5,148.35 on Amazon since 2012. Crazy!

Forecast

To forecast with Prophet, we need to massage the purchases DataFrame to conform to a specific input structure. Prophet inputs are always a DataFrame with two columns: `ds` and `y`. The `ds` (datestamp) column must contain a date, and the `y` column must be numeric and represents the measurement we want to forecast.

```
purchases = purchases[['date', 'cumsum']]
purchases.columns = ['ds', 'y']
print(purchases.head())
```

```
           ds       y
0  2012-07-25   82.55
1  2012-12-10  143.56
2  2013-02-19  155.10
3  2013-02-24  221.77
4  2013-04-20  229.76
```

Let's import Prophet.

```
from fbprophet import Prophet
```

Instantiate an instance of the `Prophet` class and fit it into the purchases data.

```
m = Prophet(daily_seasonality=False)
m.fit(purchases)
```

To forecast values one year into the future, we need to use the `.make_future_dataframe` method with the `periods` argument set to 365.

```
future = m.make_future_dataframe(periods=365)
print(future.tail())
```

```
            ds
443 2019-01-27
444 2019-01-28
445 2019-01-29
446 2019-01-30
447 2019-01-31
```

Predicting my total spending on Amazon through 2018 and into 2019 is now dead simple. We just have to call the `predict` method on top of the future DataFrame that we created earlier.

```
forecast = m.predict(future)
```

Let's inspect the `forecast` object:

```
forecast[['ds', 'yhat', 'yhat_lower', 'yhat_upper']].tail()
```

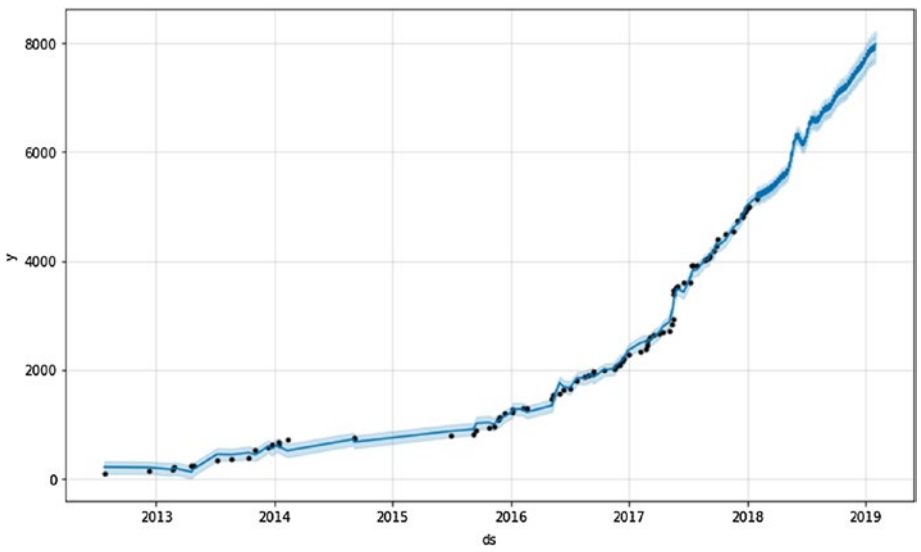
	ds	yhat	yhat_lower	yhat_upper
443	2019-01-27	7914.330291	7656.482987	8137.708873
444	2019-01-28	7891.583392	7633.735995	8129.905842
445	2019-01-29	7875.415396	7619.920169	8102.319783
446	2019-01-30	7924.375719	7667.813754	8148.685988
447	2019-01-31	7985.795633	7740.228707	8224.416333

We can see that I am meant to spend a cumulative total of $7,985 (yhat) on Amazon. But now that I know that, I'm keen to ratchet back my spending!

Visualize

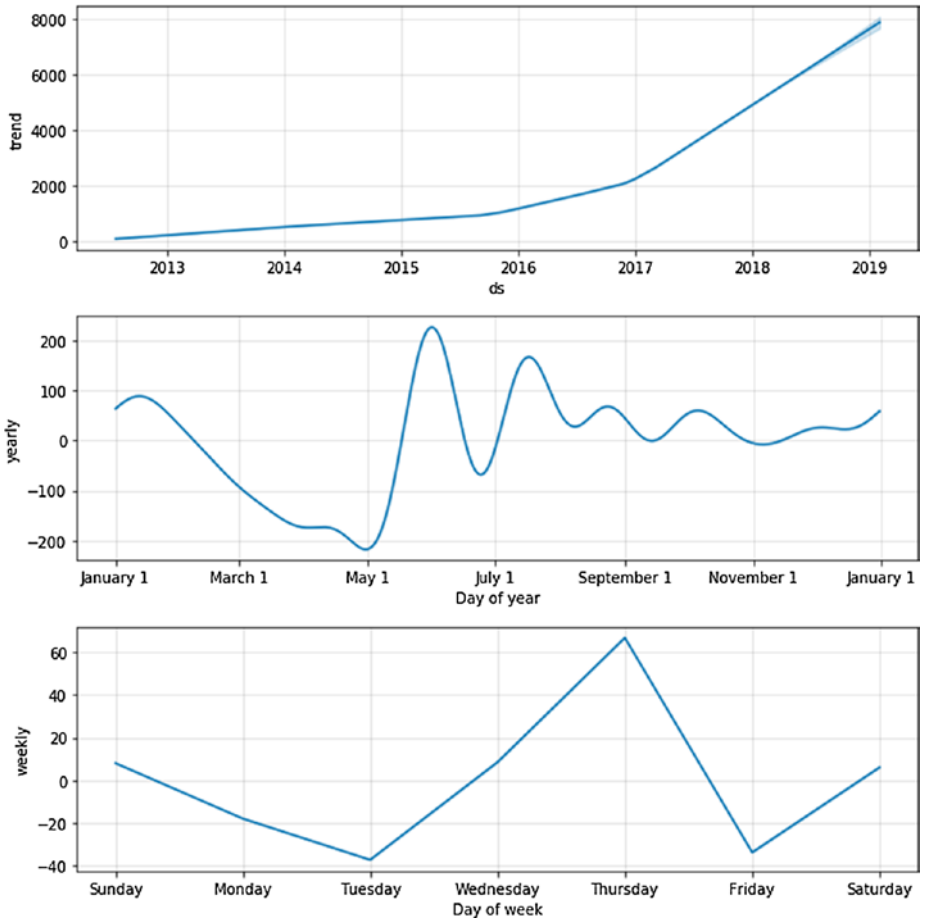
Prophet comes packaged with a really great `plot` convenience method. Running it on top of the `m` object, we can get a better sense of the trend.

```
%matplotlib inline
m.plot(forecast)
```



The library also includes a `plot_components` method that prints panels for trend, weekly, and yearly seasonalities if present.

```
m.plot_components(forecast)
```



Based on these panels, we can see that I spend a lot on Amazon in June and on Thursdays and that ever since I got Prime 2015 my spending has been growing significantly!

Conclusion

If you think that using my own personal spending data is a bit silly, you're 100 percent right. Honestly, I could have borrowed a more "professional" dataset from somewhere else, but where's the fun in that?

Like I said, this chapter was mostly meant to be a bonus. But in all seriousness, I hope you can see that learning pandas pays incredible dividends because it is the industry standard for managing and working with data in Python.

If you get good at pandas (and you should be by now!), you can extend that knowledge into some pretty interesting areas, such as time-series forecasting.

APPENDIX

Next

If you've made it to this chapter, congrats! I hope you've learned something!

If you enjoyed this book and got a thrill out of learning about Python programming, I encourage you to keep at it. It's a really great language. If you want to learn more, plenty of wonderful books and resources are available. This appendix gives a few recommendations.

Illustrated Guide to Python 3 by Matt Harrison

> This book brings developers and others who are anxious to learn Python up to speed quickly. Not only does it teach the basics of syntax, but it condenses years of experience. You will learn warts, gotchas, best practices, and hints that have been gleaned through the years by the author. You will hit the ground running in the right way.

The Hitchhiker's Guide to Python by Kenneth Reitz and Tanya Schlusser

> This book takes the journeyperson Pythonista to true expertise. More than any other language, Python was created with the philosophy of simplicity and parsimony. Now 25 years old, Python has become the primary or secondary language (after SQL) for many business users. With popularity comes diversity—and possibly dilution.

M. Humber, *Personal Finance with Python*, https://doi.org/10.1007/978-1-4842-3802-8

Think Python by Allen B. Downey

> This book is an introduction to computer science using the Python programming language. It covers the basics of computer programming, including variables and values, functions, conditionals and control flow, program development, and debugging. Later chapters cover basic algorithms and data structures.

Python for Data Analysis: Data Wrangling with Pandas, Numpy, and IPython by Wes McKinney

> Looking for complete instructions on manipulating, processing, cleaning, and crunching structured data in Python? The second edition of this hands-on guide—updated for Python 3.5 and pandas 1.0—is packed with practical cases studies that show you how to effectively solve a broad set of data analysis problems, using Python libraries such as NumPy, pandas, Matplotlib, and IPython.

Python Data Science Handbook: Tools and Techniques for Developers by Jake VanderPlas

> For many researchers, Python is a first-class tool mainly because of its libraries for storing, manipulating, and gaining insight from data. Several resources exist for individual pieces of this data science stack, but only with this book do you get them all—IPython, NumPy, pandas, Matplotlib, scikit-learn, and other related tools.

Finally, please e-mail me if you have any concerns, questions, or comments about this book. Your feedback is tremendously valuable, and I will do my best to respond to each e-mail. Again, I can be reached via e-mail.

max{dot}humber{at}gmail{dot}com

I look forward to hearing from you!

Index

M. Humber, *Personal Finance with Python*, https://doi.org/10.1007/978-1-4842-3802-8

D, E, F, G, H

I

J, K, L

M

N, O

P, Q

R

S

T

U

V, W, X

Y, Z