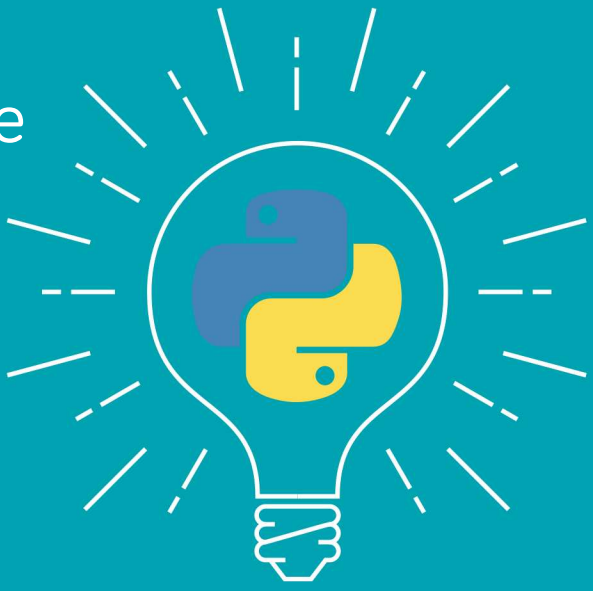
PYTHON TRICKS THE BOOK

A Buffet of Awesome Python Features



Dan Bader

Python Tricks: The Book

Dan Bader

For online information and ordering of this and other books by Dan Bader, please visit dbader.org. For more information, please contact Dan Bader at mail@dbader.org.

ISBN: 9781775093305 (paperback)

ISBN: 9781775093312 (electronic)

Cover design by Anja Pircher Design (anjapircher.com)

Updated 2017-10-27 I would like to thank Michael Howitz, Johnathan Willitts, Julian Orbach, Johnny Giorgis, Bob White, Daniel Meyer, Michael Stueben, Smital Desai, Andreas Kreisig, David Perkins, Jay Prakash Singh, and Ben Felder for their excellent feedback.

What Pythonistas Say About *Python Tricks: The Book*

"I love love love the book. It's like having a seasoned tutor explaining, well, tricks! I'm learning Python on the job and I'm coming from powershell, which I learned on the job—so lots of new, great stuff. Whenever I get stuck in Python (usually with flask blueprints or I feel like my code could be more Pythonic) I post questions in our internal Python chat room.

I'm often amazed at some of the answers coworkers give me. Dict comprehensions, lambdas, and generators often pepper their feedback. I am always impressed and yet flabbergasted at how powerful Python is when you know these tricks and can implement them correctly.

Your book was exactly what I wanted to help get me from a bewildered powershell scripter to someone who knows how and when to use these Pythonic 'tricks' everyone has been talking about.

As someone who doesn't have my degree in CS it's nice to have the text to explain things that others might have learned when they were classically educated. I am really enjoying the book and am subscribed to the emails as well, which is how I found out about the book."

— **Daniel Meyer**, Sr. Desktop Administrator at Tesla Inc.

"I first heard about your book from a co-worker who wanted to trick me with your example of how dictionaries are built. I was almost 100% sure about the reason why the end product was a much smaller/simpler dictionary but I must confess that I did not expect the outcome :)

He showed me the book via video conferencing and I sort of skimmed through it as he flipped the pages for me, and I was immediately curious to read more.

That same afternoon I purchased my own copy and proceeded to read your explanation for the way dictionaries are created in Python and later that day, as I met a different co-worker for coffee, I used the same trick on him :)

*He then sprung a different question on the same principle, and because of the way you explained things in your book, I was able to*not* guess the result but correctly answer what the outcome would be. That means that you did a great job at explaining things :)**

I am not new in Python and some of the concepts in some of the chapters are not new to me, but I must say that I do get something out of every chapter so far, so kudos for writing a very nice book and for doing a fantastic job at explaining concepts behind the tricks! I'm very much looking forward to the updates and I will certainly let my friends and co-workers know about your book."

— **Og Maciel**, Python Developer at Red Hat

”I really enjoyed reading Dan’s book. He explains important Python aspects with clear examples (using two twin cats to explain ‘is‘ vs ‘==‘ for example).

It is not just code samples, it discusses relevant implementation details comprehensibly. What really matters though is that this book makes you write better Python code!

The book is actually responsible for recent new good Python habits I picked up, for example: using custom exceptions and ABC’s (I found Dan’s blog searching for abstract classes.) These new learnings alone are worth the price.”

— **Bob Belderbos**, Engineer at Oracle & Co-Founder of PyBites

Contents

Foreword

It's been almost ten years since I first got acquainted with Python as a programming language. When I first learned Python many years ago, it was with a little reluctance. I had been programming in a different language before, and all of the sudden at work, I was assigned to a different team where everyone used Python. That was the beginning of my own Python journey.

When I was first introduced to Python, I was told that it was going to be easy, that I should be able to pick it up quickly. When I asked my colleagues for resources for learning Python, all they gave me was a link to Python's official documentation. Reading the documentation was confusing at first, and it really took me a while before I even felt comfortable navigating through it. Often I found myself needing to look for answers in StackOverflow.

Coming from a different programming language, I wasn't looking for just any resource for learning how to program or what classes and objects are. I was looking for specific resources that would teach me the features of Python, what sets it apart, and how writing in Python is different than writing code in another language.

It really has taken me many years to fully appreciate this language. As I read Dan's book, I kept thinking that I wished I had access to a book like this when I started learning Python many years ago.

For example, one of the many unique Python features that surprised me at first were list comprehensions. As Dan mentions in the book,

a tell of someone who just came to Python from a different language is the way they use for-loops. I recall one of the earliest code review comments I got when I started programming in Python was, "Why not use list comprehension here?" Dan explains this concept clearly in section 6, starting by showing how to loop the Pythonic way and building it all the way up to iterators and generators.

In chapter 2.5, Dan discusses the different ways to do string formatting in Python. String formatting is one of those things that defy the Zen of Python, that there should only be one obvious way to do things. Dan shows us the different ways, including my favorite new addition to the language, the f-strings, and he also explains the pros and cons of each method.

The Pythonic Productivity Techniques section is another great resource. It covers aspects beyond the Python programming language, and also includes tips on how to debug your programs, how to manage the dependencies, and gives you a peek inside Python bytecode.

It truly is an honor and my pleasure to introduce this book, Python Tricks, by my friend, Dan Bader.

By contributing to Python as a CPython core developer, I get connected to many members of the community. In my journey, I found mentors, allies, and made many new friends. They remind me that Python is not just about the code, Python is a community.

Mastering Python programming isn't just about grasping the theoretical aspects of the language. It's just as much about understanding and adopting the conventions and best practices used by its community.

Dan's book will help you on this journey. I'm convinced that you'll be more confident when writing Python programs after reading it.

— **Mariatta Wijaya**, Python Core Developer (mariatta.ca)

Chapter 1

Introduction

1.1 What's a Python Trick?

> **Python Trick**: *A short Python code snippet meant as a teaching tool. A Python Trick either teaches an aspect of Python with a simple illustration, or it serves as a motivating example, enabling you to dig deeper and develop an intuitive understanding.*

Python Tricks started out as a short series of code screenshots that I shared on Twitter for a week. To my surprise, they got rave responses and were shared and retweeted for days on end.

More and more developers started asking me for a way to "get the whole series." Actually, I only had a few of these tricks lined up, spanning a variety of Python-related topics. There wasn't a master plan behind them. They were just a fun little Twitter experiment.

But from these inquiries I got the sense that my short-and-sweet code examples would be worth exploring as a teaching tool. Eventually I set out to create a few more Python Tricks and shared them in an email series. Within a few days, several hundred Python developers had signed up and I was just blown away by that response.

Over the following days and weeks, a steady stream of Python developers reached out to me. They thanked me for making a part of the language they were struggling to understand *click* for them. Hearing this feedback felt awesome. I thought these Python Tricks were just code screenshots, but so many developers were getting a lot of value out of them.

That's when I decided to double down on my Python Tricks experiment and expanded it into a series of around 30 emails. Each of these was still just a a headline and a code screenshot, and I soon realized the limits of that format. Around this time, a blind Python developer emailed me, disappointed to find that these Python Tricks were delivered as images he couldn't read with his screen reader.

Clearly, I needed to invest more time into this project to make it more appealing and more accessible to a wider audience. So, I sat down to re-create the whole series of Python Tricks emails in plain text and with proper HTML-based syntax highlighting. That new iteration of Python Tricks chugged along nicely for a while. Based on the responses I got, developers seemed happy they could finally copy and paste the code samples in order to play around with them.

As more and more developers signed up for the email series, I started noticing a pattern in the replies and questions I received. Some Tricks worked well as motivational examples by themselves. However, for the more complex ones there was no narrator to guide readers or to give them additional resources to develop a deeper understanding.

Let's just say this was another big area of improvement. My mission statement for dbader.org is to *help Python developers become more awesome*—and this was clearly an opportunity to get closer to that goal.

I decided to take the best and most valuable Python Tricks from the email course, and I started writing a new kind of Python book around them:

- A book that teaches the coolest aspects of the language with short and easy-to-digest examples.
- A book that works like a buffet of awesome Python features (yum!) and keeps motivation levels high.
- A book that takes you by the hand to guide you and help you deepen your understanding of Python.

This book is really a labor of love for me and also a huge experiment. I hope you'll enjoy reading it and learn something about Python in the process!

— Dan Bader

1.2 What This Book Will Do for You

My goal for this book is to make you a better—more effective, more knowledgeable, more practical—Python developer. You might be wondering, *How will reading this book help me achieve all that?*

Python Tricks is not a step-by-step Python tutorial. It is not an entry-level Python course. If you're in the beginning stages of learning Python, the book alone won't transform you into a professional Python developer. Reading it will still be beneficial to you, but you need to make sure you're working with some other resources to build up your foundational Python skills.

You'll get the most out of this book if you already have some knowledge of Python, and you want to get to the next level. It will work great for you if you've been coding Python for a while and you're ready to go deeper, to round out your knowledge, and to make your code more Pythonic.

Reading *Python Tricks* will also be great for you if you already have experience with other programming languages and you're looking to get up to speed with Python. You'll discover a ton of practical tips and design patterns that'll make you a more effective and skilled Python coder.

1.3 How to Read This Book

The best way to read *Python Tricks: The Book* is to treat it like a buffet of awesome Python features. Each Python Trick in the book is self-contained, so it's completely okay to jump straight to the ones that look the most interesting. In fact, I would encourage you to do just that.

Of course, you can also read through all the Python Tricks in the order they're laid out in the book. That way you won't miss any of them, and you'll know you've seen it all when you arrive at the final page.

Some of these tricks will be easy to understand right away, and you'll have no trouble incorporating them into your day to day work just by reading the chapter. Other tricks might require a bit more time to crack.

If you're having trouble making a particular trick work in your own programs, it helps to play through each of the code examples in a Python interpreter session.

If that doesn't make things click, then please feel free to reach out to me, so I can help you out and improve the explanation in this book. In the long run, that benefits not just you but all Pythonistas reading this book.

Chapter 2

Patterns for Cleaner Python

2.1 Covering Your A** With Assertions

Sometimes a genuinely helpful language feature gets less attention than it deserves. For some reason, this is what happened to Python's built-in `assert` statement.

In this chapter I'm going to give you an introduction to using assertions in Python. You'll learn how to use them to help automatically detect errors in your Python programs. This will make your programs more reliable and easier to debug.

At this point, you might be wondering "What are assertions and what are they good for?" Let's get you some answers for that.

At its core, Python's assert statement is a debugging aid that tests a condition. If the assert condition is true, nothing happens, and your program continues to execute as normal. But if the condition evaluates to false, an `AssertionError` exception is raised with an optional error message.

Assert in Python — An Example

Here's a simple example so you can see where assertions might come in handy. I tried to give this some semblance of a real-world problem you might actually encounter in one of your programs.

Suppose you were building an online store with Python. You're working to add a discount coupon functionality to the system, and eventually you write the following `apply_discount` function:

```
def apply_discount(product, discount):
    price = int(product['price'] * (1.0 - discount))
    assert 0 <= price <= product['price']
    return price
```

Notice the `assert` statement in there? It will guarantee that, no matter what, discounted prices calculated by this function cannot be lower

than $0 and they cannot be higher than the original price of the product.

Let's make sure this actually works as intended if we call this function to apply a valid discount. In this example, products for our store will be represented as plain dictionaries. This is probably not what you'd do for a real application, but it'll work nicely for demonstrating assertions. Let's create an example product—a pair of nice shoes at a price of $149.00:

```
>>> shoes = {'name': 'Fancy Shoes', 'price': 14900}
```

By the way, did you notice how I avoided currency rounding issues by using an integer to represent the price amount in cents? That's generally a good idea... But I digress. Now, if we apply a 25% discount to these shoes, we would expect to arrive at a sale price of $111.75:

```
>>> apply_discount(shoes, 0.25)
11175
```

Alright, this worked nicely. Now, let's try to apply some invalid discounts. For example, a 200% "discount" that would lead to us giving money to the customer:

```
>>> apply_discount(shoes, 2.0)
Traceback (most recent call last):
  File "<input>", line 1, in <module>
    apply_discount(prod, 2.0)
  File "<input>", line 4, in apply_discount
    assert 0 <= price <= product['price']
AssertionError
```

As you can see, when we try to apply this invalid discount, our program halts with an `AssertionError`. This happens because a discount of 200% violated the assertion condition we placed in the `apply_discount` function.

You can also see how the exception stacktrace points out the exact line of code containing the failed assertion. If you (or another developer on your team) ever encounter one of these errors while testing the online store, it will be easy to find out what happened just by looking at the exception traceback.

This speeds up debugging efforts considerably, and it will make your programs more maintainable in the long-run. And that, my friend, is the power of assertions.

Why Not Just Use a Regular Exception?

Now, you're probably wondering why I didn't just use an if-statement and an exception in the previous example...

You see, the proper use of assertions is to inform developers about *unrecoverable* errors in a program. Assertions are *not* intended to signal expected error conditions, like a File-Not-Found error, where a user can take corrective actions or just try again.

Assertions are meant to be *internal self-checks* for your program. They work by declaring some conditions as *impossible* in your code. If one of these conditions doesn't hold, that means there's a bug in the program.

If your program is bug-free, these conditions will never occur. But if they *do* occur, the program will crash with an assertion error telling you exactly which "impossible" condition was triggered. This makes it much easier to track down and fix bugs in your programs. And I like anything that makes life easier—don't you?

For now, keep in mind that Python's assert statement is a debugging aid, not a mechanism for handling run-time errors. The goal of using assertions is to let developers find the likely root cause of a bug more quickly. An assertion error should never be raised unless there's a bug in your program.

Let's take a closer look at some other things we can do with assertions,

and then I'll cover two common pitfalls when using them in real-world scenarios.

Python's Assert Syntax

It's always a good idea to study up on how a language feature is actually implemented in Python before you start using it. So let's take a quick look at the syntax for the assert statement, according to the Python docs:[1]

```
assert_stmt ::= "assert" expression1 ["," expression2]
```

In this case, `expression1` is the condition we test, and the optional `expression2` is an error message that's displayed if the assertion fails. At execution time, the Python interpreter transforms each assert statement into roughly the following sequence of statements:

```
if __debug__:
    if not expression1:
        raise AssertionError(expression2)
```

Two interesting things about this code snippet:

Before the assert condition is checked, there's an additional check for the `__debug__` global variable. It's a built-in boolean flag that's true under normal circumstances and false if optimizations are requested. We'll talk some more about later that in the "common pitfalls" section.

Also, you can use `expression2` to pass an optional error message that will be displayed with the `AssertionError` in the traceback. This can simplify debugging even further. For example, I've seen code like this:

```
>>> if cond == 'x':
...     do_x()
```

[1] cf. Python Docs: "The Assert Statement"