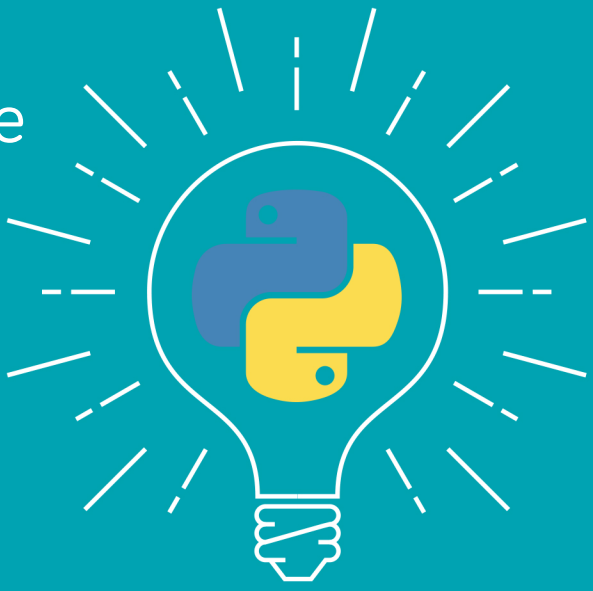
PYTHON TRICKS THE BOOK

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"

```
... elif cond == 'y':
...     do_y()
... else:
...     assert False, (
...         'This should never happen, but it does '
...         'occasionally. We are currently trying to '
...         'figure out why. Email dbader if you '
...         'encounter this in the wild. Thanks!')
```

Is this ugly? Well, yes. But it's definitely a valid and helpful technique if you're faced with a Heisenbug[2] in one of your applications.

Common Pitfalls With Using Asserts in Python

Before you move on, there are two important caveats regarding the use of assertions in Python that I'd like to call out.

The first one has to do with introducing security risks and bugs into your applications, and the second one is about a syntax quirk that makes it easy to write *useless* assertions.

This sounds (and potentially is) quite horrible, so you should probably at least skim these two caveats below.

Caveat #1 – Don't Use Asserts for Data Validation

The biggest caveat with using asserts in Python is that assertions can be globally disabled[3] with the `-O` and `-OO` command line switches, as well as the `PYTHONOPTIMIZE` environment variable in CPython.

This turns any assert statement into a null-operation: the assertions simply get compiled away and won't be evaluated, which means that none of the conditional expressions will be executed.

[2] cf. Wikipedia: Heisenbug
[3] cf. Python Docs: "Constants (__debug__)"

This is an intentional design decision used similarly by many other programming languages. As a side-effect, it becomes extremely dangerous to use assert statements as a quick and easy way to validate input data.

Let me explain—if your program uses asserts to check if a function argument contains a "wrong" or unexpected value, this can backfire quickly and lead to bugs or security holes.

Let's take a look at a simple example that demonstrates this problem. Again, imagine you're building an online store application with Python. Somewhere in your application code there's a function to delete a product as per a user's request.

Because you just learned about assertions, you're eager to use them in your code (hey, I know I would be!) and you write the following implementation:

```
def delete_product(prod_id, user):
    assert user.is_admin(), 'Must be admin'
    assert store.has_product(prod_id), 'Unknown product'
    store.get_product(prod_id).delete()
```

Take a close look at this `delete_product` function. Now, what's going to happen if assertions are disabled?

There are two serious issues in this three-line function example, and they're caused by the incorrect use of assert statements:

1. **Checking for admin privileges with an assert statement is dangerous.** If assertions are disabled in the Python interpreter, this turns into a null-op. Therefore *any user can now delete products*. The privileges check doesn't even run. This likely introduces a security problem and opens the door for attackers to destroy or severely damage the data in our online store. Not good.

2. **The `has_product()` check is skipped when assertions are disabled.** This means `get_product()` can now be called with invalid product IDs—which could lead to more severe bugs, depending on how our program is written. In the worst case, this could be an avenue for someone to launch Denial of Service attacks against our store. For example, if the store app crashes if someone attempts to delete an unknown product, an attacker could bombard it with invalid delete requests and cause an outage.

How might we avoid these problems? The answer is to *never* use assertions to do data validation. Instead, we could do our validation with regular `if`-statements and raise validation exceptions if necessary, like so:

```
def delete_product(product_id, user):
    if not user.is_admin():
        raise AuthError('Must be admin to delete')
    if not store.has_product(product_id):
        raise ValueError('Unknown product id')
    store.get_product(product_id).delete()
```

This updated example also has the benefit that instead of raising unspecific `AssertionError` exceptions, it now raises semantically correct exceptions like `ValueError` or `AuthError` (which we'd have to define ourselves.)

Caveat #2 – Asserts That Never Fail

It's surprisingly easy to accidentally write Python assert statements that always evaluate to true. I've been bitten by this myself in the past. Here's the problem, in a nutshell:

When you pass a tuple as the first argument in an `assert` statement, the assertion always evaluates as true and therefore never fails.

For example, this assertion will never fail:

```
assert(1 == 2, 'This should fail')
```

This has to do with non-empty tuples always being truthy in Python. If you pass a tuple to an assert statement, it leads to the assert condition always being true—which in turn leads to the above assert statement being *useless* because it can never fail and trigger an exception.

It's relatively easy to accidentally write bad multi-line asserts due to this, well, unintuitive behavior. For example, I merrily wrote a bunch of broken test cases that gave a false sense of security in one of my test suites. Imagine you had this assertion in one of your unit tests:

```
assert (
    counter == 10,
    'It should have counted all the items'
)
```

Upon first inspection, this test case looks completely fine. However, it would never catch an incorrect result: the assertion always evaluates to `True`, regardless of the state of the counter variable. And why is that? Because it asserts the truth value of a tuple object.

Like I said, it's rather easy to shoot yourself in the foot with this (mine still hurts). A good countermeasure you can apply to prevent this syntax quirk from causing trouble is to use a code linter.[4] Newer versions of Python 3 will also show a syntax warning for these dubious asserts.

By the way, that's also why you should always do a quick smoke test with your unit test cases. Make sure they can actually fail before you move on to writing the next one.

[4] I wrote an article about avoiding bogus assertions in your Python tests. You can find it here: dbader.org/blog/catching-bogus-python-asserts.

Python Assertions — Summary

Despite these caveats I believe that Python's assertions are a powerful debugging tool that's frequently underused by Python developers.

Understanding how assertions work and when to apply them can help you write Python programs that are more maintainable and easier to debug.

It's a great skill to learn that will help bring your Python knowledge to the next level and make you a more well-rounded Pythonista. I know it has saved me hours upon hours of debugging.

Key Takeaways

- Python's assert statement is a debugging aid that tests a condition as an internal self-check in your program.
- Asserts should only be used to help developers identify bugs. They're not a mechanism for handling run-time errors.
- Asserts can be globally disabled with an interpreter setting.

2.2 Complacent Comma Placement

Here's a handy tip for when you're adding and removing items from a list, dict, or set constant in Python: Just end all of your lines with a comma.

Not sure what I'm talking about? Let me give you a quick example. Imagine you've got this list of names in your code:

```
>>> names = ['Alice', 'Bob', 'Dilbert']
```

Whenever you make a change to this list of names, it'll be hard to tell what was modified by looking at a Git diff, for example. Most source control systems are line-based and have a hard time highlighting multiple changes to a single line.

A quick fix for that is to adopt a code style where you spread out list, dict, or set constants across multiple lines, like so:

```
>>> names = [
...     'Alice',
...     'Bob',
...     'Dilbert'
... ]
```

That way there's one item per line, making it perfectly clear which one was added, removed, or modified when you view a diff in your source control system. It's a small change but I found it helped me avoid silly mistakes. It also made it easier for my teammates to review my code changes.

Now, there are two editing cases that can still cause some confusion. Whenever you add a new item at the end of a list, or you remove the last item, you'll have to update the comma placement manually to get consistent formatting.

Let's say you'd like to add another name (*Jane*) to that list. If you add *Jane*, you'll need to fix the comma placement after the *Dilbert* line to avoid a nasty error:

```
>>> names = [
...     'Alice',
...     'Bob',
...     'Dilbert' # <- Missing comma!
...     'Jane'
]
```

When you inspect the contents of that list, brace yourself for a surprise:

```
>>> names
['Alice', 'Bob', 'DilbertJane']
```

As you can see, Python *merged* the strings *Dilbert* and *Jane* into *DilbertJane*. This so-called "string literal concatenation" is intentional and documented behavior. And it's also a fantastic way to shoot yourself in the foot by introducing hard-to-catch bugs into your programs:

> "Multiple adjacent string or bytes literals (delimited by whitespace), possibly using different quoting conventions, are allowed, and their meaning is the same as their concatenation."[5]

Still, string literal concatenation is a useful feature in some cases. For example, you can use it to reduce the number of backslashes needed to split long string constants across multiple lines:

[5] cf. Python Docs: "String literal concatenation"

```
my_str = ('This is a super long string constant '
          'spread out across multiple lines. '
          'And look, no backslash characters needed!')
```

On the other hand, we've just seen how the same feature can quickly turn into a liability. Now, how do we fix this situation?

Adding the missing comma after *Dilbert* prevents the two strings from getting merged into one:

```
>>> names = [
...     'Alice',
...     'Bob',
...     'Dilbert',
...     'Jane'
]
```

But now we've come full circle and returned to the original problem. I had to modify two lines in order to add a new name to the list. This makes it harder to see what was modified in the Git diff again... Did someone add a new name? Did someone change Dilbert's name?

Luckily, Python's syntax allows for some leeway to solve this comma placement issue once and for all. You just need to train yourself to adopt a code style that avoids it in the first place. Let me show you how.

In Python, you can place a comma after every item in a list, dict, or set constant, including the last item. That way, you can just remember to always end your lines with a comma and thus avoid the comma placement juggling that would otherwise be required.

Here's what the final example looks like:

```
>>> names = [
...     'Alice',
```

```
...     'Bob',
...     'Dilbert',
... ]
```

Did you spot the comma after *Dilbert*? That'll make it easy to add or remove new items without having to update the comma placement. It keeps your lines consistent, your source control diffs clean, and your code reviewers happy. Hey, sometimes the magic is in the little things, right?

Key Takeaways

- Smart formatting and comma placement can make your list, dict, or set constants easier to maintain.
- Python's string literal concatenation feature can work to your benefit, or introduce hard-to-catch bugs.

2.3 Context Managers and the `with` Statement

The `with` statement in Python is regarded as an obscure feature by some. But when you peek behind the scenes, you'll see that there's no *magic* involved, and it's actually a highly useful feature that can help you write cleaner and more readable Python code.

So what's the `with` statement good for? It helps simplify some common resource management patterns by abstracting their functionality and allowing them to be factored out and reused.

A good way to see this feature used effectively is by looking at examples in the Python standard library. The built-in `open()` function provides us with an excellent use case:

```
with open('hello.txt', 'w') as f:
    f.write('hello, world!')
```

Opening files using the `with` statement is generally recommended because it ensures that open file descriptors are closed automatically after program execution leaves the context of the `with` statement. Internally, the above code sample translates to something like this:

```
f = open('hello.txt', 'w')
try:
    f.write('hello, world')
finally:
    f.close()
```

You can already tell that this is quite a bit more verbose. Note that the `try...finally` statement is significant. It wouldn't be enough to just write something like this:

```
f = open('hello.txt', 'w')
f.write('hello, world')
f.close()
```

This implementation won't guarantee the file is closed if there's an exception during the `f.write()` call—and therefore our program might leak a file descriptor. That's why the `with` statement is so useful. It makes properly acquiring and releasing resources a breeze.

Another good example where the `with` statement is used effectively in the Python standard library is the `threading.Lock` class:

```
some_lock = threading.Lock()

# Harmful:
some_lock.acquire()
try:
    # Do something...
finally:
    some_lock.release()

# Better:
with some_lock:
    # Do something...
```

In both cases, using a `with` statement allows you to abstract away most of the resource handling logic. Instead of having to write an explicit `try...finally` statement each time, using the `with` statement takes care of that for us.

The `with` statement can make code that deals with system resources more readable. It also helps you avoid bugs or leaks by making it practically impossible to forget to clean up or release a resource when it's no longer needed.

Supporting `with` in Your Own Objects

Now, there's nothing special or magical about the `open()` function or the `threading.Lock` class and the fact that they can be used with a `with` statement. You can provide the same functionality in your own classes and functions by implementing so-called *context managers*.[6]

What's a context manager? It's a simple "protocol" (or interface) that your object needs to follow in order to support the `with` statement. Basically, all you need to do is add `__enter__` and `__exit__` methods to an object if you want it to function as a context manager. Python will call these two methods at the appropriate times in the resource management cycle.

Let's take a look at what this would look like in practical terms. Here's what a simple implementation of the `open()` context manager might look like:

```
class ManagedFile:
    def __init__(self, name):
        self.name = name

    def __enter__(self):
        self.file = open(self.name, 'w')
        return self.file

    def __exit__(self, exc_type, exc_val, exc_tb):
        if self.file:
            self.file.close()
```

Our `ManagedFile` class follows the context manager protocol and now supports the `with` statement, just like the original `open()` example did:

[6] cf. Python Docs: "With Statement Context Managers"

```
>>> with ManagedFile('hello.txt') as f:
...     f.write('hello, world!')
...     f.write('bye now')
```

Python calls __enter__ when execution *enters* the context of the with statement and it's time to acquire the resource. When execution *leaves* the context again, Python calls __exit__ to free up the resource.

Writing a class-based context manager isn't the only way to support the with statement in Python. The contextlib[7] utility module in the standard library provides a few more abstractions built on top of the basic context manager protocol. This can make your life a little easier if your use cases match what's offered by contextlib.

For example, you can use the contextlib.contextmanager decorator to define a generator-based *factory function* for a resource that will then automatically support the with statement. Here's what rewriting our ManagedFile context manager example with this technique looks like:

```
from contextlib import contextmanager

@contextmanager
def managed_file(name):
    try:
        f = open(name, 'w')
        yield f
    finally:
        f.close()

>>> with managed_file('hello.txt') as f:
...     f.write('hello, world!')
...     f.write('bye now')
```

[7] cf. Python Docs: "contextlib"

In this case, `managed_file()` is a generator that first acquires the resource. After that, it temporarily suspends its own execution and *yields* the resource so it can be used by the caller. When the caller leaves the `with` context, the generator continues to execute so that any remaining clean-up steps can occur and the resource can get released back to the system.

The class-based implementation and the generator-based one are essentially equivalent. You might prefer one over the other, depending on which approach you find more readable.

A downside of the `@contextmanager`-based implementation might be that it requires some understanding of advanced Python concepts like decorators and generators. If you need to get up to speed with those, feel free to take a detour to the relevant chapters here in this book.

Once again, making the right implementation choice here comes down to what you and your team are comfortable using and what you find the most readable.

Writing Pretty APIs With Context Managers

Context managers are quite flexible, and if you use the `with` statement creatively, you can define convenient APIs for your modules and classes.

For example, what if the "resource" we wanted to manage was text indentation levels in some kind of report generator program? What if we could write code like this to do it:

```
with Indenter() as indent:
    indent.print('hi!')
    with indent:
        indent.print('hello')
        with indent:
            indent.print('bonjour')
    indent.print('hey')
```

This almost reads like a domain-specific language (DSL) for indenting text. Also, notice how this code enters and leaves the same context manager multiple times to change indentation levels. Running this code snippet should lead to the following output and print neatly formatted text to the console:

```
hi!
    hello
        bonjour
hey
```

So, how would you implement a context manager to support this functionality?

By the way, this could be a great exercise for you to understand exactly how context managers work. So before you check out my implementation below, you might want to take some time and try to implement this yourself as a learning exercise.

If you're ready to check out my implementation, here's how you might implement this functionality using a class-based context manager:

```
class Indenter:
    def __init__(self):
        self.level = 0

    def __enter__(self):
        self.level += 1
        return self

    def __exit__(self, exc_type, exc_val, exc_tb):
        self.level -= 1

    def print(self, text):
        print('    ' * self.level + text)
```

That wasn't so bad, was it? I hope that by now you're already feeling more comfortable using context managers and the `with` statement in your own Python programs. They're an excellent feature that will allow you to deal with resource management in a much more Pythonic and maintainable way.

If you're looking for another exercise to deepen your understanding, try implementing a context manager that measures the execution time of a code block using the `time.time` function. Be sure to try out writing both a decorator-based and a class-based variant to drive home the difference between the two.

Key Takeaways

- The `with` statement simplifies exception handling by encapsulating standard uses of `try/finally` statements in so-called context managers.
- Most commonly it is used to manage the safe acquisition and release of system resources. Resources are acquired by the `with` statement and released automatically when execution leaves the `with` context.
- Using `with` effectively can help you avoid resource leaks and make your code easier to read.

2.4 Underscores, Dunders, and More

Single and double underscores have a meaning in Python variable and method names. Some of that meaning is merely by convention and intended as a hint to the programmer—and some of it is enforced by the Python interpreter.

If you're wondering, *"What's the meaning of single and double underscores in Python variable and method names?"* I'll do my best to get you the answer here. In this chapter we'll discuss the following five underscore patterns and naming conventions, and how they affect the behavior of your Python programs:

- Single Leading Underscore: `_var`
- Single Trailing Underscore: `var_`
- Double Leading Underscore: `__var`
- Double Leading and Trailing Underscore: `__var__`
- Single Underscore: `_`

1. Single Leading Underscore: "_var"

When it comes to variable and method names, the single underscore prefix has a meaning by convention only. It's a hint to the programmer—it means what the Python community agrees it should mean, but it does not affect the behavior of your programs.

The underscore prefix is meant as a *hint* to tell another programmer that a variable or method starting with a single underscore is intended for internal use. This convention is defined in PEP 8, the most commonly used Python code style guide.[8]

However, this convention isn't enforced by the Python interpreter. Python does not have strong distinctions between "private" and "public" variables like Java does. Adding a single underscore in front of a variable name is more like someone putting up a tiny underscore

[8] cf. PEP 8: "Style Guide for Python Code"

warning sign that says: *"Hey, this isn't really meant to be a part of the public interface of this class. Best to leave it alone."*

Take a look at the following example:

```
class Test:
    def __init__(self):
        self.foo = 11
        self._bar = 23
```

What's going to happen if you instantiate this class and try to access the `foo` and `_bar` attributes defined in its `__init__` constructor?

Let's find out:

```
>>> t = Test()
>>> t.foo
11
>>> t._bar
23
```

As you can see, the leading single underscore in `_bar` did not prevent us from "reaching into" the class and accessing the value of that variable.

That's because the single underscore prefix in Python is merely an agreed-upon convention—at least when it comes to variable and method names. However, leading underscores do impact how names get imported from modules. Imagine you had the following code in a module called `my_module`:

```
# my_module.py:

def external_func():
    return 23
```

```
def _internal_func():
    return 42
```

Now, if you use a *wildcard import* to import all the names from the module, Python will *not* import names with a leading underscore (unless the module defines an `__all__` list that overrides this behavior[9]):

```
>>> from my_module import *
>>> external_func()
23
>>> _internal_func()
NameError: "name '_internal_func' is not defined"
```

By the way, wildcard imports should be avoided as they make it unclear which names are present in the namespace.[10] It's better to stick to regular imports for the sake of clarity. Unlike wildcard imports, regular imports are not affected by the leading single underscore naming convention:

```
>>> import my_module
>>> my_module.external_func()
23
>>> my_module._internal_func()
42
```

I know this might be a little confusing at this point. If you stick to the PEP 8 recommendation that wildcard imports should be avoided, then all you really need to remember is this:

Single underscores are a Python naming convention that indicates a name is meant for internal use. It is generally not enforced by the Python interpreter and is only meant as a hint to the programmer.

[9] cf. Python Docs: "Importing * From a Package"
[10] cf. PEP 8: "Imports"

2. Single Trailing Underscore: "var_"

Sometimes the most fitting name for a variable is already taken by a keyword in the Python language. Therefore, names like `class` or `def` cannot be used as variable names in Python. In this case, you can append a single underscore to break the naming conflict:

```
>>> def make_object(name, class):
SyntaxError: "invalid syntax"

>>> def make_object(name, class_):
...     pass
```

In summary, a single trailing underscore (postfix) is used by convention to avoid naming conflicts with Python keywords. This convention is defined and explained in PEP 8.

3. Double Leading Underscore: "__var"

The naming patterns we've covered so far receive their meaning from agreed-upon conventions only. With Python class attributes (variables and methods) that start with double underscores, things are a little different.

A double underscore prefix causes the Python interpreter to rewrite the attribute name in order to avoid naming conflicts in subclasses.

This is also called *name mangling*—the interpreter changes the name of the variable in a way that makes it harder to create collisions when the class is extended later.

I know this sounds rather abstract. That's why I put together this little code example we can use for experimentation:

```
class Test:
    def __init__(self):
        self.foo = 11
```

```
        self._bar = 23
        self.__baz = 23
```

Let's take a look at the attributes on this object using the built-in `dir()` function:

```
>>> t = Test()
>>> dir(t)
['_Test__baz', '__class__', '__delattr__', '__dict__',
 '__dir__', '__doc__', '__eq__', '__format__', '__ge__',
 '__getattribute__', '__gt__', '__hash__', '__init__',
 '__le__', '__lt__', '__module__', '__ne__', '__new__',
 '__reduce__', '__reduce_ex__', '__repr__',
 '__setattr__', '__sizeof__', '__str__',
 '__subclasshook__', '__weakref__', '_bar', 'foo']
```

This gives us a list with the object's attributes. Let's take this list and look for our original variable names `foo`, `_bar`, and `__baz`. I promise you'll notice some interesting changes.

First of all, the `self.foo` variable appears unmodified as `foo` in the attribute list.

Next up, `self._bar` behaves the same way—it shows up on the class as `_bar`. Like I said before, the leading underscore is just a *convention* in this case—a hint for the programmer.

However, with `self.__baz` things look a little different. When you search for `__baz` in that list, you'll see that there is no variable with that name.

So what happened to `__baz`?

If you look closely, you'll see there's an attribute called `_Test__baz` on this object. This is the *name mangling* that the Python interpreter applies. It does this to protect the variable from getting overridden in subclasses.

Let's create another class that extends the `Test` class and attempts to override its existing attributes added in the constructor:

```
class ExtendedTest(Test):
    def __init__(self):
        super().__init__()
        self.foo = 'overridden'
        self._bar = 'overridden'
        self.__baz = 'overridden'
```

Now, what do you think the values of `foo`, `_bar`, and `__baz` will be on instances of this `ExtendedTest` class? Let's take a look:

```
>>> t2 = ExtendedTest()
>>> t2.foo
'overridden'
>>> t2._bar
'overridden'
>>> t2.__baz
AttributeError:
"'ExtendedTest' object has no attribute '__baz'"
```

Wait, why did we get that `AttributeError` when we tried to inspect the value of `t2.__baz`? Name mangling strikes again! It turns out this object doesn't even have a `__baz` attribute:

```
>>> dir(t2)
['_ExtendedTest__baz', '_Test__baz', '__class__',
 '__delattr__', '__dict__', '__dir__', '__doc__',
 '__eq__', '__format__', '__ge__', '__getattribute__',
 '__gt__', '__hash__', '__init__', '__le__', '__lt__',
 '__module__', '__ne__', '__new__', '__reduce__',
 '__reduce_ex__', '__repr__', '__setattr__',
 '__sizeof__', '__str__', '__subclasshook__',
 '__weakref__', '_bar', 'foo', 'get_vars']
```

As you can see, `__baz` got turned into `_ExtendedTest__baz` to prevent accidental modification. But the original `_Test__baz` is also still around:

```
>>> t2._ExtendedTest__baz
'overridden'
>>> t2._Test__baz
42
```

Double underscore name mangling is fully transparent to the programmer. Take a look at the following example that will confirm this:

```
class ManglingTest:
    def __init__(self):
        self.__mangled = 'hello'

    def get_mangled(self):
        return self.__mangled

>>> ManglingTest().get_mangled()
'hello'
>>> ManglingTest().__mangled
AttributeError:
"'ManglingTest' object has no attribute '__mangled'"
```

Does name mangling also apply to method names? It sure does! Name mangling affects *all* names that start with two underscore characters ("dunders") in a class context:

```
class MangledMethod:
    def __method(self):
        return 42

    def call_it(self):
```

```
        return self.__method()

>>> MangledMethod().__method()
AttributeError:
"'MangledMethod' object has no attribute '__method'"
>>> MangledMethod().call_it()
42
```

Here's another, perhaps surprising, example of name mangling in action:

```
_MangledGlobal__mangled = 23

class MangledGlobal:
    def test(self):
        return __mangled

>>> MangledGlobal().test()
23
```

In this example, I declared `_MangledGlobal__mangled` as a global variable. Then I accessed the variable inside the context of a class named `MangledGlobal`. Because of name mangling, I was able to reference the `_MangledGlobal__mangled` global variable as just `__mangled` inside the `test()` method on the class.

The Python interpreter automatically expanded the name `__mangled` to `_MangledGlobal__mangled` because it begins with two underscore characters. This demonstrates that name mangling isn't tied to class attributes specifically. It applies to any name starting with two underscore characters that is used in a class context.

Whew! That was a lot to absorb.

To be honest with you, I didn't write down these examples and explanations off the top of my head. It took me some research and editing

to do it. I've been using Python for years but rules and special cases like that aren't constantly on my mind.

Sometimes the most important skills for a programmer are "pattern recognition" and knowing where to look things up. If you feel a little overwhelmed at this point, don't worry. Take your time and play with some of the examples in this chapter.

Let these concepts sink in enough so that you'll recognize the general idea of name mangling and some of the other behaviors I've shown you. If you encounter them "in the wild" one day, you'll know what to look for in the documentation.

Sidebar: What are *dunders*?

If you've heard some experienced Pythonistas talk about Python or watched a few conference talks you may have heard the term *dunder*. If you're wondering what that is, well, here's your answer:

Double underscores are often referred to as "dunders" in the Python community. The reason is that double underscores appear quite often in Python code, and to avoid fatiguing their jaw muscles, Pythonistas often shorten "double underscore" to "dunder."

For example, you'd pronounce `__baz` as "dunder baz." Likewise, `__init__` would be pronounced as "dunder init," even though one might think it should be "dunder init dunder."

But that's just yet another quirk in the naming convention. It's like a *secret handshake* for Python developers.

4. Double Leading and Trailing Underscore: "__var__"

Perhaps surprisingly, name mangling is *not* applied if a name *starts and ends* with double underscores. Variables surrounded by a double underscore prefix and postfix are left unscathed by the Python interpreter:

```
class PrefixPostfixTest:
    def __init__(self):
        self.__bam__ = 42

>>> PrefixPostfixTest().__bam__
42
```

However, names that have both leading and trailing double underscores are reserved for special use in the language. This rule covers things like `__init__` for object constructors, or `__call__` to make objects callable.

These *dunder methods* are often referred to as *magic methods*—but many people in the Python community, including myself, don't like that word. It implies that the use of dunder methods is discouraged, which is entirely not the case. They're a core feature in Python and should be used as needed. There's nothing "magical" or arcane about them.

However, as far as naming conventions go, it's best to stay away from using names that start and end with double underscores in your own programs to avoid collisions with future changes to the Python language.

5. Single Underscore: "_"

Per convention, a single stand-alone underscore is sometimes used as a name to indicate that a variable is temporary or insignificant.

For example, in the following loop we don't need access to the running index and we can use "_" to indicate that it is just a temporary value:

```
>>> for _ in range(32):
...     print('Hello, World.')
```

You can also use single underscores in unpacking expressions as a

"don't care" variable to ignore particular values. Again, this meaning is per convention only and it doesn't trigger any special behaviors in the Python parser. The single underscore is simply a valid variable name that's sometimes used for this purpose.

In the following code example, I'm unpacking a tuple into separate variables but I'm only interested in the values for the `color` and `mileage` fields. However, in order for the unpacking expression to succeed, I need to assign all values contained in the tuple to variables. That's where "_" is useful as a placeholder variable:

```
>>> car = ('red', 'auto', 12, 3812.4)
>>> color, _, _, mileage = car

>>> color
'red'
>>> mileage
3812.4
>>> _
12
```

Besides its use as a temporary variable, "_" is a special variable in most Python REPLs that represents the result of the last expression evaluated by the interpreter.

This is handy if you're working in an interpreter session and you'd like to access the result of a previous calculation:

```
>>> 20 + 3
23
>>> _
23
>>> print(_)
23
```

It's also handy if you're constructing objects on the fly and want to interact with them without assigning them a name first:

```
>>> list()
[]
>>> _.append(1)
>>> _.append(2)
>>> _.append(3)
>>> _
[1, 2, 3]
```

Key Takeaways

- **Single Leading Underscore** "`_var`": Naming convention indicating a name is meant for internal use. Generally not enforced by the Python interpreter (except in wildcard imports) and meant as a hint to the programmer only.
- **Single Trailing Underscore** "`var_`": Used by convention to avoid naming conflicts with Python keywords.
- **Double Leading Underscore** "`__var`": Triggers name mangling when used in a class context. Enforced by the Python interpreter.
- **Double Leading and Trailing Underscore** "`__var__`": Indicates special methods defined by the Python language. Avoid this naming scheme for your own attributes.
- **Single Underscore** "`_`": Sometimes used as a name for temporary or insignificant variables ("don't care"). Also, it represents the result of the last expression in a Python REPL session.

2.5 A Shocking Truth About String Formatting

Remember the Zen of Python and how there should be "one obvious way to do something?" You might scratch your head when you find out that there are *four* major ways to do string formatting in Python.

In this chapter I'll demonstrate how these four string formatting approaches work and what their respective strengths and weaknesses are. I'll also give you my simple "rule of thumb" for how I pick the best general-purpose string formatting approach.

Let's jump right in, as we've got a lot to cover. In order to have a simple toy example for experimentation, let's assume we've got the following variables (or constants, really) to work with:

```
>>> errno = 50159747054
>>> name = 'Bob'
```

And based on these variables we'd like to generate an output string with the following error message:

```
'Hey Bob, there is a 0xbadc0ffee error!'
```

Now, *that* error could really spoil a dev's Monday morning! But we're here to discuss string formatting today. So let's get to work.

#1 – "Old Style" String Formatting

Strings in Python have a unique built-in operation that can be accessed with the `%`-operator. It's a shortcut that lets you do simple positional formatting very easily. If you've ever worked with a `printf`-style function in C, you'll instantly recognize how this works. Here's a simple example:

```
>>> 'Hello, %s' % name
'Hello, Bob'
```

I'm using the `%s` format specifier here to tell Python where to substitute the value of `name`, represented as a string. This is called "old style" string formatting.

In old style string formatting there are also other format specifiers available that let you control the output string. For example, it's possible to convert numbers to hexadecimal notation or to add whitespace padding to generate nicely formatted tables and reports.[11]

Here, I'm using the `%x` format specifier to convert an int value to a string and to represent it as a hexadecimal number:

```
>>> '%x' % errno
'badc0ffee'
```

The "old style" string formatting syntax changes slightly if you want to make multiple substitutions in a single string. Because the `%`-operator only takes one argument, you need to wrap the right-hand side in a tuple, like so:

```
>>> 'Hey %s, there is a 0x%x error!' % (name, errno)
'Hey Bob, there is a 0xbadc0ffee error!'
```

It's also possible to refer to variable substitutions by name in your format string, if you pass a mapping to the `%`-operator:

```
>>> 'Hey %(name)s, there is a 0x%(errno)x error!' % {
...     "name": name, "errno": errno }
'Hey Bob, there is a 0xbadc0ffee error!'
```

[11] cf. Python Docs: "printf-style String Formatting"

This makes your format strings easier to maintain and easier to modify in the future. You don't have to worry about making sure the order you're passing in the values matches up with the order the values are referenced in the format string. Of course, the downside is that this technique requires a little more typing.

I'm sure you've been wondering why this `printf`-style formatting is called "old style" string formatting. Well, let me tell you. It was technically superseded by "new style" formatting, which we're going to talk about in a minute. But while "old style" formatting has been deemphasized, it hasn't been deprecated. It is still supported in the latest versions of Python.

#2 – "New Style" String Formatting

Python 3 introduced a new way to do string formatting that was also later back-ported to Python 2.7. This "new style" string formatting gets rid of the `%`-operator special syntax and makes the syntax for string formatting more regular. Formatting is now handled by calling a `format()` function on a string object.[12]

You can use the `format()` function to do simple positional formatting, just like you could with "old style" formatting:

```
>>> 'Hello, {}'.format(name)
'Hello, Bob'
```

Or, you can refer to your variable substitutions by name and use them in any order you want. This is quite a powerful feature as it allows for re-arranging the order of display without changing the arguments passed to the format function:

```
>>> 'Hey {name}, there is a 0x{errno:x} error!'.format(
...     name=name, errno=errno)
'Hey Bob, there is a 0xbadc0ffee error!'
```

[12] cf. Python Docs: "str.format()"

This also shows that the syntax to format an int variable as a hexadecimal string has changed. Now we need to pass a *format spec* by adding a ":x" suffix after the variable name.

Overall, the format string syntax has become more powerful without complicating the simpler use cases. It pays off to read up on this *string formatting mini-language* in the Python documentation.[13]

In Python 3, this "new style" string formatting is preferred over %-style formatting. However, starting with Python 3.6 there's an even better way to format your strings. I'll tell you all about it in the next section.

#3 – Literal String Interpolation (Python 3.6+)

Python 3.6 adds yet another way to format strings, called *Formatted String Literals*. This new way of formatting strings lets you use embedded Python expressions inside string constants. Here's a simple example to give you a feel for the feature:

```
>>> f'Hello, {name}!'
'Hello, Bob!'
```

This new formatting syntax is powerful. Because you can embed arbitrary Python expressions, you can even do inline arithmetic with it, like this:

```
>>> a = 5
>>> b = 10
>>> f'Five plus ten is {a + b} and not {2 * (a + b)}.'

'Five plus ten is 15 and not 30.'
```

Behind the scenes, formatted string literals are a Python parser feature that converts f-strings into a series of string constants and expressions. They then get joined up to build the final string.

[13] cf. Python Docs: "Format String Syntax"

Imagine we had the following `greet()` function that contains an f-string:

```
>>> def greet(name, question):
...     return f"Hello, {name}! How's it {question}?"
...

>>> greet('Bob', 'going')
"Hello, Bob! How's it going?"
```

When we disassemble the function and inspect what's going on behind the scenes, we can see that the f-string in the function gets transformed into something similar to the following:

```
>>> def greet(name, question):
...     return ("Hello, " + name + "! How's it " +
                question + "?")
```

The real implementation is slightly faster than that because it uses the `BUILD_STRING` opcode as an optimization.[14] But functionally they're the same:

```
>>> import dis
>>> dis.dis(greet)
  2     0 LOAD_CONST      1 ('Hello, ')
        2 LOAD_FAST       0 (name)
        4 FORMAT_VALUE    0
        6 LOAD_CONST      2 ("! How's it ")
        8 LOAD_FAST       1 (question)
       10 FORMAT_VALUE    0
       12 LOAD_CONST      3 ('?')
       14 BUILD_STRING    5
       16 RETURN_VALUE
```

[14] cf. Python 3 bug-tracker issue #27078

String literals also support the existing format string syntax of the `str.format()` method. That allows you to solve the same formatting problems we've discussed in the previous two sections:

```
>>> f"Hey {name}, there's a {errno:#x} error!"
"Hey Bob, there's a 0xbadc0ffee error!"
```

Python's new Formatted String Literals are similar to the JavaScript Template Literals added in ES2015. I think they're quite a nice addition to the language, and I've already started using them in my day-to-day Python 3 work. You can learn more about Formatted String Literals in the official Python documentation.[15]

#4 – Template Strings

One more technique for string formatting in Python is Template Strings. It's a simpler and less powerful mechanism, but in some cases this might be exactly what you're looking for.

Let's take a look at a simple greeting example:

```
>>> from string import Template
>>> t = Template('Hey, $name!')
>>> t.substitute(name=name)
'Hey, Bob!'
```

You see here that we need to import the `Template` class from Python's built-in `string` module. Template strings are not a core language feature but they're supplied by a module in the standard library.

Another difference is that template strings don't allow format specifiers. So in order to get our error string example to work, we need to transform our int error number into a hex-string ourselves:

[15] cf. Python Docs: "Formatted string literals"

```
>>> templ_string = 'Hey $name, there is a $error error!'
>>> Template(templ_string).substitute(
...     name=name, error=hex(errno))
'Hey Bob, there is a 0xbadc0ffee error!'
```

That worked great but you're probably wondering when you use template strings in your Python programs. In my opinion, the best use case for template strings is when you're handling format strings generated by users of your program. Due to their reduced complexity, template strings are a safer choice.

The more complex formatting mini-languages of other string formatting techniques might introduce security vulnerabilities to your programs. For example, it's possible for format strings to access arbitrary variables in your program.

That means, if a malicious user can supply a format string they can also potentially leak secret keys and other sensible information! Here's a simple proof of concept of how this attack might be used:

```
>>> SECRET = 'this-is-a-secret'
>>> class Error:
...     def __init__(self):
...         pass
>>> err = Error()
>>> user_input = '{error.__init__.__globals__[SECRET]}'

# Uh-oh...
>>> user_input.format(error=err)
'this-is-a-secret'
```

See how the hypothetical attacker was able to extract our secret string by accessing the `__globals__` dictionary from the format string? Scary, huh! Template Strings close this attack vector, and this makes them a safer choice if you're handling format strings generated from user input:

```
>>> user_input = '${error.__init__.__globals__[SECRET]}'
>>> Template(user_input).substitute(error=err)
ValueError:
"Invalid placeholder in string: line 1, col 1"
```

Which String Formatting Method Should I Use?

I totally get that having so much choice for how to format your strings in Python can feel very confusing. This would be a good time to bust out some flowchart infographic…

But I'm not going to do that. Instead, I'll try to boil it down to the simple rule of thumb that I apply when I'm writing Python.

Here we go—you can use this rule of thumb any time you're having difficulty deciding which string formatting method to use, depending on the circumstances:

Dan's Python String Formatting Rule of Thumb:

> *If your format strings are user-supplied, use Template Strings to avoid security issues. Otherwise, use Literal String Interpolation if you're on Python 3.6+, and "New Style" String Formatting if you're not.*

Key Takeaways

- Perhaps surprisingly, there's more than one way to handle string formatting in Python.
- Each method has its individual pros and cons. Your use case will influence which method you should use.
- If you're having trouble deciding which string formatting method to use, try my *String Formatting Rule of Thumb*.

2.6 “The Zen of Python” Easter Egg

I know what follows is a common sight as far as Python books go. But there’s really no way around Tim Peters’ *Zen of Python*. I’ve benefited from revisiting it over the years, and I think Tim’s words made me a better coder. Hopefully they can do the same for you.

Also, you can tell the *Zen of Python* is a big deal because it’s included as an Easter egg in the language. Just enter a Python interpreter session and run the following:

```
>>> import this
```

The Zen of Python, by Tim Peters

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

Chapter 3

Effective Functions

3.1 Python's Functions Are First-Class

Python's functions are first-class objects. You can assign them to variables, store them in data structures, pass them as arguments to other functions, and even return them as values from other functions.

Grokking these concepts intuitively will make understanding advanced features in Python like lambdas and decorators much easier. It also puts you on a path towards functional programming techniques.

Over the next few pages I'll guide you through a number of examples to help you develop this intuitive understanding. The examples will build on top of each other, so you might want to read them in sequence and even to try out some of them in a Python interpreter session as you go along.

Wrapping your head around the concepts we'll be discussing here might take a little longer than you'd expect. Don't worry—that's completely normal. I've been there. You might feel like you're banging your head against the wall, and then suddenly things will "click" and fall into place when you're ready.

Throughout this chapter I'll be using this `yell` function for demonstration purposes. It's a simple toy example with easily recognizable output:

```
def yell(text):
    return text.upper() + '!'

>>> yell('hello')
'HELLO!'
```

Functions Are Objects

All data in a Python program is represented by objects or relations between objects.[1] Things like strings, lists, modules, and functions are all objects. There's nothing particularly special about functions in Python. They're also just objects.

Because the `yell` function is an *object* in Python, you can assign it to another variable, just like any other object:

```
>>> bark = yell
```

This line doesn't call the function. It takes the function object referenced by `yell` and creates a second name, `bark`, that points to it. You could now also execute the same underlying function object by calling `bark`:

```
>>> bark('woof')
'WOOF!'
```

Function objects and their names are two separate concerns. Here's more proof: You can delete the function's original name (`yell`). Since another name (`bark`) still points to the underlying function, you can still call the function through it:

```
>>> del yell

>>> yell('hello?')
NameError: "name 'yell' is not defined"

>>> bark('hey')
'HEY!'
```

[1] cf. Python Docs: "Objects, values and types"

By the way, Python attaches a string identifier to every function at creation time for debugging purposes. You can access this internal identifier with the `__name__` attribute:[2]

```
>>> bark.__name__
'yell'
```

Now, while the function's `__name__` is still "yell," that doesn't affect how you can access the function object from your code. The name identifier is merely a debugging aid. A *variable pointing to a function* and the *function itself* are really two separate concerns.

Functions Can Be Stored in Data Structures

Since functions are first-class citizens, you can store them in data structures, just like you can with other objects. For example, you can add functions to a list:

```
>>> funcs = [bark, str.lower, str.capitalize]
>>> funcs
[<function yell at 0x10ff96510>,
 <method 'lower' of 'str' objects>,
 <method 'capitalize' of 'str' objects>]
```

Accessing the function objects stored inside the list works like it would with any other type of object:

```
>>> for f in funcs:
...     print(f, f('hey there'))
<function yell at 0x10ff96510> 'HEY THERE!'
<method 'lower' of 'str' objects> 'hey there'
<method 'capitalize' of 'str' objects> 'Hey there'
```

[2] Since Python 3.3 there's also `__qualname__` which serves a similar purpose and provides a *qualified name* string to disambiguate function and class names (cf. PEP 3155).

You can even call a function object stored in the list without first assigning it to a variable. You can do the lookup and then immediately call the resulting "disembodied" function object within a single expression:

```
>>> funcs[0]('heyho')
'HEYHO!'
```

Functions Can Be Passed to Other Functions

Because functions are objects, you can pass them as arguments to other functions. Here's a `greet` function that formats a greeting string using the function object passed to it and then prints it:

```
def greet(func):
    greeting = func('Hi, I am a Python program')
    print(greeting)
```

You can influence the resulting greeting by passing in different functions. Here's what happens if you pass the `bark` function to `greet`:

```
>>> greet(bark)
'HI, I AM A PYTHON PROGRAM!'
```

Of course, you could also define a new function to generate a different flavor of greeting. For example, the following `whisper` function might work better if you don't want your Python programs to sound like Optimus Prime:

```
def whisper(text):
    return text.lower() + '...'

>>> greet(whisper)
'hi, i am a python program...'
```

The ability to pass function objects as arguments to other functions is powerful. It allows you to abstract away and pass around *behavior* in your programs. In this example, the `greet` function stays the same but you can influence its output by passing in different *greeting behaviors*.

Functions that can accept other functions as arguments are also called *higher-order functions*. They are a necessity for the functional programming style.

The classical example for higher-order functions in Python is the built-in `map` function. It takes a function object and an iterable, and then calls the function on each element in the iterable, yielding the results as it goes along.

Here's how you might format a sequence of greetings all at once by *mapping* the `bark` function to them:

```
>>> list(map(bark, ['hello', 'hey', 'hi']))
['HELLO!', 'HEY!', 'HI!']
```

As you saw, `map` went through the entire list and applied the `bark` function to each element. As a result, we now have a new list object with modified greeting strings.

Functions Can Be Nested

Perhaps surprisingly, Python allows functions to be defined inside other functions. These are often called *nested functions* or *inner functions*. Here's an example:

```
def speak(text):
    def whisper(t):
        return t.lower() + '...'
    return whisper(text)

>>> speak('Hello, World')
'hello, world...'
```

Now, what's going on here? Every time you call `speak`, it defines a new inner function `whisper` and then calls it immediately after. My brain's starting to itch just a little here but, all in all, that's still relatively straightforward stuff.

Here's the kicker though—`whisper` *does not exist* outside `speak`:

```
>>> whisper('Yo')
NameError:
"name 'whisper' is not defined"

>>> speak.whisper
AttributeError:
"'function' object has no attribute 'whisper'"
```

But what if you really wanted to access that nested `whisper` function from outside `speak`? Well, functions are objects—you can *return* the inner function to the caller of the parent function.

For example, here's a function defining two inner functions. Depending on the argument passed to top-level function, it selects and returns one of the inner functions to the caller:

```
def get_speak_func(volume):
    def whisper(text):
        return text.lower() + '...'
    def yell(text):
        return text.upper() + '!'
    if volume > 0.5:
        return yell
    else:
        return whisper
```

Notice how `get_speak_func` doesn't actually *call* any of its inner functions—it simply selects the appropriate inner function based on the `volume` argument and then returns the function object:

```
>>> get_speak_func(0.3)
<function get_speak_func.<locals>.whisper at 0x10ae18>

>>> get_speak_func(0.7)
<function get_speak_func.<locals>.yell at 0x1008c8>
```

Of course, you could then go on and call the returned function, either directly or by assigning it to a variable name first:

```
>>> speak_func = get_speak_func(0.7)
>>> speak_func('Hello')
'HELLO!'
```

Let that sink in for a second here... This means not only can functions *accept behaviors* through arguments but they can also *return behaviors*. How cool is that?

You know what, things are starting to get a little loopy here. I'm going to take a quick coffee break before I continue writing (and I suggest you do the same).

Functions Can Capture Local State

You just saw how functions can contain inner functions, and that it's even possible to return these (otherwise hidden) inner functions from the parent function.

Best put on your seat belt now because it's going to get a little crazier still—we're about to enter even deeper functional programming territory. (You had that coffee break, right?)

Not only can functions return other functions, these inner functions can also *capture and carry some of the parent function's state* with them. Well, what does that mean?

I'm going to slightly rewrite the previous `get_speak_func` example to illustrate this. The new version takes a "volume" *and* a "text" argument right away to make the returned function immediately callable:

```
def get_speak_func(text, volume):
    def whisper():
        return text.lower() + '...'
    def yell():
        return text.upper() + '!'
    if volume > 0.5:
        return yell
    else:
        return whisper

>>> get_speak_func('Hello, World', 0.7)()
'HELLO, WORLD!'
```

Take a good look at the inner functions `whisper` and `yell` now. Notice how they no longer have a `text` parameter? But somehow they can still access the `text` parameter defined in the parent function. In fact, they seem to *capture* and "remember" the value of that argument.

Functions that do this are called *lexical closures* (or just *closures*, for short). A closure remembers the values from its enclosing lexical scope even when the program flow is no longer in that scope.

In practical terms, this means not only can functions *return behaviors* but they can also *pre-configure those behaviors*. Here's another barebones example to illustrate this idea:

```
def make_adder(n):
    def add(x):
        return x + n
    return add

>>> plus_3 = make_adder(3)
```

```
>>> plus_5 = make_adder(5)

>>> plus_3(4)
7
>>> plus_5(4)
9
```

In this example, `make_adder` serves as a *factory* to create and configure "adder" functions. Notice how the "adder" functions can still access the `n` argument of the `make_adder` function (the enclosing scope).

Objects Can Behave Like Functions

While all functions are objects in Python, the reverse isn't true. Objects aren't functions. But they can be made *callable*, which allows you to *treat them like functions* in many cases.

If an object is callable it means you can use the round parentheses function call syntax on it and even pass in function call arguments. This is all powered by the `__call__` dunder method. Here's an example of class defining a callable object:

```
class Adder:
    def __init__(self, n):
         self.n = n

    def __call__(self, x):
        return self.n + x

>>> plus_3 = Adder(3)
>>> plus_3(4)
7
```

Behind the scenes, "calling" an object instance as a function attempts to execute the object's `__call__` method.

Of course, not all objects will be callable. That's why there's a built-in `callable` function to check whether an object appears to be callable or not:

```
>>> callable(plus_3)
True
>>> callable(yell)
True
>>> callable('hello')
False
```

Key Takeaways

- Everything in Python is an object, including functions. You can assign them to variables, store them in data structures, and pass or return them to and from other functions (first-class functions.)
- First-class functions allow you to abstract away and pass around behavior in your programs.
- Functions can be nested and they can capture and carry some of the parent function's state with them. Functions that do this are called closures.
- Objects can be made callable. In many cases this allows you to treat them like functions.

3.2 Lambdas Are Single-Expression Functions

The `lambda` keyword in Python provides a shortcut for declaring small anonymous functions. Lambda functions behave just like regular functions declared with the `def` keyword. They can be used whenever function objects are required.

For example, this is how you'd define a simple lambda function carrying out an addition:

```
>>> add = lambda x, y: x + y
>>> add(5, 3)
8
```

You could declare the same `add` function with the `def` keyword, but it would be slightly more verbose:

```
>>> def add(x, y):
...     return x + y
>>> add(5, 3)
8
```

Now you might be wondering, "Why the big fuss about lambdas? If they're just a slightly more concise version of declaring functions with `def`, what's the big deal?"

Take a look at the following example and keep the words *function expression* in your head while you do that:

```
>>> (lambda x, y: x + y)(5, 3)
8
```

Okay, what happened here? I just used `lambda` to define an "add" function inline and then immediately called it with the arguments `5` and `3`.

Conceptually, the *lambda expression* `lambda x, y: x + y` is the same as declaring a function with `def`, but just written inline. The key difference here is that I didn't have to bind the function object to a name before I used it. I simply stated the expression I wanted to compute as part of a lambda, and then immediately evaluated it by calling the lambda expression like a regular function.

Before you move on, you might want to play with the previous code example a little to really let the meaning of it sink in. I still remember this taking me awhile to wrap my head around. So don't worry about spending a few minutes in an interpreter session on this. It'll be worth it.

There's another syntactic difference between lambdas and regular function definitions. Lambda functions are restricted to a single expression. This means a lambda function can't use statements or annotations—not even a `return` statement.

How do you return values from lambdas then? Executing a lambda function evaluates its expression and then automatically returns the expression's result, so there's always an *implicit* return statement. That's why some people refer to lambdas as *single expression functions*.

Lambdas You Can Use

When should you use lambda functions in your code? Technically, any time you're expected to supply a function object you can use a lambda expression. And because lambdas can be anonymous, you don't even need to assign them to a name first.

This can provide a handy and "unbureaucratic" shortcut to defining a function in Python. My most frequent use case for lambdas is writing short and concise *key funcs* for sorting iterables by an alternate key:

```
>>> tuples = [(1, 'd'), (2, 'b'), (4, 'a'), (3, 'c')]
>>> sorted(tuples, key=lambda x: x[1])
[(4, 'a'), (2, 'b'), (3, 'c'), (1, 'd')]
```

In the above example, we're sorting a list of tuples by the second value in each tuple. In this case, the lambda function provides a quick way to modify the sort order. Here's another sorting example you can play with:

```
>>> sorted(range(-5, 6), key=lambda x: x * x)
[0, -1, 1, -2, 2, -3, 3, -4, 4, -5, 5]
```

Both examples I showed you have more concise implementations in Python using the built-in `operator.itemgetter()` and `abs()` functions. But I hope you can see how using a lambda gives you much more flexibility. Want to sort a sequence by some arbitrary computed key? No problem. Now you know how to do it.

Here's another interesting thing about lambdas: Just like regular nested functions, lambdas also work as *lexical closures*.

What's a lexical closure? It's just a fancy name for a function that remembers the values from the enclosing lexical scope even when the program flow is no longer in that scope. Here's a (fairly academic) example to illustrate the idea:

```
>>> def make_adder(n):
...     return lambda x: x + n

>>> plus_3 = make_adder(3)
>>> plus_5 = make_adder(5)

>>> plus_3(4)
7
>>> plus_5(4)
9
```

In the above example, the `x + n` lambda can still access the value of `n` even though it was defined in the `make_adder` function (the enclosing scope).

Sometimes, using a lambda function instead of a nested function declared with the `def` keyword can express the programmer's intent more clearly. But to be honest, this isn't a common occurrence—at least not in the kind of code that I like to write. So let's talk a little more about that.

But Maybe You Shouldn't...

On the one hand, I'm hoping this chapter got you interested in exploring Python's lambda functions. On the other hand, I feel like it's time to put up another caveat: Lambda functions should be used sparingly and with extraordinary care.

I know I've written my fair share of code using lambdas that looked "cool" but were actually a liability for me and my coworkers. If you're tempted to use a lambda, spend a few seconds (or minutes) to think if it is really the cleanest and most maintainable way to achieve the desired result.

For example, doing something like this to save two lines of code is just silly. Sure, technically it works and it's a nice enough "trick." But it's also going to confuse the next gal or guy that has to ship a bugfix under a tight deadline:

```
# Harmful:
>>> class Car:
...     rev = lambda self: print('Wroom!')
...     crash = lambda self: print('Boom!')

>>> my_car = Car()
>>> my_car.crash()
'Boom!'
```

I have similar feelings about complicated `map()` or `filter()` constructs using lambdas. Usually it's much cleaner to go with a list comprehension or generator expression:

```
# Harmful:
>>> list(filter(lambda x: x % 2 == 0, range(16)))
[0, 2, 4, 6, 8, 10, 12, 14]

# Better:
>>> [x for x in range(16) if x % 2 == 0]
[0, 2, 4, 6, 8, 10, 12, 14]
```

If you find yourself doing anything remotely complex with lambda expressions, consider defining a standalone function with a proper name instead.

Saving a few keystrokes won't matter in the long run, but your colleagues (and your future self) will appreciate clean and readable code more than terse wizardry.

Key Takeaways

- Lambda functions are single-expression functions that are not necessarily bound to a name (anonymous).
- Lambda functions can't use regular Python statements and always include an implicit `return` statement.
- Always ask yourself: *Would using a regular (named) function or a list comprehension offer more clarity?*

3.3 The Power of Decorators

At their core, Python's decorators allow you to extend and modify the behavior of a callable (functions, methods, and classes) *without* permanently modifying the callable itself.

Any sufficiently generic functionality you can tack on to an existing class or function's behavior makes a great use case for decoration. This includes the following:

- logging
- enforcing access control and authentication
- instrumentation and timing functions
- rate-limiting
- caching, and more

Now, why should you master the use of decorators in Python? After all, what I just mentioned sounded quite abstract, and it might be difficult to see how decorators can benefit you in your day-to-day work as a Python developer. Let me try to bring some clarity to this question by giving you a somewhat real-world example:

Imagine you've got 30 functions with business logic in your report-generating program. One rainy Monday morning your boss walks up to your desk and says: *"Happy Monday! Remember those TPS reports? I need you to add input/output logging to each step in the report generator. XYZ Corp needs it for auditing purposes. Oh, and I told them we can ship this by Wednesday."*

Depending on whether or not you've got a solid grasp on Python's decorators, this request will either send your blood pressure spiking or leave you relatively calm.

Without decorators you might be spending the next three days scrambling to modify each of those 30 functions and clutter them up with manual logging calls. Fun times, right?

If you do know your decorators however, you'll calmly smile at your boss and say: *"Don't worry Jim, I'll get it done by 2pm today."*

Right after that you'll type the code for a generic `@audit_log` decorator (that's only about 10 lines long) and quickly paste it in front of each function definition. Then you'll commit your code and grab another cup of coffee...

I'm dramatizing here, but only a little. Decorators *can be* that powerful. I'd go as far as to say that understanding decorators is a milestone for any serious Python programmer. They require a solid grasp of several advanced concepts in the language, including the properties of *first-class functions*.

I believe that the payoff for understanding how decorators work in Python can be enormous.

Sure, decorators are relatively complicated to wrap your head around for the first time, but they're a highly useful feature that you'll often encounter in third-party frameworks and the Python standard library. Explaining decorators is also a *make or break* moment for any good Python tutorial. I'll do my best here to introduce you to them step by step.

Before you dive in however, now would be an excellent moment to refresh your memory on the properties of *first-class functions* in Python. There's a chapter on them in this book, and I would encourage you to take a few minutes to review it. The most important "first-class functions" takeaways for understanding decorators are:

- **Functions are objects**—they can be assigned to variables and passed to and returned from other functions
- **Functions can be defined inside other functions**—and a child function can capture the parent function's local state (lexical closures)

Alright, are you ready to do this? Let's get started.

Python Decorator Basics

Now, what are decorators really? They "decorate" or "wrap" another function and let you execute code before and after the wrapped function runs.

Decorators allow you to define reusable building blocks that can change or extend the behavior of other functions. And, they let you do that without permanently modifying the wrapped function itself. The function's behavior changes only when it's *decorated*.

What might the implementation of a simple decorator look like? In basic terms, a decorator is *a callable that takes a callable as input and returns another callable*.

The following function has that property and could be considered the simplest decorator you could possibly write:

```
def null_decorator(func):
    return func
```

As you can see, `null_decorator` is a callable (it's a function), it takes another callable as its input, and it returns the same input callable without modifying it.

Let's use it to *decorate* (or *wrap*) another function:

```
def greet():
    return 'Hello!'

greet = null_decorator(greet)

>>> greet()
'Hello!'
```

In this example, I've defined a `greet` function and then immediately decorated it by running it through the `null_decorator` function. I

know this doesn't look very useful yet. I mean, we specifically designed the null decorator to be useless, right? But in a moment this example will clarify how Python's special-case decorator syntax works.

Instead of explicitly calling `null_decorator` on `greet` and then reassigning the `greet` variable, you can use Python's `@` syntax for decorating a function more conveniently:

```
@null_decorator
def greet():
    return 'Hello!'

>>> greet()
'Hello!'
```

Putting an `@null_decorator` line in front of the function definition is the same as defining the function first and then running through the decorator. Using the `@` syntax is just *syntactic sugar* and a shortcut for this commonly used pattern.

Note that using the `@` syntax decorates the function immediately at definition time. This makes it difficult to access the undecorated original without brittle hacks. Therefore you might choose to decorate some functions manually in order to retain the ability to call the undecorated function as well.

Decorators Can Modify Behavior

Now that you're a little more familiar with the decorator syntax, let's write another decorator that *actually does something* and modifies the behavior of the decorated function.

Here's a slightly more complex decorator which converts the result of the decorated function to uppercase letters:

```
def uppercase(func):
    def wrapper():
        original_result = func()
        modified_result = original_result.upper()
        return modified_result
    return wrapper
```

Instead of simply returning the input function like the null decorator did, this `uppercase` decorator defines a new function on the fly (a closure) and uses it to *wrap* the input function in order to modify its behavior at call time.

The `wrapper` closure has access to the undecorated input function and it is free to execute additional code before and after calling the input function. (Technically, it doesn't even need to call the input function at all.)

Note how, up until now, the decorated function has never been executed. Actually calling the input function at this point wouldn't make any sense—you'll want the decorator to be able to modify the behavior of its input function when it eventually gets called.

You might want to let that sink in for a minute or two. I know how complicated this stuff can seem, but we'll get it sorted out together, I promise.

Time to see the `uppercase` decorator in action. What happens if you decorate the original `greet` function with it?

```
@uppercase
def greet():
    return 'Hello!'

>>> greet()
'HELLO!'
```

I hope this was the result you expected. Let's take a closer look at what just happened here. Unlike `null_decorator`, our `uppercase` decorator returns a *different function object* when it decorates a function:

```
>>> greet
<function greet at 0x10e9f0950>

>>> null_decorator(greet)
<function greet at 0x10e9f0950>

>>> uppercase(greet)
<function uppercase.<locals>.wrapper at 0x76da02f28>
```

And as you saw earlier, it needs to do that in order to modify the behavior of the decorated function when it finally gets called. The `uppercase` decorator is a function itself. And the only way to influence the "future behavior" of an input function it decorates is to replace (or *wrap*) the input function with a closure.

That's why `uppercase` defines and returns another function (the closure) that can then be called at a later time, run the original input function, and modify its result.

Decorators modify the behavior of a callable through a wrapper closure so you don't have to permanently modify the original. The original callable isn't permanently modified—its behavior changes only when decorated.

This let's you tack on reusable building blocks, like logging and other instrumentation, to existing functions and classes. It makes decorators such a powerful feature in Python that it's frequently used in the standard library and in third-party packages.

A Quick Intermission

By the way, if you feel like you need a quick coffee break or a walk around the block at this point—that's totally normal. In my opinion

closures and decorators are some of the most difficult concepts to understand in Python.

Please, take your time and don't worry about figuring this out immediately. Playing through the code examples in an interpreter session one by one often helps make things sink in.

I know you can do it!

Applying Multiple Decorators to a Function

Perhaps not surprisingly, you can apply more than one decorator to a function. This accumulates their effects and it's what makes decorators so helpful as reusable building blocks.

Here's an example. The following two decorators wrap the output string of the decorated function in HTML tags. By looking at how the tags are nested, you can see which order Python uses to apply multiple decorators:

```
def strong(func):
    def wrapper():
        return '<strong>' + func() + '</strong>'
    return wrapper

def emphasis(func):
    def wrapper():
        return '<em>' + func() + '</em>'
    return wrapper
```

Now let's take these two decorators and apply them to our `greet` function at the same time. You can use the regular `@` syntax for that and just "stack" multiple decorators on top of a single function:

```
@strong
@emphasis
```

```
def greet():
    return 'Hello!'
```

What output do you expect to see if you run the decorated function? Will the `@emphasis` decorator add its `<em>` tag first, or does `@strong` have precedence? Here's what happens when you call the decorated function:

```
>>> greet()
'<strong><em>Hello!</em></strong>'
```

This clearly shows in what order the decorators were applied: from *bottom to top*. First, the input function was wrapped by the `@emphasis` decorator, and then the resulting (decorated) function got wrapped again by the `@strong` decorator.

To help me remember this bottom to top order, I like to call this behavior *decorator stacking*. You start building the stack at the bottom and then keep adding new blocks on top to work your way upwards.

If you break down the above example and avoid the `@` syntax to apply the decorators, the chain of decorator function calls looks like this:

```
decorated_greet = strong(emphasis(greet))
```

Again, you can see that the `emphasis` decorator is applied first and then the resulting wrapped function is wrapped again by the `strong` decorator.

This also means that deep levels of decorator stacking will evenutally have an effect on performance because they keep adding nested function calls. In practice, this usually won't be a problem, but it's something to keep in mind if you're working on performance-intensive code that frequently uses decoration.

Decorating Functions That Accept Arguments

All examples so far only decorated a simple *nullary* `greet` function that didn't take any arguments whatsoever. Up until now, the decorators you saw here didn't have to deal with forwarding arguments to the input function.

If you try to apply one of these decorators to a function that takes arguments, it will not work correctly. How do you decorate a function that takes arbitrary arguments?

This is where Python's `*args` and `**kwargs` feature[3] for dealing with variable numbers of arguments comes in handy. The following `proxy` decorator takes advantage of that:

```
def proxy(func):
    def wrapper(*args, **kwargs):
        return func(*args, **kwargs)
    return wrapper
```

There are two notable things going on with this decorator:

- It uses the `*` and `**` operators in the `wrapper` closure definition to collect all positional and keyword arguments and stores them in variables (`args` and `kwargs`).
- The `wrapper` closure then forwards the collected arguments to the original input function using the `*` and `**` "argument unpacking" operators.

It's a bit unfortunate that the meaning of the star and double-star operators is overloaded and changes depending on the context they're used in, but I hope you get the idea.

[3] cf. "Fun With *args and **kwargs" chapter

Let's expand the technique laid out by the `proxy` decorator into a more useful practical example. Here's a `trace` decorator that logs function arguments and results during execution time:

```
def trace(func):
    def wrapper(*args, **kwargs):
        print(f'TRACE: calling {func.__name__}() '
              f'with {args}, {kwargs}')

        original_result = func(*args, **kwargs)

        print(f'TRACE: {func.__name__}() '
              f'returned {original_result!r}')

        return original_result
    return wrapper
```

Decorating a function with `trace` and then calling it will print the arguments passed to the decorated function and its return value. This is still somewhat of a "toy" example—but in a pinch it makes a great debugging aid:

```
@trace
def say(name, line):
    return f'{name}: {line}'

>>> say('Jane', 'Hello, World')
'TRACE: calling say() with ("Jane", "Hello, World"), {}'
'TRACE: say() returned "Jane: Hello, World"'
'Jane: Hello, World'
```

Speaking of debugging, there are some things you should keep in mind when debugging decorators:

How to Write "Debuggable" Decorators

When you use a decorator, really what you're doing is replacing one function with another. One downside of this process is that it "hides" some of the metadata attached to the original (undecorated) function.

For example, the original function name, its docstring, and parameter list are hidden by the wrapper closure:

```
def greet():
    """Return a friendly greeting."""
    return 'Hello!'

decorated_greet = uppercase(greet)
```

If you try to access any of that function metadata, you'll see the wrapper closure's metadata instead:

```
>>> greet.__name__
'greet'
>>> greet.__doc__
'Return a friendly greeting.'

>>> decorated_greet.__name__
'wrapper'
>>> decorated_greet.__doc__
None
```

This makes debugging and working with the Python interpreter awkward and challenging. Thankfully there's a quick fix for this: the `functools.wraps` decorator included in Python's standard library.[4]

You can use `functools.wraps` in your own decorators to copy over the lost metadata from the undecorated function to the decorator closure. Here's an example:

[4] cf. Python Docs: "functools.wraps"

```
import functools

def uppercase(func):
    @functools.wraps(func)
    def wrapper():
        return func().upper()
    return wrapper
```

Applying `functools.wraps` to the wrapper closure returned by the decorator carries over the docstring and other metadata of the input function:

```
@uppercase
def greet():
    """Return a friendly greeting."""
    return 'Hello!'

>>> greet.__name__
'greet'
>>> greet.__doc__
'Return a friendly greeting.'
```

As a best practice, I'd recommend that you use `functools.wraps` in all of the decorators you write yourself. It doesn't take much time and it will save you (and others) debugging headaches down the road.

Oh, and congratulations—you've made it all the way to the end of this complicated chapter and learned a whole lot about decorators in Python. Great job!

Key Takeaways

- Decorators define reusable building blocks you can apply to a callable to modify its behavior without permanently modifying the callable itself.

- The `@` syntax is just a shorthand for calling the decorator on an input function. Multiple decorators on a single function are applied bottom to top (*decorator stacking*).
- As a debugging best practice, use the `functools.wraps` helper in your own decorators to carry over metadata from the undecorated callable to the decorated one.
- Just like any other tool in the software development toolbox, decorators are not a cure-all and they should not be overused. It's important to balance the need to "get stuff done" with the goal of "not getting tangled up in a horrible, unmaintainable mess of a code base."