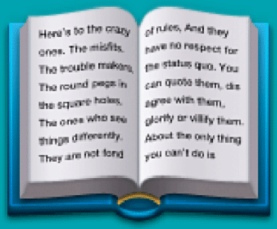
Python Tricks The Book



by Dan Bader

Dan Bader
Improve Your Python Skills

Python Tricks: The Book

Dan Bader

Cover design by Anja Pircher Design (anjapircher.com)

If you'd like to let me know about an error, or if you just have a question, or want to offer some constructive feedback, email me at mail@dbader.org.

Updated 2017-04-04 Thanks to Michael Howitz, Johnathan Willitts, Julian Orbach, Johnny Giorgis, Bob White, Daniel Meyer, Michael Stueben, and Smital Desai 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 email 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

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 serves as a motivating example 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 a raving response and were shared and retweeted for days on end.

More and more developers started asking me for a way to "get the whole series." 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. Just a fun little Twitter experiment.

But from these inquiries I got the sense that these 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 next 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. These Python Tricks were just code screenshots, I thought. But some people were getting a lot of value out of them.

Therefore I decided to double down on my Python Tricks experiment and expanded it into a series of around 30 emails. These were still just a headline and a code screenshot each, and I soon found the limits of that format. 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 also more accessible. I sat down to re-create the whole series of Python Tricks emails with a plain text version and 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 to play 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 motivating examples by themselves—but 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 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 high.
- A book that takes you by the hand to guide 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

The 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 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, this book alone won't be able to turn you into a professional Python developer. Reading the book 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 take it 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 that.

Of course, you can also read through all 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 final page.

Some of these tricks will be easy to understand in passing—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 the book.

Chapter 2

Language Features (and Quirks)

2.1 Object Comparisons: “is” vs “==”

When I was a kid, our neighbors had two twin cats. They looked seemingly identical—same charcoal fur, same piercing green eyes. Some personality quirks aside, you couldn’t tell them apart just from looking at them. But of course they were two different cats, two separate beings, even though they looked exactly the same.

There’s a difference in meaning between *equal* and *identical*. And this difference is important when you want to understand how Python’s `is` and == comparison operators behave.

The == operator compares by checking for *equality*: if these cats were Python objects and we’d compare them with the == operator, we’d get “both cats are equal” as an answer.

The `is` operator, however, compares *identities*: if we compared our cats with the `is` operator, we’d get “these are two different cats” as an answer.

But before I get all tangled up in this ball of twine of a cat analogy, let’s take a look at some real Python code.

First, we’ll create a new list object and name it `a`, and then define another variable `b` that points to the same list object:

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

Let’s inspect these two variables. We can see they point to identical looking lists:

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

```
>>> b
[1, 2, 3]
```

Because the two list objects look the same we'll get the expected result when we compare them for equality using the == operator:

```
>>> a == b
True
```

However, that doesn't tell us whether `a` and `b` are actually pointing to the same object. Of course, we know they do because we assigned them earlier, but suppose we didn't know—how might we find out?

The answer is comparing both variables with the `is` operator. This confirms both variables are in fact pointing to one list object:

```
>>> a is b
True
```

Let's see what happens when we create an identical copy of our list object. We can do that by calling `list()` on the existing list to create a copy we'll name `c`:

```
>>> c = list(a)
```

Again you'll see that the new list we just created looks identical to the list object pointed to by `a` and `b`:

```
>>> c
[1, 2, 3]
```

Now this is where it gets interesting—let's compare our list copy `c` with the initial list `a` using the `==` operator. What answer do you expect to see?

```
>>> a == c
True
```

Okay, I hope this was what you expected. What this result tells us is that `c` and `a` have the same contents. They're considered equal by Python. But are they actually pointing to the same object? Let's find out with the `is` operator:

```
>>> a is c
False
```

Boom—this is where we get a different result. Python is telling us that `c` and `a` are pointing to two different objects, even though their contents might be the same.

So, to recap let's try and break the difference between `is` and `==` down to two short definitions:

- An `is` expression evaluates to `True` if two variables point to the same (identical) object.
- An `==` expression evaluates to `True` if the objects referred to by the variables are equal (have the same contents).

Just remember, think of twin cats (dogs should work, too) whenever you need to decide between using `is` and `==` in Python. You'll be fine.

2.2 The Craziest Dict Expression in the West

Sometimes you strike upon a one-line code example that has real depth to it, one that can teach a lot about a programming language. Such an example feels like a *Zen kōan*: a question or statement used in Zen practice to provoke doubt and test the student's progress.

I believe the Python Trick in this chapter is such an example. It seems like a straightforward dictionary expression, but when considered at close range it leads you down a mind-expanding rabbit hole in the CPython interpreter.

Take a moment to reflect on the following dict expression and what it will evaluate to:

```
>>> {True: 'yes', 1: 'no', 1.0: 'maybe'}
```

I'll wait here...

Ok, ready?

This is the result we get when evaluating the dict expression in a CPython interpreter session:

```
>>> {True: 'yes', 1: 'no', 1.0: 'maybe'}
{True: 'maybe'}
```

I'll admit I was pretty surprised about this result the first time I saw it. But it all makes sense when you investigate what happens step

by step. So, let's think about why we get this—I want to say *slightly unintuitive*—result.

When Python processes our dictionary expression it first constructs a new empty dictionary object; and then assigns the keys and values to it in the order given in the dict expression.

Therefore, when we break it down our dict expression is equivalent to this sequence of statements that are executed in order:

```
>>> xs = dict()
>>> xs[True] = 'yes'
>>> xs[1] = 'no'
>>> xs[1.0] = 'maybe'
```

This gets a lot more interesting when we realize that *all dictionary keys* we're using in this example are considered to be *equal* by Python:

```
>>> True == 1 == 1.0
True
```

Okay, but—wait a minute here. We can intuitively accept that `1.0 == 1`, but why would `True` be considered equal to `1` as well?

The answer to that is Python treats `bool` as a subclass of `int`[1]. This is the case in Python 2 and Python 3:

> "The Boolean type is a subtype of the integer type, and Boolean values behave like the values 0 and 1, respectively, in almost all contexts, the exception being that

[1]Yes, this also means you can technically use bools as indexes into a list or tuple in Python: `['no', 'yes'][True] == 'yes'`. But you probably shouldn't do it for the sake of clarity (and the sanity of your colleagues).

> when converted to a string, the strings 'False' or 'True' are returned, respectively."[2]

So, as far as Python is concerned, `True`, `1`, and `1.0` all represent *the same dictionary key*. As the interpreter evaluates the dictionary expression it repeatedly overwrites the value for the key `True`. This explains why the resulting dictionary only contains a single key.

Before we move on let's have a look at our *mystery dict expression* again:

```
>>> {True: 'yes', 1: 'no', 1.0: 'maybe'}
{True: 'maybe'}
```

But why do we still get `True` as the key here? Shouldn't the key also change to `1.0` at the end through the repeated assignments?

To explain this outcome we need to know that Python doesn't replace the object instance for a key when a new value is assigned to it:

```
>>> ys = {1.0: 'no'}
>>> ys[True] = 'yes'
>>> ys
{1.0: 'yes'}
```

This is done as a performance optimization—if the keys are considered identical then why spend time updating the original? From this example we saw that the initial `True` object is never replaced as the key. Therefore the dictionary's string representation still prints the key as `True` (instead of `1` or `1.0`).

[2] cf. Python Docs: "The Standard Type Hierarchy"

At this point we might ask ourselves "Wait, aren't Python dictionaries hash maps, or something?"

With what we know now it looks like the values in the resulting dict are getting overwritten only because they compare as equal. However, it turns out that this effect isn't caused by the `__eq__` equality check alone, either.

Let's define the following class as our little detective tool:

```
class AlwaysEquals:
    def __eq__(self, other):
        return True

    def __hash__(self):
        return id(self)
```

This class is special in two ways.

First, because we implemented the `__eq__` magic method to always return `True`, all instances of this class will pretend they're equal to *any* other object:

```
>>> AlwaysEquals() == AlwaysEquals()
True
>>> AlwaysEquals() == 42
True
>>> AlwaysEquals() == 'waaat?'
True
```

Second, each instance will also each return a unique hash value generated by the built-in `id()` function[3]:

[3] In CPython `id()` returns the address of the object in memory, which is guaranteed to be unique.

```
>>> objects = [AlwaysEquals(),
               AlwaysEquals(),
               AlwaysEquals()]
>>> [hash(obj) for obj in objects]
[4574298968, 4574287912, 4574287072]
```

That'll allow us to test if dictionary keys are overwritten based on their equality comparison result alone. And you'll see that the keys are *not* getting overwritten even though they always compare as equal:

```
>>> {AlwaysEquals(): 'yes', AlwaysEquals(): 'no'}
{ <AlwaysEquals object at 0x110a3c588>: 'yes',
  <AlwaysEquals object at 0x110a3cf98>: 'no' }
```

We can also flip this idea around and check to see if returning the same hash value is enough to cause keys to get overwritten:

```
>>> class SameHash:
>>>     def __hash__(self):
>>>         """Same hash code for each object."""
>>>         return 1

>>> a = SameHash()
>>> b = SameHash()

>>> a == b
False

>>> hash(a), hash(b)
(1, 1)
```

```
>>> print({a: 'a', b: 'b'})
{ <SameHash instance at 0x7f7159020cb0>: 'a',
  <SameHash instance at 0x7f7159020cf8>: 'b' }
```

As this example shows, the "keys get overwritten" effect isn't caused by the hash value alone either.

A better explanation of what's going on here is that dictionaries check for equality *and* compare the hash value to determine if two keys are the same:

> "An object is hashable if it has a hash value which never changes during its lifetime (it needs a `__hash__()` method), and can be compared to other objects (it needs an `__eq__()` method). Hashable objects which compare equal must have the same hash value. Hashability makes an object usable as a dictionary key and a set member, because these data structures use the hash value internally." [4]

To try and summarize the findings of our investigation:

`{True: 'yes', 1: 'no', 1.0: 'maybe'}` evaluates to `{True: 'maybe'}` because the keys `True`, `1`, and `1.0` all compare as equal *and* they all have the same hash value.

```
>>> True == 1 == 1.0
True

>>> (hash(True), hash(1), hash(1.0))
(1, 1, 1)
```

[4] cf. Python Glossary: "Hashable"

That's how we end up with this—not so surprising anymore—result as the dictionary's final state:

```
>>> {True: 'yes', 1: 'no', 1.0: 'maybe'}
{True: 'maybe'}
```

This particular Python Trick can be be a bit mind-boggling at first (that's why I jokingly compared it to a Zen kōan).

If it's difficult to understand what's going on in this chapter, try playing through the code examples one by one in a Python interpreter session. Expanded knowledge of Python awaits you on the other side.

Key Takeaways

- Unexpected dictionary key collisions will lead to surprising results.
- Dictionaries treat keys as identical if their `__eq__` comparison result says they're equal and their hash values are the same.
- `True == 1 == 1.0` and `hash(True) == hash(1) == hash(1.0)`

2.3 Comprehending Comprehensions

One of my favorite features in Python are list comprehensions. They can seem a bit arcane at first but when you break them down they are actually a very simple construct.

The key to understanding list comprehensions is that they're just `for`-loops over a collection expressed in a more terse and compact syntax. Let's take the following list comprehension as an example:

```
>>> squares = [x * x for x in range(10)]
```

It computes a list of all integer square numbers from 0 to 9:

```
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

If we wanted to build the same list using a plain `for`-loop we'd probably write something like this:

```
>>> squares = []
>>> for x in range(10):
...     squares.append(x * x)
```

That's a pretty straightforward loop, right? If you try and generalize some of this structure you might end up with a template similar to this:

```
(values) = [ (expression) for (value) in (collection) ]
```

The above list comprehension is equivalent to the following plain `for`-loop:

```
(values) = []
for (value) in (collection):
    (values).append( (expression) )
```

Again, a fairly simple cookiecutter pattern you can apply to most for loops. Now there's one more useful element we need to add to this template, and that is element filtering with *conditions*.

List comprehensions can filter values based on some arbitrary condition that decides whether or not the resulting value becomes a part of the output list. Here's an example:

```
>>> even_squares = [x * x for x in range(10)
                    if x % 2 == 0]
```

This list comprehension will compute a list of the squares of all even integers from 0 to 9.

If you're not familiar with what the *modulo* (%) operator does—it returns the remainder after division of one number by another. In this example the %-operator gives us an easy way to test if a number is even by checking the remainder after we divide the number by 2.

```
>>> even_squares
[0, 4, 16, 36, 64]
```

Similarly to the first example, this new list comprehension can be transformed into an equivalent `for`-loop:

```
even_squares = []
for x in range(10):
    if x % 2 == 0:
        even_squares.append(x * x)
```

Let's try and generalize the above *list comprehension to for-loop* transform again. This time we're going to add a filter condition to our template to decide which values end up in the resulting list.

Here's the list comprehension template:

```
values = [expression
          for value in collection
          if condition]
```

And we can transform this list comprehension into a *for*-loop with the following pattern:

```
vals = []
for value in collection:
    if condition:
        vals.append(expression)
```

Again, this is a straightforward transformation—we simply apply our cookiecutter pattern again. I hope this dispelled some of the "magic" in how list comprehensions work. They're really quite a useful tool.

Before you move on I want to point out that Python not only supports *list* comprehensions but also has similar syntax for *sets* and *dictionaries*.

Here's what a *set comprehension* looks like:

```
>>> { x * x for x in range(-9, 10) }
set([64, 1, 36, 0, 49, 9, 16, 81, 25, 4])
```

And this is a *dict comprehension*:

```
>>> { x: x * x for x in range(5) }
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

Both are useful tools in practice. There's one caveat to Python's comprehensions—as you get more proficient at using them it becomes easier and easier to write code that's difficult to read. If you're not careful you might have to deal with monstrous list, set, dict comprehensions soon. Remember, too much of a good thing is usually a bad thing.

After much chagrin I'm personally drawing the line at one level of nesting for comprehensions. I found that in most cases it's better (as in "more readable" and "easier to maintain") to use *for*-loops beyond that point.

Key Takeaways

- Comprehensions are a key feature in Python. Understanding and applying them will make your code much more Pythonic.
- Comprehensions are just fancy syntax for a simple `for`-loop pattern. Once you understand the pattern, you'll develop an intuitive understanding for comprehensions.
- There are more than just list comprehensions.

2.4 List Slicing Tricks and the Sushi Operator

Python's `:` list slicing syntax is commonly used to access ranges of elements within a list—hence the name list *slicing*. Personally, I also like to call `:` the *sushi operator*. It looks like a delicious maki roll cut in half.

Besides reminding you of delicious food and accessing ranges of lists, there are a few more lesser-known applications for list slicing. The list slicing operator can be used without indices for a few fun and useful things that I'm going to show you now.

Here's the first list slicing trick: You can use the `:`-operator to clear all elements from a list without destroying the list object itself.

This is extremely helpful when you need to clear out a list that has other references pointing to it in your program. In this case you often can't just empty the list by replacing it with a new list object, as that wouldn't update the other references. But here's the sushi operator coming to your rescue:

```
>>> lst = [1, 2, 3, 4, 5]
>>> del lst[:]
>>> lst
[]
```

As you can see, this removes all elements from `lst` but leaves the list object itself intact. In Python 3 you can also use `lst.clear()` for the same job, which might be the more readable pattern depending on the circumstances. However keep in mind that `clear()` isn't available in Python 2.

Besides clearing lists, you can also use `:` list slicing to replace all ele-

ments of a list without creating a new list object. This is a nice shorthand for clearing a list and then re-populating it manually:

```
>>> original_lst = lst
>>> lst[:] = [7, 8, 9]
>>> lst
[7, 8, 9]
>>> original_lst
[7, 8, 9]
>>> original_lst is lst
True
```

The previous code example replaced all elements in `lst` but did not destroy and re-create the list itself—the old references to the original list object are therefore still valid.

Yet another use case for the `:`-operator is creating (shallow) copies of existing lists:

```
>>> copied_lst = lst[:]
>>> copied_lst
[7, 8, 9]
>>> copied_lst is lst
False
```

In this case a *shallow* copy means that the only the structure of the elements is copied, not the elements themselves. Both copies of the list share the instances of the individual elements.

If you need to duplicate everything, including the elements, then you'll need to create a *deep* copy of the list. Python's built-in `copy` module will come in handy for this.

Key Takeaways

- The `:` "sushi operator" is not only useful to select sublists of elements within a list. It can also be used to clear and to copy lists.
- But careful, this functionality is bordering on the arcane for many Python developers. Using it might make your code less maintainable for the rest of your time.

2.5 Dictionary Pretty-Printing

Have you ever tried hunting down a bug in one of your programs by sprinkling a bunch of debug "print" statements to trace the execution flow? Or maybe you needed to generate a log message to print some configuration settings...

I have—and I've often been frustrated with how difficult some data structures are to read in Python when they're printed as text strings. For example, here's a simple dictionary. Printed in an interpreter session the key order is arbitrary and there's no indentation to the resulting string:

```
>>> mapping = {'a': 23, 'b': 42, 'c': 0xc0ffee}
>>> mapping
{'b': 42, 'c': 12648430, 'a': 23}
```

Luckily there are some easy to use alternatives to a straight up *to-str* conversion that give a more readable result. One option is using Python's built-in `json` module. You can use `json.dumps()` to pretty-print Python dicts with nicer formatting:

```
>>> import json
>>> print(json.dumps(mapping, indent=4, sort_keys=True))
```

```
{
    "a": 23,
    "b": 42,
    "c": 12648430
}
```

While this looks nice and readable, it isn't a perfect solution. Printing dictionaries with the `json` module only works with dicts containing

primitive types—you'll run into trouble trying to print a dictionary containing a non-primitive data type, like a function:

```
>>> json.dumps({all: 'yup'})
TypeError: "keys must be a string"
```

Another downside of using `json.dumps()` is that it can't stringify complex data types like *sets*:

```
>>> mapping['d'] = {1, 2, 3}
>>> json.dumps(mapping)
TypeError: "set([1, 2, 3]) is not JSON serializable"
```

Also you might run into trouble with how Unicode text is represented—in some cases you won't be able to take the output from `json.dumps` and copy and paste it into a Python interpreter session to reconstruct the original dictionary object.

The classical solution to pretty-printing objects in Python is the built-in `pprint` module. Here's an example:

```
>>> import pprint
>>> pprint.pprint(mapping)
{'a': 23, 'b': 42, 'c': 12648430, 'd': set([1, 2, 3])}
```

You can see that `pprint` is able to print data types like sets and also prints the dictionary keys in a reproducible order. Compared to the standard string representation for dictionaries its readability is improved.

However, it doesn't represent nested structures as well visually compared to `json.dumps()`. Depending on the circumstances this can be an advantage or a disadvantage. I occasionally use `json.dumps()` to

print dictionaries if I'm sure they're free of non-primitive data types because of the improved readability and formatting.

Key Takeaways

- The default to-string conversion for dictionary objects can be difficult to read.
- The `pprint` and `json` module are "higher-fidelity" options built into the Python standard library.
- Careful with using `json.dumps()` and non-primitive keys and values.

2.6 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 condition is true, it does nothing and your program just continues to execute. But if the assert condition evaluates to false, it raises an `AssertionError` exception 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 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 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 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% off "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 put in `apply_discount()`.

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

This will speed up future debugging efforts considerably, and it will make your programs more maintainable in the long run. And, my friend, that's the power of assertions, in a nutshell.

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 action 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.

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 with 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[5]:

```
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:

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

You can use `expression2` to pass an *optional error message* that will be displayed with the `AssertionError` in the traceback. This can

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

simplify debugging even further—for example, I've seen code like this:

```
>>> if cond == 'x':
...     do_x()
... elif cond == 'y':
...     do_y()
... else:
...      assert False, (
...          "This should never happen, but it does "
...          "occasionally. We're 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[6] in one of your applications.

Common Pitfalls With Using Asserts in Python

Before you move on, there are two important caveats with using 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 might at least want to skim these two caveats or read their summaries below.

[6] cf. Wikipedia: Heisenbug

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

The biggest caveat with using asserts in Python is that assertions can be globally disabled[7] 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.

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 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 function. What's going to happen if asser-

[7] cf. Python Docs: "Constants – **debug**"

tions are disabled?

There are two serious issues in this three-line function example, 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 not 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 to always be 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 merily wrote a bunch of broken test cases that gave a false sense of security in one of my test suites[8]. Imagine you had this assertion in one of your unit tests:

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

[8] And that's why you should always do a quick smoke test with your unit test cases. Make sure they can actually fail.

Upon first inspection this test case looks completely fine. However, this test case would never catch an incorrect result: it always evaluates to `True`, regardless of the state of the counter variable.

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[9].

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 more maintainable and easier to debug Python programs. It's a great skill to learn that will help bring your Python to the next level and make you a more well-rounded Pythonista.

I know it saved me hours upon hours of debugging at this point.

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.

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

2.7 Cloning Objects for Fun and Profit

Assignment statements in Python do not create copies of objects, they only bind names to an object. For immutable objects that usually doesn't make a difference.

But for working with mutable objects or collections of mutable objects you might be looking for a way to create "real copies" or "clones" of these objects.

Essentially, you'll sometimes want copies that you can modify *without* modifying the original. In this chapter I'm going to give you the rundown of how to copy or "clone" objects in Python and some of the caveats involved.

Let's start by looking at how to copy Python's built-in collections. Python's built-in mutable collections like lists, dicts, and sets can be copied by calling their factory functions on an existing collection:

```
new_list = list(original_list)
new_dict = dict(original_dict)
new_set = set(original_set)
```

However this method won't work for custom objects and on top of that, it only creates *shallow copies*. For compound objects like lists, dicts, and sets there's an important difference between *shallow* and *deep* copying:

A *shallow copy* means constructing a new collection object and then populating it with references to the child objects found in the original. In essence, a shallow copy is only *one level deep*. The copying process does not recurse and therefore won't create copies of the child objects themselves.

A *deep copy* makes the copying process recursive. It means first con-

structing a new collection object and then recursively populating it with copies of the child objects found in the original. Copying an object this way walks the whole object tree to create a fully independent clone of the original object and all of its children.

I know, this was a bit of a mouthful. So let's look at some examples to drive this difference between deep and shallow copies home.

Making Shallow Copies

In the next example we create a new nested list and then *shallowly* copy it with the `list()` factory function:

```
>>> xs = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> ys = list(xs)  # Make a shallow copy
```

This means `ys` will now be a new and independent object with the same contents as `xs`. You can verify this by inspecting both objects:

```
>>> xs
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> ys
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

To confirm `ys` really is independent from the original, let's devise a little experiment. You could try and add a new sublist to the original (`xs`) and then check to make sure this modification didn't affect the copy (`ys`):

```
>>> xs.append(['new sublist'])
>>> xs
[[1, 2, 3], [4, 5, 6], [7, 8, 9], ['new sublist']]
>>> ys
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

As you can see, this had the expected effect. Modifying the copied list at a "superficial" level was no problem at all.

However, because we only created a *shallow* copy of the original list, `ys` still contains references to the original child objects stored in `xs`.

These children were *not* copied. They were merely referenced again in the copied list.

Therefore, when you modify one of the child objects in `xs`, this modification will be reflected in `ys` as well—because *both lists share the same child objects*. The copy is only a shallow, one level deep copy:

```
>>> xs[1][0] = 'X'
>>> xs
[[1, 2, 3], ['X', 5, 6], [7, 8, 9], ['new sublist']]
>>> ys
[[1, 2, 3], ['X', 5, 6], [7, 8, 9]]
```

In the above example we (seemingly) only made a change to `xs`. But it turns out that *both* sublists at index `1` in `xs` and `ys` were modified. Again, this happened because we had only created a *shallow* copy of the original list.

Had we created a *deep* copy of `xs` in the first step, both objects would've been fully independent. This is the practical difference between shallow and deep copies of objects.

Now you know how to create shallow copies of some of the built-in collection classes, and you know the difference between shallow and deep copying. The questions we still want answers for are:

- How can you create deep copies of built-in collections?
- How can you create copies (shallow and deep) of arbitrary objects, including custom classes?

And the answer to these questions lies in the `copy` module in the Python standard library. This module provides a simple interface for creating shallow and deep copies of arbitrary Python objects.

Making Deep Copies

Let's repeat the previous list copying example. But this time we're going to create a *deep* copy using the `deepcopy()` function defined in the `copy` module instead:

```
>>> import copy
>>> xs = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> zs = copy.deepcopy(xs)
```

When you inspect `xs` and its clone `zs` we created with `copy.deepcopy()`, you'll see that they both look identical again—just like in the previous example:

```
>>> xs
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> zs
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

However, if you make a modification to one of the child objects in the original object (`xs`), you'll see that this modification doesn't affect the deep copy (`zs`).

Both objects, the original and the copy, are fully independent this time. `xs` was cloned recursively, including all of its child objects:

```
>>> xs[1][0] = 'X'
>>> xs
[[1, 2, 3], ['X', 5, 6], [7, 8, 9]]
```

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

You might want to take the time to sit down with the Python interpreter and play through these examples. Wrapping your head around copying objects is easier when you get to experience and play with the examples firsthand.

By the way, you can also create shallow copies using a function in the `copy` module. The `copy.copy()` function creates shallow copies of objects.

This is useful if you need to communicate clearly that you're creating a shallow copy somewhere in your code. Using `copy.copy()` lets you indicate this fact. However, for built-in collections it's considered more Pythonic to simply use the list, dict, and set factory functions to create shallow copies.

Copying Arbitrary Objects

The question we still need to answer is how to create copies (shallow and deep) of arbitrary objects, including custom classes. Let's take a look at that now.

Again the `copy` module comes to our rescue—its `copy.copy()` and `copy.deepcopy()` functions can be used to duplicate any object.

The best way to understand how to use these is with a simple example. I'm going to base this on the previous list example. Let's start with defining a simple 2D point class we can experiment with:

```
class Point:
    def __init__(self, x, y):
        self.x = x
```

```
        self.y = y

    def __repr__(self):
        return f'Point({self.x!r}, {self.y!r})'
```

That's pretty straightforward. I added a `__repr__()` implementation so we can easily inspect objects created from this class in the Python interpreter.

Next we'll create a `Point` instance and then (shallowly) copy it using the `copy` module:

```
>>> a = Point(23, 42)
>>> b = copy.copy(a)
```

If we inspect the contents of the original `Point` object and its (shallow) clone we see what we'd expect:

```
>>> a
Point(23, 42)
>>> b
Point(23, 42)
>>> a is b
False
```

Here's something else to keep in mind. Because our point object uses primitive types (ints) for its coordinates there's no difference between a shallow and a deep copy in this case. But I'll expand the example in a second.

Let's move on to a more complex example. I'm going to define another class to represent 2D rectangles. And I'll do it in a way that

allows us to create a more complex object hierarchy—my rectangles will use `Point` objects to represent their coordinates:

```
class Rectangle:
    def __init__(self, topleft, bottomright):
        self.topleft = topleft
        self.bottomright = bottomright

    def __repr__(self):
        return (f'Rectangle({self.topleft!r}, '
                f'{self.bottomright!r})')
```

Again, we're going to create a shallow copy first:

```
rect = Rectangle(Point(0, 1), Point(5, 6))
srect = copy.copy(rect)
```

If you inspect the original rectangle and its copy you'll see a) how nicely the `__repr__()` override is working out, and b) that the shallow copy process worked as expected:

```
>>> rect
Rectangle(Point(0, 1), Point(5, 6))
>>> srect
Rectangle(Point(0, 1), Point(5, 6))
>>> rect is srect
False
```

Remember how the previous list example illustrated the difference between deep and shallow copies? I'm going to use the same technique here. I'll modify an object deeper in the object hierarchy and then you'll see this change reflected in the (shallow) copy as well:

```
>>> rect.topleft.x = 999
>>> rect
Rectangle(Point(999, 1), Point(5, 6))
>>> srect
Rectangle(Point(999, 1), Point(5, 6))
```

I hope this behaved how you expected it to. Next up, I'll create a deep copy of the original rectangle. Then I'll apply another modification and you'll see which objects are affected by that:

```
>>> drect = copy.deepcopy(srect)
>>> drect.topleft.x = 222
>>> drect
Rectangle(Point(222, 1), Point(5, 6))
>>> rect
Rectangle(Point(999, 1), Point(5, 6))
>>> srect
Rectangle(Point(999, 1), Point(5, 6))
```

Voila! This time the deep copy (`drect`) is fully independent of the original (`rect`) and the shallow copy (`srect`).

We've covered a lot of ground here. And there are still some finer points to copying objects.

It pays to go deep (ha!) on this topic and you may want to study up on the `copy` module documentation[10]. For example, objects can control how they're copied by defining the special methods `__copy__()` and `__deepcopy__()` on them. Have fun!

[10] cf. Python docs: "Shallow and deep copy operations"

Key Takeaways

- Making a shallow copy of an object won't clone child objects. Therefore the copy is not fully independent of the original.
- A deep copy of an object will recursively clone child objects. The clone is fully independent of the original but creating a deep copy is slower.
- You can copy arbitrary objects (including custom classes) with the `copy` module.

2.8 A Shocking Truth About String Formatting

Remember the Zen of Python and how there should be "one obvious way to do something in Python"? You might scratch your head when you find out that there's *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!'
```

Hey... now *that* error could really spoil a dev's Monday morning. But we're here to discuss string formatting. 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. This lets you do simple positional formatting very easily. If you've ever worked with a `printf`-style function in C you'll recognize how this works instantly. 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.

There are other format specifiers available that let you control the output format. 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, we can use 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. 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 de-emphasized 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:

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

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

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. 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 to be 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. See here for example:

```
>>> a = 5
>>> b = 10
```

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

```
>>> 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.

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:

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

```
>>> 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
```

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

Here's one more technique for string formatting in Python: 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:

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

```
>>> 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:

```
>>> 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. So when should 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 the 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 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 a hypothetical attacker was able to extract our secret string by accessing the `__globals__` dictionary? 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

- 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*.

2.9 • 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 one another, 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 expected. 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[16]. Things like strings, lists, modules, and functions are all objects. There's nothing particularly special about functions in Python.

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 pointing to it, `bark`. 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`). Because 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!'
```

[16] https://docs.python.org/3/reference/datamodel.html#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[17]:

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

While the function's `__name__` is still "yell" that won't affect how you can access it from your code. This identifier is merely a debugging aid. A *variable pointing to a function* and the *function itself* are two separate concerns.

Functions Can Be Stored in Data Structures

As 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'))
```

[17] 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)

```
<function yell at 0x10ff96510> 'HEY THERE!'
<method 'lower' of 'str' objects> 'hey there'
<method 'capitalize' of 'str' objects> 'Hey there'
```

You can even call a function object stored in the list without assigning it to a variable first. 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 `yell` function to `greet`:

```
>>> greet(yell)
'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 and an iterable and 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 `yell` function to them:

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

`map` has gone through the entire list and applied the `yell` function to each element.

Functions Can Be Nested

Python allows functions to be defined inside other functions. These are often called *nested functions* or *inner functions*. Here's an exam-

ple:

```
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.

And here's the kicker—`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* one of its inner functions—it simply selects the appropriate 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, this is 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 belts on 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.

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 bare-bones 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

Object's aren't functions in Python. 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 round parentheses `()` on it and pass function call arguments to it. Here's an example of 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 callable or not:

```
>>> callable(plus_3)
True
>>> callable(yell)
True
>>> callable(False)
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 which allows you to treat them like functions in many cases.

2.10 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:

```
>>> 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 terse 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`, just written inline. The difference is I didn't bind it to a name like `add` before I used it. I simply stated the expression I wanted to compute and then immediately evaluated it by calling it 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 took me a while to wrap my head around. So don't worry about spending a few minutes in an interpreter session.

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 its 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 a lambda expression can be anonymous, you don't even need to assign it to a name.

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:

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

Like regular nested functions, lambdas also work as *lexical closures*.

What's a lexical closure? 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 `def` can express one's intent more clearly. But to be honest this isn't a common occurrence—at least in the kind of code that I like to write.

But Maybe You Shouldn't...

Now 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 wrote my fair share of code using lambdas that looked "cool" but was 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 this 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, it technically works and it's a nice enough "trick." But it's also going to confuse the next gal or guy having 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 feel similarly 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 a lambda expression, consider defining a real function with a proper name instead.

Saving a few keystrokes won't matter in the long run. 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/generator expression offer more clarity?*

2.11 Peeking Behind the Bytecode Curtain

When the CPython interpreter executes your program it first translates it into a sequence of bytecode instructions. Bytecode is an intermediate language for the Python virtual machine used as a performance optimization.

Instead of directly executing the human-readable source code, compact numeric codes, constants, and references are used that represent the result of compiler parsing and semantic analysis.

This saves time and memory for repeated executions of programs or parts of programs. For example, the bytecode resulting from this compilation step is cached in `.pyc` and `.pyo` files so that executing the same Python file is faster the second time.

All of this is completely transparent to the programmer. You don't need to be aware that this intermediate translation step happens, or how the Python virtual machine deals with the bytecode. In fact, the bytecode format is deemed an implementation detail and not guaranteed to remain stable or compatible between Python versions.

And yet, I find it very enlightening to see *how the sausage is made* and to peek behind the scenes of the abstractions provided by the CPython interpreter. Understanding at least some of the inner workings can help you write more performant code (when that's important). And it's also a lot of fun.

Let's take this simple `greet()` function as a lab sample we can play with and use to understand Python's bytecode:

```
>>> def greet(name):
...     return 'Hello, ' + name + '!'
```

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

Remember how I said that CPython first translates our source code into an intermediate language before it "runs" it? Well, if that's true we should be able to see the results of this compilation step. And we can.

Each function has a `__code__` attribute (in Python 3) that we can use to get at the virtual machine instructions, constants, and variables used by our greet function:

```
>>> greet.__code__.co_code
b'dx01|x00x17x00dx02x17x00Sx00'
>>> greet.__code__.co_consts
(None, 'Hello, ', '!')
>>> greet.__code__.co_varnames
('name',)
```

You can see `co_consts` contains parts of the greeting string our function assembles. Constants and code are kept separate to save memory space. Constants are, well, constant—meaning they can never change and be used interchangeably in multiple places.

So instead of repeating the actual constant values in the `co_code` instruction stream, Python constants separately in a lookup table. The instruction stream can then refer to constants with an index into the lookup table. The same is true for variables (`co_varnames`).

I hope the general concept is starting to become clearer. But looking at the `co_code` instruction stream still makes me feel a little queasy. This intermediate language is clearly meant to be easy to work with for the Python virtual machine, not humans (that's what the text-based source code is for).

The developers working on CPython realized that too. So they gave us another tool called a *disassembler* to make inspecting the bytecode easier.

Python's bytecode disassembler lives in the `dis` module that's part of the standard library. So we can just import it and call `dis.dis()` on our greet function to get a slightly easier to read representation of its bytecode:

```
>>> import dis
>>> dis.dis(greet)
  2           0 LOAD_CONST               1 ('Hello, ')
              2 LOAD_FAST                0 (name)
              4 BINARY_ADD
              6 LOAD_CONST               2 ('!')
              8 BINARY_ADD
             10 RETURN_VALUE
```

The main thing disassembling did was splitting up the instruction stream and giving each *opcode* in it a human-readable name like `LOAD_CONST`.

You can also see how constant and variable references are now interleaved with the bytecode and printed in full to spare us the mental gymnastics of a `co_const` or `co_varnames` table lookup. Neat!

Looking at the human-readable opcodes we can begin to understand how CPython represents and executes the `'Hello, ' + name + '!'` expression in the original `greet()` function.

It first retrieves the first constant `('Hello, ')` and puts it on the *stack*. It then loads the contents of the `name` variable and also puts them on the *stack*.

The *stack* is the data structure used as internal working storage for

the virtual machine. There are different classes of virtual machines and one of them is called a *stack machine*. CPython's virtual machine is an implementation of such a stack machine. If the whole thing is named after the stack you can imagine what a central role this data structure plays.

By the way—I'm only touching the surface here and if you're interested in this topic I'll give some book recommendations at the end. Reading up on virtual machine theory is enlightening (and a ton of fun).

What's interesting about a *stack* as an abstract data structure is that at the bare minimum it only supports two operations: *push* and *pop*. *Push* adds a value to the top of the stack and *pop* removes and returns the topmost value. Unlike an array there's no way to access elements "below" the top level.

I find it fascinating that such a simple data structure has so many uses. But I'm getting carried away again...

Let's assume the stack starts out empty. After the first two opcodes have been executed, this is what the contents of the VM stack looks like (`0` is the topmost element):

```
0: 'Dan' (contents of "name")
1: 'Hello, '
```

The `BINARY_ADD` instruction pops the two string values off the stack, concatenates them, and then pushes the result on the stack again:

```
0: 'Hello, Dan'
```

Then there's another `LOAD_CONST` to get the exclamation mark string on the stack:

```
0: '!'
1: 'Hello, Dan'
```

And the final `BINARY_ADD` again combines the two to generate the final greeting string:

```
0: 'Hello, Dan!'
```

The last bytecode instruction is `RETURN_VALUE` which tells the virtual machine that what's currently on top of the stack is the return value for this function so it can be passed on to the caller.

And voila, we just traced back how our `greet()` function gets executed internally by the CPython virtual machine. Isn't that cool?

There's much more to say about virtual machines and this isn't the book for it. But if this got you interested I highly recommend that you do some more reading on this fascinating subject.

It can be a lot of fun to define your own bytecode language and build little virtual machine experiments for them. A book that I'd recommend is "Compiler Design: Virtual Machines" by Wilhelm and Seidl.

Key Takeaways

- CPython executes programs by first translating them into an intermediate bytecode and then running the bytecode on a stack-based virtual machine.
- You can use the built-in `dis` module to peek behind the scenes and inspect the bytecode.
- Study up on virtual machines, it's worth it.

2.12 String Conversion (Every Class Needs a __repr__)

When you define a custom class in Python and then try to print one of its instances to the console (or inspect it in an interpreter session) you get a relatively unsatisfying result. The default "to string" conversion behavior is basic and lacks detail:

```
class Car:
    def __init__(self, color, mileage):
        self.color = color
        self.mileage = mileage

>>> my_car = Car('red', 37281)
>>> print(my_car)
<__console__.Car object at 0x109b73da0>
>>> my_car
<__console__.Car object at 0x109b73da0>
```

By default all you get is a string containing the class name and the `id` of the object instance (which is the object's memory address in CPython). That's better than *nothing*, but it's also not very useful.

You might find yourself trying to work around this by printing attributes of the class directly, or even adding a custom `to_string()` method:

```
>>> print(my_car.color, my_car.mileage)
red 37281
```

The general idea here is the right one—but it ignores the conventions and built-in mechanisms Python uses to handle how objects are represented as strings.

Instead of building your own to-string conversion machinery you'll be better off adding the `__str__` and `__repr__` "dunder" methods to your class. They are the Pythonic way to control how objects are converted to strings in different situations[18].

Let's take a look at how these methods work in practice. To get started, we're going to add a `__str__` method to the `Car` class we defined earlier:

```
class Car:
    def __init__(self, color, mileage):
        self.color = color
        self.mileage = mileage

    def __str__(self):
        return f'a {self.color} car'
```

When you try printing or inspecting a `Car` instance now you'll get a different, slightly improved result:

```
>>> my_car = Car('red', 37281)
>>> print(my_car)
'a red car'
>>> my_car
<__console__.Car object at 0x109ca24e0>
```

Inspecting the car object in the console still gives the previous result containing the object's `id`. But *printing* the object resulted in the string returned by the `__str__` method we added.

`__str__` is one of Python's "dunder" (double-underscore) methods and gets called when you try to convert an object into a string through the various means available:

[18] cf. Python Docs: "The Python Data Model"

```
>>> print(my_car)
a red car
>>> str(my_car)
'a red car'
>>> '{}'.format(my_car)
'a red car'
```

With a proper `__str__` implementation you won't have to worry about printing object attributes directly or writing a separate `to_string()` function. It's the Pythonic way to control string conversion.

By the way, some people refer to Python's "dunder" methods as "magic methods." But these methods are not supposed to be *magical* in any way. The fact that these methods start and end in double underscores is simply a naming convention to flag them as core Python features. It also helps avoid naming collisions with your own methods and attributes. The object constructor `__init__` follows the same convention and there's nothing magical or arcane about it.

Don't be afraid to use Python's dunder methods—they're meant to help you.

`__str__` vs `__repr__`

Now, our string conversion story doesn't end there. Did you see how inspecting `my_car` in an interpreter session still gave that odd `<Car object at 0x109ca24e0>` result?

This happened because there are actually *two* dunder methods that control how objects are converted to strings in Python 3. The first one is `__str__` and you just learned about it. The second one is `__repr__`, and it works similarly to `__str__` but is used in different situations.

(Python 2.x also has an `__unicode__` method that I'll touch on a little later.)

Here's a simple experiment you can use to get a feel for when `__str__` or `__repr__` is used. Let's redefine our car class so it contains both *to-string* dunder methods with outputs that are easy to distinguish:

```
class Car:
    def __init__(self, color, mileage):
        self.color = color
        self.mileage = mileage

    def __repr__(self):
        return '__repr__ for Car'

    def __str__(self):
        return '__str__ for Car'
```

Now when you play through the previous examples you can see which method controls the string conversion result in each case:

```
>>> my_car = Car('red', 37281)
>>> print(my_car)
__str__ for Car
>>> '{}'.format(my_car)
'__str__ for Car'
>>> my_car
__repr__ for Car
```

This experiment confirms that inspecting an object in a Python console session simply prints the result of the object's `__repr__`.

Interestingly, containers like lists and dicts always use the result of `__repr__` to represent the objects they contain. Even if you call `str` on the container itself:

```
str([my_car])
'[__repr__ for Car]'
```

To manually choose between both string conversion methods, for example to express your code's intent more clearly, it's best to use the built-in `str()` and `repr()` functions. Using them is preferable over calling the object's `__str__` or `__repr__` directly as it looks nicer and gives the same result:

```
>>> str(my_car)
'__str__ for Car'
>>> repr(my_car)
'__repr__ for Car'
```

Even with this investigation completed you might be wondering what the "real-world" difference between `__str__` and `__repr__` is. They both seem to serve the same purpose and it might be unclear when to use each.

With questions like that it's usually a good idea to emulate what the Python standard library does. Time to devise another experiment. We'll create a `datetime.date` object and find out how it uses `__repr__` and `__str__` to control string conversion:

```
>>> import datetime
>>> today = datetime.date.today()
```

The result of the date object's `__str__` function should primarily be *readable*. It's meant to return a concise textual representation for

human consumption. Something you'd feel comfortable displaying to a user. Therefore we get something that looks like an ISO date format when we call `str()` on the date object:

```
>>> str(today)
'2017-02-02'
```

With `__repr__` the idea is that its result should be, above all, *unambiguous*. The resulting string is intended more as a debugging aid for developers. And for that it needs to be as explicit as possible about what this object is. That's why you'll get a more elaborate result calling `repr()` on the object. It even includes the full module and class name:

```
>>> repr(today)
'datetime.date(2017, 2, 2)'
```

We could copy and paste the string returned by `__repr__` and execute it as valid Python to recreate the original date object. This is a neat approach and a good goal to keep in mind while writing your own reprs.

On the other hand, I found that it is quite difficult to put into practice. Usually it won't be worth the trouble and extra work. My rule of thumb is to make my `__repr__` strings unambiguous and helpful for developers, but I don't expect them to be able to restore an object's complete state.

Why Every Class Needs a `__repr__`

If you don't add a `__str__` method Python falls back on the result of `__repr__` when looking for `__str__`. Therefore, I recommend that you always add at least a `__repr__` method to your classes. This will

guarantee a useful string conversion result in almost all cases with relatively little implementation work.

Here's how to write that minimal implementation quickly and efficiently. For our `Car` class we might start with the following `__repr__`:

```
def __repr__(self):
    return f'Car({self.color!r}, {self.mileage!r})'
```

Please note that I'm using the `!r` conversion flag to make sure the resulting string uses `repr(self.color)` and `repr(self.mileage)` instead of `str(self.color)` and `str(self.mileage)`.

This works nicely, but a downside is that we've repeated the class name inside the format string. A trick you can use here to avoid this repetition is to use the object's `__class__.__name__` attribute, which will always reflect the class's name as a string.

The benefit is you won't have to modify the `__repr__` implementation when the class name changes. This makes it easy to adhere to the *Don't Repeat Yourself* (DRY) principle:

```
def __repr__(self):
   return (f'{self.__class__.__name__}('
           f'{self.color!r}, {self.mileage!r})')
```

The downside of this implementation is that the format string is quite long and unwieldy. But with careful formatting you can keep the code nice and PEP 8 compliant.

With the above `__repr__` implementation we get a useful result when we inspect the object or call `repr()` on it directly:

```
>>> repr(my_car)
'Car(red, 37281)'
```

Printing the object or calling `str()` on it returns the same string because the default `__str__` implementation simply calls `__repr__`:

```
>>> print(my_car)
'Car(red, 37281)'
>>> str(my_car)
'Car(red, 37281)'
```

I believe this approach provides the most value with a modest amount of implementation work. It's also a fairly cookie-cutter approach that can be applied with out much deliberation. For this reason I always try to add a basic `__repr__` implementation to my classes.

Here's a complete example for Python 3, including an optional `__str__` implementation:

```
class Car:
    def __init__(self, color, mileage):
        self.color = color
        self.mileage = mileage

    def __repr__(self):
       return (f'{self.__class__.__name__}('
               f'{self.color!r}, {self.mileage!r})')

    def __str__(self):
        return f'a {self.color} car'
```

Python 2.x Differences: `__unicode__`

In Python 3 there's one data type to represent text across the board: `str`. It holds unicode characters and can represent most of the world's writing systems.

Python 2.x uses a different data model for strings[19]. There are two types to represent text: `str` which is limited to the ASCII character set, and `unicode` which is equivalent to Python 3's `str`.

Due to this difference there's yet another dunder method in the mix for controlling string conversion in Python 2.x: `__unicode__`. In Python 2.x `__str__` returns *bytes*, whereas `__unicode__` returns *characters*.

For most intents and purposes`__unicode__` is the newer and preferred method to control string conversion. There's also a built-in `unicode()` function to go along with it. It calls the respective dunder method, similarly to how `str()` and `repr()` work.

So far so good. Now it gets a little more quirky when you look at the rules for when `__str__` and `__unicode__` are called in Python 2.x:

The `print` statement and `str()` call `__str__`. The `unicode()` built-in calls `__unicode__` if it exists, and otherwise falls back to `__str__` and decodes the result with the system text encoding.

These special cases complicate the text conversion rules somewhat compared to Python 3. But there is a way to simplify things again for practical use. Unicode is the preferred and future proof way of handling text in your Python programs.

So generally, what I would recommend you do in Python 2.x is to put all of your string formatting code in `__unicode__` and then create a stub `__str__` implementation that returns the unicode representa-

[19] cf. Python 2 Docs: "Data Model"

tion encoded as UTF-8:

```
def __str__(self):
    return unicode(self).encode('utf-8')
```

The __str__ stub will be the same for most classes you write, so you can just copy and paste it around as needed (or put it into a base class where it makes sense). All of your string conversion code that is meant for non-developer use then lives in __unicode__.

Here's a complete example for Python 2.x:

```
class Car(object):
    def __init__(self, color, mileage):
        self.color = color
        self.mileage = mileage

    def __repr__(self):
       return '{}({!r}, {!r})'.format(
           self.__class__.__name__,
           self.color, self.mileage)

    def __unicode__(self):
        return u'a {self.color} car'.format(
            self=self)

    def __str__(self):
        return unicode(self).encode('utf-8')
```

Things to Remember

- You can control to-string conversion in your own classes using the __str__ and __repr__ "dunder" methods.

- The result of `__str__` should be readable. The result of `__repr__` should be unambiguous.
- Always add a `__repr__` to your classes. The default implementation for `__str__` just calls `__repr__`.
- Use `__unicode__` instead of `__str__` in Python 2.

2.13 Fun With *args and **kwargs

I once pair-programmed with a smart Pythonista who would exclaim "argh!" and "kwargh!" every time he typed out a function definition with optional or keyword parameters. We got along great otherwise. But I guess that's what programming in academia does to people eventually.

Now, while easily mocked, `*args` and `**kwargs` parameters are nevertheless a highly useful feature in Python. And understanding their power will make you a more effective developer.

So what are they used for? They allow a function to accept optional arguments so you can create flexible APIs:

```
def foo(required, *args, **kwargs):
    print(required)
    if args:
        print(args)
    if kwargs:
        print(kwargs)
```

This function will require at least one argument (`required`) but it can accept extra positional and keyword arguments as well.

If we call the function with additional arguments, `args` will collect extra positional arguments as a tuple because the parameter name has a * prefix.

Likewise, `kwargs` will collect extra keyword arguments as a dictionary because the parameter name has a ** prefix.

Both `args` and `kwargs` can be empty if no extra arguments are passed to the function.

As we call the function with various combinations of arguments you'll see how Python collects them inside the `args` and `kwargs` parameters according to their type:

```
>>> foo()
TypeError:
"foo() missing 1 required positional arg: 'required'"

>>> foo('hello')
hello

>>> foo('hello', 1, 2, 3)
hello
(1, 2, 3)

>>> foo('hello', 1, 2, 3, key1='value', key2=999)
hello
(1, 2, 3)
{'key1': 'value', 'key2': 999}
```

I want to make it clear that calling the parameters `args` and `kwargs` is simply a naming convention. The previous example would work just as well if you called them `*parms` and `**argv`. The actual syntax is just the asterisk `*` or double asterisk `**`, respectively.

I recommend that you stick with the accepted naming convention, however, to avoid confusion. (And to get a chance to yell "argh!" and "kwargh!" every once in a while.)

Forwarding Optional or Keyword Arguments

It's possible to pass optional or keyword parameters from one function to another. You can do so using the argument unpacking oper-

ators * and ** when calling the function you want to forward arguments to[20].

This also gives you an opportunity to modify the arguments before you pass them along. Here's an example:

```
def foo(x, *args, **kwargs):
    kwargs['name'] = 'Alice'
    new_args = args + ('extra', )
    bar(x, *new_args, **kwargs)
```

This technique can be useful for subclassing and writing wrapper functions.

For example, you can use it to extend the behavior of a parent class without having to replicate the full signature of its constructor in the child class. This can be quite convenient if you're working with an API that might change outside of your control:

```
class Car:
    def __init__(self, color, mileage):
        self.color = color
        self.mileage = mileage

class AlwaysBlueCar(Car):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.color = 'blue'

>>> AlwaysBlueCar('green', 48392).color
'blue'
```

[20] cf. "Function Argument Unpacking"

The `AlwaysBlueCar` constructor simply passes on all arguments to its parent class and then overrides an internal attribute. This means if the parent class constructor changes, there's a good chance that `AlwaysBlueCar` would still function as intended.

The downside here is that the `AlwaysBlueCar` constructor now has a rather unhelpful signature—we don't know what arguments it expects without looking up the parent class.

Typically you wouldn't use this technique with your own class hierarchies. The more likely scenario would be that you'll want to modify or override behavior in some external class which you don't control.

But this is always dangerous territory, so best be careful (or you might soon have yet another reason to scream "argh!").

One more scenario where this technique is potentially helpful is writing wrapper functions such as decorators. There you typically also want to accept arbitrary arguments to be passed through to the wrapped function.

And if we can do it without having to copy and paste the original function's signature, that might be more maintainable:

```
def trace(f):
    @functools.wraps(f)
    def decorated_function(*args, **kwargs):
        print(f, args, kwargs)
        result = f(*args, **kwargs)
        print(result)
    return decorated_function

@trace
def greet(greeting, name):
   return '{}, {}!'.format(greeting, name)
```

```
>>> greet('Hello', 'Bob')
<function greet at 0x1031c9158> ('Hello', 'Bob') {}
Hello, Bob!
```

With techniques like this one it's sometimes difficult to balance making your code explicit enough and adhering to the *Don't Repeat Yourself (DRY)* principle. This will always be a tough choice to make. So if you can get a second opinion from a colleague I'd encourage you to ask for one.

Key Takeaways

- `*args` and `**kwargs` let you write functions with a variable number of arguments in Python.
- `*args` collects extra positional arguments as a tuple. `**kwargs` collects the extra keyword arguments as a dictionary.
- The actual syntax is `*` and `**`. Calling them `args` and `kwargs` is just a convention (one you should stick to).

Chapter 3

Patterns for Cleaner Python

3.1 "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. I think Tim's words made me a better coder. Maybe 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!

3.2 Defining Your Own Exception Classes

Including your own exception classes can make your errors more readable within your code and also communicate to the users of your functions or modules your intent and what the errors are. It also allows to add more context. All of this will improve your Python code to make it easier to understand, easier to debug, and more maintainable.

When I started using Python I was hesitant to add custom exception classes. But it's really easy when you break it down. I'll walk you through the main points to remember with a simple code example.

Let's say we want to validate an input string representing a person's name in our application. A simple toy example might look like this:

```
def validate(name):
    if len(name) < 10:
        raise ValueError
```

If the validation fails it throws a `ValueError`. That feels kind of Pythonic already. So far, so good.

However, there's one downside to this function. Imagine one of our teammates calls it as part of a library and doesn't know much about its internals. When a name fails to validate it'll look like this in the debug stack trace:

```
>>> validate('joe')
Traceback (most recent call last):
  File "<input>", line 1, in <module>
    validate('joe')
  File "<input>", line 3, in validate
```

```
    raise ValueError
ValueError
```

This stack trace isn't really all that helpful. Sure, we know that something went wrong and that the problem has to do with an "incorrect value" of sorts.

But to be able to fix the problem our teammate almost certainly has to look up the implementation of `validate()`. But reading code costs time. And it adds up quickly.

Luckily we can do better. Let's introduce a custom exception type for when a name fails validation. We'll base our new exception class on Python's built-in `ValueError`, but make it speak for itself by giving it a more explicit name:

```
class NameTooShortError(ValueError):
    pass

def validate(name):
    if len(name) < 10:
        raise NameTooShortError(name)
```

Generally you'll want to either derive your custom exceptions from `Exception` or the other built-in Python exceptions like `ValueError` or `TypeError` that feel appropriate.

Also, see how we're passing `name` to the constructor of our custom exception class when we instantiate it inside `validate`? The new implementation results in a much nicer stack trace for our colleague:

```
>>> validate('jane')
Traceback (most recent call last):
```

```
  File "<input>", line 1, in <module>
    validate('jane')
  File "<input>", line 3, in validate
    raise NameTooShortError(name)
NameTooShortError: jane
```

Now, imagine *you* are the teammate we were talking about. Even if you're working on a code base by yourself, custom exception classes will make it easier to understand what's going on when things go wrong (and eventually they always do). A few weeks or months down the road you'll have a much easier time maintaining your code if it's well structured.

Here's an extra tip for you to consider if you're releasing a Python library or even if you're just creating a re-usable module within your company. It's good practice to create a custom exception base class and then derive all of your other exceptions from it.

This allows users of your package to write *try...except* statements that can handle all of the errors from this package without having to catch them manually:

```
try:
    validate(name)
except name_validator.BaseValidationError as err:
    handle_validation_error(err)
```

People can still catch more specific exceptions that way. But if they don't want to, at least they don't have to resort to snapping up all exceptions with a catch-all `except` statement. This is generally considered an anti-pattern—it can silently swallow and hide unrelated errors and make your programs much harder to debug.

Creating a base class for all exceptions in a module is fairly straightforward. You first declare a base class that all of our concrete errors will inherit from:

```
class BaseValidationError(ValueError):
    pass
```

Now, all of our "real" error classes are derived from the base error class. This gives a nice and clean exception hierarchy with little extra effort:

```
class NameTooShortError(BaseValidationError):
    pass

class NameTooLongError(BaseValidationError):
    pass

class NameTooCuteError(BaseValidationError):
    pass
```

Of course you can take this idea further and logically group your exceptions into a finer grained hierarchy. But be careful, it's easy to introduce unnecessary complexity by going overboard with this.

In conclusion, defining custom exception classes makes it easier for your users to adopt an *it's easier to ask for forgiveness than permission* (EAFP) coding style that's considered more Pythonic.

Key Takeaways

- Defining your own exception types will state your code's intent more clearly and make it easier to debug.

- Derive your custom exceptions from Python's built-in `Exception` class or from more specific exception classes like `ValueError` or `KeyError`.
- You can use inheritance to define logically grouped exception taxonomies.

3.3 Dictionary Default Values

Python's dictionaries have a `get()` method to look up a key while providing a fallback value. This is handy in many situations. Let me give you a simple example. Imagine we have the following data structure mapping *user IDs* to *user names*:

```
name_for_userid = {
    382: "Alice",
    950: "Bob",
    590: "Dilbert",
}
```

Now we'd like to use this data structure to write a function `greeting()` which will return a greeting for a user given their user ID. Our first implementation might look something like this:

```
def greeting(userid):
    return "Hi %s!" % name_for_userid[userid]
```

This first implementation technically works—but only if the user ID is a valid key in `name_for_userid`. If we pass an *invalid* user ID to our greeting function it throws an exception:

```
>>> greeting(382)
"Hi Alice!"

>>> greeting(33333333)
KeyError: 33333333
```

A `KeyError` exception isn't really the result we'd like to see. For our purposes it would be much better if the function returned a generic greeting if the user ID can't be found.

Let's implement this idea. Our first approach might be to simply do a *key in dict* membership check and to return a default greeting if the user ID is unknown:

```
def greeting(userid):
    if userid in name_for_userid:
        return "Hi %s!" % name_for_userid[userid]
    else:
        return "Hi there!"
```

Let's see how this implementation of `greeting()` fares with our previous test cases:

```
>>> greeting(382)
"Hi Alice!"

>>> greeting(33333333)
"Hi there!"
```

Much better. We now get the generic greeting for unknown users and we keep the personal greeting for valid user IDs found.

But there's still room for improvement. While this implementation gives us the expected results and seems small and clean enough, it could still be improved:

- It's *inefficient* because it queries the dictionary twice.
- It's *verbose* as part of the greeting string are repeated, for example.
- It's not *Pythonic*—the official Python documentation recommends an *easier to ask for forgiveness than permission* (EAFP) coding style:

> "This common Python coding style assumes the existence of valid keys or attributes and catches exceptions if the assumption proves false." [1]

Therefore a better implementation that follows the *EAFP* principle might use a *try...except* block to catch the `KeyError` instead of doing an explicit membership test:

```
def greeting(userid):
    try:
        return "Hi %s!" % name_for_userid[userid]
    except KeyError:
        return "Hi there"
```

Again, this implementation would be correct—but we can come up with a *cleaner* solution still. Python's dictionaries have a `get()` method on them which supports a default argument that can be used as a fallback value[2]:

```
def greeting(userid):
    return "Hi %s!" % name_for_userid.get(
        userid, "there")
```

When `get()` is called it checks if the given key exists in the dict. If it does, the value for the key is returned. If it does *not* exist then the value of the *default* argument is returned instead. As you can see, this final implementation of `greeting` works as intended:

---

[1] cf. Python Glossary: "EAFP"

[2] cf. Python Docs: `dict.get()` method

```
>>> greeting(950)
"Hi Bob!"

>>> greeting(333333)
"Hi there!"
```

Our final implementation of `greeting()` is concise, clean, and only uses features from the Python standard library. Therefore I believe it is the best solution for this particular situation.

Key Takeaways

- Avoid explicit *key in dict* checks when testing for membership.
- EAFP-style exception handling or using a built-in `get()` method are preferable.

3.4 Abstract Base Classes Keep Inheritance in Check

Abstract Base Classes (ABCs) enforce that derived classes implement particular methods from the base class. In this chapter you'll learn about the benefits of abstract base classes and how to define them with Python's built-in "abc" module.

So what are Abstract Base Classes good for? A while ago I had a discussion about which pattern to use for implementing a maintainable class hierarchy in Python. More specifically, the goal was to define a simple class hierarchy for a service backend in the most programmer-friendly and maintainable way.

There was a `BaseService` that defines a common interface and several concrete implementations that do different things but all provide the same interface (`MockService`, `RealService`, and so on). To make this relationship explicit the concrete implementations all subclass `BaseService`.

To be as maintainable and programmer-friendly as possible the idea was to make sure that:

- instantiating the base class is impossible; and
- forgetting to implement interface methods in one of the subclasses raises an error as early as possible.

This raises the question why you'd want to use Python's `abc` module. The above design problem is pretty common in more complex systems. To enforce that a derived class implements a number of methods from the base class we can use something like this Python idiom:

```
class Base:
    def foo(self):
        raise NotImplementedError()

    def bar(self):
        raise NotImplementedError()

class Concrete(Base):
    def foo(self):
        return 'foo() called'

    # Oh no, we forgot to override bar()...
    # def bar(self):
    #     return "bar() called"
```

So, what do we get from this first implementation? Calling methods on an instance of `Base` correctly raises `NotImplementedError` exceptions:

```
>>> b = Base()
>>> b.foo()
NotImplementedError
```

Furthermore, instantiating and using `Concrete` works as expected—and, if we call an unimplemented method on it like `bar()` this also raises an exception:

```
>>> c = Concrete()
>>> c.foo()
'foo() called'
>>> c.bar()
NotImplementedError
```

This first implementation is decent but it isn't perfect yet. The downsides here are that we can still:

- instantiate `Base` just fine without getting an error; and
- provide incomplete subclasses—instantiating `Concrete` will not raise an error until we call the missing method `bar()`.

With Python's `abc` module was added in Python 2.6[3], we can do quite a bit better and solve these outstanding issues. Here's an updated implementation using an Abstract Base Class defined with the `abc` module:

```
from abc import ABCMeta, abstractmethod

class Base(metaclass=ABCMeta):
    @abstractmethod
    def foo(self):
        pass

    @abstractmethod
    def bar(self):
        pass
```

```
class Concrete(Base):
    def foo(self):
        pass

    # We forget to declare bar() again...
```

This still behaves as expected and creates the correct class hierarchy:

[3] cf. Python Docs: abc module

```
assert issubclass(Concrete, Base)
```

Yet, we do get another very useful benefit here. Subclasses of `Base` raise a `TypeError` *at instantiation time* whenever we forget to implement any abstract methods. The raised exception tells us which method or methods we're missing:

```
>>> c = Concrete()
TypeError:
"Can't instantiate abstract class Concrete
with abstract methods bar"
```

Without `abc` we'd only get a `NotImplementedError` if a missing method is actually called. Being notified about missing methods at instantiation time is a great advantage. It makes it more difficult to write invalid subclasses. This might not be a big deal writing new code, but a few weeks or months down the line I promise it'll be helpful.

This pattern is not a full replacement for static-typing, of course. But in some situation it can make the resulting code more robust and more readily maintainable. Also, using `abc` states the code's intent more clearly. I'd encourage you to read the `abc` module documentation and to start applying this pattern where it makes sense.

Key Takeaways

- Abstract Base Classes (ABCs) enforce that derived classes implement particular methods from the base class at instantiation time.
- Using ABCs can help avoid bugs and make class hierarchies easier to maintain.

3.5 Sorting Dictionaries for Fun and Profit

Python dictionaries don't have an inherent order. You can iterate through them but there's nothing to guarantee that the iteration will return the dictionary's elements in any particular order (although this is changing with Python 3.6).

However, it's frequently useful to get a *sorted representation* of a dictionary to put the dictionary's items into an arbitrary order based on their key, value, or some other derived property. Suppose you have a dictionary `xs` with the following key/value pairs:

```
>>> xs = {'a': 4, 'c': 2, 'b': 3, 'd': 1}
```

To get a sorted list of the key/value pairs in this dictionary, you could use the dictionary's `items()` method and then sort the resulting sequence in a second pass:

```
>>> sorted(xs.items())
[('a', 4), ('b', 3), ('c', 2), ('d', 1)]
```

As you can see the key/value tuples are ordered using Python's standard lexicographical ordering for comparing sequences. First, the first two items are compared, and if they differ this determines the outcome of the comparison. If they're equal, the next two items are compared, and so on.

In some cases a lexicographical ordering might be exactly what you want. In other cases it might be desirable to sort a dictionary by value. Luckily, there's a way you can get complete control over how items are ordered. You can control the ordering by passing a *key func* to `sorted()` that will change how dictionary items are compared.

A *key func* is simply a normal Python function to be called on each element prior to making comparisons. The key func gets a dictionary item as its input and returns the desired "key" for the sort order comparisons.

Unfortunately, the word "key" is used in two contexts simultaneously here—the key func doesn't deal with dictionary keys, it merely maps each input item to an arbitrary *comparison key*.

But let's look at an example. Key funcs will be much easier to understand once you see some real code, trust me.

Let's say you wanted to get a sorted representation of a dictionary based on its *values*. To get this result you could use the following key func which returns the value of each key/value pair by looking up the second element in the tuple:

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

See how the resulting list of key/value pairs is now sorted by the values stored in the original dictionary? It's worth spending some time wrapping your head around how key funcs work. It's a powerful concept that you can apply in all kinds of Python contexts.

In fact, the concept is so common that Python's standard library includes the `operator` module. This module implements some of the frequently key funcs as plug-and-play building blocks, like `operator.itemgetter`, `operator.getitem`, and `operator.attrgetter`.

Here's an example of how you might replace the lambda-based index lookup in the first example with `operator.itemgetter`:

```
>>> import operator
>>> sorted(xs.items(), key=operator.itemgetter(1))
[('d', 1), ('c', 2), ('b', 3), ('a', 4)]
```

Using the `operator` module might communicate your code's intent more clearly in some cases. On the other hand using a simple lambda expression might be just as readable and more explicit.

Another benefit of using lambdas is that you can control the sort order in much closer detail. For example, you could sort a dictionary based on the *absolute numeric value of its values*:

```
>>> sorted(xs.items(), key=lambda x: abs(x[1]))
```

If you need to reverse the sort order so that larger values go first, you can use the `reverse=True` keyword argument when calling `sorted()`:

```
>>> sorted(xs.items(),
           key=lambda x: x[1],
           reverse=True)
[('a', 4), ('b', 3), ('c', 2), ('d', 1)]
```

Like I said earlier—it's totally worth spending some time getting a good grip on how key funcs work in Python. They provide a ton of flexibility and can often save you from writing lots of code to transform one data structure into another.

Key Takeaways

- When creating sorted "views" of dictionaries and other collections you can influence the sort order with a *key func*.

- *Key funcs* are an important concept in Python. The most-frequently used ones were even added to the `operator` module in the standard library.
- Functions are first-class citizens in Python. This is a powerful feature.

3.6 Emulating Switch/Case Statements With Dicts

Python doesn't have switch/case statements so it's sometimes necessary to write long `if…elif…else` chains as a workaround. In this chapter you'll discover a cool trick you can use to emulate switch/case statements in Python with dictionaries and first-class functions. Sounds exciting? Good, here we go!

Imagine we had the following `if`-chain in our program:

```
>>> if cond == 'cond_a':
...     handle_a()
... elif cond == 'cond_b':
...     handle_b()
... else:
...     handle_default()
```

Of course this isn't too horrible yet. But just imagine we had about 10 or more `elif` branches in this statement and things would look a little different.

If a long `if…elif…else` gets too cumbersome to work with, you can use dictionaries to emulate the behavior of a switch/case statement.

The idea here is to leverage the fact that Python has *first-class functions*. Which means they can be passed as arguments to other functions, returned as values from other functions, and assigned to variables and stored in data structures.

For example, we can define a function and then store it in a list for later access:

```
>>> def myfunc(a, b):
...     return a + b
...
>>> funcs = [myfunc]
>>> funcs[0]
<function myfunc at 0x107012230>
```

The syntax for calling this function works as you'd intuitively expect—we simply use an index into the list and then use the `()` call syntax for calling the function and passing arguments to it:

```
>>> funcs[0](2, 3)
5
```

Now how are we going to use first-class functions to cut our chained `if` statement back to size? The core idea here is to define a dictionary, mapping lookup keys for the input conditions (like `cond_a` in the first example) to functions that will carry out the intended operations (like `handle_a`):

```
>>> func_dict = {
...     'cond_a': handle_a,
...     'cond_b': handle_b
... }
```

Then we can do a dictionary key look up to get the handler function, and call it like we did earlier in the first `if`-statement example:

```
>>> cond = 'cond_a'
>>> func_dict[cond]()
```

This implementation already sort-of works, at least as long as `cond` can be found in the dictionary—if it can't be found we get a `KeyError` exception.

So let's look for a way to support a *default* case that would match the original `else` branch. Luckily all Python dicts have a `get()` method on them that returns the value for a given key, or a default value if the key can't be found. This is exactly what we need here:

```
>>> func_dict.get(cond, handle_default)()
```

This might look odd syntactically at first, but when you break it down it works exactly like the earlier example. Again, we're using Python's first-class functions to pass `handle_default` to the `get()`-lookup as a fallback value. That way, if the condition can't be found in the dictionary we avoid raising a `KeyError` and call the default handler function instead.

Let's take a look at a more complete example for using dictionary lookups and first-class functions to replace `if`-chains. After reading through the following example you'll be able to see the pattern needed to transform certain kinds of `if`-statements to a dictionary-based dispatch.

We're going to write another function with an `if`-chain that we'll then transform. The function takes a string opcode like `"add"` or `"mul"` and then does some math on the operands `x` and `y`.

To be honest, this is yet another toy example (I don't want to bore you with pages and pages of code here), but it'll serve well to illustrate the underlying design pattern. Once you "get" the pattern you'll be able to apply it in all kinds of different scenarios.

```
>>> def dispatch_if(operator, x, y):
...     if operator == 'add':
...         return x + y
...     elif operator == 'sub':
...         return x - y
...     elif operator == 'mul':
...         return x * y
...     elif operator == 'div':
...         return x / y
```

You can try out this `dispatch_if()` function to perform simple calculations by calling the function with a string opcode and two numeric operands:

```
>>> dispatch_if('mul', 2, 8)
16
>>> dispatch_if('unknown', 2, 8)
None
```

Please note that the `'unknown'` case works because Python adds an implicit `return None` statement to any function.

So far so good. Let's transform the original `dispatch_if()` into a new function which uses a dictionary to map opcodes to arithmetic operations with first-class functions.

```
>>> def dispatch_dict(operator, x, y):
...     return {
...         'add': lambda: x + y,
...         'sub': lambda: x - y,
...         'mul': lambda: x * y,
...         'div': lambda: x / y,
...     }.get(operator, lambda: None)()
```

This dictionary-based implementation gives the same results as the original `dispatch_if()`. We can call it exactly the same way:

```
>>> dispatch_dict('mul', 2, 8)
16
>>> dispatch_dict('unknown', 2, 8)
None
```

There are a couple of ways this code could be further improved if it was real "production-grade" code.

First of all, every time we call `dispatch_dict()` it creates a temporary dictionary and a bunch of lambdas for the opcode lookup. This isn't ideal from a performance perspective. For code that needs to be fast it makes more sense to create the dictionary once as a constant and to then reference it when the function is called. We don't want to recreate the dictionary every time we need to do a lookup.

Second, if we really wanted to do some simple arithmetic like `x + y` then we'd be better off using Python's built-in `operator` module instead of the lambda functions used in the example. The `operator` module provides implementations for all of Python's operators, for example `operator.mul`, `operator.div`, and so on. This is a minor point, though. The example intentionally uses lambdas to make it more generic and easier to apply the pattern in other situations.

Well, now you've got another tool in your toolbox to be able to simplify some of your `if`-chains if they get unwieldy. Just remember—this technique won't apply in every situation and sometimes you'll be better off with a plain `if`-statement.

Key Takeaways

- Python doesn't have a switch/case statement. But in some cases you can avoid long if-chains with a dictionary-based

dispatch table.
- Python's functions are first-class citizens. This is a powerful tool.
- With great power comes great responsibility.

3.7 What Namedtuples Are Good For

Python comes with a specialized "namedtuple" container type that doesn't seem to get the attention it deserves. It's one of these amazing features in Python that's hidden in plain sight.

Namedtuples can be a great alternative to defining a class manually and they have some other interesting features that I want to introduce you to in this chapter.

Now, what's a namedtuple and what makes it so special? A good way to think about namedtuples is to view them as an extension of the built-in tuple data type.

Python's tuples are a simple data structure for grouping arbitrary objects. Tuples are also immutable—they cannot be modified once they've been created.

```
>>> tup = ('hello', object(), 42)
>>> tup
('hello', <object object at 0x105e76b70>, 42)
>>> tup[2]
42
>>> tup[2] = 23
TypeError:
"'tuple' object does not support item assignment"
```

A downside of plain tuples is that the data you store in them can only be pulled out by accessing it through integer indexes. You can't give names to individual properties stored in a tuple. This can impact code readability.

Also, a tuple is always an ad-hoc structure. It's hard to ensure that two tuples have the same number of fields and the same properties

stored on them. This makes it easy to introduce "slip-of-the-mind" bugs by mixing up the field order.

Namedtuples to the Rescue

Namedtuples aim to solve these two problems.

First of all, namedtuples are immutable just like regular tuples. Once you store something in them you can't modify it.

Besides that, namedtuples are, well...*"named tuples."* Each object stored in them can be accessed through a unique (human-readable) identifier. This frees you from having to remember integer indexes, or resorting to workarounds like defining integer constants as mnemonics for your indexes.

Here's what a namedtuple looks like:

```
>>> from collections import namedtuple
>>> Car = namedtuple('Car' , 'color mileage')
```

To use namedtuples you need to import the `collections` module. They were added to the standard library in Python 2.6. In the above example we defined a simple "Car" data type with two fields: "color" and "mileage."

You might find the syntax a little weird here—Why are we passing the fields as a string encoding them `"color mileage"`?

The answer is that namedtuple's factory function calls `split()` on the field names string, so this is really just a shorthand to say the following:

```
>>> 'color mileage'.split()
['color', 'mileage']
>>> Car = namedtuple('Car', ['color', 'mileage'])
```

Of course you can also pass a list with string field names directly if you prefer how that looks. The advantage of using a proper list is that it's easier to reformat this code if you need to split it across multiple lines:

```
>>> Car = namedtuple('Car', [
...     'color',
...     'mileage',
... ])
```

However you decide, you can now create new "car" objects with the `Car` factory function. It behaves as if you had defined a `Car` class manually and given it a constructor accepting a "color" and a "mileage" value:

```
>>> my_car = Car('red', 3812.4)
>>> my_car.color
'red'
>>> my_car.mileage
3812.4
```

Besides accessing the values stored in a namedtuple by their identifiers, you can still access them by their index. That way namedtuples can be used as a drop-in replacement for regular tuples.

```
>>> my_car[0]
'red'
```

```
>>> tuple(my_car)
('red', 3812.4)
```

Tuple unpacking and the *-operator for function argument unpacking also work as expected:

```
>>> color, mileage = my_car
>>> print(color, mileage)
red 3812.4
>>> print(*my_car)
red 3812.4
```

You'll even get a nice string representation for free, which saves some typing and redundancy:

```
>>> my_car
Car(color='red' , mileage=3812.4)
```

Like tuples, namedtuples are immutable. When you try to overwrite one of their fields you'll get an `AttributeError` exception:

```
>>> my_car.color = 'blue'
AttributeError: "can't set attribute"
```

Namedtuple objects are implemented as regular Python classes internally. When it comes to memory usage they are also "better" than regular classes and just as memory efficient as regular tuples.

A good way to view them is to think that *namedtuples are a memory-efficient shortcut to defining an immutable class in Python manually*.

Subclassing Namedtuples

Because they are built on top of regular classes you can even add methods to a namedtuple's class. For example, you can extend the class like any other class and add methods and new properties to it that way. Here's an example:

```
>>> Car = namedtuple('Car', 'color mileage')
>>> class MyCarWithMethods(Car):
...     def hexcolor(self):
...         if self.color == 'red':
...            return '#ff0000'
...         else:
...             return '#000000'
```

We can now create `MyCarWithMethods` objects and call their `hexcolor()` method, just as expected:

```
>>> c = MyCarWithMethods('red', 1234)
>>> c.hexcolor()
'#ff0000'
```

However, this might be a little clunky. It might be worth doing if you want a class with immutable properties. But it's also easy to shoot yourself in the foot here.

For example, adding a new *immutable* field is tricky because of how namedtuples are structured internally. The easiest way to create hierarchies of namedtuples is to use the base tuple's `._fields` property:

```
>>> Car = namedtuple('Car', 'color mileage')
>>> ElectricCar = namedtuple(
...     'ElectricCar', Car._fields + ('charge',))
```

This gives the desired result:

```
>>> ElectricCar('red', 1234, 45.0)
ElectricCar(color='red', mileage=1234, charge=45.0)
```

Built-in Helper Methods

Besides the `_fields` property each namedtuple instance also provides a few more helper methods you might find useful. Their names all start with an underscore character (_) which usually signals that a method or property is "private" and not part of the stable public interface of a class or module.

With namedtuples the underscore naming convention has a different meaning though: These helper methods and properties *are* part of namedtuple's public interface. The helpers were named that way to avoid naming collisions with user-defined tuple fields. So go ahead and use them if you need them!

I want to show you a few scenarios where the namedtuple helper methods might come in handy. Let's start with the `_asdict()` helper. It returns the contents of a namedtuple as a dictionary:

```
>>> my_car._asdict()
OrderedDict([('color', 'red'), ('mileage', 3812.4)])
```

This is great for avoiding typos when generating JSON-output, for example:

```
>>> json.dumps(my_car._asdict())
'{"color": "red", "mileage": 3812.4}'
```

Another useful helper is the `_replace()` function. It creates a (shallow) copy of a tuple and allows you to selectively replace some of its fields:

```
>>> my_car._replace(color='blue')
Car(color='blue', mileage=3812.4)
```

Lastly, the `_make()` classmethod can be used to create new instances of a namedtuple from a sequence or iterable:

```
>>> Car._make(['red', 999])
Car(color='red', mileage=999)
```

When to Use Namedtuples

Namedtuples can be an easy way to clean up your code and to make it more readable by enforcing a better structure for your data.

I find, for example, that going from ad-hoc data types like dictionaries with a fixed format to namedtuples helps me express my intentions more clearly. Often when I attempt this refactoring I magically come up with a better solution for the problem I'm facing.

Using namedtuples over unstructured tuples and dicts can also make my coworkers' lives easier because they make the data being passed around "self-documenting" (to a degree).

On the other hand I try not to use namedtuples for their own sake if they don't help me write "cleaner" and more maintainable code. Like with most techniques shown in this book there can be *too much of a good thing*.

However if you use them with care, namedtuples can undoubtedly make your Python code better and more expressive.

Key Takeaways

- `collection.namedtuple` is a memory-efficient shortcut to defining an immutable class in Python manually.
- Namedtuples can help clean up your code by enforcing an easier to understand structure on your data.
- Namedtuples provide a few useful helper methods that all start with an _ underscore—but are part of the public interface. It's okay to use them.

3.8 • Instance, Class, and Static Methods Demystified

In this chapter I'll help demystify what's behind *class methods*, *static methods*, and regular *instance methods*.

If you develop an intuitive understanding for their differences you'll be able to write object-oriented Python that communicates its intent more clearly and will be easier to maintain in the long run.

Let's begin by writing a (Python 3) class that contains simple examples for all three method types:

```
class MyClass:
    def method(self):
        return 'instance method called', self

    @classmethod
    def classmethod(cls):
        return 'class method called', cls

    @staticmethod
    def staticmethod():
        return 'static method called'
```

Note for Python 2 users: The `@staticmethod` and `@classmethod` decorators are available as of Python 2.4 and this example will work as is. Instead of using a plain `class MyClass` declaration you might choose to declare a new-style class inheriting from `object` with the `class MyClass(object)` syntax. But other than that, you're golden!

Instance Methods

The first method on `MyClass`, called `method`, is a regular *instance method*. That's the basic, no-frills method type you'll use most of the time. You can see the method takes one parameter, `self`, which points to an instance of `MyClass` when the method is called (but of course instance methods can accept more than just one parameter).

Through the `self` parameter, instance methods can freely access attributes and other methods on the same object. This gives them a lot of power when it comes to modifying an object's state.

Not only can they modify object state, instance methods can also access the class itself through the `self.__class__` attribute. This means instance methods can also modify class state.

Class Methods

Let's compare that to the second method, `MyClass.classmethod`. I marked this method with a `@classmethod`[4] decorator to flag it as a *class method*.

Instead of accepting a `self` parameter, class methods take a `cls` parameter that points to the class—and not the object instance—when the method is called.

Because the class method only has access to this `cls` argument, it can't modify object instance state. That would require access to `self`. However, class methods can still modify class state that applies across all instances of the class.

[4] cf. docs.python.org/3/library/functions.html#classmethod

Static Methods

The third method, `MyClass.staticmethod` was marked with a `@staticmethod`[5] decorator to flag it as a *static method*.

This type of method takes neither a `self` nor a `cls` parameter (but of course it's free to accept an arbitrary number of other parameters).

Therefore a static method can neither modify object state nor class state. Static methods are restricted in what data they can access—and they're primarily a way to namespace your methods.

Let's See Them in Action!

I know this discussion has been fairly theoretical up to this point. And I believe it's important that you develop an intuitive understanding for how these method types differ in practice. We'll go over some concrete examples now.

Let's take a look at how these methods behave in action when we call them. We'll start by creating an instance of the class and then calling the three different methods on it.

`MyClass` was set up in such a way that each method's implementation returns a tuple containing information for us to trace what's going on—and which parts of the class or object the method can access.

Here's what happens when we call an **instance method**:

```
>>> obj = MyClass()
>>> obj.method()
('instance method called', <MyClass instance at 0x101a2>)
```

[5] cf. docs.python.org/3/library/functions.html#staticmethod

This confirmed that `method`, the instance method, has access to the object instance (printed as `<MyClass instance>`) via the `self` argument.

When the method is called, Python replaces the `self` argument with the instance object, `obj`. We could ignore the syntactic sugar of the dot-call syntax (`obj.method()`) and pass the instance object *manually* to get the same result:

```
>>> MyClass.method(obj)
('instance method called', <MyClass instance at 0x101a2>)
```

Can you guess what would happen if you tried to call the method without first creating an instance?

By the way, instance methods can also access the *class itself* through the `self.__class__` attribute. This makes instance methods powerful in terms of access restrictions—they can modify state on the object instance *and* on the class itself.

Let's try out the **class method** next:

```
>>> obj.classmethod()
('class method called', <class MyClass at 0x101a2>)
```

Calling `classmethod()` showed us it doesn't have access to the `<MyClass instance>` object, but only to the `<class MyClass>` object, representing the class itself (everything in Python is an object, even classes themselves).

Notice how Python automatically passes the class as the first argument to the function when we call `MyClass.classmethod()`. Calling a method in Python through the *dot syntax* triggers this behavior. The `self` parameter on instance methods works the same way.

Please note that naming these parameters `self` and `cls` is just a convention. You could just as easily name them `the_object` and `the_class` and get the same result. All that matters is that they're positioned first in the parameter list for the method.

Time to call the **static method** now:

```
>>> obj.staticmethod()
'static method called'
```

Did you see how we called `staticmethod()` on the object and were able to do so successfully? Some developers are surprised when they learn that it's possible to call a static method on an object instance.

Behind the scenes Python simply enforces the access restrictions by not passing in the `self` or the `cls` argument when a static method gets called using the dot syntax.

This confirms that static methods can neither access the object instance state nor the class state. They work like regular functions but belong to the class's (and every instance's) namespace.

Now, let's take a look at what happens when we attempt to call these methods on the class itself—without creating an object instance beforehand:

```
>>> MyClass.classmethod()
('class method called', <class MyClass at 0x101a2>)

>>> MyClass.staticmethod()
'static method called'

>>> MyClass.method()
TypeError: """unbound method method() must
```

```
    be called with MyClass instance as first
    argument (got nothing instead)"""
```

We were able to call `classmethod()` and `staticmethod()` just fine, but attempting to call the instance method `method()` failed with a `TypeError`.

And this is to be expected—this time we didn't create an object instance and tried calling an instance function directly on the class blueprint itself. This means there is no way for Python to populate the `self` argument and therefore the call fails.

This should make the distinction between these three method types a little more clear. But I'm not going to leave it at that. In the next two sections I'll go over two slightly more realistic examples for when to use these special method types.

I will base my examples around this bare-bones `Pizza` class:

```
class Pizza:
    def __init__(self, ingredients):
        self.ingredients = ingredients

    def __repr__(self):
        return f'Pizza({self.ingredients!r})'

>>> Pizza(['cheese', 'tomatoes'])
Pizza(['cheese', 'tomatoes'])
```

Delicious Pizza Factories With `@classmethod`

If you've had any exposure to pizza in the real world you'll know that there are many delicious variations available:

```
Pizza(['mozzarella', 'tomatoes'])
Pizza(['mozzarella', 'tomatoes', 'ham', 'mushrooms'])
Pizza(['mozzarella'] * 4)
```

The Italians figured out their pizza taxonomy centuries ago, and so these delicious types of pizzas all have their own names. We'd do well to take advantage of that and give the users of our `Pizza` class a better interface for creating the pizza objects they crave.

A nice and clean way to do that is by using class methods as *factory functions*[6] for the different kinds of pizzas we can create:

```
class Pizza:
    def __init__(self, ingredients):
        self.ingredients = ingredients

    def __repr__(self):
        return f'Pizza({self.ingredients!r})'

    @classmethod
    def margherita(cls):
        return cls(['mozzarella', 'tomatoes'])

    @classmethod
    def prosciutto(cls):
        return cls(['mozzarella', 'tomatoes', 'ham'])
```

Note how I'm using the `cls` argument in the `margherita` and `prosciutto` factory methods instead of calling the `Pizza` constructor directly.

[6] cf. en.wikipedia.org/wiki/Factory_(object-oriented_programming)

This is a trick you can use to follow the *Don't Repeat Yourself (DRY)*[7] principle. If we decide to rename this class at some point we won't have to remember updating the constructor name in all of the class-method factory functions.

Now, what can we do with these factory methods? Let's try them out:

```
>>> Pizza.margherita()
Pizza(['mozzarella', 'tomatoes'])

>>> Pizza.prosciutto()
Pizza(['mozzarella', 'tomatoes', 'ham'])
```

As you can see, we can use the factory functions to create new `Pizza` objects that are configured the way we want them. They all use the same `__init__` constructor internally and simply provide a shortcut for remembering all of the various ingredients.

Another way to look at this use of class methods is that they allow you to define alternative constructors for your classes.

Python only allows one `__init__` method per class. Using class methods it's possible to add as many alternative constructors as necessary. This can make the interface for your classes self-documenting (to a certain degree) and simplify their usage.

When To Use Static Methods

It's a little more difficult to come up with a good example here. But tell you what, I'll just keep stretching the pizza analogy thinner and thinner... (yum!)

Here's what I came up with:

[7] cf. en.wikipedia.org/wiki/Don't_repeat_yourself

```
import math

class Pizza:
    def __init__(self, radius, ingredients):
        self.radius = radius
        self.ingredients = ingredients

    def __repr__(self):
        return (f'Pizza({self.radius!r}, '
                f'{self.ingredients!r})')

    def area(self):
        return self.circle_area(self.radius)

    @staticmethod
    def circle_area(r):
        return r ** 2 * math.pi
```

Now what did I change here? First, I modified the constructor and `__repr__` to accept an extra `radius` argument.

I also added an `area()` instance method that calculates and returns the pizza's area (this would also be a good candidate for an `@property`—but hey, this is just a toy example).

Instead of calculating the area directly within `area()`, using the well-known circle area formula, I factored that out to a separate `circle_area()` static method.

Let's try it out!

```
>>> p = Pizza(4, ['mozzarella', 'tomatoes'])
>>> p
Pizza(4, {self.ingredients})
```

```
>>> p.area()
50.26548245743669
>>> Pizza.circle_area(4)
50.26548245743669
```

Sure, this is a bit of a simplistic example, but it'll do alright helping explain some of the benefits that static methods provide.

As we've learned, static methods can't access class or instance state because they don't take a `cls` or `self` argument. That's a big limitation—but it's also a great signal to show that a particular method is independent from everything else around it.

In the above example, it's clear that `circle_area()` can't modify the class or the class instance in any way. (Sure, you could always work around that with a global variable but that's not the point here.)

Now, why is that useful?

Flagging a method as a static method is not just a hint that a method won't modify class or instance state—this restriction is also enforced by the Python runtime.

Techniques like that allow you to communicate clearly about parts of your class architecture so that new development work is naturally guided to happen within these set boundaries. Of course, it would be easy enough to defy these restrictions. But in practice they often help avoid accidental modifications going against the original design.

Put differently, using static methods and class methods are ways to communicate developer intent while enforcing that intent enough to avoid most slip of the mind mistakes and bugs that would break the design.

Applied sparingly and when it makes sense, writing some of your

methods that way can provide maintenance benefits and make it less likely that other developers use your classes incorrectly.

Static methods also have benefits when it comes to writing test code.

Because the `circle_area()` method is completely independent from the rest of the class it's much easier to test.

We don't have to worry about setting up a complete class instance before we can test the method in a unit test. We can just fire away like we would testing a regular function. Again, this makes future maintenance easier.

Key Takeaways

- Instance methods need a class instance and can access the instance through `self`.
- Class methods don't need a class instance. They can't access the instance (`self`) but they have access to the class itself via `cls`.
- Static methods don't have access to `cls` or `self`. They work like regular functions but belong to the class's namespace.
- Static and class methods communicate and (to a certain degree) enforce developer intent about class design. This can have maintenance benefits.

3.9 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 see that there's little *magic* involved and it's actually a highly useful feature that can help you write cleaner and easier to understand Python.

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. A well-known example involves the `open()` function:

```
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 acquiring and releasing resources *properly* 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, `with` takes care of that for us.

The `with` statement can make code dealing with system resources more readable. It also helps avoid bugs or leaks by making it almost

impossible to forget cleaning up or releasing a resource after we're done with it.

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*[8].

What's a context manager? It's a simple "protocol" (or interface) that your object needs to follow so it can be used with 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 how 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()
```

[8] https://docs.python.org/3/reference/datamodel.html#context-managers

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

```
>>> 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`[9] 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 matches 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 with this technique looks like:

```
from contextlib import contextmanager

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

[9] https://docs.python.org/3/library/contextlib.html

```
    finally:
        f.close()

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

In this case, `managed_file()` is a generator that first acquires the resource. Then it temporarily suspends its own executing 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 happen and the resource gets released back to the system.

Both the class-based implementations and the generator-based are practically equivalent. Depending on which one you find more readable you might prefer one over the other.

A downside of the `@contextmanager`-based implementation might be that it requires understanding of advanced Python concepts, like decorators and generators.

Once again, making the right choice here comes down to what you and your team are comfortable using and 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:

```
hi!
    hello
        bonjour
hey
```

How would you implement a context manager to support this functionality?

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

Ready? Here's how we 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)
```

Another good exercise would be trying to refactor this code to be generator-based.

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.

3.10 Writing Pythonic Loops

One of the easiest ways to spot a developer with a background in C-style languages and who only recently picked up Python is to look at how they write loops.

For example, whenever I see a code snippet like this, that's an example of someone trying to write Python like it's C or Java:

```
my_items = ['a', 'b', 'c']

i = 0
while i < len(my_items):
    print(my_items[i])
    i += 1
```

Now, what's so "unpythonic" about this code? Two things:

First, it keeps track of the index `i` manually—initializing it to zero and then carefully incrementing it upon every loop iteration.

And second, it uses `len()` to get the size of a container in order to determine how often to iterate.

In Python you can write loops that handle both of these responsibilities automatically. It's a great idea to take advantage of that. If your code doesn't have to keep track of a running index it's much harder to write accidental infinite loops, for example. It also makes the code more concise and therefore more readable.

To refactor this first code example, I'll start by removing the code that manually updates the index. A good way to do that is with a `for`-loop in Python. Using the `range()` built-in I can generate the indexes automatically:

```
>>> range(len(my_items))
range(0, 3)

>>> list(range(0, 3))
[0, 1, 2]
```

The `range` type represents an immutable sequence of numbers. It's advantage over a regular `list` is that it always takes the same small amount of memory. Range objects don't actually store the individual values representing the number sequence—instead they function as iterators and calculate the sequence values on the fly[10].

Instead of incrementing `i` on each loop iteration I could now write something like this:

```
for i in range(len(my_items)):
    print(my_items[i])
```

This is better. However it still isn't very Pythonic—in most cases when you see code that uses `range(len(...))` to iterate over a container it can be improved and simplified even further.

As I mentioned, `for`-loops in Python are really "for-each" loops that can iterate over items from a container or sequence directly, without having to look them up by index. I can take advantage of that and simplify my loop even further:

```
for item in my_items:
    print(item)
```

[10] In Python 2 you'll need to use the `xrange()` built-in to get this memory-saving behavior, as `range()` will actually construct a list object.

I would consider this solution to be quite Pythonic. It's nice and clean and almost reads like pseudo code from a text book. I don't have to keep track of the container's size or a running index to access elements.

The container itself takes care of handing out the elements so they can be processed. If the container is ordered, so will be the resulting sequence of elements. If the container isn't ordered it will return its elements in an arbitrary order but the loop will still cover all of them.

Now, of course you won't always be able to rewrite your loops like that. What if you *need* the item index, for example? There's a Pythonic way to keep a running index that avoids the `range(len(...))` construct I recommended against. The `enumerate()` is helpful in this case:

```
>>> for i, item in enumerate(my_items):
...     print(f'{i}: {item}')

0: a
1: b
2: c
```

You see, iterators in Python can return more than just one value. They can return tuples with an arbitrary number of values that can then be unpacked right inside the `for`-statement.

This is very powerful. For example, you can use the same technique to iterate over the keys and values of a dictionary at the same time:

```
>>> emails = {
...     'Bob': 'bob@example.com',
...     'Alice': 'alice@example.com',
... }
```

```
>>> for name, email in emails.items():
...     print(f'{name} -> {email}')

'Bob -> bob@example.com'
'Alice -> alice@example.com'
```

There's one more example I'd like to show you. What if you absolutely, positively need to write a C-style loop. For example, what if you must control the step size for the index? Imagine you had the following original loop:

```
for (int i = a; i < n; i+=s) {
    // ...
}
```

How would this pattern translate to Python? The `range()` function comes to our rescue again—it can accept extra parameters to control the start value for the loop (`a`), the stop value (`n`), and the step size (`s`). Therefore our example C-style loop could be implemented as follows:

```
for i in range(a, n, s):
    # ...
```

Key Takeaways

- Writing C-style loops in Python is considered unpythonic. Avoid managing loop indexes and stop conditions manually if possible.
- Python's `for`-loops are really "for-each" loops that can iterate over items from a container or sequence directly.

3.11 Isolating Project Dependencies With Virtualenv

Python includes a powerful packaging system to manage the module dependencies of your programs. You've probably used it to install third-party packages with the `pip` package manager command.

One confusing aspect about installing packages with pip is that by default it tries to install them into your *global* environment.

Sure, this makes any new packages you install available globally on your system, which is great for convenience. But it also quickly turns into a nightmare if you're working with multiple projects that require different versions of the *same* package.

For example, what if one of your projects needs version 1.3 of a library while another project needs version 1.4 of the same library?

When you install packages globally *there can be only one* version of a Python library across all of your programs. This means you'll quickly run into version conflicts—just like the Highlander did.

And it gets worse. You might also have different programs that need different versions of Python itself. For example, some programs might still run on Python 2 while most of your new development happens in Python 3. Or, what if one of your project needs Python 3.3 while everything else runs on Python 3.6?

Besides that, installing Python packages globally can also incur a security risk. Modifying the global environment often requires you to run the `pip install` command with superuser (root/admin) credentials. Because pip downloads and executes code from the internet when you install a new package this is generally not recommended. Hopefully the code is trustworthy, but who knows what it will really do?

Virtual Environments to the Rescue

The solution to these problems is separating your Python environments with so-called *virtual environments*. They allow you to separate Python dependencies by project, including selecting between different versions of the Python interpreter.

A *virtual environment* is an isolated Python environment. Physically, it lives inside a folder containing all the packages and other dependencies, like native-code libraries and the interpreter runtime, that a Python project needs. (Behind the scenes those files might not be real copies but symbolic links to save memory.)

To demonstrate how virtual environments work I'll give you a quick walkthrough where we'll set up a new environment (or *virtualenv*, as they're called for short) and then install a third-party package into it.

Let's first check where the global Python environment currently resides. On Linux or macOS we can use the `which` command-line tool to look up the path to the pip package manager:

```
$ which pip3
/usr/local/bin/pip3
```

I usually put my virtual environments right into my project folders to keep them nice and separate. But you could also have a dedicated "python-environments" directory somewhere to hold all of your environments across projects. This choice is up to you.

Let's create a new Python virtual environment:

```
$ python3 -m venv ./venv
```

This will take a short moment and create a new `venv` folder in the current directory and seed it with a baseline Python 3 environment:

```
$ ls venv/
bin        include    lib        pyvenv.cfg
```

If you check the active version of pip (with the `which` command) you'll see it's still pointing to the global environment, `/usr/local/bin/pip3` in my case.

This means if you'd install packages now they'd still end up in the global Python environment. Creating a virtual environment folder isn't enough—you'll need to explicitly *activate* the new virtual environment:

```
$ source ./venv/bin/activate
(venv) $
```

Running the `activate` command configures your current shell session to use the Python and pip commands from the virtual environment instead[11].

Notice how this changed your shell prompt to include the name of the active virtual environment inside parentheses: `(venv)`. Let's check which pip executable is active now:

```
(venv) $ which pip3
/Users/dan/my-project/venv/bin/pip3
```

You see how running the `pip3` command would now actually run the version from the virtual environment and not the global one. The same is true for the Python interpreter executable. Running `python` from the command line would now also load the interpreter from the venv folder:

[11] On Windows there's an `activate` command you need to run directly instead of loading it with `source`.

```
(venv) $ which python
/Users/dan/my-project/venv/bin/python
```

Note that this is a blank slate, a completely clean environment. Running `pip list` will show an almost empty list of installed packages that only includes the baseline packages to support pip itself:

```
(venv) $ pip list
pip (9.0.1)
setuptools (28.8.0)
```

Let's go ahead and install a Python package now into the virtual environment now. You'll use the familiar `pip install` command for that:

```
(venv) $ pip install schedule
Collecting schedule
  Downloading schedule-0.4.2-py2.py3-none-any.whl
Installing collected packages: schedule
Successfully installed schedule-0.4.2
```

You'll notice two important changes here. First, you won't need admin permissions to run this command any longer. And second, installing or updating a package with an active virtual environment means that all files will end up in a subfolder in the virtual environment's directory.

Therefore, your project dependencies will be physically separated from all other Python environments, including the global one. In effect, you get a clone of the Python runtime that's dedicated to one project only.

Running `pip list` again you can see that the `schedule` library was installed successfully installed into the new environment:

```
(venv) $ pip list
pip (9.0.1)
schedule (0.4.2)
setuptools (28.8.0)
```

If we spin up a Python interpreter session with the `python` command, or run a standalone `.py` file with it, it will use the Python interpreter and the dependencies installed into the virtual environment—as long as the environment is still active in the current shell session.

But how do you deactivate or "leave" a virtual environment again? Similarly to the `activate` command there's a `deactivate` command that takes you back to the global environment:

```
(venv) $ deactivate
$ which pip3
/usr/local/bin
```

Using virtual environments will help keep your system uncluttered and your Python dependencies neatly organized. As a best practice, all of your Python projects should use virtual environments to keep their dependencies separate and to avoid version conflicts.

Understanding and using virtual environments also puts you on the right track to use more advanced dependency management methods like specifying project dependencies with `requirements.txt` files.

Key Takeaways

- Virtual environments keep your project dependencies separated. They help you avoid version conflicts between packages

and different versions of the Python runtime.

- As a best practice, all of your Python projects should use virtual environments to store their dependencies to avoid headaches.

Chapter 4

Pythonic Syntactic Sugar

4.1 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 one 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 it can help avoid silly mistakes and speed up code reviews.

Yet, there's two editing cases that can still cause confusion. Whenever you add a new item at the end or 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 the *DilbertJane*. This 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." [1]

Literal string concatenation is a useful feature in some cases. For example, you can use it to reduce the number of backslashes needed and to split long strings across lines. On the other hand, we've just

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

experienced how it can quickly turn into a liability. Now how do we fix this situation?

Adding the missing comma after *Dilbert* prevents the two string from getting merged, but it also 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? We've just come full circle and returned to the original problem.

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. Let me show you how.

Python allows you to end every line in a list, dict, or set constant with a comma—including the last line. That way you can just always end your items with a comma and thus avoid the comma placement fixes required otherwise.

Here's what the final example would look 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.

4.2 Function Argument Unpacking

A really cool but slightly arcane feature is the ability to "unpack" function arguments from sequences and dictionaries with the * and ** operator.

Let's define a simple function to work with as an example:

```
def print_vector(x, y, z):
    print('<%s, %s, %s>' % (x, y, z))
```

As you can see, this function takes three arguments x, y, and z and prints them nicely formatted. We might use this function to pretty-print 3-dimensional vectors in our program:

```
>>> print_vector(0, 1, 0)
<0, 1, 0>
```

Now depending on which data structure we choose to represent 3D vectors with, printing them with our *print_vector* function might feel a little awkward. For example, if our vectors are represented as tuples or lists we must explicitly specify the index for each component when printing them:

```
>>> tuple_vec = (1, 0, 1)
>>> list_vec = [1, 0, 1]
>>> print_vector(tuple_vec[0],
                 tuple_vec[1],
                 tuple_vec[2])
<1, 0, 1>
```

Using a normal function call with separate arguments seems unnecessarily verbose and cumbersome.

Thankfully here's a better way to handle this situation in Python with *Function Argument Unpacking* using the * operator:

```
>>> print_vector(*tuple_vec)
<1, 0, 1>
>>> print_vector(*list_vec)
<1, 0, 1>
```

Putting a * before will *unpack* the a sequence and pass them to a function requiring separate positional arguments.

By the way, the same technique also works for generator expressions. Using the * operator on a generator consumes all elements from the generator and passes them to the function:

```
>>> gen_expr = (x * x for x in range(3))
>>> print_vector(*gen_expr)
```

Besides the * operator for unpacking sequences like tuples, lists, and generators into positional arguments there's also the ** operator for unpacking keyword arguments from dictionaries. Imagine our vector was represented as the following dict:

```
>>> dict_vec = {'x': 1, 'y': 0, 'z': 1}
```

We could pass this dict to `print_vector` in much the same way using the ** operator for unpacking:

```
>>> print_vector(**dict_vec)
<1, 0, 1>
```

Because dictionaries are unordered (before Python 3.6) this matches up dictionary values and function arguments based on their key. If

we used the * operator to unpack the dictionary we'd pass the dictionary's keys in random order instead:

```
>>> print_vector(*dict_vec)
<y, x, z>
```

Python's function argument unpacking feature gives you a lot of flexibility for free. Often this means you won't have to implement a class for a data type needed by your program—frequently, using a simple built-in data structure like a tuple or a list will suffice and help reduce the complexity of your code.

Key Takeaways

- The * and ** operators can be used to "unpack" function arguments from sequences and dictionaries.
- Using argument unpacking effectively can help you write more flexible interfaces for your modules and functions.

4.3 So Many Ways To Merge Dictionaries

Have you ever built a configuration system for one of your Python programs? A common use case for such systems is to take a data structure with default configuration options, and then to allow the defaults to be overridden selectively from user input or some other config source.

I often found myself using dictionaries as the underlying data structure for representing configuration keys and values. And so I frequently needed a way to combine or to *merge* the config defaults and the user overrides into a single dictionary with the final configuration values.

Or, to generalize: sometimes you need a way to merge two or more dictionaries into one, so that the resulting dictionary contains a combination of the keys and values of the source dicts.

In this Python Trick I'll show you a couple of ways to achieve that. Let's look at a simple example first so we have something to discuss. Imagine you had these two source dictionaries:

```
>>> xs = {'a': 1, 'b': 2}
>>> ys = {'b': 3, 'c': 4}
```

Now, you want to create a new dict `zs` that contains all of the keys and values of `xs` and all of the keys and values of `ys`. Also, if you read the example closely, you saw that the string `'b'`appears as a key in both dicts—we'll need to think about a conflict resolution strategy for duplicate keys as well.

The classical solution for the "merging multiple dictionaries" problem in Python is to use the built-in dictionary `update()` method:

```
>>> zs = {}
>>> zs.update(xs)
>>> zs.update(ys)
```

If you're curious, a naive implementation of `update()` might look something like this:

```
>>> def update(dict1, dict2):
...     for key, value in dict2.items():
...         dict1[key] = value
```

This results in a new dictionary `zs` which now contains the keys defined in `xs` and `ys`:

```
>>> zs
>>> {'c': 4, 'a': 1, 'b': 3}
```

You'll also see that the order in which we call `update()` determines how conflicts are resolved—the last update wins and the duplicate key `'b'` is associated with the value `3` that came from, `ys`, the second source dict.

Of course you could expand this chain of `update()` calls for as long as you'd like, to merge any number of dictionaries into one. It's a practical and well-readable solution that works in Python 2 and Python 3.

Another technique that works in Python 2 and Python 3 uses the `dict()` built-in combined with the `**`-operator for "unpacking" objects:

```
>>> zs = dict(xs, **ys)
>>> zs
{'a': 1, 'c': 4, 'b': 3}
```

However, this approach only works for merging *two* dictionaries and cannot be genericized to combine an arbitrary number of dictionaries.

Starting with Python 3.5 the `**`-operator became more flexible[2]. So in Python 3.5+ there's another—and arguably, prettier—way to merge an arbitrary number of dictionaries:

```
>>> zs = {**xs, **ys}
```

This expression has the exact same result as a chain of `update()` calls. Keys and values are set in a left-to right order so we get the same conflict resolution strategy: the right hand side takes priority and a value in `ys` overrides any existing value under the same key in `xs`. This becomes clear when we look at the dictionary resulting from the merge operation:

```
>>> zs
>>> {'c': 4, 'a': 1, 'b': 3}
```

Personally, I like the terseness of this new syntax and how it still remains sufficiently readable. There's always a fine balance between verbosity and terseness to keep the code as readable and maintainable as possible.

In this case I'm leaning towards using the new syntax if I'm working with Python 3. Using the `**`-operator is also faster than using chained `update()` calls, which is yet another benefit.

[2] cf. PEP 448: Additional Unpacking Generalizations

4.4 • Nothing To Return Here

Python adds an implicit `return None` statement to the end of any function. Therefore, if a function doesn't specify a return value it returns `None` by default.

This means you can replace `return None` statements with bare `return` statements or even leave them out completely and still get the same result:

```
def foo1(value):
    if value:
        return value
    else:
        return None

def foo2(value):
    """Bare return statement implies `return None`"""
    if value:
        return value
    else:
        return

def foo3(value):
    """Missing return statement implies `return None`"""
    if value:
        return value
```

All three functions properly return `None` if you pass them a falsy value:

```
>>> type(foo1(0))
<class 'NoneType'>
```

```
>>> type(foo2(0))
<class 'NoneType'>

>>> type(foo3(0))
<class 'NoneType'>
```

Now, when is it a good idea to use this feature in your own Python code?

My rule of thumb is if a function *doesn't have a return value* (other languages would call this a *procedure*) then I leave out the `return` statement. Adding one would just be confusing. An example for a procedure would be Python's built-in `print` function which is only called for its side-effects (printing text) and never for its return value.

Let's take a function like Python's built-in `sum`. It clearly has a logical return value—typically `sum` wouldn't get called only for its side-effects. Its purpose is to add a sequence of numbers together and then deliver the result. Now, if a function *does* have a return value from a logical point of view, then you need to decide whether to use an implicit return or not.

On the one hand, you could argue that omitting an explicit `return None` statement makes the code more concise and therefore easier to read and understand. Subjectively you might also say it makes the code "prettier."

On the other hand, it might surprise some programmers that Python behaves this way. When it comes to writing clean and maintainable code, surprising behavior is rarely a good sign.

For example, I've been using an "implicit return statement" in one of the code samples in an earlier revision of the book. I didn't mention what I was doing, I just wanted a nice and short code sample to

explain some other feature in Python.

Eventually I started getting a steady stream of emails pointing me to "the missing return statement" in that code example. Python's implicit return behavior was clearly *not* obvious to everybody and a distraction in this case. I added a note to make it clear what was going on, and the emails stopped.

Don't get me wrong—I love writing clean and "beautiful" code as much as anyone. And I used to feel strongly that programmers should know the ins and outs of the language they're working with.

But when you consider the maintenance impact of even such a simple misunderstanding, it might make sense to lean towards writing more explicit and clear code. After all, *code is communication*.

Key Takeaways

- If a function doesn't specify a return value, it returns `None`. Whether to explicitly return `None` is a stylistic decision.
- This is a core Python feature but your code might communicate its intent more clearly with an explicit `return None` statement.

Chapter 5

Closing Thoughts

I hope you enjoyed this copy of *Python Tricks: The Book*! If you'd like to let me know about an error, or if you just have a question, or want to offer some constructive feedback, then please feel free to email me anytime at mail@dbader.org or send me a Twitter direct message at @dbader_org.

— Dan Bader

Free Weekly Tips for Python Developers

There's one more thing I'd like to tell you about. Want a weekly dose of Python development tips to improve your productivity and streamline your workflow? Good news—I'm running a weekly newsletter for Python developers just like you.

If you'd like to become a member and try it out then head on over to dbader.org/newsletter and drop your email address in the signup form. I'm looking forward to meet you!

This is a work-in-progress snapshot from "Python Tricks: The Book"

Thanks for supporting my work and purchasing this work-in-progress version of my book. It really means a lot to me :)

As I finish more chapters you'll receive updates and new versions of the book for free through Gumroad.

If you've got any questions or feedback for me then feel free to reach out to me at mail@dbader.org.

Thanks!

— Dan Bader
