# Appendix C. Top Ten Things we Didn't Cover: There's Always More to Learn

I think we have a problem here. There are a whole bunch of things they didn't cover.

**It was never our intention to try to cover everything.**

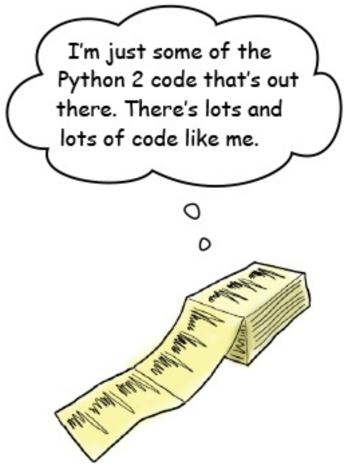
This book's goal was always to show you enough Python to get you up to speed as quickly as possible. There's a lot more we could've covered, but didn't. In this appendix, we discuss the top 10 things that—given another 600 pages or so—we would've eventually gotten around to. Not all of the 10 things will interest you, but quickly flip through them just in case we've hit on your sweet spot, or provided an answer to that nagging question. All the programming technologies in this appendix come baked in to Python and its interpreter.

## 1. What About Python 2?

As of this book's publication date (late 2016) there are two mainstream flavors of Python in widespread use. You already know quite a bit about **Python 3**, as that's the flavor you've used throughout this book.

All new language developments and enhancements are being applied to Python 3, which is on a 12- to 18-month minor release cycle. Release 3.6 is due before 2016 ends, and you can expect 3.7 to arrive late in 2017 or early in 2018.

Python 2 has been "stuck" at release 2.7 for some time now. This has to do with the fact that the *Python core developers* (the people who guide the development of Python) decided that Python 3 was the future, and that Python 2 should quietly go away. There were solid technical reasons for this approach, but no one really expected things to take so long. After all, Python 3—the future of the language—first appeared in late 2008.


I'm just some of the Python 2 code that's out there. There's lots and lots of code like me.

An entire book could be written on what's happened since late 2008 until now. Suffice it to say, Python 2 stubbornly refused to go away. There was (and still is) a huge installed base of Python 2 code and developers, with some domains dragging their heels when it comes to upgrading. There's a very simple reason for why this is: Python 3 introduced a handful of enhancements that broke backward compatibility. Put another way: there's lots of Python 2 code that will not run *unchanged* in Python 3 (even though, at a first glance, it can be hard to tell Python 2 code from Python 3 code). Also, many programmers simply believed Python 2 was "good enough," and didn't upgrade.

Recently (over the last year), there's been a sea change. The switching rate from 2 to 3 appears to be increasing. Some very popular third-party modules have released Python 3–compatible versions, and this is having a positive effect on Python 3 adoption. Additionally, the *Python core developers* keep adding extra goodness to Python 3, making it a more attractive programming language over time. The practice of "backporting" the cool new features from 3 to 2 has stopped with 2.7, and although bug and security fixes are still being applied, the *Python core developers* have announced that this activity will stop in 2020. The clock is ticking for Python 2.

Here's the common advice offered when you're trying to decide whether 3 or 2 is right for you:

**If you're starting a new project, use Python 3.**

You need to resist the urge to create more legacy code in Python 2, especially if you're starting with a blank slate. If you have to maintain some existing Python 2 code, what you know about 3 carries over: you'll certainly be able to read the code and understand it (it's still Python, regardless of the major version number). If there are technical reasons why the code has to remain running in Python 2, then so be it. If, however, the code can be ported to Python 3 without too much fuss, then we believe the gain is worth the pain, as Python 3 *is* the better language, and *is* the future.

## 2. Virtual Programming Environments

Let's imagine you have two clients, one with Python code that relies on one version of a third-party module, and another that relies on a *different* version of the same third-party module for their code. And, of course, you're the poor soul who has to maintain both projects' code.

Doing so on one computer can be problematic, as the Python interpreter doesn't support the installation of different versions of third-party modules.

That said, help is at hand thanks to Python's notion of virtual environments.

A **virtual environment** lets you create a new, clean Python environment within which you can run your code. You can install third-party modules into one virtual environment without impacting another, and you can have as many virtual environments as you like on your computer, switching between them by *activating* the one you want to work on. As each virtual environment can maintain its own copy of whatever third-party modules you wish to install, you can use two different virtual environments, one for each of your client projects discussed above.

Before doing so, however, you have to make a choice: use the virtual environment technology, called `venv`, that ships with Python 3's *standard library*, or install the `virtualenv` module from PyPI (which does the same thing as `venv`, but has more bells and whistles). It's best if you make an informed choice.

I'm pretty sure I've solved my multiple third-party module problem... all I had to do was read all of these.
All he had to do was use a virtual environment.

To learn more about `venv`, check out its documentation page:

```
https://docs.python.org/3/library/venv.html
```

To find out what `virtualenv` offers over and above `venv`, start here:

```
https://pypi.org/project/virtualenv/
```

Whether you use virtual environments for your projects is a personal choice. Some programmers swear by them, refusing to write any Python code unless it's within a virtual environment. This may be a bit of an extreme stance, but to each their own.

We chose not to cover virtual environments in the main body of this book. We feel virtual environments are—if you need them—a total godsend, but we don't yet believe every Python programmer needs to use one for everything they do.

We recommend you slowly back away from people who say that you aren't a proper Python programmer *unless* you use `virtualenv`.

## 3. More on Object Orientation

If you've read through this entire book, by now you'll (hopefully) appreciate what's meant by this phrase: "In Python, everything's an object."

Python's use of objects is great. It generally means that things work the way you expect them to. However, the fact that everything's an object does **not** mean that everything has to belong to a class, especially when it comes to your code.

In this book, we didn't learn how to create our own class until we needed one in order to create a custom context manager. Even then, we only learned as much as was needed, and nothing more. If you've come to Python from a programming language that insists all your code resides in a class (with *Java* being the classic example), the way we've gone about things in this book may be disconcerting. Don't let this worry you, as Python is much less strict than *Java* (for instance) when it comes to how you go about writing your programs.

If you decide to create a bunch of functions to do the work you need to do, then have at it. If your brain thinks in a more functional way, Python can help here too with the comprehension syntax, tipping its hat to the world of functional programming. And if you can't get away from the fact that your code needs to reside in a class, Python has full-featured object-oriented-programming syntax built right in.

OK, chaps...let's think about this for a moment. Does that code really need to be in a class?

If you do end up spending a lot of time creating classes, check out the following:

- `@staticmethod`: A decorator that lets you create a static function within a class (which does not receive `self` as its first argument).
- `@classmethod`: A decorator that lets you create a class method that expects a class as its first object (usually referred to as `cls`), not `self`.
- `@property`: A decorator that allows you to redesignate and use a method as if it were an attribute.
- `__slots__`: A class directive that (when used) can greatly improve the memory efficiency of the objects created from your class (at the expense of some flexibility).

To learn more about any of these, consult the Python docs (*https://docs.python.org/3/*). Or check out some of our favorite Python books (discussed in the next appendix).

## 4. Formats for Strings and the Like

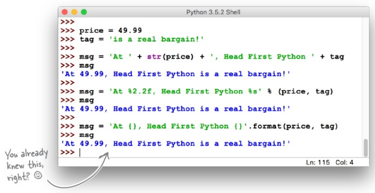
The recurring example application used in this book displayed its output in a web browser. This allowed us to defer any output formatting to HTML (specifically, we used the *Jinja2* module included with *Flask*). In doing so, we sidestepped one area where Python shines: text-based string formatting.

Let's say you have a string that needs to contain values that won't be known until your code runs. You want to create a message (`msg`) that contains the values so you can perform some later processing (perhaps you're going to print the message on screen, include the message within an HTML page you're creating with *Jinja2*, or tweet the message to your 3 million followers). The values your code generates at runtime are in two variables: `price` (the price of the item in question) and `tag` (a catchy marketing tagline). You have a few options here:

- Build the message you need using concatenation (the `+` operator).

- Use old-style string formats (using the `%` syntax).
- Take advantage of every string’s `format` method to build your message.

Here’s a short `>>>` session showing each of these techniques in action (bearing in mind that you, having worked through this book, already concur with what the generated message is telling you):



```
Python 3.5.2 Shell
>>>
>>> price = 49.99
>>> tag = 'is a real bargain!'
>>>
>>> msg = 'At ' + str(price) + ', Head First Python ' + tag
>>> msg
'At 49.99, Head First Python is a real bargain!'
>>>
>>> msg = 'At %2.2f, Head First Python %s' % (price, tag)
>>> msg
'At 49.99, Head First Python is a real bargain!'
>>>
>>> msg = 'At {}, Head First Python {}'.format(price, tag)
>>> msg
'At 49.99, Head First Python is a real bargain!'
>>> |
                                                  Ln: 115  Col: 4
```

Which of these techniques you use is a personal preference, although there’s a bit of a push on to encourage the use of the `format` method over the other two (see *PEP 3101* at https://www.python.org/dev/peps/pep-3101/). You’ll find code in the wild that uses one technique over the other, and sometimes (and not at all helpfully) mixes all three. To learn more, start here:

***https://docs.python.org/3/library/string.html#formatspec***

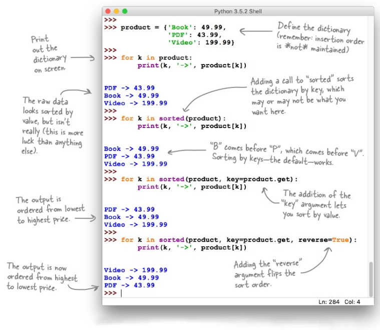
The %s and %f format specifiers are as old as the hills...but, hey, like me, they still work.

## 5. Getting Things Sorted

Python has wonderful built-in sorting capabilities. Some of the built-in data structures (lists, for example) contain `sort` methods that can be used to perform in-place ordering of your data. However, it is the `sorted` BIF that makes Python truly special (as this BIF works with *any* of the built-in data structures).

**BIF is short for “built-in function.”**

In the IDLE session below, we first define a small dictionary (`product`), which we then process with a succession of `for` loops. The `sorted` BIF is exploited to control the order in which each `for` loop receives the dictionary’s data. Follow along on your computer while you read the annotations:


```
>>>
>>> product = {'Book': 49.99,
               'PDF': 43.99,
               'Video': 199.99}
>>>
>>> for k in product:
        print(k, '->', product[k])


PDF -> 43.99
Book -> 49.99
Video -> 199.99
>>>
>>> for k in sorted(product):
        print(k, '->', product[k])


Book -> 49.99
PDF -> 43.99
Video -> 199.99
>>>
>>> for k in sorted(product, key=product.get):
        print(k, '->', product[k])


PDF -> 43.99
Book -> 49.99
Video -> 199.99
>>>
>>> for k in sorted(product, key=product.get, reverse=True):
        print(k, '->', product[k])


Video -> 199.99
Book -> 49.99
PDF -> 43.99
>>> |
```
Python 3.5.2 Shell
Define the dictionary (remember: insertion order is *not* maintained)
Print out the dictionary on screen.
The raw data looks sorted by value, but isn't really (this is more luck than anything else).
Adding a call to "sorted" sorts the dictionary by key, which may or may not be what you want here.
"B" comes before "P", which comes before "V". Sorting by keys—the default—works.
The addition of the "key" argument lets you sort by value.
The output is ordered from lowest to highest price.
Adding the "reverse" argument flips the sort order.
The output is now ordered from highest to lowest price.
Ln: 284 Col: 4

Learn more about how to sort with Python from this wonderful *HOWTO*:

***https://docs.python.org/3/howto/sorting.html#sortinghowto***

## 6. More from the Standard Library

Python's *standard library* is full of goodness. It's always a worthy exercise to take 20 minutes every once in a while to review what's available, starting here:

***https://docs.python.org/3/library/index.html***

If what you need is in the *standard library*, don't waste your precious time rewriting it. Use (and/or extend) what's already available. In addition to the Python docs, *Doug Hellmann* has ported his popular *Module of the Week* material over to Python 3. Find Doug's excellent material here:

***https://pymotw.com/3/***

We've reviewed a few of our favorite *standard library* modules below. Note that we can't stress enough how important it is to know what's in the *standard library*, as well as what all the provided modules can do for you.

### COLLECTIONS

This module provides importable data structures, over and above the built-in list, tuple, dictionary, and set. There's lots to like in this module. Here's an abbreviated list of what's in `collections`:

- `OrderedDict`: A dictionary that maintains insertion order.
- `Counter`: A class that makes counting things almost too easy.
- `ChainMap`: Combines one or more dictionaries and makes them appear as one.

Yes, yes...I get the joke, and it's very droll: "batteries included," right?
HEAVY DUTY
ALKALINE

### ITERTOOLS

You already know Python's `for` loop is great, and when reworked as a comprehension, looping is crazy cool. This module, `itertools`, provides a large collection of tools for building custom iterations. This module has a lot to offer, but be sure to also check out `product`, `permutations`, and `combinations` (and once you do, sit back and thank your lucky stars you didn't have to write any of that code).

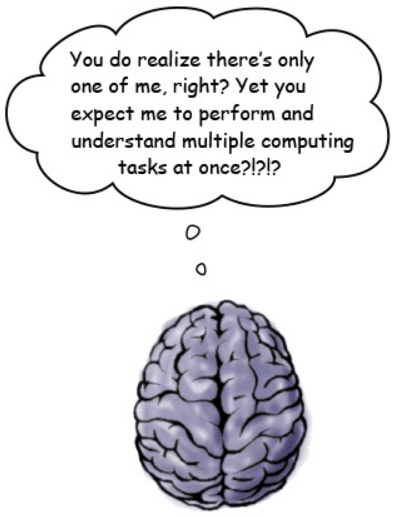
### FUNCTOOLS

The `functools` library provides a collection of higher-order functions (functions that take function objects as arguments). Our favorite is `partial`, which lets you “freeze” argument values to an existing function, then invoke the function with a new name of your choosing. You won’t know what you’re missing until you try it.

## 7. Running Your Code Concurrently

In Chapter 11¾, you used a thread to solve a waiting problem. Threads are not the only game in town when it comes to running code concurrently within your programs, although, to be honest, threads are the most used and abused of all of the available techniques. In this book, we deliberately kept our use of threads as simple as possible.

There are other technologies available to you when you find yourself in a situation where your code has to do more than one thing at once. Not every program needs these types of services, but it is nice to know that Python has a bunch of choices in this area should the need arise.


You do realize there’s only one of me, right? Yet you expect me to perform and understand multiple computing tasks at once?!?!?

In addition to the `threading` module, here are some modules worth checking out (and we also refer you back one page to #6, as *Doug Hellmann* has some great posts on some of these modules):

- `multiprocessing`: This module allows you to spawn multiple Python processes, which—if you have more than one CPU core—can spread your computational load across many CPUs.
- `asyncio`: Lets you specify concurrency via the creation and specification of coroutines. This is a relatively new addition to Python 3, so—for many programmers—it’s a very new idea (and the jury is still out).
- `concurrent.futures`: Lets you manage and run a collection of tasks concurrently.

Which of these is right for you is a question you’ll be able to answer once you’ve tried each of them with some of your code.

### NEW KEYWORDS: ASYNC AND AWAIT

The `async` and `await` keywords were added in Python 3.5, and provide a standard way to create coroutines.

The `async` keyword can be used in front of the existing `for`, `with`, and `def` keywords (with the `def` usage receiving the most attention to date). The `await` keyword can be used in front of (almost) any other code. As of the end of 2016, `async` and `await` are very new, and Python programmers the world over are only just beginning to explore what they can do with them.

GEEK BITS

“GIL” stands for “Global Interpreter Lock”. The GIL is an internal mechanism used by the interpreter to ensure stability. Its continued use within the interpreter is the subject of much discussion and debate within the Python community.

The Python docs have been updated with information on these new keywords, but, for our money, you’ll find the best descriptions of their use (and the craziness that using them induces) by searching *YouTube* for anything on the topic by *David Beazley*. **Be warned**: David’s talks are always excellent, but do tend to lean toward the more advanced topics in the Python language ecosystem.

David’s talks on Python’s *GIL* are regarded as classics by many, and his books are great too; more on this in *Appendix E*.

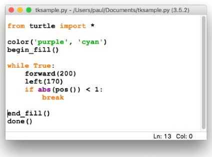
## 8. GUIs w ith Tkinter (and Fun w ith Turtles)

Python comes with a complete library called `tkinter` (the *Tk interface*) for building cross-platform GUIs. You may not realize it, but you’ve been using an application from the very first chapter of this book that is built with `tkinter`: IDLE.

What’s neat about `tkinter` is that it comes preinstalled (and ready for use) with every Python installation that includes IDLE (i.e., nearly all of them). Despite this, `tkinter` doesn’t receive the use (and love) it deserves, as many believe it to be unnecessarily clunky (compared to some third-party alternatives). Nevertheless, and as IDLE demonstrates, it is possible to produce useful and usable programs with `tkinter`. (Did we mention that `tkinter` comes preinstalled and ready for use?)

One such usage is the `turtle` module (which is also part of the *standard library*). To quote the Python docs: *Turtle graphics is a popular way for introducing programming to kids. It was part of the original Logo programming language developed by Wally Feurzig and Seymour Papert in 1966*. Programmers (i.e., mainly kids, but fun for newbies, too) can use commands like `left`, `right`, `pendown`, `penup`, and so on to draw on a GUI canvas (provided by `tkinter`).

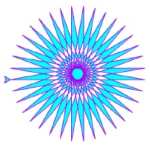
Here’s a small program, which has been adapted ever so slightly from the example that comes with the `turtle` docs:



```
from turtle import *

color('purple', 'cyan')
begin_fill()

while True:
    forward(200)
    left(170)
    if abs(pos()) < 1:
        break

end_fill()
done()
```

As well as showing “turtle” in action, this small program also demonstrates the use of Python’s “while” loop and “break” statement. They work exactly as you’d expect them to, but don’t see nearly as much action as the “for” loop and comprehensions.

And when this small `turtle` program is executed, a thing of beauty is drawn and appears on screen:



We know you can do better than this, so why not give “turtle” a try?

## 9. It’s Not Over ’Til It’s Tested

This book has barely mentioned automated testing, aside from a passing nod to the `py.test` tool for checking conformance to *PEP 8* (at the end of Chapter 4). This is not because we think automated testing isn’t important. **We think automated testing is very important**. It is such an important topic that entire books are dedicated to it.

That said, in this book, we avoided automated testing tools on purpose. This has nothing to do with how we feel about automated testing (it really *is* very important). However, when you are first learning to program in a new programming language, introducing automated testing can confuse more than it clarifies, as the creation of tests assumes a good understanding of the thing being tested, and if that “thing” happens to be a new programming language that you’re learning...well, you can see where we’re going with this, can’t you? It’s a bit like the chicken and the egg. Which comes first: learning to code, or learning how to test the code you’re learning?

Of course, now that you’re a bona-fide Python programmer, you can take the time to understand how Python’s *standard library* makes it easy to test your code. There are two modules to look at (and consider):

- `doctest`: This module lets you embed your tests in your module’s docstrings, which isn’t as weird as it sounds and *is* very useful.
- `unittest`: You may have already used a “unittest” library with another programming language, and Python comes with its very own version (which works exactly as you’d expect it to).

The `doctest` module is adored by those who use it. The `unittest` module works like most other “unittest” libraries in other languages, and a lot of Python programmers complain that it’s not *pythonic* enough. This has led to the creation of the hugely popular `py.test` (which we talk more about in the next appendix).

Don’t you dare flip on over to the next page before you’ve thought about how you’d use Python’s automated testing tools to check the correctness of the code you write.
STOP

## 10. Debug, Debug, Debug

You’d be forgiven for thinking that the vast majority of Python programmers revert to adding `print` calls to their code when something goes wrong. And you wouldn’t be far off: it’s a popular debugging technique.

Another is experimenting at the `>>>` prompt, which—if you think about it—is very like a debugging session *without* the usual debugging chores of watching traces and setting up breakpoints. It is impossible to quantify how productive the `>>>` prompt makes Python programmers. All we know is this: if a future release of Python decides to remove the interactive prompt, things will get ugly.

**You can learn all about traces and breakpoints by working through the “pdb” docs.**

If you have code that’s not doing what you think it should, and the addition of `print` calls as well as experimenting at the `>>>` prompt have left you none the wiser, consider using Python’s included debugger: `pdb`.

It’s possible to run the `pdb` debugger directly from your operating system’s terminal window, using a command like this (where `myprog.py` is the program you need to fix):

```
python3 -m pdb myprog.py
```

As always, Windows users need to use “py -3” instead of “python3”. (That’s “py”, space, then minus 3).

It’s also possible to interact with `pdb` from the `>>>` prompt, which is as close an instantiation of “the best of both worlds” as we think you’ll ever come across. The details of how this works, as well as a discussion of all the usual debugger commands (set a breakpoint, skip, run, etc.) are in the docs:

***https://docs.python.org/3/library/pdb.html***

The pdb technology is not an "also ran," nor was it an afterthought; it's a wonderfully feature-full debugger for Python (and it comes built-in).

Make sure a working understanding of Python's "pdb" debugger is part of your toolkit.

PREV
B. Pythonanywhere: Deploying Your Webapp

NEXT
D. Top Ten Projects not Covered: Even More Tools, Libraries, an…