Advanced Python Development

Using Powerful Language Features in Real-World Applications

—

Matthew Wilkes

Apress®

Advanced Python Development

Using Powerful Language Features in Real-World Applications

Matthew Wilkes

Apress®

Advanced Python Development: Using Powerful Language Features in Real-World Applications

Matthew Wilkes
Leeds, West Yorkshire, UK

ISBN-13 (pbk): 978-1-4842-5792-0 ISBN-13 (electronic): 978-1-4842-5793-7
https://doi.org/10.1007/978-1-4842-5793-7

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Celestin Suresh John
Development Editor: James Markham
Coordinating Editor: Aditee Mirashi

Cover designed by eStudioCalamar

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

Photographs by Mark Wheelwright (Matthew), Stephanie Shadbolt (Jesse) and Niteo GmbH (Nejc)

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

For information on translations, please e-mail rights@apress.com, or visit http://www.apress.com/rights-permissions.

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

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/978-1-4842-5792-0. For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

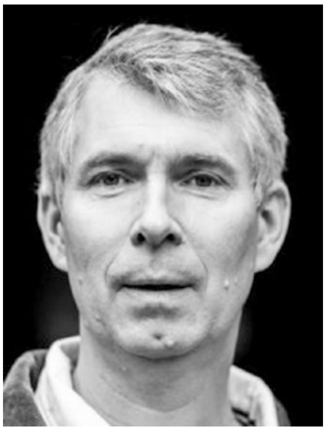
Table of Contents

About the Author

Matthew Wilkes is a European software developer who has worked with Python on web projects for the last 15 years. As well as developing software, he has a long experience in mentoring Python developers in a commercial setting.

He is also very involved in open source software, with commits to many popular frameworks. His contributions in that space are focused on the details of database and security interactions of web frameworks.

About the Technical Reviewers



Coen de Groot is a freelance Python developer and trainer. He has been passionate about computers and programming since the late 1970s when he built his first "computer."

After nearly finishing his computer science degree at Leiden University, Coen has worked for a large oil company, small startups, software agencies, and others. He has written a lot of software in many different programming languages. And he has worked in software support, delivered training, led teams, and managed technical projects.

After about 20 years in IT, Coen tried something different, trained as a business coach, hosted a large community of coaches, and organized five conferences. But he quickly got pulled back into building websites and other IT services for coaches and others.

For the last 10 years, Coen has mostly been programming in Python, with hints of SQL, JavaScript, and others. And he still enjoys learning more Python and passing on that knowledge face to face, in writing or on video.

Geek since he was able to walk, **Nejc Zupan** developed his first computer game in primary school, won the national robotics championship in high school, and cofounded niteo.co while still in college. He has spoken at conferences in five continents, mostly relating to the Web, Python, and productivity. Whenever he is not coding, he is chasing big waves around the world.

Jesse Snyder began programming after many years of deferring graduate studies in ethnomusicology and was pleasantly surprised by how completely engrossing he found the challenges and rewards of software design. After several years in the Pacific Northwest nonprofit technology scene, he now works as an independent consultant. When not at work or playing Javanese gamelan music, he is likely out for a long run through the beautiful parks and neighborhoods around his home in Seattle, Washington.

Acknowledgments

Many people helped with this book in various ways. The thousands of contributors to Python's open source ecosystem must come first; without them there would be no book to write. Thank you to Joanna for encouraging me, despite the difficulty and long hours. Thanks also to the rest of my family for their unfailing support over the years.

For this book specifically, I'd like to thank Nejc Zupan, Jesse Snyder, Tom Blockley, Alan Hoey, and Cris Ewing, all of whom gave valuable comments on the plan and implementation. Thank you also to Mark Wheelwright of ISO Photography for his excellent work in getting a good photograph of me and to the team at Apress for their hardwork.

Finally, thank you to the people who continue to make the Web as weird and wonderful a thing as it was when I was first drawn into working with the Internet. Thomas Heasman-Hunt, Julia Evans, Ian Fieggen, Foone Turing, and countless more – I doubt industrial software would have captured so much of my attention without people like you.

Introduction

Python is a very successful programming language. In the three decades that it has existed, it has become very widely used. It ships by default with major operating systems; some of the largest websites in the world use Python for their back ends, and scientists are using Python every day to advance our collective knowledge. As so many people are working on and with Python daily, improvements come thick and fast. Not all Python developers have the chance to attend conferences, or the time to follow the work done by different parts of the community, so it's inevitable that some features of the language and ecosystem are not as well known as they deserve to be.

The objective of this book is to examine parts of the language and Python tooling that may not be known to everyone. If you're an experienced Python developer, you may well know many of these tools, but a good many more may be on your to-do list of things to try when you have time. This is especially true if you're working on established systems, where rearchitecting a component to take advantage of new language features isn't something that can be done frequently.

If you've been using Python for a shorter period, you may be more familiar with recent additions to the language but less aware of some of the libraries available in the wider ecosystem. A large part of the benefit of attending events like Python conferences is the chance to notice minor quality-of-life improvements fellow programmers have made and integrate them into your workflow.

This is not a reference book with stand-alone sections covering different features of Python: the flow from chapter to chapter is dictated by how we would build a real piece of software.

With many pieces of technical documentation, there is a tendency to provide simple examples. Simple examples are great for explaining how something works, but not so useful for understanding when to use it. They can also be tricky to build on, as complex code is often architected quite differently to simple code.

By following this one example, we are able to consider technology choices in context. You will learn what considerations to bear in mind when choosing if a particular approach is suitable. Topics that are related by how they're used will be covered together, rather than topics that are related by how they work.

This book

My objective in writing this book is to share knowledge from different parts of the community and lessons learned over 15 years of writing Python code for a living. It will help you to be productive, both with the core language and add-on libraries. You will learn how to effectively use features of the language that are not strictly essential to be a productive programmer, such as asynchronous programming, packaging, and testing.

However, this book is aimed at people who want to write code, not people who are looking to understand deep magics. I will not delve too far into subjects that involve implementation details of Python. You will not be expected to grok[1] Python C extensions, metaclasses, or algorithms to benefit from this book.

Substantive code samples are shown as numbered listings, and the accompanying code for this book includes electronic versions of these listings. Some of these listings also have output shown directly beneath, rather than separately as a numbered figure.

The accompanying code for this book is also where you'll find copies of the full codebase for the example on a chapter-by-chapter basis, as well as helper code for the exercises. In general, I would recommend that you follow along with the code by checking out the Git repository from the book's website or the code distribution and changing to the relevant branch for the chapter you're reading.

As well as listings, I show some console sessions. When lines which are formatted like code begin with >, that indicates that a shell session is being shown. These sections cover commands to be run from your operating system's terminal. Any that involve >>> are demonstrating a Python console session and should be run from within a Python interpreter.

The example

This book's example is that of a general-purpose data aggregator. If you work in DevOps, then it is very likely you use a program of this sort to track the resource utilization of servers. Alternatively, as a web developer, you may use something like this for statistics aggregation from different deployments of the same system. Some scientists use similar methods, for example, for aggregating the findings of air-quality sensors distributed

[1]A jargon word that became popular during the 1960s, when computing was a much smaller field. To grok something is to understand it on a very deep and intuitive level. It is derived from Robert Heinlein's novel *Stranger in a Strange Land*.

across a city. It isn't something that every developer will need to build, but it is a problem space that is familiar to many developers.

It has been picked not just because it's a common task, but because it allows us to explore many of the subjects we want to cover in a natural, unified way. You will be able to follow the complete example perfectly well using any modern computer running any modern operating system,[2] without purchasing any additional hardware. You may find you get more out of some of the examples if you have additional computers to act as remote data sources.

I will be using a Raspberry Pi Zero equipped with some aftermarket sensors for my examples. This platform is widely available for approximately 5 US dollars and provides lots of interesting data. There are commercial sensor add-ons available from many Raspberry Pi stockists.

Although I'll be recommending things specific to the Raspberry Pi to make following the examples easier, this book is not about the Internet of Things or the Raspberry Pi itself. It's a means to an end; you should feel comfortable to adapt the examples to fit tasks that are more relevant to your interests if you like. Any of the similar problems mentioned earlier would follow the same design process.

Choice of topics

The topics covered by this book have been chosen to shine a light on a variety of different aspects of Python programming. All are underused or under-understood by the Python community as a whole, and none are things likely to be taught as a matter of course to beginners. That's not to say that they are necessarily complex or hard to understand (although some certainly are), but they are techniques that I believe all Python programmers should be familiar with, even if they choose not to use them.

Chapter 1 will introduce you to different ways of approaching the writing of very simple programs in Python and, in particular, will cover Jupyter notebooks and an introduction to the use of the Python debugger. Although both are relatively well-known tools, many people are proficient in the use of one but not both. It will also cover ways

[2]However, if using Windows, I'd suggest you consider something like the Windows Subsystem for Linux, as most add-ons are written with Linux or macOS systems in mind and so may perform better under WSL.

of approaching the writing of command-line interfaces and some useful third-party libraries to support succinct command-line tool development.

Chapter 2 will cover tools that help you identify mistakes in your code, such as automated testing and linting tools. These tools all make it easier to write code that you can be confident in, whether it's a large codebase, one that you rarely need to edit, or one that will garner contributions from third parties. The tools covered here are all ones I would recommend; however, the focus will be on understanding their advantages and disadvantages. You may have used one or more of these tools, and you may have opinions on whether some of them are appropriate to use. This chapter will help you understand the trade-offs to help you make informed decisions.

Chapter 3 covers code packaging and dependency distribution in Python. These are key features for writing applications that can be distributed to others and for designing deployment systems that work reliably. We will use this to convert our stand-alone script into an installable application.

Chapter 4 introduces plugin architectures. This is a powerful feature; it's not uncommon for people who learn them to try and apply them everywhere, which means people can be wary of teaching them. For our example, a plugin architecture is a natural fit. It also covers some advanced techniques for command-line tools that can make debugging plugin-based systems easier.

Chapter 5 covers web interfaces and techniques such as decorators and closures to write complex functions. These techniques are idiomatic in Python but hard to express in many other programming languages. It also covers the appropriate use of abstract base classes. It's common for people to advise against using ABCs because of the tendency of people who learn them to want to use them everywhere. There are definite advantages to a restrained use of ABCs in particular circumstances, especially when combined with some of the tools from Chapter 2.

Chapter 6 expands our example with another major component, the aggregation server that collects the data. This chapter also demonstrates some of the most useful third-party libraries you will use as a Python programmer, such as "requests."

Chapter 7 covers threading and asynchronous programming in Python. Threading is often the source of subtle bugs. Asynchronous code can be used for similar tasks, but it is an idiom that many Python programmers haven't used because the program behaves quite differently to synchronous programming. This chapter focuses on the real-world use of concurrency to achieve a result, rather than demonstrations of a simple example or the limits of what asynchronous programming can do. The objective is working code that is usable in the real world and a thorough understanding of the trade-offs, not a stand-alone technology demonstration.

Chapter 8 goes further with asynchronous programming, adding in the testing of asynchronous code and the various libraries that exist to write code that deals with external tools (such as databases) in an async context. We will also look briefly at some advanced techniques for writing good APIs that are helpful for asynchronous programming, like context managers and context variables.

Chapter 9 sees us return to Jupyter to use its features for data visualization and easy user interaction. We will look at how to use our asynchronous functions with widgets in Jupyter notebooks as well as advanced use of iterators and ways of implementing complex data types.

Chapter 10 details how to make Python code faster, using different types of caching and for which situations they are an appropriate choice. It covers benchmarking individual Python functions in your applications and how to interpret the results to find the reasons for slowdown.

Chapter 11 extends some of the concepts we've visited earlier in the book to handle faults more gracefully. We'll look at ways that our plugin architecture can be modified to allow for handling errors seamlessly while retaining full backward compatibility, and we'll take a closer look at designing processes that handle errors that they encounter.

In **Chapter 12**, the final chapter, we use Python's iterator and coroutine features to enhance the dashboards we've developed with features that aren't passive data gatherers but actively introspect the data we've gathered, allowing us to build multistep analysis flows.

Python version

At the time of writing, the current release of Python is 3.8, and as such the examples in this book are being tested against 3.8 and first development versions of Python 3.9. I do not recommend using older versions. Very few code samples in this book do not work on Python 3.7 or Python 3.6.

You will need Python pip installed to follow along with this book. It should already be installed on your system if you have Python installed. Some operating systems intentionally remove pip from their default installations of Python, in which case you'll need to install it using the operating system's package manager explicitly. This is common on Debian-based systems, where it can be installed with `sudo apt install` `python3-pip`. On other operating systems, use `python -m ensurepip --upgrade` to have Python find the latest version of pip itself or find instructions specific to your operating system.

Electronic versions of code samples and errata are available from the publisher and the book's website at `https://advancedpython.dev`. This should be your first port of call if you encounter any problems working through this book.

CHAPTER 1

Prototyping and environments

In this chapter, we will explore the different ways that you can experiment with what different Python functions do and when is an appropriate time to use those different options. Using one of those methods, we will build some simple functions to extract the first pieces of data that we will be aggregating and see how to build those into a simple command-line tool.

Prototyping in Python

During any Python project, from something that you'll spend a few hours developing to projects that run for years, you'll need to prototype functions. It may be the first thing you do, or it may sneak up on you mid-project, but sooner or later, you'll find yourself in the Python shell trying code out.

There are two broad approaches for how to approach prototyping: either running a piece of code and seeing what the results are or executing statements one at a time and looking at the intermediate results. Generally speaking, executing statements one by one is more productive, but at times it can seem easier to revert to running a block of code if there are chunks you're already confident in.

The Python shell (also called the REPL for **R**ead, **E**val, **P**rint, **L**oop) is most people's first introduction to using Python. Being able to launch an interpreter and type commands live is a powerful way of jumping right into coding. It allows you to run commands and immediately see what their result is, then adjust your input without erasing the value of any variables. Compare that to a compiled language, where the development flow is structured around compiling a file and then running the executable. There is a significantly shorter latency for simple programs in interpreted languages like Python.

© Matthew Wilkes 2020
M. Wilkes, *Advanced Python Development*, https://doi.org/10.1007/978-1-4842-5793-7_1

Prototyping with the REPL

The strength of the REPL is very much in trying out simple code and getting an intuitive understanding of how functions work. It is less suited for cases where there is lots of flow control, as it isn't very forgiving of errors. If you make an error when typing part of a loop body, you'll have to start again, rather than just editing the incorrect line. Modifying a variable with a single line of Python code and seeing the output is a close fit to an optimal use of the REPL for prototyping.

For example, I often find it hard to remember how the built-in function `filter(...)` works. There are a few ways of reminding myself: I could look at the documentation for this function on the Python website or using my code editor/IDE. Alternatively, I could try using it in my code and then check that the values I got out are what I expect, or I could use the REPL to either find a reference to the documentation or just try the function out.

In practice, I generally find myself trying things out. A typical example looks like the following one, where my first attempt has the arguments inverted, the second reminds me that filter returns a custom object rather than a tuple or a list, and the third reminds me that filter includes only elements that match the condition, rather than excluding ones that match the condition.

```
>>> filter(range(10), lambda x: x == 5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'function' object is not iterable
>>> filter(lambda x: x == 5, range(10))
<filter object at 0x033854F0>
>>> tuple(filter(lambda x: x == 5, range(10)))
(5,)
```

Note The built-in function help(...) is invaluable when trying to understand how functions work. As filter has a clear docstring, it may have been even more straightforward to call `help(filter)` and read the information. However, when chaining multiple function calls together, especially when trying to understand existing code, being able to experiment with sample data and see how the interactions play out is very helpful.

If we do try to use the REPL for a task involving more flow control, such as the famous interview coding test question FizzBuzz (Listing 1-1), we can see its unforgiving nature.

Listing 1-1. fizzbuzz.py – a typical implementation

```
for num in range(1, 101):
    val = ''
    if num % 3 == 0:
        val += 'Fizz'
    if num % 5 == 0:
        val += 'Buzz'
    if not val:
        val = str(num)
    print(val)
```

If we were to build this up step by step, we might start by creating a loop that outputs the numbers unchanged:

```
>>> for num in range(1, 101):
...     print(num)
...
1
.
.
.
98
99
100
```

At this point, we will see the numbers 1 to 100 on new lines, so we would start adding logic:

```
>>> for num in range(1, 101):
...     if num % 3 == 0:
...         print('Fizz')
...     else:
...         print(num)
```

```
...
1
.
.
.
98
Fizz
100
```

Every time we do this, we are having to reenter code that we entered before, sometimes with small changes, sometimes verbatim. These lines are not editable once they've been entered, so any typos mean that the whole loop needs to be retyped.

You may decide to prototype the body of the loop rather than the whole loop, to make it easier to follow the action of the conditions. In this example, the values of n from 1 to 14 are correctly generated with a three-way if statement, with `n=15` being the first to be incorrectly rendered. While this is in the middle of a loop body, it is difficult to examine the way the conditions interact.

This is where you'll find the first of the differences between the REPL and a script's interpretation of indenting. The Python interpreter has a stricter interpretation of how indenting should work when in REPL mode than when executing a script, *requiring* you to have a blank line after any unindent that returns you to an indent level of 0.

```
>>> num = 15
>>> if num % 3 == 0:
...     print('Fizz')
... if num % 5 == 0:
  File "<stdin>", line 3
    if num % 5 == 0:
     ^
SyntaxError: invalid syntax
```

In addition, the REPL only allows a blank line when returning to an indent level of 0, whereas in a Python file it is treated as an implicit continuation of the last indent level. Listing 1-2 (which differs from Listing 1-1 only in the addition of blank lines) works when invoked as `python fizzbuzz_blank_lines.py`.

Listing 1-2. fizzbuzz_blank_lines.py

```
for num in range(1, 101):
    val = ''
    if num % 3 == 0:
        val += 'Fizz'
    if num % 5 == 0:
        val += 'Buzz'

    if not val:
        val = str(num)

    print(val)
```

However, typing the contents of Listing 1-2 into a Python interpreter results in the following errors, due to the differences in indent parsing rules:

```
>>> for num in range(1, 101):
...     val = ''
...     if num % 3 == 0:
...         val += 'Fizz'
...     if num % 5 == 0:
...         val += 'Buzz'
...
>>>     if not val:
  File "<stdin>", line 1
    if not val:
    ^
IndentationError: unexpected indent
>>>         val = str(num)
  File "<stdin>", line 1
    val = str(num)
    ^
```

```
IndentationError: unexpected indent
>>>
>>>     print(val)
  File "<stdin>", line 1
    print(val)
    ^
IndentationError: unexpected indent
```

It's easy to make a mistake when using the REPL to prototype a loop or condition when you're used to writing Python in files. The frustration of making a mistake and having to reenter the code is enough to undo the time savings of using this method over a simple script. While it is possible to scroll back to previous lines you entered using the arrow keys, multiline constructs such as loops are not grouped together, making it very difficult to re-run a loop body. The use of the `>>>` and `...` prompts throughout the session also makes it difficult to copy and paste previous lines, either to re-run them or to integrate them into a file.

Prototyping with a Python script

It is very much possible to prototype code by writing a simple Python script and running it until it returns the correct result. Unlike using the REPL, this ensures that it is easy to re-run code if you make a mistake, and code is stored in a file rather than in your terminal's scrollback buffer.[1] Unfortunately, it does mean that it is not possible to interact with the code while it's running, leading to this being nicknamed "printf debugging," after C's function to print a variable.

As the nickname implies, the only practical way to get information from the execution of the script is to use the `print(...)` function to log data to the console window. In our example, it would be common to add a print to the loop body to see what is happening for each iteration:

Tip f-strings are useful for printf debugging, as they let you interpolate variables into a string without additional string formatting operations.

[1] You'll be glad of this the first time you accidentally close the window and lose the code you're working on.

```
for num in range(1,101):
    print(f"n: {num} n%3: {num%3} n%5: {num%5}")
```

The following is the result:

```
n: 1 n%3: 1 n%5: 1
.
.
.
n: 98 n%3: 2 n%5: 3
n: 99 n%3: 0 n%5: 4
n: 100 n%3: 1 n%5: 0
```

This provides an easily understood view at what the script is doing, but it does require some repetition of logic. This repetition makes it easier for errors to be missed, which can cause significant losses of time. The fact that the code is stored permanently is the biggest advantage this has over the REPL, but it provides a poorer user experience for the programmer. Typos and simple errors can become frustrating as there is a necessary context switch from editing the file to running it in the terminal.[2] It can also be more difficult to see the information you need at a glance, depending on how you structure your print statements. Despite these flaws, its simplicity makes it very easy to add debugging statements to an existing system, so this is one of the most commonly used approaches to debugging, especially when trying to get a broad understanding of a problem.

Prototyping with scripts and pdb

pdb, the built-in Python debugger, is the single most useful tool in any Python developer's arsenal. It is the most effective way to debug complex pieces of code and is practically the only way of examining what a Python script is doing inside multistage expressions like list comprehensions.[3]

[2]Some text editors integrate a terminal precisely to cut down on this kind of context switching.

[3]Pdb allows you to step through each iteration of a list comprehension, as you would do with a loop. This is useful when you have existing code that you are trying to diagnose a problem with, but frustrating when the list comprehension is incidental to your debugging.

In many ways, prototyping code is a specialized form of debugging. We know that the code we've written is incomplete and contains errors, but rather than trying to find a single flaw, we're trying to build up complexity in stages. Many of pdb's features to assist in debugging make this easier.

When you start a pdb session, you see a `(Pdb)` prompt that allows you to control the debugger. The most important commands, in my view, are **s**`tep`, **n**`ext`, **b**`reak`, **c**`ontinue`, **p**`retty`**p**`rint`, and **d**`ebug`.[4]

Both `step` and `next` execute the current statement and move to the next one. They differ in what they consider the "next" statement to be. Step moves to the next statement regardless of where it is, so if the current line contains a function call, the next line is the first line of that function. Next does not move execution into that function; it considers the next statement to be the following statement in the current function. If you want to examine what a function call is doing, then step into it. If you trust that the function is doing the right thing, use next to gloss over its implementation and get the result.

`break` and `continue` allow for longer portions of the code to run without direct examination. `break` is used to specify a line number where you want to be returned to the pdb prompt, with an optional condition that is evaluated in that scope, for example, `break 20 x==1`. The `continue` command returns to the normal flow of execution; you won't be returned to a pdb prompt unless you hit another breakpoint.

Tip If you find visual status displays more natural, you may find it hard to keep track of where you are in a debugging session. I would recommend you install the pdb++ debugger which shows a code listing with the current line highlighted. IDEs, such as PyCharm, go one step further by allowing you to set breakpoints in a running program and control stepping directly from your editor window.

Finally, `debug` allows you to specify any arbitrary python expression to step into. This lets you call any function with any data from within a pdb prompt, which can be very useful if you've already used `next` or `continue` to pass a point before you realize that's where the error was. It is invoked as `debug somefunction()` and modifies the `(Pdb)`

[4]These can all be abbreviated, as shown in bold. `step` becomes `s`, `prettyprint` becomes `pp`, etc.

prompt to let you know that you're in a nested pdb session by adding an extra pair of parentheses, making the prompt `((Pdb))`.[5]

Post-mortem debugging

There are two common ways of invoking pdb, either explicitly in the code or directly for so-called "post-mortem debugging." Post-mortem debugging starts a script in pdb and will trigger pdb if an exception is raised. It is run through the use of `python -m pdb yourscript.py` rather than `python yourscript.py`. The script will not start automatically; you'll be shown a pdb prompt to allow you to set breakpoints. To begin execution of the script, you should use the `continue` command. You will be returned to the pdb prompt either when a breakpoint that you set is triggered or when the program terminates. If the program terminates because of an error, it allows you to view the variables that were set at the time the error occurred.

Alternatively, you can use step commands to run the statements in the file one by one; however, for all but the simplest of scripts, it is better to set a breakpoint at the point you want to start debugging and step from there.

The following is the result of running Listing 1-1 in pdb and setting a conditional breakpoint (output abbreviated):

```
> python -m pdb fizzbuzz.py
> c:\fizzbuzz_pdb.py(1)<module>()
-> def fizzbuzz(num):
(Pdb) break 2, num==15
Breakpoint 1 at c:\fizzbuzz.py:2
(Pdb) continue
1
.
.
.
13
14
```

[5]I once so badly misunderstood a bug that I overused `debug` until the pdb prompt looked like `((((((Pdb))))))`. This is an antipattern as it's very easy to accidentally lose your place; try and use conditional breakpoints if you find yourself in a similar situation.